Safe Stream-Based Programming with Refinement Types

Benno Stein
University of Colorado
benno.stein@colorado.edu

Lazaro Clapp
Uber Technologies, Inc.
lazaro@uber.com

Manu Sridharan
Uber Technologies, Inc.
msridhar@uber.com

Bor-Yuh Evan Chang
University of Colorado
evan.chang@colorado.edu
Mobile app reliability is crucial

Uber

Rider

Driver

Eats
Mobile app reliability is crucial

- Rider crash: can’t get home
- Driver crash: can’t earn
Mobile app reliability is crucial

- Rider crash: can’t get home
- Driver crash: can’t earn
- Whole business depends on mobile apps
Mobile app reliability is crucial

- Rider crash: can’t get home
- Driver crash: can’t earn
- Whole business depends on mobile apps
- Patching through third-party app stores is slow
Apps are fast-moving, large, and complex

- Hundreds of developers working simultaneously
- Millions of lines of code
- Apps depend upon numerous general-purpose libraries
UI Thread Safety

Don’t touch the UI from off the main thread. *Easy enough.*
UI Thread Safety

Don’t touch the UI from off the main thread. *Easy enough.*

... not even transitively or through a library.

e.g. *innocuousLookingMethod* calls *foo* calls *bar* calls *uiMethod*
UI Thread Safety

Don’t touch the UI from off the main thread.  *Easy enough.*

... not even transitively or through a library.

  e.g. `innocuousLookingMethod` calls `foo` calls `bar` calls `uiMethod`

... especially when using stream-based programming libraries with complex threading behavior
→ Stream-Based Programming
→ Effect & Thread Type Refinements
→ UI-Thread Safety
→ Evaluation
Reactive Extensions

ReactiveX

“An API for asynchronous programming with observable streams”
Reactive Extensions

**ReactiveX**

“An API for asynchronous programming with observable streams”

- Create or receive streams of events and data
- Use expressive operators to compose and transform streams
- Subscribe callbacks to streams to perform side effects
Reactive Extensions

ReactiveX

“An API for asynchronous programming with observable streams”

- Create or receive streams of events and data
- Use expressive operators to compose and transform streams
- Subscribe callbacks to streams to perform side effects

Used by:

- Applied Duality
- Microsoft
- NETFLIX
- GitHub
- SoundCloud
- Trello
- Treehouse
- SeatGeek
- Couchbase
- futurice
- O.C.TANNER
- Uber
- Airbnb
Stream-Based Programming

Reactive Extensions (ReactiveX) example:
Stream-Based Programming

Reactive Extensions (ReactiveX) example:

```csharp
Observable<...> carLocationData = ... ;
```
Stream-Based Programming

Reactive Extensions (ReactiveX) example:

```java
Observable<...> carLocationData = ... ;
carLocationData
    .filter( car -> /* car has no passenger */ )
```
Stream-Based Programming

Reactive Extensions (ReactiveX) example:

```java
Observable<...> carLocationData = ... ;
carLocationData
    .filter( car -> /* car has no passenger */ )
    .observeOn(AndroidSchedulers.mainThread())
```
Stream-Based Programming

Reactive Extensions (ReactiveX) example:

```java
Observable<...> carLocationData = ... ;
carLocationData
    .filter( car -> /* car has no passenger */ )
    .observeOn(AndroidSchedulers.mainThread())
    .delay(100, TimeUnit.MILLISECONDS)
```
Stream-Based Programming

Reactive Extensions (ReactiveX) example:

```java
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 */ });
```
→ Stream-Based Programming
→ Effect & Thread Type Refinements
→ UI-Thread Safety
→ Evaluation
Effects & Refinement Types

Function types typically only encode input and output:

\[ \tau_{in} \rightarrow \tau_{out} \]
Effects & Refinement Types

Function types typically only encode input and output:

$$\tau_{in} \rightarrow \tau_{out}$$

*Effect types* refine function types by their side-effects:

$$\tau_{in} \rightarrow e \tau_{out}$$
Effects & Refinement Types

Function types typically only encode input and output:

\[ \tau_{in} \rightarrow \tau_{out} \]

*Effect types* refine function types by their side-effects:

\[ \tau_{in} \rightarrow_{e} \tau_{out} \]

e.g. UI access, network I/O, heavy computation
Effects

// java.lang.Math
int max(int x, int y) {...}

// android.widget.Button
void setText(String text) {...}

// com.example.MyApp
void foobar() {...}

// some obscure Android library
void poorlyDocumentedMethod() {...}
Effects

// java.lang.Math
@SafeEffect int max(int x, int y) {...}

// android.widget.Button
void setText(String text) {...}

// com.example.MyApp
void foobar() {...}

// some obscure Android library
void poorlyDocumentedMethod() {...}
Effects

// java.lang.Math
@SafeEffect int max(int x, int y) {...}

// android.widget.Button
@UIEffect void setText(String text) {...}

// com.example.MyApp
void foobar() {...}

// some obscure Android library
void poorlyDocumentedMethod() {...}
Effects

// java.lang.Math
@SafeEffect int max(int x, int y) {...}

// android.widget.Button
@UIEffect void setText(String text) {...}

// com.example.MyApp
?????? void foobar() {...}

// some obscure Android library
?????? void poorlyDocumentedMethod() {...}
Effect Typing as Call-graph Reachability

All methods
Effect Typing as Call-graph Reachability

All methods

Android UI Libraries
Effect Typing as Call-graph Reachability

All methods

foo()

bar() → Android UI Libraries
Effect Typing as Call-graph Reachability

All methods

Potentially UI-effecting methods

foo()

bar() → Android UI Libraries
Effect Type Refinements

@UIEffect

↑

@SafeEffect

“Effects for Controlling UI Object Access” Gordon et al., ECOOP ‘13
Effect Type Refinements

@UIEffect $\uparrow$

@SafeEffect

Transitivity:
A method with effect annotation $e$ can call a method with effect annotation $e'$ if and only if $e \preceq e'$

“Effects for Controlling UI Object Access” Gordon et al., ECOOP ‘13
Effect Type Refinements

@UIEffect → @SafeEffect

Transitivity:
A method with effect annotation $e$ can call a method with effect annotation $e'$ if and only if $e \preceq e'$

Inheritance:
A method with effect annotation $e$ can override a method with effect annotation $e'$ if and only if $e \preceq e'$
Effects alone are insufficient

Previous work with effect types handles UI library interfaces with \texttt{fixed} threading behavior, such as:

\begin{verbatim}
runOnUiThread : Runnable -> void
\end{verbatim}
Effects alone are insufficient

Previous work with effect types handles UI library interfaces with \textit{fixed} threading behavior, such as:

\begin{center}
\begin{verbatim}
runOnUiThread : Runnable \rightarrow \text{void}
\end{verbatim}
\end{center}

\begin{center}
\textit{Definitely runs on the UI thread, can safely touch the UI}
\end{center}

“Effects for Controlling UI Object Access” Gordon et al., ECOOP ‘13
Effects alone are insufficient

Previous work with effect types handles UI library interfaces with \textit{fixed} threading behavior, such as:

\begin{itemize}
  \item \texttt{runOnUiThread} : Runnable \to void
\end{itemize}

Stream-based interfaces have \textit{dynamic} threading behavior, such as:

\begin{itemize}
  \item \texttt{subscribe} : Observable\langle T \rangle \to Consumer\langle T \rangle \to void
\end{itemize}

“Effects for Controlling UI Object Access” Gordon et al., ECOOP ‘13
Effects alone are insufficient

Previous work with effect types handles UI library interfaces with *fixed* threading behavior, such as:

```
runOnUiThread : Runnable -> void
```

Definitely runs on the UI thread, can safely touch the UI

Stream-based interfaces have *dynamic* threading behavior, such as:

```
subscribe : Observable<T> -> Consumer<T> -> void
```

Runs on a thread determined dynamically by the scheduler of the receiver stream

“Effects for Controlling UI Object Access” Gordon et al., ECOOP ’13
Thread Type Refinement

Type Lattice:

@AnyThread

@UiThread     @CompThread

@BottomThread
Thread Type Refinement

Type Lattice:

@AnyThread

@UiThread   @CompThread

@BottomThread

Example stream function types:
Thread Type Refinement

Type Lattice:

```
@AnyThread

@UiThread    @CompThread

@BottomThread
```

Example stream function types:

```java
filter :
@PolyThread Observable<T> -> Predicate<T> -> @PolyThread Observable<T>
```
Thread Type Refinement

Type Lattice:

@AnyThread

@UiThread

@CompThread

@BottomThread

Example stream function types:

filter :
@PolyThread Observable<T> -> Predicate<T> -> @PolyThread Observable<T>

delay :
@AnyThread Observable<T> -> int -> TimeUnit -> @CompThread Observable<T>
Thread Type Refinement

Type Lattice:

@AnyThread
@UiThread
@CompThread
@BottomThread

Example stream function types:

filter :
@PolyThread Observable<T> -> Predicate<T> -> @PolyThread Observable<T>

delay :
@AnyThread Observable<T> -> int -> TimeUnit -> @CompThread Observable<T>

observeOn :
@AnyThread Observable<T>
-> @PolyThread Scheduler -> @PolyThread Observable<T>
→ Stream-Based Programming
→ Effect & Thread Type Refinements
→ UI Thread Safety
→ Evaluation
UI Thread Safety

A stream-based program is *guaranteed* never to access the UI from a non-UI thread if `@UIEffect` callbacks are only subscribed to `@UIThread` streams.
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 */ });
Example Revisited

```java
@AnyThread
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 */ });
```
Example Revisited

```java
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 */ });
```
Example Revisited

```java
@AnyThread
Observable<...> carLocationData = ...;

@SafeEffect
carLocationData
  .filter( car -> /* car has no passenger */ )
  .observeOn(AndroidSchedulers.mainThread())
  .delay(100, TimeUnit.MILLISECONDS)
  .subscribe(
    @UiThread
    car -> { /* display car on map */ },
    @UIEffect
    err -> { /* render error message */ });
```
Example Revisited

```java
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 */ });
```
Example Revisited

```java
delay: @AnyThread Observable<T> -> int -> TimeUnit -> @CompThread Observable<T>

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 */ });
```
Example Revisited

```java
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 */ });
```

@CompThread @UIEffect @UIEffect
Example Revisited

```java
Observable<...> carLocationData = ... ;
carLocationData
    .filter( car -> /* car has no passenger */ )
    .observeOn(AndroidSchedulers.mainThread())
    .delay(1000L, TimeUnit.MILLISECONDS)
    .subscribe(
        car -> { /* display car on map */ },
        err -> { /* render error message */ });
```

@CompThread
@UIEffect
@UIEffect
Observable<...> carLocationData = ...;

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

```java
@AnyThread Observable<...> carLocationData = ...;

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

@SafeEffect
@UIEffect
Fixed Example

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

- `@AnyThread` annotation
- `@SafeEffect` annotation
- `@CompThread` annotation
- `@UIEffect` annotation
Fixed Example

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

@AnyThread

@AnyThread

@CompThread

@UiThread

@SafeEffect

@UIEffect

@UIEffect
Fixed Example

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

carLocationData
  .filter( car -> /* car has no passenger */ )
  .delay(100, TimeUnit.MILLISECONDS)
  .observeOn(Schedulers.mainThread())
  .subscribe(
    car -> { /* display car on map */ },
    err -> { /* render error message */ });
→ Stream-Based Programming
→ Effect & Thread Type Refinements
→ UI Thread Safety
→ Evaluation
Experiments

RQ1: *Is the typechecker practical and easy-to-use?*
- Manual annotation burden is small
- Compile-time performance cost is low
- Error messages and warnings are understandable

RQ2: *Does the typechecker find real bugs and help fix them?*
- Stream-based threading bugs exist in practice
- Typechecker identifies them successfully
- Checked programs are reliably bug-free
Open Source Android apps:
Java applications on GitHub
... that import ReactiveX `AndroidSchedulers`,
... have at least 15 “stars”
... and had been indexed recently.

Uber Case Study:
- Deployed in production for `Driver` and `Eats` apps.
- Over 500k LoC in total
Usability

RQ1: *Is the typechecker practical and easy-to-use?*

- Manual annotation burden is small
- Compile-time performance cost is low
- Error messages and warnings are understandable by real developers

<table>
<thead>
<tr>
<th>App</th>
<th>KLoC</th>
<th>Annotations</th>
<th>Reported Errors</th>
<th>Compile Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForPDA</td>
<td>33.0</td>
<td>197</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>chat-sdk-android</td>
<td>34.6</td>
<td>102</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>trust-wallet-android</td>
<td>8.8</td>
<td>27</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>arch-components-date</td>
<td>0.7</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>MVPArms</td>
<td>6.3</td>
<td>59</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>rxbus</td>
<td>3.3</td>
<td>12</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SmartReceiptsLibrary</td>
<td>39.9</td>
<td>217</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>OpenFoodFacts</td>
<td>14.9</td>
<td>146</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>Averages</td>
<td>17.7</td>
<td>95</td>
<td>4.1</td>
<td>19.5</td>
</tr>
</tbody>
</table>
Usability

RQ1: Is the typechecker practical and easy-to-use?
• Manual annotation burden is small ✔
• Compile-time performance cost is low
• Error messages and warnings are understandable by real developers

<table>
<thead>
<tr>
<th>App</th>
<th>KLoC</th>
<th>Annotations</th>
<th>Reported Errors</th>
<th>Compile Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForPDA</td>
<td>33.0</td>
<td>197</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>chat-sdk-android</td>
<td>34.6</td>
<td>102</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>trust-wallet-android</td>
<td>8.8</td>
<td>27</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>arch-components-date</td>
<td>0.7</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>MVPArms</td>
<td>6.3</td>
<td>59</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>rxbus</td>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SmartReceiptsLib</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenFoodFacts</td>
<td>14.9</td>
<td>146</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>Averages</td>
<td>17.7</td>
<td>95</td>
<td>4.1</td>
<td>19.5</td>
</tr>
</tbody>
</table>

One annotation per 186 LoC
Usability

RQ1: *Is the typechecker practical and easy-to-use?*
- Manual annotation burden is small ✔
- Compile-time performance cost is low ✔
- Error messages and warnings are understandable by real developers

<table>
<thead>
<tr>
<th>App</th>
<th>KLoC</th>
<th>Annotations</th>
<th>Reported Errors</th>
<th>Compile Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForPDA</td>
<td>33.0</td>
<td>197</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>chat-sdk-android</td>
<td>34.6</td>
<td>102</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>trust-wallet-android</td>
<td>8.8</td>
<td>27</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>arch-components-date</td>
<td>0.7</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>MVPArms</td>
<td>6.3</td>
<td>59</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>rxbus</td>
<td>3.3</td>
<td>12</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SmartReceiptsLibrary</td>
<td>39.9</td>
<td>217</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>OpenFoodFacts</td>
<td>14.9</td>
<td>146</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td><strong>Averages</strong></td>
<td>17.7</td>
<td>95</td>
<td>4.1</td>
<td><strong>19.5</strong></td>
</tr>
</tbody>
</table>
Usability

RQ1: Is the typechecker practical and easy-to-use?
• Manual annotation burden is small ✔
• Compile-time performance cost is low ✔
• Error messages and warnings are understandable by real developers ✔

Uber Case Study:
• Over 4000 commits by 176 Uber developers
• One annotation per 178 LoC by Uber developers
Effectiveness

RQ2: *Does the typechecker find real bugs and help fix them?*

- Stream-based threading bugs exist in practice
- Typechecker identifies them successfully
- Checked programs are reliably bug-free
RQ2: *Does the typechecker find real bugs and help fix them?*

- Stream-based threading bugs exist in practice ✔️
- Typechecker identifies them successfully ✔️
- Checked programs are reliably bug-free

<table>
<thead>
<tr>
<th>App</th>
<th>KLoC</th>
<th>Annotations</th>
<th>Reported Errors</th>
<th>Compile Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForPDA</td>
<td>33.0</td>
<td>197</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>chat-sdk-android</td>
<td>34.6</td>
<td>102</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>trust-wallet-android</td>
<td>8.8</td>
<td>27</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>arch-components-date</td>
<td>0.7</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>MVPArms</td>
<td>6.3</td>
<td>59</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>rxBus</td>
<td>3.3</td>
<td>12</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SmartReceiptsLibrary</td>
<td>39.9</td>
<td>217</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>OpenFoodFacts</td>
<td>14.9</td>
<td>146</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td><strong>Averages</strong></td>
<td>17.7</td>
<td>95</td>
<td><strong>4.1</strong></td>
<td>19.5</td>
</tr>
</tbody>
</table>
Effectiveness

RQ2: *Does the typechecker find real bugs and help fix them?*

- Stream-based threading bugs exist in practice ✅❓
- Typechecker identifies them successfully ✅
- Checked programs are reliably bug-free

**Uber Case Study:**

- 41 changes to threading behavior of stream-processing code during initial setup
- 135 additions of `observeOn(mainThread)` by developers in response to alarms after initial setup
Effectiveness

RQ2: *Does the typechecker find real bugs and help fix them?*

- Stream-based threading bugs exist in practice ✓
- Typechecker identifies them successfully ✓
- Checked programs are reliably bug-free ✓

**Uber Case Study:**

- Zero `CalledFromWrongThreadException` crashes in production in checked code!
  - monitoring period of one month
  - non-zero crash rates in unchecked apps
Contributions:

- Refinement type system for stream threads
- Typechecker implementation for Android
- Evaluation on open-source and industrial apps