Points-to Analysis by Type Inference of Programs with Structures and Unions

Bjarne Steensgaard

Language

$S ::= x =_S y$

$| \quad x =_S \&y$

$| \quad x =_S {*}y$

$| \quad x =_S \mathtt{allocate}(y)$

$| \quad x =_S \mathtt{op}(y_1 \ldots y_n)$

$| \quad x =_S \&y\text{->}n$

$| \quad x =_S \mathtt{fun}(f_1 \ldots f_n)\text{->}(r_1 \ldots r_m)\ S{*}$

$| \quad x_1 \ldots x_m =_{S1 \ldots Sm} p(y_1 \ldots y_n)$

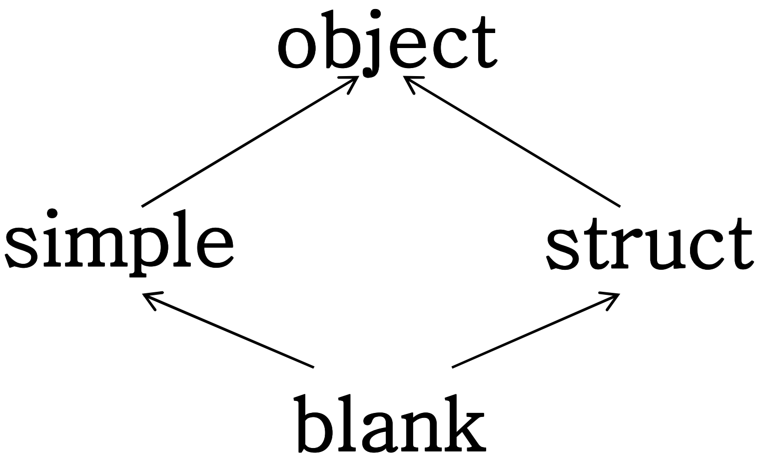
Types

$\tau ::= \bot \mid \textbf{simple}(\alpha,\lambda,s,p) \mid \textbf{struct}(m,s,p) \mid$
$\textbf{object}(\alpha,\lambda,s,p) \mid \textbf{blank}(s,p)$

Tracking

- Size $s = \text{SIZE} \mid \top$
- Offset $\alpha = \tau \times o$
- Struct elements $m = [\text{ID} \mapsto \tau]$ map
- Inconsistent usage

Types



Partial ordering tracks information flow between assigned-from location and assigned-to location. Sizes and offsets must be accommodated.

$$(\tau_1 \times o_1) \sqsubseteq_S (\tau_2 \times o_2)$$

$$a \trianglelefteq_S b$$

Expressiveness

- Aggregates
- Unions
- Data size

```
int * a;
struct { int b, int c } * d;

a = &d->c;
```

Algorithm starts off with Input:

$\texttt{a} =_s \texttt{\&d->c;}$

The Initial type of each program variable is “blank” to indicate that there is no access pattern.

τ_a : **blank**(s , {})

τ_d : **blank**(s , {})

```
int * a;
struct { int b, int c } * d;

a = &d->c;
```

We Match The Typing Rule

$$\texttt{x} =_{s} \texttt{\&y->n}$$

This promotes both types to “simple” to indicate they have the semantics of being accessed as a whole.

$\tau_a : \textbf{simple}(\tau_1 \text{ x } o_1, \bot, s, \{\})$

$\tau_1 : \textbf{blank}(s, \{\})$

$\tau_d : \textbf{simple}(\tau_2 \text{ x } o_2, \bot, s, \{\})$

$\tau_2 : \textbf{blank}(s, \{\})$

```
int * a;
struct { int b, int c } * d;

a = &d->c;
```

Since we are accessing the memory location described by τ_2 like a struct, we promote the type, and add a mapping to represent the member we are accessing.

τ_a : $\textbf{simple}(\tau_1 \times o_1, \bot, s, \{\})$ $\quad$ τ_1 : $\textbf{blank}(s, \{\})$

τ_d : $\textbf{simple}(\tau_2 \times o_2, \bot, s, \{\})$

τ_2 : $\textbf{struct}([c \mapsto \tau_3], s_{sizeof(d)}, \{\})$ $\quad$ τ_3 : $\textbf{blank}(s, \{\tau_2\})$

```
int * a;
struct { int b, int c } * d;

a = &d->c;
```

Finally, the algorithm unifies the accessed field type and the type pointed to by the variable being assigned

$$\text{unify}(\tau_1 : \mathbf{blank}(s, \{\}), \tau_3 : \mathbf{blank}(s, \{\tau_2\})) = \tau_{1,3}$$

```
int * a;
struct { int b, int c } * d;

a = &d->c;
```

Thus we get the following set of types:

$\tau_a : \textbf{simple}(\tau_{1,3} \times o_1, \perp, s, \{\})$

$\tau_d : \textbf{simple}(\tau_2 \times o_2, \perp, s, \{\})$

$\tau_2 : \textbf{struct}([c \mapsto \tau_{1,3}], s_{\mathrm{sizeof}(d)}, \{\})$

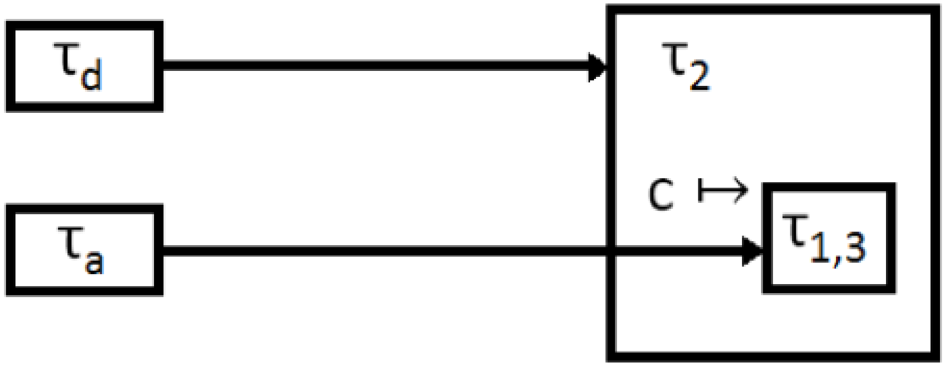
$\tau_{1,3} : \textbf{blank}(s, \{\tau_2\})$

```
int * a;
struct { int b, int c } * d;

a = &d->c;
```

$\tau_a : \textbf{simple}(\tau_{1,3}\ \mathrm{x}\ o_1, \bot, s, \{\}\ )$
$\tau_d : \textbf{simple}(\tau_2\ \mathrm{x}\ o_2, \bot, s, \{\}\ )$
$\tau_2\ : \textbf{struct}([c \mapsto \tau_{1,3}], s_{\mathrm{sizeof(d)}}, \{\})$
$\tau_{1,3} : \textbf{blank}(s, \{\tau_2\})$

Graphically, the types relate as follows:



Complexity (Theoretical)

- Any precise analysis is exponential (or worse!)
    - why?
- Does this matter for this analysis? Any analysis?

Complexity (Practical)

- S = # of variables in the program
- R = max # members in any structure
- Create O(S) type variables
- $O(RS\alpha(S,S))$
  - S passes with unions, each might loop over R elements
- No structs? $O(S\alpha(S,S))$
  - Look familiar?