Java Generics Are Turing Complete

Radu Grigore

Abstract

In Java, given a class table and two types t and u, it is undecidable whether t is a subtype of u. This result answers a question posed by Kennedy and Pierce in 2007.

1 Context

Programming languages used nowadays in practice have both subtyping and generics. The precise definition of subtyping and generics varies from one language to another. In $F_{<:}$, which is lambda calculus with subtyping and generics, type checking was shown to be undecidable [5], to some surprise. Compared to $F_{<:}$, languages like Java, C# and Scala have type systems that are significantly more messy. But, several approximations of the real type systems have been studied. The problem is usually posed as follows. Given is a class table and two types t and u. Our task is to decide if $t <: u$. The subtype relation $<:$ is defined in terms of a specific semi-algorithm $\mathcal{N}$. Thus, the question is whether there exists an algorithm equivalent to $\mathcal{N}$: it should give the answer 'yes' whenever $\mathcal{N}$ answers 'yes', and it should give the answer 'no' whenever $\mathcal{N}$ answers 'no' or does not terminate.

Viroli [6] proposed to disallow expansive-recursive class tables, which ensures $\mathcal{N}$ always terminates. The restriction is in some sense as good as possible: for any class table that is expansive-recursive, there exist two types t and u that will cause $\mathcal{N}$ to diverge. Both C# and Scala adopted this restriction. (Strictly speaking, the restriction is omitted from C#'s language specification, but it appears in the Common Language Infrastructure specification, ECMA-335, Section II.9.2.) Java, however, allows expansive-recursive class tables.

Later, Kennedy and Pierce showed that nominal subtyping with variance is undecidable [4]. Their proof does not apply to Java because it makes use of multiple instantiation inheritance. In fact, Kennedy and Pierce conjecture that multiple instantiation inheritance is necessary for undecidability. This conjecture is false.

Gil an Levy found a lower bound for type-checking that does apply to Java [2]: They showed that the Java type-checker can simulate jump-DPDAs, which are known to represent deterministic languages [1]. In fact, we will see that Java's type-checker can simulate Turing machines, and hence it can recognize any recursive language.

2 Result, Definitions, and Key Gadget

The result I wish to present is the following [3]:

Theorem 1 *In Java, it is undecidable whether $t <: u$ according to a given class table.*

Let us see the formal model of Kennedy and Pierce [4] restricted to the case of contravariant types of arity at most 1.

We assume a finite alphbet $x, y, z, \ldots$ of (type) variables, a finite alphabet $C, D, E, \ldots$ of classes of arity 1, and a single class Z of arity 0. *Types* are defined inductively: if x is a variable, then x is a type; if Z is a class of arity 0, then Z is a type; if C is a class of arity 1 and t is a type, then Ct is a type. We denote types by $t, u, v, \ldots$ A *class table* is a set of (inheritance) *rules* of the form $Cx <:: t$, where x is the only variable that may occur in t. (Note that $<:$ and $<::$ are distinct.) We say that a class table is transitively closed when, if it contains rules $Cx <:: Dt$ and $Dy <:: Eu$, then it also contains the rule $Cx <:: E(u[y \mapsto Dt])$. From now on we assume that all class tables are transitively closed. We say that multiple instantiation inheritance is forbidden by a (transitively closed) class table when $Cx <:: Dt$ and $Cx <:: Du$ imply $t = u$.

A class table defines a transition system as follows:

$$\begin{array}{ll} \mathsf{S}(Ct, Du) \to \mathsf{S}(u, v[x \mapsto t]) & \text{if } Cx <:: Dv \\ \mathsf{S}(Ct, Z) \to \mathsf{Y} & \text{if } Cx <:: Z \\ \mathsf{S}(Z, Z) \to \mathsf{Y} & \end{array}$$

Here, S is a special symbol of arity 2, and Y is a special symbol of arity 0. Note that forbidding multiple instantiation inheritance amounts to requiring this transition system to be deterministic. In turn, we can use the transition system to define a subtyping relation:

$$t <: u \qquad \text{iff} \qquad \mathsf{S}(t, u) \to^* \mathsf{Y}$$

Now consider the following class table:

$$Q^{\mathsf{L}}x <:: LNQ^{\mathsf{L}}LNx \qquad Q^{\mathsf{R}}x <:: LNQ^{\mathsf{R}}LNx \tag{1}$$

$$Q^{\mathsf{L}}x <:: EQ^{\mathsf{LR}}Nx \qquad Q^{\mathsf{R}}x <:: EQ^{\mathsf{RL}}Nx \tag{2}$$

$$Ex <:: Q^{\mathsf{LR}}NQ^{\mathsf{R}}EEx \qquad Ex <:: Q^{\mathsf{RL}}NQ^{\mathsf{L}}EEx \tag{3}$$

and let us try to decide if $Q^{\mathsf{R}}EEZ <: LNLNLNEEZ$. Instead of writing $\mathsf{S}(t, u)$, let us write $\mathsf{rev}(t) \blacktriangleleft u$ or $\mathsf{rev}(u) \blacktriangleright t$, where $\mathsf{rev}(t)$ stands for t written backwards.

The (deterministic!) transition system evolves as follows:

$$\begin{array}{lll}
 & ZEEQ^{\mathsf{R}} \blacktriangleleft LNLNLNEEZ & \\
\to & ZEENLQ^{\mathsf{R}} \blacktriangleleft LNLNEEZ & \text{by (1)} \\
\to & ZEENLNLQ^{\mathsf{R}} \blacktriangleleft LNEEZ & \text{by (1)} \\
\to & ZEENLNLNLQ^{\mathsf{R}} \blacktriangleleft EEZ & \text{by (1)} \\
\to & ZEENLNLNLNQ^{\mathsf{RL}} \blacktriangleright EZ & \text{by (2)} \\
\to & ZEENLNLNLN \blacktriangleleft NQ^{\mathsf{L}}EEZ & \text{by (3)} \\
\to & ZEENLNLNL \blacktriangleright Q^{\mathsf{L}}EEZ & \\
\to & ZEENLNL \blacktriangleright Q^{\mathsf{L}}LNEEZ & \text{by (1)} \\
\to & ZEENL \blacktriangleright Q^{\mathsf{L}}LNLNEEZ & \text{by (1)} \\
\to & ZEE \blacktriangleright Q^{\mathsf{L}}LNLNLNEEZ & \text{by (1)}
\end{array}$$

It looks very much like a Turing machine: the $\blacktriangleright$ is the head, which carries with it a state Q, while it moves on a tape on which symbols L are written. The end markers E help the head to turn around. There is some work to do to complete the reduction (how to extend the tape, why a head that moves only back and forth is enough, and so on), but it is routine. One can find an implementation of the reduction from (deterministic) Turing machines to Java subtype checking here: `http://rgrig.appspot.com/javats`.

References

[1] Bruno Courcelle. On jump-deterministic pushdown automata. *Mathematical Systems Theory*, 1977.

[2] Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. In *ECOOP*, 2016.

[3] Radu Grigore. Java generics are Turing complete. In *POPL*, 2017.

[4] Andrew J. Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

[5] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 1994. The result announced in 1991 in a technical report.

[6] Mirko Viroli. On the recursive generation of parametric types. Technical report, University of Bologna, 2000.