Compiler Optimisation
4 – Dataflow Analysis

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2019
Introduction

This lecture:
- Data flow termination
- More data flow examples
- Dominance
- Static single-assignment form
Data flow termination

Fixed Point
A fixed point, $x$, of function $f : T \rightarrow T$ is when $f(x) = x$

Partial ordering
A binary relation, $\leq$, among elements of a set, $S$, that is:

- $\forall a \in S, \quad a \leq a$ \hspace{1cm} \text{reflexive}$
- $\forall a, b \in S, \quad a \leq b \land b \leq a \implies a = b$ \hspace{1cm} \text{antisymmetric}$
- $\forall a, b, c \in S \quad a \leq b \land b \leq c \implies a \leq c$ \hspace{1cm} \text{transitive}$

Partially ordered set (poset)
A set with a partial order
Data flow termination

Join semi-lattice

Partially ordered set that has a join (a least upper bound) for any nonempty finite subset

\[ \forall x, y, z \in V \]
\[ x \land (y \land z) = (x \land y) \land z \quad \text{Associativity} \]
\[ x \land y = y \land x \quad \text{Commutativity} \]
\[ x \land x = x \quad \text{Idempotency} \]
\[ \top \land x = x \quad \text{Top element} \]

Note, *meet semi-lattice* has \( \bot \) and *complete lattice* has both

Often just use ‘semi-lattice’
Data flow termination
Lattices

Example lattice

Hasse diagram for the set of all subsets of $x, y, z$, ordered by inclusion
Is it a lattice?
Data flow termination

Lattices

Example lattice

Hasse diagram for the set of all subsets of $x, y, z$, ordered by inclusion

Yes, and is lattice for LiveOut, with variables, $x, y, z$
Data flow termination

Example non-lattices
Data flow termination
Lattices

Example non-lattices

```
<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>e</td>
</tr>
</tbody>
</table>
```
Data flow termination

Lattices

Example non-lattices
Data flow termination

Each block or statement has a particular function, e.g. based on *Kill* and *Gen* sets.

Let $F$ be the set of transfer functions.

### Sufficient termination constraints

- $V$ and $\land$ for a semi-lattice
- $F$ has identity: $I(x) = x, \forall x \in V$
- $F$ is closed under composition: $\forall f, g \in F, h = f \circ g \in F$
- $F$ is monotonic: $\forall f \in F, f(x \land y) \leq f(x) \land f(y)$

See \[\ref{10.11}\]
A variable $v$ is **live-out** of statement $s$ if $v$ is used along some control path starting at $s$.

Otherwise, we say that $v$ is **dead**.

A variable is live if it holds a value that may be needed in the future.

Information flows *backwards* from statement to predecessors.

Liveness useful for optimisations (e.g. register allocation, store elimination, dead code...).
A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

Diagram:

- $s_1: a := 2$
- $s_2: \textbf{if } x > 0$
- $s_3: a := x + 1$
- $s_4: b := 0$
- $s_5: c := a \times 2$
- $s_6: \textbf{if } y < x$

Liveness
A variable \( v \) is live-out of statement \( s \) if \( v \) is used along some control path starting at \( s \).
A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

Liveness
A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 
A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

```
\[ s_1 \quad a := 2 \]
\[ s_2 \quad \textbf{if } x > 0 \quad \{ x, y, a \} \]
\[ s_3 \quad a := x + 1 \quad \{ x, y \} \]
\[ s_4 \quad b := 0 \]
\[ s_5 \quad c := a \times 2 \quad \{ x, y, a \} \]
\[ s_6 \quad \textbf{if } y < x \quad \{ x, y \} \]
\[ \{ \} \]
```
A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

Liveness

$$s_1: a := 2$$

$$s_2: \text{if } x > 0$$

$$s_3: a := x + 1$$

$$s_4: b := 0$$

$$s_5: c := a \times 2$$

$$s_6: \text{if } y < x$$

Note: $x, y$ used but not defined

${x, y}$

${x, y, a}$

${x, y, a}$

${x, y, a}$

${x, y}$

${x, y, a}$

${x, y, a}$

${x, y}$

A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

Diagram:

- $s_1$: $a := 2$
- $s_2$: if $x > 0$
- $s_3$: $a := x + 1$
- $s_4$: $b := 0$
- $s_5$: $c := a \times 2$
- $s_6$: if $y < x$
A variable $v$ is live-out of statement $s$ if $v$ is used along some control path starting at $s$. 

Note: $b, c$ defined but not used, so $s_4, s_5$ useless, if removed $s_1, s_3$ useless.
Liveness

- Live variables come up from their successors using them
  \[ \text{Out}(s) = \bigcup_{n \in \text{Succ}(s)} \text{In}(n) \]
- Transfer back across the node
  \[ \text{In}(s) = \text{Out}(s) - \text{Kill}(s) \cup \text{Gen}(s) \]
- Used variables are live
  \[ \text{Gen}(s) = \{ u \text{ such that } u \text{ is used in } s \} \]
- Defined but not used variables are killed
  \[ \text{Kill}(s) = \{ d \text{ such that } d \text{ is defined in } s \text{ but not used in } s \} \]
- If we don’t know, start with empty
  \[ \text{Init}(s) = \emptyset \]
Others

- Constant propagation - show variable has same constant value at some point
  - Strictly speaking does not compute expressions except \( x := \text{const} \), or \( x := y \) and \( y \) is constant
  - Often combined with constant folding that computes expressions
- Copy propagation - show variable is copy of other variable
- Available expressions - set of expressions reaching by all paths
- Very busy expressions - expressions evaluated on all paths leaving block - for code hoisting
- Definite assignment - variable always assigned before use
- Redundant expressions, and partial redundant expressions
- Many more - read about them!
DOMINATORS

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

What direction?
What value set?
What transfer?
What Meet?
Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

What direction?
What value set?
What transfer?
What Meet?
Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \triangleright \triangleright b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

**Direction:** Forward

What value set?
What transfer?
What Meet?
Initial values?
**Dominators**

CFG node \( b_i \) dominates \( b_j \), written \( b_i \gg b_j \), iff every path from the start node to \( b_j \) goes through \( b_i \).

Design data flow equations to compute which nodes dominate each node.

*Direction: Forward*

*What value set?*

What transfer?

What Meet?

Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

**Direction:** Forward

**Values:** Sets of nodes

What transfer?
What Meet?
Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

*Direction:* Forward

*Values:* Sets of nodes

**What transfer?**

**What Meet?**

**Initial values?**
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

*Direction*: Forward

*Values*: Sets of nodes

*Transfer*: $Out(n) = In(n) \cup \{n\}$

What Meet?

Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

**Direction:** Forward

**Values:** Sets of nodes

**Transfer:** $\text{Out}(n) = \text{In}(n) \cup \{n\}$

**What Meet?**
Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$.

Design data flow equations to compute which nodes dominate each node.

**Direction:** Forward

**Values:** Sets of nodes

**Transfer:** $\text{Out}(n) = \text{In}(n) \cup \{n\}$

**Meet:** $\text{In}(n) = \bigcap_{\forall n \in \text{Pred}(s)} \text{Out}(s)$

Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$

Design data flow equations to compute which nodes dominate each node

**Direction:** Forward  
**Values:** Sets of nodes  
**Transfer:** $Out(n) = In(n) \cup \{n\}$  
**Meet:** $In(n) = \bigcap_{\forall n \in Pred(s)} Out(s)$

Initial values?
Dominators

CFG node $b_i$ dominates $b_j$, written $b_i \gg b_j$, iff every path from the start node to $b_j$ goes through $b_i$

Design data flow equations to compute which nodes dominate each node

*Direction*: Forward

*Values*: Sets of nodes

*Transfer*: $Out(n) = In(n) \cup \{n\}$

*Meet*: $In(n) = \bigcap_{s \in \text{Pred}(s)} Out(s)$

*Initial*: $Init(n_0) = \{n_0\}; Init(n) = \text{all}$
### Dominators

<table>
<thead>
<tr>
<th>Post-dominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node ( z ) is said to post-dominate a node ( n ) if all paths to the exit node of the graph starting at ( n ) must go through ( z )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strict dominance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node ( a ) strictly dominates ( b ) iff ( a \gg b \land a \neq b )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Immediate dominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>( idom(n) ) strictly dominates ( n ) but not any other node that strictly dominates ( n )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dominator tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree where node’s children are those it immediately dominates</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dominance frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td>( DF(n) ) is set of nodes, ( d ) s.t. ( n ) dominates an immediate predecessor of ( d ), but ( n ) does not strictly dominate ( d )</td>
</tr>
</tbody>
</table>
Dominators

Example: Dominator tree

Where are dominance frontiers?
Dominators

Example: Dominator tree

\[ DF(b_5) = \{ b_3 \} \]
Dominators

Example: Dominator tree

\[
\text{DF}(b_1) = \{b_1\}
\]
Static single-assignment form (SSA)

- Often allowing variable redefinition complicates analysis
- In SSA:
  - One variable per definition
  - Each use refers to one definition
  - Definitions merge with $\phi$ functions
  - $\Phi$ functions execute instantaneously in parallel
- Used by or simplifies many analyses
Static single-assignment form (SSA)

Example: Intuitive conversion to SSA

Original CFG

- $s_1$: $a := 2$
- $s_2$: if $x > 0$
- $s_3$: $a := x + 1$
- $s_4$: $b := 0$
- $s_6$: $c := a * 2$
- $s_7$: if $y < x$

Diagram
Static single-assignment form (SSA)

Example: Intuitive conversion to SSA

Rename multiple definitions of same variable
Static single-assignment form (SSA)

Example: Intuitive conversion to SSA

Repeatedly merge definitions with $\phi$

But $a_4$ reaches $s_4$ and $s_6$, too

Semantics changed
Static single-assignment form (SSA)

Example: Intuitive conversion to SSA

Now in SSA form
Static single-assignment form (SSA)

Types of SSA

- **Maximal SSA** - Places \( \phi \) node for variable \( x \) at every join block if block uses or defines \( x \)
- **Minimal SSA** - Places \( \phi \) node for variable \( x \) at every join block with 2+ reaching definitions of \( x \)
- **Semipruned SSA** - Eliminates \( \phi \)s not live across block boundaries
- **Pruned SSA** - Adds liveness test to avoid \( \phi \)s of dead definitions
Static single-assignment form (SSA)
Conversion to SSA sketch\(^2\)

- For each definition\(^1\) of \(x\) in block \(b\), add \(\phi\) for \(x\) in each block in \(DF(b)\)
- This introduces more definitions, so repeat
- Rename variables
- Can be done in \(T(n) = O(n)\), if liveness cheap

---

\(^1\)Different liveness tests (including none) here change SSA type
\(^2\)See \(\text{EaC 9.3.1-9.3.4}\)
Static single-assignment form (SSA)
Conversion from SSA sketch\(^3\)

- Cannot just remove \(\phi\) nodes; optimisations make this unsafe
- Place copy operations on incoming edges
- Split edges if necessary
- Delete \(\phi\)s
- Remove redundant copies afterwards

\(^3\)See EaC 9.3.5
Static single-assignment form (SSA)
Conversion from SSA

Example: Intuitive conversion from SSA

Original SSA CFG
Static single-assignment form (SSA)
Conversion from SSA

Example: Intuitive conversion from SSA

Place copies
Static single-assignment form (SSA)
Conversion from SSA

Example: Intuitive conversion from SSA

\[ a_1 := 2 \]
\[ a_3 := a_1 \]
\[ \textbf{if} \ x > 0 \]
\[ a_2 := x + 1 \]
\[ a_4 := a_2 \]
\[ a_3 := \phi(a_1, a_4) \]
\[ b := 0 \]
\[ a_4 := a_3 \]
\[ a_4 := \phi(a_2, a_3) \]
\[ c := a_4 \times 2 \]
\[ \textbf{if} \ y < x \]
\[ a_3 := a_4 \]

Split where necessary
Static single-assignment form (SSA)

Conversion from SSA

Example: Intuitive conversion from SSA

```
\begin{align*}
    & s_1: a_1 := 2 \\
    & s_2: a_3 := a_1 \\
    & s_3: \text{if } x > 0 \\
    & s_4: a_2 := x + 1 \\
    & s_6: a_4 := a_2 \\
    & s_9: c := a_4 \times 2 \\
    & s_A: \text{if } y < x \\
    & s_8: b := 0 \\
    & s_7: a_4 := a_3 \\
    & s_C: a_3 := a_4
\end{align*}
```

Remove $\phi$s
Static single-assignment form (SSA)

Extensions

- Dataflow assumes that all paths in the CFG are taken hence conservative approximations
  - Guarded SSA attempts to overcome this by having additional meet nodes $\gamma$, $\eta$ and $\mu$ to carry conditional information around
  - Array based SSA models access patterns$^4$
- Inter-procedural challenging. Pointers destroy analysis! Large research effort in points-to analysis.

$^4$Can be generalised using Presburger formula
Static single-assignment form (SSA)
Constant propagation

Three possible values per variable

<table>
<thead>
<tr>
<th></th>
<th>Not a constant</th>
<th>Constant value $k$</th>
<th>Not computed (maybe never)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\top$</td>
<td>Not a constant</td>
<td>Constant value $k$</td>
<td>Not computed (maybe never)</td>
</tr>
</tbody>
</table>

Meet is $\top \land x = x$, $\bot \land x = \bot$, $c \land c = c$, $c \land d = \bot$ if $c \neq d$

Transfer functions compute value if all inputs are constant
Summary

- Data flow termination
- More data flow examples
- Dominance
- Static single-assignment form
The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk

• 4-year programme: MSc by Research + PhD

• Research-focused: Work on your thesis topic from the start

• Collaboration between:
  ▶ University of Edinburgh’s School of Informatics
    ✴ Ranked top in the UK by 2014 REF
  ▶ Edinburgh Parallel Computing Centre
    ✴ UK’s largest supercomputing centre

• Research topics in software, hardware, theory and application of:
  ▶ Parallelism
  ▶ Concurrency
  ▶ Distribution

• Full funding available

• Industrial engagement programme includes internships at leading companies

The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk