CSC 6580
Spring 2020

Instructor: Stacy Prowell

Midterm

Ghidra, P-Code, and Semantics

A Simple Semantics*

`inc rax`	`rax := rax + 1 ; of := ...`
`lea rcx, [rax*8]`	`rcx := rax * 8`
`push rcx`	`rsp := rsp - 8 ; M[rsp] := rcx`
`push rax`	`rsp := rsp - 8 ; M[rsp] := rax`
`mov rdi, 21`	`rdi := 21`
`call _optc`	`...do whatever _optc does...`
`pop rcx`	`rcx := M[rsp] ; rsp := rsp + 8`
`pop rax`	`rax := M[rsp] ; rsp := rsp + 8`

```
; want to know rax here
```

* All math takes place in a finite-length bit field, so $a+b$ is really $(a+b) \bmod 2^{64}$, etc.

Aside: Ghidra P-Code

Ghidra is a reverse engineering tool developed by the NSA and made available as open source software.

https://ghidra-sre.org/

It can disassemble, do a passable job of decompilation, and has a semantics for many processors, including X86-64.

Aside: Ghidra P-Code

Opening the code in Ghidra displays the usual disassembly.

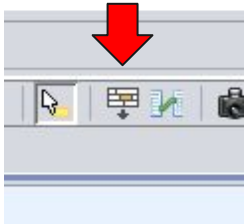
But… you can display more.

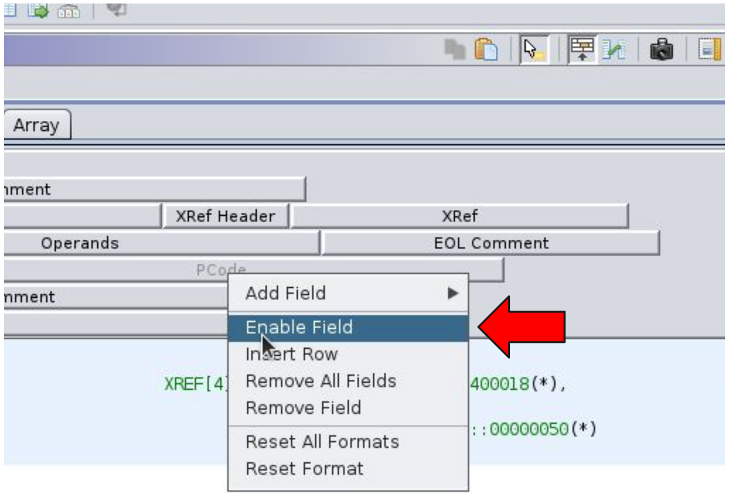
```
00401000 48 ff c0         INC     RAX
00401003 48 8d 0c         LEA     RCX,[RAX*0x8]
         c5 00 00
         00 00
0040100b 51               PUSH    RCX
0040100c 50               PUSH    RAX
0040100d bf 15 00         MOV     EDI,0x15
         00 00
00401012 e8 0f 00         CALL    _optc
         00 00
00401017 59               POP     RCX
00401018 58               POP     RAX
```

Aside: Ghidra P-Code

To enable P-Code display:

- Click on the "jenga" button above the disassembly window
- Switch to the Instruction/Data tab and find PCode
- Right-click PCode and select Enable Field





Aside: Ghidra P-Code

...and the listing is populated with P-Code semantic information!

Find the P-Code reference manual in the Ghidra distribution, or online:

ghidra.re/courses/languages/html/pcoderef.html

```
00401000 48 ff c0       INC      RAX
                                          OF = INT_SCARRY RAX, 1:8
                                          RAX = INT_ADD RAX, 1:8
                                          SF = INT_SLESS RAX, 0:8
                                          ZF = INT_EQUAL RAX, 0:8
00401003 48 8d 0c       LEA      RCX,[RAX*0x8]
         c5 00 00
         00 00
                                          $U6d0:8 = INT_MULT RAX, 8:8
                                          RCX = COPY $U6d0
0040100b 51             PUSH     RCX
                                          $U2510:8 = COPY RCX
                                          RSP = INT_SUB RSP, 8:8
                                          STORE ram(RSP), $U2510
0040100c 50             PUSH     RAX
                                          $U2510:8 = COPY RAX
                                          RSP = INT_SUB RSP, 8:8
                                          STORE ram(RSP), $U2510
0040100d bf 15 00       MOV      EDI,0x15
         00 00
                                          RDI = COPY 21:8
00401012 e8 0f 00       CALL     _optc
         00 00
                                          RSP = INT_SUB RSP, 8:8
                                          STORE ram(RSP), 0x401017:8
                                          CALL *[ram]0x401026:8
00401017 59             POP      RCX
                                          RCX = LOAD ram(RSP)
                                          RSP = INT_ADD RSP, 8:8
00401018 58             POP      RAX
                                          RAX = LOAD ram(RSP)
                                          RSP = INT_ADD RSP, 8:8
```

Aside: Ghidra P-Code

...and the listing is populated with P-Code semantic information!

Find the P-Code reference manual in the Ghidra distribution, or online:

ghidra.re/courses/languages/html/pcoderef.html

```
00401000 48 ff c0       INC     RAX
                                       OF = INT_SCARRY RAX, 1:8
                                       RAX = INT_ADD RAX, 1:8
                                       SF = INT_SLESS RAX, 0:8
                                       ZF = INT_EQUAL RAX, 0:8
00401003 48 8d 0c       LEA     RCX,[RAX*0x8]
         c5 00 00
         00 00
                                       $U6d0:8 = INT_MULT RAX, 8:8
                                       RCX = COPY $U6d0
0040100b 51             PUSH    RCX
                                       $U2510:8 = COPY RCX
                                       RSP = INT_SUB RSP, 8:8
                                       STORE ram(RSP), $U2510
0040100c 50             PUSH    RAX
                                       $U2510:8 = COPY RAX
                                       RSP = INT_SUB RSP, 8:8
                                       STORE ram(RSP), $U2510
0040100d bf 15 00       MOV     EDI,0x15
         00 00
                                       RDI = COPY 21:8
00401012 e8 0f 00       CALL    _optc
         00 00
                                       RSP = INT_SUB RSP, 8:8
                                       STORE ram(RSP), 0x401017:8
                                       CALL *[ram]0x401026:8
00401017 59             POP     RCX
                                       RCX = LOAD ram(RSP)
                                       RSP = INT_ADD RSP, 8:8
00401018 58             POP     RAX
                                       RAX = LOAD ram(RSP)
                                       RSP = INT_ADD RSP, 8:8
```

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

After each instruction we see the P-Code representation of the semantics.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

After each instruction we see the P-Code representation of the semantics.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

After each instruction we see the P-Code representation of the semantics.

INT_SCARRY

Parameters	Description
input0	First varnode to add.
input1	Second varnode to add.
output	Boolean result containing signed overflow condition.

Semantic statement

```
output = scarry(input0,input1);
```

This operation checks for signed addition overflow or carry conditions. If the result of adding input0 and input1 as signed integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

The two comma-separated items after `INT_SCARRY` are the arguments. The first is `RAX`, which we recognize, and the second is the value 1, represented as an eight-byte integer.

INT_SCARRY

Parameters	Description
input0	First varnode to add.
input1	Second varnode to add.
output	Boolean result containing signed overflow condition.

Semantic statement

```
output = scarry(input0,input1);
```

This operation checks for signed addition overflow or carry conditions. If the result of adding input0 and input1 as signed integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

Note that we specify ZF by checking to see if RAX is zero. This only works if RAX is *already* set to the incremented value… so these semantics are *sequential* assignments, and order matters.

Our simple semantics are *concurrent*, so the order does not matter.

More x86-64 Architecture

Real Mode

The CPU can operate in one of two modes: *real* and *protected*.

The CPU always starts in **real mode**.

- An address in real model is the same address in real memory; memory is directly accessed
- No virtual memory; no memory protection; no protection levels; no multitasking
- There is a 20-bit address space: $2^{20} = 2^{10} * 2^{10}$, or 1024 * 1024, or 1MiB
- Addressing is done using 16-bit registers:
  A segment register is shifted left four bits, and then a 16-bit offset address is added

Also look up "unreal mode."

Protected Mode

Protected mode (introduced in 80286) brings:

- Memory protection (writeable, executable) and protection levels
- Virtual memory and larger address space
- Multitasking

These things require data structures to tell the processor what code is privileged, what memory is protected, etc. At boot the operating system switches to protected mode.

Protected Mode

Before getting to protected mode, an operating system does the following.

Set up important tables:

- The Global Descriptor Table (GDT)
- The Interrupt Descriptor Table (IDT)

These tables are located in physical memory by the global descriptor table register (GDTR) and the interrupt descriptor table register (IDTR). These registers hold physical addresses, not virtual addresses, and are set while in real mode with the `lgdt` and `lidt` instructions. They each hold a four-byte address and a two-byte length.

You can get the table addresses with the `sgdt` and `sidt` instructions.

The Global Descriptor Table (GDT)

Defines all *segments*. These control how logical addresses are translated into linear addresses.

- Base
- Size
- Access privileges (read, write, execute)

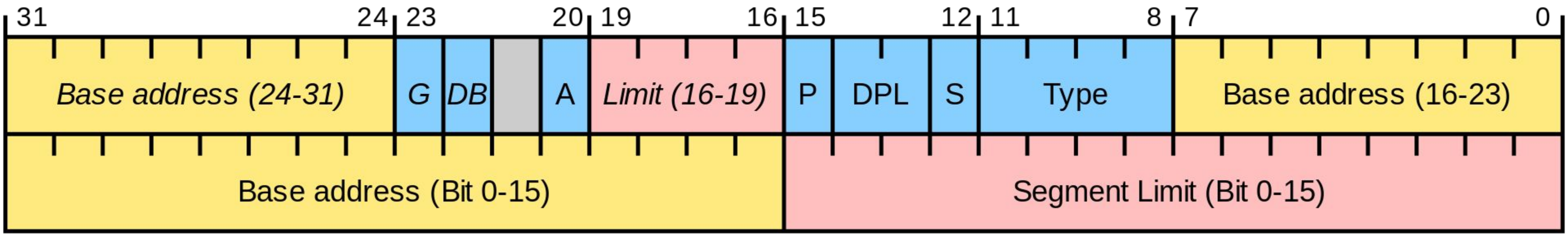


Image source: Wikipedia

The Interrupt Descriptor Table (IDT)

Maps interrupt requests (IRQs) to the correct handlers

A table of 256 interrupt vectors -- first 32 are for the processor

Hardware, software, and processor exceptions come here to get the address of the appropriate interrupt handler.

More?

Sure:
https://samypesse.gitbook.io/how-to-create-an-operating-system/

Incomplete at this time, but what’s there is pretty solid.

Model-Specific Registers (MSR)

These are registers that control processor features, and are specific to a particular model of the processor.

- Use `cpuid` to check for processor features…
- Then use `rdmsr` to read and `wrmsr` to write to these registers.

List of model specific registers?
http://www.cs.inf.ethz.ch/stricker/lab/doc/intel-part4.pdf

See also: Specter vulnerability!

Pointer analysis

Pointers

```
int (*f)(char **) =
  check ? func1 : func2;
```

Here `f` can be {`func1`, `func2`}.

Pointer analysis is a static code analysis technique that determines which pointers (or references) can point to which storage locations.

Two pointers that point to the same storage location are **aliased**.

Why do we need this?

Aliases

```
*ptr = x + y;
   z = x + y;
```

Do we have to compute `x + y` more than once?

Aliases

```
*ptr = x + y;
   z = x + y;
```

Do we have to compute `x + y` more than once?

Consider more context.

Aliases

```
int * ptr = &x;
*ptr = x + y;
   z = x + y;
```

Do we have to compute `x + y` more than once?

Consider more context.

Now we know `*ptr` is an alias for `x`.

Aliases

What happens here?

Is `p` live? Is `x` equal to `*p`?

What if `p = &x`?

Three cases: `p` (*is*, *is not*, *might be*) `&x`.

```
 x = 5;
*p = 15;
 y = x + 10;
```

Aliases in Assembly

```
mov ecx, [esi*8 + reftable]
push ecx
```

Which is the “real” value?

- The value in `ECX`
- The value on the top of the stack
- The value stored in `[ESI*8 + reftable]`

Algorithms

- Flow-Sensitive Analysis

- Flow-Insensitive Analysis
  - Steensgaard's algorithm
  - Andersen's algorithm

Algorithms

- Flow-Sensitive Analysis

- Flow-Insensitive Analysis
  - Andersen's algorithm
  - **Steensgaard's algorithm**

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**.

Points-to Analysis in Almost Linear Time

Bjarne Steensgaard

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

rusa@research.microsoft.com

Don't care about flow structures.

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**.

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**.

Analyze the whole program.

Points-to Analysis in Almost Linear Time

Bjarne Steensgaard

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

rusa@research.microsoft.com

Solve the “points to” problem.

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**.

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**.

Not pathological.

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**.

Theoretically fast.

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**.

Theoretically fast.

Linear with respect to the length of the program (number of lines).

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**.

Also *actually* fast.

Source Language (Capabilities)

The paper introduces a little language to be analyzed. This language has

- pointers to locations,
- pointers to functions,
- dynamic allocation (allocate(y)), and
- computing addresses of variables (&y).

$$
\begin{array}{lcl}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language (Control Structures)

We don’t care about flow structures since the analysis should be flow-insensitive, so add the flow structures you like.

Don’t get too worked up on this; it is pseudocode, but you should see how you could convert a program into an equivalent source language structure.

$$
\begin{array}{rcl}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

Variables are assumed to have *unique names*.

$$
\begin{array}{lcl}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

The usual “address of” operator.

$$
\begin{array}{lll}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

Dereferencing as an rvalue.

$$
\begin{array}{lll}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

Here op is any "primitive operation" such as arithmetic or computing an offset.

$$
\begin{array}{lll}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

Dereferencing as an lvalue.

$$
\begin{array}{rcl}
S & ::= & \mathsf{x} = \mathsf{y} \\
& | & \mathsf{x} = \&\mathsf{y} \\
& | & \mathsf{x} = *\mathsf{y} \\
& | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
& | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
& | & *\mathsf{x} = \mathsf{y} \\
& | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
& | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

Declaring a function with multiple parameters and multiple returns.

$$
\begin{array}{lcl}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language

Invoking a function with multiple arguments and multiple returns.

$$
\begin{array}{lcl}
S & ::= & \mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \&\mathsf{y} \\
 & | & \mathsf{x} = *\mathsf{y} \\
 & | & \mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} = \mathsf{allocate}(\mathsf{y}) \\
 & | & *\mathsf{x} = \mathsf{y} \\
 & | & \mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Source Language (Example)

```
fact = fun(x)→(r)
   if lessthan(x 1) then
      r = 1
   else
      xminusone = subtract(x 1)
      nextfac = fact(xminusone)
      r = multiply(x nextfac)
   fi

result = fact(10)
```

$$
\begin{array}{lll}
S & ::= & \mathrm{x} = \mathrm{y} \\
 & | & \mathrm{x} = \&\mathrm{y} \\
 & | & \mathrm{x} = *\mathrm{y} \\
 & | & \mathrm{x} = \mathrm{op}(\mathrm{y}_1 \ldots \mathrm{y}_n) \\
 & | & \mathrm{x} = \mathrm{allocate}(\mathrm{y}) \\
 & | & *\mathrm{x} = \mathrm{y} \\
 & | & \mathrm{x} = \mathrm{fun}(\mathrm{f}_1 \ldots \mathrm{f}_n) \rightarrow (\mathrm{r}_1 \ldots \mathrm{r}_m)\ S^* \\
 & | & \mathrm{x}_1 \ldots \mathrm{x}_m = \mathrm{p}(\mathrm{y}_1 \ldots \mathrm{y}_n)
\end{array}
$$

Source Language (Types)

Types are important; they tell us memory shapes and how pointers are used.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

The type of a thing pointed to. For composites (structs) this is still a single thing.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \bot \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \bot \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

Describing each element in a composite object by separate types would, for most imperative languages, imply that the size of the storage shape graph could potentially be exponential in the size of the input program (*e.g.*, by extreme use of `typedef` and `struct` in C). Describing the elements of composite objects by separate types may still be desirable, as the sum of sizes of variables is unlikely to be exponential in the size of the input program. Extending the type system to do so is not addressed in the present paper.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

Describing each element in a composite object by separate types would, for most imperative languages, imply that the size of the storage shape graph could potentially be exponential in the size of the input program (*e.g.*, by extreme use of `typedef` and `struct` in C). Describing the elements of composite objects by separate types may still be desirable, as the sum of sizes of variables is unlikely to be exponential in the size of the input program. Extending the type system to do so is not addressed in the present paper.

$$
\begin{aligned}
\alpha &::= \tau \times \lambda \\
\tau &::= \perp \mid \mathbf{ref}(\alpha) \\
\lambda &::= \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{aligned}
$$

Left as an exercise for the reader.

Source Language (Types)

The type of a function invocation includes the types of the arguments and the types of the returns.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

???

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

The “bottom” type, sometimes called simply “bot.” It’s the bottom of the type lattice. Think of it as “nothing.” In this case it is a non-pointer.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

Type theory is a fascinating and rich topic. Go and Google the Curry–Howard(–Lambek) correspondence.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

Simply a value. Might be

- a **location** or a pointer to a location, or
- a **function** or a pointer to a function.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \bot \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \bot \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Types)

Recursive types are allowed.

$$
\begin{aligned}
\alpha &::= \tau \times \lambda \\
\tau &::= \bot \mid \mathbf{ref}(\alpha) \\
\lambda &::= \bot \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{aligned}
$$

Source Language (Types)

Additionally there can be type variables. These are needed so recursive types can be written down.

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Source Language (Typing)

The **typing rules** specify when a program is *well-typed*.

A **well-typed program** is one for which the

- static storage shape graph indicated by the types is a safe (conservative) description of
- all possible dynamic (runtime) storage configurations.

The typing rules are given as inequalities, so they *constraint* but do not necessarily *determine* the types.

Aside: Argument Schemas

All men are mortal
Socrates is a man

Socrates is mortal

Aside: Argument Schemas

All men are mortal
Socrates is a man

Socrates is mortal

A logical argument schema.

Aside: Argument Schemas

All men are mortal
Socrates is a man

Socrates is mortal

A set of *premises*. This is our "database" of true statements.

Aside: Argument Schemas

All men are mortal
Socrates is a man

Socrates is mortal

A *conclusion*. This is true when the premises are true.

Aside: Argument Schemas

Modus Ponens
"mode that affirms"

$$\begin{array}{rl} & p \\ & p \to q \\ \hline \therefore & q \end{array}$$

Modus Tollens
"mode that denies"

$$\begin{array}{rl} & \neg q \\ & p \to q \\ \hline \therefore & \neg p \end{array}$$

Aside: Entailment

$$\vdash Q$$

I know Q is true.
Because.

Aside: Entailment

The “logical turnstile.” We can pronounce it “entails.” It typically represents provability or derivability.

$$\boxed{\vdash} Q$$

I know Q is true.
Because.

Aside: Entailment

$$P \vdash Q$$

I know Q is true.
Because I know P.

I have derived (or I can prove) Q from P.

Source Language (Typing)

A too strict typing rule A:

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

Given typing rule A, if I can derive that x has type ref(α) and y has type ref(α),then I can conclude that typing rule A correctly types x = y.

Source Language (Typing)

A too strict typing rule A:

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \qquad A \vdash \mathsf{y} : \mathbf{ref}(\alpha)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

This would force us to assume too much about x and y. If x and y are later used to hold pointers to different locations, this would require those locations to have the same type.

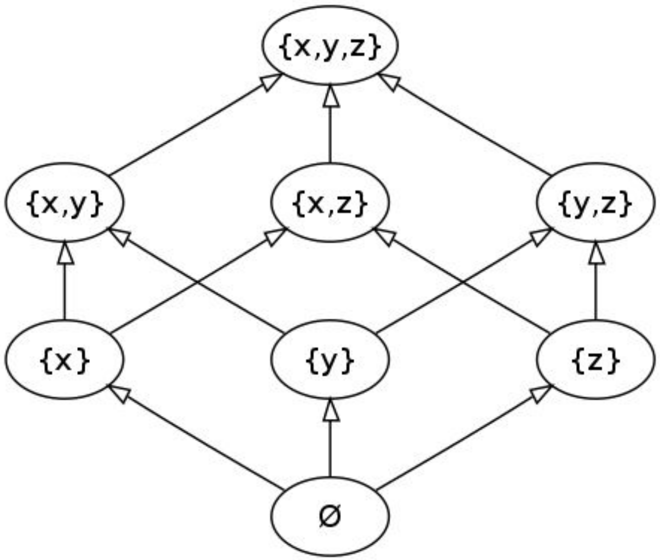
We want a more relaxed rule: Given an assignment x = y, the content component types for x and y need only be the same if y may contain a pointer.

Aside: Partial Order

A **partial order** is a relation $\trianglelefteq \subseteq A \times A$ that is:

- reflexive,
- antisymmetric, and
- transitive.

A partial order relates elements of some set, but not necessarily *all* elements of a set. If the relation is defined for all elements, then it is a *total order*.



source: Wikipedia

Source Language: Typing

We define a partial order among types.

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

Source Language: Typing

We define a partial order among types.

Reminder: Anything not a location or a pointer to a location.

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \boxed{\perp}) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

Source Language: Typing

We define a partial order among types.

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

This generalizes to any sequence because of recursive types. But note, they have to have the same length.

Source Language: Typing

We define a partial order among types.

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

You can think of this as t_1 "fits in" t_2.

Source Language: Typing

Now we can state a “good” rule for typing x = y:

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

Source Language: Typing

Now we can state a "good" rule for typing x = y:

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

If we are assigning the value of y to the variable x, then x needs to hold y. Thus y can be primitive and x a location, or x and y can have the same type.

$$\alpha_1 \trianglelefteq \alpha_2 \iff (\alpha_1 = \bot) \vee (\alpha_1 = \alpha_2)$$

Source Language: Type all the things!

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \quad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\mathbf{ref}(\alpha_2) \times \_) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = *\mathsf{y})}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \quad A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha_i) \quad \forall i \in [1 \ldots n] : \alpha_i \trianglelefteq \alpha}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ \ldots \ \mathsf{y}_n))}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\_) \times \_)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{allocate}(\mathsf{y}))}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\alpha_1) \times \_) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(*\mathsf{x} = \mathsf{y})}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \quad A \vdash \mathsf{f}_i : \mathbf{ref}(\alpha_i) \quad A \vdash \mathsf{r}_j : \mathbf{ref}(\alpha_{n+j}) \quad \forall s \in S^* : A \vdash \mathit{welltyped}(s)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m) \ S^*)}$$

$$\frac{A \vdash \mathsf{x}_j : \mathbf{ref}(\alpha'_{n+j}) \quad A \vdash \mathsf{p} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \quad A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha'_i) \quad \forall i \in [1 \ldots n] : \alpha'_i \trianglelefteq \alpha_i \quad \forall j \in [1 \ldots m] : \alpha_{n+j} \trianglelefteq \alpha'_{n+j}}{A \vdash \mathit{welltyped}(\mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

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.

Next Time: Pointers