Dex Support for Soot

Michael Markert   Frank Hartmann

May 9, 2012

1 Usage

To use the dex infrastructure you have to pass `-src-prec dex` at the commandline.

2 Dependencies

The dex support for Soot depends on smali's dexlib (support is developed on version 1.3.2). Smali can be found here https://code.google.com/p/smali/ and the `dexlib.jar` can be obtained by building smali. Only the `dexlib.jar` is necessary, not the rest of smali.

3 Architecture

3.1 Finding classes

Unlike a .class file a .dex file can contain more than one class. To support the Soot class find mechanisms `DexClassProvider` builds an index that maps the class names to the dex file that contains the class definition. This index is built by searching all .dex and .apk (which contain a `classes.dex`) in the load path and is saved in `SourceLocator`.

3.2 Sootification

The resolution of a SootClass happens in `DexResolver` and profits from an architecture that is highly symmetric to that of Soot. All elements of this architecture (with the exception of `DexClass` because `DexResolver` does that) know how to turn themselves into the Soot equivalent.

3.3 Jimplification

The Jimplification starts at `DexBody.jimplify` and is separated into 5 phases:

1. Adding `this` and parameter locals,
2. sequencially jimplifying instructions,
3. jimplifying deffered instructions,
4. trap processing
5. body transforming (splitting and typing of locals, and null treatment)

There is an instruction class for every group of Dalvik bytecodes (see `InstructionFactory` for the mapping of opcodes to instructions) that knows how to turn itself into the equivalent block of Jimple instructions. Those instructions are themselves grouped themselves under super classes to share common code. For special cases which did not translate directly into Jimple, see Pecularities in the Translation.

4 Restrictions of Dex Support

- Only instructions that are supported by dexlib are supported by us
- odex[1] instructions are not supported, but they may be obsolete by now as the android documentation shows no more trace of this

5 Pecularities in the Translation

5.1 DeferableInstruction

For a `goto` instruction (and by extension `if` and `switch`; `fill-array-data` as well) it is possible to jump further into the method body (with a positive offset). As the Translation from dexlib instructions to Jimple code happens sequentially, a jump with a positive offset cannot be turned into a Jimple instruction because the target Unit has to be known for that. To support this, the jimplification can be deferred until the rest of the method body has been translated. A deferred instruction will then be jimplified by a call to `deferredJimplify`.

5.2 DanglingInstruction

An instruction is dangling if its Jimple meaning depends on the instruction that follows it. This is the case with invocation instructions: If an invocation is followed by a `move-result` instruction it will be a RValue else be turned into an invocation statement. Another case is `filled-new-array` (in normal and in range flavor) it must be followed by a `move-object` instruction or it will be ignored. Only direct successors can resolve a dangling instruction.

5.3 `not-int` & `not-long` Bytecodes

Dalvik bytecode has other than Java bytecode (and Jimple) support for `not-int` and `not-long` instructions. The semantics are to do a bitwise not operation and we solved it by using the same technique as Java: `number xor -1`. This works because all integer numbers are two's complement. If unsigned integers would be introduced, this likely has to be changed.

5.4 fill-array-data Instruction

Dalvik offers a native instruction in order to fill an array with a static table filled with a maximum of 5 values of primitive types. Since Jimple does not offer Byte, Short or Char constants, those values of the static table have to be stored as a Integer type in Jimple.

5.5 Access modifiers

Dalvik supports 3 additional modifiers: constructor, synthetic and declared synchronized. They were added to `soot.Modifier` but not to its `toString` as Jasmin does not understand it.

5.6 Wide primitives in the method call

Dalvik has wide primitives which means that those primitives (long and double) occupy two registers instead of one. The second register is the one that immediately follows the first one (e.g. a wide occupies registers 23 and 24). We don't make any distinction between wide and normal primitives in the translation as Jimple does not know the difference. This leads to a problem with method calls: Here we retrieve the values in the specified registers as arguments for those calls. If a parameter is wide the next "register parameter" is not a parameter at all but the second part of the preceding parameter. To solve this we check if a parameter is wide and omit the next parameter in the conversion. The code for this resides in `soot.dex.instructions.MethodInvocationInstruction.buildParameters`.

[1] odex files are ahead-of-time optimized dex file

5.7 Typing of exceptions

The type of an exception in the `MoveExceptionInstruction` is not known at the point when the instruction is jimplified (only when traps are handled, after the instructions). Because of this we insert an untyped `IdentityStmt`. To retype it correctly in the trap phase we provided a `RetypeableInstruction` interface, that offers `setRealType(DexBody body, Type type)` and `retype()`. `retype()` is to be called after the locals have been split to avoid type spilling into otherwise reused registers.

5.8 Inclusion of java.lang.System

We included `java.lang.System` as a signature level basic class because we ran into errors in the test phase. `System` is only included with its shortname in the bytecode.

5.9 Const instructions

Const instructions introduce new primitive constants. There is a `const-wide` which may introduce new long and double and a `const` which may introduce new int and float. Those constants are introduced as byte patterns which smali transports as int respective long. There is no type information whatsoever besides how this constant is used. We solve this problem by looking for future instructions which use the register that is filled by the const instruction and infer if they expect a floating point number. If it is used as a floating point number it will be decoded with `Float.intBitsToFloat` respective `Double.longBitsToDouble`. To support this, a `DexlibAbstractInstruction` has to override `movesRegister` (to keep track of copies) `overridesRegister` (to determine when a copy is no longer used) and `isUsedAsFloatingPoint` (to actually infer the type). The use of a BodyTransformer along with an `UnitGraph` would have been a cleaner approach but on Jimple level we lack type information for likely uses of `float/double` (such as arithmetic) that is available via typed instructions (such as `add-float`).

5.10 Splitting of Registers

We maintain an array for every body that holds the Soot Locals[2] for the dex registers. To keep the right associations (instructions that access the same register number share the same Local object) we generate as many Locals as registers in a body exist. Because they are untyped we execute the `TypeAssigner` BodyTransformer at the end of a `DexBody.jimplify` and the `LocalSplitter` to generate the right Locals.

5.11 `null` Comparisons

For `null` comparisons `if-nez` and `if-eqz` are relevant. Jimple/Java distinguishes between 0 and null, Dalvik does not. To generate the right expressions we need to detect if we deal with objects (hence null) or with integers (hence 0). We do this with the BodyTransformer `DexNullTransformer` that is to be called after locals are split and detects `null` candidates, examines their use and rewrites `const`, `if-nez` and `if-eqz` as needed.

[2]mostly untyped