A Simple Type-Based Points-to Analysis

June 2, 2018

We take a simplified syntax from the website [1] for the Java Programming language, specified in EBNF.

$$
\begin{array}{ll}
\textit{prog} & ::= C^* \\
C & ::= \mathsf{class}\ A\ [\mathsf{extends}\ B]\ \{\mathsf{field\text{-}def};\ \mathsf{method\text{-}def}\} \\
\mathsf{field\text{-}def} & ::= f{=}e \mid \mathsf{field\text{-}def}_1; \mathsf{field\text{-}def}_2 \\
\mathsf{method\text{-}def} & ::= m(x)\ \{s\} \mid \mathsf{method\text{-}def}_1; \mathsf{method\text{-}def}_2 \\
s & ::= \ \epsilon \mid x{=}e \mid x.f{=}x \mid x.m(\mathsf{arg\text{-}list}) \\
 & \quad \mid \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid s_1; s_2 \\
e & ::= c \mid x \mid x.f \mid \mathsf{new}\ A(\mathsf{arg\text{-}list}) \mid x.m(\mathsf{arg\text{-}list}) \\
\mathsf{arg\text{-}list} & ::= e \mid e, \mathsf{arg\text{-}list} \\
c & ::= n \text{ where } n \in \mathbb{R}
\end{array}
$$

Fig. 1: The starting syntax of Java.

For this simplified Java syntax, we omit some of the basic Java syntax, such as basic types (e.g., boolean, int, double), constructor and field modifiers (e.g., public, private, static, final), the interface structure, exception handling, anonymous classes, structural braces and specialized keywords (e.g., this, super). We assume the ways of handling such structures can be added straightforwardly on this minimized syntax. Moreover, we assume that a given program has always correct syntax, otherwise it will not pass compilation. We assume values are within ranges of their defined variables, and we ignore the cases of overflows.

In this draft, we assume the program is well typed regarding class definitions. For example, in the following statement $x = \mathsf{new}\ D(\dots)$, we assume if x is defined as class C, then $D \sqsubseteq C$, which is read as D is a subclass (or subtype) of C (assuming the subtype relation is reflexive). However, for conciseness we omit all the reference definitions, and assume type compatibility throughout this draft. We assume every location is addressable (such as in its line number in source code), and add a little notation at object creation site in the form of $x = \mathsf{new}^o\ D(\dots)$, which means the location of this line is o. Here we have the base cases of our points-to constraints.

$$\frac{x = \mathsf{new}^o\ D(\dots)}{o \in \mathsf{Pts}(x)} \tag{1}$$

This rule says the location o representing this object is pointed to by reference x. For assignment to fields, we have the following rule.

$$\frac{x.f = \mathsf{new}^o\ D(\dots)}{\forall o' \in \mathsf{Pts}(x) : o \in \mathsf{HPts}(o'.f)} \tag{2}$$

Implicitly, we introduce the notion of *abstract heap*, which contains all heap objects created in a program in an abstract way. In the above rule, o' represents an arbitrary object in the abstract heap that may be pointed to by reference x. To extend the definition to heap objects, we define the HPts relation that stores the references that are defined as fields of heap objects. Again, we assume type compatibility regarding objects in the abstract heap, i.e., the type of o' may only be a subtype of the declared type of x. As a type-based analysis, we assume $\mathsf{Pts}(x)$ is an extension type attached to the variable definition of x, and $\mathsf{HPts}(o, f)$ is attached to the object creation site where o is initially defined.

The following extending rules are not surprising in Andersen's style [2] points-to construction. We omit the cases on other relationships between references and field, and assume f and g are legal fields of types declared for x and y. To this point, we assume all such compatibilities.

$$\frac{x = y}{\mathsf{Pts}(y) \subseteq \mathsf{Pts}(x)} \tag{3}$$

$$\frac{x.f = y.g}{\forall o \in \mathsf{Pts}(x), o' \in \mathsf{Pts}(y) : \mathsf{HPts}(y.o') \subseteq \mathsf{HPts}(x.o)} \tag{4}$$

Sometimes, in compiler frameworks, complicated statements can be decomposed into simple ones by introducing additional variables. For example, given a statement $x = y.z.f$, we may introduce a new reference t, such that $t = y.z$ and $x = t.f$. In this way we always get a set of statements that falls into the scope of our basic rules.

To establish all constraints we need to handle method calls. This may cause a bit of problem because given a statement $y = x.m(\dots)$, m may be a method that is defined in any subclass of the class that is defined for x. Though to precisely resolve call cases of polymorphism is theoretically impossible, it is still possible to find a set of class definitions that are pointed to by x, statically. In other words, we work on the set of methods m that are defined by classes of objects $o \in \mathsf{Pts}(x)$, which is the key of the *call relation* we need to construct. Moreover, calculation of the points-to relations Pts and HPts also depends on the call relation, as references are passed explicitly from actual parameters to formal parameters inside methods, and also passed back from return value (from $x.m(\dots)$ back to y in the above case). This means the calculation of the points-to relations and the call relation is mutual dependent. To sum up, we need to find the smallest solution that satisfies all constraints for call- and points-to relations simultaneously.

For method call, we have the following seemingly complex rule.

$$\frac{\begin{array}{c} y = x.m(a_1, a_2, \dots) \\ \hline \forall o \in \mathsf{Pts}(x), \mathsf{def}(o.m) ::= m(x_1, x_2, \dots) \Rightarrow e : \\ [\![\mathsf{Pts}(a_i)]\!] \subseteq \mathsf{Pts}(x_i), [\![\mathsf{Pts}(e)]\!] \subseteq \mathsf{Pts}(y) \end{array}}{} \tag{5}$$

All Java method definitions have a finite list of parameters (unlike vararg in C-like languages). The rule depends on the points-to set for all objects in the abstract

heap that may be referenced by x. Given such an object o, such that $o.m(\dots)$ is the method referred to in the statement, $\mathsf{def}(o.m)$ refers to the parameter list $x_1, x_2 \dots$ and return expression e as defined in the method. If a parameter or return variable is of reference type, we extend the subset construction to let the points-to set of the actual parameter be subset of the points-to set of the corresponding formal parameter, and let the points-to set of the return expression be subset of the points-to of the accepting variable. As an actual parameter may be an expression (such as in the form of $a.f$ and $func(b, c)$), we use the function $[\![.]\!]$ to extract the related points-to sets in the form of Pts or HPts. Similar way is applied on $[\![\mathsf{Pts}(z)]\!]$ for the return expression.[1] A special case may be that an abstract parameter a_i is a new allocation $\mathsf{new}^{o'}C(...)$, in which case we revise the constrain into $o' \in \mathsf{Pts}(x_i)$.

In the other paper we does not compute the points-to set of x in the statement $y = x.m(a_1, a_2, \dots)$. Instead a coarser abstraction is applied such that we take methods $m(\dots)$ in all subclasses of the class that defines x. For example, if x is of class C that defines m, which is overridden in D (D subclass of C), and suppose C has another subclass E; then all three $m(\dots)$ definitions are taken into the above calculation. Such a methodology is called Class Hierarchy Analysis (CHA), which is more efficient in program analysis of large-scaled code in millions of lines of code.

References

1. A. Møller. https://users-cs.au.dk/amoeller/RegAut/JavaBNF.html
2. L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.

[1] We may be able to recursively define the operator $[\![.]\!]$ later