Lec 2 Translation to abstract assembly

Recall

text —parse→ abstract syntax tree —IR gen→ IR tree

IR tree —instruction selection→ abstract asm —reg allocation→ asm

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!

trans ($e_1 \oplus e_2$) =
trans (e_1)
trans (e_2)
$r_1 + r_2$ ← ⚠ how do we get them?
→ pass in tmp to translate

trans (d, e) ⇒ Vec<i> that computes e and put in destination d
(e: expression; d: destination)
aka cogen

e	trans (e, d)
X (var)	$d \leftarrow X$
c (constant)	$d \leftarrow c$
$e_1 \oplus e_2$	trans (t_1, e_1) trans (t_2, e_2) $d \leftarrow t_1 \oplus t_2$

t_1, t_2: fresh temps viz. not used elsewhere (usually no need to declare them but design choice)

Statement translation

trans' (s) ⇒ Vec<i> that impls s
(s: statement)

s	trans' (s)
X = e	trans (X, e)
return e	trans (ret, e) return

ret: return reg

Tests

IR
z = (x + 1) * (y * 4)
return z
] ≙ p

Trace
trans'(p)
= trans'(z = (x + 1) * (y * 4)),
trans'(return z)
= trans (z , (x + 1) * (y * 4)),
trans (ret , z),
return
= trans (t_1 , x + 1), trans (t_2 , y * 4), $z \leftarrow t_1 * t_2$,
ret ← z , return
= $t_3 \leftarrow x$, $t_4 \leftarrow 1$, $t_1 \leftarrow t_3 + t_4$,
$t_5 \leftarrow y$, $t_6 \leftarrow 4$, $t_2 \leftarrow t_5 * t_6$,
$z \leftarrow t_1 * t_2$,
ret ← z ,
return

(... could have been $ret \leftarrow t_1 * t_2$)
(probably correct but lots of room for improvements)

Ideas to improve

→ special cases e.g. trans (d, $c \oplus e$) ↦ ...
→ 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

② remove dead code ① replace

~~$t_3 \leftarrow x$, $t_4 \leftarrow 1$,~~ $t_1 \leftarrow t_3 + t_4$, (x, 1)
$t_5 \leftarrow y$, $t_6 \leftarrow 4$, $t_2 \leftarrow t_5 * t_6$,
$z \leftarrow t_1 * t_2$,
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
$t \leftarrow 5$	$t \leftarrow 5$
$ret \leftarrow t$	$ret \leftarrow t$ ← 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 [1]

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
$t \leftarrow 5 + 2$	$t \leftarrow 5 + 2$
$ret \leftarrow d + 1$	$ret \leftarrow d + 1$ ← 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[1] 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 —(figure out what to do with temps)→ 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