6. Intermediate Representation

Prof. O. 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

Roadmap

> Intermediate representations
> Static Single Assignment
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
Intermediate Representation

**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
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 * –
```
A DAG is an AST with unique, shared nodes for each value.

\[
x := 2 \ast y + \sin(2\ast x) \\
z := x / 2
\]
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
```
3-address code

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

\[
\begin{align*}
\text{x – 2 * y} & \quad \rightarrow \quad t1 = 2 * y \\
                 & \quad \rightarrow \quad t2 = x – t1
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Assignments</th>
<th>( x = y \text{ op } z )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( x = \text{ 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 \text{ 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>t1</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>loadi</td>
<td>t2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>mult</td>
<td>t3</td>
<td>t2</td>
</tr>
<tr>
<td>4</td>
<td>load</td>
<td>t4</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>sub</td>
<td>t5</td>
<td>t4</td>
</tr>
</tbody>
</table>

**Triples**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>loadi</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
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
Static Single Assignment Form

- **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 …

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

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

http://en.wikipedia.org/wiki/Static_single_assignment_form

© Oscar Nierstrasz
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!
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

\[
\begin{align*}
a_1 &:= b_1 + c_1 \\
b_2 &:= c_1 + 1 \\
d_1 &:= b_2 + c_1 \\
a_2 &:= a_1 + 1 \\
e_1 &:= a_2 + b_2 \\
\end{align*}
\]
Example: Condition

Conditions: what to do on control-flow merge?

Original

if B then
  a := b
else
  a := c
end

... a ...

SSA

if B then
  a₁ := b
else
  a₂ := c
End

... a? ...
Solution: $\Phi$-Function

*Conditions: what to do on control-flow merge?*

Original

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

SSA

```
if B then
 a₁ := b
else
 a₂ := c
End
a₃ := $\Phi(a₁,a₂)$
... a₃ ...
```
The $\Phi$-Function

- $\Phi$-functions are always at the beginning of a basic block
- Select between values depending on control-flow
- $a_1 := \Phi(a_1...a_k)$: the block has $k$ preceding blocks

**PHI-functions are evaluated simultaneously within a basic block.**
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 *transfer of control* in a program
  — nodes are *basic blocks* (straight-line blocks of code)
  — edges represent *control flow* (loops, if/else, goto …)

```
if x = y then
  S1
else
  S2
end
S3
```

![Control Flow Graph Diagram]
if B then
  a1 := 1
else
  a2 := 2
End
a3 := PHI(a1,a2)
... a3 ...
Recall: IR

- front end produces IR
- optimizer transforms IR to more efficient program
- back end transform IR to target code
SSA as IR
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)
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.
Dominance Property of SSA

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

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

**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 \( N \)

*“Definition dominates use”*
Dominance and SSA Creation

> Dominance can be used to efficiently build SSA

> $\Phi$-Functions are placed in all basic blocks of the Dominance Frontier

— $\text{DF}(D) =$ the set of all nodes $N$ such that $D$ dominates an immediate predecessor of $N$ but does not strictly dominate $N$. 
Dominance and SSA Creation
Dominance and SSA Creation

Node 5 dominates all nodes in the gray area
Dominance and SSA Creation

Follow edges leaving the region dominated by node 5 to the region not strictly dominated by 5.

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)=

a := 1
a := 2
a
Simple Example

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

\[ a := 1 \rightarrow B2 \]
\[ a := 2 \rightarrow B3 \]
\[ a \rightarrow B4 \]
simple example
Simple Example

\[ \text{DF}(B1) = \{\} \]
\[ \text{DF}(B2) = \{B4\} \]
\[ \text{DF}(B3) = \]
\[ \text{DF}(B4) = \]
Simple Example

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

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

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

PHI-Function needed in B4 (for a)
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
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…*
Transforming out-of SSA

> Processor cannot execute $\Phi$-Function

> How do we remove it?
Simple Copy Placement

- $a_1 := b$
- $a_2 := 2$
- $a_3 := \text{PHI}(a_1, a_2)$
- ... $a_3$ ...

- $a_1 := b$
- $a_3 := a_1$
- $a_2 := 2$
- $a_3 := a_2$
- ... $a_3$ ...
Problems

- Copies need to be removed
- Wrong in some cases after reordering of code
**Idea:** transform program so that all variables in $\Phi$ are the same:

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

> Insert Copies
> Rename Variables
Φ-Congruence: Definitions

Φ-connected(x):

\[ a_3 = \Phi(a_1, a_2) \]
\[ a_5 = \Phi(a_3, a_4) \]

a1, a4 are connected

Φ-congruence-class:

Transitive closure of Φ-connected(x).
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)
A variable \( v \) is \textit{live} on edge \( e \) if there is a path from \( e \) to a use of \( v \) not passing through a definition of \( v \).

\[ \text{a := 0} \]
\[ \text{b := a + 1} \]
\[ \text{c := c + b} \]
\[ \text{a := b * 2} \]
\[ \text{a < N} \]
\[ \text{return c} \]

\[ \text{a := 0} \]
\[ \text{b := a + 1} \]
\[ \text{c := c + b} \]
\[ \text{a := b * 2} \]
\[ \text{a < N} \]
\[ \text{return c} \]

\[ \text{a := 0} \]
\[ \text{b := a + 1} \]
\[ \text{c := c + b} \]
\[ \text{a := b * 2} \]
\[ \text{a < N} \]
\[ \text{return c} \]

\[ \text{a and b are never live at the same time, so two registers suffice to hold a, b and c} \]
Interference

\(a, c\) live at the same time: interference
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
Example: Problematic case

X2 and X3 interfere

Solution: Break up

x2 = phi(x1, x3)

x3 = x2 + 1

y = phi(x1, x3)

x2 = y

x3 = x2 + 1

= x2

= x2
> 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!
SSA: Literature

Books:
- SSA Chapter in Appel
  Modern Compiler Impl. In Java
- Chapter 8.11 Muchnik:
  Advanced Compiler Construction

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

PHI-Removal: Sreedhar et at. *Translating out of Static Single Assignment Form* (LNCS 1694)
Summary

> SSA, what it is and how to create it
  — Where to place $\Phi$-functions?

> Transformation out of SSA
  — Placing copies
  — Remove $\Phi$

Next Week: Optimizations
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 a program has SSA form.
- What is a $\Phi$-function.
- When do we place $\Phi$-functions
- How to remove $\Phi$-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?
License

http://creativecommons.org/licenses/by-sa/3.0/

Attribution-ShareAlike 3.0 Unported

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work

Under the following conditions:
  Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.