Intermediate Representations and Static Single Assignment Form

Daniel Grund
grund@cs.uni-sb.de
Saarland University

CC Winter Term 07/08
Outline

1. Overview

2. Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3. Static Single Assignment Form
   - Introduction
   - Theory
   - SSA Construction
Frontend

- 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

1 Overview

2 Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3 Static Single Assignment Form
   - Introduction
   - Theory
   - SSA Construction
Outline

1 Overview

2 Intermediate Representations
  - Why?
  - How?
  - IR Concepts

3 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
Outline

1. Overview

2. Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3. 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

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

2. Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3. 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
  \[(x = a + b \ast c; \ y = \text{call}\ foo\ (3 \ast x);)\]
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

1 Overview

2 Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3 Static Single Assignment Form
   - Introduction
   - Theory
   - SSA Construction
Outline

1 Overview

2 Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3 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.

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?

\[
\begin{align*}
(a, b) &= \text{start} \\
\text{if } b < a \\
c &:= a - b \\
c &:= 0 \\
\text{return } c
\end{align*}
\]
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

\[
\begin{align*}
\text{non-SSA} & \quad \text{SSA} \\
(a, b) &= \text{start} \\
\text{if } b < a \\
\begin{align*}
&c := a - b \\
&c := 0 \\
\end{align*} & \begin{align*}
&c_1 := a - b \\
&c_2 := 0 \\
&c_3 := \phi(c_1, c_2) \\
\text{return } c_3
\end{align*}
\]
Outline

1 Overview

2 Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3 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

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

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

\[ a = 23 \]
\[ b = 42 \]

\[ t = a \]
\[ a = b \]
\[ b = t \]

\textit{call} \ printf\ ,\ str\ ,\ a\ ,\ b

\[ a_1 = 23 \]
\[ b_1 = 42 \]

\[ t = \phi(a_1, a_2) \]
\[ a_2 = \phi(b_1, b_2) \]
\[ b_2 = t \]

\textit{call} \ printf\ ,\ str\ ,\ a_2\ ,\ b_2
Why Simultaneously? Swap Example

\[ a = 23 \]
\[ b = 42 \]
\[ t = a \]
\[ a = b \]
\[ b = t \]
\[ \text{call } printf, str, a, b \]

\[ a_1 = 23 \]
\[ b_1 = 42 \]
\[ t = \phi(a_1, a_2) \]
\[ a_2 = \phi(b_1, t) \]
\[ \text{call } printf, str, a_2, t \]
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 $X > Y \land \nexists Z : X > Z > Y$
(Iterated) Dominance Frontier

- **Dominance frontier**: $\text{DF}(X)$
  Set of blocks just “one step” beyond the dominated region
  
  $\text{DF}(X) = \{ Y \mid \exists P \in \text{Pred}(Y) : X \geq P \} \setminus \{ X > Y \}$

- **Dominance frontier of a set of nodes $M$**: $\text{DF}(M)$
  
  $\text{DF}(M) = \bigcup_{X \in M} \text{DF}(X)$

- **Iterated dominance frontier**: $\text{DF}^+(M)$
  
  $\text{DF}^1(M) = \text{DF}(M)$
  
  $\text{DF}^{i+1}(M) = \text{DF}(M \cup \text{DF}^i(M))$

- **Example on black board**
Theorem

Let $M$ be the set of blocks containing definitions of variable $v$. Then $\phi$'s must be placed in all $B \in \text{DF}^+(M)$ from which a use of $v$ is reachable.

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* DF is necessary, since inserted $\phi$'s are new definitions of the variable
Outline

1. Overview

2. Intermediate Representations
   - Why?
   - How?
   - IR Concepts

3. Static Single Assignment Form
   - Introduction
   - Theory
   - SSA Construction
SSA Construction

- In the worst case each BB has a φ for each variable.
  - complexity $O(n^2)$
  - linear in practice
- $DF^+$ only says where to place φs. What are the correct arguments?
- Idea by Click 1995:
  - don’t compute $DF^+$ explicitly
  - perform global value numbering during construction
  - place φs on the fly
(Global) 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 GVN (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 \( VN(\tau, y, z) \) 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 GVN (2)

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

- getVN($y$), getVN($z$)
- compute VN($\tau, y, z$)
- if value number is new insert
  VN($\tau, y, z$) = $\tau$(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 ν_i is valid for variable ν in current BB return ν_i
- else if BB has exactly one predecessor call getVN(ν) there
- else (more predecessors):
  - call getVN(ν) for all predecessors
  - let the values ν_1, ν_2, ... be the results
  - insert VN(φ, ν, ν) = φ(ν_1, ν_2, ...) 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

\[ a := \ldots \]
\[ \ldots := a \]
\[ a := a + 1 \]

\[ a_1 := \ldots \]
\[ \ldots := a \]
\[ a := a + 1 \]

\[ a_2 := \phi'(a) \]
\[ \ldots := a_2 \]
\[ a_3 := a_2 + 1 \]

\[ a_4 := \phi(a_1, a_3) \]
\[ a_2 := a_4 \]
\[ \ldots := a_2 \]

\[ a_3 := a_2 + 1 \]

\[ \phi \text{ nicht berechenbar} \]

\[ a_2 := \phi'(a) \]
\[ \ldots := a_2 \]
\[ a := a + 1 \]

\[ a := \ldots \]
\[ \ldots := a \]
\[ a := a + 1 \]
Unknown Predecessors: Consequences

Consequence: Do construction in control-flow order (as much as possible)

- keeps number of $\phi$’s low
- dominating BBs are constructed before dominated BBs
- this makes the implicit CSE more effective
Larger Example

(1) \texttt{a:=1;}
(2) \texttt{b:=2;}

\begin{verbatim}
while (true){
    (3) \texttt{c:=a+b;}
    (4) \texttt{if (d:=c-a)}
    (5) \hspace{1em} \texttt{while (d:=b*d)}{
    (6) \hspace{2em} \texttt{d:=a+b;}
    (7) \hspace{2em} \texttt{e:=e+1;}
    }
    (8) \texttt{b:=a+b;}
    (9) \texttt{if (e:=c-a)}
    \hspace{1em} \texttt{break;}
}
(10) \texttt{a:=b*d;}
(11) \texttt{b:=a-d;}
\end{verbatim}
SSA Construction Block 1

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

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

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

```
a := b * d
b := a - d
```

```
c := a + b
```
Get value number for a first places $\phi'$ for $a$ ...

\[
a_1 := 1 \\
b_1 := 2
\]

GB₁

\[
a_2 := \phi'(a) \\
c := a_2 + b
\]

GB₂

\[
a := 1 \\
b := 2
\]

GB₅

\[
d := b*d \\
d := a + b \\
e := e + 1
\]

GB₆

\[
c := a + b \\
d := c - a
\]

GB₄

\[
b := a + b \\
e := c - a
\]

GB₃

\[
a := b*d \\
b := a - d
\]

GB₆

\[
a := b*d \\
b := a - d
\]
SSA Construction Block 2

...then for $b$...

\[
\begin{align*}
a_2 &:= \phi'(a) \\
b_2 &:= \phi'(b) \\
c &:= a_2 + b_2
\end{align*}
\]
SSA Construction Block 2

...and eventually a VN for c.
Inserting $d := c - a$ works like normal value numbering.
\[ a_1 := 1 \]
\[ b_1 := 2 \]

\[ a_2 := \phi'(a) \]
\[ b_2 := \phi'(b) \]
\[ c_1 := a_2 + b_2 \]
\[ d_1 := c_1 - a_2 \]

\[ b_3 := \phi'(b) \]
\[ d_2 := \phi'(d) \]
\[ d := b_3 \times d_2 \]

\[ a := 1 \]
\[ b := 2 \]

\[ c := a + b \]
\[ d := c - a \]

\[ d := a + b \]
\[ e := e + 1 \]
\[ e := c - a \]

\[ a := b \times d \]
\[ b := a + b \]
\[ b := a - d \]
SSA Construction Block 3

\[
a_1 := 1 \\
b_1 := 2
\]

\[
a_2 := \phi'(a) \\
b_2 := \phi'(b) \\
c_1 := a_2 + b_2 \\
d_1 := c_1 - a_2
\]

\[
b_3 := \phi'(b) \\
d_2 := \phi'(d) \\
d_3 := b_3 \cdot d_2
\]

\[
a_2 := \phi'(a) \\
b_2 := \phi'(b) \\
c_1 := a_2 + b_2 \\
d_1 := c_1 - a_2
\]

\[
a := 1 \\
b := 2
\]

\[
c := a + b \\
d := c - a
\]

\[
d := b \cdot d
\]

\[
a := b \cdot d \\
b := a + b \\
e := e + 1 \\
e := c - a
\]

\[
d := a + b \\
e := e + 1 \\
e := c - a
\]

\[
a := b \cdot d \\
b := a + b
\]
Call to getVN(a) in 4 lead to recursive call getVN(a) in 3. This in turn produces a $\phi'$ for a in 3.
SSA Construction Block 4

GB₁
\[ a₁ := 1 \]
\[ b₁ := 2 \]

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

GB₃
\[ b₃ := \phi'(b) \]
\[ d₂ := \phi'(d) \]
\[ a₃ := \phi'(a) \]
\[ e₃ := \phi'(e) \]
\[ d₃ := b₃ \times d₂ \]

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

GB₅
\[ a := 1 \]
\[ b := 2 \]
\[ c := a + b \]
\[ d := c - a \]

GB₆
\[ d := b \times d \]
\[ b := a + b \]
\[ e := e + 1 \]
\[ e := c - a \]

\[ a := b \times d \]
\[ b := a - d \]
All predecessors of 3 are now in SSA form: φs are placed. In block 2 a φ’ is recursively placed for e.
getVN(a) in 5 recognizes copies, finds unique definition: no ϕ is necessary
SSA Construction Block 5

\[ a_1 := 1 \]
\[ b_1 := 2 \]

\[ a := 1 \]
\[ b := 2 \]
\[ d := b \times d \]
\[ d := a + b \]
\[ e := e + 1 \]
\[ b := a + b \]
\[ e := c - a \]

\[ a := b \times d \]
\[ b := a + b \]
\[ c := a + b \]
\[ d := c - a \]

\[ b_3 := b_2 \]
\[ d_2 := \phi (d_1, d_4) \]
\[ a_3 := a_2 \]
\[ e_3 := \phi (e_2, e_4) \]
\[ d_3 := b_3 \times d_2 \]

\[ a := 1 \]
\[ b := 2 \]
\[ a_2 := \phi' (a) \]
\[ b_2 := \phi' (b) \]
\[ e_2 := \phi' (e) \]
\[ c_1 := a_2 + b_2 \]
\[ d_1 := c_1 - a_2 \]

\[ d := b \times d \]
\[ d := a + b \]
\[ e := e + 1 \]
\[ b := a + b \]
\[ e := c - a \]

\[ a := b \times d \]
\[ b := a + b \]
\[ c := a + b \]
\[ d := c - a \]
SSA Construction Block 5

\[ a_1 := 1 \]
\[ b_1 := 2 \]

\[ a_2 := \phi'(a) \]
\[ b_2 := \phi'(b) \]
\[ e_2 := \phi'(e) \]
\[ c_1 := a_2 + b_2 \]
\[ d_1 := c_1 - a_2 \]

\[ b_3 := b_2 \]
\[ d_2 := \phi(d_1, d_4) \]
\[ a_3 := a_2 \]
\[ e_3 := \phi(e_2, e_4) \]
\[ d_3 := b_3 * d_2 \]

\[ d_4 := a_3 + b_3 \]
\[ e_4 := e_3 + 1 \]

\[ a := 1 \]
\[ b := 2 \]

\[ c := a + b \]
\[ d := c - a \]

\[ d := a + b \]
\[ e := e + 1 \]
\[ e := c - a \]

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

Algorithm recognizes: $e$ is uninitialized! Insert undefined value $e_1$.
Recursive call to getVN(d) in 5 places complete $\phi$ function $d_5$
Optimization: Copy Propagation

\[
\begin{align*}
a_1 & := 1 \\
b_1 & := 2
\end{align*}
\]

\[
\begin{align*}
a_2 & := a_1 \\
b_2 & := b_1 * b_5 \\
e_2 & := e_1 * e_5 \\
c_1 & := a_1 + b_2 \\
d_1 & := c_1 - a_1
\end{align*}
\]

\[
\begin{align*}
b_3 & := b_2 \\
d_2 & := \phi(d_1, d_4) \\
a_3 & := a_2 \\
e_3 & := \phi(e_2, e_4) \\
d_3 & := b_2 * d_2
\end{align*}
\]

\[
\begin{align*}
d_4 & := a_1 + b_2 \\
e_4 & := e_3 + 1
\end{align*}
\]

\[
\begin{align*}
a_4 & := b_5 * d_5 \\
b_6 & := a_4 - d_5
\end{align*}
\]
Optimization: Constant Propagation

GB_1
a_1 := 1
b_1 := 2

GB_2
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

GB_3
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

GB_4
d_4 := 1+b_2
e_4 := e_3 + 1

GB_5
d_5 := \phi(d_3, d_1)
b_5 := 1+b_2
e_5 := c_1 - 1

GB_6
a_4 := b_5 * d_5
b_6 := a_4 - d_5

d:=a*b
d:=a+b
d:=c-a

a:=b*d
d:=a+b
d:=c-a

b:=a+b
b:=a-d
Optimization: Dead Code Elimination

```
Optimization: Dead Code Elimination

a := b * d
b := a + b
e := e + 1

GB
GB1

GB2

GB3

d2 := \phi(d_1, d_4)
e3 := \phi(e_2, e_4)
d3 := b2 * d2

c := a + b
d := c - a

GB4

d4 := 1 + b2
e4 := e3 + 1

GB5

b2 := \phi(2, b5)
e2 := \phi(e1, e5)
c1 := 1 + b2
d1 := c1 - 1

d5 := \phi(d3, d1)
b5 := 1 + b2
e5 := c1 - 1

GB6

a4 := b5 * d5
b6 := a4 - d5
```
Further Optimizations

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

Daniel Grund
Interm. Represent. and SSA
CC Winter Term 07/08 60 / 61
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)