(UN)DECIDABILITY OF BOUNDED QUANTIFICATION

ROSS TATE

SYSTEM $F_{\leq}$

No Type Recursion

$$\tau ::= \alpha \mid \top \mid \cancel{\tau \to \tau} \mid \forall \alpha \leq \tau . \tau$$

$$\cancel{\frac{\Gamma \vdash \tau_i' \leq \tau_i \quad \Gamma \vdash \tau_o \leq \tau_o'}{\Gamma \vdash \tau_i \to \tau_o \leq \tau_i' \to \tau_o'}}$$

$$\cancel{\frac{}{\Gamma \vdash \alpha \leq \alpha}}$$

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

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

$$\frac{}{\Gamma \vdash \tau \leq \top}$$

Still

Undecidable!

[Pierce '92]

NON-TERMINATING EXAMPLE

- $\neg\tau = \tau \to \top$ or $\forall\alpha \leq \tau.\, \alpha$ (anything contravariant)
- $\kappa(\tau) = \forall\alpha \leq \tau.\, \neg\alpha$
- $\theta = \forall\alpha.\, \neg\kappa(\alpha)$

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

POLARIZING $\mathrm{F}_{\leq}$

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
- τ^+ is contravariant with respect to all type variables
- τ^- is covariant with respect to all type variables
- $\Delta, \alpha \leq \tau_\alpha^- \vdash \tau^- \leq \tau^+ \Leftrightarrow \Delta \vdash \tau^-\left[{}^{\tau_\alpha^-}/_{\alpha}\right] \leq \tau^+\left[{}^{\tau_\alpha^-}/_{\alpha}\right]$

$F_{\leq}$ AS A REGISTER MACHINE

Halted

$$\overline{\tau^- \leq \top}$$

loads register contents

$$\frac{\tau^-\left[{}^{\tau_1^-}/_{\alpha_1}, \dots, {}^{\tau_n^-}/_{\alpha_n}\right] \leq \tau^+\left[{}^{\tau_1^-}/_{\alpha_1}, \dots, {}^{\tau_n^-}/_{\alpha_n}\right]}{\forall \alpha_1, \dots, \alpha_n. \neg\tau^+ \leq \forall \alpha_1 \leq \tau_1^-, \dots, \alpha_n \leq \tau_n^-. \neg\tau^-}$$

register contents

next instruction

- n-register machine states: Halted or $\langle \rho; \rho_1, \dots, \rho_n \rangle$
  - ρ is next instruction and $\rho_1, \dots, \rho_n$ are register contents
- n-register machine instructions ρ and type-encoding $[\![\rho]\!]$
  - HALT
    - $\forall \alpha, \alpha_1, \dots, \alpha_n. \neg\top$
  - $[\alpha_1, \dots, \alpha_n]\langle \rho; \rho_1, \dots, \rho_n \rangle$ (fetch and update)
    - $\forall \alpha, \alpha_1, \dots, \alpha_n. \neg(\forall \alpha' \leq \alpha, \alpha_1' \leq [\![\rho_1]\!], \dots, \alpha_n' \leq [\![\rho_n]\!]. \neg[\![\rho]\!])$

Can reference variables $\alpha_1, \dots, \alpha_n$

- $\sigma \leq (\forall \alpha \leq \sigma, \alpha_1 \leq [\![\rho_1]\!], \dots, \alpha_n \leq [\![\rho_n]\!]. \neg[\![\rho]\!])$ holds if and only if $\langle \rho; \rho_1, \dots, \rho_n \rangle$ eventually halts
  - $\sigma \triangleq \forall \alpha, \alpha_1, \dots, \alpha_n. \neg(\forall \alpha' \leq \alpha, \alpha_1' \leq \alpha_1, \dots, \alpha_n' \leq \alpha_n. \neg\alpha)$
- 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);
  }
  - $\kappa(\tau) = \forall\alpha \leq \tau. \neg\alpha$
- class Theta : Kappa<Theta> {
  override void func<A>(Kappa<A>) {}
  }
  - $\theta = \forall\alpha. \neg\kappa(\alpha)$
- Is the inheritance clause valid?
  - Nominally: yes (using decidable type recursion)
  - Structurally: $\theta \leq \kappa(\theta)$ 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

$\langle\alpha, Array\langle\alpha\rangle\ \text{alphas}\rangle = unpack\_nonnull(\text{objs});$

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();
}
```

- Exact type of nums (after casts):
  - $\exists \text{Integer} \leq \alpha \leq \text{Number}. Array\langle\alpha\rangle$
- Lower-and-upper-bounded variables are dangerous!
  - inconsistent bounds: $\text{String} \leq \alpha \leq \text{Integer}$
  - 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

$F_{\leq}$ IS TOO WEAK

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

Weak

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

Strong

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

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

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≤T>(A)"?
  - Assuming A ≤ Theta, is "void (Foo<A>)" a subtype of "void (A)"?
  - Assuming A ≤ Theta, is A a subtype of Foo<A>?
    - 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 $\forall\alpha.\,\tau$
- 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

- $Array\langle\alpha\rangle$ 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