A Type and Effect System for Activation Flow of Components in Android Programs

Kwanghoon Choi
Yonsei University, Wonju, Republic of Korea
Byeong-Mo Chang*
Sookmyung Women’s University, Seoul, Republic of Korea

Abstract
This paper proposes a type and effect system for analyzing activation flow between components through intents in Android programs. The activation flow information is necessary for all Android analyses such as a secure information flow analysis for Android programs. We first design a formal semantics for a core of featherweight Android/Java, which can address interaction between components through intents. Based on the formal semantics, we design a type and effect system for analyzing activation flow between components and demonstrate the soundness of the system.

Keywords: Android, Java, Program analysis, Type, Effect, Control flow

1. Introduction

Android is Google’s new open-source platform for mobile devices, and Android SDK (Software Development Kit) provides the tools and APIs (Application Programming Interfaces) necessary to develop applications for the platform in Java [1]. An Android application consists of components such as activities, services, broadcast receivers and content providers. In Android applications, components are activated through intents. An intent is an abstract description of a target component and an action to be performed. Its most significant use is in the activation of other activities. It can also be used to send system information to any interested broadcast receiver components and to communicate with a background service.

Many static analyses of Android programs [2, 3, 4, 5] have adopted the existing Java analyses unaware of Android-specific features like components or intents, which are, however, essential for correctness of the Android program analyses. Such Android features make implicit the flow of execution, hiding it under Android platform. Therefore, the plain Java analyses cannot figure out all sound properties from Java programs running on Android platform. To address this problem, some Android analysis [5] attempted to introduce “wrapper”s modeling the Android features. However, people have never formalized the soundness of the existing Java analyses with such an Android extension.

The main contribution is to introduce a new featherweight Android/Java semantics, an analysis system for activation flow between components through intents, and its soundness proof with respect to the semantics. This activation flow analysis is important because the flow information is necessary for all Android analyses such as a secure information flow analysis. This formalization can be a basis for proving the soundness of the existing Android analyses.

We present our activation flow analysis as a type and effect system [6], and the key idea is to regard as an effect each occurrence of component activa-

*Corresponding author

Email addresses: kwanghoon.choi@yonsei.ac.kr (Kwanghoon Choi), chang@sookmyung.ac.kr (Byeong-Mo Chang)

Preprint submitted to Information Processing Letters

March 19, 2014
tion through an intent. We first design a formal semantics for a core of featherweight Android/Java, which can address interaction between components through intents (Section 3). We design a type and effect system for analyzing activation flow between components (Section 4). Our system extends a Java type-based points-to analysis [8] with Android features, which demands us to introduce a simple string analysis [10] and the notion of effects [6]. The system records in the effects the name of Android components to activate, which the string analysis extracts from intents. We demonstrate the soundness of the system based on the formal semantics (Section 5). We discuss related work and sketch an implementation of our system (Section 6).

2. Overview of Android/Java

An Android program is a Java program with APIs in Android platform. Using the APIs, one can build mobile device user interfaces to make a phone call, play a game, and so on. An Android program consists of components whose types are Activity, Service, Broadcast Receiver, or Content Provider. Activity is a foreground process equipped with windows such as buttons and text areas. Service is responsible for background jobs, and so it has no user interface. Broadcast Receiver reacts to system-wide events such as notifying low power battery or SMS arrival. Content Provider is an abstraction of various kinds of storage including database management systems.

Components interact with each other by sending events called Intent in Android platform to form an application. The intent holds information about a target component to which it will be delivered, and it may hold data together. For example, a user interface screen provided by an activity changes to another by sending an intent to the Android platform, which will destroy the current UI screen and will launch a new screen displayed by a target activity specified in the intent.

The following table lists component types and some of the methods for activating components of each type [1].

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Method for Launching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>startActivity(Intent)</td>
</tr>
<tr>
<td>Service</td>
<td>startService(Intent)</td>
</tr>
<tr>
<td>Broadcast Receiver</td>
<td>sendBroadcast(Intent)</td>
</tr>
</tbody>
</table>

2

Figure 1: Score Class in a Game Program

Note that each occurrence of the above methods in an Android program is evidence for an interaction between a caller component and a callee component to be specified as a target in the intent parameter.

This paper focuses more on Activity than the other types because Activity is the most frequently used component type in Android programs. The proposed methodology in this paper will be equally applicable to the other types of components.

This paper uses an Android-based game program shown in Figure 1 and 5 using the APIs shown in Figure 2, whose details will be explained later.

Let us examine a Java class of the Android program in Figure 1.

- Activity is a class that represents a screen in the Android platform, and Score extending Activity is also a class representing a screen.
- Once the Android platform creates a Score object, it invokes the onCreate method to add a button whose integer identifier is 1.
- Now a user can press the button 1, and then the onClick method is invoked to perform some action for the button.
- Intent is a class that represents an event to launch a new screen. It specifies the name of an activity class that represents the new screen.
- The onClick method sets “Main” as a target activity in the new intent object and requests launching by invoking startActivity.
- Android accepts the request and changes the current UI screen from Score to Main, which we call an activation flow of components.
class Activity {
    Intent intent;
    Intent getIntent() { this.intent; }
    void onClick() {};
    void addButton(int button) {}
    void startActivity(Intent i) { primStartActivity(i); }
}

class Intent {
    String target;  // The setter and getter methods
    Object data;
    String action;  // for the above fields
}

Figure 2: Android Classes: Activity and Intent

The purpose of our type and effect system is to collect from an Android program all activation flows such as the above one from Score to Main by regarding Main as the effect of Score. The system needs to employ a form of string analysis [10] to infer classes (Main) from strings ("Main") stored in intents.

3. A Semantic Model for the Android Platform

The syntax of a featherweight Android/Java is defined by extending the featherweight Java [7].

\[ N ::= \text{class } C \text{ extends } C \{ \tilde{C} \mid f \mid M \} \]
\[ M ::= C ~ m(C \mid \tilde{x}) \{ e \} \]
\[ e ::= x \mid x.f \mid \text{new } C() \mid x.f = x \mid (C)x \mid x.m(x) \mid \text{if } e \text{ then } e \text{ else } e \mid Cx = e \mid \text{prim}(\tilde{x}) \]

A list of class declarations \( \tilde{N} \) denotes an Android program. A block expression \( C \mid x = e; e' \) declares a local binding of a variable \( x \) to the value of \( e \) for later uses in \( e' \). It is also used for sequencing \( e; e' \) by assuming omission of a dummy variable \( C \mid x \). The conditional expression may be written as \( \text{ite } e \text{ if } e \text{ for brevity. We write a string object as a "string literal." Also, } x.m("...") \text{ means } \text{String } s = "..."; \text{ x.m(s) in shorthand. A recursive method offers a form of loops. The primitive functions } \text{prim}(\tilde{x}) \text{ are interfaces between an Android program and the Android platform, which will be explained later.} \]

Using the syntax defined above, we can define a small set of Android class libraries in Figure 2 to model component-level activation flow in Android programs. In Activity, the member field (intent) will hold an intent object who activates this activity object. In Intent, the target field will be a target component to be activated, the data field will be an extra argument to the target component, and the action field will describe a service that any activity activated by this intent will provide. For notation, \{void\} is a block intending to return nothing, denoted by void, and it may be written simply as \{\}. We write an object of class \( C \) as \( C\{f = l\} \) with the fields \( f \) and their values \( l \). For example, \( \text{Intent}\{\text{target} = l, \text{data} = l', \text{action} = l''\} \) denotes an intent object. \( l \) is a String reference for the name of a target component, \( l' \) is another object as an argument, and \( l'' \) is another String reference for an action description. Following the convention, an object may be denoted by its reference.

As a formal model of Android programs, we define an operational semantics for the featherweight Android/Java. A quadruple \((l, w, q, h)\) of an activity stack \( l \), a set \( w \) of button windows, an intent reference \( q \), and an object heap \( h \) forms the configuration of a screen in an Android program. \((l, w, q, h) \rightarrow (l', w', q', h')\) denotes an activation flow between the two top activity components \( l_1 \) and \( l_2 \), which is activated by the intent \( q \). \( l \) and \( l' \) may be the same. \( \rightarrow^* \) denotes zero or more steps.

A stack of activities [1] is a list \( \tilde{l} \) in the first element of each quadruple:

\( (l_1 \cdots l_n, w, q, h) \)

Each new activity reference piles up on the stack in the order of activation. Only the top activity \( l_1 \) is visible to a user and the next activity \( l_2 \) becomes visible when the top activity is removed.

Inside each activity component, the evaluation of an expression \( e \) under an environment \( \mathcal{E} \) (mapping variables into references) results in a value, which is an object reference \( l \) in the final heap, and this is denoted by the form \( \mathcal{E} \triangleright e, w, q, h \rightarrow l, w', q', h' \). A set \( w \) of button windows is merely a set of integer identifiers for buttons appearing on the screen being displayed. This is the minimal machinery to allow users to interact with Android programs. We write an intent reference in a quadruple as \( q \) to denote either \( \emptyset \) or a reference where \( \emptyset \) means no intent reference is set yet. A heap \( h \) is a mapping of references into objects.

Android platform allows each intent to specify a target activity either explicitly by giving a tar-
porting the services. When the mapping table of actions (strings describing services) object of the main class starts an Android program by creating an activity in the form of non-deterministic semantic rules. (run)

• To pick a target activity class from an explicit or implicit intent reference, we define a function for our purpose.

• To resume a target activity class from an explicit or implicit intent reference, we define a function for our purpose.

• To find an activity class, it is defined to return activity-not-found error.

Every Android program accompanies a manifest file declaring various kinds of properties of the classes including such intent filters. In this paper, such a manifest file is assumed to exist in a simplified form as IntentFilter function for our purpose.

In Figure 3, our Android platform is modeled in the form of non-deterministic semantic rules. (run) starts an Android program by creating an activity object of the main class C to return x to be bound to l after invoking the onCreate method. (launch) makes an activation flow from the top activity of l to a new one l’ through the intent by q. The rule begins when q is an intent reference(q ≠ φ). The rule takes the intent reference whose target is a new activity of class C and sets q to the intent field of the activity. Subsequently, the rule invokes the onCreate method for initialization and returns the activity reference. (button) invokes the onDismiss method of the current activity when a button i of a window set w is pressed. The invocation returns nothing, denoted by void. (back-1) and (back-2) simulate the behavior when a user presses the Back button to remove the top activity l1 to resume the next top activity l2 by calling the onRestore method.

Although the semantics considers only the onCreate method of Activity class for simplicity, it can be easily extended to support the whole life cycle of Activity [1]. For example, in (launch), the onStop method of the top activity (l1) may be called before it is hidden by a new one (l’). In (back-1) and (back-2), the onRestore method
```java
class Main extends Activity {
    void onCreate() {
        this.addButton(1); // for Game
        this.addButton(2); // for Score
        this.addButton(3); // for Help
        // initialize the main screen
    }
    void onClick(int button) {
        if (button == 1) {
            L4: Intent i = new Intent();
            L5: i.setTarget("Game");
            L6: this.startActivity(i);
        } else if (button == 2) {
            L7: Intent i = new Intent();
            L8: i.setTarget("Score");
            L9: this.startActivity(i);
        } else if (button == 3) {
            L10: Intent i = new Intent();
            L11: i.setTarget("Help");
            L12: i.setArg("Main");
            L13: this.startActivity(i);
        } else {
            // do nothing
        }
    }
}
class Game extends Activity {
    void onCreate() {
        this.addButton(1); // for Help
        this.addButton(2); // for Score
        this.addButton(3); // for playing
        // display the game screen
    }
    void onClick(int button) {
        if (button == 1) {
            L14: Intent i = new Intent();
            L15: i.setTarget("Help");
            L16: i.setArg("Game");
            L17: this.startActivity(i);
        } else if (button == 2) {
            L18: Intent i = new Intent();
            L19: i.setTarget("Score");
            L20: this.startActivity(i);
        } else if (button == 3) {
            // play the game
        } else {
            // do nothing
        }
    }
}
class Help extends Activity {
    void onCreate() {
        this.addButton(1); // for Back
        // display the help screen
    }
    void onClick(int button) {
        L21: Intent i = new Intent();
        // String s=(String)
        // this.getIntent().getArg();
        L22: Intent j = this.getIntent();
        L23: Object o = j.getArg();
        L24: String s = (String)o;
        L25: i.setTarget(s);
        L26: this.startActivity(i);
    }
}
```

Figure 5: A Game Program

Due to the lack of space, we omit the semantic rules for handling null pointer reference, casting errors, and other errors such as the absence of methods or fields and type errors in Figure 3 and 4.

The semantic rules use some auxiliary functions defined in [7]. $mbody(m, C)$ returns the body expression of the method of the class, and $fields(C)$ gathers all fields belonging to the class, if necessary, following up the inheritance tree.

The proposed semantics is capable of making run an Android game program in Figure 1 and 5. The program consists of four activities: Main, Game, Help, and Score. The entry activity Main offers a user three buttons each for activating Game, Score, and Help. During playing a game, a user can check out game instruction through Help activity. After the game is finished, the score is displayed by Score.
activity and then the user moves to Main activity. Note that Help can be activated by both Main and Game. In either case, Help goes back to its caller activity properly because both caller activities set their name to the argument of an intent to activate Help with. Note that using activity stack allows to omit setting one’s own name for coming back.

4. A Type and Effect System

This section proposes a type and effect system to analyze activation flow between components through intents. Our system is a type-based points-to analysis system [8, 9] extended with a simple string analysis [10] and effects [6].

Our system abstracts objects by an annotated type \( S \), which has the form of \( C\{R\} \), where \( C \) is a class or primitive type and \( R \) is a set of program points. We are particularly interested in program points for an object creation expression in a program. The type \( C\{R\} \) represents objects of \( C \), which are created at one of program points in \( R \).

For example, in Line 22 of Figure 5, the intent variable \( j \) has (annotated) type \( Intent\{r_{10}, r_{14}\} \). This is because the reference in \( j \) points to either an intent object created in Line 10 (denoted by a program point \( r_{10} \)) or one in Line 14 (denoted by another \( r_{14} \)). For convenience, this paper uses mostly line numbers for program points.

Each effect represents a set of components, which can be activated through intents in Android programs. The effect \( \varphi \) is defined by

\[
\varphi ::= \{C\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset
\]

where \( C \) denotes a component name, which is actually a class name. So \( \varphi \) will be a set of component (class) names.

Using effects, a method type is defined to be the form of \( S \xrightarrow{} T \), denoting that calling a method of the type may make effect \( \varphi \).

In our type and effect system, typing judgments for expressions have the form of

\[
\Gamma \triangleright e : C\{R\}, \varphi
\]

where, under the typing environment \( \Gamma \) (mapping variables to annotated types), an expression \( e \) evaluates to an object of \( C \), which is created at a program point in \( R \), and during computation, side effects (activation of other components) expressed by \( \varphi \) might occur.

Our typing rules for expressions in Figure 6 depends on field and method typings \( F \) and \( M \). A field typing \( F(C, r, f) \) assigns a type to each field \( f \) of class \( C \) in objects created at a program point \( r \). A method typing \( M(C, r, m) \) associates a method type of form \( S \xrightarrow{} T \) with each method \( m \) of class \( C \) in objects created at a program point \( r \). We will discuss how to get field and method typings later.

For example, the field \( intent \) of class \( Help \) in objects created at \( r_{\text{help}} \) gets a field typing as

\[
F(\text{Help}, r_{\text{help}}, \text{intent}) = \text{Intent}\{r_{10}, r_{14}\}
\]

by the reason explained previously. Note that \( r_c \) is a program point of an object creation expression “new \( C(f) \)” in (launch). For example, \( r_{\text{help}} \) is a program point of the expression in (launch) creating a Help activity (by replacing \( C \) with \( \text{Help} \)).

For example, the \( \text{onClick} \) method of \( \text{Score} \) in objects created at \( r_{\text{score}} \) gets a method typing:

\[
M(\text{Score}, r_{\text{score}}, \text{onClick}) = \text{Int}(r_{\text{button}}) \xrightarrow{\{\text{Main}\}} \text{void}\}
\]

where \( r_{\text{button}} \) is another program point to identify integers \( i \) created at (button). In Line 2, the \( \text{setTarget} \) method sets “Main” to the target component name of an intent \( i \) (created at Line 1) to activate Main in Line 3, so the effect of the \( \text{onClick} \) method is \( \{\text{Main}\} \) obviously.

However, target component names set by the \( \text{setTarget} \) method are not always obvious. In Line 25 (the \( \text{onClick} \) method of \( \text{Help} \)), the target component name is given by a variable \( s \). The target component names will become obvious only after some string analysis is employed to uncover strings to which \( s \) will evaluate.

Our system with annotated types includes a simple form of string analysis by having a string table \( \Omega(r) \), mapping each program point \( r \) onto either a set of string literals or \( \top \) (denoting a set of all string literals). Every expression of type \( \text{String}\{R\} \) will evaluate to some string in the union of sets of strings \( \Omega(r) \) for all \( r \in R \). For example, \( \Omega(r_{12}) = \{\text{“Main”}\} \) and \( \Omega(r_{16}) = \{\text{“Game”}\} \) since the two strings occur at Line 12 and 16, respectively. The type of the variable \( s \) in Line 25 turns out to be \( \text{String}\{r_{12}, r_{16}\} \), and so \( s \) will evaluate to a string in \{“Main”, “Game”\}.

The string analysis in our system explained until now can be regarded as [8]. In addition, our system extends it to deal with Android activation flow with new typing rules using the notion of effect. Without
the new rules, the string analysis will lose some data flow among activities in Android programs and so will be unsound, as will be explained later.

Now we present a set of typing rules in Figure 6. The system needs subtyping relations defined in the standard way. \( C < D \) if \( C \) is the same as \( D \) or its descendant class; \( C[R_1] <: D[R_2] \) if \( C <: D \) and \( R_1 \subseteq R_2 \); \( \varphi_i \subseteq S \frac{x}{\Rightarrow} T \) if \( T_i <: S_i \) for all \( i \), \( S < T \), and \( \varphi_1 \subseteq \varphi_2 \). For convenience, the notation \( F(C, R, f) <: S \) means that \( F(C, r, f) <: S \) for all \( r \in R \). The reverse direction of the notation and \( M(C, R, m) <: S \frac{x}{\Rightarrow} T \) can be defined similarly.

(T-var) looks up the type of a variable from the typing environment. (T-field) directs the flow of objects stored in the field to the reader by \( x.f \) by the subtyping relation. (T-assign) specifies the flow of objects from the right-hand side \( y \) to the field \( x.f \). (T-new) abstracts all objects generated in each \( \text{new } C() \) expression with a program point \( r \) by an annotated type \( C[r] \). In (T-var), (T-field), (T-assign), (T-new), and (T-cast), each expression has no effect. (T-if) merges data flows of the branches by assigning the same type to them. In (T-if) and (T-block), the effect of each expression is the union of all the effects from its sub-expressions. (T-involve) defines the type of each method \( m \) is more specific than the method type formed with actual argument and return types in the caller. The effect of each invocation expression results from that of the called method. (T-sub) allows the same expression to have a less precise type and effect.

(T-string) collects all occurrences of string literals and their program points in the string table.

The typing rules from (T-var) to (T-string) are ones in [8] extended with effects in this paper. (T-prim-1) and (T-prim-2) are new.

(T-prim-1) has two roles as the origin of an effect and as a (data flow) bridge between caller and callee activities. First, the effect of this primitive is target component names that the primitive may activate, and will be computed by the effect function over the type \( S \) of intents that the primitive takes. Second, the rule has a condition as

\[
S <: F(C, r, \text{intent}) \quad \text{for all } C \in \text{effect}(S)
\]

to express passing intents of type \( S \) from a caller to all callee activities in (launch) by a field assignment “\( x.\text{intent} = \text{intent} \)”.

(T-prim-2) causes no effect.

The function \( \text{effect}(S) \), collecting a set of potential target component names from an intent type \( S \), is an abstraction of \( \text{target}(q, h) \) for each intent reference \( q \) of type \( S \) in heap \( h \), as will be proved later. We define this function as the least set \( \varphi \) satisfying the following conditions: \( S \) is \( \text{Intent}[R] \) and, for each \( r \in R \), there are \( R_t \) and \( R_a \) such that \( F(\text{Intent}, r, \text{target}) = \text{String}[R_t] \) and \( F(\text{Intent}, r, \text{action}) = \text{String}[R_a] \). Then,

- \( \text{Class}(\text{Omega}(r_i)) \subseteq \varphi \) for all \( r_i \in R_t \)
- \( \text{IntentFilter}(\text{Omega}(r_a)) \subseteq \varphi \) for all \( r_a \in R_a \)

For example, intent objects at \( r_{10} \) and \( r_{14} \) are passed to \( \text{Help} \) in Line 13 and 17 respectively as:

- \( M(\text{Main}, r_{main}, \text{startActivity}) \) \( \xrightarrow{\text{Intent}[r_{10}]} \) \( \text{void}() \)
- \( M(\text{Game}, r_{game}, \text{startActivity}) \) \( \xrightarrow{\text{Intent}[r_{14}]} \) \( \text{void}() \)

Due to the intent passing, (T-prim-1) forces both \( \text{Intent}[r_{10}] \) and \( \text{Intent}[r_{14}] \) to be a subtype of \( F(\text{Help}, r_{\text{help}}, \text{intent}) \), which is \( \text{Intent}[r_{10}, r_{14}] \).
In Line 26, the type of \( i \) is \( \text{Intent}(r21) \). \( \text{effect}(\text{Intent}(r21)) \) is \{Main, Game\} if the target component of \( i \) set by the setTarget method in Line 25, i.e., \( s \), evaluates to “Main” or “Game”. The evaluation is analyzed to be so by the condition of (T-prim-1) as follows. In Line 23 and 24, \( s \) is from the data field of another intent \( j \). In Line 22, \( j \) is from the intent field of Help, and it is of type \( \text{Intent}(r10, r14) \). The data field of intents of the type evaluates to “Main” or “Game” because of

- \( F(\text{Intent}, r10, \text{data}) = \text{String}(r12) \) and
- \( F(\text{Intent}, r14, \text{data}) = \text{String}(r16) \).

Therefore, so does the target component of \( i \) (of type \( \text{String}(r12, r16) \)), which cannot be analyzed without (T-prim-1). In Line 26, we thus have:

- \( M(\text{Help}, \text{startActivity}) = \text{Intent}(r21) \) for some \( \{\text{Main, Game}\} \rightarrow \text{void}() \)

The effect of a class is the union of effects of all methods in a class \( C \), denoted by \( \text{effect}(C) \). For example, the effect of Main, Score, Game, and Help is \{Game, Score, Help\}, \{Main\}, \{Score, Help\}, and \{Main, Game\}, respectively. In a well-typed Android program, \( C \) will activate one in \( \text{effect}(C) \) by the soundness of our system to be shown later.

As in [8], \( F \) and \( M \) should be well-formed: \( F(C, r, f) :<: F(D, r, f) \) and \( M(C, r, m) :<: M(D, r, m) \) for all \( r \) and \( C :<: D \), which are natural extensions of those for Java. This well-formedness is enforced by having a subtyping relationship between each pair of overriding/overridden methods.

Every Android program is well-typed if, for all classes \( C \), program points \( r \), methods \( m \), method types \( S \rightarrow^* T \) such that \( M(C, r, m) = S \rightarrow^* T \), we can derive \( \{\text{this} : C[r], \bar{x} : S\} \triangleright e : T, \varphi \) where \( \text{mbody}(m, C) = D \triangleright C, \bar{x}, e \) (\( C \) and \( D \) are \( \bar{S} \) and \( T \) without annotations).

5. Soundness of the Type and Effect System

The soundness of our type and effect system means that every activity in a well-typed Android program will activate only the activities in its effect.

For a formal statement of the soundness, we define a proposition \( \text{flow}(C, D) \) for two lists of activity classes \( C \) and \( D \) to declare that each activity on the stack is activated by an activity directly under itself. The proposition is true if

- \( D = C \), \( D = C_0 \cdot C \), or \( D = C_{2..n} \)

such that \( C_{k-1} \in \text{effect}(C_k) \) for \( k \geq 1 \).

We extend it to more than two lists of activity classes as: \( \text{flow}^*(C, D_1, ..., D_m) \) is true if \( \text{flow}(C, D_1) \), ..., and \( \text{flow}(D_{m-1}, D_m) \) are all true.

In the following theorem, we associate a stack of activities \( l \) with classes \( C \) by a relation \( l \sim C \) where each \( l_i \) is an activity object of class \( C_i \).

**Theorem 1 (Soundness for Android/Java).** Suppose an Android program \( N \) is well-typed. For a main activity class \( C \) in the program,

- \( \text{run } C \implies \bar{l}_1, w_1, q_1, h_1 \implies^* \bar{l}_m, w_m, q_m, h_m \)

such that \( \text{flow}^*(C, D_1, ..., D_m) \) where \( l_i \sim D_i \).

Otherwise, the evaluation stops with null error, cast error, or activity-not-found error.

**Lemma 1 (Intent Abstraction).** If \( h : \mathcal{H} \) and \( \mathcal{H} \triangleright l : \text{Intent}(R) \) for some \( R \)

- \( \text{target}(l, h) \in \text{effect}(\text{Intent}(R)) \).

Otherwise, the evaluation of \( \text{target}(l, h) \) returns either null error or activity-not-found error.

The soundness theorem says that, when an Android program is well-typed, the execution will have three cases, all satisfying the activation flow proposition: it may run normally to stop with \( \emptyset, \emptyset, \emptyset, h \), it may run some infinite loop, or it may get stuck due to an erroneous event from one of the null reference error, the casting error, and the activity not found error (the other kinds of errors will never happen).

Two soundness lemmas on expressions and quadruples constitutes the proof of this soundness theorem. The two lemmas and the proofs are in the extended version [11].

The intent abstraction lemma supports the soundness theorem in the case with (launch) of the proof on quadruples. In order to satisfy the activation flow proposition, the effect of the top activity class in the stack of a quadruple should include a target class \( C \) to launch, which is true because of this: the target activity of the intent in (launch) is a member of the effect of the intent type by the intent abstraction lemma, and the effect of the intent type is included in the effect of the top activity class by allowing only well-formed quadruples [11] in the execution.

6. Discussion

There are several relevant systems for Android inter-component communication analysis to report

The existing static analyses of Android programs have little semantic-based accounts on how their Android specific analyses interplay with the Java analyses they are based on. Contrary to this, we formally proved the soundness of our Android analysis. In this respect, our proposed system can be regarded as a theoretical basis for them.

In practice, one can also implement our system as a fully automatic analyzer for featherweight Android programs such as our game example. To support this claim, our prototype is available at:

http://mobilesw.yonsei.ac.kr/paper/android.html

Our Android analyzer consists of five steps to compute the effect of activities in an Android program. First, it applies the standard Java type checking procedure to an Android program to attach Java types to the abstract syntax tree. Second, it collects all classes declared and referred in the program and all program points for object creation sites. Third, it initializes field and method typings for the classes and program points with the Java types annotated with new program point set variables and effect variables. Fourth, it generates subtyping constraints and activation constraints on the variables by applying the typing rules to each method typing according to the well-typedness. Fifth, it solves all the constraints to produce a solution (mapping of the variables onto ground program point sets and effects) that completes the field/method typings.

The method typings thus computed offer information enough for the analyzer to compute the effect of each Activity class, which is our goal.

Note that the analyzer deals not only with the subtyping constraints as in [8], but it also introduce a new form of constraints $\text{Intent}\{S\} \Rightarrow \varphi$ for each use of $\text{primStartActivity}(\ldots)$ in an Android program. We call this an activation constraint. Solving each activation constraint is to generate a new subtyping constraint $S \prec : F(C, r, _i, \text{intent})$ whenever a new class $C$ becomes belong to the effect of this intent type $S$, $\text{effect}(S)$. Having all the generated subtyping constraints will enforce the universally quantified condition in (T-prim-1).

7. Conclusion

We have proposed a type and effect system for analyzing activation flows between components in Android programs where each activation of a component through an intent is regarded as an effect. Also, we have presented a featherweight Android/Java semantics that the soundness of our Android analysis is based on. To support the feasibility of our system as a full automatic analyzer, we have presented a prototype, though we need to extend it further to apply to real Android programs, which is a future work.

References

This appendix is only for reviewers and their re-
view, and is available in the extended version [11]
of this paper.

Appendix A. Definitions and Lemmas for Soundness

To formulate the soundness property of Android programs, we define typing rules for the semantic elements:

- Reference typing: $\mathcal{H} \triangleright \text{null} : \text{null} \{\}$. $\mathcal{H} \triangleright l : C[r]$ if $\mathcal{H}(l) = C[r]$. $\mathcal{H} \triangleright v : T$ if $\mathcal{H} \triangleright v : S$ and $S < T$ where $v$ is either null or $l$.

- Environment typing: either $\mathcal{H} \triangleright \emptyset : \emptyset$, or $\mathcal{H} \triangleright \mathcal{E}[x : l] : \Gamma[x : S]$, if $\mathcal{H} \triangleright \mathcal{E} : \Gamma$ and $\mathcal{H} \triangleright l : S$.

- Window typing: $\triangleright \{i_1, \ldots, i_n\} : W$ for arbitrary integers. $W$ is the window set type.

- Heap typing: $\triangleright \emptyset : \emptyset$. $\triangleright h(l) \rightarrow C[\ell = \tilde{l}] : H[l : C[r]]$ if $\triangleright h : \mathcal{H}$, fields($C$) = $f$, and $\mathcal{H} \triangleright l_1 : F(C, r, f_1)$ for some $r$.

Note that reference typing for primitive values such as int or boolean is as follows: $\mathcal{H} \triangleright i : \text{int} \{r\}$ or $\mathcal{H} \triangleright b : \text{boolean} \{r\}$ for some $r$. Also note that reference typing for intents is written as $\mathcal{H} \triangleright q : Q$ where $q$ is $l$ and $Q$ is $\text{Intent}\{R\}$ for some $R$.

Note that heap typing for strings is as follows: $\triangleright h(l) \rightarrow \text{"str"} : H[l : \text{String}\{r\}]$ if $\triangleright h : \mathcal{H}$ and $\Omega(r) = \{\text{"str"}\}$ for some $r$.

A quadruple $(l, w, q, h)$ is well-formed if there exists $\mathcal{H}$ and $Q$ such that $\mathcal{H}(l_i) = C_i[r_i]$ and $C_i <: \text{Activity}$ for all $l_i \in l$ and $r_i$s, $\triangleright w : W$, $\triangleright q : Q$, $\triangleright h : \mathcal{H}$, effect($Q$) $\subseteq$ effect($C_1$), $C_{k-1} \in$ effect($Q_k$)($k = 2, \ldots, n$), and $Q <: F(D, r_d, \text{intent})$ for all $D \in$ effect($Q$).

The two lemmas for soundness of expressions and quadruples are as follows.

Lemma 2 (Soundness of Expressions).

Suppose $\mathcal{H} \triangleright \mathcal{E} : \Gamma$, $\mathcal{E} \triangleright e : S, \varphi$, $\triangleright w : W$, $\mathcal{H} \triangleright q : Q$, $\triangleright h : \mathcal{H}$, and $Q <: F(D, r_d, \text{intent})$ for all $D \in$ effect($Q$). There exist $H'$ and $Q'$ such that:

- $\mathcal{E} \triangleright e, w, q, h \rightarrow l, w', q', h'$ such that $\mathcal{H}' \subseteq H'$, $\mathcal{H}'(l) <: S$, $\triangleright w' : W$, $\mathcal{H}' \triangleright q' : Q'$, $\triangleright h' : \mathcal{H}'$, effect($Q'$) $\subseteq$ effect($Q$) $\cup$ $\varphi$, and $Q' <: F(D', r_d, \text{intent})$ for all $D' \in$ effect($Q'$).

Otherwise, the evaluation stops with null error or cast error.

Lemma 3 (Soundness of Quadruples).

If $l, w, q, h$ is well-formed then:

- either $l, w, q, h \Longrightarrow l', w', q', h'$ such that $l', w', q', h'$ is also well-formed and flow($C, C'$) where $l \sim C$ and $l' \sim C'$, or $(\emptyset, w, q, h)$ stops.

Otherwise, the evaluation stops with null error, cast error, or activity-not-found error.

Appendix B. Proof of Theorem 1 (Soundness for Android/Java)

Since $C$ is an activity class by the assumption, $C$ has an `onCreate` method. Let $\mathcal{E}$ be $\{\}$ and let $\Gamma$ be $\{\}$. Also let $w$, $q$, and $h$ be all $\emptyset$. Let $e$ be $(C = x = \text{new} C())$. $x.onCreate(); x).$ By the well-typed program condition, $M(C, r_c, onCreate) = \{x \Rightarrow \text{void}\}$, which implies $\Gamma \triangleright e : C\{r_c\}, \varphi$ for some $\varphi$.

If we let $Q$ and $H$ both be $\emptyset$ then the hypotheses of Lemma 2 are all satisfied by the following observation. Since $Q$ is empty, $\text{effect}(Q)$ is empty too and so $Q <: F(D, r_d, \text{intent})$ for all $D \in$ effect($Q$) holds trivially.

The application of Lemma 2 allows the evaluation $\mathcal{E} \triangleright e, w, q, h \rightarrow l, w', q', h'$. Also, by Lemma 2, there exists $Q'$ and $H'$ such that $\mathcal{H} \subseteq H'$, $\mathcal{H}'(l) <: C\{r_c\}, \triangleright w' : W$, $\mathcal{H}' \triangleright q' : Q'$, $\triangleright h' : H'$, effect($Q'$) $\subseteq$ effect($Q$) $\cup$ $\varphi$, $Q' <: F(D', r_d, \text{intent})$ for all $D' \in$ effect($Q'$).

Now we verify the well-formedness of $(l, w', q', h')$. Since $Q$ is $\emptyset$, $\text{effect}(Q)$ is empty. This implies effect($Q'$) $\subseteq$ effect($Q$) $\cup$ $\varphi$ = $\emptyset$ $\cup$ $\varphi$. The effect $\varphi$ comes from invoking the onCreate method of $C$, we have $\varphi \subseteq \text{effect}(C)$, which leads to effect($Q'$) $\subseteq$ effect($C$). Since the quadruple has an activity stack of a single activity reference $l$, the well-formedness condition for activity stack is trivially satisfied; $C_{k-1} \in \text{effect}(C_k)$ for $k = 2, \ldots, n$ where $n$ is $1$.

Since $(l, w', q', h')$ is a well-formed quadruple, we can apply Lemma 3, which allows an evaluation $l, w', q', h' \Rightarrow l_1, w_1, q_1, h_1$. Again, the quadruple $(l_1, w_1, q_1, h_1)$ in the right-hand side of the evaluation is also well-formed by the lemma, and so we can repeat the application of the lemma until the evaluation stops.

The application of Lemma 3 ensures the flow condition between two subsequent quadruples. To combine these flow conditions for all such pairs of subsequent quadruples involved in the series of evaluation, we get the flow condition in Theorem 1,
which finishes the proof of the soundness of Android/Java.

Appendix C. Proof of Lemma 2 (Soundness of Expressions)

We prove this lemma by induction on the height of the evaluation of expressions. The lemma is a combination of a progress property, saying the well-typedness of a given expression allows to have a legal evaluation on it, and a subject reduction property, saying the every well-typed expression evaluates to a well-typed value.

Case (var). Let $e$ be $x$. By the hypotheses of the lemma, $\Gamma \vdash x : S$. By (T-var), $\Gamma(x) = S$. Since $H \vdash \mathcal{E} : \Gamma$, $\mathcal{E}(x)$ exists, which implies $\mathcal{E} \triangleright x, w, q, h \rightarrow \mathcal{E}(x), w, q, h$.

Let $H'$ and $Q'$ be $H$ and $Q$. Then $H'(l) = H(l) <: \Gamma(x) = S$. Also, $\text{effect}(Q') = \text{effect}(Q) \subseteq \text{effect}(Q) \cup \varphi$. The other conditions follow the hypotheses immediately.

Case (field). Let $e$ be $x.f_i$. By the hypotheses of the lemma, $\Gamma(x) = C[R]$ and $F(C, R, f_i) <: S$ by (T-field). By the environment typing, $H \triangleright l_x : C[R]$, which implies $H(l_x) = D[r] <: C[R]$ for some $D[r]$ by reference typing. Since $D <: C$, $D$ has a field $f_i$, which allows an evaluation as $\mathcal{E} \triangleright x, f_i, w, q, h \rightarrow l_i, w, q, h$ where $l_i$ is a reference in $f_i$ of an object $h(l_x)$.

Let $H'$ and $Q'$ be $H$ and $Q$. By heap typing, $H' \triangleright l_i : F(D, r, f_i)$, which implies $H'(l_i) <: F(D, r, f_i)$. $F(D, r, f_i) <: F(C, r, f_i)$ holds by $D <: C$, and $F(C, r, f_i) <: S$ by the hypothesis of (T-field) and $r \in R$. To sum these up, $H'(l_i) <: S$. The other conditions immediately follow the hypotheses.

Case (assign). By (T-assign), $\Gamma(x) = C[R]$ and $\Gamma(y) = T$ such that $T <: F(C, R, f_i)$. The environment typing says that there exists $l_x$ and $l_y$ such that $\mathcal{E}(x) = l_x$ and $\mathcal{E}(y) = l_y$, and it also says that $h(l_x) = D[f = l]$ which implies $H(l_x) = D[r]$ for some $r$ by the heap typing. Since $H(l_x) <: C[R]$ and $H(l_y) <: T$ by environment typing and heap typing, we have $D[r] <: C[R]$ such that $r \in R$ and $H(l_y) <: T <: F(C, R, f_i)$. By the field typing $F(C, R, f_i)$ and the type of $x$, $l_x$ is a reference to an object who has $f_i$ as a field. Since $D[r] <: C[R]$, $D <: C$ and the field $f_i$ of $C$ is also available in $D$. This makes an evaluation $\mathcal{E} \triangleright x.f_i = y, w, q, h \rightarrow \text{void}, w, q, h'$ where $h' = h\{l_x \mapsto D\{f = l_{i-1}l_{i+1}, n\}\}$.

Since $H(l_y) <: F(C, r, f_i)$, we have $H \triangleright l_y : F(C, r, f_i)$. This implies $h\{l_x \mapsto D\{f = l_{i-1}l_{i+1}\}\} : H\{l_x : D[r]\}$ by heap typing. If we let $H'$ be $H$, then $h' : H'$. Also if we let $Q'$ be $Q$, then the remaining conditions of the conclusion in the lemma are trivially satisfied.

Case (new). Let $e$ be new $C()$. We have an evaluation $\mathcal{E} \triangleright e, w, q, h \rightarrow l, w, q, h'$ where $h' = h\{l \mapsto C\{f = \text{null}\}\}$.

Let $H'$ be $H \cup \{l : S\}$ where $S = C[r]$. Then $H'(l) <: S$ trivially holds. By (new), $q'$ is the same as $q$. By the hypothesis, $H \triangleright q : Q$ is given. Let $Q'$ be $Q$. Then $H' \triangleright q' : Q'$ by the weakening of the reference typing since $H \subseteq H'$.

Similarly, $H' \triangleright l_i : S_i$ for all $l_i \in dom(H)$ by $H \triangleright l_i : S_i$ and the weakening of the heap typing. Also, $H' \triangleright l : C[r]$ because $H' \triangleright \text{null} : \text{null}\{\}$ and $\text{null}\{\} <: F(C, r, f_i)$. Consequently, we have $h' : H'$. Since $Q'$ is defined as $Q$, the remaining conditions of the conclusion in the lemma are satisfied immediately.

Case (cast). Let $e$ be (C)x. Suppose $\Gamma(x) = E[R]$ by (T-cast). $\mathcal{E}(x) = l$ and $h(l) = D[f = l]$ by (cast). By heap typing, $H(l) = D[r]$ for some $r$. By environment typing, we have $D[r] <: E[R]$, which implies $r \in R$.

When $E <: C$, (cast) allows to have an evaluation $\mathcal{E} \triangleright e, w, q, h \rightarrow l, w, q, h$. Let $H'$ be $H$. Then we can verify $H'(l) = D[r] <: E[R] <: C[R]$. The remaining conditions are satisfied immediately by letting $Q'$ be $Q$.

When $D <: C < E$, we have the same evaluation as above. When we let $H'$ be $H$, $H'(l) = D[r] <: C[R]$.

When $C <: D$ or $C$ and $D$ do not have any inheritance relationship, it leads to the casting error.

Case (if). Let $e$ be if $e_0 \ e_1 \ e_2$. By (T-iff), we have a typing $\Gamma \vdash e_0 : \text{boolean}\{R\}, \varphi_0$, which, together with the hypotheses of the lemma, allows to apply an induction to $e_0$.

The induction allows to have an evaluation $\mathcal{E} \triangleright e_0, w, q, h \rightarrow b_0, w_0, q_0, h_0$. It also implies that there exist $H_0$ and $Q_0$ such that $H \subseteq H_0$, $h_0 : H_0$, $H_0 \triangleright q_0 : Q_0$ and $\text{effect}(Q_0) \subseteq \text{effect}(Q) \cup \varphi_0$.

We know that $b_0$ is either true or false by the (extended) reference typing.
By (T-if), we have another typing $\Gamma \vdash e_k : S_k, \varphi_k$ for each $k = 1, 2$. It is straightforward to verify the induction hypotheses over $e_k$.

The second induction implies that, depending on the result of evaluation of $e_0$, we have $\mathcal{E} \triangleright e_k, w_0, q_0, l_0 \rightarrow l, w', q', h'$ by (if). By the induction, there also exist $H'$ and $Q'$ such that $H'(l) \subseteq S$ and $\text{effect}(Q') \subseteq \text{effect}(Q_0) \cup \varphi_k$. $\text{effect}(Q_0) \cup \varphi_k \subseteq \text{effect}(Q) \cup \varphi_0 \cup \varphi_1 \cup \varphi_2$.

Case (block). Let $e$ be $C \ x = e_1; e_2$. By (T-block), we have a typing $\Gamma \vdash e_1 : C \{R\}, \varphi_1$. Together with the hypotheses of the lemma, we can apply an induction over $e_1$.

The induction allows to have an evaluation $\mathcal{E} \triangleright e_1, w, q, h \rightarrow l_1, w_1, l_1$. It also implies that there exists $H_1$ and $Q_1$ such that $\mathcal{E} \triangleright Q_1, H_1 \rightarrow q_1 : Q_1, H_1(l_1) \subset C \{R\}$, and $\text{effect}(Q_1) \subseteq \text{effect}(Q) \cup \varphi_1$.

Since $H_1 \rightarrow l_1 : C \{R\}, H_1 \triangleright \mathcal{E} \{x \mapsto l_1\} : \Gamma \{x : C \{R\}\}$ by environment typing. Then it is easy to verify the induction hypotheses over $e_2$.

The second induction allows to have an evaluation $\mathcal{E} \{x \mapsto l_1\} \triangleright e_2, w_1, q_1, h_1 \rightarrow l, w', q', h'$. It also implies that there exist $H'$ and $Q'$ such that $H_1 \subseteq H', H'(l) \subset S$. $H' \triangleright q : Q'$, and $\text{effect}(Q') \subseteq \text{effect}(Q_1) \cup \varphi_2$.

Then we see $\text{effect}(Q') \subseteq \text{effect}(Q) \cup \varphi_1 \cup \varphi_2$.

Case (invoke). Let $e$ be $x.\text{m}(\bar{y})$. By the environment typing, there exist $l_x$ and $l_\bar{y}$ such that $\mathcal{E}(x) = l_x$ and $\mathcal{E}(\bar{y}) = l_\bar{y}$. It also suggests that $\mathcal{H} \triangleright l_x : C \{R\}$, which implies that $h(l_x) = D(f = l_\bar{y}), H(l_x) = D\{r\}$ for some $r$, and $D\{r\} \subset C \{R\}$ by the heap typing.

By (T-invoke), the method typing $M(C, R, m)$ implies that there exists a method $m$ in $C$, and a method named $m$ is also available in $D$ by $D \subset C$. Therefore, we have $\text{mbody}(m, D) = B \bar{z} \ x_0$.

(T-invoke) also offers $M(C, R, m) \subset S_i \bar{z} \ S$. Let $M(D, r, m) = S'_i \bar{z} \ S'$. Since $D \subset C$, the well-formedness of method typings allows to have $M(D, r, m) \subset M(C, R, m) \subset M(C, R, m)$. This subtyping relations imply $S' \subset S$ and $S_i \subset S'_i$.

Let $\mathcal{E}_0$ be $\{\bar{z} \rightarrow l_x, \bar{z} \rightarrow l_\bar{y}\}$ and $\Gamma_0$ be $\{\bar{z} : D\{r\}, \bar{z} \bar{z} : S'_i\}$. By the well-typedness of a given program, we have a typing $\Gamma_0 \triangleright e_0 : S', \varphi'$. By environment typing, $\mathcal{H} \triangleright \mathcal{E}_0 : \Gamma_0$. The rest of the induction hypotheses immediately come from the hypotheses of the lemma. This allows to use an induction over $e_0$ to have an evaluation $\mathcal{E}_0 \triangleright e_0, w, q, h \rightarrow l, w', q', h'$.

Once the induction is applied, there exist $H'$ and $Q'$ such that $\mathcal{H} \subset H', H'(l) \subset S'$, and $\text{effect}(Q') \subseteq \text{effect}(Q) \cup \varphi'$. Since $S' \subset S$, we have $H'(l) \subset S$. Also, by $\varphi' \subseteq \varphi$, $\text{effect}(Q') \subseteq \text{effect}(Q) \cup \varphi$.

Case (prim-1). We have an evaluation $\mathcal{E} \triangleright \text{primStartActivity}(x), w, q, h \rightarrow \text{void}, w, \mathcal{E}(x), h$ unconditionally. By environment typing, $\mathcal{H} \triangleright \mathcal{E}(x) : \Gamma(x)$. By (T-prim-1), there exists $S_2$ such that $\Gamma(x) = S_2$ and $\text{effect}(S_2) = \varphi$. Since $q$ is $\mathcal{E}(x)$, if we let $Q' = S_2$ and $H' = H$, then we have $\mathcal{H} \triangleright \mathcal{E}(x) : Q'$. Now it is easy to verify $\text{effect}(Q') = \text{effect}(S_2) = \varphi \subseteq \text{effect}(Q) \cup \varphi$.

(T-prim-1) offers an invariant condition $S_2 \subset F(C, r_w, \text{intent})$ for all $C \in \text{effect}(S)$, which is a condition required in the conclusion.

Case (prim-2). We have an evaluation $\mathcal{E} \triangleright \text{primAddButton}(x), w, q, h \rightarrow \text{void}, w \cup \mathcal{E}(x), q, h$ unconditionally. By (T-prim-2), $\mathcal{H} \triangleright \mathcal{E}(x) : \text{int}\{R\}$ for some $R$. By the extended reference typing, $\mathcal{E}(x)$ is an integer, and so $w \cup \mathcal{E}(x)$ is a set of integers by the hypothesis $\triangleright w : W$. Therefore, $\triangleright w \cup \mathcal{E}(x) : W$.

Appendix D. Proof of Lemma 3 (Soundness of Quadruples)

We prove this lemma by case analysis of the semantic rules used for quadruples.

Case (launch). Let $\mathcal{E}$ be $\{\text{intent} \rightarrow q\}$ and let $\Gamma$ be $\{\text{intent} : Q\}$. Since the hypothesis $\mathcal{H} \triangleright q : Q$ is given, we have $\mathcal{H} \triangleright \mathcal{E} : \Gamma$ by environment typing. Also let $e$ be $C \ x = \text{new} \ C(); \ x.\text{intent} = \text{intent}; \ x.\text{onCreate}(); \ x$. We let $r_e$ be a unique program point for the object creation expression "new $C()" in $e$. By applying typing rules (T-block), (T-assign), (T-invoke), (T-var), we derive a typing judgment $\Gamma \triangleright e : C\{R\}, \varphi$ for some $R$ using the hypotheses of the well-formedness of quadruples $Q \subset F(C, r_e, \text{intent})$ in (T-assign) and using the hypothesis $C \subset \text{Activity}(M(C, r_e, \text{onCreate}) \subset \text{void}\{\bar{z} \rightarrow \text{void}\})$ with the well-typed program condition in (T-invoke). It is easy to construct the rest of hypotheses necessary to apply Lemma 2 with to result in an expression evaluation $\mathcal{E} \triangleright e, \emptyset, \emptyset, h \rightarrow l', w', q', h'$. This leads to an activation flow by (launch) as $l, w, q, h \rightarrow l' \cdot l, w', q', h'$. $h'(l')$ is another activity (of class
C) since the expression $e$ is defined to return it by $x$. By Lemma 2, $\text{effect}(Q') \subseteq \text{effect}(\emptyset) \cup \varphi = \emptyset \cup \varphi$, and, by def. of $\varphi$ over classes, we have $\varphi \subseteq \text{effect}(C)$ because $\varphi$ is the effect from invoking the onCreate method of $C$. To combine them, we have $\text{effect}(Q') \subseteq \text{effect}(C)$.

Let $D$s are the classes for the objects referred to as $l_i$ in $l$. Then $\text{flow}(D, C \cdot D)$ holds if $C \in \text{effect}(D_1)$, which is true because, by the intent abstraction (Lemma 1), $\text{target}(q) = C \in \text{effect}(Q)$ and, by the hypothesis in the well-formedness of quadruples, $\text{effect}(Q) \subseteq \text{effect}(D_1)$. The flow condition $\text{flow}(D, C \cdot D)$ of the lemma holds because of $C \in \text{effect}(D_2)$.

**Case (button).** Let $E$ be $\{x \mapsto l_1, b \mapsto i\}$ and let $\Gamma$ be $\{x : C_1\{r_1\}, b : \text{int}\{\}\}$. By the well-formedness of quadruples, $(H(l_1)) = C_1\{r_1\}$. Then we have $H \triangleright E : \Gamma$ by environment typing. By the well-typed program condition and by $C_1 <: \text{Activity}$, we have $M(C_1, r_1, \text{onClick}) <: \text{int}\{\} \rightarrow \text{void}\{\}$ for some $\varphi$. This allows to apply (T-invoke), giving a typing judgment $\Gamma \triangleright x.\text{onClick}() : \text{void}\{\}, \varphi$. It is easy to verify the rest of hypotheses necessary to apply Lemma 2.

The application of Lemma 2 allows to have an expression evaluation $E \triangleright x.\text{onClick}() \rightarrow void, w', q', h'$ and $\text{effect}(Q') \subseteq \text{effect}(Q) \cup \varphi$. Then (button) allows a quadruple evaluation $l_1 \cdot l, w', q', h' \rightarrow l_1 \cdot l, w', q', h'$.

Now let us verify the well-formedness of $(l_1 \cdot l, w', q', h')$. $\text{effect}(Q') \subseteq \text{effect}(C_1)$ holds by combining two things since $\text{effect}(Q') \subseteq \text{effect}(Q) \cup \varphi$. We have $\text{effect}(Q) \subseteq \text{effect}(C_1)$ by the well-formedness hypothesis of $(l_1 \cdot l, w, q, h)$. The effect of calling the onClick method of $C_1$ is $\varphi$ by the typing judgment above, and so we have $\varphi \subseteq \text{effect}(C_1)$. The rest of the well-formedness hypotheses of $(l_1 \cdot l, w, q, h')$ come from the well-formedness hypotheses of $(l_1 \cdot l, w, q, h)$ and the application of Lemma 2.

The flow condition $\text{flow}(C_1 \cdot C_1, C_1 \cdot C_1)$ of the lemma trivially holds.

**Case (back-1).** Let $E$ be $\{x \mapsto l_2\}$ and let $\Gamma$ be $\{x : C_2\{r_2\}\}$. By the well-formedness of quadruples, $(H(l_2)) = C_2\{r_2\}$. Then we have $H \triangleright E : \Gamma$ by environment typing.

By the well-typed program condition and by $C_2 <: \text{Activity}$, we have $M(C_2, r_2, \text{onCreate}) <: () \rightarrow \text{void}\{\}$ for some $\varphi$. This allows to apply (T-invoke), giving a typing judgment $\Gamma \triangleright x.\text{onCreate}() : \text{void}\{\}, \varphi$. It is easy to verify the rest of hypotheses necessary to apply Lemma 2. By the application of the lemma, we have $E \triangleright x.\text{onCreate}(), \emptyset, \emptyset, h \rightarrow void, w', q', h'$ and $\text{effect}(Q') \subseteq \text{effect}(\emptyset) \cup \varphi = \emptyset \cup \varphi$. This makes $l_1 \cdot l_2 \cdot l, w, q, h \rightarrow l_1 \cdot l, w', q', h'$.

$(l_2 \cdot l, w', q', h')$ can be verified to be well-formed. First, $\text{effect}(Q') \subseteq \text{effect}(C_2)$ because $\text{effect}(Q') \subseteq \varphi$ by the application of Lemma 2 and $\varphi \subseteq \text{effect}(C_2)$ by the fact that the effect $\varphi$ comes from invoking the onCreate method of $C_2$. The rest of the well-formedness hypotheses of $(l_2 \cdot l, w', q', h')$ come from the well-formedness hypotheses of $(l_2 \cdot l, w', q', h')$ and the application of Lemma 2.

The flow condition $\text{flow}(C_1 \cdot C_2 \cdot C_1, C_2 \cdot C_1)$ holds because $C_{k-1} \in \text{effect}(C_k)$ for $k = 2, \ldots, n$ by the well-formedness of $(l_1 \cdot l_2 \cdot l, w', q', h')$.

**Case (back-2).** The well-formedness of $(l_1 \cdot l, w, q, h)$ includes the condition $\triangleright h : H$, implying that $(\emptyset, \emptyset, \emptyset, h)$ is well-formed. The flow condition $\text{flow}(C_1, \emptyset)$ also holds.

### Appendix E. Proof of Lemma 1 (Intent Abstraction)

Let $h(l)$ be $D\{\text{target} = l, \text{action} = l_a, \ldots\}$ where $D <: \text{Intent}$. $\text{Case } l_t \neq \text{null}. \text{ } h(l_t) = "C" \text{ for some string since the type of the target field is String. Then } C\{\text{Class}(h(l_t))\} = C$. By heap typing with $l_t$, there exists $r_t$ such that $\text{H}(l_t) = \text{String}\{r_t\}$ and $\Omega(r_t) = \{"C"\}$. Also, by heap typing with $l$, we have $H \triangleright l_t : F(\text{Intent}, r, \text{target})$ for some $r$, and so $\text{String}(r_t) <: F(\text{Intent}, r, \text{target})$ where $F(\text{Intent}, r, \text{target}) = \text{String}\{R_t\}$ and $r_t \in R_t$ for some $R_t$. By def. of $\text{effect}(S)$ and by $r_t \in R_t, C \in \text{Class}(\text{"C"}) \subseteq \text{Class}(\Omega(r_t)) \subseteq \text{effect}(\text{Intent}\{R\})$.

$\text{Case } l_t = \text{null} \text{ and } l_a \neq \text{null}$. We apply the similar argument as above to this case to have $h(l_a) = "\text{action}"$, $\text{H}(l_a) = \text{String}\{r_a\}$, $\Omega(r_a) = \{"\text{action}"\}$, and $\text{String}(r_a) <: \text{String}\{R_a\} = F(\text{Intent}, r, \text{action})$ for some $r_a$ and $R_a$. Then $\text{IntentFilter("action")}$ either returns $C$ or raises activity-not-found error, depending on the presence of a pair ("action", C) in the intent filter. Unless we get activity-not-found error, $C = \text{IntentFilter("action")} \subseteq \text{IntentFilter}(\Omega(r_a)) \subseteq \text{effect}(\text{Intent}\{R\})$. 


Case $l_t = null$ and $l_a = null$. We get null error in the evaluation of the target calculating function. \hfill \square

Appendix F. Implementation

We have implemented in Haskell the proposed type and effect system for featherweight Android/Java as an automatic analyzer, which is available in http://mobilesw.yonsei.ac.kr/paper/android.html.

Our Android analyzer takes five steps: it first applies the standard Java type checking procedure to an Android program to attach Java types to the abstract syntax tree. Second, it collects all classes declared and referred in the program and all program points, i.e., program points for object creation sites. Third, it initializes field and method typings for the classes and program points by annotating the Java types with program point set variables and effect variables. Fourth, it generates subtyping constraints and activation constraints on the variables by applying the typing rules to each method typing. Fifth, it solves the constraints to produce a solution (mapping of the variables onto ground literals, e.g., $\Omega(r5) = \{"Game"\}$).

Third, the analyzer initializes field and method typings by annotating the Java types with program point set and effect variables, $X_is$ and $E_is$. For example,

- $F(Intent, r4, target) = String\{X_1\}$
- $M(Intent, r4, setTarget) = String\{X_2\} \xrightarrow{E_1} void()$
- $M(Main, r_main, onClick) = int\{X_3\} \xrightarrow{E_2} void()$
- $M(Activity, r_main, startActivity) = Intent\{X_4\} \xrightarrow{E_3} void()$
- $F(Activity, r_game, intent) = Intent\{X_5\}$

In field and method typings, we group triples, (class, program point, field) or (class, program point, method), as this: each program point is an object creation site where we know the class name, and the preliminary Java type checking lets us know which fields and methods each class has and inherits.

Fourth, the analyzer applies our typing rules to the body of each method mentioned in the method typings to verify that the Android program is well-typed. This derivation will generate some constraints to resolve. For example, to have $M(Main, r_main, onClick)$ as the third step, we should derive \{this : Main\{r_main\}, button : int\{X_3\}\} $\triangleright e : void\{\}$, $E_2$ for the method body $e$, resolving all including some from Line 4-6 as,

- $String\{X_2\} \xrightarrow{E_2} void\{\}$ $\triangleright String\{r_3\} \xrightarrow{E_2} void\{\}$, $Intent\{X_4\} \xrightarrow{E_3} void\{\} \triangleright String\{r_4\} \xrightarrow{E_3} void\{\}$, $Intent\{X_4\} \triangleright E_4$, $E_4 \cup E_5 \subseteq E_2$, and $E_6 \subseteq E_3$.

Note that a new form of the constraint $Intent\{S\} \triangleright \varphi$, read as intents of type $Intent\{S\}$ activating classes in $\varphi$, is introduced to enforce the Android invariant in (T-prim-1). In the next step, solving this new constraint will generate an extra constraint as $Intent\{X_4\} \triangleright : Intent\{X_5\}$, to pass
each intent of type $\text{Intent}\{X_4\}$ to the field $\text{intent}$ of type $\text{Intent}\{X_5\}$ in $\text{Game}$ that the intent activates.

To guarantee the well-formedness of field and method typings, it is enough to introduce a subtyping constraint between the method types for each pair of a method and the overridden method.

Lastly, the analyzer solves the constraints by the conventional algorithm. The least solution of the constraints above is $X_1 = X_2 = \{r_5\}$, $X_4 = X_5 = \{r_4\}$, $E_1 = E_4 = \{\}$, $E_3 = E_5 = E_6 = \{\text{Game}\} \subseteq E_2$. Therefore, the analyzer detects that $\text{Main}$ may activate $\text{Game}$. 
