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? \( \leadsto \) hard to tell 😞
Intermediate Representations: Motivation

\[
x_1 \leftarrow 5 + 3 \\
y_1 \leftarrow x_1 + 1 \\
x_2 \leftarrow 12 \\
z_1 \leftarrow x_2 + 1 \\
tmp_1 \leftarrow z_1 - y_1 \\
\text{return } tmp_1
\]

Question: how to optimize? Is \(x+1\) redundant? \(\leadsto\) No! 😊
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)
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

```
stmt_1
while (exp_1)  
  stmt_2
  stmt_3
```

```
stmt_1
  stmt_while
  stmt_2
  stmt_3
```

```
stmt_1
  exp_1
  stmt_2
  stmt_3
```
Build CFG from AST – Function

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

![Diagram of function structure with nodes for return type, name, arguments, function prologue, and function epilogue]
Build CFG from AST – While Loop

stmt\_while

\(c=\text{condition}\)

if(!\(c\))

else

\(\downarrow\)

B

\(\downarrow\)

\(\downarrow\)

B
Build CFG from AST – If Condition

\[
c = \text{condition} \\
\text{if}(c) \rightarrow \text{else} \\
\]

\[
\text{stmt}_\text{if} \\
\text{condition} \rightarrow \text{T} \rightarrow \text{E} \\
\]
Build CFG from AST: Switch

Linear search
\[
\begin{align*}
t & \leftarrow \text{exp} \\
\text{if } t &= 3: \text{goto } B_3 \\
\text{if } t &= 4: \text{goto } B_4 \\
\text{if } t &= 7: \text{goto } B_7 \\
\text{if } t &= 9: \text{goto } B_9 \\
goto B_D
\end{align*}
\]

- Trivial
- Slow, lot of code

Binary search
\[
\begin{align*}
t & \leftarrow \text{exp} \\
\text{if } t &= 7: \text{goto } B_7 \\
\text{elif } t &> 7: \\
\quad \text{if } t &= 9: \text{goto } B_9 \\
\quad \text{else:} \\
\quad \quad \text{if } t &= 3: \text{goto } B_3 \\
\quad \quad \text{if } t &= 4: \text{goto } B_4 \\
goto B_D
\end{align*}
\]

- Good: sparse values
- Even more code

Jump table
\[
\begin{align*}
t & \leftarrow \text{exp} \\
\text{if } 0 \leq t < 10: \\
\quad \text{goto table}[t] \\
goto B_D
\end{align*}
\]

- 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
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
Graph IRs: Call Graph

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

```
main
  ↓
parseArgs
  ↓
strtol
  ↓
write
  ↓
printf
  ↓
fibonacci
```

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.

\[
\text{SELECT } s.\text{name}, h.\text{vorlnr} \\
\text{FROM studenten } s, \text{hoeren } h \\
\text{WHERE } s.\text{matrnr} = h.\text{matrnr}
\]
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 x
add
pop y
push 12
pop x
push x
push x
add
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

```
x ← 5 + 3
y ← x + 1
x ← 12
z ← x + 1
tmp_1 ← z − y
return tmp_1
```
Example: High GIMPLE

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

```c
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

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

```c
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
```
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?

  ⇒ Disallow mutations of variables

\[
\begin{align*}
x & \leftarrow 5 + 3 \\
y & \leftarrow x + 1 \\
x & \leftarrow 12 \\
z & \leftarrow x + 1 \\
tmp_1 & \leftarrow z - y \\
\text{return} & \quad tmp_1
\end{align*}
\]
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

\[
\begin{align*}
x & \leftarrow 5 + 3 \\
y & \leftarrow x + 1 \\
x & \leftarrow 12 \\
z & \leftarrow x + 1 \\
tmp_1 & \leftarrow z - y \\
\text{return} & \quad tmp_1 \\
\end{align*}
\]

\[
\begin{align*}
v_1 & \leftarrow 5 + 3 \\
v_2 & \leftarrow v_1 + 1 \\
v_3 & \leftarrow 12 \\
v_4 & \leftarrow v_3 + 1 \\
v_5 & \leftarrow v_4 - v_2 \\
\text{return} & \quad v_5 \\
\end{align*}
\]
Single Static Assignment: Control Flow

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

\[
\begin{align*}
\text{entry} : & \quad x \leftarrow \ldots \\
& \quad \text{if } (x > 2) \text{ goto } \text{cont} \\
\text{then} : & \quad x \leftarrow x \ast 2 \\
\text{cont} : & \quad \text{return } x
\end{align*}
\]

\[
\begin{align*}
\text{entry} : & \quad v_1 \leftarrow \ldots \\
& \quad \text{if } (v_1 > 2) \text{ goto } \text{cont} \\
\text{then} : & \quad v_2 \leftarrow v_1 \ast 2 \\
\text{cont} : & \quad v_3 \leftarrow \Phi(entry : v_1, then : v_2) \\
& \quad \text{return } v_3
\end{align*}
\]
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>
}
```
SSA Construction – Local Value Numbering

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

<table>
<thead>
<tr>
<th>Code</th>
<th>SSA IR</th>
<th>Variable Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leftarrow 5 + 3 )</td>
<td>( v_1 \leftarrow \text{add } 5, 3 )</td>
<td>( x \rightarrow v_3 )</td>
</tr>
<tr>
<td>( y \leftarrow x + 1 )</td>
<td>( v_2 \leftarrow \text{add } v_1, 1 )</td>
<td>( y \rightarrow v_2 )</td>
</tr>
<tr>
<td>( x \leftarrow 12 )</td>
<td>( v_3 \leftarrow \text{const } 12 )</td>
<td>( z \rightarrow v_4 )</td>
</tr>
<tr>
<td>( z \leftarrow x + 1 )</td>
<td>( v_4 \leftarrow \text{add } v_3, 1 )</td>
<td>( tmp_1 \rightarrow v_5 )</td>
</tr>
<tr>
<td>( tmp_1 \leftarrow z - y )</td>
<td>( v_5 \leftarrow \text{sub } v_4, v_2 )</td>
<td></td>
</tr>
<tr>
<td>return ( tmp_1 )</td>
<td>( \text{ret } v_5 )</td>
<td></td>
</tr>
</tbody>
</table>
SSA construction with control flow is non-trivial

Key problem: find value for variable in predecessor

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

Why is this a bad idea?
- Extremely inefficient, code size explosion, many dead $\Phi$

$\implies$ don't do this!
SSA Construction – Across Blocks (“simple”⁴)

- 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

SSA Construction – Example

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

```plaintext
func foo(v₁)

entry: sealed; varmap: n→v₁, res→v₂
    v₂ ← 1

header: sealed; varmap: n→φ₁, res→φ₂
    φ₁ ← φ(entry: v₁, body: v₆)
    φ₂ ← φ(entry: v₂, body: v₅)
    v₃ ← equal φ₁, 0
    br v₃, cont, body

body: sealed; varmap: n→v₆, res→v₅
    v₄ ← mul φ₁, φ₁
    v₅ ← mul φ₂, v₄
    v₆ ← sub φ₁, 1
    br header

cont: sealed; varmap: res→φ₂
    ret φ₂
```
SSA Construction – Pruned/Minimal Form

▶ Resulting SSA is pruned – all $\phi$ are used
▶ But not minimal – $\phi$ nodes might have single, unique value

▶ When filling $\phi$, check that multiple real values exist
  ▶ Otherwise: replace $\phi$ with the single value
  ▶ On replacement, update all $\phi$ using this value, they might be trivial now, too
▶ Sufficient?  Not for irreducible CFG
  ▶ Needs more complex algorithms\(^5\) or different construction method\(^6\)

AD IN2053 “Program Optimization” covers this more formally


SSA: Implementation

- Value is often just a pointer to instruction
- $\phi$ nodes placed at beginning of block
  - They execute “concurrently” and on the edges, after all
- Variable number of operands required for $\phi$ nodes
- Storage format for instructions and basic blocks
  - Consecutive in memory: hard to modify/traverse
  - Array of pointers: $O(n)$ 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 $\phi$-instructions differ from normal instructions?