Global constant propagation

As an example of a general global transformation involving global dataflow analysis.

Recall: To replace a use of `x` by a constant `k`, we must know: *On every path to the use of `x`, the last assignment to `x` is `x := k`*. Let's call this condition AA.

Global constant propagation can be performed at any point where AA holds.

Let's first consider the case of computing AA for a single variable `X` at all program points. One can potentially repeat this procedure for each variable (inefficient but okay; improvements possible by doing all at once -- later).

To make the problem precise, we associate one of the following values with `X` at every program point:

value	interpretation
\top	This statement never executes (or we have not executed it so far)
C	X = constant C
\bot	X is not a constant

\bot is our safe situation; we can always say that X is not a constant (means that we do not know if it is a constant or not).

```
BB1:
               === X = \bot (at this program point)
X := 3
               === X = 3
B > 0 then goto BB2
         else goto BB3
              === X = 3 on both branches

BB2:
             ==== X = 3
Y := Z + W
             ==== X = 3
X := 4
             ==== X = 4

BB3:
             ==== X = 3
Y := 0
             ==== X = 3

BB4:
             ==== X = \bot
A := 2 * X
```

Given global constant information, it is easy to perform the optimization

- Simply inspect the `x = ?` associated with a statement using `x`
- If `x` is a constant at that point, replace all uses of `x` with that constant.
- But how do we compute `x = ?` at each program point.

The analysis of a compicated program can be expressed as a combination of simple rules relating the change in information between adjacent statements.

The idea is to "push" or "transfer" information from one statement to the next.

For each statement `s`, we compute information about the value of `x` immediately before and after `s`

- Define a function `C` that stands for "constant information".
- `C(x,s,in)` = value of `x` before `s`.
- `C(x,s,out)` = value of `x` after `s`.

Define a *transfer function* that transfers information from one statement to another.

In the following rules, let statement `s` have immediate predecessor statements `p1, ..., pn`.

```
        |
        |
        v
---> s <---
```

1. if C(pi, x, out) = \bot for any i, then C(s, x, in) = \bot. For all we know, the execution may come down that predecessor, and so in that case we can make no prediction about the value of x.
2. if C(pi, x, out) = c and C(pi, x, out) = d and c <> d, then C(s, x, in) = \bot. We saw this in the example that we did by hand.
3. If C(pi, x, out) = c or \top for all i, then C(s, x, in) = c. If we come along a path where x=c, this is trivial to see. For a path with \top, it means that this never executes and so it can be ignored.
4. If C(pi, x, out) = \top for all i, then C(s, x, in) = \top. Every predecessor is unreachable, and so x itself is unreachable.

Rules 1-4 relate the *out* of one statement to the *in* of the next statement. Now we need rules relating the *in* of a statement to the *out* of the same statement.

If C(s,x,in)=\top then C(s,x,out)=\top, for all s. Let's call this rule 5.

C(x:=c, x, out) = c if `c` is a constant. This rule (rule 6) has lower priority than the previous rule (rule 5). i.e., this rule applies only if C(x:=c, x, in) = d or \bot.

C(x:=f(...), x, out) = \bot if the value of `x` cannot be determined to be a constant after this statement.

C(y:=..., x, out) = C(y:=..., x, in) if x<>y

Algorithm

- For every entry `s` to the program, set `C(s,x,in) = \bot`
- For every other statement (non-entry), set `C(s,x,in)=C(s,x,out)=\top`.
- Repeat until all points satisfy rules 1-8:
  - Pick `s` not satisfying 1-8 and update using the appropriate rule.

Notice we expect that at some point all rules will be satisfied at all points. This is called the *fixed-point* and the corresponding solution, the *fixed-point solution*.

Let's run the algorithm on this example:

```
BB1:
               === X = \bot (at this program point)
X := 3
               === X = \top
B > 0 then goto BB2
         else goto BB3
              === X = \top

BB2:
             ==== X = \top
Y := Z + W
             ==== X = \top
X := 4
             ==== X = \top

BB3:
             ==== X = \top
Y := 0
             ==== X = \top

BB4:
             ==== X = \bot
A := 2 * X
```

Some guarantees: the fixed-point solution will satisfy the equations at each program point.

Some un-answered questions: what is the guarantee that this analysis will converge? What is the guarantee that this algorithm will result in the best possible solution for this system of solutions?

Analysis of loops

```
BB1:
X := 3
B > 0 then goto BB2
         else goto BB3

BB2:
Y := Z + W
X := 4

BB3:
Y := 0

BB4:
A := 2 * X
```

Assume there is an edge from BB4 to BB3.

Now, when we are computing C(BB3,x,in), we need to know the value of C(BB4,x,out) and in turn C(BB4, x, in) and so on... And we are in a loop (we get back to C(BB3,x,in)).

- Consider the statement `Y:=0`
- To compute whether `X` is constant at this point, we need to know whether `X` is constant at the two predecessors
  - X:=3
  - A:=2*X
- But info for `A:=2*X` depends on its predecessors including `Y:=0`!

Because of cycles, all points must have values at all times.

Intuitively, assigning some initial value allows the analysis to break cycles.

The initial value \top means "So far as we know, control never reaches this point". The initial value is not what is expected at the end but allows us to get going (by breaking the cycle).

Analyzing the example: if we run the algorithm assuming that C(BB4,x,out) is \top, then we will be able to reach a fixed-point solution.

Orderings

We can simplify the presentation of the analysis by ordering the (abstract) values.

```
\bot < c < \top
```

Drawing a picture with "lower" values drawn lower, we get

```
      \top
  /   / | \  \
... -1  0  1 ...
  \   \ | /  /
      \bot
```

Notice that this is a partial order; not all elements are comparable to each other. e.g., 0 and 1 are not comparable.

With orderings defined:

- \top is the greatest value, \bot is the least.
  - All constants are in between and incomparable.
- Let *glb* be the greatest-lower bound in this ordering:
  - glb(\top, 1) = 1
  - glb(\bot, \top) = \bot
  - glb(\bot, 1) = \bot
  - glb(1, 2) = \bot
- Rules 1-4 can be written using `glb`:
  - `C(s,x,in) = glb{C(p,x,out) | p is a predecessor of s}.`

Simply saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes (it could oscillate forever for example). The use of glb explains why the algorithm terminates:

- Values start as \top can only decrease.
- \top can change to a constant, and a constant to \bot.
- Thus, `C(s,x,_)` (for every statement, for every variable) can change at most twice.

Also, we maintain the invariant that the value at any intermediate point of the algorithm is always *greater* than the best solution value. This is because we initialize all values to \top. Also, we relax minimally --- assuming this invariant is true for the predecessors, it is guaranteed to be true for the successor.

Thus the constant propagation algorithm is linear in program size. Number of steps per variable = Number of C(...) values computed * 2 = Number of program statements * 4 (two C values per program statement, in and out).

Liveness analysis

Once constants have been globally propagated, we would like to eliminate dead code.

```
BB1:
X := 3
B > 0 then goto BB2
         else goto BB3

BB2:
Y := Z + W

BB3:
Y := 0

BB4:
A := 2 * 3
```

After constant-propagating X at BB4, the assignment to X in BB1 is no longer useful. In other words, X:=3 is dead (assuming X not used elsewhere).

Defining what is used (live) and what is not used (dead).

```
X := 3
X := 4
Y := X
```

- The first value of `x` is *dead* (never used).
- The second value of `x` is *live* (may be used).
- Liveness is an important concept.

A variable `x` is live at statement `s` if

- There exists a statement `s'` that uses `x`.
- There is a path from `s` to `s'`.
- That path has no intervening assignment to `x`.

A statement `x:=...` is dead code if `x` is dead after the assignment. Dead statements can be eliminated from the program. But we need liveness information first . . .

We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation.

Liveness is simpler than constant propagation, since it is a boolean property (true or false).

Here the set of values is False (definitely not live) and True (may be live). False > True. The `glb` function is the boolean-OR function in this set of values.

Rule 1: `L(p,x,out) = OR{L(s,x,in) | s is a successor of p}`. `x` is live at `p` if `x` is live at one of the successor nodes of `p`.

Rule 2: s: ... := f(x). L(s,x,in) = true if `s` refers to `x` on the RHS.

Rule 3: s: x := e where `e` does not refer to `x`. `L(x:=e,x,in)=false` if `e` does not refer to `x`. `x` is dead before an assignment to `x` because the current value of `x` at that point will not be used in the future.

Rule 4: `L(s,x,in) = L(s,x,out)` if `s` does not refer to `x`.

Algorithm

1. Let all `L(...) = false` initially.
2. Repeat until all statements `s` satisfy rules 1-4
    - Pick `s` where one of 1-4 does not hold and update using the appropriate rule.

Example:

```
    ==== L(x) = false
x := 0
    ==== L(x) = false
while (x!= 10) {
                ==== L(x) = false -> true (step 1)
  x = x + 1;
    ==== L(x) = false
}
    ==== L(x) = false
return;
    ==== L(x) = false

    ==== L(x) = false
x := 0
                ==== L(x) = false -> true (step 3)
while (x!= 10) {
                ==== L(x) = false -> true (step 1)
  x = x + 1;
    ==== L(x) = false -> true (step 2)
}
    ==== L(x) = false
return;
    ==== L(x) = false
```

- A value can change from `false` to `true`, but not the other way round. So we use the ordering false > true
- Each value can change only once, so termination is guaranteed.
- Once the analysis is computed, it is simple to eliminate dead code.

Notice that information flowed in the forward direction (in the direction of the program execution) for constant propagation but flowed in the reverse direction (against the direction of the program execution) for liveness analysis. The former types of analyses are called *forward dataflow analyses*. The latter types of analyses are called *backward dataflow analyses*.

Some other analyses that can be modeled as dataflow analyses

- Common-subexpression elimination: maintain a set of *available expressions* of the form x = op(y,z) at each statement. If we see another expression of the type `u = op(y,z)`, replace by `u=x`. Available expressions is a forward data-flow analysis. A value is available at a program location only if it is available at all the predecessor program locations, i.e., the available expressions at a statement is the intersection of the available expressions at the predecessors. In other words, the *meet* operator is intersection. Works best with single-assignment form because can maintain a larger set of available expressions.
- Copy-propagation: an analysis similar to available-expressions, except only for expressions that are of the form x=y. The main difference is in the transformation (substituting variable names vs. substituting expressions by variable names).