Flow-Insensitive Points-To Analysis with Term and Set Constraints

Pointer analysis in type theory's clothing!

Mihir Mehta (based on the paper by Foster, Fahndrich and Aiken)

Department of Computer Science
University of Texas, Austin
mihir@cs.utexas.edu

19 January, 2016

Set constraints 101 I

Definition

Set constraints [1] describe relationships between sets of *terms*. They take the form $X \subseteq Y$, where X and Y are set expressions, generated by this grammar.

$E ::= \alpha | 0 | E_1 \cup E_2 | E_1 \cap E_2 | \neg E_1 | c(E_1, \ldots, E_{a(c)}) | c^{-i}(E_1)$

Definition

A system of set constraints is a finite conjunction of constraints $\bigwedge_i X_i \subseteq Y_i$.

Definition

A solution to a system of set constraints is an assignment σ of sets to variables in the system such that all the constraints in the system are satisfied when σ is extended to set expressions under these rules.

Set constraints 101 II

$$\begin{aligned}
\sigma(0) &= \phi \\
\sigma(E_1 \cup E_2) &= \sigma(E_1) \cup \sigma(E_2) \\
\sigma(E_1 \cap E_2) &= \sigma(E_1) \cap \sigma(E_2) \\
\sigma(\neg E_1) &= H - \sigma(E_1) \\
\sigma(c(E_1, \ldots, E_n)) &= \{c(t_1, \ldots, t_n) | t_i \in \sigma(E_i)\} \\
\sigma(c^{-i}(E)) &= \{t_i | \exists c(t_1, \ldots, t_n) \in \sigma(E)\}
\end{aligned}$$

- H is the Herbrand universe, that is, the set of all terms.
- We use $X = Y$ as a notational convenience to denote the pair of constraints $X \subseteq Y$ and $Y \subseteq X$. In the Hindley-Milner type system, constraints occur only in this form, and are solved by unification.
- Set constraints, in general, are solved by repeated application of re-writing rules, as we shall see.

This paper I

- Andersen [2] and Steensgard [5] both presented algorithms for pointer analysis; this paper formulates both of these algorithms as sets of typing rules.
- Typing rules are set constraints and can thus be solved in the general framework [3] which the authors had previously developed.
- The resulting implementation (in ML) is compared to a previous, more application-specific implementation (in C) [4]; the former's running time is found to be within a small constant factor of the latter.
- More concretely, this paper develops typing rules common to both algorithms (Common), typing rules specific to each algorithm (And and Ste) and shows that the type relations/equivalences which arise from And+Common and Ste+Common are equivalent to the points-to relations in the original algorithms.

This paper II

- Further, the paper shows that the two algorithms are similar enough to be represented by a combined set of rules (Comb), in which the only difference is in the constructor signatures.

The analysis framework I

- Set of sorts S, including Term and Set.
- Each n-ary constructor c has a signature

$$c : \iota_1 \dots \iota_n \to S$$

  where each ι_i is s (covariant) or $\overline{s}$ (contravariant) for some $s \in S$.
- Sort Term: set of constructors Σ_{Term}, set of variables V_{Term}. Terms: variables from V_{Term} or constructed terms following some constructor's signature:

$$c : \underbrace{Term \dots Term}_{arity(c)} \to Term, c \in \Sigma_{Term}$$

The analysis framework II

- Sort Set: set of constuctors Σ_{Set}, operations $\cup, \cap, 0, 1$. Set expressions follow some constructor's signature:

$$c : \underbrace{Set \ldots Set}_{arity(c)} \rightarrow Set, c \in \Sigma_{Set}$$

$$\begin{array}{lcl} \cup & : & \text{Set Set} \rightarrow \text{Set} \\ \cap & : & \text{Set Set} \rightarrow \text{Set} \\ 0 & : & \text{Set} \\ 1 & : & \text{Set} \end{array}$$

- In general, a sort s has a constraint relation $\subseteq_s$ and resolution rules for $\subseteq_s$. Term has two, $=_t$ (Hindley-Milner style equality) and $=_c$ (Steensgard-style conditional equality.).

The analysis framework III

$$
\begin{array}{rcll}
S \wedge f(T_1, \ldots, T_n) =_{\mathrm{t}} f(T'_1, \ldots, T'_n) & \equiv & S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge T'_1 \subseteq_{\iota_1} T_1 \wedge \cdots \wedge & \text{if } f : \iota_1 \cdots \iota_n \to \texttt{Term} \\
 & & T_n \subseteq_{\iota_n} T'_n \wedge T'_n \subseteq_{\iota_n} T_n & \\
S \wedge f(\ldots) =_{\mathrm{t}} g(\ldots) & \equiv & \text{inconsistent} & \text{if } f \neq g \\
S \wedge f(\ldots) =_{\mathrm{t}} \alpha \wedge \alpha =_{\mathrm{c}} T & \equiv & \alpha =_{\mathrm{t}} T &
\end{array}
$$

(a) Resolution rules for sort `Term`.

$$
\begin{array}{rcll}
S \wedge 0 \subseteq_{\mathrm{s}} T & \equiv & S & \\
S \wedge T \subseteq_{\mathrm{s}} 1 & \equiv & S & \\
S \wedge c(T_1, \ldots, T_n) \subseteq_{\mathrm{s}} c(T'_1, \ldots, T'_n) & \equiv & S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge \cdots \wedge T_n \subseteq_{\iota_n} T'_n & \text{if } c : \iota_1 \cdots \iota_n \to \texttt{Set} \\
S \wedge c(\ldots) \subseteq_{\mathrm{s}} d(\ldots) & \equiv & \text{inconsistent} & \text{if } c \neq d \\
S \wedge T_1 \cup T_2 \subseteq_{\mathrm{s}} T & \equiv & S \wedge T_1 \subseteq_{\mathrm{s}} T \wedge T_2 \subseteq_{\mathrm{s}} T & \\
S \wedge T \subseteq_{\mathrm{s}} T_1 \cap T_2 & \equiv & S \wedge T \subseteq_{\mathrm{s}} T_1 \wedge T \subseteq_{\mathrm{s}} T_2 & \\
S \wedge \alpha \subseteq_{\mathrm{s}} \alpha & \equiv & S & \\
S \wedge \alpha \cap T \subseteq_{\mathrm{s}} \alpha & \equiv & S &
\end{array}
$$

(b) Resolution rules for sort `Set`.

$$
\begin{array}{rcl}
S \wedge X \subseteq_{\iota} \alpha \wedge \alpha \subseteq_{\iota} Y & \equiv & S \wedge X \subseteq_{\iota} \alpha \wedge \alpha \subseteq_{\iota} Y \wedge X \subseteq_{\iota} Y \\
S \wedge T_1 \subseteq_{\overline{\iota}} T_2 & \equiv & S \wedge T_2 \subseteq_{\iota} T_1
\end{array}
$$

(c) General rules.

Andersen-style analysis I

We analyse this fragment of C. Note that this excludes several control-flow constructs such as for loops, while loops and goto which are redundant under flow-insensitivity.

def	::=	$\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) \rightarrow \texttt{y} = e$	Function definition
	|	$def_1\, def_2$	Sequencing
e	::=	n	Constant integer n
	|	$\texttt{x}$	Variable
	|	$*e$	Pointer dereference
	|	$\&e$	Address of e
	|	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	Conditional
	|	$e_1 = e_2$	Assignment
	|	$\texttt{f}(\texttt{z}_1, \ldots, \texttt{z}_m)$	Function application
	|	*"string"*	String constant
	|	$\texttt{malloc}(e)$	Heap allocation
	|	$e_1 op\ e_2$	Scalar operation (*e.g.* `+`, `--`, `*`)
	|	e_1, e_2	Sequencing
	|	$(type) e$	Type cast
	|	$e.id$	Field access
	|	$e \rightarrow id$	Field indirection ($= (*e).id$)
	|	$e_1[e_2]$	Array access ($\approx *(e_1 + e_2)$)

Andersen-style analysis II

- In this system, types follow this grammar.

$$\begin{array}{rcl} \tau_A & ::= & \alpha \mid ref(p_{get} = \tau_A, p_{set} = \tau_A, f_{get} = \lambda_A, f_{set} = \lambda_A) \\ \lambda_A & ::= & \beta \mid lam_n(\tau_{A_0}, \tau_{A_1}, \ldots, \tau_{A_n}) \end{array}$$

- The constructors *ref* and lam_n have these signatures.

$$\begin{array}{rl} ref: & (p_{get} = \mathtt{Set}, p_{set} = \overline{\mathtt{Set}}, f_{get} = \mathtt{Set}, f_{set} = \overline{\mathtt{Set}}) \to \mathtt{Set} \\ lam_n: & \underbrace{\mathtt{Set} \cdots \mathtt{Set}}_{n+1} \to \mathtt{Set} \end{array}$$

- Having α renamed the program variables, we can be sure each is unique, and use the inference rules in And+Common.

Andersen-style analysis III

And

$$\frac{}{n : 0} \quad (\text{Const} - \text{Int}_\text{A})$$

$$\frac{}{\mathtt{x} : ref(p_{get} = p_{set} = x, f_{get} = f_{set} = x')} \quad (\text{Var}_\text{A})$$

$$\frac{e : \tau}{\&e : ref(p_{get} = p_{set} = \tau)} \quad (\text{Addr}_\text{A})$$

$$\frac{e : \tau \quad \tau \subseteq_s ref(p_{get} = \alpha)}{*e : \alpha} \quad (\text{Deref}_\text{A})$$

$$\frac{\begin{array}{cc} e_1 : \tau_1 & e_2 : \tau_2 \\ \tau_1 \subseteq_s ref(p_{set} = \alpha, f_{set} = \alpha') & \tau_2 \subseteq_s ref(p_{get} = \beta, f_{get} = \beta') \\ \beta \subseteq_s \alpha & \beta' \subseteq_s \alpha' \end{array}}{e_1 = e_2 : \tau_2} \quad (\text{Asst}_\text{A})$$

$$\frac{\begin{array}{c} \mathtt{f} : ref(f_{get} = f_{set} = f) \\ \mathtt{y} : \tau_y \quad e : \tau_e \quad \forall 1 \leq i \leq n \;.\; \mathtt{x}_i : \tau_i \\ lam_n(\tau_y, \tau_1, \ldots, \tau_n) \subseteq_s f \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \rightarrow \mathtt{y} = e : \mathtt{void}} \quad (\text{Defn}_\text{A})$$

$$\frac{\begin{array}{ccc} \mathtt{f} : \tau_f & \mathtt{x}_i : \tau_i & \forall 1 \leq i \leq n \\ \tau_f \subseteq_s ref(f_{get} = lam_n(\alpha_0, \alpha_1, \ldots, \alpha_n)) & & \\ \alpha_i \subseteq_s ref(p_{set} = \alpha_{i_f}, f_{set} = \alpha'_{i_f}) & \tau_i \subseteq_s ref(p_{get} = \alpha_{i_x}, f_{get} = \alpha'_{i_x}) & \\ \alpha_{i_x} \subseteq_s \alpha_{i_f} & \alpha'_{i_x} \subseteq_s \alpha'_{i_f} & \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) : \alpha_0} \quad (\text{App}_\text{A})$$

Andersen-style analysis IV

Common

$$\frac{\&V_l : \tau \qquad V_l \text{ fresh}}{\text{“}string_l\text{”} : \tau} \quad (\text{Const} - \text{Str})$$

$$\frac{\&V_l : \tau \quad e : \tau_e \qquad V_l \text{ fresh}}{malloc_l(e) : \tau} \quad (\text{Malloc})$$

$$\frac{\begin{array}{c} V_l : \tau \quad e_1 : \tau_1 \quad e_2 : \tau_2 \quad e_3 : \tau_3 \qquad V_l \text{ fresh} \\ V_l = e_2 : \tau_1' \quad V_l = e_3 : \tau_2' \end{array}}{\texttt{if}_l\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau} \quad (\text{Cond})$$

$$\frac{\begin{array}{c} V_l : \tau \quad e_1 : \tau_1 \quad e_2 : \tau_2 \qquad V_l \text{ fresh} \\ V_l = e_1 : \tau_1' \quad V_l = e_2 : \tau_2' \end{array}}{e_1\ op_l\ e_2 : \tau} \quad (\text{Op})$$

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{e_1, e_2 : \tau_2} \quad (\text{Seq})$$

$$\frac{def_1 : \texttt{void} \quad def_2 : \texttt{void}}{def_1\, def_2 : \texttt{void}} \quad (\text{Def} - \text{Seq})$$

$$\frac{e : \tau}{(type)\ e : \tau} \quad (\text{Cast})$$

$$\frac{e : \tau}{e.id : \tau} \quad (\text{Field} - \text{direct})$$

$$\frac{*e : \tau}{e \rightarrow id : \tau} \quad (\text{Field} - \text{indirect})$$

$$\frac{*(e_1 + e_2) : \tau}{e_1[e_2] : \tau} \quad (\text{Array})$$

Andersen-style analysis V

- Let $\vdash_A$ mean "provable in And+Common".
- Then, Andersen's points-to relation $P_A : L \to 2^L$ is given by $y \in P_A(x)$ if

$$CS \vdash_A ref(p_{get} = p_{set} = y, f_{get} = f_{set} = y') \subseteq_s x$$

Steensgard-style analysis I

- In this system, types follow this grammar.

$$\begin{array}{rcl} \tau_S & ::= & \alpha \mid ref(p = \tau_S, f = \lambda_S, t = \tau_S) \\ \lambda_S & ::= & \beta \mid lam_n(\tau_{S_0}, \tau_{S_1}, \ldots, \tau_{S_n}) \end{array}$$

- The constructors *ref* and lam_n have these signatures.

$$ref : (p = \texttt{Term}, f = \texttt{Term}, t = \texttt{Term}) \to \texttt{Term}$$
$$lam_n : \underbrace{\texttt{Term} \cdots \texttt{Term}}_{n+1} \to \texttt{Term}$$

- Steensgard's system uses unification where Andersen's had inclusion, thus:
  - the old p_{get}(covariant) and p_{set}(contravariant) merge into p(invariant).
  - the old f_{get}(covariant) and f_{set}(contravariant) merge into f(invariant).
  - the field t is added to store a tag; this refers to the equivalence class the reference belongs to.

Steensgard-style analysis II

- Having α renamed the program variables, we can be sure each is unique, and use the inference rules in Ste+Common.

Ste

$$\frac{}{n : \_} \quad (\text{Const} - \text{Int}_S)$$

$$\frac{}{\mathtt{x} : ref(p = x, f = x', t = x_t)} \quad (\text{Var}_S)$$

$$\frac{e : \tau}{\&e : ref(p =_c \tau)} \quad (\text{Addr}_S)$$

$$\frac{e : \tau \quad \tau =_c ref(p = \alpha)}{*e : \alpha} \quad (\text{Deref}_S)$$

$$\frac{\begin{array}{cc} e_1 : \tau_1 & e_2 : \tau_2 \\ \tau_1 =_c ref(p = \alpha, f = \alpha') & \tau_2 =_c ref(p = \beta, f = \beta') \\ \beta =_c \alpha & \beta' =_c \alpha' \end{array}}{e_1 = e_2 : \tau_2} \quad (\text{Asst}_S)$$

$$\frac{\begin{array}{c} \mathtt{f} : ref(p = f, f = f', t = f'') \\ \mathtt{y} : \tau_y \quad e : \tau_e \quad \forall 1 \le i \le n \;.\; \mathtt{x}_i : \tau_i \\ lam_n(\tau_y, \tau_1, \ldots, \tau_n) =_c f' \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \to \mathtt{y} = e : \mathtt{void}} \quad (\text{Defn}_S)$$

$$\frac{\begin{array}{ccc} \mathtt{f} : \tau_f & \mathtt{x}_i : \tau_i & \forall 1 \le i \le n \\ \tau_f =_c ref(f = lam_n(\alpha_0, \alpha_1, \ldots, \alpha_n)) & & \\ \alpha_i =_c ref(p = \alpha_{i_1}, f = \alpha_{i_2}) & \tau_i =_c ref(p = \alpha_{i'_1}, f = \alpha_{i'_2}) & \\ \alpha_{i'_1} =_c \alpha_{i_1} & \alpha_{i'_2} =_c \alpha_{i_2} & \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) : \tau_0} \quad (\text{App}_S)$$

Steensgard-style analysis III

- In this analysis, $x =_c y$ (conditional unification) iff

$$\exists c. \vdash c(\dots)_t x \Rightarrow x_t = y_t$$

  Intuitively, conditionally unified terms are unified only when one of them is unified with a constructed type.
- Let $\vdash_S$ mean "provable in Ste+Common".
- Then, Steensgard's points-to relation $P_S : L \to 2^L$ is given by $y \in P_S(x)$ if

$$CS \vdash_S ref(p = y, f = y', t = y_t) =_t x$$

- In this system, the equivalence class $[x]$ containing a memory location x is given by $[x] = y : CS \vdash_S x_t = y_t$

Exploiting the correspondence between Andersen's and Steengard's analyses I

- These two analyses are similar enough that we can build a common system, Comb that can implement either.
- This is the type grammar.

$$\begin{array}{lcl} \tau_{AS} & ::= & \alpha \mid ref(p_{get} = \tau_{AS}, p_{set} = \tau_{AS}, f_{get} = \lambda_{AS}, f_{set} = \lambda_{AS}, t = \tau_{AS}) \\ \lambda_{AS} & ::= & \beta \mid lam_n(\tau_{AS_0}, \tau_{AS_1}, \ldots, \tau_{AS_n}) \end{array}$$

- For the Andersen-style analysis, we use these constructor signatures. The t field is ignored.

$$ref : (p_{get} = \mathtt{Set}, p_{set} = \overline{\mathtt{Set}}, f_{get} = \mathtt{Set}, f_{set} = \overline{\mathtt{Set}}, t = \mathtt{Set}) \to \mathtt{Set}$$
$$lam_n : \underbrace{\mathtt{Set} \cdots \mathtt{Set}}_{n+1} \to \mathtt{Set}$$

- For the Steensgard-style analysis, we use these constructor signatures. $p_{get} = p_{set}$ and $f_{get} = f_{set}$ are required.

$$ref : (p_{get} = \mathtt{Term}, p_{set} = \mathtt{Term}, f_{get} = \mathtt{Term}, f_{set} = \mathtt{Term}, t = \mathtt{Term}) \to \mathtt{Term}$$
$$lam_n : \underbrace{\mathtt{Term} \cdots \mathtt{Term}}_{n+1} \to \mathtt{Term}$$

Exploiting the correspondence between Andersen's and Steengard's analyses II

Comb$_\iota$

$$\frac{}{\mathbf{x} : ref(p_{get} = p_{set} = x, f_{get} = f_{set} = x', t = x_t)} \quad (\mathrm{Var}_{\mathrm{Comb}_\iota})$$

$$\frac{e : \tau}{\&e : ref(p_{get} = p_{set} = \tau)} \quad (\mathrm{Addr}_{\mathrm{Comb}_\iota})$$

$$\frac{e : \tau \quad \tau \subseteq_\iota ref(p_{get} = \alpha)}{*e : \alpha} \quad (\mathrm{Deref}_{\mathrm{Comb}_\iota})$$

$$\frac{\begin{array}{c} e_1 : \tau_1 \\ \tau_1 \subseteq_\iota ref(p_{set} = \alpha', f_{set} = \alpha'') \\ \beta' \subseteq_\iota \alpha' \end{array} \quad \begin{array}{c} e_2 : \tau_2 \\ \tau_2 \subseteq_\iota ref(p_{get} = \beta', f_{get} = \beta'') \\ \beta'' \subseteq_\iota \alpha'' \end{array}}{e_1 = e_2 : \tau_2} \quad (\mathrm{Asst}_{\mathrm{Comb}_\iota})$$

- The rules for this system are similar to those in And; The rules for function definition and application remain the same.
- For Andersen-style analysis, use subset constraints ($\subseteq_\iota = \subseteq_s$) and constraint system CS_s.
- For Steensgard-style analysis, use conditional equality constraints ($\subseteq_\iota = \subseteq_c$) and constraint system CS_c .

Exploiting the correspondence between Andersen's and Steengard's analyses III

- The parameterised points-to relation $Q_\iota : L \Rightarrow 2^L$ is given by $y \in Q_\iota(x)$ if:

$$CS_\iota \vdash ref(p_{get} = p_{set} = y, f_{get} = f_{set} = y', t = y_t) \subseteq_\iota x$$

- This lemma shows the equivalence between the points-to sets generated by And+Common and $Comb_s + Common$, and also those generated by Ste+Common and $Comb_c + Common$.

(a) $\forall \mathbf{x}.Q_{\iota=s}(\mathbf{x}) = P_A(\mathbf{x})$, and
(b) $\forall \mathbf{x}.Q_{\iota=c}(\mathbf{x}) = P_S(\mathbf{x})$, and
(c) $\forall \mathbf{x}.[\mathbf{x}] = [\mathbf{x}]'$ where $[\cdot]$ is from **Ste+Common** and $[\cdot]'$ is from $Comb_{\iota=c}$.

- Next lemma: Andersen-style subset inclusion is a sufficient condition for Steensgard-style conditional unification. More formally, if the system CS_s is translated to the system CS_c (i. e. each $\tau_1 \subseteq_s \tau_2$ translated to $\tau_1 =_c \tau_2$), then

$$CS_s \vdash \tau_1' \subseteq_s \tau_2' \Rightarrow CS_c \vdash \tau_1' =_c \tau_2'$$

Exploiting the correspondence between Andersen's and Steengard's analyses IV

- Using these two lemmas, it's simple to show that

$$y \in P_A(x) \Rightarrow y \in P_s(x)$$

- Pretty cool, right?

Soundness I

- For a soundness result, we'll have to construct operational semantics for a fragment of C - best to keep it simple.

$$e ::= x | *e | \&e | e_1 = e_2 | e1, e2$$

- Let's construct a mapping θ from dynamic memory locations *Loc* to syntactic locations *SyntacticLoc*.

$$\begin{aligned} \theta \quad = \quad & \{[l_{\mathtt{x}} \mapsto \mathtt{x}] \mid \mathtt{x} \text{ is a program variable}\} \cup \\ & \{[l' \mapsto \&_i] \mid \&_i \text{ appears in the program, } l' \text{ is the corresponding location from (Addr)}\} \end{aligned}$$

- Program transformation: every time the address of a variable appears in an expression, add a new program variable $\&_i$ to store the result and substitute $\&_i$ in the result. The type inference rule is fairly obvious.

$$\frac{e : \tau}{\&_i e : ref(p_{set} = p_{get} = \tau) \quad \&_i : ref(p_{get} = p_{set} = \tau)} \qquad (\mathrm{Addr}'_{\mathrm{A}})$$

Soundness II

Domains: $Loc = \{l_{\mathtt{x}}, l_{\mathtt{y}}, \dots\}$
$SyntacticLoc = \{\mathtt{x}, \mathtt{y}, \dots \ \&_i, \dots\}$
$Expr = SyntacticLoc \mid *Expr \mid \&_i Expr \mid Expr_1 = Expr_2 \mid Expr_1, Expr_2$
$Store : Loc \to Loc$
$\cdot \to \cdot : < Expr, Store > \to < Loc, Store >$

$$\frac{}{< \mathtt{x}, \sigma > \to < l_{\mathtt{x}}, \sigma >} \quad \text{(Var)}$$

$$\frac{< e_1, \sigma > \to < l_1, \sigma_1 > \quad < e_2, \sigma_1 > \to < l_2, \sigma_2 >}{< (e_1, e_2), \sigma > \to < l_2, \sigma_2 >} \quad \text{(Seq)}$$

$$\frac{< e, \sigma > \to < l, \sigma' > \qquad l' \text{ fresh}}{< \&_i e, \sigma > \to < l', \sigma'[l' \mapsto l] >} \quad \text{(Addr)}$$

$$\frac{< e, \sigma > \to < l, \sigma' >}{< *e, \sigma > \to < \sigma'(l), \sigma' >} \quad \text{(Deref)}$$

$$\frac{< e_1, \sigma > \to < l_1, \sigma_1 > \quad < e_2, \sigma_1 > \to < l_2, \sigma_2 >}{< e_1 = e_2, \sigma > \to < l_2, \sigma_2[l_1 \mapsto \sigma_2(l_2)] >} \quad \text{(Asst)}$$

Soundness III

- For each syntactic location x, let $\phi(x)$ be the type τ such that $\vdash_A x : ref\,(p_{get} = p_{set} = \tau)$. If x is a program variable, then $\phi(x) = x$.
- From this, we get a result showing something like an upper and lower bound for Andersen-style analysis.

For any program e, if $< e, \sigma > \stackrel{*}{\rightarrow} < l, \sigma' >$ and $\forall x \in Dom\,(\sigma).\theta(\sigma(x)) \in P_A(\theta(x))$, then

(a) $\vdash_A e : \tau$ where $ref(p_{get} = p_{set} = \phi(\theta(l))) \subseteq \tau$, and
(b) $\forall x \in Dom\,(\sigma').\theta(\sigma'(x)) \in P_A(\theta(x))$

- From this, we have a final result showing the soundness of both Andersen's and Steensgard's analyses.

Corollary 7.7 (Soundness$_A$) For any program e, if $< e, \emptyset > \stackrel{*}{\rightarrow} < l, \sigma >$, then $\forall x \in Dom\,(\sigma).\theta(\sigma(x)) \in P_A(\theta(x))$.

Corollary 7.8 (Soundness$_S$) For any program e, if $< e, \emptyset > \stackrel{*}{\rightarrow} < l, \sigma >$, then $\forall x \in Dom\,(\sigma).\theta(\sigma(x)) \in P_S(\theta(x))$.

Implementation and results I

Andersen-style Analysis								
		And+Common			SH			
Name	Lines	Time (s)	Size	Sets	Time (s)	Size	Sets	$\frac{\text{SH Time}}{\text{Framework}}$
diff.diffh	293	0.13	42	19	0.06	42	19	0.45
genetic	324	0.12	34	16	0.07	34	16	0.62
anagram	344	0.11	33	25	0.07	34	26	0.60
allroots	428	0.04	11	7	0.04	11	7	0.90
ul	445	0.16	10	10	0.08	10	10	0.53
ks	574	0.26	190	55	0.17	222	62	0.66
compress	652	0.25	15	15	0.10	15	15	0.40
ft	1179	0.40	140	64	0.15	140	64	0.37
ratfor	1540	1.11	642	111	0.42	618	113	0.38
compiler	1895	0.50	406	29	0.40	406	29	0.80
eqntott	2316	1.72	506	159	0.47	296	158	0.27
assembler	2987	1.48	596	179	0.66	522	183	0.45
simulator	4230	6.51	14721	288	8.79	14377	289	1.35
ML-typecheck	4903	3.20	10070	254	2.03	9973	252	0.63
li	5798	473.72	809489	1314	3880.47	809834	1316	8.19
flex-2.4.7	9358	13.59	3929	373	45.28	3661	371	3.33
less-177	12108	8.43	33497	492	11.25	32794	504	1.33
make-3.72.1	15214	190.67	221937	1141	384.60	222147	1144	2.02
espresso	21583	713.16	249965	1932	343.56	243521	1928	0.48

Implementation and results II

Steensgaard-style Analysis								
		Ste+Common			**SH**			
Name	Lines	Time (s)	Size	Sets	Time (s)	Size	Sets	$\frac{\text{SH Time}}{\text{Framework}}$
diff.diffh	293	0.13	192	19	0.04	192	19	0.33
genetic	324	0.10	92	16	0.05	92	16	0.53
anagram	344	0.10	151	30	0.04	220	31	0.41
allroots	428	0.03	14	7	0.03	14	7	0.88
ul	445	0.13	10	10	0.08	11	11	0.61
ks	574	0.11	537	57	0.08	607	64	0.71
compress	652	0.15	15	15	0.10	15	15	0.63
ft	1179	0.13	223	64	0.10	259	73	0.76
ratfor	1540	0.70	13567	177	0.42	13154	181	0.60
compiler	1895	0.58	1080	49	0.18	1080	49	0.32
eqntott	2316	0.93	1495	160	0.30	1116	160	0.32
assembler	2987	0.92	4069	227	0.38	3201	229	0.41
simulator	4230	2.45	24012	304	1.38	25320	325	0.56
ML-typecheck	4903	1.29	13685	262	0.53	13556	261	0.41
li	5798	5.67	1054796	1409	100.03	1152974	1417	17.64
flex-2.4.7	9358	10.03	23775	415	2.04	25854	452	0.20
less-177	12108	2.73	147795	646	5.15	151839	656	1.89
make-3.72.1	15214	7.10	927441	1609	73.72	938203	1634	10.38
espresso	21583	12.51	315372	1935	27.23	316999	1970	2.18

References I

Alexander Aiken.
Introduction to set constraint-based program analysis.
Sci. Comput. Program., 35(2):79–111, 1999.

Lars Ole Andersen.
Program analysis and specialization for the c programming language.
Technical report, 1994.

Manuel F?hndrich and Alexander Aiken.
Program analysis using mixed term and set constraints.
In *IN PROCEEDINGS OF THE 4TH INTERNATIONAL STATIC ANALYSIS SYMPOSIUM*, pages 114–126.
Springer-Verlag, 1997.

References II


Marc Shapiro and Susan Horwitz.
Fast and accurate flow-insensitive points-to analysis.
In *In Symposium on Principles of Programming Languages*, pages 1–14, 1997.


Bjarne Steensgaard.
Points-to analysis in almost linear time, 1996.