CS 747: Formal Verification of Overapproximating Pointer Analysis Algorithms

Kevin Lee

Motivation

- Pointer analysis is an essential precursory step to many other forms of static analysis.
  - Knowledge of points-to determines whether two variables may be treated as the same (alias analysis).
- E.g. SeaDSA: used by SeaHorn and SMACK

Formalization of points-to relation

Concrete State Points-to

$\mathsf{SPt}(\Sigma : \mathsf{State}, v : \mathsf{Var}, h : \mathsf{AllocSite}) :=$

$$\exists a : \mathsf{Addr}, \Sigma_{[\sigma]}(v) = a \to \Sigma_{[H]}(a)_{[\mathsf{AllocSite}]} = h$$

Concrete Points-to

$\mathsf{Pt}(p : \mathsf{Control}, v : \mathsf{Var}, h : \mathsf{AllocSite}) :=$

$$\exists \Sigma \in \Sigma^*(p), \mathsf{SPt}(\Sigma, v, h)$$

Formalization of a sound points-to analysis

We restrict work to only *overapproximating* pointer analyses.

Given a language $\mathcal{L}$, an **abstract pointer analysis** $\mathsf{Pt}^\sharp_\mathcal{L}$ is a *sound approximation* of a **concrete pointer analysis** $\mathsf{Pt}_\mathcal{L}$, i.e.

$$\mathsf{Pt}_\mathcal{L} \preceq \mathsf{Pt}^\sharp_\mathcal{L}$$

iff $\forall P \in \mathcal{L}, \forall v : \mathsf{Var}, \forall h : \mathsf{AllocSite},$

$$\mathsf{Pt}_\mathcal{L}(P, v, h) \rightarrow \mathsf{Pt}^\sharp_\mathcal{L}(P, v, h)$$

Goal: Start simple, then build up

$\mathcal{L}_{\mathsf{Alloc}}$

- Consists of `skip`, $P_1; P_2$, `alloc` a, $a \leftarrow b$.
- Builds the foundational proof framework *that is generalizable to* more complex languages.

$\mathcal{L}_{\mathsf{While}}$

- Simple Imperative Language with `alloc` a, `read` a, `write` a.
- More desirable proof result. May get to if there is enough time.

All will be compared against Anderseen-style pointer analysis.

Basic types of $\mathcal{L}_{\mathsf{Alloc}}$

Variable	$\mathsf{Var} := \mathsf{nat}$
Memory address	$\mathsf{Addr} := \mathsf{nat}$
Values	$\mathsf{Val} := \mathsf{Addr}$
Allocation site label	$\mathsf{AllocSite} := \mathsf{nat}$
Heap object	$\mathsf{Option\ Val} \times \mathsf{AllocSite}$
Heap	$H := \mathsf{Addr} \rightharpoonup \mathsf{HeapObj}$
Valuation	$\sigma := \mathsf{Var} \rightharpoonup \mathsf{Val}$
State	$\mathsf{State} := H \times \sigma$

Syntax of $\mathcal{L}_{\mathsf{Alloc}}$

```
Inductive Control : Ensemble AllocSite -> Type :=
| Skip : Control (Empty_set _)
| Assign : Var -> Var -> Control (Empty_set _)
| Alloc (l : AllocSite) (vto : Var) : Control (Singleton _ l)
| Seq : forall s1 s2,
  Control s1 -> Control s2 -> Disjoint _ s1 s2  -> Control
(Union _ s1 s2).
```

Note that the syntax is dependent on the set of AllocSite which the program contains.

- Every `Alloc` must be labelled a unique AllocSite!

Semantics of $\mathcal{L}_{\mathsf{Alloc}}$

$$\frac{\langle P_1, s\rangle \Rightarrow \langle P_1', s\rangle}{\langle P_1; P_2, s\rangle \Rightarrow \langle P_1'; P_2, s'\rangle} \Rightarrow_{\text{SeqR}}$$

$$\frac{}{\langle \texttt{skip}; P_1, s\rangle \Rightarrow \langle P_1, s\rangle} \Rightarrow_{\text{SeqL}}$$

$$\frac{s'_{[\sigma]} = s_{[\sigma]}\left[v_{\text{to}} \mapsto s_{[\sigma]}(v_{\text{from}})\right]}{\langle v_{\text{to}} \leftarrow v_{\text{from}}, s\rangle \Rightarrow \langle \texttt{skip}, s'\rangle} \Rightarrow_{\text{Assign}}$$

Semantics of $\mathcal{L}_{\mathsf{Alloc}}$

$$\dfrac{\mathsf{Unallocated}(h, \mathrm{addr}) \qquad s'_{[h]} = s_{[h]}[\mathrm{addr} \mapsto \mathrm{site}] \quad s'_{[\sigma]} = s_{[\sigma]}[v \mapsto \mathrm{addr}]}{\langle \mathtt{alloc}_{\mathrm{site}}\, v, s \rangle \Rightarrow \langle \mathtt{skip}, s' \rangle} \Rightarrow_{\mathrm{Alloc}}$$

Anderseen-style pointer-analysis of $\mathcal{L}_{\mathsf{Alloc}}$

The original set of Datalog rules are filtered down to just two simple inference rules.

$$\frac{\mathsf{HasAlloc}(P, v, \mathrm{site})}{\mathsf{Ander}(P, v, \mathrm{site})}\mathsf{Ander}_{\mathrm{Alloc}}$$

$$\frac{\mathsf{HasMove}(P, v_{\mathrm{to}}, v_{\mathrm{from}}) \quad \mathsf{Ander}(P, v_{\mathrm{from}}, \mathrm{site})}{\mathsf{Ander}(P, v_{\mathrm{to}}, \mathrm{site})}\mathsf{Ander}_{\mathrm{Move}}$$

Stepping stones to connect the two forms of analysis

$$\mathsf{Pt} \longrightarrow ??? \longrightarrow \mathsf{CarryAlloc} \longrightarrow \mathsf{Ander}$$

What is proven so far:

$$\mathsf{CarryAlloc}(P, v_{\text{to}}, \text{site}) \to \mathsf{Ander}(P, v_{\text{to}}, \text{site})$$

Where

- $\mathsf{CarryAlloc}(P, v_{\text{to}}, \text{site}) =$

$$\exists v_{\text{from}}, \mathsf{HasMove}^*(P, v_{\text{to}}, v_{\text{from}}) \land \mathsf{Alloc}(P, v_{\text{from}}, \text{site})$$

- $\mathsf{HasMove}^*$ is the transitive closure of $\mathsf{HasMove}$.

Conjecture: FPt is a stepping stone

1. Perform state erasure on the small-step semantics
2. Extract points-to precondition/postcondition for each $\mathsf{Control}$

Successor: $\succ$

- Small-step semantics with state erased

$$\frac{P_1 \prec P_1'}{P_1; P_2 \prec P_1'; P_2} \succ_{\text{SeqR}} \qquad \frac{}{\texttt{skip}; P_1 \prec P_1} \succ_{\text{SeqL}}$$

$$\frac{}{v_{\text{to}} \leftarrow v_{\text{from}} \prec \texttt{skip}} \succ_{\text{Assign}} \qquad \frac{}{\texttt{alloc}_{\text{site}}\ v \prec \texttt{skip}} \succ_{\text{Alloc}}$$

Conjecture: FPt is a stepping stone

Head: $\mathsf{head}(p : \mathsf{Control})$

- The set of possible next Control to be run by the small-step on p, for any possible state.

Head is used to anticipate the next action to be taken by the program by FPt.

Head-successor is a more semantically consistent way to destruct a program than destructing by the program's syntactical sequence.

Conjecture: FPt is a stepping stone

We needed State to reason about points-to relation. When it's erased, we need another method to reason about it.

Points-to graph: $\mathsf{PtG} := \mathsf{Var} \times \mathsf{AllocSite} \to \mathsf{Prop}$

- Abstract State to only information about points-to relations within the state.

Conjecture: FPt is a stepping stone

Flow points-to: $\mathsf{FPt}\ (g_1 : \mathsf{PtG}, P_1 : \mathsf{Control}, P_2 : \mathsf{Control}, g_2 : \mathsf{PtG})$

Note: this is not yet proven to be a sound approximation of Pt. *There may be formulation mistakes.*

$$\frac{}{\mathsf{FPt}(g, P, P, g)}\,\mathsf{FPt}_{\mathrm{Reflect}}$$

$$\frac{\mathsf{FPt}(g_1, P_1, P_2, g_2) \quad \mathsf{FPt}(g_2, P_2, P_3, g_3)}{\mathsf{FPt}(g_1, P_1, P_3, g_3)}\,\mathsf{FPt}_{\mathrm{Trans}}$$

$$\frac{P_1 \prec P_2 \quad \mathsf{head}(P_1) = \mathtt{skip}}{\mathsf{FPt}(g, P_1, P_2, g)}\,\mathsf{FPt}_{\text{Skip}}$$

$$\frac{\begin{array}{c} P_1 \prec P_2 \quad \mathsf{head}(P_1) = \mathtt{alloc}_{\text{site}}\, v \quad \langle v, \text{site}\rangle \in g_2 \\ (v \neq v_1 \wedge \langle v_1, \text{site}_1\rangle \in g_1) \to \langle v_1, \text{site}_1\rangle \in g_2 \end{array}}{\mathsf{FPt}(g_1, P_1, P_2, g_2)}\,\mathsf{FPt}_{\text{Alloc}}$$

$$\frac{\begin{array}{c} P_1 \prec P_2 \quad \mathsf{head}(P_1) = v_{\text{to}} \leftarrow v_{\text{from}} \quad \langle v_{\text{from}}, \text{site}\rangle \in g_1 \\ \langle v_{\text{to}}, \text{site}\rangle \in g_2 \quad (v_{\text{to}} \neq v_1 \wedge \langle v_1, \text{site}_1\rangle \in g_1) \to \langle v_1, \text{site}_1\rangle \in g_2 \end{array}}{\mathsf{FPt}(g_1, P_1, P_2, g_2)}\,\mathsf{FPt}_{\text{Move}}$$

Conjecture: FPt is a stepping stone

Why might this work?

- Generalizes well to more complex languages.
- Eliminates overhead reasoning about irrelevant parts of the state.
- Unlike Pt, defined inductively in Coq and easier to determine possible subprograms using `econstructor`.
- Head-successor is a more semantically consistent way to destruct a program than destructing by the program's syntactical sequence.

The work to bridge the gap is ongoing.