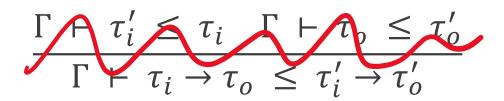
(UN)DECIDABILITY OF BOUNDED QUANTIFICATION

ROSS TATE

SYSTEM F_≤

No Type Recursion

$$\tau ::= \alpha \mid \top \mid A$$



$$\frac{\Gamma \vdash \tau'_{\alpha} \leq \tau_{\alpha} \quad \Gamma, \quad \alpha \leq \tau'_{\alpha} \vdash \tau \leq \tau'}{\Gamma \vdash \forall \alpha \leq \tau_{\alpha}. \tau \leq \forall \alpha \leq \tau'_{\alpha}. \tau'}$$





$$\frac{\Gamma \vdash \Gamma(\alpha) \leq \tau'}{\Gamma \vdash \alpha \leq \tau'}$$

$$\overline{\Gamma \vdash \tau \leq \mathsf{T}}$$

NON-TERMINATING EXAMPLE

- ¬τ = τ → T or ∀α ≤ τ.α (anything contravariant)
- $\kappa(\tau) = \forall \alpha \le \tau. \neg \alpha$
- $\bullet \theta = \forall \alpha. \neg \kappa(\alpha)$

- $\vdash \theta \leq \kappa(\theta)$
- $\vdash \forall \alpha. \neg \kappa(\alpha) \leq \forall \alpha \leq \theta. \neg \alpha$
- $\alpha_1 \leq \theta \vdash \neg \kappa(\alpha_1) \leq \neg \alpha_1$
- $\alpha_1 \leq \theta \vdash \alpha_1 \leq \kappa(\alpha_1)$
- $\alpha_1 \leq \theta \vdash \theta \leq \kappa(\alpha_1)$
- $\alpha_1 \leq \theta \vdash \forall \alpha . \neg \kappa(\alpha) \leq \forall \alpha \leq \alpha_1 . \neg \alpha$
- $\alpha_1 \leq \theta, \alpha_2 \leq \alpha_1 \vdash \neg \kappa(\alpha_2) \leq \neg \alpha_2$
- $\alpha_1 \leq \theta$, $\alpha_2 \leq \alpha_1 \vdash \alpha_2 \leq \kappa(\alpha_2)$

POLARIZING F_{\leq}

No Type Recursion

$$\tau^{+} ::= \top \mid \forall \alpha_{1} \leq \tau_{1}^{-}, \dots, \alpha_{n} \leq \tau_{n}^{-}, \neg \tau^{-}$$

$$\tau^{-} ::= \alpha \mid \forall \alpha_{1}, \dots, \alpha_{n}, \neg \tau^{+}$$

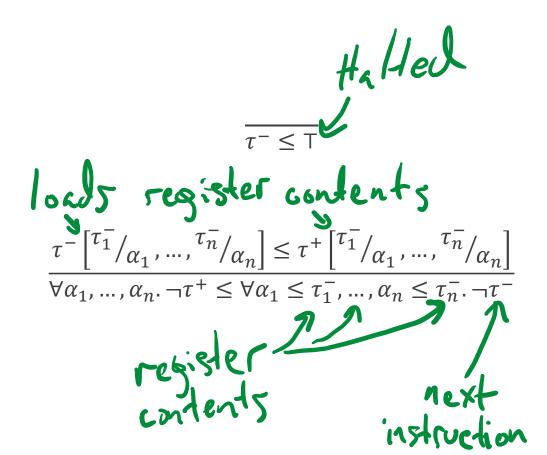
$$\Delta ::= \emptyset \mid \Delta, \alpha \leq \tau^{-}$$

where \neg is any contravariant function on types

$$\Delta \vdash \tau^- \leq \tau^+$$
 is undecidable

- Proof search preserves polarity
- Reflexivity never applies
- au^+ is contravariant with respect to all type variables
- τ^- is covariant with respect to all type variables

F_< AS A REGISTER MACHINE



- n-register machine states: Halted or $\langle \rho; \rho_1, ..., \rho_n \rangle$
 - ρ is next instruction and ρ_1, \dots, ρ_n are register contents
- n-register machine instructions ρ and type-encoding ρ

• $\forall \alpha, \alpha_1, \dots, \alpha_n$. $\neg \top$ Can reference variables $\alpha_1, \dots, \alpha_n$

- $[\alpha_1, ..., \alpha_n] \langle \rho; \rho_1, ..., \rho_n \rangle$ (fetch and update)
 - $\forall \alpha, \alpha_1, \dots, \alpha_n. \neg (\forall \alpha' \leq \alpha, \alpha_1' \leq \llbracket \rho_1 \rrbracket, \dots, \alpha_n' \leq \llbracket \rho_n \rrbracket. \neg \llbracket \rho \rrbracket)$
- $\sigma \leq (\forall \alpha \leq \sigma, \alpha_1 \leq \llbracket \rho_1 \rrbracket, ..., \alpha_n \leq \llbracket \rho_n \rrbracket, \neg \llbracket \rho \rrbracket)$ holds if and only if $\langle \rho; \rho_1, ..., \rho_n \rangle$ eventually halts
 - $\bullet \quad \sigma \triangleq \forall \alpha, \alpha_1, \dots, \alpha_n, \neg(\forall \alpha' \leq \alpha, \alpha_1' \leq \alpha_1, \dots, \alpha_n' \leq \alpha_n, \neg \alpha)$
- Halting problem is undecidable
 - Pierce encodes two-counter machines

IMPLICATIONS FOR WEBASSEMBLY

INHERITANCE AND GENERIC METHODS

```
    abstract class Kappa<in T> {
        virtual void func<A≤T>(A);
    }
    κ(τ) = ∀α ≤ τ. ¬α
    class Theta : Kappa<Theta> {
        override void func<A>(Kappa<A>) {}
    }
    θ = ∀α. ¬κ(α)
```

- Is the inheritance clause valid?
 - Nominally: yes (using decidable type recursion)
 - Structurally: $\theta \le \kappa(\theta)$ loops forever (without type recursion)

WRITING TO JAVA/C#/KOTLIN ARRAYS

```
void fill(Object[] objs) {
	for (int i = 0; i < objs.length; i++)
	objs[i] = i;
}
	Naively requires a cast
	on every assignment

fill(ints :Array<in Integer>) \{...\}
	Array<in Integer> = \exists \alpha \geq \text{Integer}. Array \langle \alpha \rangle
```

Lower-Bounded Existential Subtyping is Undecidable!

```
void fill(Object[] objs) { // hoist cast out of loop \langle \alpha, Array \langle \alpha \rangle alphas\rangle = unpack\_nonnull(objs); if (alphas.length == 0) return; Class\langle \alpha \rangle alpha_class = alphas.elem\_class; if (subtypes(Integer.class, alpha\_class)) // Integer \leq \alpha for (int i = 0; i < alphas.length; i++) alphas[i] = Integer.valueOf(i); else throw new ClassCastException();
```

Type-checks with loop-hoised cast

READING AND WRITING JAVA/C#/KOTLIN ARRAYS

```
void intify(Number[] nums) {
    for (int i = 0; i < nums.length; i++)
        nums[i] = nums[i].intValue();
}</pre>
```

- Exact type of nums (after casts):
 - \exists Integer $\leq \alpha \leq$ Number. $Array(\alpha)$
- Lower-and-upper-bounded variables are dangerous!
 - inconsistent bounds: String $\leq \alpha \leq$ Integer
 - subtyping either not transitive or not decidable
 - Java is unsound due to inconsistent bounds
 - Hard to detect algorithmically in presence of recursion!
 - Checking consistency uses subtyping algorithm
 - Correctness of subtyping algorithm relies on consistency
- Lower-and-upper-bounded variables are necessary
 - See example to the left

F_< IS TOO WEAK

- String[]'s low-level type is $Array \langle String \rangle$
 - not just $\exists \alpha \leq \text{String}. Array(\alpha)$
 - because String is final (has no strict subclasses)
- String[] is a subtype of Object[]
 - so Array(String) needs to be a subtype of $\exists \alpha. Array(\alpha)$
 - but that is not true using F_<'s (existential) rules



$$\frac{\Gamma \vdash \tau_{\alpha} \leq \tau_{\alpha}' \quad \Gamma, \alpha \leq \tau_{\alpha} \vdash \tau \leq \tau'}{\Gamma \vdash \exists \alpha \leq \tau_{\alpha}. \tau \leq \exists \alpha \leq \tau_{\alpha}'. \tau'}$$



$$\frac{\Gamma, \alpha \leq \tau_{\alpha} \vdash \tau \leq \tau'}{\Gamma \vdash \exists \alpha \leq \tau_{\alpha}. \tau \leq \tau'}$$

$$\frac{\Gamma \vdash \tau_{\alpha} \leq \tau_{\alpha}' \quad \Gamma \vdash \tau \leq \tau' \left[\tau_{\alpha} / \alpha \right]}{\Gamma \vdash \tau \leq \exists \alpha \leq \tau_{\alpha}' . \tau'}$$

Non-Deterministic!

DECIDABLE BOUNDED QUANTIFICATION

STRATIFYING SUBTYPING

- abstract class Kappa<in T> {
 virtual void func<A≤T>(A);
 }
- class Theta : Kappa<Theta> {
 override void func<A>(Kappa<A>) {}
 }
- Is the inheritance clause valid?

- For every method of Kappa<Theta>:
 - Is there a corresponding method in Theta?
 - Is that method's signature a subtype?
- Is "void A>(Foo<A>)" a subtype of "void $A\leq T>(A)$ "?
 - Assuming A \leq Theta, is "void (Foo \leq A \geq)" a subtype of "void (A)"?
 - Assuming $A \le Theta$, is A a subtype of FooA>?
 - a nominal subtyping question!
- Structural subtyping always reduces to nominal subtyping
 - Uses stratification to make subtyping decidable

CAUSE OF UNDECIDABILITY

Impredicative Quantification

- Quantifiers can represent quantification terms
 - e.g. α is substitutable with $\forall \alpha$. τ
- Often undecidable
 - even with weak rules
 - Quantification terms can encode register instructions that manipulate terms that can themselves be quantification terms
 - "Code as data" often makes things undecidable

Predicative Quantification

- Quantification is stratified
 - e.g. structural types of classes have quantifiers, but those only quantify over nominal types, and nominal types do not have quantifiers
- Decidable with care
 - either use weak rules
 - or address non-determinism in strong rules
 - Stratification keeps "code" and "data" separate

EXACT TYPES TO THE RESCUE

- $Array(\alpha)$ uniquely determines α
 - because it denotes an exact type
- Consider $Array\langle \tau \rangle <: \exists \alpha \ni \phi(\alpha). Array\langle \alpha \rangle$
 - Holds if and only if α can represent τ
 - Subtyping reduces to checking if the proposition $\phi(\tau)$ holds
 - Decidable if constraint satisfaction is decidable
- Used by iTalX to decide (and infer) assembly-level type-checking for C#
 - with arrays and user-generated low-level casts (no rtt.cast macro-instruction)
 - and with general generics (unpublished)
- Same technique can be used for subtyping of common polymorphic functions
 - and for eliminating type-argument annotations on many polymorphic instructions