Grigore, Radu (2017) *Java Generics are Turing Complete.* **In: POPL 2017 Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL Principles of Programming Languages . pp. 73-85. ACM, New York, USA ISBN 978-1-4503-4660-3.**

Java Generics are Turing Complete

Radu Grigore

University of Kent, United Kingdom

Abstract

This paper describes a reduction from the halting problem of Turing machines to subtype checking in Java. It follows that subtype checking in Java is undecidable, which answers a question posed by Kennedy and Pierce in 2007. It also follows that Java's type checker can recognize any recursive language, which improves a result of Gil and Levy from 2016. The latter point is illustrated by a parser generator for fluent interfaces.

1. Introduction

Is Java type checking decidable? This is an interesting theoretical question, but it is also of interest to compiler developers (Breslav 2013). Since Java's type system is cumbersome for formal reasoning, several approximating type systems have been studied. Two of these type systems are known to be undecidable: (Kennedy and Pierce 2007) and (Wehr and Thiemann 2009). Alas, neither of these is a subset of Java's type system: the reduction of (Kennedy and Pierce 2007) uses multiple instantiation inheritance, which is forbidden in Java; the reduction of (Wehr and Thiemann 2009) uses bounded existential types in a way that goes beyond what Java wildcards can express. So, neither result carries over to Java.

Not knowing a decidability proof was enough to spur interest into decidable fragments. In (Kennedy and Pierce 2007), certain 'recursive–expansive' patterns in the inheritance are forbidden; this solution was adopted in Scala, whose typechecking remains undecidable for other reasons. In (Tate et al. 2011), some restrictions on the use of '? super' are imposed. In (Greenman et al. 2014), one distinguishes between classes and 'shapes'. Here, we shall prove that Java's type system is indeed undecidable, which justifies further these existing studies.

In a separate line of work, (Gil and Levy 2016) show that Java's type system can recognize deterministic context free languages, thus giving the first nontrivial lower bound directly applicable to Java's type system. Our main result immediately implies that Java's type system can recognize any recursive language. In addition, we shall show that Java's type system can recognize context free language in polynomial time. More precisely, the two results are the following:

Theorem 1. *It is undecidable whether $t <: t'$ according to a given class table.*

Theorem 2. *Given is a context free grammar G that describes a language $\mathcal{L} \subseteq \Sigma^*$ over an alphabet Σ of method names. We can construct Java class definitions, a type T, and expressions $Start$, $Stop$ such that the code*

$$T\,\ell \quad = \quad Start.f^{(1)}().f^{(2)}()\ldots f^{(m)}().Stop$$

type checks if and only if $f^{(1)} f^{(2)} \ldots f^{(m)} \in \mathcal{L}$. Moreover, the class definitions have size polynomial in the size of G, and the Java code can be type-checked in time polynomial in the size of G.

Theorem 1 is proved by a reduction from the halting problem of Turing machines to subtype checking in Java (Section 5). The proof is preceded by an informal introduction to Java wildcards (Section 2) and by some formal preliminaries (Sections 3 and 4). It is followed by Theorem 2, which is an application to generating parsers for fluent interfaces (Section 6). The parser generator makes use of a compiler from a simple imperative language into Java types; this compiler is described next (Section 7). Before we conclude, we reflect on the implications of Theorems 1 and 2 (Section 8).

2. Background: Java Wildcards

This section introduces Java generics, wildcards, and their bounds, by example. The presentation is necessarily incomplete. For the definitive reference, see (Gosling et al. 2015).

Java generics are used, for example, to implement lists. When implementing the list, its elements are given a generic type. When using the list, the type of its elements is fixed. The interaction between generics and subtyping is not trivial. List⟨Number⟩ is not a subtype of List⟨Integer⟩: if it were then we could extract integers from a list of numbers; nor is List⟨Integer⟩ is a subtype of List⟨Number⟩: if it were then we could insert numbers in a list of integers.

Intuitively, there is a difference between extracting and inserting data. For example, suppose we want to implement a function `firstNum` that extracts the first number in a list. The implementation should work for both List⟨Integer⟩ and List⟨Number⟩. We could use bounded generics

```
<x extends Number> Number firstNum(List<x> xs)
{ return xs.get(0); }
```

or bounded wildcards

```
Number firstNum(List<? extends Number> xs)
{ return xs.get(0); }
```

Both variants *bound* the elements to be subtypes of Number. Now let us consider the converse situation, in which we want to insert an integer into the list. In this case, Java lets us use only the variant with wildcards:

```
void addOne(List<? super Integer> xs)
{ xs.add(1); }
```

As with `firstNum`, the method `addOne` can be used with both of `List⟨Integer⟩` and `List⟨Number⟩`.

Now let us consider the call to `firstNum`, for a list of integers. For the bounded generics version, the call is

$$\texttt{this.}\langle\texttt{Integer}\rangle\texttt{firstNum(xs)} \quad\text{or}\quad \texttt{firstNum(xs)}$$

The latter is a simplified version, made possible by type inference and syntactic sugar. So, let us focus on the former. In it, we set the type variable x to `Integer`, thus selecting which version of the generic method `firstNum` we are using. On the other hand, for the bounded wildcard version, the call is

$$\texttt{this.firstNum(xs)} \quad\text{or}\quad \texttt{firstNum(xs)}$$

This code looks similar to the one for bounded generics, but it type checks for a different reason: `List⟨Integer⟩` is considered a subtype of `List⟨? extends Number⟩`, because `Integer` is a subtype of `Number`.

Also, if B is a subtype of A, then `List⟨? extends B⟩` is a subtype of `List⟨? extends A⟩`, and `List⟨? super A⟩` is a subtype of `List⟨? super B⟩`. Finally, `List⟨?⟩` is a supertype of all lists.

Bounded wildcards are used liberally in the implementation of Java's standard library. For a more interesting example, consider the following method from java.util.Collections:

```
static <T> int binarySearch(
  List<? extends Comparable<? super T>> list,
  T key) { /* ...*/ }
```

'To search for a key of type `T` in a list, we must have a list whose elements are comparable to `T`.' To express this constraint, we need both '`? extends`' and '`? super`'

Java's mechanism for deciding whether `List⟨α⟩` is a subtype of `List⟨β⟩` is known as *use-site variance* because it involves getting a hint from inside α and β: Do they mention '`? super`' or '`? extends`'? The alternative, *declaration-site variance*, is to take the hint from the declaration of `List`, where the generic type can be annotated as being covariant, invariant, or contravariant. We can simulate declaration site variance by use site variance as follows: if the declaration of L annotates type T to be covariant, then we erase the annotation and replace all uses $L\langle\ldots, T, \ldots\rangle$ by $L\langle\ldots, \texttt{? extends } T, \ldots\rangle$; for contravariant annotations, we proceed similarly, but using '`? super`' instead of '`? extends`'. In what follows, we only need the contravariant case, so it is the only one we formalize.

3. Preliminaries

We formalize a fragment of Java's type system, following (Kennedy and Pierce 2007). *Types* are defined inductively: if x is a type variable, then x is a type; if C is a class of arity $m \geq 0$ and $t_1, \ldots, t_m$ are types, then $Ct_1 \ldots t_m$ is a type. Given a class C, we write $arity(C)$ for its arity; given a type t, we write $vars(t)$ for the type variables occurring in t. A *substitution* σ is a mapping from a finite set $\mathrm{dom}(\sigma)$ of type variables to types; we write $t\sigma$ for the result of applying σ to t. A *class table* is a set of *inheritance rules* of the form

$$Cx_1 \ldots x_m <:: Dt_1 \ldots t_n$$

such that $vars(t_j) \in \{x_1, \ldots, x_m\}$ for $j \in \{1, \ldots, n\}$, where $m = arity(C)$ and $n = arity(D)$. A class table defines a binary relation $<::$ between types: if $t_L <:: t_R$ is an inheritance rule, then $t_L\sigma <:: t_R\sigma$ for all substitutions σ. Note that we slightly abuse the notation $<::$, using it both as a binary relation on types and as a syntactic separator in inheritance rules. When $<::$ is used as a relation, we denote by $<::^*$ its reflexive transitive closure. (Note also that dropping the symbols '`<`,`>`' that Java uses is unambiguous, as long as arities are known.)

Java forbids *multiple instantiation inheritance*: that is, in Java, if $t <::^* Ct_1 \ldots t_m$ and $t <::^* Ct'_1 \ldots t'_m$, then $t_i = t'_i$ for $1 \leq i \leq m$. In particular, if $Cx_1 \ldots x_m <::^* Ct_1 \ldots t_m$, then $x_i = t_i$ for $1 \leq i \leq m$; that is, the class table is *acyclic*.

The subtyping relation $<:$ is defined inductively by the rule

$$\frac{At_1 \ldots t_m <::^* Ct'_1 \ldots t'_n \qquad t''_1 <: t'_1 \quad \ldots \quad t''_n <: t'_n}{At_1 \ldots t_m <: Ct''_1 \ldots t''_n} \quad (1)$$

If $t <: t'$, we say that t is a *subtype* of t', and t' is a *supertype* of t. In rule (1), types t''_* occur in the supertype of the goal but in the subtype of assumptions. In other words, we consider only the *contravariant* case. It is known that Java's wildcards can encode declaration-site contravariance, and that subtyping is decidable in the absence of contravariance (Kennedy and Pierce 2007).

Theorem 1 is proved by a reduction from the halting problem of Turing machines. A *Turing machine* $\mathcal{T}$ is a tuple $(Q, q_I, q_H, \Sigma, \delta)$, where Q is a finite set of *states*, q_I is the *initial state*, q_H is the *halt state*, Σ is a finite *alphabet*, and $\delta : Q \times \Sigma_\perp \to Q \times \Sigma \times \{L, S, R\}$ is a *transition function*. We require $\delta(q_H, a) = (q_H, a, S)$ for all $a \in \Sigma$, and $\delta(q_H, \perp) = (q_H, a, S)$ for some $a \in \Sigma$. A *configuration* is a tuple (q, α, b, γ) of the *current state* $q \in Q$, the *left part of the tape* $\alpha \in \Sigma^*$, the *current symbol* $b \in \Sigma_\perp$, and the *right part of the tape* $\gamma \in \Sigma^*$. The *execution steps* of $\mathcal{T}$ are the following:

$$\begin{aligned}
(q, \alpha a, b, \gamma) &\to (q', \alpha, a, b'\gamma) & \text{for } \delta(q, b) = (q', b', L) \\
(q, \alpha, b, \gamma) &\to (q', \alpha, b', \gamma) & \text{for } \delta(q, b) = (q', b', S) \\
(q, \alpha, b, c\gamma) &\to (q', \alpha b', c, \gamma) & \text{for } \delta(q, b) = (q', b', R)
\end{aligned}$$

We also allow for execution steps that go outside the existing tape:

$$\begin{aligned}
(q, \epsilon, b, \gamma) &\to (q', \epsilon, \perp, b'\gamma) & \text{for } \delta(q, b) = (q', b', L) \\
(q, \alpha, b, \epsilon) &\to (q', \alpha b', \perp, \epsilon) & \text{for } \delta(q, b) = (q', b', R)
\end{aligned}$$

(Here and throughout, ϵ stands for the empty string.) A *run on input tape* α_I is a sequence of execution steps starting from configuration $(q_I, \epsilon, \perp, \alpha_I)$. If $\mathcal{T}$ reaches q_H we say that $\mathcal{T}$ *halts* on α_I.

Theorem 3 (Turing 1936). *It is undecidable whether a Turing machine $\mathcal{T}$ halts on input α_I.*

4. Subtyping Machines

The fragment of Java's type system defined in the previous section does not seem to share much with Turing machines. To clarify the connection, let us define a third formalism: subtyping machines. The plan is to see how subtyping machines correspond to a fragment of a fragment of Java's type system, and at the same time can simulate Turing machines. So far, we saw a fragment of Java's type system that included generic classes of arbitrary arity. Subtyping machines can only handle the case in which all classes have arity 1, apart from one distinguished class Z which has arity 0. The configuration of a subtyping machine is a subtyping query

$$C_1 C_2 \ldots C_m Z <: D_1 D_2 \ldots D_n Z$$

For reasons that will become clear later (Example 5), we introduce two alternative notations for the *same* configuration as above:

$$\begin{aligned}
& ZC_m C_{m-1} \ldots C_1 \blacktriangleleft D_1 D_2 \ldots D_n Z \\
= \quad & ZD_n D_{n-1} \ldots D_1 \blacktriangleright C_1 C_2 \ldots C_m Z
\end{aligned}$$

Since we restrict our attention to arity ≤ 1, there are two ways in which proof rule (1) can be applied:

$$\frac{C_1 \ldots C_m Z <::^* Z}{C_1 \ldots C_m Z <: Z}$$

$$\frac{C_1 \ldots C_m Z <::^* D_1 E_2 \ldots E_p Z \qquad D_2 \ldots D_n Z <: E_2 \ldots E_p Z}{C_1 \ldots C_m Z <: D_1 \ldots D_n Z}$$

Correspondingly, we define two types of *execution steps* $\cdot \rightsquigarrow \cdot$ for the subtyping machine:

$$(ZC_m \ldots C_1 \blacktriangleleft Z) \;\rightsquigarrow\; \bullet$$

if $C_1 \ldots C_m Z <::^* Z$, and

$$(ZC_m \ldots C_1 \blacktriangleleft D_1 \ldots D_n Z) \;\rightsquigarrow\; (ZE_p \ldots E_2 \blacktriangleright D_2 \ldots D_n Z)$$

if $C_1 \ldots C_m Z <::^* D_1 E_2 \ldots E_p Z$. The special configuration $\bullet$ is called the *halting* configuration.

Recall that $<::^*$ is the reflexive transitive closure of a relation defined by the class table. Thus, in particular, if the class table contains the inheritance rule

$$C_1 x <:: D_1 E_2 E_3 \ldots E_p x$$

then we can instantiate it with $x := C_2 \ldots C_m Z$ to enable the following execution step:

$$
\begin{aligned}
& Z C_m \ldots C_2 C_1 \blacktriangleleft D_1 D_2 \ldots D_n Z \\
\rightsquigarrow\; & Z C_m \ldots C_2 E_p \ldots E_2 \blacktriangleright D_2 \ldots D_n Z
\end{aligned}
\tag{2}
$$

Also, because $<::^*$ is reflexive, the following execution step is enabled:

$$
\begin{aligned}
& Z C_m \ldots C_1 N \blacktriangleleft N D_1 \ldots D_n Z \\
\rightsquigarrow\; & Z C_m \ldots C_1 \blacktriangleright D_1 \ldots D_n Z
\end{aligned}
\tag{3}
$$

The runs of subtyping machines correspond to (partial) proofs. Runs that halt correspond to completed proofs. Runs that get stuck correspond to failed proofs. The subtyping machine may be nondeterministic, which corresponds to situations in which one may apply proof rule (1) in several ways; but, if multiple instantiation inheritance is forbidden, as it is in Java, then the subtyping machine is deterministic. There is nothing deep about subtyping machines: they are introduced simply because the new notation will make it much easier to notice certain patterns.

Proposition 4. *Consider a subtyping machine described by a given class table. We have*

$$C_1 \ldots C_m Z <: D_1 \ldots D_n Z$$

if and only if there exists a halting run

$$(ZC_m \ldots C_1 \blacktriangleleft D_1 \ldots D_n Z) \;\rightsquigarrow^*\; \bullet$$

Example 5. In preparation for the main reduction, let us look at one particular subtyping machine. Consider the following class table:

$$Q^{\mathsf{L}} x <:: LNQ^{\mathsf{L}} LNx \qquad Q^{\mathsf{R}} x <:: LNQ^{\mathsf{R}} LNx \tag{4}$$

$$Q^{\mathsf{L}} x <:: EQ^{\mathsf{LR}} Nx \qquad Q^{\mathsf{R}} x <:: EQ^{\mathsf{RL}} Nx \tag{5}$$

$$Ex <:: Q^{\mathsf{LR}} NQ^{\mathsf{R}} EEx \qquad Ex <:: Q^{\mathsf{RL}} NQ^{\mathsf{L}} EEx \tag{6}$$

and the query $Q^{\mathsf{R}} EEZ <: LNLNLNEEZ$. Then, the subtyping machine runs as follows:

$$
\begin{array}{lll}
& ZEEQ^{\mathsf{R}} \blacktriangleleft LNLNLNEEZ & \\
\rightsquigarrow^2 & ZEENLQ^{\mathsf{R}} \blacktriangleleft LNLNEEZ & \text{by (4)+(2), then (3)} \\
\rightsquigarrow^2 & ZEENLNLQ^{\mathsf{R}} \blacktriangleleft LNEEZ & \text{by (4)+(2), then (3)} \\
\rightsquigarrow^2 & ZEENLNLNLQ^{\mathsf{R}} \blacktriangleleft EEZ & \text{by (4)+(2), then (3)} \\
\rightsquigarrow & ZEENLNLNLNQ^{\mathsf{RL}} \blacktriangleright EZ & \text{by (5)+(2)} \\
\rightsquigarrow & ZEENLNLNLN \blacktriangleleft NQ^{\mathsf{L}} EEZ & \text{by (6)+(2)} \\
\rightsquigarrow & ZEENLNLNL \blacktriangleright Q^{\mathsf{L}} EEZ & \text{by (3)} \\
\rightsquigarrow^2 & ZEENLNL \blacktriangleright Q^{\mathsf{L}} LNEEZ & \text{by (4)+(2), then (3)} \\
\rightsquigarrow^2 & ZEENL \blacktriangleright Q^{\mathsf{L}} LNLNEEZ & \text{by (4)+(2), then (3)} \\
\rightsquigarrow^2 & ZEE \blacktriangleright Q^{\mathsf{L}} LNLNLNEEZ & \text{by (4)+(2), then (3)}
\end{array}
$$

```
interface Z {}  interface N<x> {}  interface L<x> {}
interface Qlr<x> {}  interface Qrl<x> {}
interface E<x> extends
  Qlr<N<?super Qr<?super E<?super E<?super x>>>>,
  Qrl<N<?super Ql<?super E<?super E<?super x>>>>> {}
interface Ql<x> extends
  L<N<?super Ql<?super L<?super N<?super x>>>>,
  E<Qlr<?super N<?super x>>> {}
interface Qr<x> extends
  L<N<?super Qr<?super L<?super N<?super x>>>>,
  E<Qrl<?super N<?super x>>> {}
class Main {
    L<?super N<?super L<?super N<?super L<?super N<?super
    E<?super E<?super Z>>>>>>>
  doit(Qr<? super E<? super E<? super Z>>> v) {return v;}
}
```

Figure 1. Java code for Example 5

Some lines group together two execution steps so that the overall pattern is clearer: we have a head traveling back-and-forth over a tape! The Java code corresponding to (4), (5), (6) appears in Figure 1. □

Well-formedness. From (Kennedy and Pierce 2007), we know that the subtyping relation is transitive if and only if certain well-formedness conditions hold. Instead of stating these conditions in full generality, let us do it only for our special case, in which arities are ≤ 1 and everything is contravariant. Under these conditions, *well-formedness* requires that, if an inheritance rule has the form $Ax <:: D_1 \ldots D_n x$, then n must be odd.

5. Main Result

Given a Turing machine $\mathcal{T}$ and an input tape α_{I}, we will construct types t_1, t_2 and a class table such that $t_1 <: t_2$ if and only if $\mathcal{T}$ halts on α_{I}. For each state $q_s \in Q$, we have six classes: Q_s^{wL}, Q_s^{wR}, Q_s^{L}, Q_s^{R}, Q_s^{LR} and Q_s^{RL}; for each letter $a \in \Sigma \cup \{\#\}$, we have a class L_a. Here, $\#$ is a fresh letter. We also make use of four auxiliary classes: N, E, M^{L}, and M^{R}.

Remark 6. Informally, these classes will be used as follows. A class Q_s^* indicates that we are simulating the Turing state s. A class $Q_*^{\mathsf{w}*}$ indicates that we are waiting: the head of the subtyping machine is not in the same position as the head of the simulated Turing machine; conversely, if w is missing, then the head of the subtyping machine is in the same position as the head of the simulated Turing machine. The superscripts L/R indicate that the head of the subtyping machine is moving **l**eft/**r**ight. The class Q_*^{LR} indicates that the subtyping machine used to move towards **l**eft but is now in the process of turning **r**ight. The classes M^{L} and M^{R} mark the position of the Turing machine head, which can be on the **l**eft of the marker or on the **r**ight of the marker. The special letter $\#$ marks the two endpoints of the tape, and is used for extending the tape of the subtyping machine, if more space is needed. The class E is an end of tape marker, on the outside of $\#$, which helps the subtyping machine head turn around. Roughly, the class N is a trick that lets us have covariance if we want it, without putting it in the formalism. □

The subtyping machine will have configurations of one of the forms in Figure 2. These configurations obey several simple invariants. If we read off only the symbols of the form L_*, we get

$$L_{a_i} L_{a_{i+1}} \ldots L_{a_{k-1}} L_{a_k}$$

This will represent the tape of a Turing machine, with content $a_{i+1} \ldots a_{k-1}$. We use L_{a_i} and L_{a_k} as markers of the two endpoints; in fact we impose the invariant that $i < k$ and $a_i = a_k = \#$.

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

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

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

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

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

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

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

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

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

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

Figure 2. The form of configurations of the subtyping machine being constructed in Section 5. All these subtyping machine configurations simulate the same Turing machine configuration; namely, $(q_s, a_{i+1}\ldots a_{j-1}, a_j, a_{j+1}\ldots a_{k-1})$. By convention, $a_i = a_k = \#$. The head of the subtyping machine is between l and $l+1$; in all cases, $i-1\le l\le k$. The head of the simulated Turing machine is at position j.

If, in addition to L_*, we also read off $M^{\mathsf{L}}, M^{\mathsf{R}}, Q_s^{\mathsf{L}}$ and Q_s^{R}, then we get one of

$$L_{a_i}\ldots L_{a_{j-1}}L_{a_j}M^{\mathsf{L}}L_{a_{j+1}}\ldots L_{a_k}$$
$$L_{a_i}\ldots L_{a_{j-1}}L_{a_j}Q_s^{\mathsf{L}}L_{a_{j+1}}\ldots L_{a_k}$$
$$L_{a_i}\ldots L_{a_{j-1}}M^{\mathsf{R}}L_{a_j}L_{a_{j+1}}\ldots L_{a_k}$$
$$L_{a_i}\ldots L_{a_{j-1}}Q_s^{\mathsf{R}}L_{a_j}L_{a_{j+1}}\ldots L_{a_k}$$

This is how we mark the current symbol a_j of the simulated Turing machine: M^{L} and Q_s^{L} indicate that the head is on the letter to the left; M^{R} and Q_s^{R} indicate that the head is on the letter to the right. The markers M^{L} and M^{R} always have at least one letter on both their sides; the marker Q_s^{L} must have a letter on its left (but not necessarily on it right); the marker Q_s^{R} must have a letter on its right (but not necessarily on it left). These conditions account for the constraints on i, j, k, l in Figure 2.

If we read off only the symbols of the form $Q_s^{\mathsf{L}}, Q_s^{\mathsf{R}}, Q_s^{\mathsf{wL}}$ and Q_s^{wR}, then we see that each configuration in Figure 2 contains exactly one such symbol: It tells us that the Turing machine we simulate is in state q_s.

Putting all this together, we see that all subtyping machine configurations in Figure 2 simulate the same configuration of the Turing machine, namely $(q_s, a_{i+1}\ldots a_{j-1}, a_j, a_{j+1}\ldots a_{k-1})$.

Now let us move to the class table, which describes how the subtyping machine runs. The inheritance rules are given in Figure 3. We want to start the subtyping machine from the configuration

$$ZEE\,NL_{\#}\,NM^{\mathsf{L}}\,NL_{a_1}\ldots NL_{a_m}\,NL_{\#}\,Q_{\mathsf{I}}^{\mathsf{wR}}\blacktriangleleft EEZ$$

where $a_1\ldots a_m$ is the content of the initial tape α_{I}. Equivalently, we can say that we want to ask the subtyping query

$$Q_{\mathsf{I}}^{\mathsf{wR}}\,L_{\#}N\,L_{a_m}N\ldots L_{a_1}N\,M^{\mathsf{L}}N\,L_{\#}N\,EEZ <: EEZ$$

It is trivial to check that the inheritance rules in Figure 3 are well-formed. Now let us check whether they use multiple instantiation inheritance. For this, we build an inheritance graph, which has an arc $A\to B$ for each inheritance rule of the form $Ax <:: Bt$.
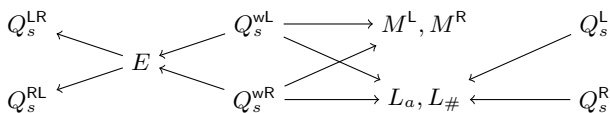
Recall that the Turing machine we simulate is deterministic; in particular, for each (q_s, a) there is a single arc $Q_s^{\mathsf{L}}\to L_a$. We see that all walks (which are possibly self-intersecting paths) are uniquely identified by their endpoints. Thus, multiple instantiations are not possible.

The class table is large, but it is based on a few simple ideas. In the middle of the tape, all evolutions follow the following pattern, which is a simple generalization of what we saw in Example 5:

$$\alpha Q\blacktriangleleft SN\beta$$
$$\rightsquigarrow \quad \alpha NS_1NS_2Q'N\blacktriangleright N\beta \qquad \text{by (2) and a rule (7)}$$
$$\rightsquigarrow \quad \alpha NS_1NS_2Q'\blacktriangleleft\beta \qquad \text{by (3)}$$

Here, (i) α and β are some strings of symbols, (ii) S, S_1, S_2 are letters L_* or markers M^*, and (iii) the class table contains an inheritance rule of the form

$$Qx <:: SN\,Q'\,S_2N\,S_1N\,x \tag{7}$$

If we ignore the padding with N temporarily, we see that the effect is the following rewriting:

$$Q\blacktriangleleft S \quad\implies\quad S_1S_2Q'\blacktriangleleft$$

The 'state' changes from Q to Q', and the 'symbol' S is replaced with S_1S_2. Also, the 'head' $\blacktriangleleft$ passes over the symbol S that it processed. In fact, the head of the subtyping machine will keep moving back-and-forth, just as in Example 5.

The reversal of the 'head' at the end of the 'tape' is performed exactly as in Example 5. During reversal, configurations contain Q_s^{RL} or Q_s^{LR}, thus stepping outside of the set listed in Figure 2, but only temporarily.

At this point we have a head that traverses the tape back-and-forth, carries with it a state, and can replace a symbol on the tape by several symbols. It is now a simple exercise to figure out how to use markers M^{L} and M^{R} to track the head of the Turing machine we simulate. It is also a simple exercise to figure out how to use the markers $L_{\#}$ at the endpoints of the tape in order to handle the case in which the tape needs to be extended. Finally, it is an easy exercise to figure out how to make the subtyping machine halt (soon) after it reaches a configuration that simulates the halting state q_{H} of the Turing machine. In fact, all these exercises are solved in Figure 3.

We have proved the main result:

Theorem 1. *It is undecidable whether $t <: t'$ according to a given class table.*

$$Q_s^{\mathsf{wL}}x <:: M^\mathsf{L}NQ_s^\mathsf{L}x \qquad\qquad \text{for } q_s \in Q$$

$$Q_s^{\mathsf{wL}}x <:: M^\mathsf{R}NQ_s^{\mathsf{wL}}M^\mathsf{R}Nx \qquad \text{for } q_s \in Q$$

$$Q_s^{\mathsf{wL}}x <:: L_aNQ_s^{\mathsf{wL}}L_aNx \qquad \text{for } q_s \in Q \text{ and } a \in \Sigma \cup \{\#\}$$

$$Q_s^{\mathsf{wL}}x <:: EQ_s^{\mathsf{LR}}Nx \qquad\qquad \text{for } q_s \in Q \setminus \{q_\mathsf{H}\}$$

$$Q_\mathsf{H}^{\mathsf{wL}}x <:: EEZ$$

$$Ex <:: Q_s^{\mathsf{LR}}NQ_s^{\mathsf{wR}}EEx \qquad \text{for } q_s \in Q$$

$$Q_s^\mathsf{L}x <:: L_aNQ_{s'}^{\mathsf{wL}}M^\mathsf{L}NL_bNx \qquad \text{for } \delta(q_s, a) = (q_{s'}, b, \mathsf{L})$$

$$Q_s^\mathsf{L}x <:: L_aNQ_{s'}^{\mathsf{wL}}M^\mathsf{R}NL_bNx \qquad \text{for } \delta(q_s, a) = (q_{s'}, b, \mathsf{S})$$

$$Q_s^\mathsf{L}x <:: L_aNQ_{s'}^{\mathsf{wL}}L_bNM^\mathsf{R}Nx \qquad \text{for } \delta(q_s, a) = (q_{s'}, b, \mathsf{R})$$

$$Q_s^\mathsf{L}x <:: L_\#NQ_{s'}^{\mathsf{wL}}L_\#NM^\mathsf{L}NL_bNx \qquad \text{for } \delta(q_s, \bot) = (q_{s'}, b, \mathsf{L})$$

$$Q_s^\mathsf{L}x <:: L_\#NQ_{s'}^{\mathsf{wL}}L_\#NM^\mathsf{R}NL_bNx \qquad \text{for } \delta(q_s, \bot) = (q_{s'}, b, \mathsf{S})$$

$$Q_s^\mathsf{L}x <:: L_\#NQ_{s'}^{\mathsf{wL}}L_\#NL_bNM^\mathsf{R}Nx \qquad \text{for } \delta(q_s, \bot) = (q_{s'}, b, \mathsf{R})$$

Figure 3. Class table used to simulate a Turing machine $\mathcal{T} = (Q, q_\mathsf{I}, q_\mathsf{H}, \Sigma, \delta)$. There are twelve more inheritance rules, obtained by swapping $\mathsf{L} \leftrightarrow \mathsf{R}$ in the rules above.

The construction described in the previous proof is implemented (Grigore 2016), so that the reader can test the proof by trying it on example Turing machines. The examples include counting in binary, computing the Ackermann function, and checking whether ZF is consistent.

6. Application: Fluent Interfaces

The reduction from Turing machines (Section 5) lets us conclude that typechecking is undecidable. But, it is not immediately clear whether we can harness the reduction to do something useful. In this section, we see that the reduction can be used to implement a parser generator for fluent interfaces. The parser generator is efficient in theory (Theorem 2), but not in practice. Before Theorem 2, we had no reason to believe that a practically efficient parser generator for fluent interfaces is possible; after Theorem 2, we have no reason to believe that it is impossible.

Consider a Java library. We say that it has a *fluent interface* when it encourages its users to chain method calls: $f().g().h()$. Programs written in the fluent style are more readable than programs written in the traditional, sequential style: $f(); g(); h()$. Why? In the fluent style, the order in which methods are called is constrained by the type checker. If the library is well designed, its types will constrain the chains of method calls such that they are easy to read and understand. In the traditional style, we cannot use the type checker to enforce readability.

(Gil and Levy 2016) propose a principled method for designing fluent interfaces. We start by describing the allowed chains of method calls using a context free grammar: method names are tokens, and allowed chains of method calls correspond to words in a context free language. Their main result is the following:

Theorem 7 (Gil and Levy 2016). *Given is a deterministic context free grammar G that describes a language $\mathcal{L} \subseteq \Sigma^*$ over an alphabet Σ of method names. We can construct Java class definitions, a type T, and expressions Start, Stop such that the code*

$$T \; \ell \;\; = \;\; Start.f^{(1)}().f^{(2)}()\ldots f^{(m)}().Stop$$

type checks if and only if $f^{(1)}f^{(2)}\ldots f^{(m)} \in \mathcal{L}$.

Their construction relies on a conversion from deterministic pushdown automata to realtime jump-deterministic pushdown automata (Courcelle 1977). Instead, we can use Theorem 1 to show a similar but stronger result: (a) the grammar will not be required to be deterministic, (b) the generated code will be guaranteed to be polynomial in the size of the grammar, and (c) the generated code can be type checked in polynomial time. Point (a) means that instead of a parser generator for $\mathrm{LR}(k)$ grammars, we give a parser generator for all context free grammars. But, the key improvements are points (b) and (c), which mean that the parser is (theoretically) efficient.

Theorem 2. *Given is a context free grammar G that describes a language $\mathcal{L} \subseteq \Sigma^*$ over an alphabet Σ of method names. We can construct Java class definitions, a type T, and expressions Start, Stop such that the code*

$$T \; \ell \;\; = \;\; Start.f^{(1)}().f^{(2)}()\ldots f^{(m)}().Stop$$

type checks if and only if $f^{(1)}f^{(2)}\ldots f^{(m)} \in \mathcal{L}$. Moreover, the class definitions have size polynomial in the size of G, and the Java code can be type-checked in time polynomial in the size of G.

Proof outline. From the grammar G, we generate a CYK parser, implemented in a simple imperative language. Then, we compile the parser into a Turing machine. Then, we compile the Turing machine into Java class definitions, as in Figure 3. Finally, we construct the expression

$$Start.f^{(1)}()\ldots f^{(m)}().Stop \qquad (8)$$

so that it has a type S and so that checking whether $S <: T$ is equivalent to running the parser on the word $f^{(1)}\ldots f^{(m)}$. $\qquad\square$

Once we know Theorem 1, it is not difficult to see that the transformations mentioned above are possible, even if working out all details is tedious. In what follows, we unravel enough details to clarify that the construction is polynomial. This description will also give a high level view of the accompanying proof of concept implementation (Grigore 2016).

6.1 Builders

We start with the last part of the proof: how to choose the expressions *Start* and *Stop*, which occur in (8). We want to initialize the subtyping machine in the following configuration:

$$\overbrace{\underbrace{ZEENL_\#NM^\mathsf{L}}_{Start}\underbrace{NL_{f(1)}}_{f^{(1)}()}\ldots\underbrace{NL_{f(m)}}_{f^{(m)}()}\underbrace{NL_\#Q_\mathsf{I}^{\mathsf{wR}}}_{Stop}}^{S} \blacktriangleleft \overbrace{EEZ}^{T}$$

Over braces, we have shorthand names for the types:

$$S := Q_\mathsf{I}^{\mathsf{wR}}L_\#NL_{f(m)}N\ldots L_{f(1)}NM^\mathsf{L}NL_\#NT$$

$$T := EEZ$$

Under braces, we see how various parts of the expression (8) build the type S. More precisely, we make use of an abstract class B that is implemented as in Figure 4. Given such a class B, we can easily build the required type S; for example,

```
E<?super E<?super Z>> l =
    B.start.a().a().b().c().stop();
```

would check if the word $aabc$ is in the language being considered.

```
abstract class B<x> {
  static B<ML<?super N<?super
          Lhash<?super N<?super
          E<?super E<?super Z>>>>>> start;
  abstract QWRstart<?super
          Lhash<?super N<?super x>>> stop();
  abstract B<La<?super N<?super x>>> a();
  abstract B<Lb<?super N<?super x>>> b();
  abstract B<Lc<?super N<?super x>>> c();
}
```

Figure 4. Builder class, for obtaining a Turing tape out of a chain of method calls. In this example, the alphabet is $\Sigma = \{a, b, c\}$. In the general case, there is one method for each letter.

6.2 Background: Grammars and Parsers

We continue with the first part of the proof: how to go from a context free grammar to a parser. The content of this section is standard, but it is included for completeness and ease of reference.

Given a finite set Σ, we aim to specify subsets of Σ^*. We call Σ an *alphabet*, we call its elements *letters* or *terminal symbols* or *terminals*, and we call subsets of Σ^* *languages*. To define languages, we introduce another finite set Γ, whose elements we call *nonterminal symbols* or *nonterminals*. We denote terminals by lowercase Latin letters (a, b, c, ...), and nonterminals by uppercase Latin letters (A, B, C, ...). We denote strings of symbols, from $\Sigma \cup \Gamma$, by lowercase Greek letters (α, β, γ, ...), with ϵ reserved for the empty string. We write $|\alpha|$ for the length of α. We denote symbols (terminal or nonterminal) by $\theta_1, \ldots, \theta_n$. A *production* is a relation $A \to \theta_1 \ldots \theta_n$ for $n \geq 0$, where A is a nonterminal and $\theta_1 \ldots \theta_n$ is a string of symbols. A (context free) *grammar* G is a set of productions together with a distinguished *start nonterminal S*.

Given a grammar, we can use its productions to inductively define *parse trees*: (a) if a is a nonterminal, then a is a parse tree of a; and (b) if $A \to \theta_1 \ldots \theta_n$ is a production, and T_k is a parse tree of the symbol θ_k for all $k \in \{1, \ldots, n\}$, then $(A, [T_1, \ldots, T_n])$ is a parse tree of A. If the leaves of a parse tree T read from left to write give $\alpha \in \Sigma^*$, we say that *T is a parse tree of α*. The *language $\mathcal{L}$* defined by G is the set of strings $\alpha \in \Sigma^*$ that have some parse tree among the parse trees of the start nonterminal S.

Given string α and a grammar G the membership problem asks whether $\alpha \in \mathcal{L}$, where $\mathcal{L}$ is the language defined by G. The membership problem can be solved by a straightforward recursion, known as the CYK algorithm. We define a relation cyk inductively: (a) if a is a nonterminal, then $cyk(a, a)$ holds; and (b) if $A \to \theta_1 \ldots \theta_n$ is a production, α is the concatenation $\alpha_1 \ldots \alpha_n$, and $cyk(\alpha_k, \theta_k)$ holds for all $k \in \{1, \ldots, n\}$, then $cyk(\alpha, A)$ holds. We have $\alpha \in \mathcal{L}$ if and only if $cyk(\alpha, S)$ holds, where S is the start nonterminal.

One could compute cyk by simply using the definition from above and a Datalog engine that knows about strings. Alternatively, one could compute cyk using a memoized recursive function, but then some care is needed to avoid nontermination. Because the definition of cyk is monotone, the fix is easy: If $cyk(\alpha, A)$ is called while computing $cyk(\alpha, A)$, then return **false** immediately. The intuition is that, in an inductive definition (as opposed to a coinductive one), $cyk(\alpha, A)$ cannot justify itself.

What is the complexity of the CYK algorithm? The first argument of cyk is a substring of α, so there are $O(|\alpha|^2)$ possibilities. If the second argument is a terminal, then cyk does only a constant amount of work; thus, the work for all terminals is $O(|\alpha|^2 \cdot |\Sigma|)$. If the second argument is a nonterminal A, then the algorithm iterates over all productions of the form $A \to \theta_1 \ldots \theta_n$. For each such

production, the string α is split into n substrings $\alpha_1, \ldots, \alpha_n$. There are $\binom{|\alpha|+n-1}{n-1}$ ways to do so, which is exponential in n. For example, if $|\alpha| = n$, then the number of ways is $\Theta(4^n / \sqrt{n})$.

Luckily, there is a simple fix: We can transform the grammar into another one which defines the same language but has short productions. We can do so by repeatedly replacing productions of the form $A \to \theta_1 \ldots \theta_i \theta_{i+1} \ldots \theta_n$ by $A \to \theta_1 \ldots \theta_i A'$ and $A' \to \theta_{i+1} \ldots \theta_n$, where A' is a fresh nonterminal. This will increase the size of the grammar. Let us define the size formally as

$$|G| := |\Sigma| + \sum_{A \to \theta_1 \ldots \theta_n} (n+1)$$

which is the size of the alphabet Σ plus the number of symbols necessary to write down the productions of G. Each replacement increases $|G|$ by 2, because of the two occurrences of the fresh nonterminal A'. If we take care to apply replacements only when both θ_i and θ_{i+1} come from the original grammar (as opposed to being fresh nonterminals introduced by previous replacements), then we will do at most $|G|$ replacements. So, overall, the size $|G|$ of the grammar increases only linearly.

Now suppose that, using the previous transformation, we limit the maximum size of productions to $n = 2$. Then, for one fixed production, there are $|\alpha| + 1$ possible ways to split α into substrings. Thus, the total runtime is $O(|\alpha|^3 \cdot |G|)$.

The CYK algorithm is usually implemented not in the memoized version discussed so far but in a dynamic programming version. In dynamic programming, we must be explicit about the order in which we evaluate $cyk(\alpha, A)$. It is clear we should should look at substrings from shortest to longest. But, it is not clear in which order we should look at nonterminals. The standard solution to this conundrum is to preprocess the grammar even more. We compute the set $\{ A \mid cyk(\epsilon, A) \}$ of *nullable* nonterminals, perhaps by using our memoized version of cyk. Then, for each production $A \to \theta_1 \ldots \theta_{i-1} B \theta_{i+1} \ldots \theta_n$ where B is a nullable nonterminal, we introduce a new production $A \to \theta_1 \ldots \theta_{i-1} \theta_{i+1} \ldots \theta_n$, and repeat this until a fixed-point is reached. Finally, we remove all productions of the form $A \to \epsilon$. In the resulting grammar, $cyk(\epsilon, A)$ is **false** for all nonterminals A, but cyk remains otherwise unchanged. The grammar increases in size, but only linearly if the productions had size bounded by a constant. The main advantage of the new grammar is that it is now safe split α only in *nonempty* substrings $\alpha_1, \ldots, \alpha_n$. This is not sufficient to ensure a unique order, dependant just on $|G|$, which means we still have to perform a fixed point computation to handle the productions of the form $A \to \theta$.

The pseudocode is given in Figure 5. On line 1, we check the special case of the empty string. This is precomputed, perhaps using the memoized version. In lines 3–5, we handle terminals; there is one copy of line 5 for each terminal a. The loop on line 6 goes over substrings of length k, from $k = 2$ to $k = |\alpha|$. Lines 7–11 handle binary productions $A \to \theta_1 \theta_2$; lines 12–15 handle unary productions $A \to \theta$. There are no other productions, because of the preprocessing we performed. For unary productions, the loop on line 13 is responsible with finding a fixed point. A fixed point is guaranteed to be reached in $|G|$ iterations, because each iteration must change the status of at least one nonterminal, but the loop could also stop early if it detects a fixed point.

The variant of CYK we use (Figure 5) runs in $\Theta(|\alpha|^3 \cdot |G| + |\alpha|^2 \cdot |G|^2)$ time and $\Theta(|\alpha|^2 \cdot |G|)$ space. Moreover, the size of the parser's source code is $\Theta(|G|)$. For a plethora of other CYK variants, see (Lange and Leiß 2009).

The bounds from above are important for establishing Theorem 2. First, if we want the generated Java classes to be polynomial in the size of the grammar, it must be that the size of the parser is polynomial in the size of the grammar. We showed above that the

MEMBER(α)

```
 1  if α = ε return nullable(S)
 2  initialize table T with false everywhere
 3  for i ∈ {0, 1, ..., |α| − 1}
 4        // for each terminal a,
 5        if α[i] = a then T[i, i + 1, a] := true
 6  for k ∈ {2, 3, ..., |α|}
 7        for i ∈ {0, 1, ..., |α| − k}
 8              for j ∈ {i + 1, i + 2, ..., i + k − 1}
 9                    // for each production A → θ₁θ₂,
10                    if T[i, j, θ₁] ∧ T[j, i + k, θ₂]
11                          T[i, i + k, A] := true
12        for i ∈ {0, 1, ..., |α| − k}
13              repeat |G| times
14                    // for each production A → θ,
15                    if T[i, i + k, θ] then T[i, i + k, A] := true
16  return T[0, |α|, S]
```

Figure 5. Pseudocode of the CYK parser variant used in the implementation. The parser is for a grammar G whose starting nonterminal is S. We use $T[i, j, \theta]$ to represent $cyk(\alpha[i : j], \theta)$. The pseudocode assumes that the grammar has been preprocessed as described in the main text. Line 5 is a template: it is repeated for each terminal a; lines 10–11 form a template: it is repeated for each binary production; line 15 is a template: it is repeated for each unary production.

size of the parser is $O(|G|)$. Second, if we want the typechecking of the Java code to be done in polynomial time, it must be that the parser works in polynomial time. We showed above that the parser works in $O(|\alpha|^3 \cdot |G|^2)$ time.

The reader may ask why not stick to the simpler memoized version, instead of going for the traditional dynamic programming version? The reason is a pragmatic one. We will implement the parser in a language that we will then translate into Turing machines. To ease the translation to Turing machines, we want the intermediate language to be simple. One way we make it simple is by not having procedure calls. These are not needed for dynamic programming, but they would be needed for memoization.

The existing result of (Gil and Levy 2016) handles only context free languages that are deterministic. Intuitively, as explained by (Knuth 1965), *deterministic languages* are those that can be parsed in one pass from left to right. Formally, *deterministic languages* are those defined by LR(k) grammars or, equivalently, by LR(1) grammars or by deterministic pushdown automata. *Unambiguous grammars* are those for which each string has at most one parse tree. Unambiguous grammars are a strict superset of LR grammars. The latter could be called prefix-unambiguous: not only that strings need to correspond to at most one parse tree, but also prefixes must correspond to at most one incomplete parse tree. A language is said to be *inherently ambiguous* if there exist grammars that define it but all are ambiguous; a standard example (Hopcroft and Ullman 1979) is $\mathcal{L}_{\mathrm{ambig}} := \mathcal{L}_{()()} \cup \mathcal{L}_{(())}$ where

$$\mathcal{L}_{()()} := \{\, a^m b^m c^n d^n \mid m, n \in \mathbb{N} \,\}$$
$$\mathcal{L}_{(())} := \{\, a^m b^n c^n d^m \mid m, n \in \mathbb{N} \,\}$$

Clearly, inherently ambiguous languages are not deterministic and therefore cannot be recognized with the approach of (Gil and Levy 2016). That is why we shall use $\mathcal{L}_{\mathrm{ambig}}$ as an example later on.

6.3 A Simper Language

This section introduces a simple, imperative language, which we call *Simper*. In Simper, we can easily implement CYK parsers.

Furthermore, we can easily compile any Simper program into a Turing machine (§7.2).

The syntax of Simper is given in Figure 6. Minsky machines have counters and zero tests. The RAM computation model adds an infinite array with store and load statements (Goldreich 2008). Instead, Simper adds named arrays of arbitrary size, a tiny type system, and some syntactic sugar. These features are added to ease the implementation of parsers, such as the one in Figure 5. Other features, such as procedures and arithmetic operations, are not added in order to keep the translation to Turing machines simple.

The type system of Simper is given in Figure 7. The basic types are nat (for nonnegative integers), sym (for symbols), and bool (for booleans). The type $\mathrm{array}\, n\, t$ is the type of n-dimensional arrays with elements of type t. We assume a special 1-dimensional array of symbols called $input$, whose length is given by a special variable n. Thus, $input\,:\,\mathrm{array}\,1\,\mathrm{sym}$ and $n\,:\,\mathrm{nat}$. Simper programs output only one bit: whether they halt (by executing halt) or they get stuck (by reaching the end of the list of statements).

The semantics are straightforward, with a few quirks. First, since nat holds only nonnegative integers, $-- 0$ evaluates to 0. Second, the notation $\mathrm{array}[v_1, \ldots, v_n](v')$ stands for an array filled with v' that has n dimensions, and the valid indices along the ith dimension are $0, \ldots, v_i - 1$.

6.4 Example: Parsers for $\mathcal{L}_{\mathrm{ambig}}$

Instead of giving formal semantics for Simper, let us just look at some examples. The following is a specialized parser for $\mathcal{L}_{\mathrm{ambig}}$:

```
i := 0
a := 0  while i != n && input[i] == "a" { ++i ++a }
b := 0  while i != n && input[i] == "b" { ++i ++b }
c := 0  while i != n && input[i] == "c" { ++i ++c }
d := 0  while i != n && input[i] == "d" { ++i ++d }
if i == n {
  if a == b && c == d { halt }
  if a == d && b == c { halt }
}
```

Like in Java, in the expression $c_1 \, \&\& \, c_2$, we evaluate c_2 only if c_1 evaluates to false.

Alternatively, we can describe $\mathcal{L}_{\mathrm{ambig}}$ using a context free grammar, and then we can apply the recipe from §6.2. Let us do this, to illustrate what the proof of concept implementation actually generates.

We start with the following grammar:

$$
\begin{array}{llll}
S \to X & X \to aXd & X \to F & Y \to EG \\
S \to Y & E \to aEb & F \to bFc & G \to cGd \\
 & E \to \epsilon & F \to \epsilon & G \to \epsilon
\end{array}
$$

After limiting the length of the right hand sides to ≤ 2 and after eliminating ϵ from the language of each nonterminal, we are left with the binary productions

$$
\begin{array}{lll}
Y \to EG & X \to aX' & X' \to Xd \\
F \to bF' & F' \to Fc & E \to aE' \\
E' \to Eb & G \to cG' & G' \to Gd
\end{array}
$$

and the unary productions

$$
\begin{array}{llll}
S \to X & X \to F & Y \to E & Y \to G \\
S \to Y & E' \to b & F' \to c & G' \to d
\end{array}
$$

We can now start implementing the pseudocode from Figure 5 in Simper. Since $\epsilon \in \mathcal{L}_{\mathrm{ambig}}$, we start with a check for this special case:

```
if n == 0 { halt }
```

$$s \;\to\; \ell_{\underline{:}} \;\mid\; \underline{\text{goto}}\,\ell \;\mid\; l\,\underline{:=}\,v \;\mid\; \underline{\text{if}}\,c\,\underline{\{}s^*\underline{\}}\,\big(\underline{\text{else}}\,\underline{\{}s^*\underline{\}}\big)^? \;\mid\; \underline{++}l \;\mid\; \underline{--}l \;\mid\; \underline{\text{halt}} \qquad\text{statements (core)}$$

$$s \;\to\; \underline{\text{while}}\,c\,\underline{\{}s^*\underline{\}} \;\mid\; \underline{\text{switch}}\,v\,\underline{\{}\big(v\,\underline{\{}s^*\underline{\}}\big)^*\underline{\}} \qquad\text{statements (sugar)}$$

$$v \;\to\; l \;\mid\; r \qquad\text{values}$$

$$l \;\to\; id\big(\underline{[}v(\underline{,}v)^*\underline{]}\big)^? \qquad\text{left values}$$

$$r \;\to\; nat \;\mid\; \underline{\text{``}}string\underline{\text{''}} \;\mid\; \underline{\text{array}}[v(\underline{,}v)^*]\underline{(}v\underline{)} \qquad\text{right values}$$

$$c \;\to\; c\,\underline{\&\&}\,c \;\mid\; c\,\underline{||}\,c \;\mid\; v\,\underline{==}\,v \;\mid\; v\,\underline{!=}\,v \qquad\text{conditions}$$

Figure 6. Simper syntax. The underlined parts appear as given; that is, they are terminals. The notation $a \to b \mid c$ is shorthand for $a \to b$ and $a \to c$, as usual. The notation s^* is shorthand for 'list of zero or more s'. The notation $s^?$ is shorthand for 'zero or one s'.

$$\frac{l \,:\, t \qquad v \,:\, t}{l := v \;:\; \text{unit}} \qquad \frac{c \,:\, \text{bool} \qquad b_1 \,:\, \text{unit} \qquad b_2 \,:\, \text{unit}}{\text{if } c\,\{b_1\}\,\text{else}\,\{b_2\} \;:\; \text{unit}} \qquad \frac{l \,:\, \text{nat}}{++l \;:\; \text{unit}} \qquad \frac{}{\text{halt} \;:\; \text{unit}}$$

$$\frac{c_1 \,:\, \text{bool} \qquad c_2 \,:\, \text{bool}}{c_1 \,\&\&\, c_2 \;:\; \text{bool}} \qquad \frac{x \,:\, \text{array}\,n\,t \qquad e_1 \,:\, \text{nat} \quad \dots \quad e_n \,:\, \text{nat}}{x[v_1, \dots, v_n] \;:\; t} \qquad \frac{s_1 \,:\, \text{unit} \quad \dots \quad s_n \,:\, \text{unit}}{s_1 \dots s_n \;:\; \text{unit}} \qquad \frac{}{0 \;:\; \text{nat}}$$

$$\frac{v_1 \,:\, t \qquad v_2 \,:\, t}{v_1 == v_2 \;:\; \text{bool}} \qquad \frac{v' \,:\, t \qquad v_1 \,:\, \text{nat} \quad \dots \quad v_n \,:\, \text{nat}}{\text{array}[v_1, \dots, v_n](v') \;:\; \text{array}\,n\,t} \qquad \frac{}{x \;:\; t} \qquad \frac{}{\text{``foo''} \;:\; \text{sym}}$$

Figure 7. Some representative typing rules of Simper.

Next, we choose some arbitrary injective mapping from symbols to nonnegative integers.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S	X	X'	Y	Y'	E	E'	F	F'	G	G'	a	b	c	d

We can now continue with the initialization of the array T:

```
sn := n   ++sn   T := array[sn,sn,15](0)
i := 0 si := 1  while i != n {
  switch input[i] {
    "a" { T[i,si,11] := 1 }
    "b" { T[i,si,12] := 1 }
    "c" { T[i,si,13] := 1 }
    "d" { T[i,si,14] := 1 }
  }
  ++i   ++si
}
```

Finally, we have the main loop:

```
k := 2  while k != sn {
  i := 0  ik := k  while ik != sn {
    j := i  ++j  while j != ik {
      // for Y  →  EG
      if T[i,j,5] == 1 && T[j,ik,9] == 1 {
        T[i,ik,3] := 1
      }
      ... eight other binary productions ...
      ++j
    }
    ++i   ++ik
  }
  i := 0  ik := k  while ik != sn {
    j := 0  while j != 11 { // 11 nonterminals
      // for S  →  X
      if T[i,ik,1] == 1 { T[i,ik,0] := 1 }
      ... seven other unary productions ...
      ++j
    }
    ++i   ++ik
  }
}
if T[0,n,0] == 1 { halt }
```

There are three things to notice. First, the lack of arithmetic in Simper forces us to introduce some auxiliary variables, such as `ik` which stands for $i + k$. Overall, however, the inconveniences are minor. Second, these deviations from the pseudocode do not affect the size and the running time of the program. But, third, we also notice that the CYK parser is significantly more complicated than the specialized parser we started with.

To complete the argument for Theorem 2, we continue by showing how a Java type checker can be used as an interpreter for any Simper program.

7. Using a Java Compiler as an Interpreter

This section describes a compiler of Simper programs into Java code. The generated Java code type checks if and only if the original Simper program reaches a `halt` statement. Therefore, a Java type checker can be used as an interpreter. The compilation is done in two phases: a translation from Simper programs into Turing machines (§7.2), followed by a translation from Turing machines into Java code (§7.3). The reason for splitting the compilation in two phases is that we get to reuse the proof of Theorem 1, for the second phase. We shall use an extended version of Turing machines (§7.1), which slightly complicates the second phase but greatly simplifies the first phase.

The compiler described in this section is part of the proof of Theorem 2. So, this section has three goals: (1) to argue that the compilation is correct, (2) to argue that the increase in code size is polynomial only, and (3) to argue that programs with a polynomial runtime will still have a polynomial runtime after compilation. The last two points do not guarantee that the compilation is practical, but they do suggest that practical compilation can be achieved (§7.4).

7.1 Extended Turing Machines

Why do we want to extend Turing machines? The type nat of Simper allows arbitrarily large nonnegative integers. This means that we do not know in advance how many digits are necessary to store the value of a nat variable. In turn, this means that the space allocated

for some nat variables will occasionally need to be extended. With a normal Turing machine tape, one would need to shift the content of roughly half of the tape. Instead, we will allow extended Turing machines to insert symbols, as a primitive operation. The proof of Theorem 1 can easily be adjusted to account for such insertions.

An *extended Turing machine* $\mathcal{E}$ is a tuple $(Q, q_\mathsf{I}, q_\mathsf{H}, \Sigma, \delta)$, where Q is a finite set of *states*, q_I is the *initial state*, q_H is the *halt state*, Σ is a finite *alphabet*, and $\delta : Q \times \Sigma_\perp \to Q \times \Sigma^* \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}$ is a *transition function*. This definition is the same with that of Turing machines, except for the type of the transition function. A *configuration* is a tuple (q, α, b, γ) of the current state q, the left part of the tape $\alpha \in \Sigma^*$, the current symbol $b \in \Sigma_\perp$, and the right part of the tape $\gamma \in \Sigma^*$. The *execution steps* of $\mathcal{E}$ are the following:

$$(q, \alpha a, b, \gamma) \to (q', \alpha, a, \beta\gamma) \qquad \text{for } \delta(q,b) = (q', \beta, \mathsf{L})$$
$$(q, \alpha, b, \gamma) \to (q', \alpha, b', \gamma) \qquad \text{for } \delta(q,b) = (q', b', \mathsf{S})$$
$$(q, \alpha, b, c\gamma) \to (q', \alpha\beta, c, \gamma) \qquad \text{for } \delta(q,b) = (q', \beta, \mathsf{R})$$

If $\delta(q,b) = (q', \beta, d)$, we require that $d = \mathsf{S}$ only if $|\beta| = 1$. We also allow for execution steps that go outside the existing tape:

$$(q, \epsilon, b, \gamma) \to (q', \epsilon, \perp, \beta\gamma) \qquad \text{for } \delta(q,b) = (q', \beta, \mathsf{L})$$
$$(q, \alpha, b, \epsilon) \to (q', \alpha\beta, \perp, \epsilon) \qquad \text{for } \delta(q,b) = (q', \beta, \mathsf{S})$$

As before, we require that $\delta(q_\mathsf{H}, b) = (q_\mathsf{H}, b, \mathsf{S})$ for all $b \in \Sigma$, and $\delta(q_\mathsf{H}, \perp) = (q_\mathsf{H}, b, \mathsf{S})$ for some $b \in \Sigma$. A *run* on input tape α_I is a sequence of execution steps starting from configuration $(q_\mathsf{I}, \epsilon, \perp, \alpha_\mathsf{I})$. If $\mathcal{E}$ reaches q_H we say that $\mathcal{E}$ halts on α_I.

Proposition 8. *We can convert between Turing machines and extended Turing machine with only polynomial increases in machine size and while preserving polynomial runtimes.*

Proof sketch. The conversion from Turing machines to extended Turing machines is trivial. The conversion from extended Turing machines to Turing machines can be done by shifting half-tapes whenever the length of the tape increases. If the original runtime was polynomial, then the length of the tape is polynomial and so the extra work for shifting is also polynomial. □

7.2 From Programs to Turing Machines

This section describes a compiler of Simper programs into extended Turing machines. We will see (a) how the Turing tape is organized, (b) what are the main ideas for translating the core Simper statements, and (c) how the compiler itself is organized.

Tape Content. Simper variables hold values of two types, nat and sym. (No variable has type bool, which is used for conditions.) The type nat is unbounded: any integer can be incremented. The type sym is bounded: we can check if two symbols are equal but we cannot combine symbols to produce other symbols. Values of type nat will be represented using bits; values of type sym will be present in the alphabet of the target machine. So, each constant of type sym from the Simper program (such as "foo") will have a letter in the alphabet of the target machine.

In general, the alphabet of the target machine is made out of bits, symbols, and markers. One use of markers is to delimit tape zones that store variable values. Each variable x of the Simper program is stored in a tape zone delimited by the markers $\langle_x$ and $\rangle_x$. The tape starts with zones for the special variables (*input* and n) and continues with zones for the other variables, in no particular order.

We also use markers to represent arrays. Each element of a 1-dimensional array is delimited by markers $\langle^0$ and $\rangle^0$. If a is an n-dimensional array, let us write $a[i]$ for an $(n-1)$-dimensional subarray such that $a[i][i_1, \ldots, i_{n-1}] = a[i, i_1, \ldots, i_{n-1}]$ for all $i_1, \ldots, i_{n-1}$. Let us also write $\lfloor a \rfloor$ for the representation of value a.

Then, if a is an $(n+1)$-dimensional array whose kth coordinate ranges over $0, \ldots, d_k - 1$, we define

$$\lfloor a \rfloor := \langle^n \lfloor a[0] \rfloor \rangle^n \langle^n \lfloor a[1] \rfloor \rangle^n \ldots \langle^n \lfloor a[d_0 - 1] \rfloor \rangle^n \qquad (9)$$

This is a recursive definition. To specify the base case, we simply regard array $0\, t$ as isomorphic to t:

$$\text{array } 0\, t \simeq t \qquad (10)$$

For example, the representation of the 2-dimensional array $\left(\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right)$ is the following:

$$\lfloor \left(\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right) \rfloor = \langle^1 \lfloor (0\ 1) \rfloor \rangle^1 \langle^1 \lfloor (2\ 3) \rfloor \rangle^1 \qquad \text{by (9)}$$
$$= \langle^1 \langle^0 \lfloor (0) \rfloor \rangle^0 \langle^0 \lfloor (1) \rfloor \rangle^0 \rangle^1 \langle^1 \langle^0 \lfloor (2) \rfloor \rangle^0 \langle^0 \lfloor (3) \rfloor \rangle^0 \rangle^1 \qquad \text{by (9)}$$
$$= \langle^1 \langle^0 \lfloor 0 \rfloor \rangle^0 \langle^0 \lfloor 1 \rfloor \rangle^0 \rangle^1 \langle^1 \langle^0 \lfloor 2 \rfloor \rangle^0 \langle^0 \lfloor 3 \rfloor \rangle^0 \rangle^1 \qquad \text{by (10)}$$
$$= \langle^1 \langle^0 0 \rangle^0 \langle^0 1 \rangle^0 \rangle^1 \langle^1 \langle^0 01 \rangle^0 \langle^0 11 \rangle^0 \rangle^1$$

On the last line, nat values are represented in binary, with the most significant bit towards the right. (This bit order, which is used in the implementation, is an arbitrary convention.)

Translation of Statements. Roughly speaking, Simper programs can be visualized as flowgraphs whose arcs are labeled by statements. Each vertex of the flowgraph will correspond to a state of the Turing machine. The translation work consists of transforming each statement arc into a set of Turing transitions which have the same effect.

The increment statement $++l$ is simulated in two phases. First, the head of the Turing machine moves to the tape zone designated by l. Second, a prefix of the form $11 \ldots 10$ is changed into $00 \ldots 01$. If the content of the tape zone designated by l is all 1s, then they are all changed to 0s and an extra 1 is inserted at the end. We can do such insertions because we use an extended model of Turing machines. The decrement statement $--l$ is simulated analogously.

What is the tape zone designated by l? If l is a variable x, then the tape zone it designates is the zone between the markers $\langle_x$ and $\rangle_x$. One could go left until $\langle_{input}$ is reached, and then right until $\langle_x$ is reached. At this point, the head of the Turing machine is at the left extremity of the tape zone we want. If l is an array element $a[v_0, \ldots, v_{n-1}]$, then the tape zone it designates is more difficult to find. We begin similarly, by finding $\langle_a$. Let i_k be the integer to which v_k evaluates. What we want to do is to move right until the $(i_0 + 1)$th $\langle^{n-1}$ marker is found, then move right until $(i_1 + 1)$th $\langle^{n-2}$ marker is found, and so on. At the end we would have found a marker $\langle^0$. The tape zone we want is between this marker and its $\rangle^0$ pair.

However, since the values $i_0, \ldots, i_{n-1}$ are unbounded, we must keep track of them using the tape. The easiest way to do so is to introduce auxiliary variables, and hence allocate extra tape zones. We do so in a preprocessing step that works as follows. Suppose a statement s contains the subexpression $a[v_0, \ldots, v_{n-1}]$. We replace the subexpression by $a[x_0, \ldots, x_{n-1}]$, where $x_0, \ldots, x_{n-1}$ are fresh variables, and we insert before s assignments $x_k := v_k$ for each k in $0, \ldots, n-1$. Because some of the expressions $v_0, \ldots, v_{n-1}$ might themselves refer to array elements, we apply this transformation recursively. The result is that all array accesses have the form $a[x_0, \ldots, x_{n-1}]$. Moreover, the values stored in $x_0, \ldots, x_{n-1}$ are used for this array access only, which means that after the access we can change their values without affecting the semantics of the program. In fact, we will change their value *while* we access the array element.

Finding the tape zone where an array element is stored involves several steps of the form 'move right until the $(i_k + 1)$th $\langle^{n-k-1}$ marker'. We will implement this by executing the step 'move right until a $\langle^{n-k-1}$ marker' $i_k + 1$ times. How do we repeat $i_k + 1$ times? Recall that, after preprocessing, i_k is the value stored in variable x_k,

and also recall that we can change the content of variable x_k. So, we alternate between moving right and decrementing x_k. This, of course, requires yet another marker to remember our position in the array while we decrement x_k.

To simulate assignments $l := r$, we proceed as follows. First, we find the tape zone designated by l, we remove its content, and we mark its position by $\downarrow$. These operations would be cumbersome with traditional Turing machines, but are easy with extended Turing machines. Second, we store the value of r in the zone marked by $\downarrow$. How we do this second step depends on what exactly r is. If r is a variable or an array access, then it designates a tape zone. In that case, we identify that tape zone, mark its left side by $\uparrow$, and then copy from $\uparrow$ to $\downarrow$ until a $\rangle$ delimiter is found. If r is a constant of type nat or sym, then we simply write that value in the zone marked by $\downarrow$. If r is a constant of type array $n\,t$, then we construct an array representation as in (9), using the trick with auxiliary variables for counting, just as we did for locating array elements. And third, we finish by removing markers, such as $\uparrow$ and $\downarrow$.

To simulate an if statement, we must evaluate conditions. Let us see how to evaluate an atomic condition $v_1 == v_2$. The expressions v_1 and v_2 can be variables, array accesses, or constants. Using auxiliary variables, as we did for array accesses, we can reduce the problem of evaluating $v_1 == v_2$ to the problem of evaluating $x_1 == x_2$, with x_1 and x_2 being variables. To compare the values of x_1 and x_2, we mark the left extremities of their tape zones by $\downarrow$ and $\uparrow$, respectively. Then we repeatedly compare the symbols on the right of the two markers, and move the markers to the right, until a $\rangle$ delimiter is reached. To compare one pair of symbols, we must remember one of the symbols while we move from $\downarrow$ to $\uparrow$. Because there is only a finite number of symbols, we can encode the required information in the state of the Turing machine. The other atomic condition, $v_1 \mathrel{!=} v_2$, is simulated analogously.

Composing conditions, labels, and goto statements do not pose any interesting challenges.

Organization of the Compiler. The compiler is written in OCaml. Its source code reads like that of an interpreter that is written in an imperative style. For example, the translation of assignments is done by the following function:

```
let convert_assignment left right : transformer =
  ( markLvalue target_mark left
  & eraseBitsAfter target_mark
  & assign right )
```

'Find and mark the tape zone designated by `left`; then erase its content; then fill it with the value of `right`.' This code closely matches the informal description we saw earlier. Let us see how this is achieved.

Turing machines are represented essentially as lists of transitions.

```
type machine =
  { states : state list
  ; alphabet : letter list
  ; transitions : transition list }
```

The compiler makes heavy use of transformers.

```
type transformer = (state * machine) -> (state * machine)
```

Intuitively, if t is a transformer, then calling $t(q, \mathcal{T})$ will add some transitions emanating from state q and eventually converging in a fresh state q'. The call $t(q, \mathcal{T})$ then returns a pair $(q', \mathcal{T}')$, where $\mathcal{T}'$ is a machine obtained by adding the extra transitions to $\mathcal{T}$. Suppose we have a transformer t_1 that simulates the Simper statement s_1, and a transformer t_1 that simulates the Simper statement s_2. If we want to simulate executing s_1 and then s_2, then we need to apply t_1 and then t_2 to the Turing machine being built. To compose transformers, we use the following straightforward combinators:

```
let ( & ) f g x = g (f x)
let seqs = List.fold_left ( & ) (fun x->x)
```

We can now see that the definition of `convert_assignment` simply composes three transformers.

To implement if statements, we also make use of three new types of transformers: `branch`, `join`, and `transformer2`.

```
type state2 = { yes_branch : state; no_branch : state }
type branch = state * machine -> state2 * machine
type join = state2 * machine -> state * machine
type transformer2 = state2 * machine -> state2 * machine
```

Such transformers are useful whenever we need to take decisions. For example, when we copy a value from one variable to another, we need to stop when we see a $\rangle$ delimiter. For taking such decisions, we use branching and joining transformers. For a concrete and simple example of how such transformers are used, let us look at how the if statement is translated.

```
let convert_if condition ys ns : transformer =
  ( branch_on condition
    & yes_branch (convert_body ys)
    & no_branch (convert_body ns)
  & join )
```

The types involved are the following:

```
val branch_on : condition -> branch
val yes_branch : transformer -> transformer2
val no_branch : transformer -> transformer2
val join : join
```

The combinator `yes_branch` is implemented as follows:

```
let yes_branch t = fun ({ yes_branch; no_branch }, tm) ->
  let yes_branch, tm = t (yes_branch, tm) in
  ({ yes_branch; no_branch }, tm)
```

The combinator `no_branch` is implemented analogously. The (join) transformer `join` adds a fresh state and no-op transitions to it from both of the given states. The (branch) transformer builder `branch_on` is more involved, since it depends on `condition`. The function `convert_body` is defined as follows:

```
let rec convert_body (xs : statement list) : transformer =
  seqs (List.map convert_statement xs)
and convert_statement (x : statement) : transformer =
  (* ... uses convert_body ... *)
```

where `seqs`, defined earlier, takes a list of transformers and chains them sequentially.

Transformers and combinators for transformers – this is the mortar used to build the compiler. The bricks are transformers such as `goLeft`, `goToMark`, `insertOnRight`, `write`, and many others.

7.3 From Turing Machines to Java

The next step is to compile Turing machines into Java code. This is achieved by a tiny compiler, which implements the proof of Theorem 1, with small adaptations to allow for our extensions (§7.1).

The difference between normal and extended Turing machines can be seen in the transition function type:

normal type $Q \times \Sigma_\perp \to Q \times \Sigma \times \{-1, 0, +1\}$

extended type $Q \times \Sigma_\perp \to Q \times \Sigma^* \times \{-1, 0, +1\}$

Consider an extended Turing machine such that $|\beta| = 1$ whenever $\delta(q, b) = (q', \beta, d)$. It is easy to see that such an extended machine trivially corresponds to a normal one. Accordingly, we translate such an extended machine in exactly the same way as we would translate the normal one. We only need to do something different for those transitions with $|\beta| \neq 1$. These differences are illustrated in Figure 8. Briefly, each $\beta = b_0 \ldots b_{n-1}$ in the transition function results in a string $L_{b_0} N \ldots L_{b_{n-1}} N$ of classes in the corresponding inheritance rule. The special case $|\beta| = 1$ is identical to the translation for non-extended transition functions.

This concludes our translation from Simper programs to Java code. The Simper program halts if and only if the Java code type

$$Q_s^L x <:: L_b N Q_{s'}^{wL} M^L N L_{b_0} N L_{b_1} N \ldots L_{b_{n-1}} N x \qquad\qquad \text{for } \delta(q_s, b) = (q_{s'}, b_0 b_1 \ldots b_{n-1}, \mathsf{L})$$

$$Q_s^L x <:: L_b N Q_{s'}^{wL} M^R N L_{b'} N x \qquad\qquad \text{for } \delta(q_s, b) = (q_{s'}, b', \mathsf{S})$$

$$Q_s^L x <:: L_b N Q_{s'}^{wL} L_{b_0} N L_{b_1} N \ldots L_{b_{n-1}} N M^R N x \qquad\qquad \text{for } \delta(q_s, b) = (q_{s'}, b_0 b_1 \ldots b_{n-1}, \mathsf{R})$$

$$Q_s^L x <:: L_\# N Q_{s'}^{wL} L_\# N M^L N L_{b_0} N L_{b_1} N \ldots L_{b_{n-1}} N x \qquad\qquad \text{for } \delta(q_s, \bot) = (q_{s'}, b_0 b_1 \ldots b_{n-1}, \mathsf{L})$$

$$Q_s^L x <:: L_\# N Q_{s'}^{wL} L_\# N M^R N L_{b'} N x \qquad\qquad \text{for } \delta(q_s, \bot) = (q_{s'}, b', \mathsf{S})$$

$$Q_s^L x <:: L_\# N Q_{s'}^{wL} L_\# N L_{b_0} N L_{b_1} N \ldots L_{b_{n-1}} N M^R N x \qquad\qquad \text{for } \delta(q_s, \bot) = (q_{s'}, b_0 b_1 \ldots b_{n-1}, \mathsf{R})$$

Figure 8. Class table used to simulate an *extended* Turing machine $\mathcal{E} = (Q, q_I, q_H, \Sigma, \delta)$. There are six more rules obtained from the ones above by swapping $\mathsf{L} \leftrightarrow \mathsf{R}$. The inheritance rules not involving the transition function δ are the same as in Figure 3.

checks, on all inputs. For Simper programs, the input is provided in the special array $input$. For Java code, the input is provided by a builder (§6.1).

7.4 Discussion on Efficiency

The previous sections (§7.2 and §7.3) describe a translation from Simper programs to Java code. Is it efficient? In theory, yes; in practice, no. We consider two aspects of the generated Java code: (1) its size, and (2) how fast it can be type checked.

In theory. The Java code has size quadratic in the size of the source Simper program. More precisely, the translation from Simper programs to Turing machines is linear, while the translation from Turing machines to Java code is quadratic. Let us see why.

To simulate increments, decrements, copying and comparisons of values, we put, in the Turing machine, loops that go over the bits or symbols of a variable or a couple of variables. Such a loop requires only a constant number of states and transitions. The most involved construction we had was for identifying the tape zone of an array element. That construction was essentially a sequence of n loops, where n is the dimension of the array. In all cases, the part of a Turing machine that simulates a statement s has a size proportional to the size of s. In particular, referring to an element of an n-dimensional array requires $\Theta(n)$ characters in the Simper program, and $\Theta(n)$ states and transitions in the Turing machine. In addition, our construction can be done with a transition function such that $|\beta| \leq 2$ whenever $\delta(q, b) = (q', \beta, d)$.

From the Turing machine, we generate two pieces of Java code: (i) interface declarations corresponding to Figure 8, and (ii) a builder class as in Figure 4. By examining these figures, we see that the interface declarations will have size $\Theta(|Q| \cdot |\Sigma|)$, and that the builder class will have size $\Theta(|\Sigma|)$. Speaking roughly, we can describe this as being quadratic in the size of the Turing machine.

Let us now analyze the speed of the simulation. We start with a Simper program whose runtime is t, possibly depending on the size of the input. We end up with Java code that specifies a subtyping machine, whose runtime is t'. The subtyping machine simulates the Simper program. The goal in what follows is to put on upper bound on t', as a function of t.

To find a bound, we need to be more precise about how to measure resource usage of Simper programs, Turing machines, and subtyping machines. Let us start with Simper programs. We consider that each value of type nat or sym takes 1 memory cell. The space taken by an array is obtained by adding the space taken by its elements. For time, we count each core statement as taking 1 step, with one exception. The exception is the creation of arrays as a result of using an array literal $\mathsf{array}[v_0, \ldots, v_{n-1}](v')$: the time for creating such an array is the space occupied by the array. With these rules, we see that, if the runtime of a Simper program is t, then the

space it uses is $\leq t$. Also, since nat values can only be incremented, all such values are at most $O(t)$.

Now let us move on to extended Turing machines. We define the size of a configuration (q, α, b, γ) as $\lg|Q| + |\alpha b \gamma|$. The time taken to execute a Turing machine is the distance in the configuration graph from the initial configuration to a final configuration. We organized the tape into zones, one for each variable in the Simper program, plus a few zones for auxiliary variables. Each type zone for a value of type sym has size $O(1)$; each type zone for a value of type nat has size $O(\log t)$; each type zone for a value of type array has size $O(t \log t)$. All auxiliary variables have type nat or sym, and the number of auxiliary variables is at most $O(p)$, where p is the size of the text of the Simper program. In all, the tape of the Turing machine has size at most $O(pt \log t)$. What about time? Each step of a Simper program is simulated by a constant number of Turing transitions. These transitions may loop over the tape symbols used to represent one Simper value. In the case of nat values, this gives an overhead of $O(\log t)$. For operations such as comparison, the head of the Turing machine need to move from one tape zone to another, for another overhead of $O(pt \log t)$. In all, a Simper program that runs in t time steps is simulated by an extended Turing machine in $O(pt^2 \log^2 t)$ time steps.

Now let us move on to subtyping machines. For these we measure space and time as we do for extended Turing machine: space is the size of the tape; time is the number of transitions in the configuration graph. As explained in Section 5 (Figure 2), there is a direct correspondence between configurations of Turing machines and those of subtyping machines. Thus, the space used by subtyping machines is also $O(pt \log t)$. For time, we note that a subtyping machine head may move back-and-forth once in order to simulate one step of the Turing machine. Thus, the time spent in the subtyping machine is $O(p^2 t^3 \log^3 t)$.

Let us see how the slowdown affects one particular type of Simper programs: the CYK parsers we used earlier (§6.2). In that case, $t \in O(|\alpha|^3 \cdot |G|^2)$ and $p \in O(|G|)$. Therefore, the subtyping machine runs in time $\tilde{O}(|\alpha|^9 \cdot |G|^8)$. This establishes the polynomial time promised in Theorem 2. However, the degree of the polynomial is not exactly encouraging. There is room for improvement, and work to be done to achieve a practical parser generator for fluent interfaces.

In practice. Let us consider the example of recognizing the language $\mathcal{L}_{\mathrm{ambig}}$. With the current implementation, the generated Java code is over 1 GiB, and is too big for javac to handle in reasonable time. However, if we hand-craft a Turing machine that recognizes $\mathcal{L}_{\mathrm{ambig}}$, then the generated Java code is only a few MiB in size. It is then possible to use javac to recognize $\mathcal{L}_{\mathrm{ambig}}$, with strings of up to ten letters being handled in minutes.

The main bottleneck now is the compiler from Simper to Turing machines. To improve its performance, one may extend Turing

machines further; for example, with a 'go to marker' primitive operation. Or, perhaps one might find a reduction that does not use Turing machines as an intermediate representation.

8. Discussion

Does the reduction from Section 5 apply to other languages like $C^\sharp$ and Scala? No. Both of them adopted the recursive–expansive restriction of (Viroli 2000), as recommended by (Kennedy and Pierce 2007). Roughly, this restriction is a syntactic check that succeeds if and only if our Turing tapes are bounded. For bounded tapes, the halting problem is decidable.

What is the practical relevance of Theorem 1? For C, there exists a formally verified type checker in CompCert (Leroy 2009, version 2.5). For Java, Theorem 1 implies that a formally verified type checker that guarantees partial correctness cannot also guarantee termination. In most applications, partial correctness suffices, but not so in security critical applications (where users are malicious), nor in mission critical applications (where nontermination is costly). Since there cannot be a totally correct Java type checker, Theorem 1 strengthens the motivation behind research into alternative type systems, such as (Greenman et al. 2014) and (Zhang et al. 2015).

It is perhaps difficult to imagine how could the Java type checker be the target of a security attack. However, consider a scenario in which generics are reified, as has been discussed for a future version of Java. In that case, one needs to perform subtype checks at runtime to implement `instanceof`. In other words, `instanceof` becomes a potential security vulnerability for any Java program that uses it. A simple fix might be to change the specification of `instanceof` to allow it to throw a timeout exception. Let us hope that a better solution exists. Similar problems occur if one tries to add gradual typing to Java.

What is the practical relevance of Theorem 2? On the one hand, it hints that it may be possible to implement parser generators for fluent interfaces, which would make it easier to embed domain specific languages in Java. On another hand, the techniques used in its proof may be reusable for encoding other computations in Java's type system.

To summarize, Theorems 1 and 2 have practical implications:

1. A formally verified type checker for Java can guarantee partial correctness but cannot guarantee termination.

2. If reified generics are added to Java, then one needs to find some solution for not turning `instanceof` into a security problem. Similarly, any technique (such as gradual typing) that involves subtype checking at runtime is a potential security problem.

3. It may be possible to develop parser generators that help with embedding domain specific languages in Java.

Still, Theorem 1 is primarily of theoretical interest: It strengthens the best lower bound on Java type checking (Gil and Levy 2016), and finally answers a question posed almost a decade ago by (Kennedy and Pierce 2007).

9. Related Work

Java's type system is not only undecidable, but also unsound (Amin and Tate 2016). Many other languages have undecidable type systems: Haskell with extensions (Wansbrough 1998; Sulzmann et al. 2007), OCaml (Lillibridge 1997; Rossberg 1999), C++ (Veldhuizen 2003), and Scala (Bjarnason 2009; Odersky 2016), to name a few. It is often the case that undecidable type systems are a fun playground for metaprogramming. For example, in Haskell, undecidable instances are useful for deriving type classes (Weirich 2006); and, in C++, libraries like Boost.Hana (Dionne 2016) allow one to easily

write (mostly) functional programs that run at compile time. Will Java's type system become a playground for metaprogramming? (Gil and Levy 2016) and Theorem 2 show that it is possible in principle. It remains to be seen whether it is possible in practice.

On the theory side, perhaps the closest result is that subtype checking is undecidable in $F_{<:}$ (Pierce 1994). That proof has some high level similarities with the one presented here: it considers a deterministic type system, and shows a reduction from (a kind of) Turing machines, by using an intermediate formalism. We also know that System F is undecidable (Wells 1999). But, its subset known as Hindley–Milner is decidable, in fact type inference is DEXPTIME-complete (Mairson 1990).

Why should we care about toy type systems like $F_{<:}$ and that of System F? Because real type systems like Java's are too difficult for humans to reason about. There are three ways in which one can settle the decidability of type checking for a real world language X: (i) find a simple type system that is less expressive than that of X, and show it is undecidable; (ii) find a simple type system that is more expressive than that of X, and show that it is decidable; or (iii) tackle the original type system but do a mechanically verified proof. For (iii), a huge effort is required. This paper falls in category (i). The simple type system is neither $F_{<:}$ nor System F. Instead, it is taken from (Kennedy and Pierce 2007). Kennedy and Pierce study several variations of the type system they propose. Alas, those variations they prove decidable are less expressive than Java, and those variations they prove undecidable are not less expressive than Java. The variant they prove undecidable allows classes of arity 2 and multiple instantiation inheritance, which in our setting translates to having nondeterministic subtyping machines. Their reduction is from PCP. They conjecture that multiple instantiation inheritance is essential for undecidability. That turns out not to be the case.

10. Conclusion

It is possible to coerce Java's type checker into performing *any* computation. This opens possibilities for use and abuse.

References

N. Amin and R. Tate. Java and Scala's type systems are unsound: The existential crisis of null pointers. In *OOPSLA*, 2016.

R. O. Bjarnason. More Scala typehackery. https://apocalisp.wordpress.com/2009/09/02/ (see also https://michid.wordpress.com/2010/01/29), 2009.

A. Breslav. Mixed-site variance in Kotlin. http://blog.jetbrains.com/kotlin/2013/06/mixed, 2013.

B. Courcelle. On jump-deterministic pushdown automata. *Mathematical Systems Theory*, 1977.

L. Dionne. Boost.Hana. http://boostorg.github.io/hana, 2016.

Y. Gil and T. Levy. Formal language recognition with the Java type checker. In *ECOOP*, 2016.

O. Goldreich. *Computational complexity: a conceptual perspective*. Cambridge University Press, 2008.

J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java language specification, 2015. Java SE 8 Edition.

B. Greenman, F. Muehlboeck, and R. Tate. Getting F-bounded polymorphism into shape. In *PLDI*, 2014.

R. Grigore. Parser generator for fluent interfaces. http://rgrig.appspot.com/javats, 2016.

J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison–Wesley, 1979.

A. J. Kennedy and B. Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 1965.

M. Lange and H. Leiß. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 2009.

X. Leroy. Formal verification of a realistic compiler. *CACM*, 2009.

M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Modules*. PhD thesis, CMU, 1997.

E. Lippert. A simple problem whose decidability is not known. Theoretical Computer Science Stack Exchange, 2013. http://cstheory.stackexchange.com/q/18866.

H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL*, 1990.

M. Odersky. Scaling DOT to Scala – soundness. http://scala-lang.org/blog/2016/02/17/scaling-dot-soundness.html, 2016.

B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 1994.

A. Rossberg. Undecidability of OCaml type checking. Caml mailing list, 1999.

http://caml.inria.fr/pub/old_caml_site/caml-list/1507.html.

M. Sulzmann, G. J. Duck, S. L. P. Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 2007.

R. Tate, A. Leung, and S. Lerner. Taming wildcards in Java's type system. In *PLDI*, 2011.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 1936.

T. L. Veldhuizen. C++ templates are Turing complete. Technical report, Indiana University, 2003.

M. Viroli. On the recursive generation of parametric types. Technical report, University of Bologna, 2000.

K. Wansbrough. Instance declarations are universal. http://www.lochan.org/keith/publications/undec.html (see also https://wiki.haskell.org/Type_SK), 1998.

S. Wehr and P. Thiemann. On the decidability of subtyping with bounded existential types. In *APLAS*, 2009.

S. Weirich. RepLib: a library for derivable type classes. In *Haskell*, 2006.

J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 1999.

Y. Zhang, M. C. Loring, G. Salvaneschi, B. Liskov, and A. C. Myers. Lightweight, flexible object-oriented generics. In *PLDI*, 2015.