CS411 Notes 13 Subtypes

A. Demers

13 Mar 2001

Here we introduce the notion of a *subtype*. Using the language fragment from Notes 10, we present rules for deriving subtype relations. We show how to modify the typing and evaluation rules to allow a value of a subtype to be used whenever a value of a supertype is expected. Finally, we discuss how the type soundness arguments of the previous Notes could be changed to incorporate the subtype relation.

1 Subtyping

We are interested in cases where one type can be considered a subset or *subtype* of another. We'll define a relation $\prec$ on types such that if

$$\tau \prec \tau'$$

holds then a value of type τ may be used in any context where a value of type τ' is expected.

The motivation that $\prec$ reflects "usable in place of" implies that $\prec$ should be

reflexive:

$$\tau \prec \tau$$

transitive:

$$((\tau_1 \prec \tau_2) \wedge (\tau_2 \prec \tau_3)) \quad \Rightarrow \quad (\tau_1 \prec \tau_3)$$

antisymmetric:

$$((\tau_1 \prec \tau_2) \wedge (\tau_2 \prec \tau_1)) \quad \Rightarrow \quad (\tau_2 = \tau_1)$$

The last of these properties is perhaps slightly questionable; but we choose to say that two types τ_1 and τ_2 that are extensionally indistinguishable – have exactly the same members – are the same type.

These three properties are the definition of a *reflexive partial order*. It is interesting that $\prec$ is a partial order, though we won't make technical use of this fact anytime soon.

As we have already seen, reflexivity and transitivity are easily represented by proof rules. We give them as our first two subtyping rules.

$$\frac{}{\vdash \tau \prec \tau} \qquad \text{(SbT.1)}$$

$$\frac{\vdash \tau_1 \prec \tau_2 \quad \vdash \tau_2 \prec \tau_3}{\vdash \tau_1 \prec \tau_3} \qquad \text{(SbT.2)}$$

These rules apply in general, to all types. Next we develop subtype rules for specific types. Recall our set of types:

$$\begin{aligned} \tau \ ::= \ & \text{int} \mid \text{bool} \mid \text{string} \\ & \mid \mathbf{var}(\tau) \\ & \mid \mathbf{prod}(\ \ldots\ x_i : \tau_i\ \ldots\ ) \\ & \mid \mathbf{fun}(\tau_1)\tau_2 \end{aligned}$$

The first few constructors yield nothing interesting: since there are no meaningful ways to mix integer, boolean and string values, there are no $\prec$ relations among these types.

We shall defer discussion of $\mathbf{var}(\tau)$ until later.

For now, let's discuss product types. First consider the type

$$\tau = \mathbf{prod}(\ \ldots\ x_i : \tau_i\ \ldots\ )$$

Every value of this type is a tuple with named components ("fields"),

$$v \ = \ \langle\ \ldots\ x_i \sim v_i\ \ldots\ \rangle$$

and the only way to use such a value is to select one of its fields. Now consider

$$\tau' = \mathbf{prod}(\ \ldots\ x_i : \tau_i'\ \ldots\ )$$

and a value

$$v'\ =\ \langle\ \ldots\ x_i \sim v_i'\ \ldots\ \rangle$$

Selecting field x_i from a v produces the τ_i value v_i; selecting the identically named field x_i from v'value produces a tau_i' value v_i'. Suppose

$$(\forall i)\ \tau_i \prec \tau_i'$$

then in every case the values selected from v and v' are in types τ and τ' related by $\prec$, and we conclude that the product types τ and τ' are similarly related by $\prec$. Thus we have the rule

$$\frac{\ldots\ \vdash \tau_i \prec \tau_i'\ \ldots}{\vdash \mathbf{prod}(\ldots\ x_i : \tau_i\ \ldots) \ \prec\ \vdash \mathbf{prod}(\ldots\ x_i : \tau_i'\ \ldots)} \qquad \text{(SbT.3)}$$

Now consider a product type like

$$\mathbf{prod}(\ x :\ \text{int},\ y :\ \text{int}\ )$$

and a value

$$v\ =\ \langle\ x \sim 1,\ y \sim 2\ \rangle$$

from that type. As above, the value is a tagged tuple, from which fields may be selected by name. The only way to use such a value is to select one of its fields, and the only fields that may be selected are the ones named in the type, in this case 'x' and 'y'. Now consider the type

$$\mathbf{prod}(\ x :\ \text{int},\ y :\ \text{int},\ z :\ \text{int}\ )$$

and the tuple

$$v'\ =\ \langle\ x \sim 1,\ y \sim 2,\ z \sim 3\ \rangle$$

This tuple has all the fields of the previous tuple, together with an additional field z. It should be clear that every legal selection from v is legal for v' as well

(and in fact yields identical results). Thus we have the counterintuitive behavior that adding an additional field to a product type actually generates a subtype – it makes the type "smaller" under the $\prec$ relation. The rule describing this behavior is

$$\frac{}{\vdash \mathbf{prod}(\ldots\ x_i : \tau_i\ \ldots, y : \tau_y) \ \prec\ \vdash \mathbf{prod}(\ldots\ x_i : \tau_i\ \ldots)} \qquad \text{(SbT.4)}$$

Together with transitivity, these two rules are sufficient. That is, they allow us to prove arbitrary subtyping relations of the form

$$\vdash \mathbf{prod}(\ldots\ x_i : \tau_i\ \ldots,\ \ldots\ y_j : \tau_j^y\ \ldots) \ \prec\ \mathbf{prod}(\ldots\ x_i : \tau_i'\ \ldots)$$
$$\text{where } \tau_i \prec \tau_i'$$

as required.

We can apply a similar analysis to function types. Consider

$$f : \mathbf{fun}(\tau_1)\tau_2 \qquad and \qquad f' : \mathbf{fun}(\tau_1')\tau_2'$$

What conditions will make f usable in place of f' in all contexts? First consider the result type. An invocation of f will produce a result of type τ_2; the invoker (which we assume is correct for f') expects a value of type τ_2'. Thus, the result will be usable if

$$\tau_2 \prec \tau_2'$$

holds. Now consider the parameter types. For f to be usable in place of f', we require that every argument value that might be passed to f' can legally be passed to f. The argument values that might be passed to f' are the values of τ_1', while the values that can legally be passed to f are those of τ_1. Thus, we require

$$\tau_1' \prec \tau_1$$

This antimonotonic behavior in the parameter type is similar to the order-reversal we noted for product types, for a good reason. You can think of record field selection as a function defined on the set of field names. Making the set of field names larger makes the product type smaller in the $\prec$ ordering. Similarly, making the parameter type τ_1 larger makes the function type smaller in the $\prec$ ordering. The resulting rule is

$$\frac{\vdash \tau_1' \prec \tau_1 \qquad \vdash \tau_2 \prec \tau_2'}{\vdash \mathbf{fun}(\tau_1)\tau_2 \ \prec\ mathbffun(\tau_1')\tau_2'} \qquad \text{(SbT.5)}$$

Function types are said to be *covariant* in the result position – increasing the result type increases the function type under $\prec$. They are *contravariant* in the parameter position – increasing the parameter type causes the function type to *decrease* under $\prec$.

Now we return to considering **var** types, which we skipped earlier. Consider the types

$$\mathbf{var}(\tau) \qquad \text{and} \qquad \mathbf{var}(\tau')$$

Are these two types ordered by $\prec$? Equivalently, is the $\mathbf{var}(\tau)$ type constructor covariant or contravariant in τ? We claim it can be neither. Let $\tau \prec \tau'$ be distinct types, so there is some value v' in τ' but not in τ. Let x have type $\mathbf{var}(\tau)$, and let x' have type $\mathbf{var}(\tau')$.

Can x be used in place of x'? No, because of the assignments

$$x' \leftarrow v' \quad \text{(legal)} \qquad \text{and} \qquad x \leftarrow v' \quad \text{(illegal)}$$

Can x' be used in place of x? Again no, because of the reference

$$(x' \uparrow) : \tau'$$

which could produce v', even though the invoker (which expects to be using x rather than x') is only prepared for values in τ.

Thus when τ increases or decreases, the corresponding $\mathbf{var}(\tau)$ types are unrelated. So $\mathbf{var}(\tau)$ is neither covariant nor contravariant in τ; its behavior is sometimes called *non-variant* or *invariant*. Consequently, there is no subtyping rule for $\mathbf{var}(\tau)$.

2 Typing Rules

A simple (and in some ways intuitively appealing) approach to incorporating subtyping into our type checking rules would be to add a single rule

$$\frac{\vdash \tau \prec \tau' \qquad \vdash e : \tau}{\vdash e : \tau'}$$

This rule asserts that an expression e can be typed with any supertype of its "ordinary" type. The problem with this approach is it sacrifices unique typing

– in general it allows an expression to be typed in many ways, sometimes even infinitely many ways. Our evaluation rules rely heavily on unique typing of expressions, so this is not a change that should be made lightly. In fact we shall not make it at all.

We could instead make major changes in our typing rules, in effect taking account of the $\prec$ relation in every one of our rules. For example, for each operator θ there could be a rule like

$$\frac{\begin{array}{ll}\pi \vdash e_1 : \tau_1' & \vdash \tau_1' \prec \tau_1 \\ \pi \vdash e_2 : \tau_2' & \vdash \tau_2' \prec \tau_2\end{array}}{\pi \vdash (e_1 \theta e_2) : \tau_3} \quad \text{where } \theta \text{ is } \tau_1 \times \tau_2 \to \tau_3$$

This approach quickly leads to difficulties, however. Consider the obvious attempt at a subtype-aware typing rule for conditionals:

$$\frac{\begin{array}{l}\ldots\ \pi \vdash e_i : \tau_i\ \ldots \\ \vdash \tau_1 \prec \text{bool} \\ \vdash \tau_2 \prec \tau \\ \tau_3 \prec \tau\end{array}}{\pi \vdash (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) : \tau}$$

While this rule is sound, the result type τ appears nowhere in the original expression. Thus τ may be chosen arbitrarily during rule instantiation, allowing us to prove a conditional's result type to be *any* supertype of the types τ_2 and tau_3 derived for the arms. In short, we have lost the unique typing property.

We could regain unique typing if we could invent some way to identify the "least" supertype of τ_2 and tau_3. However, this is technically a bit challenging, and we won't pursue it yet.

An alternative would be to extend the expression syntax to allow an optional result type annotation on conditionals. Several other constructs – e.g. **let** blocks – would require similar type annotation syntax.

We can avoid ether of these by noting that the language already has type annotations at the single most critical point: the parameter specification of a **lambda** expression:

lambda $x : \tau$ **dot** e

A single additional type rule allows an argument to be any subtype of the corresponding parameter:

$$\frac{\pi \vdash e_1 : \mathbf{fun}(\tau')\tau \qquad \pi \vdash e_2 : \tau_2 \qquad \vdash \tau_2 \prec \tau'}{\pi \vdash e_1(e_2) : \tau} \tag{SbTC.1}$$

Again, the critical point about this rule is to allow an application where the argument type (τ_2) is a subtype of the parameter type (τ').

Given this rule, we can use a sleazy trick to introduce type annotations at essentially arbitrary points in a program. Observe for any type τ' we can write a typed identity function

$$I_{\tau'} \;=\; \mathbf{lambda}\, x : \tau'\, \mathbf{dot}\; x \qquad\qquad I_{\tau'} : \mathbf{fun}(\tau')\tau'$$

Now with no additional rules other than (SbTC.1) above, we have

$$((\pi \vdash e : \tau) \;\wedge\; (\vdash \tau \prec \tau')) \;\Rightarrow\; (\pi \vdash (I_{\tau'})(e) : \tau')$$

That is, $I_{\tau'}$ may be used as a "typecast" to coerce the value of e to any supertype τ'.

We have given up something with this approach: there is no automatic inference of subtypes, everything has to be given explicitly through insertion of typed identity functions. However, this language has few syntactic creature comforts anyway ...so we simply "put up with" the inconvenience of having to specify type annotations in places where they should be "obvious" to a clever compiler, in exchange for the convenience of needing only the single additional type rule (SbTC.1) to exploit the subtyping relation.

3 Evaluation Rules

Amusingly, we can incorporate subtyping with no changes at all in the evaluation rules. Perhaps this should not be too surprising. After all, the subtyping rules were motivated by a notion that every value of a subtype should be usable in a context where the supertype is expected. The notion of "usable" was quite conservative – basically, an oblivious user of the supertype must be able to manipulate values of the subtype with no change in behavior. The requirement of no change in behavior leads naturally to no change in the evalutation rules.

4 Soundness

Little change is required to incorporate subtyping into the type soundness argument outlined in Notes 10. Roughly, every place the theorem used to say something like

e has type τ

we change it to say something more like

e has type τ' where $\tau' \prec \tau$

instead. Through the magic of cut-and-paste, we reproduce the argument here with the few changes required by subtyping:

The type assignment π is

$$\pi = \{\ldots\ x_i : \tau_i\ \ldots\}$$

The environment ϕ is

$$\phi = \{\ldots\ x_i \sim b_i\ \ldots\}$$

where b_i is a bindable (closed) value as in Notes 10.

We say ϕ is *consistent* with π if

$$(\mathbf{dom}(\phi) = \mathbf{dom}(\pi)) \wedge \\ (((x \sim \tau) \in \pi) \wedge ((x \sim b) \in \phi)) \Rightarrow ((\{\} \vdash b : \tau') \wedge (\vdash \tau' \prec \tau))$$

that is, π and ϕ define the same set of names in a type-consistent way – the type of the value bound to x in the environment is always a subtype of the type specified for x in the type assignment.

The store σ is

$$\sigma = \{\ldots\ a_i^\tau \sim v_i\ \ldots\}$$

where v_i is a storable value as in Notes 10.

We say σ is *proper* if

$$((a^\tau \sim v) \in \sigma)\ \Rightarrow\ ((\{\} \vdash v : \tau') \wedge (\tau' \prec \tau))$$

that is, the type of a value in the store is a subtype of the type associated with its typed storage address.

We say σ *supports* expression e if

$$AC(e) \ \subseteq \ \mathbf{dom}(\sigma)$$

where $AC(e)$ is the set of all address constants that occur anywhere in e.

Finally, the soundness theorem becomes

Theorem (soundness of type rules): Suppose e, π, ϕ and σ satisfy

$\pi \vdash e : \tau$
ϕ is consistent with π
σ is proper
σ supports e

and suppose

$$\langle e, \phi, \sigma \rangle \ \rightarrow_1^* \ \langle e', \sigma' \rangle$$

Then (subject reduction):

$\pi \vdash e' : \tau'$ where $\vdash \tau' \prec \tau$
σ' is proper
σ' supports e'

and (progress):

$$(e' \in V) \vee (\exists e'', \sigma'')(\langle e', \phi, \sigma' \rangle \rightarrow_1 \langle e'', \sigma'' \rangle)$$

□