Data Flow Analysis

Want to compute some info about program

- at program points, i.e. edges in CFG/DFG/
- to identify opportunities for improving transformations

Can model data flow analysis as solving system of constraints

- each node in graph imposes constraints relating info at
  predecessor and successor points
- solution to constraints is result of analysis

Solution must be safe a.k.a. sound
Solution can be conservative

Key issues:
- how to represent info efficiently?
- how to represent & solve constraints efficiently?
  - how long does constraint solving take? does it terminate?
- what if multiple solutions are possible?
- how do transformations interact with analyses?
- how to reason about whether analyses & transformations
  are sound, i.e., semantics-preserving?

Example: reaching definitions

For each program point in CFG,
want to compute set of definitions (statements) that
may reach that point
- reach: are the last definition of some variable

Info = set of var->stmt bindings
E.g.:
{(x->s1, y->s5, y->s9)}

Can use reaching definition info to:
- build def-use chains
- do constant & copy propagation
- detect references to undefined variables
- present use/def info to programmer
  - ...

Safety rule (for these intended uses of this info):
can have more bindings than the “true” answer,
but can’t miss any

Constraints for reaching definitions

Main constraints:

A simple assignment removes any old reaching def's for the lhs
and replaces them with this stmt:

- strong update
  \[ s; \ x := \cdots : \quad \text{info}_{\text{succ}} = \text{info}_{\text{pred}} - (x \rightarrow s) \cap (s \rightarrow s) \]

A pointer assignment may modify anything, but doesn’t definitely
replace anything

- weak update
  \[ s; \ \ast \ p := \cdots : \quad \text{info}_{\text{succ}} = \text{info}_{\text{pred}} \cup (x \rightarrow s) \cap (\forall x \in \text{may-point-to}(p)) \]

Other statements: do nothing

\[ \text{info}_{\text{succ}} = \text{info}_{\text{pred}} \]

Constraints for reaching definitions, continued

Branches pass through reaching def's to both successors

\[ \text{info}_{\text{succ}}[i] = \text{info}_{\text{pred}}[i], \quad \forall i \]

Merges take the union of all incoming reaching def's

- we don’t know which path is being taken at run-time
  \Rightarrow be conservative

\[ \text{info}_{\text{succ}} = \cup \text{info}_{\text{pred[i]}} \]

Conditions at entry to CFG: “definitions” of formals

\[ \text{info}_{\text{entry}} = (x \rightarrow \text{entry}) \cap (\forall x \in \text{formals}) \]
Solving constraints

A given program yields a system of constraints
Need to solve constraints

For reaching definitions,
can traverse instructions in forward topological order,
computing successor info from predecessor info
• because of how the constraints are defined

Example

Another example

Loop terminology

loop: strongly-connected component in CFG with single entry

loop entry edge: source not in loop, target in loop
loop exit edge: the reverse
back edge: target is loop head node

loop head node: target of loop entry edge
loop tail node: source of back edge
loop preheader node:
single node that’s source of loop entry edge

nested loop: loop whose head is inside another loop

reducible flow graph: all SCC’s have single entry
### Analysis of loops

If CFG has a loop, data flow constraints are recursively defined:

- \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}} \)
- \( \text{info}_{\text{back-edge}} = \ldots \text{info}_{\text{loop-head}} \ldots \)

Substituting definition of \( \text{info}_{\text{back-edge}} \):

- \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup ( \ldots \text{info}_{\text{loop-head}} \ldots ) \)

Summarizing r.h.s. as \( F \):

- \( \text{info}_{\text{loop-head}} = F(\text{info}_{\text{loop-head}}) \)

A legal solution to constraints is a **fixed-point** of \( F \)

Recursive constraints can have many solutions
- want least or greatest fixed-point, whichever corresponds to the most precise answer

How to find least/greatest fixed-point of \( F \)?
- for restricted CFGs can use specialized methods
  - e.g. interval analysis for reducible CFGs
  - for arbitrary CFGs, can use iterative approximation

### Solving constraints by iterative approximation

1. Start with initial guess of info at loop head:
   \( \text{info}_{\text{loop-head}} = \text{guess} \)

2. Solve equations for loop body:
   \( \text{info}_{\text{back-edge}} = F_{\text{body}}(\text{info}_{\text{loop-head}}) \)
   \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}} \)

3. Test if found fixed-point:
   \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-head}} ? \)

   A. if same, then done

   B. if not, then adopt result as (better) guess and repeat:

   \( \text{info}_{\text{back-edge}} = F_{\text{body}}(\text{info}_{\text{loop-head}}) \)
   \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}} \)
   \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-head}} ? \)
   \( \ldots \)

### When does iterating work?

**Sufficient conditions:**

1. need to be able to make an initial guess
2. \( \text{info}^{n+1} \) must be closer to the fixed-point than \( \text{info}^n \)
   (true if \( F_{\text{body}} \) is monotonic)
3. must eventually reach the fixed-point in a finite number of iterations
   (true if info drawn from a finite-height domain)

To reach best fixed-point, initial guess for loop head should be **optimistic**
- easy choice: \( \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \)

(Even if guess is overly optimistic, iteration will ensure we won’t stop analysis until the answer is safe.)
The example, again

1 \ x := \ldots
2 \ y := \ldots
3 \ y := \ldots
4 \ p := \ldots

\ldots \ x \ \ldots
5 \ x := \ldots
\ldots \ y \ \ldots
6 \ x := \ldots
7 \ p := \ldots

\ldots \ x \ \ldots
\ldots \ y \ \ldots
8 \ y := \ldots

Direction of dataflow analysis

In what order are constraints solved, in general?

Constraints are declarative, not directional/procedural, so may require mixing forward & backward solving, or other more global solution methods.

But often constraints can be solved by (directional) propagation & iteration

- may be forward or backward propagation of info

Directional constraints often called flow functions

- often written as functions on input info to compute output
  \[ RD_{x,s} := \ldots (in) = in - \{x \to s' \} \cup \{x \to s\} \]
  \[ RD_{x,s} := \ldots (in) = in \cup \{x \to s \} \cup \{x \to s' \} \]

For greatest solving efficiency:

- analyze acyclic subgraphs in topological order
- analyze loops till convergence before analyzing downstream of loops

GEN and KILL sets

Can often think of flow functions in terms of each’s GEN set and KILL set

- GEN = new information added
- KILL = old information removed

Then
\[ F_{\text{instr}}(in) = in - KILL_{\text{instr}} \cup \text{GEN}_{\text{instr}} \]

E.g., for reaching defs:
\[ RD_{x,s} := \ldots (in) = in - (x \to s') \cup (x \to s) \]
\[ RD_{x,s} := \ldots (in) = in \cup (x \to s) \cup (x \in \text{mpt}(s')) \]

Bit vectors

For efficiency, can sometimes represent info/KILL/GEN sets as bit vectors

- if can express abstractly as set of things (e.g. statements, vars), drawn from a statically known set of things, each thing getting a statically determined bit position
- bitvector encodes characteristic function of set

E.g., for reaching defs:

- \text{info} = bitvector over statements, each stmt getting a distinct bit position
- statement implies which variable is defined

Bit vectors compactly represent sets
Bit-vector operations efficiently perform set difference, union, ...

Flow function may be able to be represented simply by a pair of bit vectors, if they don’t depend on input bit vector

- can merge the KILL and GEN bit vectors of a whole basic block of instructions into a single overall KILL and GEN set, for faster iterating
Another example: constant propagation

What info computed for each program point?

\( I \) is a conservative approximation to “true” info \( I_{\text{true}} \) iff:

Direction of analysis?

Initial info?

\[ \text{CP}_X := \gamma_i^I(n) = \]

\[ \text{CP}_X := \gamma + z^I_i^I(n) = \]

\[ \text{CP}_{SP} := \gamma_{\bigcap}^I + z_{\bigcap}^I_i(n) = \]

Merge function?

Can use bit vectors?

---

Example

\[ x := 5 \]
\[ v := 2 \]

\[ x := x + 1 \]
\[ w := v + 1 \]

\[ w := 3 \]
\[ y := x * 2 \]
\[ z := y + 5 \]

\[ w := w * v \]

---

May vs. must info

Some kinds of info imply guarantees: must info
Some kinds of info imply possibilities: may info
  - the complement of may info is must not info

<table>
<thead>
<tr>
<th>Desired info</th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>small set</td>
<td>small set</td>
<td>big set</td>
</tr>
<tr>
<td>overly big set</td>
<td>overly big set</td>
<td>overly small set</td>
</tr>
<tr>
<td>GEN</td>
<td>add everything that might be true</td>
<td>add only if guaranteed true</td>
</tr>
<tr>
<td>KILL</td>
<td>remove only if guaranteed wrong</td>
<td>remove everything possibly wrong</td>
</tr>
<tr>
<td>MERGE</td>
<td>( \cup )</td>
<td>( \cap )</td>
</tr>
</tbody>
</table>

Another example: live variables

Want the set of variables that are live at each pt. in program
  - live: might be used later in the program
Supports dead assignment elimination, register allocation

What info computed for each program point?

May or must info?

\( I \) is a conservative approximation to \( I_{\text{true}} \) iff:

Direction of analysis?

Initial info, at what program point(s)?

\[ \text{LV}_X := \gamma + z_i^I(n) = \]

\[ \text{LV}_{SP} := \gamma_{\bigcap}^I + z_{\bigcap}^I_i(n) = \]

Merge function?

Can use bit vectors?
Example

```
x := 5
y := x * 2

x := x + 1
y := x + 10

... y ...
```