Verifying Object Construction

How to use the builder pattern with the type safety of constructors

Martin Kellogg\textsuperscript{a}, Manli Ran\textsuperscript{b}, Manu Sridharan\textsuperscript{b}, Martin Schäf\textsuperscript{c}, Michael D. Ernst\textsuperscript{a,c}

\textsuperscript{a}University of Washington \quad \textsuperscript{b}University of California, Riverside \quad \textsuperscript{c}Amazon Web Services
Object construction APIs

```java
public class UserIdentity {
    private final String name; // required
    private final int id; // required
    private final String nickname; // optional
}
```
Object construction APIs

```java
public class UserIdentity {
    private final String name; // required
    private final int id; // required
    private final String nickname; // optional
}

public UserIdentity(String name, int id);
public UserIdentity(String name, int id, String nickname);
```
Object construction APIs

```java
public UserIdenity(String name, int id);
public UserIdenity(String name, int id, String nickname);

new UserIdentity("myName");
```
Object construction APIs

public UserIdentity(String name, int id);
public UserIdentity(String name, int id, String nickname);

new UserIdentity("myName");

error: constructor UserIdentity in class UserIdentity cannot be applied to given types;
    new UserIdentity("myName");
    ^
required: String,int
found: String
reason: actual and formal argument lists differ in length
Pros and cons of constructors

+ compile-time verification that arguments are sensible
Pros and cons of constructors

+ compile-time verification that arguments are sensible

- user must define each by hand
- exponentially many in number of optional parameters
- arguments are positional (hard to read code)
The builder pattern

```java
public class UserIdentity {
    public static UserIdentityBuilder builder();
    public class UserIdentityBuilder {
        public UserIdentityBuilder name();
        public UserIdentityBuilder id();
        public UserIdentityBuilder nickname();
        public UserIdentity build();
    }

    ...
}
```
The builder pattern

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```
Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically
The builder pattern

```
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .build(userId)
    .build();
```
The builder pattern

```
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .build();
```

Possible outcomes:
- Run-time error (bad!)
- Malformed object is used (worst!)
The builder pattern

UserIdentity identity = UserIdentity.builder()
    .name(username)
    .build();

Possible outcomes:
- Run-time error (bad!)
- Malformed object is used (worst!)
Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided
Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided
Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided

“We get this feature request every other week”
- Reinier Zwitserloot, Lombok project lead
Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided

Our approach:

- Provides **type safety** for uses of the builder pattern
- **Keeps advantages** of builder pattern vs. constructors

“We get this feature request every other week”
- Reinier Zwitserloot, Lombok project lead
Builder correctness as a typestate analysis

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```
Builder correctness as a typestate analysis

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

Diagram of state transitions for `identity`.
Builder correctness as a typestate analysis

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```
Builder correctness as a typestate analysis

```
UserIdentity identity = 
    UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

**Problem:**
Arbitrary typestate analysis is expensive: a whole-program alias analysis is required for soundness.
Builder correctness as a typestate analysis

Key insight:
Transitions flow in one direction!

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

Diagram:
- From `name()` to `id()`
- From `id()` to `build()`
- From `build()` to `name()`
- From `build()` to `id()`
- From `id()` to `build()`
- From `name()` to `build()`
- From `build()` to `X` (dead end)
Builder correctness as a typestate analysis

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

**Key insight:**
Transitions flow in one direction!
Builder correctness as a typestate analysis

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

Key insight:
Transitions flow in one direction!
Builder correctness as a typestate analysis

`UserIdentity identity = `UserIdentity.builder().
  .name(username)
  .id(userId)
  .build();

“accumulation analysis”

Key insight:
Transitions flow in one direction!
Advantages of accumulation analysis

- always safe to under-approximate
Advantages of accumulation analysis

- always safe to under-approximate
- does not require alias analysis for soundness
Advantages of accumulation analysis

- always safe to under-approximate
- does not require alias analysis for soundness
- can be implemented modularly (e.g., as a type system)
Advantages of a type system

- provides guarantees
- no alias analysis + modular $\Rightarrow$ scalable
- type inference reduces need for annotations
build()’s specification

build(@CalledMethods({"name", "id"}))
UserIdentityBuilder this;
Results (1 of 3): security vulnerabilities

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>9.1M</td>
</tr>
<tr>
<td>Vulnerabilities found</td>
<td>16</td>
</tr>
<tr>
<td>False warnings</td>
<td>3</td>
</tr>
<tr>
<td>Annotations</td>
<td>34</td>
</tr>
</tbody>
</table>
Contributions

- **Static safety** of constructors with flexibility of **builders**
- **Accumulation analysis**: special case of typestate
  - Does not require whole-program alias analysis

https://github.com/kelloggm/object-construction-checker
Accumulation doesn’t need alias analysis

```java
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
UserIdentityBuilder b2 = b;
b2.id(userId)
UserIdentity identity = b.build();
```
Accumulation doesn’t need alias analysis

```java
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
UserIdentityBuilder b2 = b;
b2.id(userId)
UserIdentity identity = b.build();
```

False positive here is worst-case scenario
Why typestate needs alias analysis

```java
File f = ...;
f.open();
File f2 = f;
f.close();
f2.read();
```
Why typestate needs alias analysis

File f = ...
File f2 = f;
f.close();
f2.read();

No alias analysis leads to false negative
Example: Netflix/SimianArmy

```java
public List<Image> describeImages(String... imageIds) {
    DescribeImagesRequest request = new DescribeImagesRequest();

    if (imageIds != null) {
        request.setImageIds(Arrays.asList(imageIds));
    }

    DescribeImagesResult result = ec2client.describeImages(request);

    return result.getImages();
}
```
The builder pattern

@Builder
public class UserIdentity {
    private final String name;  // required
    private final int id;       // required
    private final String nickname;  // optional
}

```java
The builder pattern

@Builder
day [text]
```
The builder pattern

@Builder
public class UserIdentity {
    private final @NonNull String name;
    private final @NonNull int id;
    private final String nickname; // optional
}

The builder pattern

@Builder
public class UserIdentity {
    private final @NonNull String name;
    private final @NonNull int id;
    private final String nickname; // optional
}

UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
Type hierarchy

@CalledMethods({}) Object

@CalledMethods({"name"}) Object

@CalledMethods({"name", "id"}) Object
What’s the type of b?

```
UserIdentityBuilder b = UserIdentity.builder();

b.name(username);

b.id(userId)

UserIdentity identity = b.build();
```
What’s the type of $b$?

```java
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
b.id(userId)

UserIdentity identity = b.build();
```
What’s the type of b?

```java
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
b.id(userId);

UserIdentity identity = b.build();
```
What’s the type of `b`?

```java
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
b.id(userId);

UserIdentity identity = b.build();
```
Fluent APIs and receiver aliasing

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```
Fluent APIs and receiver aliasing

```
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

@CalledMethods({"id"})
Fluent APIs and receiver aliasing

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

How do we know that the return type of `id()` is the **same object** that `name()` was called on?
Returns receiver checking

A special case of aliasing, needed for **precision**!
Returns receiver checking

A special case of aliasing, needed for precision!

class UserIdentityBuilder {
    @This UserIdentityBuilder name();
    @This UserIdentityBuilder id();
}
Showing correct code is safe

```java
UserIdentity identity = UserIdenity.builder()
    .name(username)
    .id(userId)
    .build();
```
Showing correct code is safe

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(userId)
    .build();
```

Accumulate more “called methods”
Results (2 of 3): Lombok user study

6 industrial developers with Java + Lombok experience

Task: add a new `@NonNull` field to a builder, and update all call sites

Results:

- 6/6 succeeded with our tool, only 3/6 without
- Those who succeeded at both 1.5x faster with our tool
- “It was easier to have the tool report issues at compile time”
Results (3 of 3): case studies

5 projects: 2 Lombok, 3 AutoValue (~200k sloc)

653 calls verified, 1 true positive (google/gapic-generator)

131 annotations, 14 false positives

"your static analysis tool sounds truly amazing!"
- gapic-generator engineer