$\mathsf{Java}_{\mathsf{UI}}$: Effects for Controlling UI Object Access (Extended Version)*

Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman

University of Washington
{csgordon,wmdietl,mernst,djg}@cs.washington.edu

Abstract. Most graphical user interface (GUI) libraries forbid accessing UI elements from threads other than the UI event loop thread. Violating this requirement leads to a program crash or an inconsistent UI. Unfortunately, such errors are all too common in GUI programs.

We present a polymorphic type and effect system that prevents non-UI threads from accessing UI objects or invoking UI-thread-only methods. The type system still permits non-UI threads to hold and pass references to UI objects. We implemented this type system for Java and annotated 8 Java programs (over 140KLOC) for the type system, including several of the most popular Eclipse plugins. We confirmed bugs found by unsound prior work, found an additional bug and code smells, and demonstrated that the annotation burden is low.

We also describe code patterns our effect system handles less gracefully or not at all, which we believe offers lessons for those applying other effect systems to existing code.

1 Introduction

Graphical user interfaces (GUIs) were one of the original motivations for object-oriented programming [1], and their success has made them prevalent in modern applications. However, they are an underappreciated source of bugs. A Google search for "SWT invalid thread access" — the exception produced when a developer violates multithreading assumptions of the SWT GUI framework — produces over 150,000 results, including bug reports and forum posts from confused developers and users. These bugs are user-visible, and programs cannot recover from them. Typically, they terminate program execution. Furthermore, these bugs require non-local reasoning to locate and fix, and can require enough effort that some such bugs persist for years before being fixed [2]. Because these bugs are common, severe, and difficult to diagnose, it is worthwhile to create specialized program analyses to find errors in GUI framework clients.

A typical user interface library (such as Java's Swing, SWT, and AWT toolkits, as well as toolkits for other languages) uses one UI thread running a polling event loop to handle input events. The library assumes that all updates to UI elements run on the UI thread. Any long-running operation on this thread would prevent the UI from responding to user input, and for this reason the UI library includes methods for running tasks on

* Extended version of the ECOOP 2013 paper of the same title. The extensions are appendices with additional details, including proof of soundness for the λ_{UI} core language.

background threads. A background thread runs independently from the UI thread, and therefore it does not block UI interactions, but it is also restricted in its interactions with the UI. The UI library also provides mechanisms for background threads to update UI elements, primarily by executing a closure on the UI thread, synchronously or asynchronously. For SWT, these correspond to the static methods `syncExec` and `asyncExec` respectively, in the `Display` class. Each accepts a `Runnable` instance whose `run()` method will be executed (synchronously or asynchronously) on the UI thread. For example, a method to update the text of a label from a background thread might look like this:

```
private final JLabel mylabel;
...
public void updateText(final String str) {
    Display.syncExec(new Runnable {
        public void run() { mylabel.setText(str); }
    });
}
```

The separation between the UI and background threads gives several advantages to the UI library implementor:

- Forced atomicity specifications: background threads must interact with the UI only indirectly through the closure-passing mechanism, which implicitly specifies *UI transactions*. Because all UI updates occur on one thread, each transaction executes atomically with respect to other UI updates.
- Minimal synchronization: Assuming clients of the UI library never access UI objects directly, no synchronization is necessary within the UI library when the UI thread accesses UI elements. The only required synchronization in the UI library is on the shared queue where background threads enqueue tasks to run on the UI thread.
- Simple dynamic enforcement: Any library method that is intended to run only on the UI thread can contain an assertion that the current thread is the UI thread.

These advantages for the library implementor become sources of confusion and mistakes for client developers. Each method may be intended to run on the UI thread (and may therefore access UI elements) or may be intended to run on another, background, thread (and therefore must not access UI elements). Client developers must know at all times which thread(s) a given block of code might execute on. In cases where a given type or routine may be used sometimes for background thread work and sometimes for UI work, maintaining this distinction becomes even more difficult. There are alternative designs for GUI frameworks that alleviate some of this confusion, but they are undesirable for other reasons explained in Section 3.5.

The key insight of our work is that a simple type-and-effect system can be applied to clients of UI frameworks to detect all UI thread-access errors statically. There is a one-time burden of indicating which UI framework methods can be called only on the UI thread, but this burden is tractable. Annotations in client code are kept small thanks to a judicious use of default annotations and simple effect polymorphism.

We present a sound static polymorphic effect system for verifying the absence of (and as a byproduct, finding) UI thread access errors. Specifically, we:

- Present a concise formal model of our effect system, λ_{UI} . (Section 2)

- Describe an implementation $\mathsf{Java}_{\mathsf{UI}}$ for the Java programming language, including effect-polymorphic types, that requires no source modifications to UI libraries or clients beyond Java annotations for type qualifiers and effects. (Section 3)
- Evaluate $\mathsf{Java}_{\mathsf{UI}}$ by annotating 8 UI programs and Eclipse plugins totalling over 140KLOC. Our experiments confirm bugs found by unsound previous work [2], find an additional bug, and verify the absence of such bugs in several large programs. (Section 4)
- Identify coding and design patterns that cause problems for our effect system and probably for other effect systems as well, and discuss possible solutions. (Section 4.4)

Our experience identifying UI errors in large existing code bases is likely to prove useful for designing other program analyses that may have nothing to do with UI code. Applying a static type-and-effect system to a large existing code base requires a design that balances expressiveness with low annotation burden. In particular, we found that while effect polymorphism was important, only a limited form (one type-level variable per class) is needed to verify almost all code. However, some programming idioms that do occur in practice are likely to remain beyond the reach of existing static analyses. Overall, we believe our work can pave the way for practical static analyses that enforce heretofore unchecked usage requirements of modern object-oriented frameworks.

$\mathsf{Java}_{\mathsf{UI}}$ and our annotated subject programs are publicly available at:

`http://github.com/csgordon/javaui`

2 Core Language λ_{UI}

The basis for our Java type system is λ_{UI}, a formal model for a multithreaded lambda calculus with a distinguished UI thread. Figure 1 gives the syntax and static semantics for the core language. The language includes two constructs for running an expression on another thread: $\mathsf{spawn}\{e\}$ spawns a new non-UI thread that executes the expression e, while $\mathsf{asyncUI}\{e\}$ enqueues e to be run (eventually) on the UI thread. There are also two kinds of references: $\mathsf{ref}_{\mathsf{safe}}\ e$ creates a standard reference, while $\mathsf{ref}_{\mathsf{ui}}\ e$ creates a reference that may be dereferenced only on the UI thread. Other constructs are mostly standard (dereference, application, function abstraction, assignment, natural numbers, and a unit element) though the lambda construct includes not only an argument and body, but an effect bound ξ on the body's behavior.

Effects ξ include ui for the effect of an expression that must run on the UI thread, and safe for the effect of an expression that may execute on any thread. Types include natural numbers, unit, the standard effectful function type, and reference types with an additional parameter describing the effect of dereferencing that reference.

Figure 1 also gives the static typing rules for λ_{UI} . The form $\Gamma \vdash e : \tau;\xi$ can be read as: given a type environment Γ, the expression e evaluates to a value of type τ, causing effects at most ξ. Most of the rules are fairly standard modulo the distinctions between $\mathsf{spawn}\{e\}$ and $\mathsf{asyncUI}\{e\}$ and between $\mathsf{ref}_{\mathsf{safe}}\ e$ and $\mathsf{ref}_{\mathsf{ui}}\ e$. The rules also assume a least-upper-bound operator on effects ($\sqcup$) for combining multiple effects, and the rules include T-SubEff for permitting safe bodies in ui functions. Richer subtyping/subeffecting, for example full subtyping on function types, would be straightforward to add.

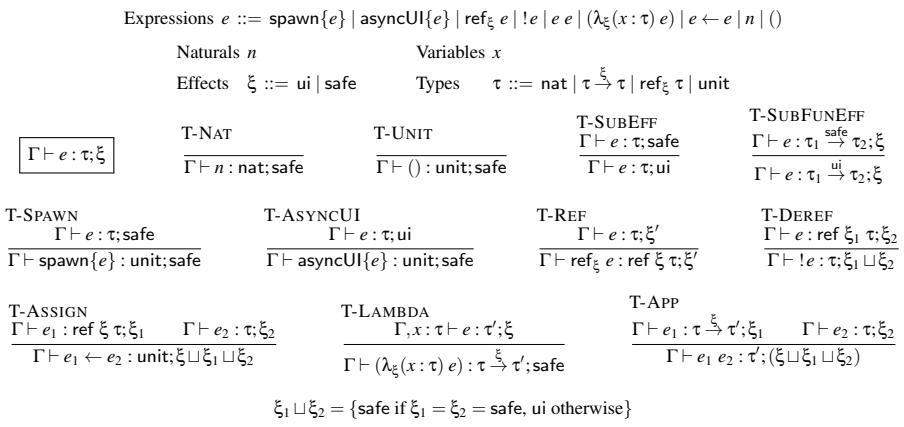

Fig. 1. λ_{UI} syntax for terms and types, and monomorphic type and effect system.

The operational semantics in Figure 2 are mostly standard, with the exception that the distinguished UI thread has a FIFO queue of expressions to execute. After reducing its current expression to a value it dequeues the next expression if available. The expression reduction relation is labeled with the effect of a given action, and the background threads get "stuck" if the next reduction would have the UI effect.[1] Typing for runtime program states is given in Figure 3.

The type system in Figure 1 is sound with respect to the semantics we define in Figure 2 (using the runtime and state typing from Figure 3) and is expressive enough to demonstrate soundness of the effect-related subtyping in the source language $\mathsf{Java}_{\mathsf{UI}}$. λ_{UI} models the ability to statically confine actions to a distinguished thread. Effect-polymorphism is not modeled, but mostly orthogonal: it would require a handful of additional checks at polymorphic instantiation, and an extended subeffecting relation ($\mathsf{safe} \sqsubset \mathsf{polyui} \sqsubset \mathsf{ui}$). Full proofs via syntactic type soundness [3] are available in Appendix C. We state the additional notation, judgments, and main lemmas here:

- Σ for standard-style heap type.
- $\Sigma \vdash H$ for typing the heap H.
- $\Sigma;\Gamma \vdash e : \tau;\xi$ as the expression typing judgment extended for heap typing.
- $H, e \rightarrow_\xi H', e', O$ for small-step expression reduction. The effect ξ is the runtime-observed effect of the reduction; e.g., dereferencing a ui reference has the runtime effect ui. O is an optional effect-expression pair for reductions that spawn new threads; the effect indicates whether a new background thread is spawned or a new UI task is enqueued for the UI thread.
- $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle$ for machine typing: the heap type, a vector of expression types for the UI thread's pending (and single active) expressions, and a vector of expression types for background threads. The maximal effects for each expression are implicit; they

[1] This requires labeling heap cells with the effect of dereferencing the cell. These labels are used only for proving soundness and need not exist in an implementation.

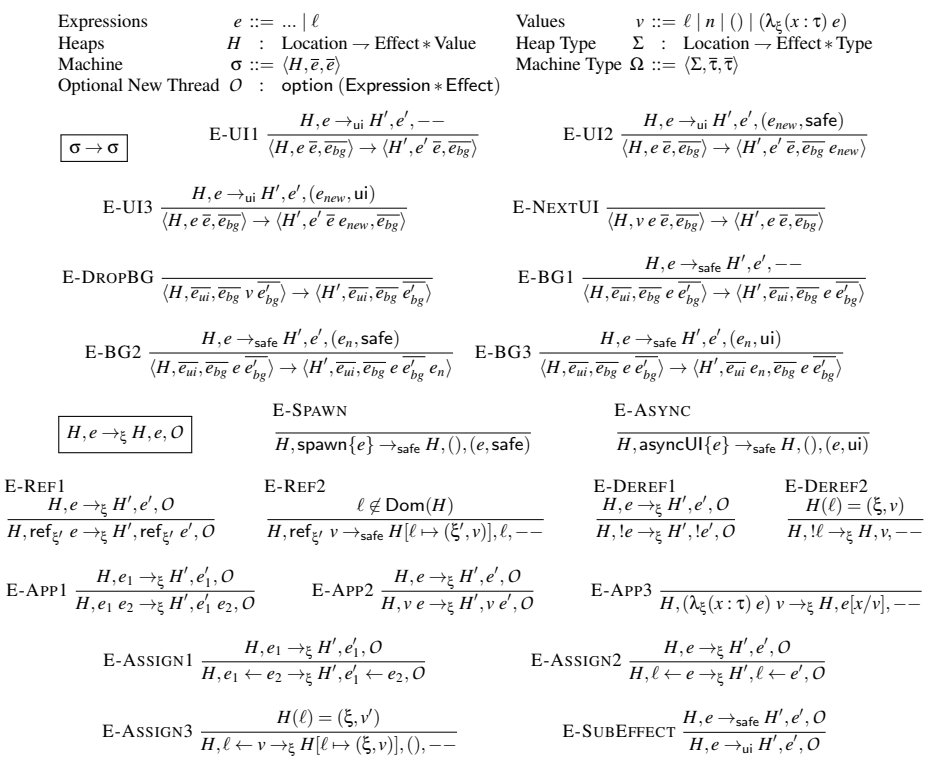

Fig. 2. λ_{UI} runtime expression syntax and operational semantics.

depend on which thread the expression is for: UI expressions may type with effect ui, while background threads must type with the effect safe).

- $\langle H, \overline{e_u}, \overline{e} \rangle$ for machine states: heap, pending UI expressions, and background threads, as with machine state typing.
- $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ for typing a machine state.
- $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e'_u}, \overline{e'} \rangle$ as machine reduction, nondeterministically selecting either the first UI expression or an arbitrary background thread expression to reduce with the heap.

We prove soundness using syntactic type soundness [3]. First, we prove single-threaded type soundness. Contingent on that result, we prove soundness for all threads: all operations with the UI effect execute on the UI thread.

Lemma 1 (Expression Progress). *If $\Sigma \vdash H$ and $\Sigma; \Gamma \vdash e : \tau; \xi$ then either e is a value, or there exists some H', e', and O such that $H, e \to_\xi H', e', O$.*

Lemma 2 (Expression Preservation). *If $\Sigma; \Gamma \vdash e : \tau; \xi$, $\Sigma \vdash H$ and $H, e \to_\xi H', e', O$, then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash H'$, $\Sigma'; \Gamma \vdash e' : \tau; \xi$, and if $O = (e'', \xi')$ then there also exists τ_n such that $\Sigma; \Gamma \vdash e'' : \tau_n; \xi'$.*

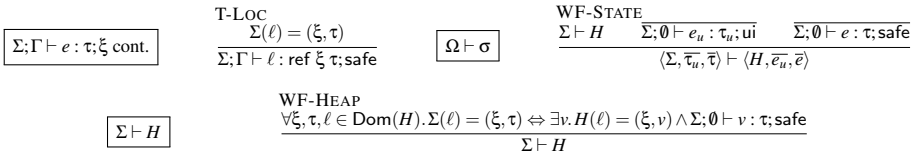

Fig. 3. λ_{UI} program and runtime typing, beyond extending the source typing with the additional Σ.

Corollary 1 (Expression Type Soundness). *If $\Sigma;\Gamma \vdash e : \tau;\xi$, and $\Sigma \vdash H$, then e is a value or there exists $\Sigma' \supseteq \Sigma$, e', H', and O such that $\Sigma' \vdash H'$, and $H, e \to H', e', O$, and $\Sigma';\Gamma \vdash e' : \tau;\xi$ and if $O = (e'', \xi')$ then there exists τ_n such that $\Sigma';\emptyset \vdash e'' : \tau_n;\xi'$.*

Lemma 3 (Machine Progress). *If $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ for non-empty e_u, then either $\overline{e_u} = v :: []$ and $\overline{e} = \emptyset$ or there exists $H', \overline{e_u}', \overline{e}'$ such that $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e_u}', \overline{e}' \rangle$.*

Lemma 4 (Machine Preservation). *If $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ and $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e_u}', \overline{e}' \rangle$, then there exists $\Sigma', \overline{\tau_u}', \overline{\tau}'$ such that $\Sigma' \supseteq \Sigma$ and $\langle \Sigma', \overline{\tau_u}', \overline{\tau}' \rangle \vdash \langle H', \overline{e_u}', \overline{e}' \rangle$*

Corollary 2 (Machine Type Soundness). *If $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ for non-empty e_u, then either $\overline{e_u} = v :: []$ and $\overline{e} = \emptyset$ or there exists $\Sigma', H', \overline{\tau_u}', \overline{e_u}', \overline{\tau}', \overline{e}'$ such that $\Sigma' \supseteq \Sigma$ and $\langle \Sigma', \overline{\tau_u}', \overline{\tau}' \rangle \vdash \langle H', \overline{e_u}', \overline{e}' \rangle$ and $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e_u}', \overline{e}' \rangle$.*

3 $\mathsf{Java}_{\mathsf{UI}}$: Extending λ_{UI} to Java

$\mathsf{Java}_{\mathsf{UI}}$ soundly prevents inappropriate access to UI objects by background threads. $\mathsf{Java}_{\mathsf{UI}}$ extends Java with type qualifier annotations and method effect annotations to indicate which code should run on the UI thread and which code should not directly access UI objects. From the Java developer's perspective, $\mathsf{Java}_{\mathsf{UI}}$ prevents invalid thread access errors, which occur when a background thread calls UI-only methods (such as `JLabel.setText()`) that may only be called from the UI thread. $\mathsf{Java}_{\mathsf{UI}}$ handles the full Java language, including sound handling of inheritance and effect-polymorphic types. A major design goal for $\mathsf{Java}_{\mathsf{UI}}$ was to avoid changes to UI library code. Specifically, we did not modify the implementation or underlying Java type signature of any library.

We implemented our qualifier and effect system on top of the Checker Framework [4,5], which is a framework for implementing Java 8 type annotation processors, providing support for, among other things, AST and type manipulation, and specifying library annotations separately from compiled libraries themselves. The $\mathsf{Java}_{\mathsf{UI}}$ syntax is expressed via Java 8's type annotations. Type annotations are propagated to bytecode, but have no runtime effect, thus maintaining binary API compatibility between $\mathsf{Java}_{\mathsf{UI}}$ and Java code (developers use the real UI libraries, not one-off versions coupled to $\mathsf{Java}_{\mathsf{UI}}$). The implementation consists of two main components: a core checker for the effect system, and a sizable annotation of some standard Java UI framework libraries.

Role	Annotation	Target	Purpose
Effects	@SafeEffect	Method	Marks a method as safe to run on any thread (default)
	@UIEffect	Method	Marks a method as callable only on the UI thread
	@PolyUIEffect	Method	Marks a method whose effect is polymorphic over the receiver type's effect parameter
Defaults	@UIType	Type Decl.	Changes the default method effect for a type's methods to @UIEffect
	@UIPackage	Package	Changes the default method effect for all methods in a package to @UIEffect
	@SafeType	Type Decl.	Changes the default method effect for a type's methods to @SafeEffect (useful inside a @UIPackage package)
Polymorphism	@PolyUIType	Type Decl.	Marks an effect-polymorphic type (which takes exactly one effect parameter)
Instantiating Polymorphism	@Safe	Type Use	Instantiates an effect-polymorphic type with the @SafeEffect effect (also used for monomorphic types, considered subtypes of @Safe `Object`)
	@UI	Type Use	Instantiates an effect-polymorphic type with the @UIEffect effect
	@PolyUI	Type Use	Instantiates an effect-polymorphic type with the @PolyUIEffect effect (the effect parameter of the enclosing type)

Table 1. $\mathsf{Java}_{\mathsf{UI}}$ annotations.

3.1 $\mathsf{Java}_{\mathsf{UI}}$ Basics

There are two method annotations that indicate whether or not code may access UI objects (call methods on UI objects):

- @UIEffect annotates a method that may call methods of UI objects (directly or indirectly), and must therefore run on the UI thread.
- @SafeEffect annotates a method that does *not* call methods of UI objects, and may therefore run on any thread.

@UIEffect corresponds to λ_{ui} from the λ_{UI} core language: it annotates a method as having a (potentially) UI-effectful body. Similarly, @SafeEffect corresponds to λ_{safe}.

The other annotations in Table 1 are all about changing the default effect (@UIType, @SafeType, @UIPackage) or handling polymorphism (@PolyUIEffect, @PolyUI, @Safe, @UI, @PolyUIType).

A @SafeEffect method (the default) is not permitted to call @UIEffect methods. Only @UIEffect methods (which have the UI effect) are permitted to call other @UIEffect methods. With few exceptions (e.g., `syncExec()`, which background threads may call to enqueue an action to run on the UI thread), methods of UI elements are annotated @UIEffect, and that is how improper calls to UI methods are prevented:

```
public @SafeEffect void doSomething(JLabel l) {
    l.setText(...); // ERROR: setText() has the UI effect
}
```

The other method effect annotation @UIEffect annotates a method as able to call all @UIEffect methods, and only other methods with the UI effect may call it.

```
public @UIEffect void doSomethingUI(JLabel l) {
    l.setText(...); // OK: setText() and this method have UI effect
}
public @SafeEffect void doSomethingSafe(JLabel l) {
    doSomethingUI(l); // ERROR: doSomethingUI() has the UI effect
}
```

The safe effect @SafeEffect is a subeffect of the UI effect @UIEffect. So a @SafeEffect method may override a @UIEffect method, but a @UIEffect method cannot override a @SafeEffect method:

```
public class Super { public @SafeEffect void doSomethingSafe(){ ... }}
public class Sub extends Super {
    public @UIEffect void doSomethingSafe(){ ... }}
           //^ ERROR: invalid effect override
```

Controlling Defaults Especially for subclasses of UI elements, it could be tedious to write @UIEffect on every method of a class. Three additional annotations locally change the default method effect:

- @UIType: A class (or interface) declaration annotation that makes all methods of that class, including constructors, default to the UI effect.
- @UIPackage: A package annotation that makes all types in that package behave as if they were annotated @UIType. Subpackages must be separately annotated; @UIPackage is not recursive.
- @SafeType: Like @UIType, but changes the default to @SafeEffect (useful inside @UIPackage packages).

In all three cases, individual methods may be annotated @SafeEffect or @UIEffect . This is how we have annotated SWT, Swing, and JFace: we annotated each package @UIPackage, then annotated the few safe methods (such as `repaint()`) as @SafeEffect.

3.2 Effect-Polymorphic Types

A single class may be used sometimes for UI-effectful work, and other times for work safe to run on any thread. @SafeEffect and @UIEffect do not handle this common use case. In particular, consider Java's `java.lang.Runnable` interface:

```
public interface Runnable {
    void run();
}
```

The `Runnable` interface is used to encapsulate both code that must run on the UI thread (which is passed to the `syncExec()` and `asyncExec()` methods in UI libraries), and also code that should not run on the UI thread, such as the code passed to various general dispatch queues, including thread pools.

Java$_{UI}$ provides three type qualifiers [4,5,6], each corresponding to instantiating an effect-polymorphic type with a specific effect:

- @Safe: A qualifier to instantiate a type with the @SafeEffect effect.
- @UI: A qualifier to instantiate a type with the @UIEffect effect.
- @PolyUI: A qualifier to instantiate a type with the @PolyUIEffect effect (when used inside an effect-polymorphic class).

as well as a type declaration annotation @PolyUIType that annotates a class or interface as effect-polymorphic.[2]

The final declaration for the `Runnable` interface is:

```
@PolyUIType public interface Runnable {
    @PolyUIEffect void run();
}
```

This declares `Runnable` as being a class polymorphic over one effect, and the effect of the `run()` method is that effect. Our type system implicitly adds a receiver qualifier of @PolyUI so the body can be checked for any instantiation.

Given an instance of a `Runnable`, the effect of calling the `run()` method depends on the qualifier of the `Runnable`. For example:

```
@Safe Runnable s = ...;
s.run(); // has the safe effect
@UI Runnable u = ...;
u.run(); // has the UI effect
@PolyUI Runnable p = ...;
p.run(); // has a polymorphic effect
```

Assuming the last line appears inside an effect-polymorphic type, its effect will be whatever effect the type is instantiated with. Note that the @Safe annotation on `s` is not necessary: the default qualifier for any type use is @Safe if none is explicitly written. Since most code does not interact with the UI, this means that most code requires no explicit annotations.

Effect-Monomorphic Subtypes of Effect-Polymorphic Supertypes Deriving a concrete subtype from an effect-polymorphic supertype is as simple as writing the appropriate qualifier on the supertype:

```
public class SafeRunnable implements @Safe Runnable {
    @SafeEffect void run() {
        // any effect other than safe causes a type error here
    }
}
```

[2] Java$_{UI}$ uses different annotations for monomorphic effects (@SafeEffect) and for instantiating polymorphic effects (@Safe) because of a parsing ambiguity in Java 8 where method annotations and return type annotations occupy the same source locations.

Again, note that @SafeEffect is the default effect for unannotated methods, so the use of @SafeEffect here is not strictly necessary. Inside the body of `run()`, this will have type @Safe `SafeRunnable`. In this case, the @Safe could be omitted from the `implements` clause due to defaults, but a class that implements @UI Runnable would require the explicit qualifier.

It is also possible to derive from a polymorphic type *without* instantiating, by simply declaring a polymorphic type that derives from a polymorphic instantiation of the supertype:

```
@PolyUIType public interface StoppableRunnable implements @PolyUI Runnable {
    @PolyUIEffect void run();
    @PolyUIEffect void stop();
}
```

Concrete or abstract classes may be effect-polymorphic. The body of an effect-polymorphic method is limited to calling safe methods and other methods with their same effect (e.g., other effect-polymorphic methods of the same instance, which will have the same effect as the executing method).

It is not permitted to derive a polymorphic type from a non-polymorphic type (see Section 3.4 for why). Therefore, `Object` is declared as effect-polymorphic.

Qualifier Subtyping and Subeffecting In addition to nominal subtyping (e.g., @Safe `SafeRunnable` above is a subtype of @Safe `Runnable`), Java$_{\mathsf{UI}}$ also permits qualifier subtyping, which reflects effect subtyping ("subeffecting"). For example, it may be useful to pass a @Safe `Runnable` where a @UI `Runnable` is expected: any `run()` implementation with the safe effect is certainly safe to execute where a UI effect is allowed. Similarly, a @Safe `Runnable` can be passed in place of a @PolyUI `Runnable`, which can be passed in place of a @UI `Runnable` since both subtyping relations are sound for any instantiation of @PolyUI.

Restricting Methods of Polymorphic Types On occasion, it may be desirable to make some functionality of a polymorphic type only applicable to a certain instantiation of the type. As a simple example:

```
@PolyUIType public interface AnnoyingRunnable extends @PolyUI Runnable {
    @PolyUIEffect void run(@Safe Runnable this);
}
```

The code above declares an effect-polymorphic type that may be instantiated with @UI or @Safe, but whose `run()` method may only be called on @Safe instantiations:

```
@Safe AnnoyingRunnable s = ...; s.run(); // has safe effect
@UI AnnoyingRunnable u = ...; u.run(); // type error! Receiver is not safe
```

Anonymous Inner Classes Many programs use anonymous inner classes to pass "closures" to be run on the UI thread. Qualifiers can be written on the type name. Thus code such as the following is valid (type-correct) Java$_{\mathsf{UI}}$ code:

```
asyncExec(new @UI Runnable() { public @UIEffect void run(){/* UI stuff */}});
```

If an effect-monomorphic supertype declares @UIEffect methods, no annotation is needed on the anonymous inner class, and all overriding methods default to the effect declared in the parent type, without additional annotation.

3.3 Annotating UI Libraries

To update UI elements, non-UI code uses special methods provided by the UI library that run code on the UI thread: the Java equivalent of λ_{UI}'s asyncUI$\{e\}$ construct. Real UI libraries have both synchronous and asynchronous versions:

- For Swing, these special functions are methods of `javax.swing.SwingUtilities`:
  - `static void invokeAndWait(Runnable doRun);`
  - `static void invokeLater(Runnable doRun);`
- For SWT, these special functions are methods of the class `org.eclipse.swt.widgets.Display`:
  - `static void syncExec(Runnable runnable);`
  - `static void asyncExec(Runnable runnable);`

Other UI libraries have analogous functionality. We annotated each of these library methods as @SafeEffect (safe to call on any thread) and accepting a @UI instance of the `Runnable` interface (allowing UI-effectful code in the `Runnable`). This is comparable to λ_{UI}'s T-ASYNCUI in Figure 1. Our library annotations use the Checker Framework's *stub file* support [5,4] for stating trusted annotations for code in pre-compiled JAR files. We did not check the internals of GUI libraries, which would require dependent effects (Section 3.4).

3.4 Limitations

There are a few theoretical limits to this effect system. In our evaluation (Section 4), these did not cause problems.

One Effect Parameter Per Type Java$_{UI}$ cannot describe the moral equivalent of

```
class TwoEffectParams<E1 extends Effect, E2 extends Effect> {
  @HasEffect(E1) public void m1() { ... }
  @HasEffect(E2) public void m2() { ... }
}
```

In our evaluation on over 140,000 lines of code (Section 4), this was never an issue. We found that effect-polymorphic types are typically limited to interfaces that are used essentially as closures. They are used for either safe code or UI code, rarely a mix. This restriction also gives us the benefit of allowing very simple qualifier annotations for instantiated effect-polymorphic types. Supporting multiple parameters would require a variable-arity qualifier for instantiating effects, and introduce naming of effect parameters. (We have found one instance of a *static method* requiring multiple effect parameters: `BusyIndicator.showWhile()`, discussed in Section 4.4.)

Polymorphic Types May not Extend Monomorphic Types Java_{UI} does not permit, for example, declaring a subtype `PolyUIRunnable` of `UIRunnable` that takes an effect parameter, because it further complicates subtyping. It is possible in theory to support this, but we have not found it necessary. To do so, effect instantiations of the effect-polymorphic subclass would instantiate only the *new* polymorphic methods of the subclass (polymorphic methods inherited from further up the hierarchy and instantiated by a monomorphic supertype may not be incompatibly overridden). Subtyping would then become more complex, as the qualifier of a reference could alternate almost arbitrarily during subtyping depending on the path through the subtype hierarchy.

Splitting the class hierarchy Because an effect-polymorphic type may not inherit from a monomorphic type, this forces the inheritance hierarchy into three partitions: @UI types, @Safe types, and @PolyUI types (`Object` is declared as @PolyUI, making the root of the type hierarchy @UI `Object`). All may freely reference each other, but it does impose some restrictions on code reuse. This was not an issue in our evaluation. Some classes implement multiple interfaces that each dictate methods with different effects (e.g., a listener for a UI event and a listener for background events, each handler having a different effect; Eclipse's `UIJob` interface has methods of both effects), but we found no types implementing multiple polymorphic interfaces using different instantiations.

No effect-polymorphic field types We do not allow effect-polymorphic (@PolyUI) fields. This avoids reference subtyping problems. Solutions exist (e.g., variance annotations [7]) but we have not found them necessary. Note however that we do inherit Java's unsoundness from covariant array subtyping, though we encountered no arrays of any effect-polymorphic types (or any @UI elements) during evaluation.

Cannot check UI library internals The effect system currently is not powerful enough to check the internals of a UI library, mainly because it lacks the dependent effects required to reason about the different effects in separate branches of dynamic thread checks. This means for example the effect system cannot verify the internals of safe UI methods like `repaint()`, which are typically implemented with code like:

```
if (runningOnUIThread()) { /*direct UI access*/ } else { /*use syncExec()*/ }
```

3.5 Alternatives

There are four possible approaches to handling UI threading errors: unchecked exceptions (the approach used by most GUI libraries), a sound static type and effect system, a Java checked exception (a special case of effect systems), and making every UI method internally call `syncExec()` if called from a background thread. The unchecked exception is undesirable for reasons described in the introduction: the resulting bugs are difficult to diagnose and fix. We propose a sound static type system independent from Java checked exceptions, and most of this paper explores that option. This section focuses on the two remaining options, and why our approach is different from existing concurrency analyses.

Why not make the thread access exceptions checked? Java's checked exceptions are another sound static effect system that could prevent these bugs. But there are reasons to use a separate effect system rather than simply making the thread access error exception a checked (rather than the current unchecked) exception:

- Polymorphism: Certain types, such as `Runnable`, are used for both UI thread code and non-UI code — such types are polymorphic over an effect. Java does not support types that are polymorphic over whether or not exceptions are thrown. We aim to minimize changes to existing code and to avoid code duplication.
- Reducing annotation burden: For common source code structures, such as putting most UI code in a separate package, a programmer can switch the default effect for whole types or packages at a time. Java provides no way to indicate that all methods in a type or package throw a certain checked exception. $\mathsf{Java}_{\mathsf{UI}}$ provides such shorthands.
- Backwards Compatibility: New checked exceptions breaks compilation for existing code. This is also the reason we do not leverage Java's generics support for our effect-polymorphic types.
- Catching such exceptions would almost always be a bug.

Why not have every UI method automatically use `syncExec()` if run on a background thread? This solution masks atomicity errors on UI elements. A background thread may call multiple UI methods — for example, to update a label and title bar together. Different background threads could interleave non-deterministically in this approach, creating inconsistencies in the user interface. Additionally, these atomicity bugs would hurt performance by increasing contention on the shared queue of messages from background threads due to the increased thread communication.

Why not use an existing concurrency analysis? Our effect system is different from prior type and effect systems for concurrency. Our goal is to constrain some actions to a specific distinguished thread, which is not a traditionally-studied concurrency safety property (as opposed to data races, deadlocks, and atomicity or ordering violations). In particular, this effect system permits most concurrency errors! This is by design, because preventing better-known concurrency errors is neither necessary nor sufficient to eliminate UI thread access errors, and allows the effect system design to focus on the exact error of interest. Data races on model structures are not UI errors. Because UI libraries dictate no synchronization other than the use of `syncExec()` and `asyncExec()` (and equivalents in other frameworks), deadlocks are not UI errors. It is also possible for a program to have a UI error without having any traditional concurrency bugs. $\mathsf{Java}_{\mathsf{UI}}$ only guarantees the UI library's assumption that all UI widget updates run uninterrupted (by other UI updates) in the same thread. In general, other static or dynamic concurrency analyses would complement $\mathsf{Java}_{\mathsf{UI}}$'s benefits, but the systems would not interact.

4 Evaluation

We evaluated the effectiveness of our design on 8 programs with substantial user interface components (Table 2). 4 of these programs were evaluated in prior work [2]. The others

Program	LOC	UI LOC	Classes	Methods
EclipseRunner	3101	3101	48	354
HudsonEclipse	11077	11077	74	649
S3dropbox	2353	1732	42	224
SudokuSolver	3555	3555	10	62
Eclipse Color Theme	1513	1193	48	215
LogViewer	5627	5627	117	644
JVMMonitor	31147	17657	517	2766
Subclipse	83481	53907	539	4480

Table 2. Subject programs. Pre-annotation LOC are calculated by `sloccount` [8]. UI LOC is the LOC for the main top-level package containing most of the application's UI code; other parts of a project may have received some annotation (for example, one method in a model might be executed asynchronously to trigger a UI update), and some projects were not well-separated at the package level.

were the first 4 UI-heavy Eclipse plugins we could get to compile out of the 50 most-installed[3] (as of May 2012).

We wrote trusted annotations for major user interface libraries (Swing, SWT, and an SWT extension called JFace), annotated the programs, and categorized the resulting type-checking warnings. Where false warnings were issued due to the type system being conservative, we describe whether there are natural extensions to the type system that could handle those use cases.

4.1 Annotation Approach

Trusted Library Annotations We conservatively annotated the UI libraries used by subject programs before annotating the programs themselves. Swing contains 1714 classes, SWT contains 708 classes, and JFace 537 classes. We erred on the side of giving too many methods the UI effect, and we adjusted our annotations later if we found them to be overly conservative. We intermingled revisions to library annotations with subject program annotation, guided by the compiler warnings (type errors). We examined the documentation, and in some cases the source, for every library method that caused a warning. When appropriate, we annotated library methods as @SafeEffect, annotated polymorphic types and effects for some interfaces, and changed some UI methods to accept @UI instantiations of effect-polymorphic types. The annotated library surface is quite large: we annotated 160 library packages as @UIPackage, as well as specifying non-default effects for several dozen classes (8 effect-polymorphic).

Our results are sound up to the correctness of our library annotations and the type checker itself. We can only claim the UI framework annotations to be as accurate as our reading of documentation and source. A few dozen times we annotated "getter" methods that returned a field value as safe when it was not perfectly clear they were *intended* as safe. They were widely called in safe contexts by subjects and example code without issue. There are three primary sources of potential unsoundness in library annotations:

[3] http://marketplace.eclipse.org/metrics/installs/last30days

1. Incorrectly annotating a method that does perform some UI effect as safe.
2. Incorrectly annotating a method that requires a safe variant of a polymorphic type as accepting a UI variant.
3. Incorrectly annotating a callback method invoked by a library as @UIEffect .

To mitigate the first source, we began the process by annotating every UI-related package and subpackage we could find as @UIPackage. Java$_{UI}$ mitigates the second by the fact that unspecified polymorphic variants default to safe variants. We addressed the third by reading the documentation on the several ways the UI frameworks start background threads, and annotating the relevant classes correctly early on.

Annotating Subject Programs To annotate each subject program, we worked through the files with the most warnings first. We frequently annotated a class @UIType if most methods in the class produced warnings; otherwise we annotated individual methods with warnings @UIEffect . For files with fewer warnings, we determined by manual code inspection and perusing UI library documentation whether some methods came from an interface with UI-effectful methods, annotating them @UIEffect if needed.

In an effort to make the final warning count more closely match the number of possible conceptual mistakes, when the body of a method that must be safe (due to its use or inheritance) contained one basic block calling multiple @UIEffect methods (e.g. `myJLabel.setText(myJLabel.getText()+"...")`), we annotated the method body @UIEffect, taking 1 warning over possibly multiple warnings about parts of the method body. We believe this makes the final warning counts correspond better to conceptual errors. Multiple UI method calls in a safe context likely reflects 1 missing `asyncExec()` (a developer misunderstanding the calling context for a method) or 1 misunderstanding on our part of the contexts in which a method is called, not multiple bugs or misunderstandings. If multiple separated (different control flow paths) basic blocks called @UIEffect methods, we left the method annotation as @SafeEffect.

We made no effort during annotation to optimize annotation counts (the counts we do have may include annotations that could be removed). We simply worked through the projects as any developer might.

We identified several patterns in library uses that cause imprecision; we discuss those in Section 4.4.

Distinguishing warnings that correspond to program bugs, incorrect (or inadequate) annotations, tool bugs, or false positives requires understanding two things: the semantics of Java$_{UI}$'s effect system, and the intended threading design of the program (which code runs on which thread). The user has detected a program bug or a bad annotation when JavaUI indicates that the program annotations are not consistent with the actual program behavior. Finding the root cause may require the user to map the warning to a call path from a safe method to a UI method. This is similar to other context- and flow-sensitive static analyses, and when the user does not understand the program, iteratively running JavaUI can help. The user can recognize a false positive by understanding what is expressible in JavaUI. A reasonable rule of thumb is that if a warning could be suppressed by conditioning an effect by a value or method call whose result depends on the thread it runs on, it is likely a false positive. The user can recognize a tool bug in the same way, by understanding JavaUI's simple rules.

	Zhang et al. [2]				Defects			
Program	Warnings	Defects	Time to Annotate	Warnings	UI	Other	False Pos.	Other
EclipseRunner	6	1	<1hr	1	1	0	0	0
HudsonEclipse	3	3*	<1hr	13**	3	0	2	0
S3dropbox	1	1	<1hr	2	2	0	0	0
SudokuSolver	2	2	<1hr	2	2	0	0	0
Eclipse Color Theme	0	0	8m	0	0	0	0	0
LogViewer	0	0	3h50m	1	0	0	1	0
JVMMonitor	7	0	6h45m	9	0	0	9	0
Subclipse	24	0	17h20m	19	0	1	13	5

Table 3. Java$_{UI}$ warnings (type errors). Java$_{UI}$ finds all bugs found by Zhang's technique [2], plus one additional bug. The table indicates UI threading defects, non-exploitable code defects found because of annotation, definite false positives, and a separate category for other reports, which includes reports we could not definitively label as defects or false positives, as well as other warnings such as poor interactions between plugins' Java 1.4-style code and the 1.7-based compiler and JDK we used.

*Zhang et al. report 1 bug, but its repair requires adding `syncExec()` in 3 locations, so we consider it 3 bugs.
**Java$_{UI}$ found the same 3 bugs as Zhang et al.'s GUI Error Detector, each with 3–4 warnings due to compound statements.

Program	@UIPackage	@UIType	@SafeType	@UIEffect	@SafeEffect	@UI	@Safe	Anno/KLOC
EclipseRunner	0	26	0	2	5	0	0	10.6
HudsonEclipse	0	17	0	9	4	14	0	3.9
S3dropbox	0	30	0	4	2	14	0	21.2
SudokuSolver	0	2	0	18	1	9	0	8.4
Eclipse Color Theme	0	1	0	3	0	0	0	2.6
LogViewer	0	53	0	5	23	15	1	17.2
JVMMonitor	0	129	0	12	47	29	0	6.9
Subclipse	17	126	60	102	128	138	0	6.8

Table 4. Annotation statistics for the 8 subject programs. @PolyUIEffect, @PolyUIType, and @PolyUI were not used in the subject programs themselves — only in annotating the UI libraries.

4.2 Study Results: Bugs and Annotations

Annotating these projects taught us a lot about interesting idioms adopted at scale, and about the limitations of our current effect system. The annotation results appear in Table 3, including the final warning counts and classifications, and Table 4 gives the number of each annotation used for each project. The first four projects each took under an hour[4] to annotate, by the process described in Section 4.1. Eclipse Color Theme took only 8 minutes to annotate, and required only 4 annotations. The effort required for these five projects was low even though we had never seen the code before starting annotation. The other projects (LogViewer, JVMMonitor, and Subclipse) were substantially larger and more complex, and they required substantially more effort to annotate.

Overall we found all known bugs in the subject programs (only the first four were known to have UI threading related errors [2]), plus one new UI threading defect, and one defect in unreachable (dead) code.

Table 3 includes the error counts for Zhang et al.'s unsound GUI Error Detector [2] when run on the same program versions. We found all bugs they identified, as well as

[4] We lack precise timing information because each annotation was interleaved with fixing Java$_{UI}$ implementation bugs.

1 new bug in S3dropbox. The S3dropbox developer has confirmed the bug, though he does not plan to fix it because it does not crash the program (Swing does not check the current thread in every UI method, allowing some races on UI objects). If a user drops a file using drag-and-drop onto the UI, the interface sometimes forks a background thread that then calls back into UI code. GUI Error Detector misses this bug because of an unsoundness in WALA [9], which GUI Error Detector uses for call graph extraction. For scalability, by default WALA does not analyze certain libraries, including Swing. GUI Error Detector uses WALA's default settings, so the drag-and-drop handler appears (to the tool) to be unreachable. Call graph construction precision is also a bottleneck for GUI Error Detector on large programs: Subclipse analysis required a less precise control flow analysis to finish (0-CFA, others used 1-CFA).

The dead code defect we found was a UI-effectful implementation of a safe interface in Subclipse. The type with the invalid override was never used at that interface type (removing the `implements` clause fixes the warning) and the method was never called. We consider this a defect, though it is not exploitable.

In Table 3 the number of final warnings exceeding the number of bugs found does not necessarily indicate false positives: our type system issues a warning for every type-incorrect *expression* that could correspond to a thread access error. So a single line with a composite UI expression (e.g., a UI method call with UI expressions as arguments) in a non-UI context would (correctly) produce multiple compiler warnings. Each subexpression may have an individual work-around that does not require adding an `(a)syncExec()`. We consider a warning to be a false positive only if the target expression's execution in context would not improperly access UI elements from a non-UI thread.

Our sound type and effect system found no new UI threading errors in the larger projects, but found several bugs in the smaller projects. There are good reasons to expect this result. First, the first four projects were previously evaluated by Zhang et al. [2], so we knew we should find bugs in those projects (Zhang selected several of those subjects by finding SWT threading errors in bug databases). For the larger projects, we simply took the most recent release version of each Eclipse plugin. Second, the larger Eclipse plugins are all mature, heavily used projects, making it quite likely that any UI bugs would be found and fixed quickly (each has at least 3,000 installs total; Subclipse was installed 23,855 times between 11/19 and 12/19 2012 alone). We believe $\mathsf{Java}_{\mathsf{UI}}$ would have more benefits when used from the beginning of a project, and the relative prevalence of UI errors in the younger projects compared to the mature projects supports this theory.

After annotating these programs, we searched all four projects' issue trackers for threading-related issues. There were several bug reports for data races between models and views, and issues with several UI methods that behave differently on the UI thread than on other threads; the latter are not uncommon, because many JFace methods return one result (often null) on non-UI threads, but a different result on the UI thread. The latter could have been caught by our type system with custom library annotations for those methods.

The one report we found of a UI threading error [10] is triggered when a background thread calls into a native method, which then calls back into UI-effectful Java code. The call occurs as a result of a logic error in Subversion itself, and the bug was marked

"WONTFIX" (the fix was a patch to Subversion). With a proper annotation on the native methods, our effect system would have issued a warning.

Non-annotation Changes We made minor changes to the code beyond simply adding effect-related annotations (and the required `import` statements) for two reasons: when naming an anonymous inner class as a new subtype would fix effect (type) errors, and when converting Java 1.4-style code to use generics would remove warnings. The `Action` interface is generally used as a closure for @UIEffect work, but HudsonEclipse in one case made an anonymous inner class whose `run()` method was safe, stored it as an `Action` (whose `run()` is @UIEffect), and called the `run()` of the safe subclass explicitly in several safe contexts. Rather than making `Action` (and several supertypes) polymorphic, which seems to contradict the suggested uses in the documentation, we declared a subclass of `Action` that overrode `run()` as @SafeEffect, and stored the anonymous inner class as an instance of that (removing 3 warnings). In JVMMonitor, Java 1.4-style (no generics) use of Java Beans interacted poorly with the Checker Framework's promotion of an argument of type `Class` to `Class<? extends @Safe Object>`, which was passed as an argument to a Java Bean method accepting a `Class<@Safe Object>`. We added a generic method parameter `T` and changed the argument to `Class<T>`, removing one warning. In Subclipse, we converted 3 `Vector` fields to `Vector<String>`, due to a similar problem with `Vector.copyInto`, removing 3 warnings.

We allowed these changes because they were minor, and because we expect they would be natural changes for a project interested in using our effect system to verify the absence of UI threading errors. Most of the remaining false positives could be fixed with more engineering work (e.g. splitting interfaces, splitting callback registration into safe and UI-effectful, etc.) but we judged such changes to be too invasive to give a clear picture of developer effort, and restricted ourselves to only these small changes.

4.3 False Positives and Other Reports

Our evaluation produced 30 false positives in over 140,000 lines of code (0.2 per 1000 lines of code). We consider this an acceptable false positive rate. The false positives fall into 5 general categories, including limitations of our type system and what we consider to be poor designs in the UI libraries and client programs.

Registering Callbacks of Both Effects Four of the projects shared a common source of false positives: HudsonEclipse (1 false positive), LogViewer (1), JVMMonitor (5), and Subclipse (1) each suffered imprecision from JFace's global property store. The plugin code adds a UI-effectful property change listener (a @UI instantiation of a polymorphic interface) to JFace's global property store. Listeners will fire on any thread that sets a property, so in general the listeners must be safe. However, in some projects all calls to setting properties are performed from within the UI thread, making this not a bug. A potential solution for this class of false positives is to change the library annotations for JFace's preference store to permit @UI handlers, and to annotate the property setters @UIEffect in a project-specific library annotation file used to override the global file. (The global file would follow documentation as much as possible.) In other cases, particular

Client code listener (callback) implementations:

```
class SafePropertyChangeListener extends PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent event) {
        if (event.getProperty().equals("stuff")) {
            // do @SafeEffect stuff
}}}
class UIPropertyChangeListener extends PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent event) {
        if (event.getProperty().equals("uistuff")) {
            // Call @UIEffect methods directly
}}}
```

On UI thread:

```
store.addPropertyChangeListener(new UIPropertyChangeListener());
```

On a background thread:

```
store.addPropertyChangeListener(new BackgroundPropertyChangeListener());
// Next line executes UIPropertyChangeListener's UI callback on BG thread
store.setValue("stuff", true);
```

Fig. 4. JFace global property store issues. Assume `store` is any expression that accesses a shared static JFace `PreferenceStore`; in Eclipse plugins, there is one such store initialized for every plugin. Listeners for property changes are registered with both possible effects, but all handlers will run on any thread that updates any property, making UI thread errors possible. As long as specific properties that actually cause @UIEffect methods to be called (in this case, `uistuff`) are only updated from the UI thread, no errors will occur, but this pattern is fragile.

properties are only updated from the UI thread, and the UI-effectful handlers are guarded by condition checks that only pass for those UI properties, as in Figure 4.

JVMMonitor created its own specific instance of this same problem: the other 4 of its false positives are from a property store it creates for CPU model changes, where some properties are only updated from UI code, and the handlers have UI effects but run only for specific properties. We feel this property store design is faulty, which we elaborate on in Section 4.4.

Subtyping Limitations 5 other false positives (1 in HudsonEclipse and 4 in Subclipse) are from a weakness in our subtyping relation. HudsonEclipse's subtyping false positive occurs because a @UI instantiation of a polymorphic type is not a subtype of @Safe `Object`. The subtyping false positives in Subclipse are from a combination of that with the requirement that generic parameters are subtypes of the default Object variant: the code uses a Java 1.4 style list, but `List<T>` is implicitly `List<T extends` @Safe `Object>` which makes some types unusable as type parameters. These uses would be enabled by making the upper bound on `List` (and `Iterator`) @UI `Object`, but doing so would force many additional annotations where an object was pulled out of a list or iterator as (implicitly safe) `Object`, so we opted for the lower annotation burden. This would not be an issue in properly generic code.

Interface Abuse One false positive in Subclipse is an instance of *interface abuse*: subclassing a definitively safe supertype using @UIEffect overrides, then calling the @UIEffect overrides directly, only from UI contexts. We could have fixed this type error by making the entire hierarchy of the abusing class's superclasses effect-polymorphic (from documentation one superclass is clearly intended as safe), or by introducing a new type and copying code from parent classes (which is poor design for its own reasons).

Lack of Dependent Types/Effects Subclipse had 7 additional definite false positives, most of which would require dependent effects (as in dependent types; an effect determined by a runtime result) to handle:

- 3 warnings resulting from lack of dependent effects (see Section 4.5).
- 3 warnings that were subject to some type of dynamic thread check, and would therefore have executed only on the UI thread
- 1 instance where our type system could not express the proper effect (it would require a combination of dependent effects with multiple method effect parameters and explicit least-upper-bound-of-parameters effects)

Other Reports The remaining 5 Subclipse reports are about unsafe effects, but we cannot determine whether or not the application is using a safe interface as polymorphic (at a @UI variant) or if the JFace interface, documented as unrelated to interfaces or threads, should actually just allow UI effects.

4.4 Sources of Difficulty

Weak Documentation The main source of difficulty in annotating all of our subject programs was understanding the design of the UI libraries, with respect to which methods must only be called in the UI thread, or were polymorphic. Once we understood the design, adding library annotations was easy. Remarkably, none of the UI libraries clearly and consistently documents the thread safety of all UI methods. $\mathsf{Java}_{\mathsf{UI}}$'s annotations are precise documentation, and are machine-checkable for client code.

AWT and Swing's documentation was concise and unambiguous: all methods except `invokeAndWait` and `invokeLater` must be called only from the UI thread. SWT claims the same about `syncExec()` and `asyncExec()`. Confusingly, there are some exceptions to this rule in SWT (classes `Color`, `Font`, and `Image`).

The prevailing wisdom about JFace is that most of JFace assumes it is running on the UI thread. But clients call much of JFace directly from non-UI threads, and the JFace documentation rarely specifies thread assumptions. Clients often interpret the lack of documentation as license to call: there are many methods of UI elements that documentation suggests are intended to be called only from the UI thread, but happen to be safe (such as getter methods) and are therefore often called by clients from contexts that must be safe.

Problematic Idioms There are several idioms that cause problems for our type system. Most could be handled with richer polymorphism or dependent effects, but we believe most of these idioms are poor design. Rewriting the offending code is a better option, and $\mathsf{Java}_{\mathsf{UI}}$'s type system encourages this better design.

The most common source of false positives was JFace's global property store design. JFace often shares global property sets among all threads, and listeners can be registered for a callback in the event of a property changing. These listener callbacks will be executed on whichever thread updates a property, thus all properties callbacks should be @SafeEffect. However, some programs register @UIEffect callbacks, but avoid issues by updating the global property store (for any property) only from the UI thread, or updating the properties with UI-effectful handlers only from the UI thread. Two of these safe approaches can be handled using custom library annotations for individual projects: either callbacks can have the UI effect and properties may only be updated from the UI thread, or callbacks must be safe and the properties may be updated by any thread. The related case as seen with the CPU model change listeners in JVMMonitor could be handled with some dependent effects, annotating the listener update code with the properties whose handlers may have UI effects, and allowing updates to those properties only from UI contexts. The shared property store design appears to us to be a very error-prone design; we would prefer separate property stores or listener registrations for handlers that run on or off the UI thread.

Another problematic idiom, seen only in Subclipse, is code that dynamically checks which thread it is executing on, and optionally redirects a closure to the UI thread if necessary. Our design does not support these dynamic checks, which are typically found only inside UI libraries themselves. In Subclipse, SWT's `Display.getCurrent()` is used; it returns `null` when executed off the UI thread, and otherwise returns a valid `Display` object. The same method also sets a local boolean indicating whether `Display.getCurrent()` returned `null`, so handling this code is not a simple matter of specializing to a particular `if-then-else` construct.

An idiom responsible for both a number of false reports and for a number of workarounds in our library annotations is to make ad-hoc polymorphic instances of a particular type. By this we mean creating a subtype with a UI-effectful override of a safe method, storing the reference at the (safe) supertype, but only allowing said values to flow to and be used in UI contexts. This frequently occurred with a JDK or Eclipse interface that appeared from documentation to be intended as safe, but some part of JFace or a custom design by a plugin developer co-opted it and treated it as an effect-polymorphic type. Similarly, developers sometimes implement a totally safe subclass of a UI-effectful supertype (often as an anonymous inner class), store references as the supertype, and call methods of said class in safe contexts (only allowing safe subtype instances to flow to those call sites). Checking these uses in general would effectively require allowing every type to take a separate effect parameter for each method. In our evaluation, we typically annotated such classes as effect-polymorphic, annotated each method of those classes as @PolyUIEffect, and added UI instances to the subject programs as necessary.[5] This generally sufficed for UI variants of safe supertypes, unless the problematic class would then inherit from multiple polymorphic types, which our system does not handle. The latter case (safe variants of a UI interface) could be worked around by explicitly introducing a named subtype that declares a safe override, and storing references as the safe subtype (which we did once for HudsonEclipse). We speculate that these designs

[5] This is always sound: each instantiation is treated soundly, and @Safe instances may flow into variables for @UI instances.

arise from the designers of the relevant class hierarchies and interfaces not considering the option of using type hierarchies to separate code that must run on the UI thread from code that may run anywhere (including hijacking otherwise safe interfaces). The type hierarchy splitting that fits these cases into our effect system generally seems less error-prone even without our strict effect system checking, because such splitting already introduces some type-based barriers to confusing the calling context of UI code.

A class of idioms that definitely indicates shortcomings of our current effect system is methods requiring qualifier-dependent method effects: effects that depend on the qualifier of one or more effect-polymorphic arguments. There are also some methods whose proper types require a much richer type system. A good example of this is `org.eclipse.swt.custom.BusyIndicator.showWhile()`, whose effect depends on:

- The variant of `Runnable` it receives (@UI or @Safe),
- The calling context (@UIEffect or @SafeEffect), and
- Whether the `Display` it receives is `null`

This method calls the passed `Runnable` on the current thread, displaying a busy cursor on the passed Display if any, and no busy information otherwise. Because it is often used for UI-effectful work, we annotated it @UIEffect, taking a @UI `Runnable`. A better, but still conservative type would be

```
@PolyUIEffect
public static void showWhile(Display display, @PolyUI Runnable runnable);
```

Our type system cannot check calls to this presently because there is no receiver qualifier to tie to the effect at the call site. Checking the method internals would require richer types, including effect refinements based on the `Display`.

4.5 Potential Type System Extensions

Our experiences revealed several ways our type system could be extended to verify more client code.

First, what would be simple polymorphism in the most natural core calculus to write for the polymorphic effect system[6] becomes lightweight dependent effects in the implementation. This happens in cases where a method takes a Runnable of some instantiation and runs it in the current thread. The example we encountered in our evaluation is the `showWhile()` method from the previous section, a good signature for which would be

```
@EffectFromQualOf("runnable")
public static void showWhile(Display display, @PolyUI Runnable runnable);
```

The effect of this runner call will be whatever the effect of the `Runnable`'s `run()` method is. If we could use the same annotation for both qualifiers and method effects, simply specifying the polymorphic qualifier would be sufficient, and the Checker Framework's existing support for polymorphism would handle this. Unfortunately, Java 8's type annotation syntax leaves parsing ambiguities: if @UI applied to both types and methods,

[6] Recall that λ_{UI} is effect-*monomorphic*.

there would be no way to disambiguate a use as a method effect annotation from a use as a method return type annotation. Thus we must use different annotations. So to support this type of qualifier-dependent effect, we would need lightweight dependent effects, simply due to limitations of Java's grammar.

A need for dependent effects comes from a pattern seen in Subclipse, where methods either take a boolean argument (`ProgressMonitorDialog.run()` and `Activator.showErrorDialog()`) or call a related method (`Action.canRunAsJob()`) indicating whether or not to fork another thread to execute a Runnable (otherwise the effect of the runner is polymorphic as in the previous example). If the fork flag is true, the provided Runnable must be a @Safe Runnable. So for example, the effect of the main work method for the `Action` class should be (informally):

```
@EffectIfTrueElse(this.canRunAsJob(),@SafeEffect,@UIEffect)
```

indicating that the method must be safe if the method is required to run as a `Job` (a background thread task), and otherwise will run on the UI thread. Checking this then requires executing the `canRunAsJob()` method during type-checking, and ensuring that this computation is independent of subtyping (that the `canRunAsJob()` implementation is `final` when used). Other utility methods take a flag with opposite polarity, but the required type system extension would be the same.

5 Related Work

The most similar work to ours is an open source tool, CheckThread.[7] It is a Java compiler plugin that aims to catch arbitrary concurrency bugs, and includes a `@ThreadConfined("`*`threadName`*`")` annotation similar to our @UIEffect, but supporting arbitrary thread confinement rather than being limited to one distinguished thread. It appears to be an effect system, but the authors never describe it as such in the documentation or code. They allow applying various annotations to whole types as a shorthand for applying an annotation to all methods on a type (similar to our @SafeType and @UIType). However, their system does not support polymorphism, and it is not clear from its documentation if it treats inheritance soundly (from source inspection, it appears not).

Another piece of closely related work is Zhang et al.'s GUI Error Detector [2], which searches for the same errors as our type and effect system via a control flow analysis. Essentially they extract a static call graph of a program from bytecode, and find any call path that begins in non-UI code and reaches a UI method without being "interrupted" by a call to `syncExec()` or `asyncExec()`. The false positive rate from the naïve approach is too high, so they couple this with several heuristics (some unsound, potentially removing correct warnings) to reduce the number of warnings. On four examples from their evaluation, we find all of the bugs they located, plus one more (Section 4.2). They also annotated 19 subject program methods in their evaluation as trusted safe when they performed dynamic thread checks; at least 3 of our false positives would disappear if we suppressed warnings in such methods. Another interesting result of our work is empirical confirmation of the GUI Error Detector's unsound heuristics for filtering reports. GUI Error Detector found most of the bugs our sound approach identified, and the missed bug was due to WALA's

[7] http://checkthread.org/

defaults for scalable call-graph generation, not due to unsound heuristics. Some idioms, like the global property store design, cause false positives for both techniques.

Our approach has several advantages over Zhang et al.'s approach. Most notably, our technique is sound, while their heuristics may filter out true reports. Our type and effect system is also modular and incremental (we naturally support separate compilation and development) and can reason soundly about code (for example, subclasses) that may not exist yet. The GUI Error Detector on the other hand must have access to all JAR files that would be used by the running application in order to gather a complete call graph, and must rerun its analysis from scratch on new versions. If JARs are unavailable, its unsoundness increases. For performance reasons, their underlying call graph extraction tool (WALA [9]) skips some well-known large libraries — including Swing, meaning that GUI Error Detector misses all callbacks from Swing library code into application code. This results in additional unsoundness, and is the reason GUI Error Detector did not find the drag-and-drop bug in S3Dropbox. Our support for polymorphism is also important: 14 of GUI Error Detector's 24 false positives on Subclipse were because Zhang et al.'s technique does not treat the `Runnable` interface effect-polymorphically.

Our system does have several disadvantages compared to Zhang et al.'s. Our type and effect system requires manual code annotation, requiring many hours for some large projects (though this effort is only required once, with only incremental changes to annotations as the program evolves). Because GUI Error Detector runs on Java bytecode, it is possible to run the GUI Error Detector on binaries for which source is unavailable, without writing a trusted stub file. It is also possible (though untested) that their approach can handle other JVM languages (such as Scala or Clojure) or multi-language programs. Our type system produces only localized reports; Zhang et al. examine the whole call graph, so a warning's report includes a full potentially-erroneous call sequence. When annotating the sample programs in our evaluation, much of our time was spent manually reconstructing this information.

Sutherland and Scherlis proposed the technique of *thread coloring* [11], which is in some ways a generalization of our UI effect. They permit declaration of arbitrary thread roles and enforcing that methods for certain roles execute only on the thread(s) holding those roles. They use a complex combination of abstract interpretation, type inference, and call-graph reconstruction to reduce annotation burden; the one subject they specify annotation burden for is lower than ours, but they do not provide annotation counts for other subjects. They do not describe their false positive rates. They do annotate AWT and Swing applications successfully; we found AWT and Swing had relatively consistent policies, and expect they would have had more difficulty with Eclipse's SWT and JFace libraries, which were the source of many of our false positives. Like Zhang et al.'s technique, Sutherland's implementation lacks role (effect) polymorphism, which results in an unspecified number of false positives in their evaluation.

There is a long line of work on effect systems, ranging from basic designs [12] to abstract effect systems designed for flexible instantiation [13,14]. Rytz et al. [14] propose an effect-polymorphic function type for reasoning about effects, where for functions of type $T_1 \stackrel{e}{\Rightarrow} T_2$ the effect of invoking it is the join of the concrete effect e with the effect of the argument T_1, implying that the function may call T_1 (if the argument is itself a function). This is similar to what we would need to support qualifier-dependent

effects (e.g., `showWhile` in Section 4.5). Other effect systems are designed to reason about more general safe concurrency properties [15,16], but we are the first to build a polymorphic effect system for the issue of safe UI updates. Another approach would be to encapsulate UI actions within a monad, since every effect system gives rise to a monad [17]. Functional languages such as Haskell use monads to encapsulate many effects, and in fact some Haskell UI libraries use a UI monad to package UI updates safely (e.g., Phooey [18]). Another use of monads in functional languages is to support software transactional memory [19,20], including a strong separation between data accessed inside or outside a transaction. Viewing the closures run on the UI thread as *UI transactions*, our type system enforces a weakly atomic [21] form of transactions, where UI elements are guaranteed to be transaction-only but non-UI elements have no atomicity guarantees.

6 Conclusion

In almost every UI framework, it is an error for a background thread to directly access UI elements. This error is pervasive and severe in practice. We have developed an approach — a type and effect system — for preventing these errors that is both theoretically sound and practical for real-world use. We have proven soundness for a core calculus λ_{UI}. Our implementation, $\mathsf{Java}_{\mathsf{UI}}$, is both precise and effective: in 8 projects totalling over 140,000 LOC, $\mathsf{Java}_{\mathsf{UI}}$ found all known bugs with only 30 spurious warnings, for a modest effort of 7.4 annotations per 1000 LOC on average.

We have identified error-prone coding idioms that are common in practice and explained how to avoid them. We also identified application patterns that $\mathsf{Java}_{\mathsf{UI}}$ cannot type check that will probably be issues for other effect systems applied to existing code: ad-hoc effect polymorphism, value-dependent effects, and data structures mixing callbacks with different effects. These idioms suggest improvements to existing code (such as segregating callbacks with different known effects) and profitable extensions to effect systems.

Acknowledgements We thank the anonymous referees for their helpful comments, and Sai Zhang for helpful discussions and running GUI Error Detector on additional subject programs for us. This work was funded in part by NSF grant CNS-0855252 and DARPA contracts FA8750-12-C-0174 and FA8750-12-2-0107.

References

1. Ingalls, D.H.H.: The Smalltalk-76 Programming System: Design and Implementation. In: POPL. (1978)
2. Zhang, S., Lü, H., Ernst, M.D.: Finding Errors in Multithreaded GUI Applications. In: ISSTA. (2012)
3. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Inf. Comput. **115**(1) (November 1994) 38–94
4. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical Pluggable Types for Java. In: ISSTA. (2008)
5. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.: Building and Using Pluggable Type-Checkers. In: ICSE. (2011)
6. Foster, J.S., Fähndrich, M., Aiken, A.: A Theory of Type Qualifiers. In: PLDI. (1999)

7. Emir, B., Kennedy, A., Russo, C., Yu, D.: Variance and Generalized Constraints for $C^\sharp$ Generics. In: ECOOP. (2006)
8. David A. Wheeler's SLOCCount `http://www.dwheeler.com/sloccount/`.
9. WALA `http://wala.sourceforge.net`.
10. Subclipse Issue 889 `http://subclipse.tigris.org/issues/show_bug.cgi?id=889`.
11. Sutherland, D.F., Scherlis, W.L.: Composable Thread Coloring. In: PPoPP. (2010)
12. Lucassen, J.M., Gifford, D.K.: Polymorphic Effect Systems. In: POPL. (1988)
13. Marino, D., Millstein, T.: A Generic Type-and-Effect System. In: TLDI. (2009)
14. Rytz, L., Odersky, M., Haller, P.: Lightweight Polymorphic Effects. In: ECOOP. (2012)
15. Bocchino, Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A Type and Effect System for Deterministic Parallel Java. In: OOPSLA. (2009)
16. Kawaguchi, M., Rondon, P., Bakst, A., Jhala, R.: Deterministic Parallelism via Liquid Effects. In: PLDI. (2012)
17. Wadler, P.: The Marriage of Effects and Monads. In: ICFP. (1998)
18. Phooey UI Framework `http://www.haskell.org/haskellwiki/Phooey`.
19. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC. (2003)
20. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP. (2005)
21. Blundell, C., Lewis, E., Martin, M.: Subtleties of Transactional Memory Atomicity Semantics. Computer Architecture Letters **5**(2) (2006)

A Notes

A.1 Ambiguous Type Qualifiers

Java 8 syntax does not fully support using the same annotation as both a type qualifier and a method annotation, because the location for return type qualifiers is the same as the location for method annotations (likely because Java 8 was designed with type annotations rather than effect annotations in mind; for systems with only data type qualifiers this poses no issue). For example, if @PolyUIEffect were both a type annotation and a method annotation, there are three natural ways the following method signature might parse:

```
@PolyUIEffect Runnable m();
```

a An implicitly safe method returning an explicitly polymorphic `Runnable`?
b An explicitly polymorphic effect method returning an implicitly safe `Runnable`?
c An explicitly polymorphic effect method returning an explicitly polymorphic `Runnable` (i.e., the annotation applies to both the method *and* the return type)?

Java 8 chose c. Adding additional annotations does not help: in that case, both annotations will apply both places, making for an invalid return type and an invalid method effect. As a result, we split the type qualifiers from their associated effects.

A.2 Anonymous Inner Classes

Instead of closures, Java (prior to Java 8) provides anonymous inner classes to instantiate interfaces without declaring a new type. Unfortunately, even in Java 8, Java syntax provides fewer distinct annotation locations for anonymous inner classes than top level class declarations. For example, we would like to write code such as the following:

```
...
   asyncExec(new @UI Runnable {
      @UIEffect public void run() {
         // do UI stuff
      }
   });
...
```

We would hope that the code above would be elaborated as roughly:

```
class $1 extends @UI Runnable {
    ...
    @UIEffect public void run() { ... }
    ...
}
...
    asyncExec(new $1(...));
...
```

Unfortunately it parses, due to defaults, as roughly:

```
class @UI $1 extends @Safe Runnable { // due to default annotations
    ...
    @UIEffect public void run() { ... }
    ...
}
...
    asyncExec((@UI $1)new $1(...));
...
```

This is part of the reason we treat @UI on class declarations as implying @UIType. In the implementation, we special case anonymous inner classes to copy declaration qualifiers onto the supertype.

B A Semi-formal $\mathsf{Java}_{\mathsf{UI}}$

To make the relationship between qualifiers and effects, and the subtyping relation clearer, we present a semi-formal model of type validity and inheritance for $\mathsf{Java}_{\mathsf{UI}}$, solely for precise exposition of the relationship between qualifiers and effect parameters.

$$
\begin{aligned}
q &::= \mathsf{@Safe} \mid \mathsf{@UI} \mid \mathsf{@PolyUI} \\
\xi &::= \mathsf{@SafeEffect} \mid \mathsf{@UIEffect} \mid \mathsf{@PolyUIEffect} \\
T &::= C\langle \overline{\xi} \rangle \\
\tau &::= q\, T
\end{aligned}
$$

We do not require a class of identifiers for polymorphic qualifier or effect names because in any context, there will be at most one such parameter in scope. This does impose restrictions on code that might otherwise be highly polymorphic, but we have yet to see any code that requires multiple effect parameters.

There is a mapping relation $\approx \in q \times \xi$ associating qualifiers with the related effect, and an ordering $\sqsubseteq \in \xi \times \xi$ relation on effects:

$$\frac{}{\mathsf{@UI} \approx \mathsf{@UIEffect}} \qquad \frac{}{\mathsf{@Safe} \approx \mathsf{@SafeEffect}} \qquad \frac{}{\mathsf{@PolyUI} \approx \mathsf{@PolyUIEffect}}$$

$$\frac{}{\mathsf{@SafeEffect} \sqsubseteq \mathsf{@PolyUIEffect}} \qquad \frac{}{\mathsf{@PolyUIEffect} \sqsubseteq \mathsf{@UIEffect}} \qquad \frac{}{\xi \sqsubseteq \xi}$$

$$\frac{\xi \sqsubseteq \xi' \qquad \xi' \sqsubseteq \xi''}{\xi \sqsubseteq \xi''}$$

These relations impose a subtyping relation among qualifiers, in addition to the normal reflexive transitive nominal-inheritance-based subtyping:

$$\frac{}{\mathsf{@Safe}\ C \prec \mathsf{@PolyUI}\ C} \qquad \frac{}{\mathsf{@PolyUI}\ C \prec \mathsf{@UI}\ C}$$

For a type to be well-formed, its qualifier must match its effect parameter:

T-WF
$$\frac{\mathsf{class}\ q\ C\ \mathsf{extends}\ q'\ D\{\_;\_\} \in P \qquad q'' = q \vee q = \mathsf{@PolyUI} \qquad \vdash q'\ D \qquad q'' \approx \xi}{\vdash q''\ C\langle\xi\rangle}$$

T-WF-OBJECT
$$\frac{}{\vdash q\ \mathsf{Object}\langle q\rangle}$$

For this reason, and because we only permit one effect parameter for each type, we do not actually require the effect parameter ($\langle\xi\rangle$) to be syntactically present in the source implementation. Instead we use Java 8's type qualifier support and use the qualifier to set the single effect parameter.

Type declarations must also be valid. A type declaration is only valid if its qualifier matches the declared qualifier of its supertype, or if it instantiates a generic parent:

TDECL-WF
$$\frac{\vdash q'\ D \qquad q = q' \vee q' = \mathsf{@PolyUI} \qquad \overline{C \vdash mdecl} \qquad \overline{\vdash fdecl}}{\vdash \mathsf{class}\ q\ C\ \mathsf{extends}\ q'\ D\{\overline{fdecl};\overline{mdecl}\}}$$

MDECL-WF
$$\frac{q\ C \in P \qquad \overline{\vdash \tau_a} \qquad \vdash \tau_{ret} \qquad \xi = \mathsf{@PolyUIEffect} \iff q = \mathsf{@PolyUI} \qquad \mathsf{this} : q\ C, \overline{\tau_a a} \mid \xi \vdash e : \tau_{ret} \qquad \text{(plus override checks)}}{C \vdash \tau_{ret}\ m(\overline{\tau_a a})\ \xi\ \{e\}}$$

FDECL-WF
$$\frac{\vdash q\ D \qquad q \neq \mathsf{@PolyUI}}{C \vdash q\ D\ f}$$

Method declarations are mostly standard, plus checking the body effects, using the declaration qualifier for the receiver. Field declarations are restricted to be monomorphic, which allows effect parameters to be treated covariantly, aligning the natural qualifier subtyping with subeffecting.

Expression and statement checking is mostly standard, using qualifier and nominal subtyping for assignments and argument passing, and checking the effect bounds. The only rule with any subtlety is the method call rule:

$$\textsc{T-Call}\ \frac{\Gamma \mid \xi \vdash e : q\ T \qquad q \approx \xi_p \qquad \tau_{ret}\, m(\overline{\tau_{arg}\, a})\xi_m \in T}{\Gamma \mid \xi \vdash e_{arg} : \tau_{arg}[\mathsf{@PolyUI} \mapsto q] \qquad \xi_m[\mathsf{@PolyUIEffect} \mapsto \xi_p] \sqsubseteq \xi}}{\Gamma \mid \xi \vdash e.m(\overline{e_{arg}}) : \tau_{ret}[\mathsf{@PolyUI} \mapsto q]}$$

Note: This rule and the method declaration rule are slightly simplified, as they do not permit restricting the receiver permission to one more specific than that corresponding to the method's declared effect.

C Soundness for λ_{UI}

To prove soundness for λ_{UI}, we must first define an operational semantics for the core language, as in Figure 2.

Lemma 5 (Substitution). *If* $\Sigma;\Gamma,x:\tau \vdash e:\tau';\xi$ *and* $\Sigma;\Gamma \vdash v:\tau;\mathsf{safe}$*, then* $\Sigma;\Gamma \vdash e[x/v]:\tau;\xi$.

Proof. By induction on the typing derivation.

Lemma 6 (Canonical Forms). *If* $\Sigma;\Gamma \vdash v:\tau;\xi$*, then*

- *If* $\tau = \tau_1 \xrightarrow{\xi'} \tau_2$*, then v is of the form* $(\lambda_{\xi'''}(x:\tau_1)\ e)$*, where* $\xi''' \sqsubseteq \xi'$.
- *If* $\tau = \mathsf{ref}\ \xi'\ \tau$*, then v is of the form* ℓ*, and* $\Sigma(\ell) = (\xi',\tau)$.
- *If* $\tau = \mathsf{nat}$ *then v is of the form n.*
- *If* $\tau = \mathsf{unit}$ *then v is of the form* $()$.

Proof. By induction on the typing derivation.

Lemma 1 (Expression Progress). *If* $\Sigma \vdash H$ *and* $\Sigma;\Gamma \vdash e:\tau;\xi$ *then either e is a value, or there exists some* H'*,* e'*, and O such that* $H,e \to_\xi H',e',O$.

Proof. By induction on the typing derivation.

- T-Nat, T-Unit, T-Loc, T-SubEff, T-SubFunEff, T-Lambda: These cases are all either typing of values, or straightforward use of the inductive hypothesis.
- T-Spawn: Apply E-Spawn.
- T-AsyncUI: Apply E-Async.
- T-Ref: $e = \mathsf{ref}_\xi\ e'$. If e' is a value, then apply E-Ref2. If e' is not a value, then by induction there exists H',e'',O such that $H,e' \to H',e'',O$, allowing application of E-Ref1.

– T-DEREF: Similar to the reference creation rules, but using E-DEREF1 and E-DEREF2, with Lemma 6 and inversion on the heap typing in the latter case.
– T-ASSIGN: $e = e_1 \leftarrow e_2$. If e_1 is not a value, then by induction

$$\exists H', e_1', O.H, e_1 \rightarrow_\xi H', e_1', O$$

allowing application of E-ASSIGN1. If e_1 is a value but e_2 is not, then similarly apply the inductive hypothesis and E-ASSIGN2. If both are values, then by Lemma 6 $e_1 = \ell$ and $\Sigma(\ell) = (\xi', \tau')$, which by inversion on heap typing means $\ell \in \mathsf{Dom}(H)$, so the expression may step by E-ASSIGN3.
– T-APP: Similar to the assignment case with its three-way split, using the application rules instead.

Lemma 2 (Expression Preservation). *If* $\Sigma;\Gamma \vdash e : \tau;\xi$, $\Sigma \vdash H$ *and* $H, e \rightarrow_\xi H', e', O$, *then there exists* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \vdash H'$, $\Sigma';\Gamma \vdash e' : \tau;\xi$, *and if* $O = (e'', \xi')$ *then there also exists* τ_n *such that* $\Sigma;\Gamma \vdash e'' : \tau_n;\xi'$.

Proof. By induction on the derivation of $H, e \rightarrow_\xi H', e', O$. In each case, any nested induction on the expression typing can produce two cases: a use of the direct rule for the reduced expression form, and an indirect typing by way of the subtyping rule. The subtyping case is a straightforward use of the inductive hypothesis, so we do not discuss those cases further.

– E-REF1, E-DEREF1, E-APP1, E-APP2, E-ASSIGN1, E-ASSIGN2: The so-called "context rules" (which would be absent if we used evaluation contexts) are all fairly similar, so we only present the case for E-APP2. By inversion on the typing derivation:
  • $\Sigma;\Gamma \vdash v : \tau_{arg} \xrightarrow{\xi_f} \tau;\xi_v$
  • $\Sigma;\Gamma \vdash e_{arg} : \tau_{arg};\xi_{arg}$
  • $\xi = \xi_f \sqcup \xi_v \sqcup \xi_{arg}$

  By inversion on E-APP2:
  • $H, e_{arg} \rightarrow H', e'_{arg}, O$

  And the goal follows from applying the inductive hypothesis to this inner reduction, and re-applying T-APP.
– E-SPAWN, E-ASYNC: These rules are similar, so we present only the E-SPAWN case. By inversion on the typing derivation:
  • $\Sigma;\Gamma \vdash e_{body} : \tau_n;\mathsf{safe}$

  The new expression $()$ trivially preserves the local typing, and the inversion results prove the typing for the new thread body.
– E-REF2: By inversion on the typing derivation:
  • $\Sigma;\Gamma \vdash v : \tau_v;\xi$

  Let $\Sigma' = \Sigma[\ell \mapsto (\xi', \tau_v)]$. The new expression ℓ (the new location) is well-typed under Σ' by T-LOC (and if $\xi = \mathsf{ui}$, T-SUBEFF). $\Sigma' \vdash H'$ by extending the inversion of $\Sigma \vdash H$ by the inversion result (the value is well-typed under Σ and $\emptyset$ since value typing does not depend on the environment) with environment weakening.
– E-DEREF2: By inversion on the typing derivation:
  • $\Sigma;\Gamma \vdash \ell : \mathsf{ref}\ \xi_{rd}\ \tau;\xi_\ell$

- $\xi = \xi_{rd} \sqcup \xi_\ell$

By inversion on the evaluation step:

- $H(\ell) = (\xi, v)$

And by inversion on $\Sigma \vdash H$:

- $\Sigma; \emptyset \vdash v : \tau; \mathsf{safe}$

If necessary, T-SUBEFF may be applied as well.

– E-APP3: By inversion on T-APP:

- $\Sigma; \Gamma \vdash (\lambda_\xi (x : \tau_{arg})\ e_{body}) : \tau_{arg} \xrightarrow{\xi_s} \tau; \xi_f$
- $\Sigma; \Gamma \vdash v : \tau_{arg}; \xi_{arg}$
- $\xi = \xi_s \sqcup \xi_f \sqcup \xi_{arg}$

By inversion on the function typing. Two typing rules apply; both are similar, with the T-SUBFUNEFF being similar to T-LAMBDA but with $\xi = \xi_s = \mathsf{ui}$, so we focus on T-LAMBDA, where inversion produces:

- $\Gamma, x : \tau_{arg} \vdash e_{body} : \tau; \xi$

Then by Lemma 5 (Substitution) and the fact that any value can be typed at a safe effect,

- $\Gamma \vdash e_{body}[x/v] : \tau; \xi$

– E-ASSIGN3: Similar to the cases for new references, but no extension of Σ is necessary. Heap typing is preserved by straightforward use of the typing for the value stored and the fact that any value can be typed with a safe effect.
– E-SUBEFFECT: By the inductive hypothesis and T-SUBEFF.

Corollary 1 (Expression Type Soundness). *If $\Sigma; \Gamma \vdash e : \tau; \xi$, and $\Sigma \vdash H$, then e is a value or there exists $\Sigma' \supseteq \Sigma$, e', H', and O such that $\Sigma' \vdash H'$, and $H, e \to H', e', O$, and $\Sigma'; \Gamma \vdash e' : \tau; \xi$ and if $O = (e'', \xi')$ then there exists τ_n such that $\Sigma'; \emptyset \vdash e'' : \tau_n; \xi'$.*

Proof. Direct consequence of Lemmas 1 and 2.

Lemma 3 (Machine Progress). *If $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ for non-empty e_u, then either $\overline{e_u} = v :: []$ and $\overline{e} = \emptyset$ or there exists $H', \overline{e_u}', \overline{e}'$ such that $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e_u}', \overline{e}' \rangle$.*

Proof. By inversion on the state typing:

– $\Sigma \vdash H$
– $\overline{\Sigma; \emptyset \vdash e_u : \tau_u; \mathsf{ui}}$
– $\overline{\Sigma; \emptyset \vdash e : \tau; \mathsf{safe}}$

If the background thread vector is nonempty, select an arbitrary component e. If that component is a value, apply E-DROPBG. If not, then by that component's typing and Lemma 1, there exists H', e', O such that $H, e \to_{\mathsf{safe}} H', e', O$, allowing an E-BG* rule to be applied depending on the value of O. For the UI thread stack, a similar argument applies. If the first expression has been reduced to a value and there is another UI expression to run, apply E-NEXTUI. If the first expression has not been reduced to a value, then a similar use of Lemma 1 applies. And if the background thread vector is empty, and the UI thread has only a single expression that is already a value, the program has terminated.

Lemma 4 (Machine Preservation). *If* $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ *and* $\langle H, \overline{e_u}, \overline{e} \rangle \rightarrow \langle H', \overline{e_u}', \overline{e}' \rangle$, *then there exists* $\Sigma', \overline{\tau_u}', \overline{\tau}'$ *such that* $\Sigma' \supseteq \Sigma$ *and* $\langle \Sigma', \overline{\tau_u}', \overline{\tau}' \rangle \vdash \langle H', \overline{e_u}', \overline{e}' \rangle$

Proof. By induction on the machine execution step, sometimes using Lemma 2.

Corollary 2 (Machine Type Soundness). *If* $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ *for non-empty* e_u, *then either* $\overline{e_u} = v :: []$ *and* $\overline{e} = \emptyset$ *or there exists* $\Sigma', H', \overline{\tau_u}', \overline{e_u}', \overline{\tau}', \overline{e}'$ *such that* $\Sigma' \supseteq \Sigma$ *and* $\langle \Sigma', \overline{\tau_u}', \overline{\tau}' \rangle \vdash \langle H', \overline{e_u}', \overline{e}' \rangle$ *and* $\langle H, \overline{e_u}, \overline{e} \rangle \rightarrow \langle H', \overline{e_u}', \overline{e}' \rangle$.

Proof. Direct consequence of Lemmas 3 and 4.