Python Type Hints Are Turing Complete

Ori Roth
Department of Computer Science, Technion, Haifa, Israel

Abstract

Grigore proved that Java generics are Turing complete by describing a reduction from Turing machines to Java subtyping. Furthermore, he demonstrated that his "subtyping machines" could have metaprogramming applications if not for their extremely high compilation times. The current work reexamines Grigore's study in the context of another prominent programming language – Python. We show that the undecidable Java fragment used in Grigore's construction is included in Python's type system, making it Turing complete. In contrast to Java, Python type hints are checked by third-party static analyzers and run-time type checkers. The new undecidability result means that both kinds of type checkers cannot fully incorporate Python's type system *and* guarantee termination. The paper includes a survey of infinite subtyping cycles in various type checkers and type reification in different Python distributions. In addition, we present an alternative reduction in which the Turing machines are simulated in real time, resulting in a significantly faster compilation. Our work is accompanied by a Python implementation of both reductions that compiles Turing machines into Python subtyping machines.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases nominal Subtyping with Variance, Python

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.44

Category Pearl/Brave New Idea

Related Version *Previous Version*: `https://doi.org/10.48550/arxiv.2208.14755`

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.1`

Acknowledgements The author would like to thank to Yossi Gil for inspiring them to study the topic of nominal subtyping with variance.

1 Introduction

Python enhancement proposal (PEP) 484 introduced optional type hints to the Python programming language, together with a full-blown gradual type system [16]. Tools such as Mypy [9] use type hints to type-check Python programs. Certain programs, however, cause Mypy to enter an infinite loop (we show an example below). We argue that the reason behind these failures is not a Mypy bug, but a deeper issue in the PEP 484 type system. We use Grigore's reduction from Turing machines (TMs) to nominal subtyping with variance [6] to prove that Python type hints are, in fact, Turing complete. In other words, checking whether a Python program is correctly typed is as hard as the halting problem.

1.1 Nominal Subtyping With Variance

Subtyping is a type system decision problem. Given types t and s, the type system should decide whether type t is a subtype of s, $t <: s$, meaning that every t object is also a member of s. For example, every string is an object, `str` $<:$ `object`, but not every object is a string, `object` $\not<:$ `str`. Subtyping is needed, for example, for checking variable assignments:

```
x: t = ...
y: s = x  # t<:s
```

The second assignment compiles if and only if `t` <: `s`.

Object-oriented languages such as Scala, C#, and Python use *nominal subtyping with variance* [7]. *Nominal* means that subtyping is guided by inheritance. Type `t` is a subtype of type `s` if class `t` is a descendant of `s`:

```
class s: ...
class t(s): ...
x: t = ...
y: s = x  # ✓
```

Variance enables variability in type arguments. By default, type `t[u]` is a subtype of `t[v]` if and only if `u` = `v`. If `t`'s type parameter is covariant, `u` can be any subtype of `v`, `u` <: `v`; if it is contravariant, then `u` :> `v`. In Python, variance is specified in an argument to `TypeVar`, the constructor of type parameters. For example, the following Python program uses contravariance and is correctly typed:

```
x = TypeVar("x", contravariant=True)
class t(Generic[x]): ...  # t has a contravariant type parameter x
x: t[str] = t[object]()  # ✓ (str<:object)
```

Using a reduction from the Post correspondence problem (PCP) [13], Kennedy and Pierce showed that nominal subtyping with variance is undecidable [7]. Their work focused on three subtyping features (characteristics):

1. *Contravariance*: The presence of contravariant type parameters, as described above.
2. *Expansive-recursive inheritance*: The closure of types under the inheritance relation and type decomposition (`t[v]` → `v`) is unbounded. Intuitively, expansive inheritance requires class `t` to recur in one of its supertypes:

   ```
   class t(Generic[x], s[s["t[t[x]]"]]): ...
   # note: forward refrences are put in string literals
   ```

   A formal definition of expansive-recursive inheritance is provided in Section 2.
3. *Multiple instantiation inheritance*: Class `t` is allowed to derive class `s[·]` multiple times using different type arguments:

   ```
   class t(s[object], s[str]): ...  # not legal in Python
   ```

Kennedy and Pierce proved that subtyping becomes decidable when contravariance or expansive inheritance are removed, but they were uncertain about the contribution of multiple instantiation inheritance to undecidability.

1.2 Subtyping Machines

Ten years after them, Grigore showed that Java subtyping is Turing complete using a direct reduction from TMs [6]. This reduction uses a subset of Java that conforms to Kennedy and Pierce's nominal subtyping with variance. Grigore's result settled Kennedy and Pierce's open problem since Java does not support multiple instantiation inheritance [12, §8.1.5]. Intuitively, multiple instantiation inheritance corresponds to non-deterministic subtyping [7, 14], so while it was useful in the PCP reduction, it is redundant in the TM reduction because deterministic TMs are as expressive as non-deterministic TMs.

Grigore's reduction can be described as a function

$$e(M, w) = \Delta \cdot q \tag{1}$$

that encodes TM M and input word w as program $P = \Delta \cdot q$ consisting of class table Δ and subtyping query q. The encoding ensures that TM M accepts w if and only if query q type-checks against Δ.

The idea behind the reduction is to encode the TM configuration (instantaneous description) in the subtyping query q. Recall that the configuration of TM M comprises *(i)* the content of the memory tape, *(ii)* the location of the machine head on the tape, and *(iii)* the current state of the machine's finite control. Figure 1 illustrates the initial configuration of M: The input word $w = a_1a_2 \ldots a_m$ is written on the tape, the machine head points to the first letter a_1, and the current state is the initial state q_I.

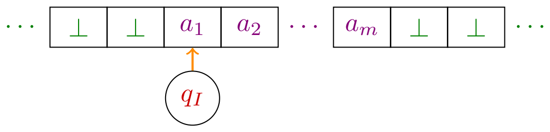


Figure 1 The initial configuration of a Turing machine.

To explain how configurations are encoded as subtyping queries, let us first introduce some syntax (adopted from Grigore's paper). We write a generic type `A[B[C]]` as ABC for short. The use of ◀ instead of <: in a subtyping query means that the type on the left-hand side should be read in reverse (the same goes for ▶ and :>), e.g., ABC ◀ DE is equivalent to $CBA <: DE$.

The initial TM configuration, depicted in Figure 1, is encoded by the following subtyping query:

$$ZEEL_{\perp}NM^{L}NL_{a_1}NL_{a_2}N \cdots NL_{a_m}NL_{\perp}Q_I^{wR} \blacktriangleleft EEZ \tag{2}$$

Observe that the types in Equation (2) have the same colors as the machine configuration elements in Figure 1 that they encode. For example, the type L_{a_1} encodes the tape symbol a_1, and both are purple. As the type on the left-hand side is written in reverse, the content of the encoded tape can be obtained by reading the L types from the left to the right. The full legend in listed in Table 1.

Table 1 The components of Grigore's subtyping machine. All types use a single contravariant type parameter x, except Z, which is monomorphic. The superscripts vary.

Type	Corresponding TM component / purpose
$L_{\perp}x$	an infinite blank tail of the tape
$L_{\sigma}x$	a tape cell containing the symbol σ
$M^{L}x$	the location of the machine head
$Q_I^{wR}x$	the current TM state q_I
Z, Ex, Nx	utility types without corresponding TM components

Grigore referred to the subtyping query in Equation (2) as a *subtyping machine* because when the subtyping algorithm tries to resolve it, it simulates the computation steps of the original TM. The subtyping deduction preserves the general structure of the query, except that it steadily pushes the state type Q_I along the tape. When Q_I reaches the head type M^L, the subtyping algorithm simulates a single TM transition: It overwrites the current tape cell ($L_{a_1} \Rightarrow L_\sigma$), moves the machine head (by repositioning M^L), and changes the machine state ($Q_I \Rightarrow Q_J$). The resulting subtyping query correctly encodes the next configuration in the TM run. This process continues until the machine accepts and the query is resolved, or the machine aborts and a compilation error is raised. If the machine runs indefinitely, the subtyping algorithm does not terminate.

While TMs move the machine head to the left or right freely, subtyping machines can change direction only when reaching the end of the type, EEZ. After simulating a TM transition, the subtyping machine must reach the end(s) of the tape, rotate, and then reach the location of the machine head M in the right orientation, before it can simulate the next transition. In general, Grigore's subtyping machines can make $O(m)$ operations for every computation step of the TM they simulate, where m is the number of symbols on the tape, resulting in a substantial slowdown. For example, Grigore's simulation of the CYK algorithm, which usually runs in $O(n^3)$, takes $O(n^9)$ subtyping deduction steps to be completed.

1.3 Subtyping Metaprogramming

Beyond the undecidability result, Grigore demonstrated metaprogramming applications for their subtyping machines. Although the computational power of subtyping machines is unlimited, harnessing this power for conducting meaningful type-level metaprogramming as done in, e.g., C++ [18], is not easy. To integrate subtyping machines into programs, Grigore proposed to wrap them in fluent APIs [2]. Fluent API methods are called in a stream (chain) of consecutive invocations, as demonstrated in Listing 1.

Listing 1 Running a subtyping machine with a fluent API.

```
p: Palindrome = a().b().b().a().b().b().a()
```

In Grigore's design, the fluent chain produces the tape of the subtyping machine. For example, the chain in Listing 1 produces a types tape containing the input word *abbabba*. We run the subtyping machine by assigning the chain to a variable, invoking a the subtyping query in Equation (2).

The purpose of the subtyping machine is to validate a property of the chain at compile time. For example, a subtyping machine that recognizes palindromes forces the fluent chain to encode a palindrome word, or else it would not compile. Since the chain in Listing 1 encodes a palindrome, the subtyping machine accepts and the assignment type-checks. This sort of type-level metaprogramming in fluent APIs is employed for embedding domain-specific languages (DSLs) and enforcing higher-level API protocols [3, 10, 4, 19, 14]. In theory, Grigore's subtyping machines can encode any computable DSL or API protocol. Unfortunately, the slowdown ingrained into the design of the subtyping machines results in extremely high compilation times, making the technique impractical.

1.4 Contributions

This work revisits Grigore's study of subtyping machines in the context of Python. We show that Python's type system includes the Java fragment used in Grigore's construction and is, therefore, Turing complete. We review the impact of undecidable subtyping on the Python ecosystem, covering compile-time and run-type type checkers and different Python distributions. Finally, we present a new subtyping machine design that avoids the inherent slowdown imposed by Grigore's construction – an essential step towards practical subtyping machine applications. Our subtyping machines simulate TMs in real time, i.e., make $O(1)$ operations for each TM transition. The paper is accompanied by a Python implementation of Grigore's and our reductions, producing subtyping machines on top of Python type hints.

Outline

The rest of the paper is organized as follows. In Section 2, we show that Python type hints are Turing complete and discuss possible ways to make them more tractable. Section 3 includes the survey of infinite subtyping and type reification in Python. Section 4 introduces our alternative design for subtyping machines that simulate TMs in real time. Section 5 presents our Python implementations of Grigore's original reduction and the new reduction, and compares their performances. Section 6 concludes.

2 Python Subtyping is Undecidable

Type hints were introduced into the Python programming language with PEP 484 [16]. PEP 484 defines the syntax of type hints but only provides an informal description of their semantics, referring the reader to the supplementary PEP 483 [17] for an in-depth discussion. Type hints are used as annotations and are entirely optional:

```python
def positive(x: int) -> bool:
    return x > 0
```

Static analysis on type hints is not performed by the Python interpreter but by third-party tools. For instance, Mypy [9] is a type checker for Python type hints; in fact, PEP 484 was originally inspired by Mypy [16].

The type system described in PEP 484 supports declaration-site nominal subtyping with variance, similar to the abstract type system studied by Kennedy and Pierce [7]. Although originally designed for Java, Grigore's subtyping machines conform to Kennedy and Pierce's type system. To implement subtyping machines with Python type hints, we need to show that Python's type system includes the two subtyping features essential for Grigore's construction: contravariance and expansive-recursive inheritance.

In Python, type variables are specified using a special constructor, `TypeVar`. Making a contravariant (or covariant) type variable is as simple as passing an argument to `TypeVar`:

```python
z = TypeVar("z", contravariant=True)
class N(Generic[z]): ...  # class N has a contravariant parameter z
```

Expansive-recursive inheritance is a more elusive aspect of nominal subtyping. Kennedy and Pierce [7] defined expansive inheritance using the *inheritance and decomposition closure* $\mathsf{cl}(t)$ of type t. Here we provide a brief description of the closure; the full definition can be found in Kennedy and Pierce's paper. Recall that we use a shorthand notation for generic types, $Cs = \texttt{C[}s\texttt{]}$. If $\mathsf{cl}(t)$ contains the type Cs, then by decomposition it also contains s:

$$\text{(DECOMPOSITION)}\quad \frac{Cs \in \mathsf{cl}(t)}{s \in \mathsf{cl}(t)} \tag{3}$$

If, in addition, class `C[x]` inherits from type u, denoted $Cx : u$, then $\mathsf{cl}(t)$ also contains the type $u[x \leftarrow s]$, in which every occurrence of type parameter x is substituted by s:

$$\text{(INHERITANCE)}\quad \frac{Cs \in \mathsf{cl}(t) \quad Cx : u}{u[x \leftarrow s] \in \mathsf{cl}(t)} \tag{4}$$

Kennedy and Pierce proved that a class table is expansive-recursive if and only if the set $\mathsf{cl}(t)$ is infinite for some type t. For example, consider the following class declaration:

```python
x = TypeVar("x")
class C(Generic[x], N[N["C[C[x]]"]]): ...  # Cx:NNCCx
```

The inheritance of class `C` is expansive-recursive since the set $\mathsf{cl}(Ct)$ is infinite for any type t:

$$\begin{array}{lr}
1.\ Ct \in \mathsf{cl}(Ct) & \\
2.\ NNCCt \in \mathsf{cl}(Ct) & (\textsc{Inheritance}) \\
3.\ NCCt \in \mathsf{cl}(Ct) & (\textsc{Decomposition}) \\
4.\ CCt \in \mathsf{cl}(Ct) & (\textsc{Decomposition}) \\
5.\ NNCCCt \in \mathsf{cl}(Ct) & (\textsc{Inheritance}) \\
\text{and so on\ldots} &
\end{array} \tag{5}$$

Between steps 1 and 4 the type Ct was transformed to CCt, increasing the size of the type by a single C. By continuing the deduction we get that the set $\mathsf{cl}(Ct)$ contains the type $C^n t$ for any $n \geq 1$ and is, therefore, infinite.

As PEP 484 and the accompanying PEP 483 do not mention any restrictions on expansive-recursive inheritance (at least, that the author could find), we conclude that Python implicitly supports expansive inheritance. As evidence, the example above correctly compiles in Python and Mypy.

By enabling both contravariance and expansive-recursive inheritance, the designers of PEP 484 opened up Python type hints to the same pitfalls of nominal subtyping with variance studied by Kennedy, Pierce, and Grigore. For example, the code in Listing 2, adapted from Kennedy and Pierce [7], shows how contravariance and expansive inheritance can be combined to induce an infinite subtyping cycle.

Listing 2 Contravariance, expansive inheritance, and infinite subtyping with Python type hints.

```python
from typing import TypeVar, Generic, Any
z = TypeVar("z", contravariant=True)
class N(Generic[z]): ...
x = TypeVar("x")
class C(Generic[x], N[N["C[C[x]]"]]): ...
class T: ...
class U: ...
_: N[C[U]] = C[T]() # CT <: NCU ✗ infinite subtyping
```

The last line of Listing 2 contains a variable assignment that invokes the subtyping query $CT <: NCU$. The query is resolved using two subtyping rules [7]: Super, allowing us to replace a type with its supertype using an inheritance rule:

$$(\textsc{Super})\quad \frac{Cx : u \quad u[x \leftarrow s] <: t}{Cs <: t} \tag{6}$$

And Var, allowing us to remove a single type from both sides of the query:

$$(\textsc{Var})\quad \frac{C\text{'s type parameter } x \text{ is contravariant} \quad t <: s}{Cs <: Ct} \tag{7}$$

We use the rules Super and Var to resolve the query as follows:

$$\begin{array}{lr}
1.\ CT <: NCU & \\
2.\ NNCCT <: NCU & (\textsc{Super}) \\
3.\ CU <: NCCT & (\textsc{Var}) \\
4.\ NNCCU <: NCCT & (\textsc{Super}) \\
5.\ CCT <: NCCU & (\textsc{Var}) \\
\text{and so on\ldots} &
\end{array} \tag{8}$$

Between steps 1 and 5, the subtyping query was not reduced but increased in size. This deductive process continues indefinitely. When Mypy checks this program, it throws a segmentation fault. In Section 5, we show that Mypy crashes because it gets stuck in an infinite recursion during the subtyping algorithm.

Since contravariance and expansive inheritance are enough to implement Grigore's subtyping machines, we get that Python type hints are Turing complete. Section 5 presents our implementation of Grigore's reduction with Python type hints. The resulting subtyping machines correctly run with Mypy.

2.1 Taming Python's Type System

The source of undecidability in PEP 484 is the hazardous combination of contravariance and expansive-recursive inheritance, whose presence enables the construction of Grigore's subtyping machines. Kennedy and Pierce proved that nominal subtyping with variance becomes decidable once either contravariance or expansive inheritance are removed [7]. Restrictions to expansive inheritance are implemented, for example, in the Scala programming language [11, §5.1.5] and the .NET framework [1, §II.9.2]. Removing expansive inheritance, however, might not be enough to make Python type hints decidable. A more thorough inspection of the various features of Python's type system is required to determine this.

Greenman et al. [5] and Tate et al. [15] presented alternative subtyping algorithms for Java that are decidable. Besides the theoretical work, they surveyed extensive corpora of Java projects to show that their new algorithms do not break existing code and are thus backward compatible. Future attempts at making Python's type hints decidable may follow a similar line.

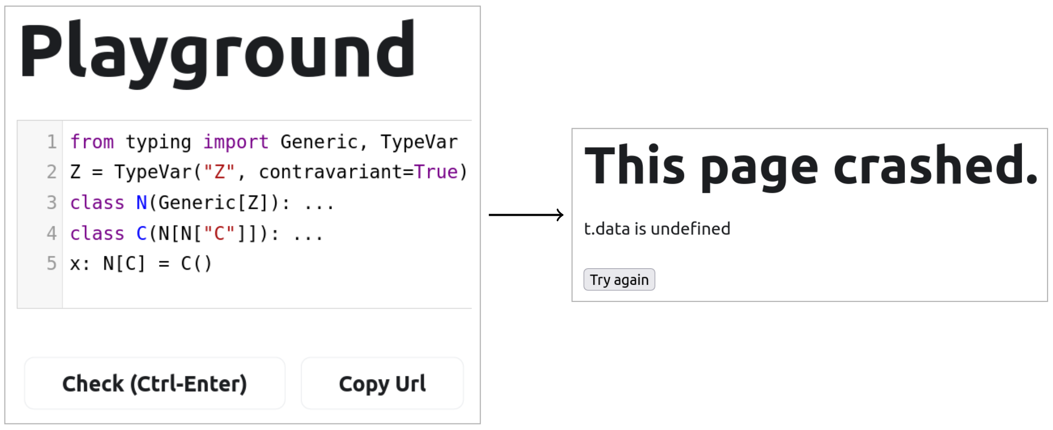
3 Type Reification and Infinite Subtyping in the Wild

A remote code execution vulnerability allows an attacker to run arbitrary *code* on a remote machine. Python type hints, on the other hand, can be used to run arbitrary *computations*. The distinction between code and computation is critical in the context of security: we can use type hints to compute prime numbers but not to access the file system or to open an SSH connection. Nevertheless, the fact that Python type hints are Turing complete does raise a few practical concerns. Type hints can induce infinite subtyping cycles, as demonstrated in Listing 2. These cycles cannot be detected by a "smart" type checker due to the undecidability of the halting problem. Thus, we can use infinite subtyping to attack Python type checkers and cause them to loop indefinitely or crash. For example, we managed to use infinite subtyping to crash the online playground of the Pyre Python type checker, as depicted in Figure 2. The playground in available at `https://pyre-check.org/play/`.

In Java, the compiler eliminates type arguments to generic types in a process called *type erasure*, making infinite subtyping a compile-time-only issue. Although PEP 484 mentions type erasure, it does not explicitly prohibit run-time type reification. The later PEP 585 [8] settles the matter, stating:

> *"The generic parameters are not preserved in instances created with parameterized types, in other words generic types erase type parameters during object creation."*

In practice, Python implementations *ignore* type erasure and keep records of generic types in run-time objects. Listing 3 demonstrates how to obtain information about type parameters and generic arguments at run time.



```python
from typing import Generic, TypeVar
Z = TypeVar("Z", contravariant=True)
class N(Generic[Z]): ...
class C(N[N["C"]]): ...
x: N[C] = C()
```

This page crashed.

t.data is undefined

Figure 2 Crashing the Pyre playground with an infinite subtyping cycle.

Listing 3 Obtaining reified generic types in Python at run time.

```python
from typing import TypeVar, Generic
x = TypeVar("x", contravariant=True)
class C(Generic[x]): ...
print(C.__parameters__)  # (-x,)
y = C[int]()
print(y.__orig_class__)  # __main__.C[int]
```

The field `__parameters__` of the generic class `C` reveals that the class uses a single contravariant type parameter `x`. The generic type of the parameterized instantiation `C[int]()` is retrieved using the `__orig_class__` field. We tested the code in Listing 3 on several Python distributions. The results are presented in Table 2.

Table 2 Type reification in different Python distributions.

Python Distribution	Version	Type Reification	Notes
CPython	3.9.13	●	
PyPy	7.3.11	●	
RustPython	3.5.0alpha	●	
WinPython	3.0.20	●	
Brython	3.11.1	◐	`__orig_class__` is bugged

Table 2 shows that all the Python implementations tested practice type reification. As an exception, Brython fails to retrieve the field `__orig_class__`. After inspecting the source code, the author believes it to be a bug. At the moment of writing these lines, an open issue[1] on CPython's GitHub project calls for a proper API wrapper for the `__orig_class__` field. This development indicates that the Python community sees type reification not as a mere implementation detail but as a desired feature that should be established further.

Although Python reifies generic types, it does not mean that run-time type checkers use them in subtyping checks. And while static analyzers have all the type hints available, there is no guarantee that they implement the complete Python type system described in

[1] https://github.com/python/cpython/issues/101688

PEP 484. Whether or not a type checker gets stuck on infinite subtyping or successfully runs a subtyping machine is determined by its support of type parameter variance. In particular, both cases depend on contravariant type parameters. We reviewed various static and dynamic Python type checkers and checked whether they support variance as described in PEP 484. The results are presented in Table 3.

Table 3 Variance support in Python type checkers.

Type Checker	Version	Typing Discipline	Variance Support	Notes
Mypy	0.991	static	●	
Pyre	0.9.17	static	●	
Pyright	1.1.279	static	◐	Pyright is unsound
Pytype	2022.11.10	static	○	
Pyanalyze	0.8.0	static	○	
Pydantic	1.10.2	dynamic	○	
Typeguard	2.13.3	dynamic	○	
Pytypes	1.0b10	dynamic	○	Delegates to `isinstance` (sometimes)
Typical	2.8.0	dynamic	○	Delegates to `isinstance`

We found that Mypy and Pyre are the only type checkers to fully support subtyping with variance. Pyright also acknowledges variant type parameters, but reports errors for correctly-typed programs, meaning that it is unsound. We showed this by running one of our (accepting) subtyping machines: Mypy and Pyre reported no errors while Pyright did report one. The code is found in the supplementary material. Pytype reported that it does not support contravariant parameters. For the rest of the type checkers, we had to search the source code for any mention of variance to conclude that they do not support it. Typical seems to delegate subtyping checks to `isinstance` or equivalent methods that reject generic types altogether. We observed similar behavior in Pytypes when using type forward references, i.e., types in string literals.

To fully support PEP 484, Python type checkers must support covariant and contravariant type parameters. Variance, however, introduces a security vulnerability in the form of unavoidable infinite subtyping cycles. Crashes of static type checkers may be forgiven, but for dynamic checkers, infinite subtyping poses a major concern. This also holds for Python's `isinstance` check, if the designers of Python or one of its implementations ever consider adding run-time subtyping checks against generic types.

4 Real-Time Subtyping Machines

Grigore's subtyping machines must scan the entire tape memory before they can simulate a single TM transition. This is because the subtyping machines can change their direction (from ◀ to ▶ and vice versa) only when reaching the end of the tape. We now present an alternative design for subtyping machines that simulate TMs in real time, i.e., where a single TM computation step is simulated by $O(1)$ subtyping deductions.

Let $M = \langle Q, \Sigma, q_I, q_h, \delta \rangle$ be a TM where Q is the set of machine states, Σ is the set of tape symbols, q_I is the initial state, q_h is the termination state, and

$$\delta : Q \times (\Sigma \cup \{\bot\}) \to Q \times \Sigma \times \{L, R\}$$

is the transition function. In each computation step, the TM changes its state, overwrites the current tape cell, and moves the machine head to the left (L) or right (R). The TM accepts its input if and only if it reaches the termination state q_h. We use the symbol $\bot \notin \Sigma$ to denote blank tape cells.

The new encoding of subtyping machines is shown in Table 4. The encoding comprises ten inheritance rules *(i)* to *(x)* (recall that we use a colon to denote inheritance). To encode a TM M as a subtyping machine, fill in the inheritance rules' missing values using elements from M that satisfy the conditions on the right-hand side. For example, if M contains state $q_4 \in Q$ and tape symbol $p \in \Sigma$, then by rule *(v)*,

$$Q_4^{LL}x : L_p N Q_4^L L_p N x.$$

When multiple rules apply to the same type, multiple inheritance is used. Symbol x is a contravariant type parameter and symbol ? denotes the wildcard type, i.e., the type that is consistent with (can be substituted by) any type. In Python, this is the `Any` type [16]. All the type parameters used in the encoding are contravariant. There are ten more inheritance rules in addition to those in Table 4, obtained by swapping L and R in each rule in the table.

Table 4 Real-time subtyping machines. For each inheritance rule, swap L and R to get the symmetrical rule. The type parameter `x` is contravariant.

(i)	$Q_s^L x : L_a N Q_{s'}^L L_b N x$	$\delta(q_s, a) = \langle q_{s'}, b, L\rangle$
(ii)	$Q_s^L x : L_a Q_{s'}^{LRR} N L_b N x$	$\delta(q_s, a) = \langle q_{s'}, b, R\rangle$
(iii)	$Q_s^L x : L_\perp Q_{s'}^{L\perp L} N L_b N x$	$\delta(q_s, \perp) = \langle q_{s'}, b, L\rangle$
(iv)	$Q_s^L x : L_\perp Q_{s'}^{L\perp R} N L_b N x$	$\delta(q_s, \perp) = \langle q_{s'}, b, R\rangle$
(v)	$Q_s^{LL} x : L_a N Q_s^L L_a N x$	$\forall q_s \in Q, \forall a \in \Sigma$
(vi)	$N x : Q_s^{LRR} N Q_s^{RR} x$	$\forall q_s \in Q$
(vii)	$N x : Q_s^{L\perp L} Q_s^{RL} N L_\perp N x$	$\forall q_s \in Q$
(viii)	$N x : Q_s^{L\perp R} N Q_s^{RR} L_\perp N x$	$\forall q_s \in Q$
(ix)	$N x : Q_s^{LR} N Q_s^R x$	$\forall q_s \in Q$
(x)	$Q_h^L x : L_a ?$	$\forall a \in \Sigma \cup \{\perp\}$

The roles of the types in Table 4 are mostly the same as in Grigore's encoding (Table 1), i.e., L_a is a tape cell containing the symbol a, N is a buffer type, and Q_s is a state type. The new encoding, however, does not use a type for the machine head (M in Grigore's encoding) because the state type Q also indicates the location of the machine head. The superscripts of Q imply the head's movement direction; e.g., Q_s^{LL} means that the head is about to go two cells to the left, $Q_s^{L\perp R}$ means that the head is about to move left into a blank cell and then rotate, and so on.

The initial TM configuration is encoded by the following subtyping query:

$$Z N L_\perp Q_I^R \blacktriangleleft L_{a_1} N L_{a_2} N \cdots L_{a_{m-1}} N L_{a_m} N L_\perp N Z \tag{9}$$

The content of the tape is encoded by the L types, read from the left to the right. The current state and the position of the machine head are encoded by the type Q_I^R – the current cell is on the right (R) of the type, which is also the direction of the query ($\blacktriangleleft$). The infinite blank ends of the tape are encoded by the type $L_\perp$. Observe that the colors of the types in Equation (9) match the colors of the corresponding TM components in Figure 1. Type Q_s^R is half red and half orange because it represents both the current state *and* the head location.

To prove the correctness of the simulation, we show that the subtyping query in Equation (9) simulates the TM transitions while preserving the encoding of the machine tape, head, and state comprising the TM configuration. There are three variables to be considered:

the initial orientation of the head, whether or not the current cell is blank, and whether or not the machine head changes direction. For example, the machine head Q_I^R in Equation (9) points to the right (R) and reads a non-blank cell L_{a_1}. The next cell could be either L_{a_2}, if the head continues right, or blank $L_\perp$, if it rotates.

We now cover four cases, assuming that the initial orientation is *left* (this is the orientation used in Table 4). The other four cases are symmetrical. Next to each subtyping deduction step, we mention the subtyping rule used in this step. Recall that there are two subtyping rules, Super and Var (Equations (6) and (7)). As all the type parameters are contravariant, the query changes direction (from ◀ to ▶ and vice versa) after applying Var a single time.

Case I. The head points to a *non-blank* cell a, replaces it with symbol b, and continues *left*. The relevant TM transition is

$$\delta(q_s, a) = \langle q_{s'}, b, L\rangle$$

and the resulting subtyping query is

$$\begin{array}{ll}
\cdots N L_a \blacktriangleright Q_s^L \cdots & \\
\cdots N L_a \blacktriangleright L_a N Q_{s'}^L L_b N \cdots & \textit{(i)} + (\textsc{Super}) \\
\cdots \blacktriangleright Q_{s'}^L L_b N \cdots & (\textsc{Var}) \times 2
\end{array}$$

"*(i)*+Super" means applying rule Super with inheritance rule *(i)* from Table 4. "(Var)×2" means applying rule Var twice. Observe that the subtyping query simulates the TM transition while preserving the encoding: symbol L_a is replaced by L_b, state Q_s is replaced by $Q_{s'}$ (the next machine state), and the head moves to the cell on the left.

Case II. The head points to a *non-blank* cell a, replaces it with symbol b, and continues *right*. The relevant TM transition is

$$\delta(q_s, a) = \langle q_{s'}, b, R\rangle$$

and the resulting subtyping query is

$$\begin{array}{ll}
\cdots N L_a \blacktriangleright Q_s^L \cdots & \\
\cdots N L_a \blacktriangleright L_a Q_{s'}^{LRR} N L_b N \cdots & \textit{(ii)} + (\textsc{Super}) \\
\cdots N \blacktriangleleft Q_{s'}^{LRR} N L_b N \cdots & (\textsc{Var}) \\
\cdots Q_{s'}^{RR} N Q_{s'}^{LRR} \blacktriangleleft Q_{s'}^{LRR} N L_b N \cdots & \textit{(vi)} \ + (\textsc{Super}) \\
\cdots Q_{s'}^{RR} \blacktriangleleft L_b N \cdots & (\textsc{Var}) \times 2 \\
\cdots N L_b Q_{s'}^R N L_b \blacktriangleleft L_b N \cdots & \textit{(v)} \ + (\textsc{Super}) \\
\cdots N L_b Q_{s'}^R \blacktriangleleft \cdots & (\textsc{Var}) \times 2
\end{array}$$

Case III. The head points to a *blank* cell (the end of the tape), replaces it with symbol b, and continues *left*. The relevant TM transition is

$$\delta(q_s, \perp) = \langle q_{s'}, b, L\rangle$$

and the resulting subtyping query is

$$\begin{array}{ll}
Z N L_\perp \blacktriangleright Q_s^L \cdots & \\
Z N L_\perp \blacktriangleright L_\perp Q_{s'}^{L\perp L} N L_b N \cdots & \textit{(iii)} \ + (\textsc{Super}) \\
Z N \blacktriangleleft Q_{s'}^{L\perp L} N L_b N \cdots & (\textsc{Var}) \\
Z N L_\perp N Q_{s'}^{RL} Q_{s'}^{L\perp L} \blacktriangleleft Q_{s'}^{L\perp L} N L_b N \cdots & \textit{(vii)} \ + (\textsc{Super}) \\
Z N L_\perp N Q_{s'}^{RL} \blacktriangleright N L_b N \cdots & (\textsc{Var}) \\
Z N L_\perp N Q_{s'}^{RL} \blacktriangleright Q_{s'}^{RL} N Q_{s'}^L L_b N \cdots & \textit{(ix)} \ + (\textsc{Super}) \\
Z N L_\perp \blacktriangleright Q_{s'}^L L_b N \cdots & (\textsc{Var}) \times 2
\end{array}$$

Case IV. The head points to a *blank* cell, replaces it with symbol b, and continues *right*. The relevant TM transition is

$$\delta(q_s, \bot) = \langle q_{s'}, b, R\rangle$$

and the resulting subtyping query is

$$\begin{array}{ll}
ZNL_\bot \blacktriangleright Q^L_s \cdots & \\
ZNL_\bot \blacktriangleright L_\bot Q^{L\bot R}_{s'} NL_b N \cdots & \textit{(iv)} + (\textsc{Super}) \\
ZN \blacktriangleleft Q^{L\bot R}_{s'} NL_b N \cdots & (\textsc{Var}) \\
ZNL_\bot Q^{RR}_{s'} NQ^{L\bot R}_{s'} \blacktriangleleft Q^{L\bot R}_{s'} NL_b N \cdots & \textit{(viii)} + (\textsc{Super}) \\
ZNL_\bot Q^{RR}_{s'} \blacktriangleleft L_b N \cdots & (\textsc{Var}) \times 2 \\
ZNL_\bot NL_b Q^R_{s'} NL_b \blacktriangleleft L_b N \cdots & \textit{(v)} + (\textsc{Super}) \\
ZNL_\bot NL_b Q^R_{s'} \blacktriangleleft \cdots & (\textsc{Var}) \times 2
\end{array}$$

The TM rejects its input when its current state is q_s, the current tape symbol is a, and the transition $\delta(q_s, a)$ is not defined. In this case, the subtyping query $L_a \blacktriangleright Q^L_s$ also rejects since there is no inheritance rule in Table 4 with which rule Super can be applied. On the other hand, if the TM reaches state q_h and accepts its input, the subtyping query $L_a \blacktriangleright Q^L_h$ is resolved by applying rule *(x)*.

Note that our simulation is clearly real-time. To simulate a single TM transition, the subtyping machine performs at most eight subtyping deductions (in cases II and IV).

The wildcard type used in rule *(x)* is not a part of Kennedy and Pierce's system of nominal subtyping with variance. Instead of using the wildcard, the subtyping machine could go to either side of the tape before resolving the query, as done in Grigore's simulation. This makes the subtyping machine design a bit more complicated, and its simulation of the TM returns to be non-real-time, but the computational complexity of the simulation is not increased.

5 Implementation and Performance Experiment

We present our Python implementation of Grigore's original reduction and our new real-time simulation introduced in Section 4 in the supplementary material. Our implementation compiles TMs into Python subtyping machines that use the type hints and generics described in PEP 484 [16]. Each subtyping machine comprises a class table (Table 4) and a variable assignment that invokes a subtyping query (Equation (9)). To run the subtyping machine, we use Mypy to type-check the generated Python code.

If the subtyping machine accepts its input, Mypy terminates successfully, and if the input is rejected, Mypy reports a typing error. But what happens when the subtyping machine runs indefinitely? When running Mypy on the code in Listing 2, containing an infinite subtyping cycle, Mypy crashes with a segmentation fault. To uncover the reason for the segmentation fault, we remove a call to `sys.setrecursionlimit` from Mypy's source code and run it again with the flag `--show-traceback`. Mypy reports the following error:

```
...
File "mypy/types.py", line 1283, in accept
File "mypy/subtypes.py", line 585, in visit_instance
File "mypy/subtypes.py", line 345, in check_type_parameter
File "mypy/subtypes.py", line 339, in check
File "mypy/subtypes.py", line 179, in is_subtype
File "mypy/subtypes.py", line 329, in _is_subtype
```

```
  File "mypy/types.py", line 1283, in accept
  File "mypy/subtypes.py", line 585, in visit_instance
  File "mypy/subtypes.py", line 345, in check_type_parameter
  File "mypy/subtypes.py", line 339, in check
  File "mypy/subtypes.py", line 147, in is_subtype
RecursionError: maximum recursion depth exceeded while calling a Python object
```

The stack trace confirms that Mypy's subtyping algorithm is implemented using recursion, causing it to crash with a segmentation fault on infinite subtyping. This observation makes it possible to measure the run time of the subtyping machine, i.e., the number of subtyping deductions it performs, by calculating the minimal size of the call stack required for Mypy to type-check the machine.

Figure 3 describes the results of our experiment, in which we measured the run times of subtyping machines accepting input words of various lengths. The TM used in the experiment recognizes palindromes over $\{a, b\}$ and runs in $O(n^2)$. We compiled the TM together with random palindromes of increasing lengths into subtyping machines, once using Grigore's method and once with our construction. Then, binary search was used to find the minimal call stack size (in megabytes (MB)) required for Mypy to type-check the machine without getting a segmentation fault.

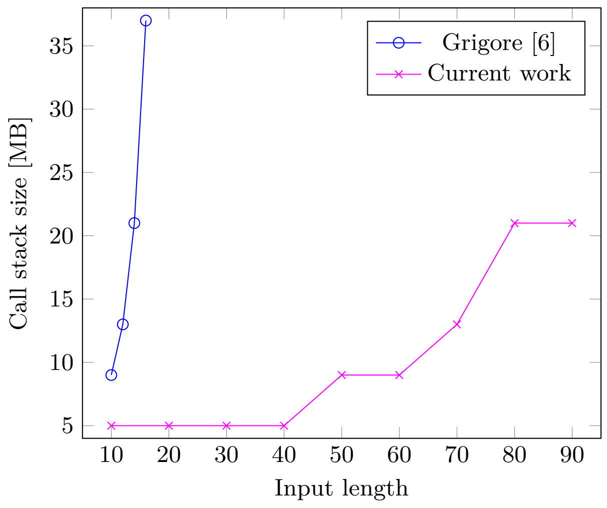


Figure 3 Run times of the palindrome subtyping machines: Grigore's reduction vs. the new reduction.

In theory, Grigore's subtyping machines should run in $O(n^3)$ due to their inherent slowdown, while our machines are expected to run in $O(n^2)$ since they simulate the palindromes TM in real time. In practice, we see that our subtyping machines are much faster than Grigore's and require significantly fewer resources to be type-checked.

6 Conclusions

Python type hints are Turing complete because PEP 484 supports nominal subtyping with variance, including contravariance and expansive-recursive inheritance. These two subtyping features are sufficient to implement Grigore's subtyping machines that simulate TMs at compile time.

We demonstrated that infinite subtyping cause Python type checkers to crash with a stack overflow error. Due to the undecidability of the halting problem, fixing this problem requires changing the Python type system described in PEP 484. Even run-time type checkers are in danger because existing Python distributions reify generic types – in spite of the language specifications. In practice, we found that only two static analyzers (Mypy and Pyre) are vulnerable to infinite subtyping since they provide the most complete implementations of Python's type system.

We described an alternative subtyping machine design that simulates TMs in real time, removing the inherent slowdown introduced in Grigore's original design. Our experiment shows that the new subtyping machines compile significantly faster. Our design is an essential step towards practical subtyping machine applications. Nevertheless, such applications would most likely depend on the type checker's implementation of the subtyping algorithm – specifically, that it is not recursive.

References

1 ECMA International. *ECMA Standard 335: Common Language Infrastructure*, 3 edition, June 2005. Available at `http://www.ecma-international.org/publications/standards/Ecma-335.htm` (accessed Aug. 2022).

2 Martin Fowler. Fluentinterface, 2005. URL: `https://www.martinfowler.com/bliki/FluentInterface.html`.

3 Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th Europ. Conf OO Prog. (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Inf. (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2016.10`.

4 Yossi Gil and Ori Roth. Fling – A fluent API generator. In Alastair F. Donaldson, editor, *33rd Europ. Conf OO Prog. (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Inf. (LIPIcs)*, pages 13:1–13:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2019.13`.

5 Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting f-bounded polymorphism into shape. *SIGPLAN Not.*, 49(6):89–99, June 2014. `doi:10.1145/2666356.2594308`.

6 Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3009837.3009871`.

7 Andrew Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *Int. Work. Found. & Devel. OO Lang.*, FOOL/WOOD'07, Nice, France, January 2007. ACM. URL: `http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html`.

8 Łukasz Langa. *PEP 585 – Type Hinting Generics In Standard Collections.* Available at `https://peps.python.org/pep-0585/` (accessed Nov. 2022).

9 Jukka Lehtosalo, Guido van Rossum, Ivan Levkivskyi, and Michael J. Sullivan. *mypy.* Available at `http://mypy-lang.org/` (accessed Aug. 2022).

10 Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: a fluent API generator. In *Proc. 16th ACM SIGPLAN Int. Conf Generative Prog.*, GPCE'17, pages 199–211, Vancouver, BC, Canada, 2017. ACM.

11 Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. Scala language specification version 2.13. Available on: `https://scala-lang.org/files/archive/spec/2.13/` (accessed Aug. 2022).

12 Oracle. *The Java Language Specification, Java SE 8 Edition*, February 2015. Available at `https://docs.oracle.com/javase/specs/`.

13 Emil Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

14 Ori Roth. Study of the subtyping machine of nominal subtyping with variance. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. `doi:10.1145/3485514`.

15 Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in java's type system. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 614–627, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1993498.1993570`.

16 Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints.* Available at `https://peps.python.org/pep-0484/` (accessed Nov. 2022).

17 Guido van Rossum and Ivan Levkivskyi. *PEP 483 – The Theory of Type Hints.* Available at `https://peps.python.org/pep-0483/` (accessed Nov. 2022).

18 Todd Veldhuizen. *Using C++ Template Metaprograms*, pages 459–473. SIGS Publications, Inc., USA, 1996.

19 Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. Generating a fluent api with syntax checking from an LR grammar. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. `doi:10.1145/3360560`.