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:

- $\forall a \in S, \ a \leq a$  reflexive
- $\forall a, b \in S, \ a \leq b \land b \leq a \Rightarrow a = b$  antisymmetric
- $\forall a, b, c \in S \ a \leq b \land b \leq c \Rightarrow a \leq c$  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

\[
\forall x, y, z \in V \\
x \land (y \land z) = (x \land y) \land z
\]

- **Associativity**
- **Commutativity**
- **Idempotency**
- **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

Lattices

Example non-lattices
Data flow termination

Lattices

Example non-lattices

Left: c and d have no common upper bound
Data flow termination

Lattices

Example non-lattices

Right: b and c have common upper bounds (d, e, f), but no least upper bound
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 $\wedge$ 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: $\forall f \in F$, $f(x \wedge y) \leq f(x) \wedge f(y)$

See $\textit{10.11}$
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 $s$. 

```
$s_1$: $a := 2$

$s_2$: 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 $s$. 

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

Diagram: 

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

Variables used: $\{x, y\}$
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 $s$. 

Note: $x, y$ used but not defined.
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
  \[ Out(s) = \bigcup_{n \in \text{Succ}(s)} \text{In}(n) \]

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

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

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

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

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

### Post-dominator
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 $z$.

### Strict dominance
Node $a$ strictly dominates $b$ iff $a \gg b \land a \neq b$.

### Immediate dominator
$idom(n)$ strictly dominates $n$ but not any other node that strictly dominates $n$.

### Dominator tree
Tree where node's children are those it immediately dominates.

### Dominance frontier
$DF(n)$ is set of nodes, $d$ s.t. $n$ dominates an immediate predecessor of $d$, but $n$ does not strictly dominate $d$. 
Dominators

Example: Dominator tree

Where are dominance frontiers?
Dominators

Example: Dominator tree

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

Example: Dominator tree

\[ DF(b_1) = \{ b_1 \} \]
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

\[ \begin{align*}
&s_1: a := 2 \\
&s_2: \textbf{if } x > 0 \\
&s_3: a := x + 1 \\
&s_4: b := 0 \\
&s_5: c := a \times 2 \\
&s_6: \textbf{if } y < x
\end{align*} \]
Static single-assignment form (SSA)

Example: Intuitive conversion to SSA

```
S1: a1 := 2
S2: if x > 0
S3: a2 := x + 1
S4: b := 0
S6: c := a * 2
S7: if y < x
```

Could be either a1 or a2

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

Example: Intuitive conversion to SSA

Repeatedly merge definitions with $\phi$
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 EaC 9.3.1-9.3.4
Static single-assignment form (SSA)
Conversion from SSA sketch

- 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

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

```
a := 2
a := a
if x > 0

a := x + 1
a := a

a := \Phi(a, a)

b := 0
a := a

a := \Phi(a, a)
c := a * 2
if y < x

Place copies
```

Cannot insert

Actually fine here, just for demo
Static single-assignment form (SSA)
Conversion from SSA

Example: Intuitive conversion from SSA

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

Split where necessary
Static single-assignment form (SSA)

Conversion from SSA

Example: Intuitive conversion from SSA

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>$k$</td>
<td>$\bot$</td>
<td></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