Code Generation for Data Processing

Lecture 3: Intermediate Representations

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

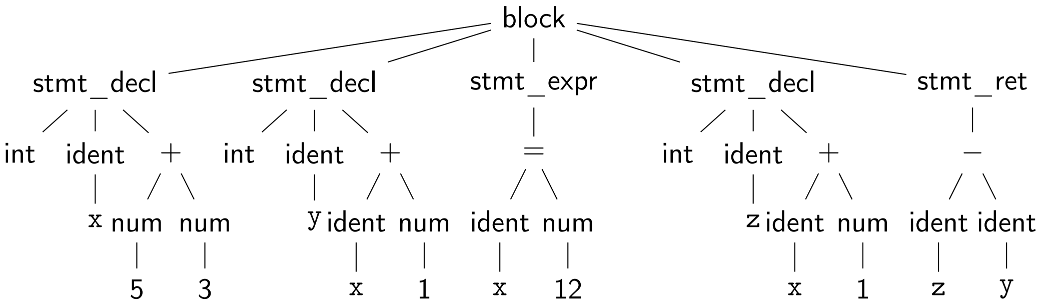
Intermediate Representations: Motivation

- So far: program parsed into AST

+ Great for language-related checks
+ Easy to correlate with original source code (e.g., errors)

– Hard for analyses/optimizations due to high complexity
    - variable names, control flow constructs, etc.
    - Data and control flow implicit
– Highly language-specific

Intermediate Representations: Motivation



Question: how to optimize? Is `x+1` redundant? $\rightsquigarrow$ hard to tell $\ddot{\frown}$

Intermediate Representations: Motivation

$$
\begin{array}{llllll}
x_1 & \leftarrow & 5 & + & 3 \\
y_1 & \leftarrow & x_1 & + & 1 \\
x_2 & \leftarrow & 12 & & \\
z_1 & \leftarrow & x_2 & + & 1 \\
tmp_1 & \leftarrow & z_1 & - & y_1 \\
\text{return} & & tmp_1 & &
\end{array}
$$

Question: how to optimize? Is `x+1` redundant? $\rightsquigarrow$ No! $\ddot\smile$

Intermediate Representations

- Definitive program representation inside compiler
  - During compilation, only the (current) IR is considered
- Goal: simplify analyses/transformations
  - *Technically*, single-step compilation is possible for, e.g., C
    … but optimizations are hard without proper IRs

- Compilers *design* IRs to support frequent operations
  - IR design can vary strongly between compilers
- Typically based on **graphs** or **linear instructions** (or both)

Compiler Design: Effect of Languages – Imperative

- Step-by-step execution of program modification of state
- Close to hardware execution model
- Direct influence of result

- Tracking of state is complex
- Dynamic typing: more complexity
- Limits optimization possibilities

```
void addvec(int* a, const int* b) {
  for (unsigned i = 0; i < 4; i++)
    a[i] += b[i]; // vectorizable?
}
```

```
func:
  mov [rdi], rsi
  mov [rdi+8], rdx
  mov [rdi], 0 // redundant?
  ret
```

Compiler Design: Effect of Languages – Declarative

- Describes execution target
- Compiler has to derive good mapping to imperative hardware

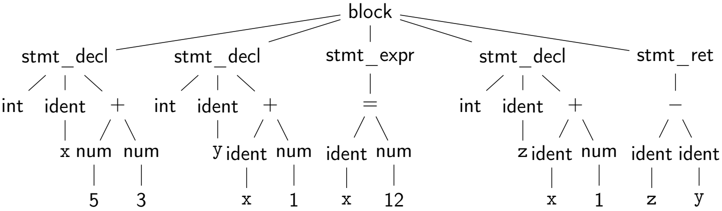
- Allows for more optimizations
- Mapping to hardware non-trivial
  - Might need more stages
  - Preserve semantic info for opt!
- Programmer has less "control"

```
select s.name
from studenten s
where exists (select 1
              from hoeren h
              where h.matrno=s.matrno)
```

```
let rec fac = function
    | 0 | 1 -> 1
    | n -> n * fac (n - 1)
```

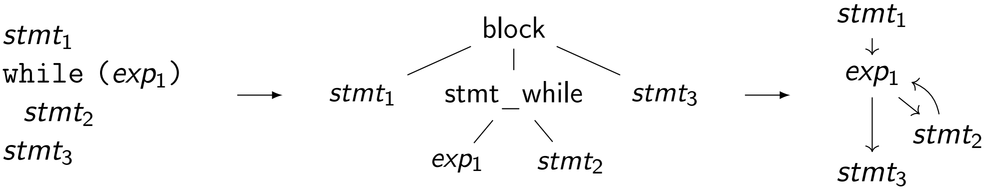
Graph IRs: Abstract Syntax Tree (AST)

- Code representation close to the source
- Representation of types, constants, etc. might differ
- Storage might be problematic for large inputs



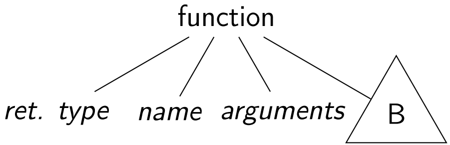
Graph IRs: Control Flow Graph (CFG)

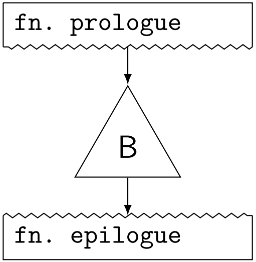
- Motivation: model control flow between different code sections
- Graph nodes represent **basic blocks**
  - Basic block: sequence of branch-free code (modulo exceptions)
  - Typically represented using a linear IR



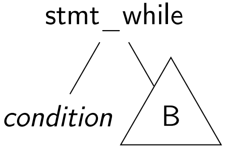
Build CFG from AST – Function

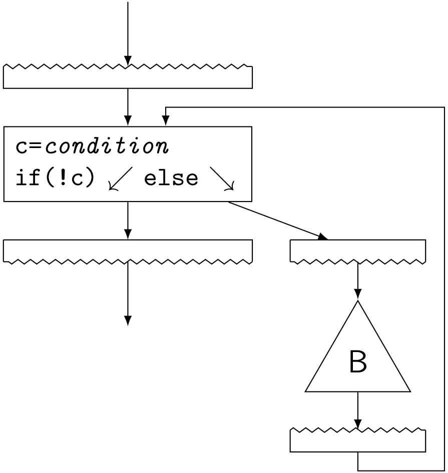
- Idea: Keep track of current insert block while walking through AST



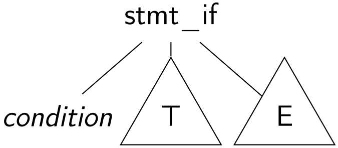


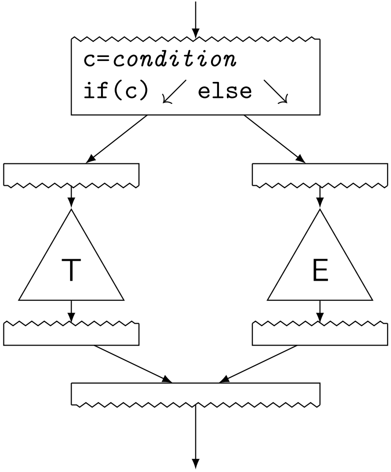
Build CFG from AST – While Loop





Build CFG from AST – If Condition





Build CFG from AST: Switch

Linear search

```
t ← exp
if t == 3: goto B3
if t == 4: goto B4
if t == 7: goto B7
if t == 9: goto B9
goto BD
```

+ Trivial
– Slow, lot of code

Binary search

```
t ← exp
if t == 7: goto B7
elif t > 7:
  if t == 9: goto B9
else:
  if t == 3: goto B3
  if t == 4: goto B4
goto BD
```

+ Good: sparse values
– Even more code

Jump table

```
t ← exp
if 0 ≤ t < 10:
  goto table[t]
goto BD

table = {
  BD, BD, BD, B3,
  B4, BD, ... }
```

+ Fastest
– Table can be large, needs ind. jump

Build CFG from AST: Break, Continue, Goto

- `break`/`continue`: trivial
  - Keep track of target block, insert branch
- `goto`: also trivial
  - Split block at target label, if needed
  - But: may lead to irreducible control flow graph (see later)

CFG: Formal Definition

- **Flow graph:** $G = (N, E, s)$ with a digraph (N, E) and entry $s \in N$
  - Each node is a basic block, s is the entry block
  - $(n_1, n_2) \in E$ iff n_2 might be executed immediately after n_1
  - All $n \in N$ shall be reachable from s (unreachable nodes can be discarded)
  - Nodes without successors are end points

CFG from C – Example

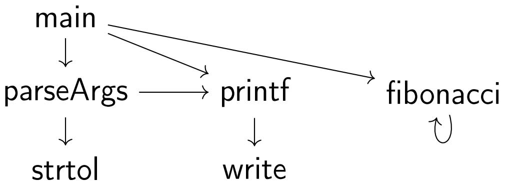
Derive the CFG for the these functions. Assume a `switch` instruction exists.

```
int fn1() {
  if (a()) {
    while (b()) {
      c();
      if (d())
        continue;
      e();
    }
  } else {
    f();
  }
}
```

```
int fn2() {
  a();
  do switch (c()) {
  case 1:
    while (d()) {
      e();
    case 2:
      f();
    }
  default:
    g();
  } while (h());
  return b();
}
```

Graph IRs: Call Graph

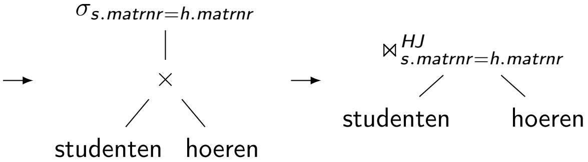
- Graph showing (possible) call relations between functions
- Useful for interprocedural optimizations
  - Function ordering
  - Stack depth estimation
  - ...



Graph IRs: Relational Algebra

- Higher-level representation of query plans
  - Explicit data flow
- Allow for optimization and selection actual implementations
  - Elimination of common sub-trees
  - Joins: ordering, implementation, etc.

```sql
SELECT s.name, h.vorlnr
FROM studenten s, hoeren h
WHERE s.matrnr = h.matrnr
```



Linear IRs: Stack Machines

- Operands stored on a stack
- Operations pop arguments from top and push result
- Typically accompanied with variable storage

- Generating IR from AST: trivial
- Often used for bytecode, e.g. Java, Python

\+ Compact code, easy to generate and implement

− Performance, hard to analyze

```
push 5
push 3
add
pop x
push x
push 1
add
pop y
push 12
pop x
push x
push 1
add
pop z
```

Linear IRs: Register Machines

- Operands stored in registers
- Operations read and write registers
- Typically: infinite number of registers
- Typically: three-address form
  - *dst = src1 op src2*

- Generating IR from AST: trivial
- E.g., GIMPLE, eBPF, Assembly

$$
\begin{array}{lllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
$$

Example: High GIMPLE

```c
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
int fac (int n)
gimple_bind < // <-- still has lexical scopes
  int D.1950;
  int res;

  gimple_assign <integer_cst, res, 1, NULL, NULL>
  gimple_goto <<D.1947>>
  gimple_label <<D.1948>>
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
  gimple_assign <plus_expr, n, n, -1, NULL>
  gimple_label <<D.1947>>
  gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
  gimple_label <<D.1946>>
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  gimple_return <D.1950>
>
```

```
$ gcc -fdump-tree-gimple-raw -c foo.c
```

Example: Low GIMPLE

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
int fac (int n)
{
  int res;
  int D.1950;

  gimple_assign <integer_cst, res, 1, NULL, NULL>
  gimple_goto <<D.1947>>
  gimple_label <<D.1948>>
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
  gimple_assign <plus_expr, n, n, -1, NULL>
  gimple_label <<D.1947>>
  gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
  gimple_label <<D.1946>>
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  gimple_goto <<D.1951>>
  gimple_label <<D.1951>>
  gimple_return <D.1950>
}
```

```
$ gcc -fdump-tree-lower-raw -c foo.c
```

Example: Low GIMPLE with CFG

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
int fac (int n) {
  int res;
  int D.1950;
  <bb 2> :
  gimple_assign <integer_cst, res, 1, NULL, NULL>
  goto <bb 4>; [INV]
  <bb 3> :
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
  gimple_assign <plus_expr, n, n, -1, NULL>
  <bb 4> :
  gimple_cond <ne_expr, n, 0, NULL, NULL>
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]
  <bb 5> :
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  <bb 6> :
gimple_label <<L3>>
  gimple_return <D.1950>
}
```

```
$ gcc -fdump-tree-cfg-raw -c foo.c
```

Linear IRs: Register Machines

- Problem: no clear def–use information
  - Is $x + 1$ the same?
  - Hard to track actual values!
- **How to optimize?**

$\Rightarrow$ Disallow mutations of variables

$$
\begin{array}{lllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
$$

Single Static Assignment: Introduction

- Idea: disallow mutations of variables, value set in declaration
- Instead: create new variable for updated value

- SSA form: every computed value has a unique definition
  - Equivalent formulation: each name describes result of one operation

$$
\begin{array}{llllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
\quad \longrightarrow \quad
\begin{array}{lllll}
v_1 & \leftarrow & 5 & + & 3 \\
v_2 & \leftarrow & v_1 & + & 1 \\
v_3 & \leftarrow & 12 & & \\
v_4 & \leftarrow & v_3 & + & 1 \\
v_5 & \leftarrow & v_4 & - & v_2 \\
\text{return} & v_5 & & &
\end{array}
$$

Single Static Assignment: Control Flow

- How to handle diverging values in control flow?
- Solution: Φ-nodes to merge values depending on predecessor
  - Value depends on edge used to enter the block
  - All Φ-nodes of a block execute concurrently (ordering irrelevant)

$$
\begin{array}{rlll}
\textit{entry}: & x & \leftarrow & \ldots \\
& \multicolumn{3}{l}{\text{if } (x > 2) \text{ goto } \textit{cont}} \\
\textit{then}: & x & \leftarrow & x \ * \ 2 \\
\textit{cont}: & \text{return} & x &
\end{array}
\quad \rightarrow \quad
\begin{array}{rlll}
\textit{entry}: & v_1 & \leftarrow & \ldots \\
& \multicolumn{3}{l}{\text{if } (v_1 > 2) \text{ goto } \textit{cont}} \\
\textit{then}: & v_2 & \leftarrow & v_1 \ * \ 2 \\
\textit{cont}: & v_3 & \leftarrow & \Phi(\textit{entry}: v_1, \textit{then}: v_2) \\
& \text{return} & v_3 &
\end{array}
$$

Example: GIMPLE in SSA form

```c
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
int fac (int n) { int res, D.1950, _1, _6;
  <bb 2> :
  gimple_assign <integer_cst, res_4, 1, NULL, NULL>
  goto <bb 4>; [INV]
  <bb 3> :
  gimple_assign <mult_expr, _1, n_2, n_2, NULL>
  gimple_assign <mult_expr, res_8, res_3, _1, NULL>
  gimple_assign <plus_expr, n_9, n_2, -1, NULL>
  <bb 4> :
  # gimple_phi <n_2, n_5(D)(2), n_9(3)>
  # gimple_phi <res_3, res_4(2), res_8(3)>
  gimple_cond <ne_expr, n_2, 0, NULL, NULL>
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]
  <bb 5> :
  gimple_assign <ssa_name, _6, res_3, NULL, NULL>
  <bb 6> :
gimple_label <<L3>>
  gimple_return <_6>
}
```

```
$ gcc -fdump-tree-ssa-raw -c foo.c
```

SSA Construction – Local Value Numbering

- Simple case: inside block – keep mapping of variable to value

Code					SSA IR			Variable Mapping		
x	$\leftarrow$	5	$+$	3	v_1	$\leftarrow$	add 5, 3	x	$\rightarrow$	v_3
y	$\leftarrow$	x	$+$	1	v_2	$\leftarrow$	add v_1, 1	y	$\rightarrow$	v_2
x	$\leftarrow$	12			v_3	$\leftarrow$	const 12	z	$\rightarrow$	v_4
z	$\leftarrow$	x	$+$	1	v_4	$\leftarrow$	add v_3, 1	tmp_1	$\rightarrow$	v_5
tmp_1	$\leftarrow$	z	$-$	y	v_5	$\leftarrow$	sub v_4, v_2			
return		tmp_1					ret v_5			

SSA Construction – Across Blocks

- SSA construction with control flow is non-trivial
- Key problem: find value for variable in predecessor

- Naive approach: Φ-nodes for all variables everywhere
  - Create empty Φ-nodes for variables, populate variable mapping
  - Fill blocks (as on last slide)
  - Fill Φ-nodes with last value of variable in predecessor

- Why is this a bad idea? $\Rightarrow$ *don't do this!*
  - *Extremely inefficient, code size explosion, many dead* Φ

SSA Construction – Across Blocks ("simple"[5])

- Key problem: find value in predecessor
- Idea: *seal* block once all direct predecessors are known
  - For acyclic constructs: trivial
  - For loops: seal header once loop block is generated

- Current block not sealed: add Φ-node, fill on sealing
- Single predecessor: recursively query that
- Multiple preds.: add Φ-node, fill now

[5] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122.

SSA Construction – Example

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

func *foo* (v_1)

entry: sealed; varmap: n$\to v_1$, res$\to v_2$

$v_2 \leftarrow 1$

header: sealed; varmap: n$\to \phi_1$, res$\to \phi_2$

$\phi_1 \leftarrow \phi(\text{entry: } v_1, \text{body: } v_6)$

$\phi_2 \leftarrow \phi(\text{entry: } v_2, \text{body: } v_5)$

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n$\to v_6$, res$\to v_5$

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ϕ_2, v_4

$v_6 \leftarrow$ sub ϕ_1, 1

br header

cont: sealed; varmap: res$\to \phi_2$

ret ϕ_2

SSA Construction – Example

Construct an IR in SSA form for the following C code.

```
int phis(int a, in b){
  a = a * b;
  if (a > b * b) {
    int c = 1;
    while (a > 0)
      a = a - c;
  } else {
    a = b * b;
  }
  return a;
}
```

SSA Construction – Pruned/Minimal Form

- Resulting SSA is *pruned* – all ϕ are used
- But not *minimal* – ϕ nodes might have single, unique value

- When filling ϕ, check that multiple real values exist
  - Otherwise: replace ϕ with the single value
  - On replacement, update all ϕ using this value, they might be trivial now, too
- Sufficient? Not for irreducible CFG
  - Needs more complex algorithms[6] or different construction method[7]

AD	IN2053 "Program Optimization" covers this more formally

[6] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122.

[7] R Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *TOPLAS* 13.4 (1991), pp. 451–490.

SSA: Implementation

- Value is often just a pointer to instruction
- ϕ nodes placed at beginning of block
  - They execute "concurrently" and on the edges, after all
- Variable number of operands required for ϕ nodes
- Storage format for instructions and basic blocks
  - Consecutive in memory: hard to modify/traverse
  - Array of pointers: $\mathcal{O}(n)$ for a single insertion...
  - Linked List: easy to insert, but pointer overhead

Is SSA a graph IR?

Only if instructions have no side effects,
consider `load`, `store`, `call`, ...

These *can* be solved using explicit dependencies as SSA values, e.g. for memory

Intermediate Representations – Summary

- An IR is an internal representation of a program
- Main goal: simplify analyses and transformations

- IRs typically based on graphs or linear instructions
- Graph IRs: AST, Control Flow Graph, Relational Algebra
- Linear IRs: stack machines, register machines, SSA

- Single Static Assignment makes data flow explicit
- SSA is extremely popular, although non-trivial to construct

Intermediate Representations – Questions

- Who designs an IR? What are design criteria?
- Why is an AST not suited for program optimization?
- How to convert an AST to another IR?
- What are the benefits/drawbacks of stack/register machines?
- What benefits does SSA offer over a normal register machine?
- How do ϕ-instructions differ from normal instructions?