Information Processing Letters 114 (2014) 620–627

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A type and effect system for activation flow of components in Android programs

Kwanghoon Choi [a,1], Byeong-Mo Chang [b,*,2]

[a] Yonsei University, Wonju, Republic of Korea
[b] Sookmyung Women's University, Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 1 August 2013
Received in revised form 23 March 2014
Accepted 22 May 2014
Available online 27 May 2014
Communicated by M. Yamashita

Keywords:
Android
Java
Program analysis
Control flow
Formal semantics

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.

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 [2–5] 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, one has 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.

* Corresponding author.
E-mail addresses: kwanghoon.choi@yonsei.ac.kr (K. Choi), chang@sookmyung.ac.kr (B.-M. Chang).

[1] This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF 2011-0009225).
[2] This research was supported by the Sookmyung Women's University (Research Grants 1-1403-0212).

http://dx.doi.org/10.1016/j.ipl.2014.05.011
0020-0190/

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 activation 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 [7] with Android features, which demands us to introduce a simple string analysis [8] 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].

Component type	Method for launching
Activity	startActivity(Intent)
Service	startService(Intent)
Broadcast Receiver	sendBroadcast(Intent)

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 Figs. 1 and 5 using the APIs shown in Fig. 2, whose details will be explained later.

```
class Score extends Activity {
    void onCreate() {
        this.addButton(1);
        // display the score screen
    }
    void onClick(int button) {
L1:     Intent i = new Intent();
L2:     i.setTarget("Main");
L3:     this.startActivity(i);
    }
}
```

Fig. 1. Score class in a game program.

Let us examine a Java class of the Android program in Fig. 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*.

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 [8] 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 [9].

$$N ::= \text{class } C \text{ extends } C\ \{\bar{C}\ \bar{f};\ \bar{M}\}$$

$$M ::= C\ m(\bar{C}\ \bar{x})\ \{\ e\ \}$$

$$e ::= x \mid x.f \mid \text{new } C() \mid x.f = x \mid (C)x \mid x.m(\bar{x})$$

$$\mid \text{if } e \text{ then } e \text{ else } e \mid Cx = e;\ e \mid prim(\bar{x})$$

A list of class declarations $\bar{N}$ denotes an Android program. A block expression $C\ x = e; e'$ 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 $C\ x$. The conditional expression may be written as ite $e\ e\ e$ for brevity. We write a string object as a "string literal." Also, x.m("...") means *String* $s =$ "..."; $x.m(s)$ in shorthand. A recursive method offers a form of loops.

```
class Activity {
    Intent intent;
    Intent getIntent() { this.intent; }
    void onCreate() { }
    void onClick(int button) { }
    void addButton(int button)
        { primAddButton(button); }
    void startActivity(Intent i)
        { primStartActivity(i); }
}
class Intent {
    String target;
    Object data;
    String action;
    // The setter and getter methods
    // for the above fields
}
```

Fig. 2. Android classes: activity and intent.

The primitive functions $prim(\bar{x})$ 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 Fig. 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 $C\{\bar{f} = \bar{l}\}$ with the fields $\bar{f}$ and their values $\bar{l}$. For example, $Intent\{target = l, data = l', action = l''\}$ 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 $(\bar{l}, w, q, h)$ of an activity stack $\bar{l}$, 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. $\bar{l}, w, q, h \Longrightarrow \bar{l}', w', q', h'$ denotes an activation flow between the two top activity components l_1 and l_1', which is activated by the intent q. $\bar{l}$ and $\bar{l}'$ may be the same. $\Longrightarrow^*$ denotes zero or more steps.

A stack of activities [1] is a list $\bar{l}$ 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 $\mathcal{E} \rhd e, w, q, h \longrightarrow l, w', q', h'$.

$$\text{(run)}\ \frac{\emptyset \rhd e, \emptyset, \emptyset, \emptyset \longrightarrow l, w, q, h \quad e = (C\ x = \text{new } C(); x.onCreate(); x)}{run\ C \Longrightarrow l, w, q, h}$$

$$\text{(launch)}\ \frac{C = \text{target}(q, h) \quad e = (C\ x = \text{new } C(); \ x.intent = intent;\ x.onCreate();\ x) \quad \{intent \mapsto q\} \rhd e, \emptyset, \emptyset, h \longrightarrow l', w', q', h'}{\bar{l}, w, q, h \Longrightarrow l' \cdot \bar{l}, w', q', h'}$$

$$\text{(button)}\ \frac{i \in w, \quad e = x.onClick(b) \quad \{x \mapsto l, b \mapsto i\} \rhd e, w, q, h \longrightarrow void, w', q', h'}{l \cdot \bar{l}, w, q, h \Longrightarrow l \cdot \bar{l}, w', q', h'}$$

$$\text{(back-1)}\ \frac{e = x.onCreate() \quad \{x \mapsto l_2\} \rhd e, \emptyset, \emptyset, h \longrightarrow void, w', q', h'}{l_1 \cdot l_2 \cdot \bar{l}, w, q, h \Longrightarrow l_2 \cdot \bar{l}, w', q', h'}$$

$$\text{(back-2)}\ l_1 \cdot \emptyset, w, q, h \Longrightarrow \emptyset, \emptyset, \emptyset, h$$

Fig. 3. Semantic rules for the Android platform.

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 target class name or implicitly by suggesting only actions. The former is called explicit intents, useful for the intra-application components, and the latter is called implicit intents, useful for the inter-application components [1]. Our Android semantics models both of explicit and implicit intents.

To pick a target activity class from an explicit/implicit intent reference, we define a function $\text{target}(l, h)$ as: Suppose $h(l) = Intent\{target = l_t, action = l_a, \ldots\}$, and then the function returns

- $Class(h(l_t))$ if $l_t \neq null$
- $IntentFilter(h(l_a))$ if $l_t = null$ and $l_a \neq null$

where $Class(\text{“}C\text{”}) = C$ such that C is an activity class, and where $IntentFilter(\text{“}action_i\text{”}) = C_i$, a mapping table of actions (strings describing services) onto activity classes that are capable of supporting the services. When the $\text{target}(l, h)$ fails to find any activity class, it is defined to return activity-not-found error.

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

In Fig. 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 $\bar{l}$ to a new one l' through the intent by q. The rule begins when q is an intent reference($q \neq \emptyset$). 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 *onClick* method of the current activity when

$$\text{(var)}\quad \mathcal{E} \rhd x, w, q, h \longrightarrow \mathcal{E}(x), w, q, h$$

$$\text{(field)}\quad \frac{\mathcal{E}(x) = l_x \qquad h(l_x) = C\{\bar{f} = \bar{l}\}}{\mathcal{E} \rhd x.f_i, w, q, h \longrightarrow l_i, w, q, h}$$

$$\text{(assign)}\quad \frac{\mathcal{E}(x) = l_x,\ h(l_x) = C\{\bar{f} = \bar{l}\},\ \mathcal{E}(y) = l_y \qquad h' = h\{l_x \mapsto C\{\bar{f} = \bar{l}_{1,i-1} l_y \bar{l}_{i+1,n}\}\}}{\mathcal{E} \rhd x.f_i = y, w, q, h \longrightarrow \mathit{void}, w, q, h'}$$

$$\text{(new)}\quad \frac{\mathsf{fields}(C) = \bar{D}\ \bar{f},\ l \text{ fresh} \qquad h' = h\{l \mapsto C\{\bar{f} = \bar{null}\}\}}{\mathcal{E} \rhd \mathsf{new}\ C(), w, q, h \longrightarrow l, w, q, h'}$$

$$\text{(cast)}\quad \frac{\mathcal{E}(x) = l,\ h(l) = D\{\bar{f} = \bar{l}\},\ D <: C}{\mathcal{E} \rhd (C)x, w, q, h \longrightarrow l, w, q, h}$$

$$\text{(if)}\quad \frac{\mathcal{E} \rhd e_0, w, q, h \longrightarrow b_0, w_0, q_0, h_0 \qquad \text{if } b_0 \text{ then } i = 1 \text{ else } i = 2. \qquad \mathcal{E} \rhd e_i, w_0, q_0, h_0 \longrightarrow l', w', q', h'}{\mathcal{E} \rhd \mathsf{ite}\ e_0\ e_1\ e_2, w, q, h \longrightarrow l', w', q', h'}$$

$$\text{(block)}\quad \frac{\mathcal{E} \rhd e_0, w, q, h \longrightarrow l_0, w_0, q_0, h_0 \qquad \mathcal{E}\{x \mapsto l_0\} \rhd e, w_0, q_0, h_0 \longrightarrow l, w', q', h'}{\mathcal{E} \rhd C\ x = e_0; e, w, q, h \longrightarrow l, w', q', h'}$$

$$\text{(invoke)}\quad \frac{\mathcal{E}(x) = l,\ h(l) = C\{\bar{f} = \bar{l'}\},\ \mathcal{E}(\bar{y_i}) = \bar{l_i} \qquad \mathsf{mbody}(m, C) = \bar{B}\ \bar{z}.e \qquad \mathcal{E}_0 = \{\mathsf{this} \mapsto l, \bar{z} \mapsto \bar{l_i}\} \qquad \mathcal{E}_0 \rhd e, w, q, h \longrightarrow l', w', q', h'}{\mathcal{E} \rhd x.m(\bar{y}), w, q, h \longrightarrow l', w', q', h'}$$

$$\text{(prim-1)}\quad \frac{\mathit{prim} \text{ is } \mathit{primStartActivity}}{\mathcal{E} \rhd \mathit{prim}(x), w, q, h \longrightarrow \mathit{void}, w, \mathcal{E}(x), h}$$

$$\text{(prim-2)}\quad \frac{\mathit{prim} \text{ is } \mathit{primAddButton} \qquad w' = w \cup \{\mathcal{E}(x)\}}{\mathcal{E} \rhd \mathit{prim}(x), w, q, h \longrightarrow \mathit{void}, w', q, h}$$

Fig. 4. Semantic rules for expressions.

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 l_1 to resume the next top activity l_2 by calling the *onCreate* 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 (l_1) may be called before it is hidden by a new one (l'). In (back-1) and (back-2), the *onDestroy* method of the top activity l_1 may be called before it is removed from the activity stack. Also, instead of calling the *onCreate* method in (back-1), we may call the *onResume* method of the activity l_2 to prepare the reappearance of the hidden activity (l_2).

The semantic rules for evaluating expressions are defined as in Fig. 4, which is mostly standard [9,7]. The main difference is an introduction of a window set and an intent reference to the semantic rules as our Android runtime system. The standard Java constructs such as variable, field, and method invocation do not access nor change them. *primStartActivity*(x), which we introduce, replaces the current intent reference q with a new intent reference bound to x. Also, *primAddButton*(x) adds a new button whose identifier is bound to x.

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 Figs. 3 and 4.

The semantic rules use some auxiliary functions defined in [9]. $\mathsf{mbody}(m, C)$ returns the body expression of the method of the class, and $\mathsf{fields}(C)$ 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 Figs. 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 [7,10] extended with a simple string analysis [8] 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 Fig. 5, the intent variable j has (annotated) type $\mathit{Intent}\{r10, r14\}$. This is because the reference in j points to either an intent object created in Line 10 (denoted by a program point r10) or one in Line 14 (denoted by another r14). 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 φ 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 φ will be a set of component (class) names.

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

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

$$\Gamma \rhd e : C\{R\}, \varphi$$

where, under the typing environment Γ (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 φ might occur.

Our typing rules for expressions in Fig. 6 depend 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 $\bar{S} \xrightarrow{\varphi} T$ 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.

```
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);
    }
}
```

Fig. 5. A game program.

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

- $F(Help, r_{help}, intent) = Intent\{r10, r14\}$

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

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

- $M(Score, r_{score}, onClick) = int\{r_{button}\} \xrightarrow{\{Main\}} void\{\}$

where r_{button} is another program point to identify integers *i* created at (button). In Line 2, the *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 *onClick* method is {*Main*} obviously.

However, target component names set by the *setTarget* method are not always obvious. In Line 25 (the *onClick* method of *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 $String\{R\}$ will evaluate to some string in the union of sets of strings $\Omega(r)$ for all $r \in R$. For example, $\Omega(r12) = \{$"*Main*"$\}$ and $\Omega(r16) = \{$"*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 $String\{r12, r16\}$, and so *s* will evaluate to a string in {"*Main*", "*Game*"}.

The string analysis in our system explained until now can be regarded as [7]. 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 Fig. 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 $R_1 \subseteq R_2$; $\bar{S}_i \xrightarrow{\varphi_1} S <: \bar{T}_i \xrightarrow{\varphi_2} T$ 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 $M(C, R, m) <: \bar{S}_i \xrightarrow{\varphi} T$ 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

$$\text{(T-var)}\quad \Gamma\{x:S\} \rhd x : S, \emptyset$$

$$\text{(T-field)}\quad \frac{F(C,R,f) <: S}{\Gamma\{x:C\{R\}\} \rhd x.f : S, \emptyset}$$

$$\text{(T-assign)}\quad \frac{S <: F(C,R,f)}{\Gamma\{x:C\{R\}, y:S\} \rhd x.f = y : void\{\}, \emptyset}$$

$$\text{(T-new)}\quad \Gamma \rhd \text{new } C() : C\{r\}, \emptyset \quad \text{for unique } r$$

$$\text{(T-cast)}\quad \frac{C1 <: C2 \quad \text{or} \quad C2 <: C1}{\Gamma\{x:C1\{R\}\} \rhd (C2)x : C2\{R\}, \emptyset}$$

$$\text{(T-if)}\quad \frac{\Gamma \rhd e_0 : boolean\{R\}, \varphi_0 \quad \Gamma \rhd e_i : S, \varphi_i \ (i=1,2)}{\Gamma \rhd \text{ite } e_0\ e_1\ e_2 : S, \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

$$\text{(T-block)}\quad \frac{\Gamma \rhd e_1 : C\{R\}, \varphi_1 \quad \Gamma\{x:C\{R\}\} \rhd e_2 : S, \varphi_2}{\Gamma \rhd C\ x = e_1;\ e_2 : S, \varphi_1 \cup \varphi_2}$$

$$\text{(T-invoke)}\quad \frac{M(C,R,m) <: \bar{S}_i \xrightarrow{\varphi} T}{\Gamma\{x:C\{R\}, y_i:S_i\} \rhd x.m(\bar{y}) : T, \varphi}$$

$$\text{(T-sub)}\quad \frac{\Gamma \rhd e : S, \varphi_1 \quad S <: T \quad \varphi_1 \subseteq \varphi_2}{\Gamma \rhd e : T, \varphi_2}$$

$$\text{(T-string)}\quad \frac{\text{“}s\text{”} \in \Omega(r) \text{ for unique } r}{\Gamma \rhd \text{“}s\text{”} : String\{r\}, \emptyset}$$

$$\text{(T-prim-1)}\quad \frac{(x:S) \in \Gamma \qquad \text{effect}(S) = \varphi \quad S <: F(C, r_c, intent) \text{ for all } C \in \varphi}{\Gamma \rhd primStartActivity(x) : void\{\}, \varphi}$$

$$\text{(T-prim-2)}\quad \frac{(x : int\{R\}) \in \Gamma}{\Gamma \rhd primAddButton(x) : void\{\}, \emptyset}$$

Fig. 6. A type and effect system.

side y to the field $x.f$. (T-new) abstracts all objects generated in each *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 subexpressions. (T-invoke) 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 [7] 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

$$S <: F(C, r_c, intent) \quad \text{for all } C \in \text{effect}(S)$$

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

(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 φ satisfying the following conditions: S is $Intent\{R\}$ and, for each $r \in R$, there are R_t and R_a such that $F(Intent, r, target) = String\{R_t\}$ and $F(Intent, r, action) = String\{R_a\}$. Then,

- $Class(\Omega(r_t)) \subseteq \varphi$ for all $r_t \in R_t$
- $IntentFilter(\Omega(r_a)) \subseteq \varphi$ for all $r_a \in R_a$

For example, intent objects at $r10$ and $r14$ are passed to *Help* in Line 13 and 17 respectively as:

- $M(Main, r_{main}, startActivity) = Intent\{r_{10}\} \xrightarrow{\{Help\}} void\{\}$
- $M(Game, r_{game}, startActivity) = Intent\{r_{14}\} \xrightarrow{\{Help\}} void\{\}$

Due to the intent passing, (T-prim-1) forces both $Intent\{r10\}$ and $Intent\{r14\}$ to be a subtype of $F(Help, r_{help}, intent)$, which is $Intent\{r10, r14\}$. In Line 26, the type of i is $Intent\{r21\}$. $\text{effect}(Intent\{r21\})$ 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 $Intent\{r10, r14\}$. The data field of intents of the type evaluates to “*Main*” or “*Game*” because of

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

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

- $M(Help, r_{help}, startActivity) = Intent\{r_{21}\} \xrightarrow{\{Main, Game\}} void\{\}$

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 [7], 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 $\bar{S} \xrightarrow{\varphi} T$ such that $M(C,r,m) = \bar{S} \xrightarrow{\varphi} T$, we can derive $\{this : C\{r\}, \bar{x} : \bar{S}\} \rhd e : T, \varphi$ where $\text{mbody}(m, C) = D\ \bar{C}\ \bar{x}.\ e$ ($\bar{C}$ 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 $flow(\bar{C}, \bar{D})$ for two lists of activity classes $\bar{C}$ and $\bar{D}$ to declare that each activity on the stack is activated by an activity directly under itself. The proposition is true if

- $\bar{D} = \bar{C}$, $\bar{D} = C_0 \cdot \bar{C}$, or $\bar{D} = \bar{C}_{2..n}$

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

We extend it to more than two lists of activity classes as: $flow^*(\bar{C}, \bar{D}_1, ..., \bar{D}_m)$ is true if $flow(\bar{C}, \bar{D}_1)$, ..., and $flow(\bar{D}_{m-1}, \bar{D}_m)$ are all true.

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

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

- *run* $C \implies \bar{l}_1, w_1, q_1, h_1 \implies^* \bar{l}_m, w_m, q_m, h_m$ *such that* $flow^*(C, \bar{D}_1, ..., \bar{D}_m)$ *where* $\bar{l}_i \sim \bar{D}_i$.

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

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

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

Otherwise, the evaluation of $\mathsf{target}(l, h)$ *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 security problems. ScanDroid [2] is a security certification tool to check if data flows in Android programs are consistent with their specifications. ComDroid [3] searches for predefined patterns of potential vulnerabilities. ScanDal [4] analyzes data flows between Android security sources and sinks. EPIC [5] is a scalable inter-procedural analysis for detecting attacks for Android vulnerabilities.

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 [7], but it also introduce a new form of constraints $Intent\{S\} \Rightarrow \varphi$ for each use of *primStartActivity*(...) in an Android program. We call this an activation constraint. Solving each activation constraint is to generate a new subtyping constraint $S <: F(C, r_c, intent)$ whenever a new class C becomes belong to the effect of this intent type S, $\mathsf{effect}(S)$. 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

[1] The Android developers site, http://developers.android.com, 2013.

[2] A.P. Fuchs, A. Chaudhuri, J.S. Foster, Scandroid: automated security certification of Android applications, Technical Report CS-TR-4991, Dept. of Computer Science, University of Maryland, 2009.

[3] E. Chin, A.P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in Android, in: Proceedings of the 9th Annual Int'l Conference on Mobile Systems, Applications, and Services, 2011.

[4] J. Kim, Y. Yoon, K. Yi, J. Shin, SCANDAL: static analyzer for detecting privacy leaks in Android applications, in: Mobile Security Technologies, 2012.

[5] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. Le Traon, Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis, in: 22nd USENIX Security Symposium, 2013, pp. 543–558.

[6] T. Amtoft, F. Nielson, H.R. Nielson, Type and Effect Systems: Behaviors for Concurrency, World Scientific Publishing Company, 1999.

[7] L. Beringer, R. Grabowski, M. Hofmann, Verifying pointer and string analyses with region type systems, in: Proceedings of the 16th Int'l Conference on Logic for Programming, Artificial Intelligence, and Reasoning, 2010, pp. 82–102.

[8] A.S. Christensen, A. Møller, M.I. Schwartzbach, Precise analysis of string expressions, in: Proceedings of the 10th Int'l Static Analysis Symposium, 2003, pp. 1–18.

[9] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450.

[10] A. Milanova, A. Rountev, B.G. Ryder, Parameterized object sensitivity for points-to analysis for Java, ACM Trans. Softw. Eng. Methodol. 14 (1) (2005) 1–41.

[11] K. Choi, B. Chang, A type and effect system for activation flow of components in Android programs, Technical Report TR-Mar-2014-1, Yonsei University, Wonju, 2014.