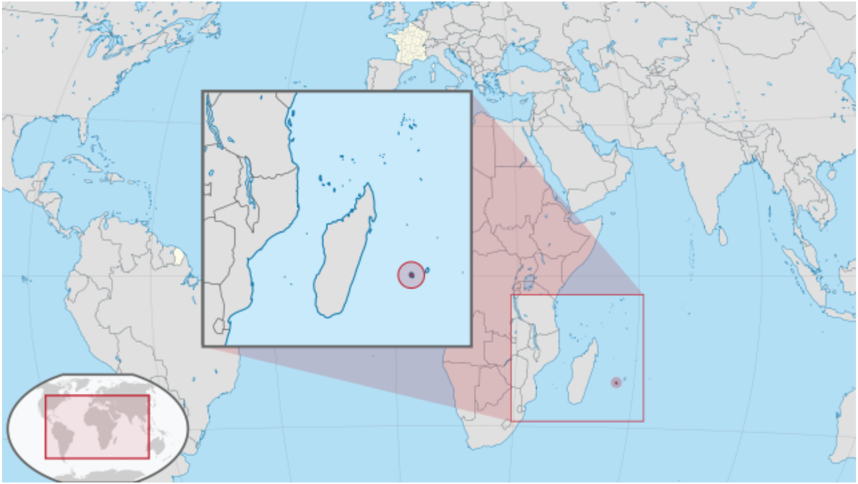
Non-termination of Dalvik bytecode via compilation to CLP

Étienne Payet and **Fred Mesnard**

LIM, université de la Réunion

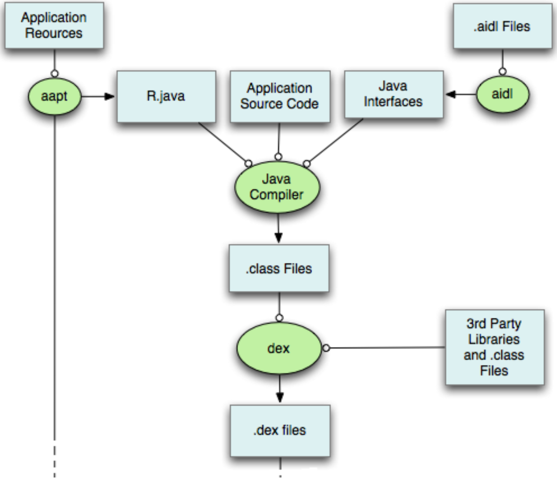
WST'14

Reunion, a part of France and Europe



Outline

Building an Android application



http://developer.android.com/tools/building/index.html

.dex files

- their format is optimized for minimal memory usage
- they contain Dalvik bytecode
- dex stands for Dalvik executable

Outline

Dalvik bytecode

- is run by an instance of the Dalvik Virtual Machine (DVM)
- DVM $\neq$ JVM (register-based vs stack-based)
- register-based VMs better suited for devices with constrained processing power

Dalvik registers

- each method has a fresh array of registers
- invoked methods do not affect the registers of invoking methods

Some Dalvik instructions

- *const* d, c
  move constant c into register d
- *move* d, s
  move the content of register s into register d
- *add* d, s, c
  store the $+$ of the content of register s and constant c into register d
- *if-lt* i, j, q
  if the content of register i is less than the content of register j then jump to program point q, otherwise go on
- *goto* q
  jump to program point q
- *return*
  return from a void method
- *new-instance* d, κ
  move a reference to a new object of class κ into register d

Some Dalvik instructions

- *invoke* $S, meth$ ($S = s_0, s_1, \ldots, s_p$ is a sequence of register indexes) The content r^{s_0} of register s_0, ..., r^{s_p} of register s_p are the *actual parameters* of the call. Value r^{s_0} is called *receiver* of the call and must be 0 (the equivalent of `null` in Java) or a reference to an object o. In the former case, the computation stops with an exception. Otherwise, a *lookup procedure* is started from the class of o upwards along the superclass chain, looking for a method with the same signature as m. That method is run from a state where its last registers are bound to $r^{s_0}, r^{s_1}, \ldots, r^{s_p}$.
- *iget* d, i, f (resp. *iput* s, i, f)
  The content r^i of register i must be 0 or a reference to an object o. If r^i is 0, the computation stops with an exception. Otherwise, $o(f)$ (the value of field f of o) is stored into register d (resp. the content of register s is stored into $o(f)$).

Outline

Memory model

- a *memory* is a pair (a, i) where a is an array of *objects* and i is the index into this array where the next insertion will take place
- an *object* o is an array of terms of the form $[w, f_1(v_1), \ldots, f_n(v_n)]$ where w is the name of the class of o, f_1, ..., f_n are the names of the fields defined in this class and v_1, ..., v_n are the current values of these fields in o

The first component of a memory is an array of arrays of terms and a memory location is an index into this array.
Memory locations start at 1 and 0 corresponds to the `null` value.

Compilation rules: introduction

- we associate a predicate symbol p_q to each program point q of the Dalvik program
- we generate clauses with constraints on integer and array terms
- $a[i]$ returns the value stored at position i of the array a
  $a\{i \leftarrow e\}$ is a modified so that position i has value e
- each rule considers an instruction *ins* occurring at a program point q
- let r is the number of registers used by a method m, for each $i \in [0, r-1]$, variable V_i (resp. V'_i) models the content of register i before (resp. after) executing m
- M denotes the input memory and M' the output memory
- $\tilde{V}$ and M (or $[A, I]$) in the head of the clauses are input parameters while M' is an output parameter

Compilation rules

const d, c moves constant c into register d, the output register variable V'_d is set to c while the other register variables remain unchanged (modelled with id_{-d})

$$\frac{const\ \ d, c}{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = c\} \cup id_{-d},\ \ p_{q+1}(\tilde{V}', M, M')}$$

$$\frac{if\text{-}lt\ \ i, j, q'}{\begin{array}{l} \{ \quad p_q(\tilde{V}, M, M') \leftarrow \{V_i < V_j\} \cup id,\ \ p_{q'}(\tilde{V}', M, M'), \\ \quad\ \ p_q(\tilde{V}, M, M') \leftarrow \{V_i \geq V_j\} \cup id,\ \ p_{q+1}(\tilde{V}', M, M') \quad \} \end{array}}$$

Compilation rules

$$\frac{\mathit{return}}{p_q(\tilde{V}, M, M') \leftarrow \{M' = M\}}$$

$$\frac{\mathit{invoke}\ s_0, \ldots, s_p, m}{\left\{ \begin{array}{rl|l} p_q(\tilde{V}, M, M') \leftarrow & \{V_{s_0} > 0\} \cup id, & m' \in sign(m) \\ & lookup_P(M, V_{s_0}, m, q_{m'}), & \text{and } \tilde{X}_{m'} = 0, \ldots, 0, V_{s_0}, \ldots, V_{s_p} \\ & p_{q_{m'}}(\tilde{X}_{m'}, M, M_1), & \text{with } |\tilde{X}_{m'}| = reg(m') \\ & p_{q+1}(\tilde{V}', M_1, M') & \end{array} \right\}}$$

Compilation rules

$$\frac{\textit{new-instance } d, \kappa \qquad w \text{ is the name of class } \kappa \text{ and } f_1, \ldots, f_n \text{ are the names of the fields defined in } \kappa}{\begin{array}{l} p_q(\tilde{V}, [A, I], M') \leftarrow \{O[0] = w,\ O[1] = f_1(0),\ \ldots,\ O[n] = f_n(0), \\ \qquad A_1 = A\{I \leftarrow O\},\ V'_d = I,\ I_1 = I + 1\} \cup id_{-d},\ p_{q+1}(\tilde{V}', [A_1, I_1], M') \end{array}}$$

$$\frac{\textit{iget } d, i, f}{p_q(\tilde{V}, [A, I], M') \leftarrow \{V_i > 0,\ A[V_i, F] = f(V'_d)\} \cup id_{-d},\ p_{q+1}(\tilde{V}', [A, I], M')}$$

$$\frac{\textit{iput } s, i, f}{\begin{array}{l} p_q(\tilde{V}, [A, I], M') \leftarrow \{V_i > 0,\ O = A[V_i],\ O[F] = f(X),\ O_1 = O\{F \leftarrow f(V_s)\}, \\ \qquad A_1 = A\{V_i \leftarrow O_1\}\} \cup id,\ p_{q+1}(\tilde{V}', [A_1, I], M') \end{array}}$$

Compilation rules

Theorem

Let P_{CLP} denote the CLP program resulting from the compilation of P.

P_{CLP} operationally mimics P.

Outline

Non-termination

Theorem

Let $r = p(\tilde{x}) \leftarrow c, p(\tilde{y})$ *and* $r' = p'(\tilde{x}') \leftarrow c', p(\tilde{y}')$ *be two CLP clauses. Suppose there exists a set* $\mathcal{G}$ *such that*

- $\left[\forall \tilde{x} \exists \tilde{y}\ \tilde{x} \in \mathcal{G} \Rightarrow (c \wedge \tilde{y} \in \mathcal{G})\right]$
- $\left[\exists \tilde{x}' \exists \tilde{y}'\ c' \wedge \tilde{y}' \in \mathcal{G}\right]$

are true. Then p' *has an infinite computation in* $\{r, r'\}$*:*
$r' \rightarrow r \rightarrow r \rightarrow \ldots$

An example

Consider the following Android program with the Java syntax on the left and the corresponding Dalvik bytecode *P* on the right, where v0, v1, ... denote registers 0, 1, ...

```
public class Loops {
  int i;
  public void m(int n, Loops x) {
    while (this.i < n) {
      this.i++;
      x.i--;
    }
  }
}
```

```
.method public m(ILoops)V
    .registers 4
0:  iget v0, v1, Loops->i:I
1:  if-lt v0, v2, 3
2:  return-void
3:  iget v0, v1, Loops->i:I
4:  add-int/lit8 v0, v0, 0x1
5:  iput v0, v1, Loops->i:I
6:  iget v0, v3, Loops->i:I
7:  add-int/lit8 v0, v0, -0x1
8:  iput v0, v3, Loops->i:I
9:  goto 0
.end method
```

An example

The non-terminating method `loop` is called when the user taps a button. Execution of this method does not terminate because in the call to `m`, the objects `o1` and `o2` are aliased and therefore by decrementing `x.i` we are also decrementing `this.i` in the loop of method `m`.

```
public class MyActivity extends Activity {
  ...
  public void loop(View v) {
    Loops o1 = new Loops();
    Loops o2 = o1;
    o1.m(2, o2);
  }
  ...

}
```

```
.method public loop(Landroid/view/View;)V
    .registers 5
10:  new-instance v0, Loops
11:  invoke-direct {v0}, Loops-><init>()V
12:  move-object v1, v0
13:  const/16 v2, 0x2
14:  invoke-virtual {v0, v2, v1},
                        Loops->m(ILoops)V
15:  return-void
.end method
```

An example

E.g., we get the following clauses of P_{CLP} for program points 0 and 14:

$$
\begin{aligned}
p_0(\tilde{V}, [A, I], M') \leftarrow\ & \{A[V_1, F] = i(V_0')\} \cup id_{-0}, \\
& p_1(\tilde{V}', [A, I], M')
\end{aligned}
$$

$$
\begin{aligned}
p_{14}(\tilde{V}, M, M') \leftarrow\ & \{V_0 > 0\} \cup id, \\
& lookup_P(M, V_0, \texttt{Loops->m(ILoops)V}, 0), \\
& p_0(0, V_0, V_2, V_1, M, M_1), \\
& p_{15}(\tilde{V}', M_1, M')
\end{aligned}
$$

An example

The set of *binary unfoldings* of P_{CLP} contains:

$$
\begin{aligned}
r: \quad & p_0(\tilde{V}, [A, I], M') \leftarrow \{ V_1 > 0,\ O = A[V_1],\ O[F] = i(X),\ X < V_2, \\
& \quad O_1 = O\{F \leftarrow i(X+1)\},\ A_1 = A\{V_1 \leftarrow O_1\}, \\
& \quad V_3 > 0,\ O' = A_1[V_3],\ O'[F'] = i(X'),\ V'_0 = X' - 1, \\
& \quad O'_1 = O'\{F' \leftarrow i(V'_0)\},\ A_2 = A_1\{V_3 \leftarrow O'_1\}\} \cup id_{-0}, \\
& \quad p_0(\tilde{V}', [A_2, I], M')
\end{aligned}
$$

$$
\begin{aligned}
r': \quad & p_{10}(\tilde{V}, [A, I], M') \leftarrow \{ O[0] = loops,\ O[1] = i(0),\ A_1 = A\{I \leftarrow O\} \\
& \quad I_1 = I + 1,\ I > 0\}, \\
& \quad p_0((0, I, 2, I), [A_1, I_1], M_1)
\end{aligned}
$$

where r corresponds to the path $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow \cdots \rightarrow 9 \rightarrow 0$ and r' to the path $10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 0$ in P.

An example

P has an infinite execution from program point 10.

Details:

- In r', O corresponds to both o_1 and o_2, which expresses that o_1 and o_2 are aliased. Note that I, the address of O, is passed to p_0 both as second and fourth parameter, which corresponds in r to V_1 (`this` in method `m`) and V_3 (`x` in `m`).
- Moreover, when $V_1 = V_3$ in r, we have $O' = O_1$, $F' = F$ and $X' = X + 1$, hence $V_0' = X' - 1 = X$. Therefore, we have $O_1' = O$, so $A_2 = A$.
- The logical formulas of the non-termination theorem are true for $\mathcal{G} = \{(\tilde{v}, mem, mem') \in \mathcal{D}^3 | v_1 = v_3\}$.

Outline

Conclusion

Summary:

- a technique to detect potential loops in Dalvik bytecode

Future works:

- write a solver for array constraints and fully implement the technique
- extend the compilation rules by considering the operational semantics of components of Android

Thank you!

Questions?