Variable Flow Analysis for Callsite Abstraction

June 5, 2019

Jinan University

The Overall Framework

Our current focus is to define a new heap abstraction algorithm that is similar to the MAHJONG paper, but with a slightly different focus, which we clarify in the formalism. The work flow is as follows.

1. We define a core calculus that models the basic features of a typical Object-Oriented programming language, with an evaluation semantics.
2. We formalize points-to analysis and heap abstraction in the above language, and propose the soundness of the points-to analysis (later to prove).
3. A type system for points-to and a new recursive type, and prove equivalence regarding call site devirtualization. In general, we shall have object points-to (OPT) more precise than recursive type flow analysis (RTFA), followed by variable type analysis (VTA), followed by class hierarchy analysis (CHA).
4. Type checking algorithm
5. In theory, extend the result to context sensitive systems (call sensitive, object sensitive and type sensitive).
6. Experiment (only work out the context-insensitive inter-procedural RTA)

$$
\begin{array}{lll}
C & ::= & \mathsf{class}\ c\ [\mathsf{extends}\ c]\ \{\overline{F};\ \overline{M}\} \\
F & ::= & c\ f \\
D & ::= & c\ z \\
M & ::= & m(x)\ \{\overline{D}; s\} \\
s & ::= & e \mid x{=}\mathsf{new}^{o}\ c \mid x{=}e \mid x.f{=}y \\
 & & \mid \mathsf{if}\ x\ \mathsf{then}\ s\ \mathsf{else}\ s \mid s; s \\
e & ::= & \mathsf{null} \mid x \mid x.f \mid x.m(y) \\
prog & ::= & \overline{C}; \overline{D}; s
\end{array}
$$

Fig. 1. Abstract syntax for the core language.

We define a core calculus in Figure 1 consisting of all the key Object-Oriented language features. A program is defined as a code base $\overline{C}$ (i.e., a collection of class definitions) with s to be evaluated. To run a program, one may assume that s is the default (static) entry method with local variable declarations $\overline{D}$, similar to e.g., Java and C++, which may differ in specific language design. Let S and H be the runtime stack and heap, where $S : \mathtt{VAR} \to \mathtt{V}$ maps local variables to values and $H : \mathtt{V} \to \mathtt{OBJ} \cup \{\mathsf{null}\}$ maps values to objects. Two auxiliary

functions are also given. Function $fields$ maps class names to its fields, and $class$ provides types (or class names) for objects. Given class c, if $f \in fields(c)$, then $type(c, f)$ is the defined class type of field f in c. Similarly, give an object o, if $f \in fields(class(o))$, then $o.f$ may refer to an object of type $type(class(o), f)$, or an object of any of its subclass at runtime. We define the following evaluation semantics in Figure 2.

$$\text{[NULL]}\ \frac{}{S\ H\ \mathsf{null}\ \Downarrow\ S\ H\ \mathsf{null}} \qquad \text{[VAR]}\ \frac{S(x) = v}{S\ H\ x\ \Downarrow\ S\ H\ v} \qquad \text{[LOAD]}\ \frac{S(x) = v_1 \qquad H(v_1)(f) = v_2}{S\ H\ x.f\ \Downarrow\ S\ H\ v_2}$$

$$\text{[CALL]}\ \frac{S\ H\ x\ \Downarrow\ v_1 \qquad S\ H\ y\ \Downarrow\ v_2 \qquad class(H(v_1)).m = m(z)\{\overline{D}; s\} \qquad \{\mathsf{this} \mapsto v_1, z \mapsto v_2, z' \mapsto \mathsf{null} \mid z' \in \overline{D}\}\ H\ s\ \Downarrow\ S_1\ H_1\ v_3}{S\ H\ x.m(y)\ \Downarrow\ S\ H_1\ v_3}$$

$$\text{[ASSIGN]}\ \frac{S\ H\ e\ \Downarrow\ S'\ H'\ v}{S\ H\ x{=}e\ \Downarrow\ S[x \mapsto v]\ H\ v} \qquad \text{[STORE]}\ \frac{S(x) = v_1 \qquad S\ H\ y\ \Downarrow\ S\ H\ v_2}{S\ H\ x.f{=}y\ \Downarrow\ S\ H[(v_1, f) \mapsto v_2]\ v_2}$$

$$\text{[NEW]}\ \frac{fields(c) = \overline{f} \qquad v \notin dom(H)}{S\ H\ (x{=}\mathsf{new}\ c)\ \Downarrow\ S[x \mapsto v]\ H \cup \{v \mapsto (\overline{f \mapsto \mathsf{null}})\}} \qquad \text{[SEQ]}\ \frac{E\ s_1\ \Downarrow\ E_1\ v_1 \qquad E_1\ s_2\ \Downarrow\ E_2\ v_2}{E\ s_1; s_2\ \Downarrow\ E_2\ v_2}$$

$$\text{[TRUE]}\ \frac{S(x) \neq \mathsf{null} \qquad E\ s_1\ \Downarrow\ E_1\ v_1}{E\ (\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)\ \Downarrow\ E_1\ v_1} \qquad \text{[FALSE]}\ \frac{S(x) = \mathsf{null} \qquad E\ s_2\ \Downarrow\ E_2\ v_2}{E\ (\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)\ \Downarrow\ E_2\ v_2}$$

Fig. 2. Big-step operational semantics.

The operational semantics has a common form of $S\ H\ s\ \Downarrow\ S'\ H'\ v$, which represents the execution of s updates S and H into S' and H', leaving v as the resulting value. Sometimes we use E to represent the stack and heap pair. In particular, all values of interest are locations, and we only apply the calculus to illustrate the type system for call graph construction in the basic object-oriented setting. Note that we treat fields and methods of an object differently. A method is like a static member of a class, which is denoted as $c.m$ where c is a class name. A field defined in a class may refer to different objects of the same type at runtime.

The variable type analysis (VTA) provides a set of constraints which can be formalized in the follow way. We introduce the system as a context insensitive style in which a variable a in `VAR` is uniquely referred to by where it is defined in the program code, such as $c.m.a$ for method m of class c. An object is determined by its creation site, such as $c.m.\ell$ denotes that it is syntactically the ℓ's new statement in method $c.m$. To formalize VTA, we define $\Omega_c : \texttt{VAR} \to \mathcal{P}(\mathcal{C})$ and $\Phi_c : \mathcal{C} \times \mathcal{F} \to \mathcal{P}(\mathcal{C})$ as the smallest solution for the syntax directed constraints as shown in Figure 3.

statement	VTA	Points-to
$x = \mathsf{new}\ c$	$c \in \Omega_c(x)$	$o_i \in \Omega_o(x)$
$x = y$	$\Omega_c(y) \subseteq \Omega_c(x)$	$\Omega_o(y) \subseteq \Omega_o(x)$
$x = y.f$	$\forall c \in \Omega_c(y) : \Phi_c(c, f) \subseteq \Omega_c(x)$	$\forall o \in \Omega_o(y) : \Phi_o(o, f) \subseteq \Omega_o(x)$
$x.f = y$	$\forall c \in \Omega_c(x) : \Omega_c(y) \subseteq \Phi_c(c, f)$	$\forall o \in \Omega_o(x) : \Omega_o(y) \subseteq \Phi_o(o, f)$
$x = y.m(z)$	$\forall c \in \Omega_c(y) : \begin{cases} \Omega_c(z) \subseteq \Omega_c(param(c, m)) \\ \Omega_c(this(c, m)) = \{c\} \\ \forall x' \in return(c, m) : \\ \quad \Omega_c(x') \subseteq \Omega_c(x) \end{cases}$	$\forall o \in \Omega_o(y) : \begin{cases} \Omega_o(z) \subseteq \Omega_o(param(type(o), m)) \\ \Omega_0(this(type(o), m)) = \{o\} \\ \forall x' \in return(type(o), m) : \\ \quad \Omega_o(x') \subseteq \Omega_o(x) \end{cases}$

Fig. 3. Constraints for variable type analysis (VTA) and points-to analysis.

The context insensitive points-to analysis can be formalized in a similar way, as shown on the third column of Figure 3, where we define $\Omega_o : \texttt{VAR} \to \mathcal{P}(\texttt{OBJ})$ and $\Phi_o : \texttt{OBJ} \times \mathcal{F} \to \mathcal{P}(\texttt{OBJ})$.

Lemma 1 *Need to formalize (1) points-to derives an abstraction for dynamic calls (2) VTA is an abstract form of points-to.*

The points-to analysis provides a connection between local variables via the heap structure. Comparing to VTA, points-to has better precision with the price of extra space consumption in $\mathcal{O}(|C_{\mathsf{new}}|^2 \times |\mathcal{F}|)$. (Need to figure out more about the difference on space usage later.)

The imprecision of VTA in comparison with Points-to (the motivation example) lies in the domain of Ω_c that maintains an abstract heap of type $\mathcal{C} \times \mathcal{F} \to \mathcal{P}(\mathcal{C})$, so that an update of any object of class c affects all objects of class c, which brings imprecision.

We enrich the variable type analysis with the new type flow analysis with three relations, a partial order on variables $\sqsubseteq \subseteq \texttt{VAR} \times \texttt{VAR}$, a type flow relation $\dashrightarrow \subseteq \mathcal{C} \times \texttt{VAR}$, as well as a value access relation $\to \subseteq \texttt{VAR} \times \mathcal{F} \times \texttt{VAR}$, which are initially given as follows.

- $x = \mathsf{new}\ c$: $c \dashrightarrow x$;
- $x = y$: $y \sqsubseteq x$;
- $x.f = y$: $x \xrightarrow{f} y$;
- $x = y.f$: for all $y \xrightarrow{f} z : z \sqsubseteq x$.

In order to extend the three relations inter-procedurally, we firstly complete the transitivity of type flows.

- $c \dashrightarrow^* y$ if $c \dashrightarrow y \vee (c \dashrightarrow x \wedge x \sqsubseteq y)$;
- $y \sqsubseteq^* x$ if $y \sqsubseteq x \vee (\exists z \in \texttt{VAR} : y \sqsubseteq z \wedge z \sqsubseteq x)$;
- $x \xrightarrow{f} z$ if $x \xrightarrow{f} z \wedge x \sqsubseteq^* y$.

The type information is then used to resolve each method call $x = y.m(z)$.

$$\forall c \dashrightarrow^* y : \begin{cases} z \sqsubseteq^* param(c, m) \\ c \dashrightarrow^* this(c, m) \\ \forall x' \in return(c, m) : x' \sqsubseteq^* x \end{cases}$$

All the above constraints then derive the smallest relations $\dashrightarrow^*$, $\to$ and $\sqsubseteq$.

Lemma 2 *In a context-insensitive analysis, $c \dashrightarrow^* x$ iff there exists o of class c such that $o \in \Omega_o(x)$.*

Proof. We simulate the points-to computation by defining a series of intermediate functions Ω_n and Φ_n for $n \in \mathbb{N}$. Initially, for all $x \in \texttt{VAR}$, $\Omega_o(x) = \{o_i \mid x = \mathsf{new}c \text{ is at line } i\}$.

This basically means type flow analysis has the same precision regarding callsite resolution and cast failure check.

Next we need to show (1) regardless of the above, there is structural difference between type flow analysis and points-to analysis; (2) the results of Lemma 2 extends to context sensitive analysis. However, TFA only needs to build contexts for variables without a need to introduce contexts for objects, which may yield better scalability ?

Minimization on Type Flow Graphs