

# Compiler Construction

⋈ Linearization ⋈

# From HIR to LIR

## Inadequacy of HIR

- No nested sequences
- Assembly is imperative: there is no “expression”
- Calling conventions
- Two Way Conditional Jumps
- Limited Number of Registers

# LIR & Backend

## LIR

Produce a Generic and registerless assembly code

## Backend

Translation of the LIR into a correct ASM.

# Linearization: Principle

- *eseq* and *seq* must be eliminated (except the outermost *seq*).
- Similar to cut-elimination: permute inner *eseq* and *seq* to lift them higher, until they vanish.

# A simple rewriting system (1/7)

```
seq
  seq
    s1
    s2
  seq end
s3
seq end
```

$\rightsquigarrow$

```
seq
  s1
  s2
  s3
seq end
```

# A simple rewriting system (2/7)

## Generalization

```
seq s1
  seq s2 seq end
  s3
seq end
```

$\rightsquigarrow$

```
seq s1
  s2
  s3
seq end
```

## A simple rewriting system (3/7)

eseq

s1

eseq

s2

e

$\rightsquigarrow$

eseq

seq

s1

s2

seq end

e

## A simple rewriting system (4/7)

sxp

eseq

s1

e

$\rightsquigarrow$

seq

s1

sxp

e



# A simple rewriting system (5/7)

sxp

eseq

s1

e

$\rightsquigarrow$

seq

s1

sxp

e

# A simple rewriting system (6/7)

```
call
  f
  eseq s1 e
  es
call end
```

$\rightsquigarrow$

```
eseq
  s1
  call
    f
    e
    es
  call end
```

# A simple rewriting system (7/7)

binop

add

eseq s e1

e2

$\rightsquigarrow$

eseq

s

binop

add

e1

e2

# Incorrect changes!

⇒ This is incorrect!

binop

add

e1

eseq s e2

↔

eseq

s

binop

add

e1

e2

# High Level counterexample

```
let var t := 51
in
  t + (t := 42, 0)
end
```

$\rightsquigarrow$

```
let var t := 51
in
  (t := 42, t + 0)
end
```

# High Level Solution

⇒ Save values into temporaries

≈⇒

```
let var t := 51
    var t0 := t
in
    (t := 42, t0 + 0)
end
```

# Low level solution

binop

add

e1

eseq s e2

↗

eseq

seq

move temp t0 e1

s

seq end

binop

add

temp t0

e2

# Linearization: More Temporaries

When “de-expressioning” fresh temporaries are needed



# Naive approach

Save systematically every sub expression  
into temporaries!

⇒ This is extremely inefficient when not  
needed

# Exploit commutativity

Save useless extra temporaries and moves

## Problem

Commutativity cannot be known statically!

E.g., `move (mem (t1), e)` and `mem (t2)` commute iff  $t1 \neq t2$ .

## Error

There is an error in this example.

What is it?

# Conservative Approximation

Never say “commute” when they don’t !

“if `e` is a `const` then `s` and `e` definitely commute”.

# Summary

