5. Intermediate Representation

Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes. 
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

SSA lecture notes by Marcus Denker
Roadmap

> Intermediate representations
> Static Single Assignment
> SSA generation
> Dominance and SSA generation
> Applications of SSA
> $\Phi$-congruence and SSA removal

Roadmap

- Intermediate representations
- Static Single Assignment
- SSA generation
- Dominance and SSA generation
- Applications of SSA
- $\Phi$-congruence and SSA removal
Why use intermediate representations?

1. Software engineering principle
   — break compiler into manageable pieces
2. Simplifies retargeting to new host
   — isolates back end from front end
3. Simplifies support for multiple languages
   — different languages can share IR and back end
4. Enables machine-independent optimization
   — general techniques, multiple passes
IR scheme

- front end produces IR
- optimizer transforms IR to more efficient program
- back end transforms IR to target code
Kinds of IR

> Abstract syntax trees (AST)
> Linear operator form of tree (e.g., postfix notation)
> Directed acyclic graphs (DAG)
> Control flow graphs (CFG)
> Program dependence graphs (PDG)
> Static single assignment form (SSA)
> 3-address code
> Hybrid combinations
Categories of IR

> **Structural**
  — graphically oriented (trees, DAGs)
  — nodes and edges tend to be large
  — heavily used on source-to-source translators

> **Linear**
  — pseudo-code for abstract machine
  — large variation in level of abstraction
  — simple, compact data structures
  — easier to rearrange

> **Hybrid**
  — combination of graphs and linear code (e.g. CFGs)
  — attempt to achieve best of both worlds
ASTs, linear postfix, and DAGs are all just different ways to represent the syntactic structure of the program, i.e., the parse tree. CFGs and PDGs capture more of the semantics of the program, namely control flow and data dependencies. SSA and 3-address codes are even more detailed, expressing an abstract representation of the code to be generated.
Important IR properties

> Ease of generation
> Ease of manipulation
> Cost of manipulation
> Level of abstraction
> Freedom of expression (!)
> Size of typical procedure
> Original or derivative

Subtle design decisions in the IR can have far-reaching effects on the speed and effectiveness of the compiler!

Degree of exposed detail can be crucial
Abstract syntax tree

An AST is a parse tree with nodes for most non-terminals removed.

Since the program is already parsed, non-terminals needed to establish precedence and associativity can be collapsed!

A linear operator form of this tree (postfix) would be:

```
x 2 y * –
```
Reminder: Concrete syntax trees, or parse trees, show all of the intermediate non-terminals needed to produce an unambiguous grammar. An abstract syntax tree collapses these to offer a much simpler version of the parse tree.
A DAG is an AST with unique, shared nodes for each value.

\[
x := 2 \times y + \sin(2\times x) \\
z := \frac{x}{2}
\]
Control flow graph

- A CFG models *transfer of control* in a program
  - nodes are *basic blocks* (straight-line blocks of code)
  - edges represent *control flow* (loops, if/else, goto …)

```plaintext
if x = y then
  S1
else
  S2
end
S3
```
A basic block is a sequence of straightline code, i.e., without any branching.
3-address code

> Statements take the form: \( x = y \ op \ z \)
  — single operator and at most three names

\[
\begin{align*}
x - 2 \times y & \\
t1 & = 2 \times y \\
t2 & = x - t1
\end{align*}
\]

> Advantages:
  — compact form
  — names for intermediate values
## Typical 3-address codes

<table>
<thead>
<tr>
<th>Assignments</th>
<th>x = y op z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = op y</td>
</tr>
<tr>
<td></td>
<td>x = y[i]</td>
</tr>
<tr>
<td></td>
<td>x = y</td>
</tr>
<tr>
<td>Branches</td>
<td>goto L</td>
</tr>
<tr>
<td>Conditional branches</td>
<td>if x relop y goto L</td>
</tr>
<tr>
<td>Procedure calls</td>
<td>param x</td>
</tr>
<tr>
<td></td>
<td>param y</td>
</tr>
<tr>
<td></td>
<td>call p</td>
</tr>
<tr>
<td>Address and pointer assignments</td>
<td>x = &amp;y</td>
</tr>
<tr>
<td></td>
<td>*y = z</td>
</tr>
</tbody>
</table>
3-address code — two variants

**Quadruples**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>y</td>
</tr>
<tr>
<td>(1)</td>
<td>load</td>
<td>t1</td>
</tr>
<tr>
<td>(2)</td>
<td>load</td>
<td>t2</td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>t3</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>t4</td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>t5</td>
</tr>
</tbody>
</table>

**Triples**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>y</td>
</tr>
<tr>
<td>(1)</td>
<td>load</td>
<td>y</td>
</tr>
<tr>
<td>(2)</td>
<td>load</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>(1)</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>x</td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>(4)</td>
</tr>
</tbody>
</table>

- simple record structure
- easy to reorder
- explicit names

- table index is implicit name
- only 3 fields
- harder to reorder
As we shall see later (next lecture), optimizations may need to reorder operations to minimize shunting of values in and out of registers. For this reason it is good to have an IR that makes reordering easy.
IR choices

> Other hybrids exist
  — combinations of graphs and linear codes
  — CFG with 3-address code for basic blocks

> Many variants used in practice
  — no widespread agreement
  — compilers may need several different IRs!

> Advice:
  — choose IR with right level of detail
  — keep manipulation costs in mind
Roadmap

- Intermediate representations
- Static Single Assignment
- SSA generation
- Dominance and SSA generation
- Applications of SSA
- $\Phi$-congruence and SSA removal
SSA: Literature

Books:
- SSA Chapter in Appel
- Chapter 8.11 Muchnik

SSA Creation:
Cytron et. al: *Efficiently computing Static Single Assignment Form and the Control Dependency Graph* (TOPLAS, Oct 1991)

> **Goal**: simplify procedure-global optimizations

> **Definition**: Program is in SSA form if every variable is only assigned once
Static Single Assignment (SSA)

> Each assignment to a temporary is given a unique name
  — All uses reached by that assignment are renamed
  — Compact representation
  — Useful for many kinds of compiler optimization …

\[
\begin{align*}
  x & := 3; \\
  x & := x + 1; \\
  x & := 7; \\
  x & := x*2;
\end{align*}
\]

\[
\begin{align*}
  x_1 & := 3; \\
  x_2 & := x_1 + 1; \\
  x_3 & := 7; \\
  x_4 & := x_3*2;
\end{align*}
\]


http://en.wikipedia.org/wiki/Static_single_assignment_form
Converting to SSA is easy: simply introduce a new name each time a variable is assigned to. Here we assign to x four times, so we introduce \( x_1 \) through \( x_4 \). Note that we can now see clearly that the value of \( x_2 \) is never used.
Why Static?

> Why Static?
  — *We only look at the static program*
  — *One assignment per variable in the program*

> At runtime variables are assigned multiple times!
SSA is only used for static analysis and optimization. It will disappear when we generate the final executable code.
Example: Sequence

Easy to do for sequential programs:

Original

\[
\begin{align*}
    a & := b + c \\
    b & := c + 1 \\
    d & := b + c \\
    a & := a + 1 \\
    e & := a + b
\end{align*}
\]

SSA

\[
\begin{align*}
    a_1 & := b_1 + c_1 \\
    b_2 & := c_1 + 1 \\
    d_1 & := b_2 + c_1 \\
    a_2 & := a_1 + 1 \\
    e_1 & := a_2 + b_2
\end{align*}
\]
SSA form makes clear that $a_1$ is not the same as $a_2$, which is easier for analysis.
Example: Condition

Conditions: what to do on control-flow merge?

**Original**

```plaintext
if B then
    a := b
else
    a := c
end
... a ...
```

**SSA**

```plaintext
if B then
    a_1 := b
else
    a_2 := c
end
... a? ...
```
After the if-then-else we have a problem. What is the value of a? Is it $a_1$ or $a_2$? It could be either one.
Solution: Φ-Function

Conditions: what to do on control-flow merge?

Original

```plaintext
if B then
  a := b
else
  a := c
end
... a ...
```

SSA

```plaintext
if B then
  a₁ := b
else
  a₂ := c
end
a₃ := Φ(a₁,a₂)
... a₃ ...
```
The trick is to introduce a new variable $a_3$ whose value is assigned by a “magical” function $\Phi$ that chooses between $a_1$ and $a_2$.

Of course $\Phi$ does not really exist at run time, but we pretend it does to support static analysis. $\Phi$ will disappear by the time we generate actual code.
The $\Phi$-Function

- $\Phi$-functions are always at the beginning of a basic block
- Selects between values depending on control-flow
- $a_{k+1} := \Phi(a_1 \ldots a_k)$: the block has $k$ preceding blocks

$\Phi$-functions are evaluated simultaneously within a basic block.
If there are multiple $\Phi$ functions (for multiple variables) then they are “evaluated” simultaneously. (In any case they will disappear later.)
SSA and CFG

- SSA is normally used for control-flow graphs (CFG)

- Basic blocks are in 3-address form
Recall: Control flow graph

> A CFG models \textit{transfer of control} in a program
  — nodes are \textit{basic blocks} (straight-line blocks of code)
  — edges represent \textit{control flow} (loops, if/else, goto …)

\begin{verbatim}
if x = y then
  S1
else
  S2
end
S3
\end{verbatim}
SSA: a Simple Example

```plaintext
if B then
    a1 := 1
else
    a2 := 2
end
a3 := \Phi(a1,a2)
... a3 ...
```
Roadmap

- Intermediate representations
- Static Single Assignment
- **SSA generation**
- Dominance and SSA generation
- Applications of SSA
- $\Phi$-congruence and SSA removal
Recall: IR

- front end produces IR
- optimizer transforms IR to more efficient program
- back end transforms IR to target code
SSA as IR
Current trend in compiler community is to use SSA as the IR for everything in back end.
Transforming to SSA

> **Problem: Performance / Memory**
  — Minimize number of inserted $\Phi$-functions
  — Do not spend too much time

> **Many relatively complex algorithms**
  — We do not go too much into detail
  — See literature!
Minimal SSA

> Two steps:
  — Place $\Phi$-functions
  — Rename Variables

> Where to place $\Phi$-functions?

> We want minimal amount of needed $\Phi$
  — Save memory
  — Algorithms will work faster
Path Convergence Criterion

- There should be a $\Phi$ for $a$ at node $Z$ if:
  1. There is a block $X$ containing a definition of $a$
  2. There is a block $Y$ ($Y \neq X$) containing a definition of $a$
  3. There is a nonempty path $P_{xz}$ of edges from $X$ to $Z$
  4. There is a nonempty path $P_{yz}$ of edges from $Y$ to $Z$
  5. Path $P_{xz}$ and $P_{yz}$ do not have any nodes in common other than $Z$
  6. The node $Z$ does not appear within both $P_{xz}$ and $P_{yz}$ prior to the end (although it may appear in one or the other)

- I.e., $Z$ is the first place where two definitions of $a$ collide
NB: Here “definition” means *assignment*. 
Iterated Path-Convergence

> Inserted $\Phi$ is itself a definition!

while there are nodes $X, Y, Z$ satisfying conditions 1–5
and $Z$ does not contain a $\Phi$-function for a

do

insert $\Phi$ at node $Z$.

A bit slow, other algorithms used in practice
1. block X contains a definition of a
2. block Y (Y ≠ X) contains a definition of a
3. path $P_{xz}$ of edges from X to Z.
4. path $P_{yz}$ of edges from Y to Z.
5. path $P_{xz}$ and $P_{yz}$ do not have any nodes in common other than Z
6. node Z does not appear within both $P_{xz}$ and $P_{yz}$ prior to the end
Roadmap

> Intermediate representations
> Static Single Assignment
> SSA generation
> Dominance and SSA generation
> Applications of SSA
> \( \Phi \)-congruence and SSA removal
Dominance Property of SSA

> Dominance: node \( D \) \textit{dominates} node \( N \) if every path from the start node to \( N \) goes through \( D \).

(“strictly dominates”: \( D \neq N \))

\textbf{Dominance Property of SSA:}

1. If \( x \) is used in a \( \Phi \)-function in block \( N \), then the node defining \( x \) dominates every predecessor of \( N \).
2. If \( x \) is used in a non-\( \Phi \) statement in \( N \), then the node defining \( x \) dominates \( N \)

\textit{“Definition dominates use”}
If x is used in a Φ-function in node N, then there exists another path to N (i.e., via the other nodes that contribute to Φ), but not to its predecessors.

Dominance is a property of basic blocks: one node (basic block) dominates a set of nodes. For the dominance property, “definition of x” thus means the basic block in which x is defined (assigned).
Dominance can be used to efficiently build SSA

\( \Phi \)-Functions are placed in all basic blocks of the *Dominance Frontier*

\[ \text{DF}(D) = \text{the set of all nodes } N \text{ such that } D \text{ dominates an immediate predecessor of } N \text{ but does not strictly dominate } N. \]
Dominance frontier
Node 5 dominates all nodes in the gray area
Follow edges leaving the region dominated by node 5 to the region not \textit{strictly dominated by 5}.

\[ \text{DF}(5) = \{ 4, 5, 12, 13 \} \]
Simple Example

DF(B1) =
DF(B2) =
DF(B3) =
DF(B4) =
Simple Example

DF(B1) = {?}
DF(B2) =
DF(B3) =
DF(B4) =
Simple Example

DF(B1) = {}
DF(B2) =
DF(B3) =
DF(B4) =

B1 dominates all nodes, so its DF is empty.
Simple Example

DF(B1) = {}
DF(B2) = {?} 
DF(B3) =  
DF(B4) =
Simple Example

DF(B1) = {}
DF(B2) = {B4}
DF(B3) =
DF(B4) =

B2 only dominates itself, so its DF is \{ B4 \}. 
Simple Example

DF(B1) = {}
DF(B2) = {B4}
DF(B3) = {B4}
DF(B4) =
Simple Example

DF(B1) = {}
DF(B2) = {B4}
DF(B3) = {B4}
DF(B4) = {}
Simple Example

DF(B1) = {}
DF(B2) = \{B4\}
DF(B3) = \{B4\}
DF(B4) = \{\}

Φ-Function needed in B4 (for a)
Roadmap

- Intermediate representations
- Static Single Assignment
- SSA generation
- Dominance and SSA generation
- Applications of SSA
- $\Phi$-congruence and SSA removal
Properties of SSA

> **Simplifies many optimizations**
  — *Every variable has only one definition*
  — *Every use knows its definition, every definition knows its uses*
  — *Unrelated variables get different names*

> **Examples:**
  — *Constant propagation*
  — *Value numbering*
  — *Invariant code motion and removal*
  — *Strength reduction*
  — *Partial redundancy elimination*

Next lecture!
• **Constant propagation**: substitute constants and evaluate constant expressions

• **Value numbering**: number values & expressions to eliminate redundant computation

• **Invariant code motion and removal**: move invariant code out of loops

• **Strength reduction**: replace expensive operations by equivalent, cheaper ones (e.g. multiplication by addition)

• **Partial redundancy elimination**: move common subexpressions to eliminate re-computation
SSA in the Real World

> Invented end of the 80s, a lot of research in the 90s

> Used in many modern compilers
  — *ETH Oberon 2*
  — *LLVM*
  — *GNU GCC 4*
  — *IBM Jikes Java VM*
  — *Java Hotspot VM*
  — *Mono*
  — *Many more…*
Roadmap

> Intermediate representations
> Static Single Assignment
> SSA generation
> Dominance and SSA generation
> Applications of SSA
> $\Phi$-congruence and SSA removal

Transforming out-of SSA

> Processor cannot execute $\Phi$-Function

> How do we remove it?
Naive copy placement may produce incorrect results after optimization …
Here we simply push the assignments to $a_3$ up to each branch. Sreedhar shows that the naive approach can be wrong if variables “interfere”.


**Φ-Congruence**

**Idea:** transform program so that all variables in Φ are the same:

\[ a_1 = Φ(a_1, a_1) \quad \rightarrow \quad a_1 = a_1 \]

> Insert Copies
> Rename Variables
A choice between $a_1$ and $a_1$ is no choice at all. But how can we transform the IR so that all $\Phi$ functions take this form?
Φ-Congruence: Definitions

Φ-connected(x):

\[ a3 = \Phi(a1, a2) \]
\[ a5 = \Phi(a3, a4) \]

a1, a2, a3 are Φ-connected
a3, a4, a5 are Φ-connected

Φ-congruence-class:
Transitive closure of Φ-connected(x).

a1-a5 are Φ-congruent
Variables $x$ and $y$ are *connected* if they are used or defined in the same $\Phi$ instruction.
Φ-Congruence Property

Φ-congruence property:

All variables of the same congruence class can be replaced by one representative variable without changing the semantics.

SSA without optimizations has Φ-congruence property

Variables of the congruence class never live at the same time (by construction)
The property obviously holds before optimization, since all $\Phi$-connected variables started out as the same variable. Unfortunately, after optimizations have been introduced, this is no longer true.
Liveness

A variable \( v \) is *live* on edge \( e \) if there is a path through \( e \) to a use of \( v \) not passing through an assignment to \( v \).

*a and b are never live on the same edges, so two registers suffice to hold a, b and c*
To determine if a variable is *live* on edges along a path, follow paths from assignments to the last use before a new assignment. NB: c is implicitly assigned when it is defined, so is live from the start to its first use.
a and c are live at the same time: interference
If two variables are alive along the same edge, they interfere, and require separate memory locations (registers) to hold them. If they do not interfere, then they can share the same memory.
Φ-Removal: Big picture

> CSSA: SSA with Φ-congruence-property.
   — *directly after SSA generation*
   — *no interference*

> TSSA: SSA without Φ-congruence-property.
   — after optimizations
   — Interference

1. Transform TSSA into CSSA (fix interference)
2. Rename Φ-variables
3. Delete Φ
CSSA = Conventional SSA
TSSA = Transformed SSA

The idea is to eliminate the liveness interference introduced during optimization, and thus return to a CSSA form. The paper by Sreedhar mentioned above discusses possible approaches based on introducing copies of variables.
SSA and Register Allocation

> Idea: remove $\Phi$ as late as possible

> Variables in $\Phi$-function never live at the same time!
  — *Can be stored in the same register*

> Do register allocation on SSA!
So, don’t remove Φ functions before register allocation! Keep them till end. (There are many reasons to keep SSA as the IR for various phases in the back end.)
What you should know!

- Why do most compilers need an intermediate representation for programs?
- What are the key tradeoffs between structural and linear IRs?
- What is a “basic block”?
- What are common strategies for representing case statements?
- When does a program have an SSA form?
- What is a Φ-function?
- Where do we place Φ-functions?
- How to remove Φ-functions?
Can you answer these questions?

✎ Why can’t a parser directly produce high quality executable code?
✎ What criteria should drive your choice of an IR?
✎ What kind of IR does JTB generate?
✎ Why can we not directly generate executable code from SSA?
✎ Why do we use 3-address code and CFG for SSA?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/