

Variable Flow Analysis for Callsite Abstraction

June 10, 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)

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

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 $S : \text{VAR} \rightarrow \mathbb{V}$ maps local variables to values and $H : \mathbb{V} \rightarrow \text{OBJ} \cup \{\text{null}\}$ 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 $f \in \text{fields}(c)$, then $\text{type}(c, f)$ is the defined class type of field *f* in *c*. Similarly, give an object *o*, if $f \in \text{fields}(\text{class}(o))$, then $o.f$ may refer to an object of type $\text{type}(\text{class}(o), f)$, or an object of any of its subclass at runtime. We define the following evaluation semantics in Figure 2.

$$\begin{array}{c}
\begin{array}{c} \text{[NULL]} \\ \hline S \ H \ \text{null} \Downarrow S \ H \ \text{null} \end{array} \quad \begin{array}{c} \text{[VAR]} \\ \hline S(x) = v \\ S \ H \ x \Downarrow S \ H \ v \end{array} \quad \begin{array}{c} \text{[LOAD]} \\ \hline S(x) = v_1 \quad H(v_1)(f) = v_2 \\ S \ H \ x.f \Downarrow S \ H \ v_2 \end{array} \\
\begin{array}{c} \text{[CALL]} \\ \hline S \ H \ x \Downarrow v_1 \quad S \ H \ y \Downarrow v_2 \quad \text{class}(H(v_1)).m = m(z)\{\overline{D}; s\} \\ \{\text{this} \mapsto v_1, z \mapsto v_2, z' \mapsto \text{null} \mid z' \in \overline{D}\} \ H \ s \Downarrow S_1 \ H_1 \ v_3 \\ \hline S \ H \ x.m(y) \Downarrow S \ H_1 \ v_3 \end{array} \\
\begin{array}{c} \text{[ASSIGN]} \quad \text{[STORE]} \\ \hline S \ H \ e \Downarrow S' \ H' \ v \quad S(x) = v_1 \quad S \ H \ y \Downarrow S \ H \ v_2 \\ S \ H \ x=e \Downarrow S[x \mapsto v] \ H \ v \quad S \ H \ x.f=y \Downarrow S \ H[(v_1, f) \mapsto v_2] \ v_2 \end{array} \\
\begin{array}{c} \text{[NEW]} \quad \text{[SEQ]} \\ \hline \text{fields}(c) = \overline{f} \quad v \notin \text{dom}(H) \quad E \ s_1 \Downarrow E_1 \ v_1 \quad E_1 \ s_2 \Downarrow E_2 \ v_2 \\ S \ H \ (x=\text{new } c) \Downarrow S[x \mapsto v] \ H \cup \{v \mapsto (f \mapsto \text{null})\} \quad E \ s_1; s_2 \Downarrow E_2 \ v_2 \end{array} \\
\begin{array}{c} \text{[TRUE]} \quad \text{[FALSE]} \\ \hline S(x) \neq \text{null} \quad E \ s_1 \Downarrow E_1 \ v_1 \quad S(x) = \text{null} \quad E \ s_2 \Downarrow E_2 \ v_2 \\ E \ (\text{if } x \text{ then } s_1 \text{ else } s_2) \Downarrow E_1 \ v_1 \quad E \ (\text{if } x \text{ then } s_1 \text{ else } s_2) \Downarrow E_2 \ v_2 \end{array}
\end{array}$$

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 $\Omega_c : \text{VAR} \rightarrow \mathcal{P}(\mathcal{C})$ and $\Phi_c : \mathcal{C} \times \mathcal{F} \rightarrow \mathcal{P}(\mathcal{C})$ as the smallest solution for the syntax directed constraints as shown in Figure 3.

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

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 $\Omega_o : \text{VAR} \rightarrow \mathcal{P}(\text{OBJ})$ and $\Phi_o : \text{OBJ} \times \mathcal{F} \rightarrow \mathcal{P}(\text{OBJ})$.

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 $\mathcal{O}(|C_{\text{new}}|^2 \times |\mathcal{F}|)$. (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 $\mathcal{C} \times \mathcal{F} \rightarrow \mathcal{P}(\mathcal{C})$, 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 $\sqsubseteq \subseteq \text{VAR} \times \text{VAR}$, a type flow relation $\dashv\vdash \subseteq \mathcal{C} \times \text{VAR}$, as well as a value access relation $\rightarrow \subseteq \text{VAR} \times \mathcal{F} \times \text{VAR}$, which are initially given as follows.

Definition 1. *We have the following base conditions.*

1. $x = \text{new } c : c \dashv\vdash x$;
2. $x = y : y \sqsubseteq x$;
3. $x.f = y : x \xrightarrow{f} y$;
4. $x = y.f : \text{for all } y \xrightarrow{f} z : z \sqsubseteq x$.

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

Definition 2. *We extend the three relations in the following way.*

1. $c \dashv\vdash^* y$ if $c \dashv\vdash y \vee (c \dashv\vdash^* x \wedge x \sqsubseteq^* y)$;
2. $y \sqsubseteq^* x$ if $y \sqsubseteq x \vee (\exists z \in \text{VAR} : y \sqsubseteq^* z \wedge z \sqsubseteq^* x)$;
3. $y \xrightarrow{f} z$ if $x \xrightarrow{f} z \wedge x \sqsubseteq^* y$;

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

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

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

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

Proof. (sketch) For inter-procedural analysis we only check the pass of type from actual parameters to formal parameters, as the other information passage can be treated similarly.

For the if (\Leftarrow) direction, we simulate the points-to computation by defining a series of intermediate results Ω_n and Φ_n for $n \in \mathbb{N}$. Initially, for all $x \in \text{VAR}$, $\Omega_o^0(x) = \{o_i \mid x = \text{new } c \text{ is at line } i\}$, and $\Phi_o^0(o, f) = \emptyset$ for all $o \in \text{OBJ}$ and $f \in \mathcal{F}$, where OBJ is the set of object abstractions that are defined in the form of $x = \text{new } c$, all are mutually distinguishable by their positions (i.e., line numbers) in the program. Obviously, we have $\text{type}(o) = c \dashrightarrow^* x$ for all such statements.

For the induction step, given Ω_o^n and Φ_o^n defined, for all $v \in \text{VAR}$, $o \in \text{OBJ}$ and $f \in \mathcal{F}$, we have

$$\begin{aligned} \Omega_o^{n+1}(x) &= \Omega_o^n(x) \cup \bigcup_{x=y} \Omega_o^n(y) \\ &\quad \cup \bigcup_{x=y.f} \bigcup_{o \in \Omega_o^n(f)} \Phi_o^n(o, f) \\ &\quad \cup \bigcup_{x'=y.m(z)} \{\Omega_o^n(z) \mid o \in \Omega_o^n(y) \wedge \text{method}(\text{type}(o), m) = m(x)\{\dots\}\}. \\ \Phi_o^{n+1}(o, f) &= \Phi_o^n(o, f) \cup \bigcup_{x.f=y} \{o'' \in \Omega_o^n(y) \mid o' \in \Omega_o^n(x)\}. \end{aligned}$$

Suppose for all $x \in \text{VAR}$ and $o \in \Omega_o^n(x)$, we have $\text{type}(o) \dashrightarrow^* x$, we prove the case for $n+1$. We also need an intermediate proof obligation as follows. For all $o' \in \Phi_o^n(o, f)$ and $o \in \Omega_o^n(x)$, there exists $y \in \text{VAR}$ such that $x \xrightarrow{f} y$ and $o' \in \Omega_o^n(y)$, which vacuously holds at the basis.

Then we have the following cases for $o \in \Omega_o^{n+1}(x) \setminus \Omega_o^n(x)$.

- Suppose $o \in \Omega_o^n(y)$ where $x = y$ is a statement. Then $y \sqsubseteq x$ by Def. 1(2). Since $o \dashrightarrow^* y$ by I.H., we have $o \dashrightarrow^* x$ by Def. 2(1).
- Suppose $o \in \Phi_o^n(o', f)$ and $o' \in \Omega_o^n(y)$ where $x = y.f$ is a statement. Then by I.H., there exists $z \in \text{VAR}$ with $y \xrightarrow{f} z$ and $o \in \Omega_o^n(z)$. Therefore $\text{type}(o) \dashrightarrow^* z$ by I.H.. Then $\text{type}(o) \dashrightarrow^* x$ by Def. 1(4).
- Suppose $o \in \Omega_o^n(z)$, $o' \in \Omega_o^n(y)$ and $\text{method}(\text{type}(o'), m) = m(x)\{\dots\}$, where $x' = y.m(z)$ is a statement. First we have $\text{type}(o) \dashrightarrow^* z$. Then by Def. 2(4), we have $z \sqsubseteq^* x$, then $\text{type}(o) \dashrightarrow^* x$ by Def. 2(1).

We also need to shown for all $o' \in \Phi_o^{n+1}(o, f)$ and $o \in \Omega_o^{n+1}(x) \setminus \Omega_o^n(x)$, there exists $y \in \text{VAR}$ such that $x \xrightarrow{f} y$ and $o' \in \Omega_o^{n+1}(y)$. The only possibility is that $o \in \Omega_o^n(x)$ and $o' \in \Omega_o^n(y)$, where $x.f = y$ is a statement. By definition, we have $o \in \Omega_o^{n+1}(x)$ and $o' \in \Omega_o^{n+1}(y)$. Moreover, since $x.f = y$ is a statement, by Def. 1(3), we have $x \xrightarrow{f} y$, as required.

For the only if (\Rightarrow) direction, we let each statement $x = \text{new } c$ at line i returns a type c_i which serves as a bridge between the type flow analysis and points-to analysis. To simulate the computation of the three relations $\sqsubseteq^* \subseteq \text{VAR} \times \text{VAR}$, $\dashv\vdash^* \subseteq \mathcal{C} \times \text{VAR}$ and $\rightarrow \subseteq \text{VAR} \times \mathcal{F} \times \text{VAR}$, we have initially $c_i \dashv\vdash^* x$ for each statement $x = \text{new } c$ at line i , then we apply the rules at Def. 1(2-4) and Def. 2(1-4) whenever possible until the computation reaches least fixpoints for the three relations, i.e. when no rule can be applied, and prove the following constraints are satisfied for the three fixpoint relations \sqsubseteq^* , $\dashv\vdash^*$ and \rightarrow .

- $c_i \dashv\vdash^* x$ implies $c_i \in \Omega_o(x)$;
- $y \sqsubseteq^* x$ implies $\Omega_o(y) \subseteq \Omega_o(x)$;
- $x \xrightarrow{f} y$ implies for all $c_i \in \Omega_o(x)$ and $c_j \in \Omega_o(y)$, $c_j \in \Phi_o(c_i, f)$.

We check all rules in Def. 1(1-4) and Def. 2(1-4) as follows.

- For the cases in Def. 1(1-2) are straightforward.
- For the case in Def. 1(3) with a statement $x.f = y$ and $x \xrightarrow{f} y$, by definition of points-to, we have $\Omega_o(y) \subseteq \bigcap_{c_i \in \Omega_o(x)} \Phi_o(c_i, f)$. Therefore, given $c_j \in \Omega_o(y)$ and $c_i \in \Omega_o(x)$, we must have $c_j \in \Phi_o(c_i, f)$.
- TBD

□

This lemma basically says type flow analysis has the same precision regarding type based check, such as callsite resolution and cast failure check, when comparing with points-to analysis.

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 yield better scalability ?

TBD

- ★ Need to discuss the difference by examples
- ★ Need experiments to test efficiency by comparing the two approaches on real world examples.

Minimization on Type Flow Graphs

What are the benefits