TYPE RECURSION AND (UN)DECIDABILITY

Ross Tate

GENERICS WITH VARIANCE

Enumerator<String> <: Enumerator<Object>

- Enumerator is covariant: Enumerator<out T>

Property<Object> <: Property<String>

- Property is contravariant: Property<in T>

Array<Number> <: Array<in Integer out Object>

- Array is invariant, but has contravariant and covariant "projections"
  - You can put Integers *in* to an Array<Number>
  - You can get Objects *out* of an Array<Number>

Map<K, out V> <: Enumerable<Pair<K,V>>

- Inheritance

GENERICS IN C#

```
    interface Tree :
        List<Tree> { }

    class Program {
        bool EquateTrees(Tree x, Tree y) {
            //TODO: Implement
        }
    }
}
```

THEORY OF UNDECIDABILITY

POST-CORRESPONDENCE PROBLEM

Given a non-empty list of pairs of strings: $\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_n, r_n \rangle$

Is there a non-empty sequence of indices $i_1, \ldots, i_k$ such that $\ell_{i_1} \ldots \ell_{i_k} = r_{i_1} \ldots r_{i_k}$?

Undecidable!

Reducible to subtyping of generics with variance

- "On Decidability of Nominal Subtyping with Variance" by Kennedy and Pierce in 2007

UNDECIDABILITY OF GENERICS WITH VARIANCE

class E	C_1<C_2<E>> represents the string $c_2 c_1$
class C_1<X> … (for each letter)	S_sX where $s = c_1 \dots c_n$ short for C_n<…C_1<X>…>
interface I_1<in X> … interface I_n<in X>	Selects an index
interface D<in X>	Used to drive the search
class F<X>	Finishes the search
class R<X> extends F<X>	Enumerates the right options
implements I_1<D<R<S_{r_1}X>>>, …	
class L<X> implements D<F<X>>	Enumerates the left options
implements D<I_1<L<S_{ℓ_1}X>>>, …	

UNDECIDABILITY OF GENERICS WITH VARIANCE

class E

class C_1<X> …

interface I_1<in X> … interface I_n<in X>

interface D<in X>

class F<X>

class R<X> extends F<X>

  implements I_1<D<R<S_{r_1}X>>>, …

class L<X> implements D<F<X>>

  implements D<I_1<L<S_{ℓ_1}X>>>, …

L<$S_{\ell_{i_1}\dots}$E> <: D<R<$S_{r_{i_1}\dots}$E>> holds if and only if

either $\ell_{i_1} \dots = r_{i_1} \dots$

- D<F<$S_{\ell_{i_1}\dots}$E>> <: D<R<$S_{r_{i_1}\dots}$E>> (inheritance)
- R<$S_{r_{i_1}\dots}$E> <: F<$S_{\ell_{i_1}\dots}$E> (contravariance)
- F<$S_{r_{i_1}\dots}$E> <: F<$S_{\ell_{i_1}\dots}$E> (inheritance)
- $S_{r_{i_1}\dots}$E = $S_{\ell_{i_1}\dots}$E (invariance)

or L<$S_{\ell_{i_1}\dots\ell_i}$E> <: D<R<$S_{r_{i_1}\dots r_i}$E>> for some i

- D<I_i<L<$S_{\ell_{i_1}\dots\ell_i}$E>>> <: D<R<$S_{r_{i_1}\dots}$E>> (inheritance)
- R<$S_{r_{i_1}\dots}$E> <: I_i<L<$S_{\ell_{i_1}\dots\ell_i}$E>> (contravariance)
- I_i<D<R<$S_{r_{i_1}\dots r_i}$E>>> <: I_i<L<$S_{\ell_{i_1}\dots\ell_i}$E>> (inheritance)
- L<$S_{\ell_{i_1}\dots\ell_i}$E> <: D<R<$S_{r_{i_1}\dots r_i}$E>> (contravariance)

UNDECIDABILITY OF SINGLE-INSTANTIATION INHERITANCE

$$ZEE\,NL_{a_i}\ldots NL_{a_l}Q_s^{\mathsf{wR}} \blacktriangleleft L_{a_{l+1}}N\ldots L_{a_{j-1}}N\,M^{\mathsf{R}}N\,L_{a_j}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_l}Q_s^{\mathsf{wR}} \blacktriangleleft L_{a_{l+1}}N\ldots L_{a_j}N\,M^{\mathsf{L}}N\,L_{a_{j+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_{j-1}}\,NM^{\mathsf{R}}\,NL_{a_j}\ldots NL_{a_l}Q_s^{\mathsf{wR}} \blacktriangleleft L_{a_{l+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_j}\,NM^{\mathsf{L}}\,NL_{a_{j+1}}\ldots NL_{a_l}Q_s^{\mathsf{wR}} \blacktriangleleft L_{a_{l+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_l}Q_s^{\mathsf{R}} \blacktriangleleft L_{a_{l+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_l} \blacktriangleright Q_s^{\mathsf{wL}}L_{a_{l+1}}N\ldots L_{a_{j-1}}N\,M^{\mathsf{R}}N\,L_{a_j}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_l} \blacktriangleright Q_s^{\mathsf{wL}}L_{a_{l+1}}N\ldots L_{a_j}N\,M^{\mathsf{L}}N\,L_{a_{j+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_{j-1}}\,NM^{\mathsf{R}}\,NL_{a_j}\ldots NL_{a_l} \blacktriangleright Q_s^{\mathsf{wL}}L_{a_{l+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_j}\,NM^{\mathsf{L}}\,NL_{a_{j+1}}\ldots NL_{a_l} \blacktriangleright Q_s^{\mathsf{wL}}L_{a_{l+1}}N\ldots L_{a_k}N\,EEZ$$

$$ZEE\,NL_{a_i}\ldots NL_{a_l} \blacktriangleright Q_s^{\mathsf{L}}L_{a_{l+1}}N\ldots L_{a_k}N\,EEZ$$

“Java Generics are Turing Complete” by Radu Grigore, 2017

UNDECIDABILITY OF EXPANSIVE-RECURSIVE TYPES

Whole-program implementation of interface-method dispatch:

- Every v-table has a table of interface method implementations
- Every interface method is assigned an offset within that table
  - Such that no two methods implemented by the same class have the same offset
- If an object has a certain interface type, that guarantees the offsets corresponding to its methods have the expected function types
- Interface-method dispatch simply loads appropriate offset and calls it

Key point: every interface has a corresponding structural type

- if interface definitions are mutually recursive, so are their structural types

UNDECIDABILITY OF EXPANSIVE-RECURSIVE TYPES

Suppose every interface has a corresponding method

- Inputs of method correspond to the contravariant and invariant type parameters
- Outputs correspond to the covariant and invariant type parameters
- E.g. interface Foo<in I, out O, inv X> { foo : [I, X] → [O, X] }

Suppose every method is given its own index (i.e. no overlapping)

Single-instantiation inheritance ensures inherited methods have unique signature

τ is a nominal subtype of τ'

⇕

Structure of τ is a structural subtype of structure of τ'

UNDECIDABILITY OF EXPANSIVE-RECURSIVE TYPES

(The illustrative example that should have been in the slides to help clarify.)

Hierarchy:

- interface Foo<in X> { foo : [X] → [] }
- interface Bar<out Y> extends Foo<Foo<Y>> { bar : [] → [Y] }
- Interface Baz<Z> extends Bar<Baz<Z>> { baz : [Z] → [Z] }

Structure:

- FOO<x> = (struct (struct (func [x] → []) anyref anyref))
- BAR<y> = (struct (struct (func [FOO<y>] → []) (func [] → [y]) anyref))
- BAZ<z> = (struct (struct (func [FOO<BAZ<z>>] → []) (func [] → [BAZ<z>]) (func [z] → [z]))

PRACTICE OF DECIDABILITY

Restricting Inheritance

EXPANSIVE RECURSION

Kennedy and Pierce
Ensures recursion cycles
Used in CLI

APPLICATION OF EXPANSIVE INHERITANCE

interface Equatable<in T> { bool equals(T that); }

class Integer : Equatable<Integer> { … }

interface List<out E>

: Equatable<List<Equatable<E>>> { … }

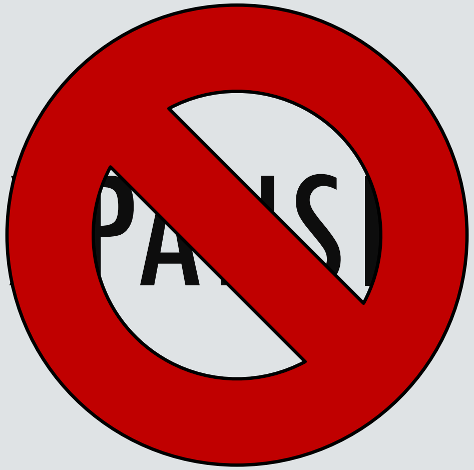
Challenge: make covariant lists equatable if their elements are equatable

Expansive inheritance!

Ceylon's objection: Equatable is meant for constraining types, not values. List<Equatable<…>> is nonsense.

Restricting Inheritance

EXPANSIVE RECURSION



MATERIAL-SHAPE SEPARATION

Materials	Shapes
List, Integer, Comparator	Equatable, Comparable
Variable/Field/Return types	Recursive Inheritance
Type arguments	Recursive Type Constraints

Higher Kinds

Computable Joins

Decidable Subtyping

No class/interface is both a material and a shape

13.5 Million Lines of Java Code ✓

CONDITIONAL METHODS AND SATISFACTION

Shapes can only be used to constrain types

List<Equatable<Integer>> is not a valid type because Equatable<…> is not a type

```
shape Equatable<in T> { bool equals(T that); }
shape Hashable<in T> extends Equatable<T> { int hash(); }
class Integer satisfies Hashable<Number> { … }
class HashSet<E satisfies Equatable<E>> { … }
interface List<out E>
        satisfies Equatable<List<T>> ∀ T <: E where E satisfies Equatable<T> {
    bool equals(List<T> that) ∀ T <: E where E satisfies Equatable<T>;
}
```

"satisfies" constraint ensures structure even though it is not a subtyping constraint

Decidable

Type-safe equality without type recursion

IMPLICATIONS FOR STRUCTURAL TYPES

Key point: every interface has corresponding structural type

- if interface definitions are mutually recursive, so are their structural types

Inheritance is irrelevant, so while restricting inheritance makes nominal subtyping decidable, it has no effect on structural subtyping!

Structural subtyping is undecidable for many real-world programs with decidable nominal types

KEY TAKEAWAYS

Expansive recursion is undecidable

Recursive subtyping definitions are unnecessary

Stratified systems better capture common invariants

Non-subtyping constraints can ensure structure exists