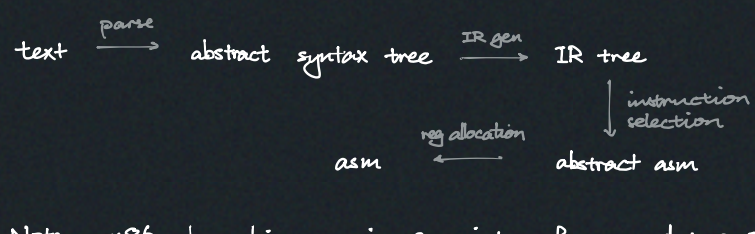


Lec 2

Translation to abstract assembly

Recall



Note: x86 has bin ops in 2-register form $d \leftarrow s_1 \oplus s_2$
 3-register form usually easier

The translation using Maximal Munch

↳ always try to match deepest possible pattern

Expression translation - recursion!

$$\text{trans}(e_1 \oplus e_2) = \begin{matrix} \text{trans}(e_1) \\ \text{trans}(e_2) \end{matrix}$$

$r_1, r_2 \leftarrow \Delta$ how do we get them?
 ↳ pass in tup to translate

$\text{trans}(d, e) \Rightarrow \text{Vec}(i)$ that computes e and put in destination d
 ↳ expression
 ↳ destination
 aka cogen

<u>e</u>	<u>trans(e, d)</u>
X (var)	$d \leftarrow X$
C (constant)	$d \leftarrow C$
$e_1 \oplus e_2$	$\text{trans}(t_1, e_1)$ fresh temps $\text{trans}(t_2, e_2)$ viz. not used elsewhere $d \leftarrow t_1 \oplus t_2$ (usually no need to declare them but design choice)

Statement translation

$\text{trans}'(s) \Rightarrow \text{Vec}(i)$ that impls s
 ↳ statement

<u>s</u>	<u>trans'(s)</u>
$X = e$	$\text{trans}(X, e)$
return e	$\text{trans}(\text{ret}, e)$ return reg
	return

Tests

IR
$$\begin{matrix} z = (x+1) * (y*4) \\ \text{return } z \end{matrix} \triangleq p$$

Trace $\text{trans}'(p)$
 $= \text{trans}'(z = (x+1) * (y*4)), \text{trans}'(\text{return } z)$
 $= \text{trans}(z, (x+1) * (y*4)), \text{trans}(\text{ret}, z), \text{return}$
 $= \text{trans}(t_1, x+1), \text{trans}(t_2, y*4), z \leftarrow t_1 * t_2, \text{ret} \leftarrow z, \text{return}$
 $= t_3 \leftarrow x, t_4 \leftarrow 1, t_1 \leftarrow t_3 + t_4, t_5 \leftarrow y, t_6 \leftarrow 4, t_2 \leftarrow t_5 * t_6, z \leftarrow t_1 * t_2, \text{ret} \leftarrow z, \text{return}$
] ... could have been ret ← t₁ * t₂] probably correct but lots of room for improvement

Ideas to improve

- special cases e.g. $\text{trans}(d, c \oplus e) \mapsto \dots$
- instruction count minimisation ← possible to some extent, undecidable if things get too complicated
- 2nd pass on the output to optimise ← also helps optimise some of the source code!
- different translation e.g. not pass in destination ↳ might save move ... but causes other problems to address

Standard approach: not worry about translate quality, optimise later

Replace + Dead code elimination

ⓐ remove dead code ⓑ replace

```

t3 ← x, t4 ← 1, t1 ← t2 + t4,
t5 ← y, t6 ← 4, t2 ← t5 * t6,
z ← t1 * t2,
ret ← z,
return
  
```

Constant Propagation

Goal: elim move $t \leftarrow c$ by replacing t with c in p but stop replacing if t occurs again

Ex. $t \leftarrow 4$ $t \leftarrow 4$
 $x \leftarrow t + 1$ $x \leftarrow 4 + 1$ ← replace here
 $t \leftarrow 5$ $t \leftarrow 5$
 $\text{ret} \leftarrow t$ $\text{ret} \leftarrow t$ ← but not here
 return return

Copy Propagation

Goal: elim move $d \leftarrow t$ by replacing d with t in p but stop if t or d is written □

Ex. $t \leftarrow 5 + 1$ $t \leftarrow 5 + 1$ (Copy Prop sometimes help)
 $d \leftarrow t$ $d \leftarrow t$ ← can't really elim this
 $x \leftarrow d + 1$ $x \leftarrow t + 1$ ← replace d with t here
 $t \leftarrow 5 + 2$ $t \leftarrow 5 + 2$
 $\text{ret} \leftarrow d + 1$ $\text{ret} \leftarrow d + 1$ ← but not here
 return return

Static Single Assignment (SSA) form ← LLVM uses this

Enforce that every temp is assigned at most once
 ↳ then write check □ not needed in Copy Prop

Converting to SSA: use version numbers

Ex. $t_0 \leftarrow 5 + 1$
 $d_0 \leftarrow t_0$
 $x_0 \leftarrow d_0 + 1$
 $t_1 \leftarrow 5 + 2$
 ⋮

- This simplifies optimisation, but uses many temps that it makes reg allocation harder
- Tricky to do in loops & branching

Register Allocation

abstract asm $\xrightarrow[\text{to do with temps}]{\text{figure out what}}$ real asm

Constraint usually only 14 regs to use on x86

- Problems
- when to reuse register?
 - which regs to keep
 - stack allocation in case of spill
 - how to use as many regs as possible?

Strategies

- naive: put all temps on stack SLOW!
- ...

Representation choice

- on x86 level
- on 3-reg assembly ← recommended
- on abstract assembly