Safe Stream-Based Programming with Refinement Types

Jason Chen

Jesse Huang

Stream Based Programming

Data is viewed as a stream of values and can be manipulated with call back functions.

Stream based programming is popular because it is great for handling asynchronous data sources in an interactive software.

Some benefits of this is there are no worries of data races or deadlocks.

Some Issues

Popular UI frameworks reserve a single main thread where all UI interactions must occur

Application developers need avoid interacting with UI from other threads

Such invalid thread access bugs are very common in practice but are hard to debug

Current testing technique and program analysis are inadequate in handling the issue.

Using a Type-checker

Annotate codebase with types to statically verify UI access

Implementation Goals of the type-checker:

- Reliably guarantees safety and correctness
- Easy to use and integrate into workflow
- Applicable to legacy codebase

Effect Typing

Effect typing is an annotation that explains what is being done.

Using effect typing can help detect improper UI access.

Effect typing enables the verification of proper UI access with little burden on the developers.

The complexity of large frameworks comes with a steep learning curve will inevitably lead to threading bugs that require nuanced understanding of the framework.

Effects

Effects system is a technique in which functions are annotated with effect qualifiers which can be used to check the function so that it does not perform any action that is not permitted.

There are 2 effects @UIEffect and @SafeEffect.

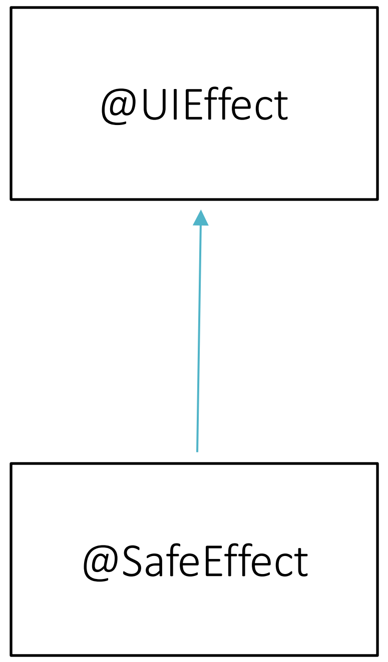
@UIEffect denotes that the method may interact with the UI

@SafeEffect denotes that the method is guaranteed to not interact with the UI.

Refinement Type

We give all methods and call-back like objects a qualifier which assumes the highest effect that can occur.

If a method can access the UI then it will be annotated with @UIEffect.



Effects Relation

The sub-effecting relation $\preceq$ is given by @SafeEffect $\preceq$ @UIEffect and the two reflexive relationships, @SafeEffect $\preceq$ @SafeEffect and @UIEffect $\preceq$ @UIEffect.

This means that a method with a safe effect can be used instead of a UI effect.

This does not work the other way around.

Transitivity and Inheritance

A method with an effect annotation (e) can only call a method with a different annotation (e') if $e' \preceq e$.

A method with an effect annotation (e) can override a method with a different effect annotation (e') if $e' \preceq e$.

Example

```
class A {
@UIEffect void foo () {...}
@SafeEffect void bar () {...} }
class B extends A {
// Transitivity violation
@SafeEffect void baz () { foo () ; }
// Inheritance violation
@UIEffect void bar () {...} }
```

baz() with effect-type @SafeEffect attempts to call foo() with effect-type @UIEffect

@UIEffect typed bar() cannot override a @SafeEffect typed function

Example

```
Observable <... > carLocationData = ... ;
carLocationData
. filter ( car -> /* car has no passenger */)
. observeOn ( AndroidSchedulers . mainThread () )
. delay (100 , TimeUnit . MILLISECONDS )
. subscribe (
car -> { /* display car on map */ } ,
err -> { /* render error message */ })
```

```
error: [rx.thread.violation] Subscribing a callback with
@UIEffect to an observable scheduled on @CompThread;
@UIEffect effects are limited to @UIThread observables
```

@AnyThread is given to carLocationData stream,

@AnyThread for the result of filter,

@UIThread for the result of observeOn,

@CompThread for the result of delay, and

@UIEffect for the lambda expressions passed to subscribe.

Subscription of a @UIEffect function onto a @CompThread function is what caused the bug.

Dynamic Threading

Android applications that use stream-based programming frameworks have stream operators with dynamic threading behavior.

```
runOnUiThread VS subscribe
```

Effect typing of functions alone does not address this issue.

Solution

Expand the effect type system with annotations that define stream types by their thread

This introduces 4 refinement types:

- @AnyThread
- @UIThread
- @CompThread
- @BottomThread

Solution cont.

Combining thread refinement types for streams and effect types for methods enables

the verification of UI thread-safety.

UI effectful callbacks will only be able to occur on UI-threaded streams (`@AnyThread` and `@UIThread`)

Thread Type Refinement Hierarchy

Place refinement types on threads.

Threads that cannot be determined statically will be given @AnyThread.

$$\begin{tikzcd}
 & \boxed{\text{@Anythread}} & \\
\boxed{\text{@CompThread}} \arrow[ur] & & \boxed{\text{@UIThread}} \arrow[ul] \\
 & \boxed{\text{@BottomThread}} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Qualifier Polymorphism

Design patterns that take advantage of modularity and reusability (OOP) have effect and thread behavior that cannot be expressed by a fixed type and refinement.

```
Set<T> singleton(T obj){...}
```

Qualifier polymorphism employs a generic refinement variable, defined as @PolyThread and @PolyUIEffect.

Anonymity and Lambda Expressions

Anonymous class or lambda must have a compatible type qualifier with its receiver

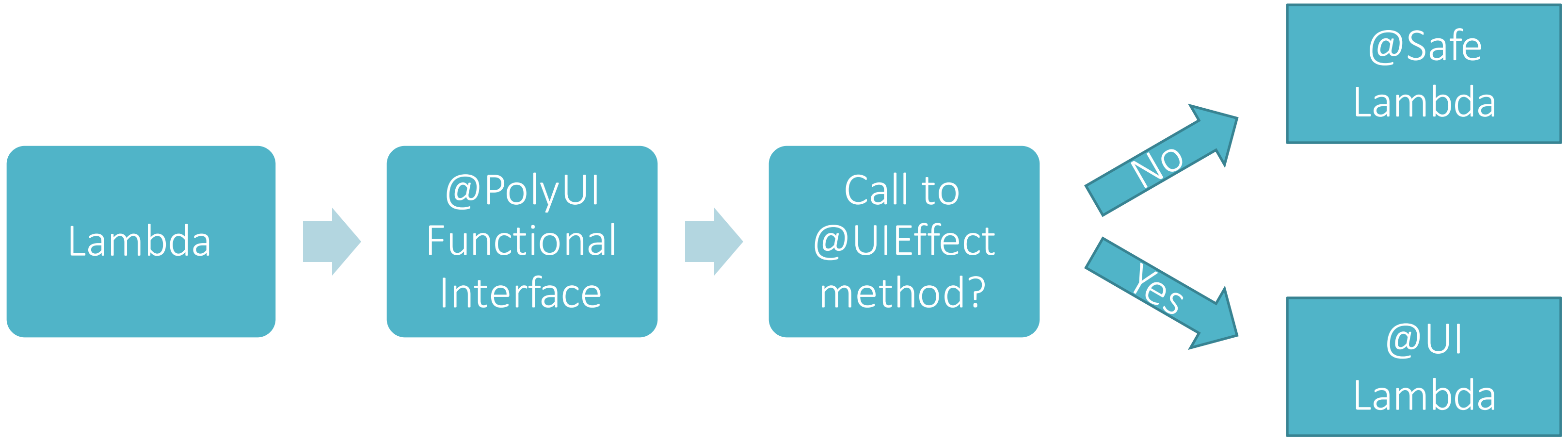
`new` ***`@UI`*** `Consumer{...}` **where Consumer has a `@UIEffect` method**

Lambda expressions cannot be explicitly type annotated

Anonymity and Lambda Expressions Cont.

The base type inference mechanism does not work with type qualifiers

Instead, inspect the body of lambda to determine type.



External Code

Integration Tools

Annotation stubs

Class-level short-hands eg.`@UIType`

```
@UIType class ScrollView {//...
  @SafeEffect boolean post(@UI Runnable action);}
class Observable <T> {//...
  @CompThread Observable <T> delay(long delay , TimeUnit unit);
  @PolyThread Observable <T> observeOn(@PolyThread Scheduler thread);
  @PolyThread Observable <T> take(@PolyThread Observable <T> this , int k);}
```

Effectiveness

Of 8 open-source applications:

An average of 2.3 hours per application, with one annotation for every 186 lines of source code.

Total of 33 threading errors across 6 apps in the evaluation, an average of 4.1 errors per application.

Questions?