CSC 6580
Spring 2020

Instructor: Stacy Prowell

Midterm

loop

Can you predict what this little program will do?

ECX is initialized to 5. Will this go around the loop five times? Six times? What will it print?

```
        section .text
        global main
        extern printf

main:   push rbp
        mov rbp, rsp

        mov ecx, 5
.top:   mov rdi, format
        mov esi, ecx
        mov eax, 0
        push rcx
        call printf wrt ..plt
        pop rcx
        loop .top

        leave
        ret

        section .data

format: db "rcx = %d",10,0
```

loop

Can you predict what this little program will do?

ECX is initialized to 5. Will this go around the loop five times? Six times? What will it print?

```
$ loopy
5
4
3
2
1
```

```
        section .text
        global main
        extern printf

main:   push rbp
        mov rbp, rsp

        mov ecx, 5
.top:   mov rdi, format
        mov esi, ecx
        mov eax, 0
        push rcx
        call printf wrt ..plt
        pop rcx
        loop .top

        leave
        ret

        section .data

format: db "rcx = %d",10,0
```

loop

Can you predict what this little program will do?

ECX is initialized to 5. Will this go around the loop five times? Six times? What will it print?

From the midterm:

`loop target`	Decrement RCX and, if not zero, jump to *target*. Otherwise continue to the next instruction.

```
        section .text
        global main
        extern printf

main:   push rbp
        mov rbp, rsp

        mov ecx, 5
.top:   mov rdi, format
        mov esi, ecx
        mov eax, 0
        push rcx
        call printf wrt ..plt
        pop rcx
        loop .top

        leave
        ret

        section .data

format: db "rcx = %d",10,0
```

Steensgaard’s Algorithm

Points-to Analysis in Almost Linear Time

Bjarne Steensgaard

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

rusa@research.microsoft.com

We present a

- **flow insensitive**,
- **interprocedural**
- **points-to analysis** algorithm

that has a desirable

- **linear space** and
- almost **linear time** complexity and

is also **very fast in practice**.

The Central Claim

The task of performing a points-to analysis has now been reduced to the task of inferring a typing environment under which a program is well-typed. More precisely, the typing environment we seek is the minimal solution to the well-typedness problem, *i.e.*, each location type variable in the typing environment describes as few locations as possible.

Source Language: Typing and Storage Shape

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$$
\begin{array}{ll}
\texttt{a:} & \tau_1 = \textbf{ref}(\tau_4 \times \bot) \\
\texttt{b:} & \tau_2 = \textbf{ref}(\tau_5 \times \bot) \\
\texttt{c:} & \tau_3 = \textbf{ref}(\tau_5 \times \bot) \\
\texttt{x:} & \tau_4 = \textbf{ref}(\bot \times \bot) \\
\texttt{y:} & \tau_5 = \textbf{ref}(\tau_4 \times \bot) \\
\texttt{z:} & \tau_4 \\
\texttt{p:} & \tau_6 = \textbf{ref}(\bot \times \bot)
\end{array}
$$

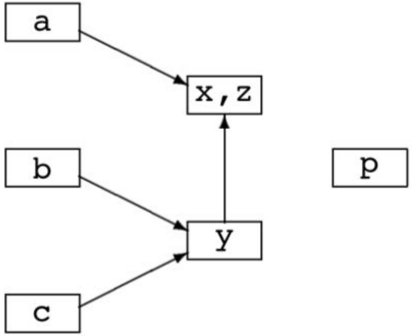


Figure 4: Example program, a typing of same that obeys the typing rules, and graphical representation of the corresponding storage shape graph. Note that variables `x` and `z` are described by the same type. Even though types τ_1 and τ_5 are structurally equivalent (as are τ_2 and τ_3, and τ_4 and τ_6), they are not considered the same types.

Source Language: Typing and Storage Shape

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$$
\begin{aligned}
&\texttt{a:}\ \ \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
&\texttt{b:}\ \ \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
&\texttt{c:}\ \ \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
&\texttt{x:}\ \ \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\texttt{y:}\ \ \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
&\texttt{z:}\ \ \tau_4\\
&\texttt{p:}\ \ \tau_6 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$

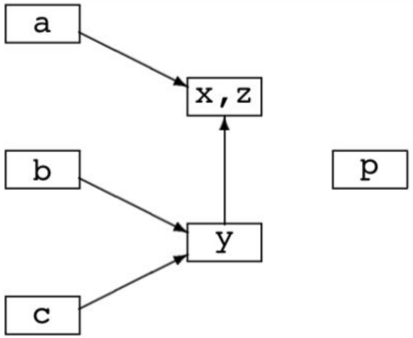


Figure 4: Example program, a typing of same that obeys the typing rules, and graphical representation of the corresponding storage shape graph. Note that variables `x` and `z` are described by the same type. Even though types τ_1 and τ_5 are structurally equivalent (as are τ_2 and τ_3, and τ_4 and τ_6), they are not considered the same types.

Source Language: Typing and Storage Shape

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

a: $\tau_1 = \mathbf{ref}(\tau_4 \times \perp)$
b: $\tau_2 = \mathbf{ref}(\tau_5 \times \perp)$
c: $\tau_3 = \mathbf{ref}(\tau_5 \times \perp)$
x: $\tau_4 = \mathbf{ref}(\perp \times \perp)$
y: $\tau_5 = \mathbf{ref}(\tau_4 \times \perp)$
z: τ_4
p: $\tau_6 = \mathbf{ref}(\perp \times \perp)$

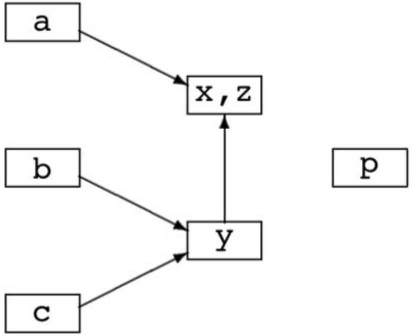


Figure 4: Example program, a typing of same that obeys the typing rules, and graphical representation of the corresponding storage shape graph. Note that variables x and z are described by the same type. Even though types τ_1 and τ_5 are structurally equivalent (as are τ_2 and τ_3, and τ_4 and τ_6), they are not considered the same types.

Something to Consider

Can you apply this to assembly?

Sketch of the Algorithm

The basic principle of the algorithm is that we start with the assumption that all variables are described by different types (type variables) and then proceed to merge types as necessary to ensure well-typedness of different parts of the program. Merging two types means replacing the two type variables with a single type variable throughout the typing environment. Joining is made fast by using fast union/find data structures. We first describe the initialization and our assumptions about how the program is represented. Then we describe how to deal with equalities and inequalities in the typing rules in a manner ensuring that we only have to process each statement in the program exactly once. Finally we argue that the algorithm has linear space complexity and almost linear time complexity.

Aside: Union-Find Data Structures

(Also called a **disjoint set** data structure.)

This is a data structure that tracks a *partition* of a set of elements (like type variables). Blocks of the partition can be quickly combined (union) and it is fast to check whether elements are in the same set (find).

{1} {2} {3} {4,5}

merge {2} and {4,5}

{1} {2,4,5} {3}

Aside: Union-Find Data Structures

- Initially each variable is an equivalence class
- If two variables are the same, their equivalence classes are merged

Aside: Union-Find Data Structures

Using both *path* ***compression***, ***splitting***, *or* ***halving*** and *union by* ***rank*** *or* ***size*** ensures that the amortized time per operation is only $O(\alpha(n))$,[4][5] which is optimal,[6] where $\alpha(n)$ is the inverse Ackermann function. This function has a value $\alpha(n) < 5$ for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.

- Wikipedia

Phase 1

- All variables have a corresponding type variable, and all types are initially:

$$\mathbf{ref}(\perp \times \perp)$$

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$\mathbf{a}$: $\tau_1 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{b}$: $\tau_2 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{c}$: $\tau_3 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{x}$: $\tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{y}$: $\tau_5 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{z}$: $\tau_6 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{p}$: $\tau_7 = \mathbf{ref}(\perp \times \perp)$

Phase 2

- Process each statement exactly once.
    - Use the type rules to determine if two type variables must be joined.
    - If the left-hand side type is $\perp$, then there is no need to join.
    - If the left-hand side type is not $\perp$, then the two types must be joined.

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$\mathbf{a}$: $\tau_1 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{b}$: $\tau_2 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{c}$: $\tau_3 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{x}$: $\tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{y}$: $\tau_5 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{z}$: $\tau_6 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{p}$: $\tau_7 = \mathbf{ref}(\perp \times \perp)$

Phase 2

Now a holds a reference to x, so the type of a must be a reference to the type of x.

$$\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)$$

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$\mathbf{a}\colon \tau_1 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{b}\colon \tau_2 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{c}\colon \tau_3 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{z}\colon \tau_6 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)$

Phase 2

Now b holds a reference to y, so the type of b must be a reference to the type of y.

$$\mathbf{b}\text{: } \tau_2 = \mathbf{ref}(\tau_5 \times \bot)$$

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$\mathbf{a}$: $\tau_1 = \mathbf{ref}(\tau_4 \times \bot)$
$\mathbf{b}$: $\tau_2 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{c}$: $\tau_3 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{x}$: $\tau_4 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{y}$: $\tau_5 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{z}$: $\tau_6 = \mathbf{ref}(\bot \times \bot)$
$\mathbf{p}$: $\tau_7 = \mathbf{ref}(\bot \times \bot)$

Phase 2

Now y holds a reference to z, so the type of y must be a reference to the type of z.

$$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_6 \times \bot)$$

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$$\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)$$
$$\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{c}\colon \tau_3 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{z}\colon \tau_6 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)$$

Phase 2

Now y holds a reference to x, so the type of y must be a reference to the type of x.

$$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_6 \times \bot)$$
$$\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$$

Now the left hand side is not bottom; we have to merge the two type variables: τ_4, τ_6

$$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_4 \times \bot)$$
$$\mathbf{z}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$$

```
a = &x
b = &y
if p then
   y = &z;
else
   y = &x
fi
c = &y
```

$$\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)$$
$$\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{c}\colon \tau_3 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_6 \times \bot)$$
$$\mathbf{z}\colon \tau_6 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)$$

Phase 2

Now c holds a reference to y, so the type of c must be a reference to the type of y.

$$\mathbf{c}\colon \tau_3 = \mathbf{ref}(\tau_5 \times \bot)$$

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$$\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)$$
$$\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{c}\colon \tau_3 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_4 \times \bot)$$
$$\mathbf{z}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)$$

Phase 2

Complete!

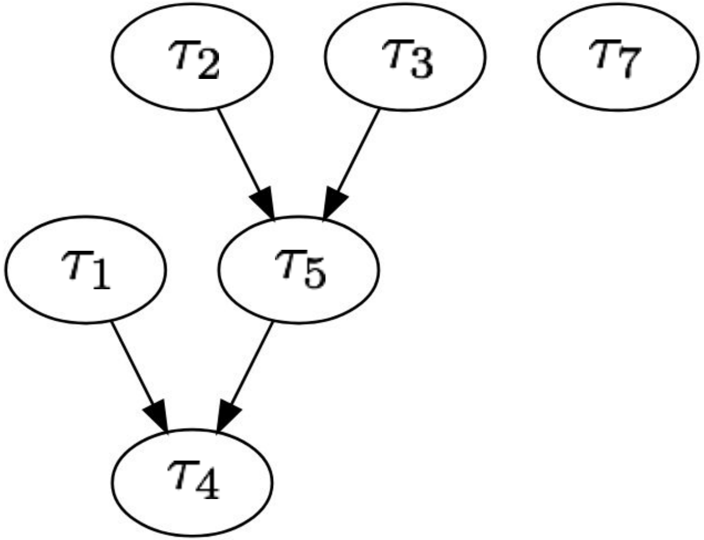
```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

$\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \perp)$
$\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \perp)$
$\mathbf{c}\colon \tau_3 = \mathbf{ref}(\tau_5 \times \perp)$
$\mathbf{x}\colon \tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_4 \times \perp)$
$\mathbf{z}\colon \tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{p}\colon \tau_7 = \mathbf{ref}(\perp \times \perp)$

Phase 2

We can graph this.

$$
\begin{aligned}
&\mathbf{a}\!: \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{b}\!: \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{c}\!: \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{x}\!: \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{y}\!: \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{z}\!: \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{p}\!: \tau_7 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$

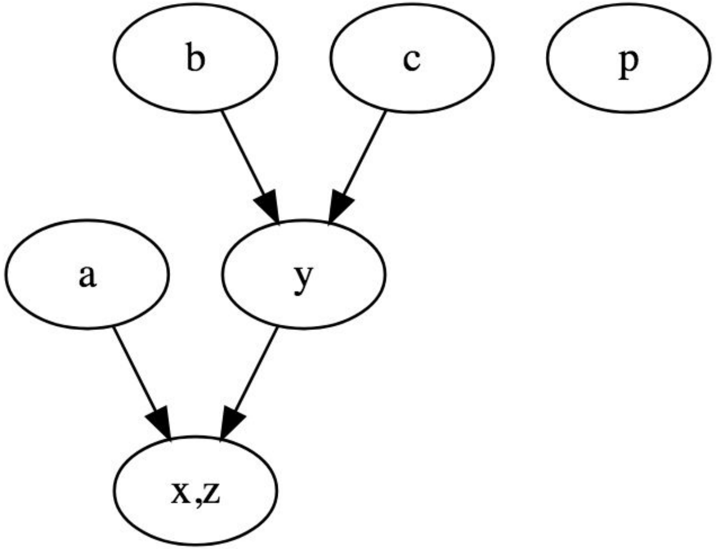


Phase 2

We can use the variables as labels.

This gives us the complete “points to” analysis of the program.

$$
\begin{aligned}
&\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{c}\colon \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{z}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$



Next Time: More!