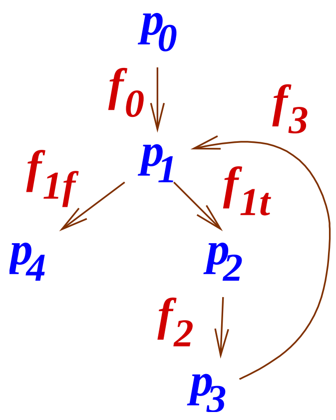
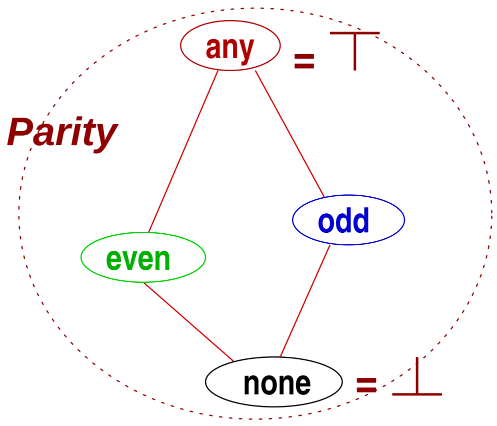
Review of yesterday's lecture

We compute properties of the storage vectors that arrive at the program points:

p_0 : `y = 1;`
p_1 : `while Even(x) {`
    p_2 : `y = y * x;`
    p_3 : `x = x div2;`
  `}`
p_4 : *exit*





Abstract model:

$p_0, \langle \top, \top \rangle$
$p_1, \langle \top, odd \rangle$

$p_4, \langle odd, odd \rangle$ $p_2, \langle even, odd \rangle$
$p_3, \langle even, even \rangle$

$p_1, \langle \top, even \rangle$

$p_4, \langle odd, even \rangle$ $p_2, \langle even, even \rangle$

Sticky collecting semantics:

$p_0 \mapsto \{\langle \top, \top \rangle\}$

$p_1 \mapsto \{\langle \top, odd \rangle, \langle \top, even \rangle\}$

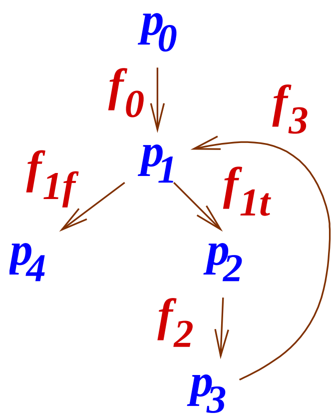
$p_2 \mapsto \{\langle even, odd \rangle, \langle even, even \rangle\}$

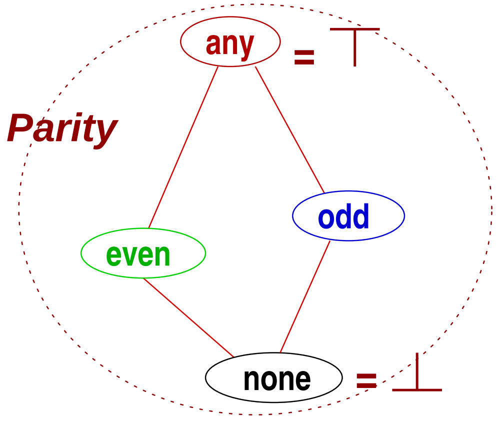
$p_3 \mapsto \{\langle even, even \rangle\}$

$p_4 \mapsto \{\langle odd, odd \rangle, \langle odd, even \rangle\}$

Data-flow equations compute the information more efficiently:

```
p0 :  y = 1;
p1 :  while Even(x) {
          p2 : y = y * x;
          p3 : x = x div2;
      }
p4 :  exit
```





Flow equation format: $p_i\mathit{Store} = \bigsqcup_{p_j \in pred(p_i)} f_j^{\#}(p_j\mathit{Store})$

Iterative solution to flow equations (independent-attribute semantics):

$$p_0\mathit{Store} = \langle x : \top, y : \top \rangle = \langle \top, \top \rangle$$

$$p_1\mathit{Store} = f_0^{\#}(p_0\mathit{Store}) \sqcup f_3^{\#}(p_3\mathit{Store}) = \langle \top, \top \rangle$$

$$p_2\mathit{Store} = f_{1t}^{\#}(p_1\mathit{Store}) = \langle even, \top \rangle$$

$$p_3\mathit{Store} = f_2^{\#}(p_2\mathit{Store}) = \langle even, even \rangle$$

$$p_4\mathit{Store} = f_{1f}^{\#}(p_1\mathit{Store}) = \langle odd, \top \rangle$$

(At p_2 and p_4, y's value was both $even$ and odd, so it is weakened to $even \sqcup odd = \top$.)

Flow equations can also compute patterns/properties of the *execution paths* from the program's entry to a program point:

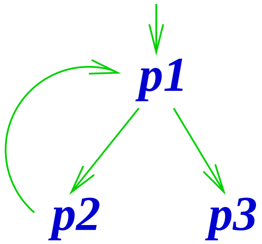
Example: *reaching definitions:* which assignments reach a program point

An assignment, $\mathrm{p} : \mathtt{x} = \mathtt{e}$, *reaches* p' if it precedes p' in some execution path, and its variable, $\mathtt{x}$, is not modified between p and p'.

Example (the reaching definitions are written as a set):

$$
\begin{array}{l}
\{\} \\
\quad \mathrm{p}_0 : \mathtt{x} = \mathtt{z} + 1 \\
\{\mathrm{p}_0\} \\
\quad \mathrm{p}_1 : \mathtt{y} = \mathtt{x} * \mathtt{x} \\
\{\mathrm{p}_0, \mathrm{p}_1\} \\
\quad \mathrm{p}_2 : \mathtt{x} = 2 \\
\{\mathrm{p}_1, \mathrm{p}_2\}
\end{array}
$$

Example:



p_1: `while x = 0`

p_2: `do x := y + 1`

p_3 : exit

Assignment, p : `x := e`, *possibly reaches* p' if it is *Defined* in some execution path to p', and its variable `x` is *not Modified* between p and p'.

$$RD(p) = \bigcup_{p' \in \mathrm{pred}\ p} (RD(p') \cap notModified(p')) \cup Defined(p')$$

p_i : `x := e`

$Defined(p_i) = \{p_i\}$

$notModified(p_i) = \{p_j \mid p_j : \mathtt{y\ :=\ e}, \mathtt{y} \neq \mathtt{x}\}$

p_i : `if e`

$Defined(p_i) = \{\}$

$notModified(p_i) =$ all the p_js

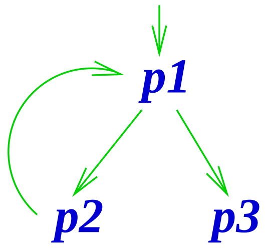
In the example, we are merely calculating the program points reached by p_2:

$$RD(p_1) = \{\} \cup ((RD(p_2) \cap \{p_2\}) \cup \{p_2\}) = \{p_2\}$$

$$RD(p_2) = (RD(p_1) \cap \{p_2\}) \cup \{\} = \{p_2\}$$

$$RD(p_3) = (RD(p_1) \cap \{p_2\}) \cup \{\} = \{p_2\}$$

An example backwards analysis — Very-busy expressions: which expressions must be referenced in the future



p_1: $\texttt{while x} = 0$

p_2: $\quad \texttt{do x} := \texttt{y} + 1$

p_3 : exit

let $\quad e_0 \equiv \texttt{x} = 0$

$\qquad e_1 \equiv \texttt{y} + 1$

e *is very busy at* p if for all paths from p, e is *not Modified* before it is *Used*:

$$VBE(\mathrm{p}) = \mathit{Used}(\mathrm{p}) \cup (\mathit{notModified}(\mathrm{p}) \cap (\textstyle\bigcap_{\mathrm{p}' \in \mathrm{succ}\ \mathrm{p}} VBE(\mathrm{p}')))$$

We must analyze the program backwards, to learn which expressions are referenced in the future, in order to calculate which expressions are very busy at the program points.

$$\begin{aligned}
VBE(\mathrm{p}_1) &= \{e_0\} \cup (\{e_0, e_1\} \cap (VBE(\mathrm{p}_2) \cap VBE(\mathrm{p}_3))) && = \{e_0, e_1\} \\
VBE(\mathrm{p}_2) &= \{e_1\} \cup (\{e_0\} \cap VBE(\mathrm{p}_1)) && = \{e_1\} \\
VBE(\mathrm{p}_3) &= \{\} \cup (\{e_0, e_1\} \cap (\textstyle\bigcap\{\})) && = \{e_0, e_1\}
\end{aligned}$$

Finally, we saw that the information computed by a set of data-flow equations can be presented in different styles when the programming language is *block structured*:

1. as a set of denotational semantics equations: $[\![\mathrm{C}]\!]\mathrm{in} = ...\mathrm{out}$
2. as a set of inference rules in a Hoare logic: $\mathrm{in}\ \mathrm{C}\ \mathrm{out}$
3. as a set of type checking rules: $\mathrm{C} : \mathrm{in} \to \mathrm{out}$

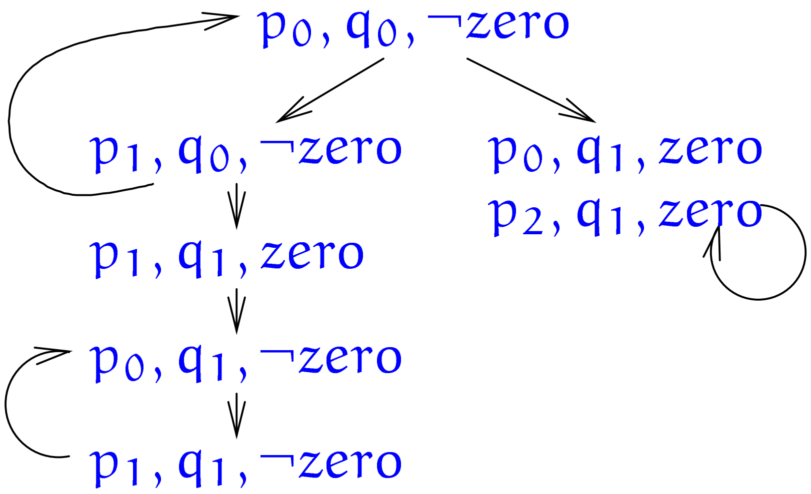
An introduction to model checking

Abstract models can be *model checked*

p_0 : `while x > 0 {`
  p_1 : *use resource*
    `x = x + 1;  }`
p_2 : *sleep forever*

||

q_0 : `x = 0;`
q_1 : *use resource forever*



Starting from p_0, q_0, will *A*ll executions enter p_1 in the *F*uture?

$$p_0, q_0, \neg zero \models AF\, p_1 \ ?$$

Starting from p_0, q_0, will *A*ll executions “*G*enerally/Globally” avoid resource misuse ?

$$p_0, q_0, \neg zero \models AG\, \neg(p_1 \wedge q_1) \ ?$$

Can *A*ll executions reach a *F*uture state where it is possible that x is permanently zero?

$$p_0, q_0, \neg zero \models AF\, EG\, zero \ ?$$

Computation Tree Logic (CTL):

$s \models \phi$ means that logical property ϕ holds true at program state s.

Here is a syntax for a simple version of CTL:

$$q \in \mathit{PrimitivePropositionAboutAState} \qquad \phi \in \mathit{CTL}$$

$$\phi \ ::= \ q \mid \mathrm{EF}\ \phi \mid \mathrm{AF}\ \phi \mid \mathrm{EG}\ \phi \mid \mathrm{AG}\ \phi$$

The semantics of $s \models q$ is immediate, e.g., $p_0, q_0, zero \models zero$ holds true. The semantics of the other constructions depend on

$s \models \Diamond\phi$ iff *there exists* a successor state, s', such that $s' \models \phi$
$s \models \Box\phi$ iff *for all* successor states s', $s' \models \phi$

Then,

$$\begin{aligned}
\mathrm{EF}\ \phi &=_{lfp} \phi \vee \Diamond(\mathrm{EF}\ \phi)\\
\mathrm{EG}\ \phi &=_{gfp} \phi \wedge \Diamond(\mathrm{EG}\ \phi)\\
\mathrm{AF}\ \phi &=_{lfp} \phi \vee \Box(\mathrm{AF}\ \phi)\\
\mathrm{AG}\ \phi &=_{gfp} \phi \wedge \Box(\mathrm{EF}\ \phi)
\end{aligned}$$

The modal-mu calculus assertion language:

$s \in \mathit{FiniteStateSet} \quad \phi \in \mathit{Prop} \quad q \in \mathit{PrimProp} \quad Z \in \mathit{Iden}$

$\phi \ ::= \ q \ | \ \neg q \ | \ \phi_1 \wedge \phi_2 \ | \ \phi_1 \vee \phi_2 \ | \ \Box\phi \ | \ \Diamond\phi \ | \ \mu Z.\phi \ | \ \nu Z.\phi \ | \ Z$

$s \models q$ is given

$s \models \neg q$ iff not $s \models q$

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$

$s \models \phi_1 \vee \phi_2$ iff $s \models \phi_1$ or $s \models \phi_2$

$s \models \Box\phi$ iff for all s' such that $s \longrightarrow s'$, $s' \models \phi$

$s \models \Diamond\phi$ iff there exists s' such that $s \longrightarrow s'$ and $s' \models \phi$

$s \models \mu Z.\phi$ iff there exists $i \geq 0$ such that $s \models \phi_i$, where $\begin{cases} \phi_0 = \mathit{false} \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$

$s \models \nu Z.\phi$ iff for all $i \geq 0, s \models \phi_i$, where $\begin{cases} \phi_0 = \mathit{true} \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$

Examples

Primitive properties, q, are "first-order" properties about state, e.g., "variable $\mathtt{x}$'s value at state s is positive" (written $s \models (\mathtt{x} > 0)$).

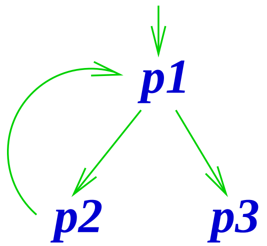
The *modalities* define "second-order" properties, e.g., "there exists a path starting from s such that variable $\mathtt{x}$ has a positive value in two transitions" ($s \models \Diamond\Diamond(\mathtt{x} > 0)$).

Another example: "at all possible next transitions, $\mathtt{x}$ has a positive value": $s \models \Box(\mathtt{x} > 0)$.

The *least fixed-point operator*, μ, defines properties that hold true at some point finitely far into the future, e.g., "at some state in the future, $\mathtt{x}$ will be positive" ($s \models \mu Z.(\mathtt{x} > 0) \vee \Diamond Z$), and the *greatest fixed-point operator*, ν, defines properties that hold true indefinitely, even infinitely, e.g., "from now on, $\mathtt{x}$ is always positive" ($s \models \nu Z.(\mathtt{x} > 0) \wedge \Box Z$).

Using a model check to do data-flow analysis:

We wish to deallocate variable cells when they are definitely not referenced in the future. The property is encoded for a variable `x` as $\nu Z.\ \neg\, Used(\mathtt{x}) \wedge \Box Z$. For the tree,



p_1: `while x = 0`

p_2: `do x := y + 1`

p_3 : exit

the analysis discovers that the property holds only at the exit state, p_3.