Outline

Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Frontend

Frontends: C, Java, Fortran

Intermediate Representation

Backends: IA-32, PowerPC, Alpha

- Checks correctness of source code wrt. a given language definition
- Transforms (valid) source into the intermediate representation
Intermediate Representation (IR)

- Compiler internal data structures representing a program
- *Uniform abstraction* from source languages and target architectures
  \[n + m\] compiler components instead of \[n \cdot m\] compilers
- *Optimizations* are performed on the IR
Backend

- Encapsulates all details of a target architecture

  - Code generation
    - Instruction selection
    - Instruction scheduling
    - Register allocation
Outline

Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Outline

Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Motivating IRs

- Bridge the gap between abstract syntax tree and object code
- Provide data structures more suitable for analyses/optimizations
- Easier retargetability (reuse of IR for source-target pairs)
- Reuse of machine independent optimizations
Overview

Intermediate Representations
   Why?
   How?
   IR Concepts

Static Single Assignment Form
   Introduction
   Theory
   SSA Construction
Design Issues

- Consider source language and target
- Consider (type) of planned optimizations
- Choose the right “level”
  - Higher level means closer to source
  - Lower level closer to target loses some structure/information
- Procedure cloning, inlining, and loop optimizations need structural high-level information
- Branch optimization, software pipelining, and register allocation need representation close to machine

⇒ Possibly multiple levels in one IR (same generic data structures). So called “lowering” transforms them from high to low.
Lowering

Typical constructs subject to lowering

- array accesses
- struct accesses
- calls (calling convention, ABI)
- instruction selection can be seen as lowering

\[
\begin{align*}
t_1 & := a[i, j+2] \\
t_2 & := j+2 \\
t_3 & := 10 \times i \\
t_4 & := t_1 + t_2 \\
t_5 & := 4 \times t_3 \\
t_6 & := \text{addr}(a) \\
t_7 & := t_4 + t_5 \\
t_8 & := \ast t_6
\end{align*}
\]
Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Different IR Concepts

Representation of control flow
- Control-flow graph (CFG)
- Basic Block Graph (BBG)

Representation of computation
- Triple code
- Expression trees
- Data dependence graphs
Control Flow Graph (CFG)

Definition
In a CFG there is 1:1 correspondence of nodes to statements/instructions. Edges represent possible control flow.
Basic Block Graph (BBG)

**Definition**
A basic block (BB) is a maximal sequence of statements/instructions such that if any is executed all are executed.

**Definition**
In a BBG nodes are BBs and control flow is represented only between basic blocks.
Inside a BB there are no control dependencies.

Remark: Most people call this CFG.
Triple Code and Expression Trees

Representation of computation/data flow.
What is inside the BBs?

- **Triple code**: List of elementary instructions
  \( x = \text{op} \ a \ b \)

- **Expression trees**: List of trees
  \( x = a + b \ast c; \ y = \text{call} \ \text{foo} \ (3 \ast x) ; \)
Data Dependence Graphs

- Nodes represent computation results (operators)
- Edges represent data dependencies (data flow)
- Problem with concept of variables (state)
- No problem with side-effect-free operators (functional programming)
- Suitable representation for SSA form
Outline

Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Outline

Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Motivation

Main goal:
- Make data-flow analyses more efficient
- Make optimizations more effective

Nice “side-effects”:  
- Some analyses/optimizations happen implicitly for free
- SSA-construction can implicitly perform CSE
- Use-Def chains are explicit in representation
- Def-Use chains are cheaper to represent
Static Single Assignment is a property of an IR regarding variables.

Definition
A program is in SSA form if every variable is statically assigned at most once. I.e. there are no two program locations assigning the same variable.
Intuition Behind Construction

- Replace concept of variable by concept of abstract values
- The entity statically referred to is a value
- For each assignment to a variable \( v \) a new abstract value \( v_i \) is defined. \( v \) is replaced by \( v_1, v_2, \ldots \)
- For each use of \( v \) the definition \( v_i \) valid at that location is used instead
Merge Problem and Phi-Functions

- Problem: What to do when control flow merges?
- Here: Which $c$ to use at the return?

non-SSA

\[(a, b) = \text{start}\]

**if** $b < a$

$\begin{cases} c := a - b \\ c := 0 \end{cases}$

return $c$
Merge Problem and Phi-Functions

- Problem: What to do when control flow merges?
- Here: Which $c$ to use at the return?
- Solution: Introduce pseudo operation, $\phi$-functions
- $\phi$s select the correct value dependent on control flow

```
non-SSA
(a, b) = start

if b < a
    c := a - b
    c := 0

return c

SSA
(a, b) = start

if b < a
    c1 := a - b
    c2 := 0
    c3 := \phi(c1, c2)

return c3
```
Outline

Overview

Intermediate Representations
  Why?
  How?
  IR Concepts

Static Single Assignment Form
  Introduction
  Theory
  SSA Construction
Phi-Functions

- \( \phi \)s have as many operands as the corresponding BB has predecessors
- Each operand is uniquely associated with one of these predecessors
- The result of a \( \phi \) is the operand associated to the predecessor through which the BB was reached

- \( \phi \)s always are the first “instructions” in a BB
- all \( \phi \)s in a BB must be evaluated simultaneously
Why Simultaneously? Swap Example

\[
\begin{align*}
a &= 23 \\
b &= 42
\end{align*}
\]

\[
\begin{align*}
t &= a \\
a &= b \\
b &= t \\
call &\quad printf, str, a, b
\end{align*}
\]
Why Simultaneously? Swap Example

```
\[
\begin{align*}
  a &= 23 \\
  b &= 42 \\
  t &= a \\
  a &= b \\
  b &= t \\
  \text{call} &\quad \text{printf, str, } a, b
\end{align*}
\]
```

```
\[
\begin{align*}
  a_1 &= 23 \\
  b_1 &= 42 \\
  t &= \phi(a_1, a_2) \\
  a_2 &= \phi(b_1, b_2) \\
  b_2 &= t \\
  \text{call} &\quad \text{printf, str, } a_2, b_2
\end{align*}
\]
```
Why Simultaneously? Swap Example

\[
\begin{align*}
    a &= 23 \\
    b &= 42 \\
    t &= a \\
    a &= b \\
    b &= t \\
    \text{call } &\text{ printf, str, } a, b
\end{align*}
\]

\[
\begin{align*}
    a_1 &= 23 \\
    b_1 &= 42 \\
    t &= \phi(a_1, a_2) \\
    a_2 &= \phi(b_1, t) \\
    \text{call } &\text{ printf, str, } a_2, t
\end{align*}
\]
Dominance

Given a CFG with basic blocks X, Y, Z, and S, where S is the start block.

- Dominance: \( X \geq Y \)
  Each path from S to Y goes through X

- Strict dominance: \( X > Y \)
  \( X > Y \) if \( X \geq Y \land X \neq Y \)

- Dominance is a tree order

- Immediate dominator: \( \text{idom}(X) \)
  \( X = \text{idom}(Y) \) if \( X > Y \land \forall Z : X > Z > Y \)
A CFG is in SSA form iff

- every variable has exactly one program point where it is defined
- for every use of a variable \( x \)
  \[
  \ell : \cdots \leftarrow \tau(\cdots, x, \cdots)
  \]
  the definition of \( x \) either
  - dominates \( \ell \) if \( \tau \neq \phi \)
  - dominates the \( i \)-th predecessor of \( \ell \) if \( \tau = \phi \) and \( x \) is the \( i \)-th argument
(Iterated) Join Points

- Consider two paths \( p : p_1, \ldots, p_n \), \( q : q_1, \ldots q_m \) of nodes in the CFG.
- Say \( p \) and \( q \) converge at \( z \) if
  \[
  \exists k \leq n, l \leq m. (p_k = q_l = z) \land (\forall 1 \leq i < k, 1 \leq j < l. p_i \neq q_j)
  \]
- Let \( \mathcal{J}(x, y) \) be the set of convergence/join points of \( x \) and \( y \):
  \[
  \mathcal{J}(x, y) := \{ z \mid \exists p. x \rightarrow^+ z, q : y \rightarrow^+ z. p, q \text{ converge at } z \}
  \]
- \( \mathcal{J}(x, y) \) can be extended to sets of nodes:
  \[
  \mathcal{J}(\{x_1, \ldots, x_n\}) := \bigcup_{1 \leq i < j \leq n} \mathcal{J}(x_i, x_j)
  \]
- When putting a program to SSA form, \( \phi \)-functions have to be inserted for a variable \( v \) at all \( \mathcal{J}(\text{defs}(v)) \).
- But \( \phi \)-functions constitute new definitions of SSA variables related to \( v \).
- Hence \( \mathcal{J} \) needs to be iterated:
  \[
  \mathcal{J}^1(X) := \mathcal{J}(X)
  
  \mathcal{J}^{i+1}(X) := \mathcal{J}(\mathcal{J}^i(X) \cup X)
  
  \mathcal{J}^+ := \mathcal{J}^n \text{ for } n > 1 \text{ and } \mathcal{J}^n = \mathcal{J}^{n+1}
  \]
Placement of Phi-Functions

Theorem ($\phi$ placement)

Given a non-SSA CFG and a variable $x$. Let $\text{defs}(x)$ be the set of program points where $x$ is defined. A correct SSA construction algorithm has to place a $\phi$ for $x$ at all program points in

$$\mathcal{J}^+(\text{defs}(x)) \cap \text{live}(x)$$

Proof sketch:

- Let $X$ and $Y$ contain definitions of $v$ and $Z$ be a join point of two paths $X \rightarrow^+ Z$ and $Y \rightarrow^+ Z$
- $\phi$ can not be placed before $Z$
- $\phi$ must not be placed after $Z$, e.g. in $Z'$ with $Z \rightarrow^+ Z'$
  Disambiguation of paths in a $Z'$ would be impossible
- Iterated join points are necessary, since inserted $\phi$s are new definitions of the variable
Outline

Overview

Intermediate Representations
    Why?
    How?
    IR Concepts

Static Single Assignment Form
    Introduction
    Theory
    SSA Construction
In the worst case each BB has a \( \phi \) for each variable.
- complexity \( O(n^2) \)
- linear in practice

Join criterion only says where to place \( \phi \)s. What are the correct arguments?

Idea by Click 1995:
- don’t compute join sets explicitly
- perform global value numbering during construction
- place \( \phi \)s on the fly
Value Numbering

- Find congruent variables
- Reuse instead of recomputation
- Two computations are congruent if
  - identical operators w/o side-effects (includes constants)
  - congruent operands
- Normalize expressions. More congruence detectable.
- In $c = a + 1$ and $d = 1 + b$
  $c$ and $d$ are congruent if $a$ and $b$ are congruent
SSA Construction with VN (1)

Starting point:
- AST or BBG
- w.l.o.g. computations are in form $x = \tau(y, z)$

Proceeding:
- in each BB store valid value number $\text{VN}(\tau, y, z)$ for each variable
  - store value number: $\text{setVN}(x, vn)$
  - get value number: $\text{getVN}(x)$
- $\text{getVN}(x)$ possibly inserts $\phi$s if VN not defined in current BB

Nice:
- $\phi$s are only inserted if variable is live
SSA Construction with VN (2)

For each $x = \tau(y, z)$ do:

- getVN($y$), getVN($z$)
- compute VN($\tau, y, z$)
- if value number is new insert
  VN($\tau, y, z$) = $\hat{\tau}$(getVN($y$), getVN($z$)) into the basic block
- store value number of $x$: setVN($x$, VN($\tau, y, z$))

Nice:

- computation of VN implicitly performs CSE
Details of getVN(ν):

- if value νᵢ is valid for variable ν in current BB return νᵢ
- else if BB has exactly one predecessor call getVN(ν) there
- else (more predecessors):
  - call getVN(ν) for all predecessors
  - let the values ν₁, ν₂, ... be the results
  - insert VN(φ, ν, ν) = φ(ν₁, ν₂, ...) into BB
  - avoid unnecessary φs
  - store new value of ν: setVN(ν, VN(φ, ν, ν))
  - return this new value
Unknown Predecessors: Problem

Observation: getVN might be undefined for some predecessors (loops!)
Solution: Two-stage approach

- mark a BB as ready when it is in SSA form
- if all predecessors are ready proceed as described
- else insert $\phi'$ and remember operand for finishing later
- when marking a BB as ready check successors and possibly finish them
Unknown Predecessors: Example
Consequence: Do construction in control-flow order (as much as possible)

- Use post-order of a reverse depth-first search
- keeps number of $\phi$’s low
- dominating BBs are constructed before dominated BBs
- this makes the implicit CSE more effective
Larger Example

(1) \( a := 1; \)
(2) \( b := 2; \)
   while (true) {
(3) \( c := a + b; \)
(4) \( \text{if } (d := c - a) \)
(5) \( \quad \text{while } (d := b * d) \{
(6) \quad d := a + b;
(7) \quad e := e + 1;
\}
(8) b := a + b;
(9) \text{if } (e := c - a)
   \quad \text{break; }
\}
(10) a := b * d;
(11) b := a - d;
Get value number for a first places $\phi'$ for a ...

SSA Construction Block 2
…then for \( b \) …
...and eventually a VN for \( c \).
Inserting $d := c - a$ works like normal value numbering.
Call to getVN(a) in 4 lead to recursive call getVN(a) in 3. This in turn produces a $\phi'$ for a in 3.
All predecessors of 3 are now in SSA form: $\phi$'s are placed. In block 2 a $\phi'$ is recursively placed for $e$. 
getVN(a) in 5 recognizes copies, finds unique definition: no $\phi$ is necessary
SSA Construction Block 5

GB₁
\[ a₁ := 1 \]
\[ b₁ := 2 \]

GB₂
\[ a₂ := \phi'(a) \]
\[ b₂ := \phi'(b) \]
\[ e₂ := \phi'(e) \]
\[ c₁ := a₂ + b₂ \]
\[ d₁ := c₁ - a₂ \]

GB₃
\[ b₃ := b₂ \]
\[ d₂ := \phi(d₁, d₄) \]
\[ a₃ := a₂ \]
\[ e₃ := \phi(e₂, e₄) \]
\[ d₃ := b₃ * d₂ \]

GB₄
\[ d₄ := a₃ + b₃ \]
\[ e₄ := e₃ + 1 \]

GB₅
\[ b₅ := a₂ + b₂ \]

GB₆

\[ d := b * d \]
\[ c := a + b \]
\[ d := a + b \]
\[ b := a + b \]
\[ e := e + 1 \]
\[ e := c - a \]
\[ a := b * d \]
\[ b := a - d \]
All predecessors of 2 are now in SSA form: $\phi$'s are placed.

Algorithm recognizes: $e$ is uninitialized! Insert undefined value $e_1$
SSA Construction Block 6

Recursive call to getVN(d) in 5 places complete \( \phi \) function \( d_5 \)
SSA Construction Block 6

GB₁
a₁ := 1
b₁ := 2

GB₂
a₂ := a₁
b₂ := φ(b₁, b₅)
e₂ := φ(e₁, e₅)
c₁ := a₂ + b₂
d₁ := c₁ - a₂

GB₃
b₃ := b₂
d₂ := φ(d₁, d₄)
a₃ := a₂
e₃ := φ(e₂, e₄)
d₃ := b₃ * d₂

GB₄
d₄ := a₃ + b₃
e₄ := e₃ + 1

GB₅
d₅ := φ(d₃, d₁)
b₅ := a₂ + b₂
e₅ := c₁ - a₂

GB₆
a₄ := b₅ * d₅
b₆ := a₄ - d₅
Optimization: Copy Propagation

a_1 := 1
b_1 := 2

\[ a_4 := b_5 \cdot d_5 \]
\[ b_6 := a_4 - d_5 \]
Optimization: Constant Propagation

GB₁
\[
a_1 := 1 \\
b_1 := 2
\]

GB₂
\[
a_2 := 1 \\
b_2 := \phi(2, b_5) \\
e_2 := \phi(e_1, e_5) \\
c_1 := 1 + b_2 \\
d_1 := c_1 - 1
\]

GB₃
\[
b_3 := 2 \\
d_2 := \phi(d_1, d_4) \\
a_3 := 1 \\
e_3 := \phi(e_2, e_4) \\
d_3 := b_2 * d_2
\]

GB₄
\[
d_4 := 1 + b_2 \\
e_4 := e_3 + 1
\]

GB₅
\[
d_5 := \phi(d_3, d_1) \\
b_5 := 1 + b_2 \\
e_5 := c_1 - 1
\]

GB₆
\[
a_4 := b_5 * d_5 \\
b_6 := a_4 - d_5
\]

a := 1 \\
b := 2 \\
c := a + b \\
d := c - a \\
d := b * d \\
b := a + b \\
e := e + 1 \\
e := c - a \\
a := b * d \\
b := a - d
Optimization: Dead Code Elimination

\[
\begin{align*}
    d_2 & := \phi(d_1, d_4) \\
    e_3 & := \phi(e_2, e_4) \\
    d_3 & := b_2 \cdot d_2 \\
    d_4 & := 1 + b_2 \\
    e_4 & := e_3 + 1 \\
    b_5 & := 1 + b_2 \\
    e_5 & := c_1 - 1 \\
    a_4 & := b_5 \cdot d_5 \\
    b_6 & := a_4 - d_5
\end{align*}
\]
Further Optimizations

- common subexpressions
- reassociation
- evaluation of constant expressions
- copy propagation
- dead code elimination

```plaintext
a4 := c1 * d5
b6 := a4 - d5

a := 1
b := 2
c := a + b
d := c - a
d := b * d
e := e + 1
```

```
GB1

GB2

GB3

d2 := \phi(b2, c1)
e3 := \phi(e2, e4)
d3 := b2 * d2

GB4

e4 := e3 + 1

GB5

d5 := \phi(d3, b2)

GB6

a := 1
b := 2
c := a + b
d := c - a
d := b * d
e := e + 1
```
1. S. Muchnick: Advanced Compiler Design and Implementation (On IR issues and SSA)
2. C. Click et al.: His papers from 1995. Confer to DBLP (On practical SSA construction and an SSA-IR proposal)
3. R. Cytron et al.: An efficient method of computing SSA form (Original work on SSA. POPL 1989, similar article in TOPLAS 1991)
4. www.libfirm.org (optimizing graph-based SSA IR)