Lieber ein Typparameter zu viel als einer zu wenig

Lars Hupel
JUG Ostfalen
2019-07-18

INNOQ

Fazit

1. Type Erasure ist nicht schlecht, sondern gut.
2. Es gibt zu wenig Type Erasure.
3. Neue Typparameter braucht das Land.
4. In Scala ist alles besser.

Fazit

1. Type Erasure ist nicht schlecht, sondern gut.
2. Es gibt zu wenig Type Erasure.
3. Neue Typparameter braucht das Land.
4. In Kotlin ist vieles besser.

Die fünf Phasen der Trauer über Type Reification

Verweigerung

Java Generics

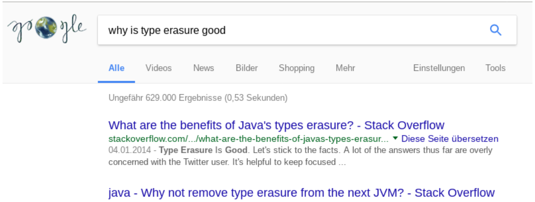
- seit Java 5 (09/2004)
- `List<String>` statt `List`
- `list.get(0)` statt `(String) list.get(0)`
- Syntax inspiriert von C++
- aber: komplett anders implementiert!


Making Java
Making Java easier to type and easier to type
Making Java easier to type and easier to type
Making Java easier to type and easier to type


GJ
GJ
GJ
GJ

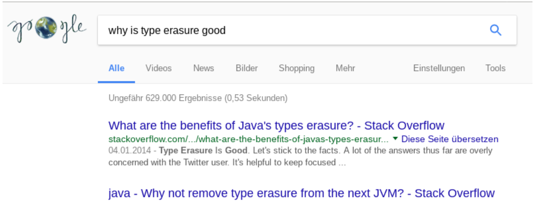
Wadler Odersky Bracha Stoutamire

Type Erasure

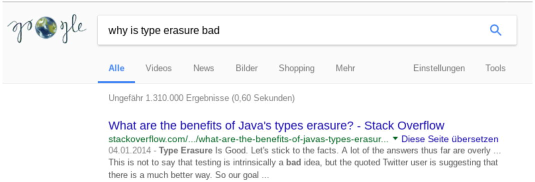


≈ 629k

Type Erasure



≈ 629k



≈ 1310k

Warum ist Type Erasure so unbeliebt?

Fast jede Java-Programmiererin ist schon einmal darüber gestolpert …

Arrays of Wisdom of the Ancients

Collection.toArray(new T[0]) or Collection.toArray(new T[size]), that's the question

Table of Contents

Aleksey Shipilёv, @shipilev, aleksey@shipilev.net

This post is also available in ePUB and mobi.

Thanks to Claes Redestad, Brian Goetz, Ilya Teterin, Yurii Lahodiuk, Gleb Smirnov, Tim Ellison, Stuart Marks, Marshall Pierce, Fabian Lange and others for reviews and helpful suggestions!

Arrays of Wisdom of the Ancients

Collection.toArray(new T[0]) or Collection.toArray(new T[size]), that's the question

Collection.toArray(new T[0]) or Collection.toArray(new T[size]), that's the question

Table of Contents

Aleksey Shipilëv, @shipilev, aleksey@shipilev.net

This post is also available in ePUB and mobi.

Thanks to Claes Redestad, Brian Goetz, Ilya Teterin, Yurii Lahodiuk, Gleb Smirnov, Tim Ellison, Stuart Marks, Marshall Pierce, Fabian Lange and others for reviews and helpful suggestions!

Arrays of Wisdom of the Ancients

Collection.toArray(new T[0]) or Collection.toArray(new T[size]), that's the question

Collection.toArray(new T[0]) or Collection.toArray(new T[size]), that's the question

Table of Contents

… 36 Seiten

This post is also available in ePUB and mobi.

Thanks to Claes Redestad, Brian Goetz, Ilya Teterin, Yurii Lahodiuk, Gleb Smirnov, Tim Ellison, Stuart Marks, Marshall Pierce, Fabian Lange and others for reviews and helpful suggestions!

Begriffsklärung

> *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."*
>
> – Benjamin Pierce

Begriffsklärung

> *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."*

– Benjamin Pierce

Begriffsklärung

> *“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”*
>
> – Benjamin Pierce

Begriffsklärung

> “*A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*”

– Benjamin Pierce

Begriffsklärung

> "*A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*"

– Benjamin Pierce

Folgerungen

1. „Laufzeittyp" ist ein Widerspruch in sich

Folgerungen

1. „Laufzeittyp" ist ein Widerspruch in sich
2. Der Compiler verwirft Information

Folgerungen

1. „Laufzeittyp" ist ein Widerspruch in sich
2. Der Compiler verwirft Information
3. Typdispatching ist zum Scheitern verurteilt

Zorn

Ko- und Kontravarianz

Java

```java
List<Dog> goodDogs = new List<Dog>();
List<? extends Animal> goodAnimals = goodDogs;
```

Ko- und Kontravarianz

Java

```java
List<Dog> goodDogs = new List<Dog>();
List<? extends Animal> goodAnimals = goodDogs;
```

Scala

```scala
val goodDogs: List[Dog] = List.empty
val goodAnimals: List[Animal] = goodDogs
```

Ko- und Kontravarianz

Java (use site)

```java
List<Dog> goodDogs = new List<Dog>();
List<? extends Animal> goodAnimals = goodDogs;
```

Scala (declaration site)

```scala
val goodDogs: List[Dog] = List.empty
val goodAnimals: List[Animal] = goodDogs
```

Ko- und Kontravarianz

Reifizierte Generics führen zu zwei Problemen:

1. Festlegung auf "use site"-Varianz
2. Laufzeitprüfung von parametrisierten Subtyp-Beziehungen

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.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords Java, subtype checking, decidability, fluent interface, parser generator, Turing machine

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

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*

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

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).

Java Generics are Turing Complete

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

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.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords Java, subtype checking, decidability, fluent interface, parser generator, Turing machine

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

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).

Java Generics are Turing Complete

Theorem 1. *It is undeci ... er t <: t′ according to a given class table.*

Abstract

This paper describes a reduction fron
machines to subtype checking in Ja
checking in Java is undecidable, which
Kennedy and Pierce in 2007. It also follows
can recognize any recursive language, v
Gil and Levy from 2016. The latter po
generator for fluent interfaces.

Categories and Subject Descriptors [
and Features]

Keywords Java, subtype checking, decidability,
parser generator, Turing machine

1. Introduction

Is Java type checking decid

s undecidable whether $t <: t'$ according to a given

n is a context free grammar G that describes a
over an alphabet Σ of method names. We can
class definitions, a type T, and expressions Start,
the code

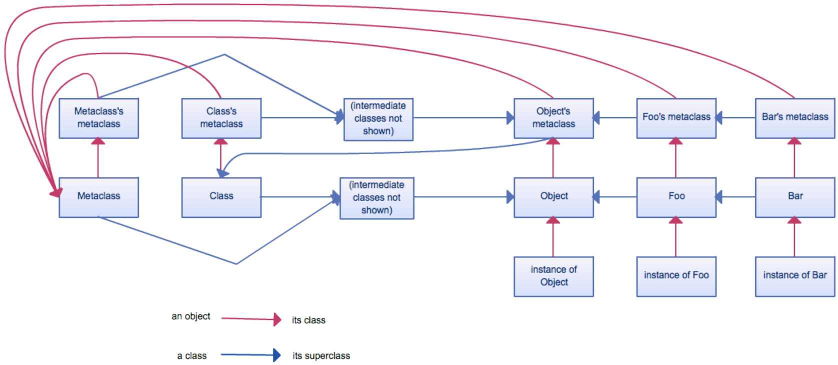
$$= \quad Start.f^{(1)}().f^{(2)}()\dots f^{(m)}().Stop$$

nd only if $f^{(1)}f^{(2)}\dots f^{(m)} \in \mathcal{L}$. Moreover, the class
size polynomial in the size of G, and the Java code
pe-checked in time polynomial in the size of G.

eorem 1 is proved by a reduction from the halting problem
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 prelimina (Sections 3 and 4).
followed by Theorem 2, which is on to generating
s for fluent interfaces rator makes
types;

Verhandlung

Warum nicht alles reifizieren?



Situation in C#

C# hat reifizierte Generics erkauft durch duplizierte APIs

Situation in Haskell

```haskell
unsafeCast :: a -> b
```

Situation in Haskell

```
unsafeCast :: a -> b

cast :: (Typeable a, Typeable b) => a -> Maybe b
```

Was ist eigentlich Polymorphismus?

Subtyp-Polymorphismus

Die konkrete Methode, die bei einem Aufruf `obj.f(x)` aufgerufen wird, ist erst zur Laufzeit ermittelbar.

Was ist eigentlich Polymorphismus?

Parametrischer Polymorphismus

Eine Funktion (oder Klasse) ist nicht nur über Werte, sondern auch über Typen parametrisiert.

Was ist eigentlich Polymorphismus?

Ad-hoc-Polymorphismus

Die Implementierung einer überladenen Methode wird vom Compiler statisch an Hand der involvierten Typen selektiert.

You’re NOT gonna need it!

Apr 4, 1998 • [Practices, XProgramming]

Often you will be building some class and you’ll hear yourself saying “We’re going to need…”.

Resist that impulse, every time. Always implement things when you actually need them, never when you just foresee that you need them. Here’s why:

- Your thoughts have gone off track. You’re thinking about what the class might be, rather than what it must be. You were on a mission when you started building that class. Keep on that mission rather than let yourself be distracted for even a moment.
- Your time is precious. Hone your sense of progress to focus on the real task, not just on banging out code.
- You might not need it after all. If that happens, the time you spend implementing the method will be wasted; the time everyone else spends reading it will be wasted; the space it takes up will be wasted.

YAGNI und Typen

> “ *Folks think parametric polymoprhism is an example of YAGNI – this is backwards.*

YAGNI und Typen

> "*Folks think parametric polymoprhism is an example of YAGNI – this is backwards.*
> *Premature concretization is an example of YAGNI.*"
>
> – Michael Pilquist

Mehr Typparameter!

Traditioneller Entwurf

```scala
case class BlogPost(
  date: Date,
  title: String,
  author: User,
  text: String
)
```

Mehr Typparameter!

Parametrischer Entwurf

```
case class BlogPost[T](
  date: Date,
  title: String,
  author: User,
  text: T
)
```

Mehr Typparameter!

Parametrischer Entwurf

```scala
case class BlogPost[T](
  date: Date,
  title: String,
  author: User,
  text: T
)
```

Vorteile

- `BlogPost` interessiert sich nicht für den Text:
  - kodiert (UTF-8)
  - dekodiert (Codepoints)
  - internationalisiert
  - escaped (HTML)

Mehr Typparameter!

Parametrischer Entwurf

```
case class BlogPost[T](
  date: Date,
  title: String,
  author: User,
  text: T
)
```

Vorteile

- Operationen interessieren sich nicht für den Text:
  - Archivübersicht
  - Publikation
  - Bearbeiten von Metadaten

Mehr Typparameter!

Parametrischer Entwurf

```
case class BlogPost[T](
  date: Date,
  title: String,
  author: User,
  text: T
)
```

Vorteile

- Sätze für lau

Sätze für lau

- basierend auf „Parametrizität"
- vereinfacht: Funktion mit Typparameter weiß nichts und darf nichts

Sätze für lau

- basierend auf „Parametrizität“
- vereinfacht: Funktion mit Typparameter weiß nichts und darf nichts

Beispiel

```java
// Signatur beschreibt exakt eine mögliche Funktion
public <T> T id(T t);
```

Sätze für lau

- basierend auf „Parametrizität“
- vereinfacht: Funktion mit Typparameter weiß nichts und darf nichts

Fortgeschrittenes Beispiel

```java
// Summary enthält nicht den Text
public <T> Html renderSummary(BlogPost<T> post);
```

Sätze für lau

- basierend auf „Parametrizität"
- vereinfacht: Funktion mit Typparameter weiß nichts und darf nichts

Fortgeschrittenes Beispiel

```java
// Summary enthält nicht den Text
public Html renderSummary(BlogPost<?> post);
```

Sätze für lau

- basierend auf „Parametrizität"
- vereinfacht: Funktion mit Typparameter weiß nichts und darf nichts

Beispiel aus der Bibliothek

```
list.map(f).map(g) == list.map(f andThen g)
list.map(f).filter(g) == list.filter(f andThen g).map(f)
```

Theorems for free!

Philip Wadler
University of Glasgow*

June 1989

Abstract

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

1 Introduction

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick. But first, let's look at an example.

list of A yielding a list of A', and $r_A : A^* \to A^*$ is the instance of r at type A.

The intuitive explanation of this result is that r must work on lists of X for *any* type X. Since r is provided with no operations on values of type X, all it can do is rearrange such lists, independent of the values contained in them. Thus applying a to each element of a list and then rearranging yields the same result as rearranging and then applying a to each element.

For instance, r may be the function $\mathit{reverse} : \forall X.\ X^* \to X^*$ that reverses a list, and a may be the function $\mathit{code} : \mathit{Char} \to \mathit{Int}$ that converts a character to its ASCII code. Then we have

$$\begin{array}{ll} & \mathit{code}^*\ (\mathit{reverse}_{\mathit{Char}}\ [\text{'a'}, \text{'b'}, \text{'c'}]) \\ = & [99, 98, 97] \\ = & \mathit{reverse}_{\mathit{Int}}\ (\mathit{code}^*\ [\text{'a'}, \text{'b'}, \text{'c'}]) \end{array}$$

Verbotene Dinge

- `null`
- Exceptions (außer `Error`, sofern unfangbar)
- `isinstanceof`
- Casting
- `equals`, `toString`, `hashCode`
- `getClass`
- globale Seiteneffekte

Verbotene Dinge

- `null`
- Exceptions (außer `Error`, sofern unfangbar)
- `isinstanceof`
- Casting
- `equals`, `toString`, `hashCode`
- `getClass`
- globale Seiteneffekte

Depression

Aber ich brauche Sammlungen von Objects …

> “ *If you encounter an Object in your code you should worry.*

Aber ich brauche Sammlungen von Objects …

> *“If you encounter an Object in your code you should worry. Where did it lose its type information?”*
>
> – Jens Schauder

Metaprogrammierung

> “*Metaprogramming is a workaround for an abstraction your programming language doesn’t have yet.*”

Metaprogrammierung

> *“Metaprogramming is a workaround for an abstraction your programming language doesn’t have yet.”*

In Java ist Metaprogrammiering (Reflection) grundsätzlich unsicher.

Was kann man tun?

Lösungsvorschlag

- „Super Type Tokens“ nach Neal Gafter,[1] 2006
- ermöglicht „Typesafe Heterogeneous Containers“

[1] `https://gafter.blogspot.de/2006/12/super-type-tokens.html`

Was kann man tun?

Lösungsvorschlag

- „Super Type Tokens“ nach Neal Gafter,[1] 2006
- ermöglicht „Typesafe Heterogeneous Containers“

```
public abstract class TypeToken<T> {}

new TypeToken<List<Int>> {}
```

[1] https://gafter.blogspot.de/2006/12/super-type-tokens.html

Type Tokens in Scala

```
scala> classTag[List[Int]]
res0: ClassTag[List[Int]] = List

scala> typeTag[List[Int]]
res1: TypeTag[List[Int]] = TypeTag[List[Int]]
```

Type Tokens in Scala

```
scala> classTag[List[Int]]
res0: ClassTag[List[Int]] = List

scala> typeTag[List[Int]]
res1: TypeTag[List[Int]] = TypeTag[List[Int]]

scala> def foo[T] = typeTag[T]
<console>:17: error: No TypeTag available for T
     def foo[T] = typeTag[T]
```

Akzeptanz

In der Praxis?

Feststellung

Type Erasure ist gut.

In der Praxis?

Feststellung

Type Erasure ist gut.

Problem

- sichere Abhilfen existieren (z.B. Scala)
- aber: bringen wenige Vorteile gegenüber Type Reification

In der Praxis?

Feststellung

Type Erasure ist gut.

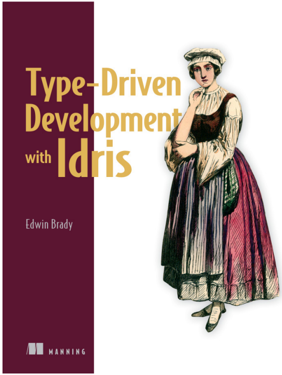
Problem

- sichere Abhilfen existieren (z.B. Scala)
- aber: bringen wenige Vorteile gegenüber Type Reification
- besser: Entwurfsmuster überdenken

Typgetriebene Entwicklung

Leitlinien

- konkrete Typen vermeiden
- Implementation durch Typen einschränken
- Compiler als Hilfe, nicht als Hindernis
- ungültige Zustände unrepräsentierbar machen
- „Type Tetris“[2]



[2] http://underscore.io/blog/posts/2017/04/11/type-tetris.html

Q & A

www.innoq.com

Lars Hupel

lars.hupel@innoq.com

@larsr_h

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim a. Rh.
Germany
+49 2173 3366-0

Ohlauer Str. 43
10999 Berlin
Germany

Ludwigstr. 180 E
63067 Offenbach
Germany

Kreuzstr. 16
80331 München
Germany

c/o WeWork
Hermannstrasse 13
20095 Hamburg
Germany

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
+41 41 743 01 11

Albulastr. 55
8048 Zürich
Switzerland

LARS HUPEL

Consultant
innoQ Deutschland GmbH

Lars enjoys programming in a variety of languages, including Scala, Haskell, and Rust. He is known as a frequent conference speaker and one of the founders of the Typelevel initiative which is dedicated to providing principled, type-driven Scala libraries.

Bildquellen

- Braunschweig: `https://pixabay.com/photos/architecture-colorful-facade-houses-2260836/`
- Generic Java crew: `https://homepages.inf.ed.ac.uk/wadler/gj/`
- Tiger: `https://pixabay.com/photos/tiger-tiergarten-nuremberg-big-cat-2940963/`
- Metaclass diagram: `https://commons.wikimedia.org/wiki/File:Smalltalk_metaclass.png`, Xuan Luo
- Sad Rainbow Dash: `https://www.deviantart.com/dasprid/art/Sad-Rainbow-Dash-418857601`