Kildall's Generalization of DFA

- Generalizes DFA problems previously discussed.
- The principles lead to some more powerful DFA schemes.

The DFA Framework

There are three essential ingredients.

1. A set V of *values* to be propagated. For example, V = the set of sets of definitions in the RD problem.
2. A set F of *functions* from V to V. These are the transfer functions associated with blocks. For example, in RD, F is the set of functions of the form

   $$f(S) = (S - K) \cup G$$

   for any sets K and G in V.
3. A binary *meet* operation, $\wedge$, on V, to represent the effect of confluence of paths. In RD, $\wedge$ is union.

Axioms

To make the algorithms of DFA work, we need some assumptions about how V, F, and $\wedge$ behave and interact. The most fundamental assumptions are

1. F is closed under composition.
2. F includes the identity function.
3. $\wedge$ is

   a) Commutative: $a \wedge b = b \wedge a$.
   b) Associative: $a \wedge (b \wedge c) = (a \wedge b) \wedge c$.
   c) Idempotent: $a \wedge a = a$.

   That is, V with operation $\wedge$ is a *semilattice*.
4. There is a *top element* T in V such that

   $$T \wedge a = a$$

   for all a.

Intuition

- Elements of V represent information, for example, a set of definitions that we believe reach a certain point.
- $\wedge$ tells us how information combines when we reach the confluence of two paths. For example, reaching definition sets combine by taking the union of the sets.
- T represents "no information." When it meets value a, representing any information whatsoever, the combination of a with T yields a.

Example of a Theorem

To see how the axioms can be used formally, let us consider the proof of a simple theorem: "Top is unique."

Proof: Suppose there were two "tops," T and S. Since $T \wedge a = a$ for any a, it follows that $T \wedge S = S$.

Similarly, $S \wedge a = a$ for any a, so $S \wedge T = T$.

By commutativity, $S \wedge T = T \wedge S$. Thus,

$S = T \wedge S = S \wedge T = T$

Example: Available Expressions

- V = the set of sets of expressions computed by statements in the program.
- F = the set of functions of the form

  $f(S) = (S - K) \cup G$

  where K and G are in V.

  → The identity function corresponds to

  $$K = G = \emptyset$$

  → Composition of $f(S) = (S - K_1) \cup G_1$ with $g(S) = (S - K_2) \cup G_2$ is

  $$\begin{aligned} g\big(f(S)\big) &= \big[\big(S - K_1\big) \cup G_1\big) - K_2\big] \cup G_2 \\ &= \big(S - (K_1 \cup K_2)\big) \cup \big(G_2 \cup (G_1 - K_2)\big) \\ &= (S - K) \cup G \end{aligned}$$

  where $K = K_1 \cup K_2$ and

  $$G = G_2 \cup (G_1 - K_2)$$

- $\wedge$ = intersection. Surely the laws

  a) Commutativity: $a \cap b = b \cap a$,
  b) Associativity: $a \cap (b \cap c) = (a \cap b) \cap c$, and
  c) Idempotence: $a \cap a = a$

  hold. The top element T is the set of all expressions, since then $T \cap a = a$ for any a in V.
- Notice that "information" is of the form "I definitely know that this expression is unavailable here." T, which is all expressions, represents a state in which we have not ruled out the availability of any expressions.
- Subtle point: AE is anomalous in that we artificially make all expressions unavailable on entrance to the header by "killing" all.

Example: Range Computation

This framework is not like the bit-vector frameworks considered so far. V is much more complex than a set of subsets of "all expressions" or "all definitions."

- We want to associate with each variable `A`, at each point in the program, a range $[A_{min}, A_{max}]$ such that the value of A at that point is known to lie between A_{min} and A_{max}.
- Let a value in the set V be a set of limits of the form $\texttt{A} \leq c$ or $\texttt{A} \geq c$, for some variable `A` and constant c. No variable has more than one upper limit or more than one lower limit in the set.

- Subtle point: the interpretation of $\texttt{A} \leq c$ is that we have found reason to believe that, at the point in question, `A` can have values up to c, but no reason to believe it can have higher values. If we have not seen any values for `A` at all, the range of `A` is *not* $[-\infty, \infty]$; it is no range at all. We represent the absence of a range by having *no* entry for `A`.
- Application: Insert range checks for all array accesses. Use range computation to deduce that certain indices are within range for the array, thus allowing the inserted checks to be deleted. Those that remain may be very useful!

Aside: Modifying the Flow Graph to Take Advantage of Comparisons

In our DFA framework to find ranges of (integer-valued) variables, we shall add blocks to the flow graph and invent a new kind of intermediate statement.

A statement of the form $[[A \leq c]]$ or $[[A \geq c]]$ is an *assertion*. Here, c can be a constant or another variable. It has no effect on the computation of the program, but will affect the values IN and OUT that we compute during data-flow analysis.

- We use assertions to indicate the result of taking a branch after a comparison.

Example

Consider the following simple loop.

```
i := 1

a[i] := 0
i := i+1
i ≤ 100?
```

Presumably, when a ≤ 100, we go around the loop, and we fall through when a ≥ 101. Thus, we can add boxes for assertions and obtain the following flow graph.

	`i := 1`	B_1
$[[i \leq 100]]$		B_2
	`a[i] := 0` `i := i+1`	B_3
	$[[i \geq 101]]$	B_4
		B_5

The Set of Transfer Functions

The set F is rather hard to describe formally. First, let us discuss only the subset of F that corresponds to blocks with a single statement. The full set F is the closure of the basis under composition.

Each function f in F turns each set of bounds S into another set of bounds $T = f(S)$.

- Remember, S and T are assumed to have at most one upper and one lower bound for any variable.

We consider the various types of intermediate statements in turn.

1. The identity is in F, that is the function $f(S) = S$ for all sets of bounds S.
2. The function corresponding to an assignment `A := c` is in F. That is, T is S with any bounds for `A` removed and the bounds `A` $\leq c$ and `A` $\geq c$ added.

---

3. The function corresponding to an assignment `A := B` for variable `B`. To construct T, start with S, but remove any bounds for `A`. If there is a bound `B` $\leq c$ in S, then we add `A` $\leq c$ to T; likewise for a lower bound on `B`.

- Notice that if `B` is undefined, that is, it has no bounds, then `A` becomes undefined.

4. The function corresponding to **read** `A`. To construct T, start with S and remove any bounds for `A`. Then add the bounds `A` $\leq \infty$ and `A` $\geq -\infty$ to T.
5. Functions corresponding to the arithmetic operations. For example, consider the assignment `A := B+C`. Construct T by starting with S, removing any bounds for `A`.
   a) If there is a bound `B` $\leq b$ and a bound `C` $\leq c$ in S, then add bound `A` $\leq b + c$ to T.
   b) If either `B` or `C` have no upper bound in S, i.e., they are undefined, then `A` has no upper bound in T.

   Lower bounds are handled analogously.

---

6. Functions corresponding to assertions. For example, consider the assertion $[[A \leq c]]$. T is S with the following modifications.
   a) If `A` has a lower bound `A` $\geq e$ in S, such that $e > c$, then in T, there is no lower or upper bound for `A`. That is, we have found no value `A` can have at this point.
   b) Otherwise, if `A` has an upper bound `A` $\leq d$ in S, then in T, the upper bound for `A` is `A` $\leq \min(c, d)$.
   c) If `A` has no upper bound in S, then `A` has no upper bound in T.

   Lower bounds are handled analogously.

- For ambiguous assignments, e.g., `A := *P`, give `A` a range that covers all the ranges of the variables that `P` could point to.

Confluence Operator

What is the appropriate value for $S \wedge T$? Recall that `A` $\leq c$ says we have reason to believe that `A` can have values up to c but have seen no higher values. Thus,

1. If S has `A` $\leq c$ and T has `A` $\leq d$, set $S \wedge T$ should have `A` $\leq \max(c, d)$.
2. If one of S and T has `A` $\leq c$, and the other has no upper bound for `A`, then $S \wedge T$ has `A` $\leq c$.
3. If neither S nor T has an upper bound for `A`, then neither does $S \wedge T$.

Lower bounds are handled analogously.

A Simple Data-Flow Analysis Algorithm

We can generalize the DFA solutions that involve bit vectors. Start with an initial set of bounds for integer variables, and repeatedly apply the confluence operator and transfer functions of the blocks until convergence.

The simplest way to start out is to assume IN[B] gives each variable the range $[-\infty, \infty]$ for all blocks B, and to apply the transfer function for the block to get an initial value of OUT[B].

Example

For the loop of our previous example, we proceed in rounds as follows.

- We concern ourselves only with the value of `i`, which we give as a range, rather than an upper and lower bound.

	IN	OUT	IN	OUT	IN	OUT
B_1	$[-\infty, \infty]$	$[1,1]$	$[-\infty, \infty]$	$[1,1]$	$[-\infty, \infty]$	$[1,1]$
B_2	$[-\infty, \infty]$	$[-\infty, 100]$	$[-\infty, \infty]$	$[-\infty, 100]$	$[-\infty, 101]$	$[-\infty, 100]$
B_3	$[-\infty, \infty]$	$[-\infty, \infty]$	$[-\infty, 100]$	$[-\infty, 101]$	$[-\infty, 100]$	$[-\infty, 101]$
B_4	$[-\infty, \infty]$	$[101, \infty]$	$[-\infty, 101]$	$[101, 101]$	$[-\infty, 101]$	$[101, 101]$

- This analysis correctly deduces that `i` has the value 101 on exit from B_4.
- It also deduces that `i` ≤ 100 on entry to B_3, which is useful because it means a range check to see that `i` has not exceeded the upper limit of the array `a` can be eliminated.
- However, we have failed to deduce the lower limit `i` ≥ 1 on entry to B_3, so we cannot eliminate the check for the lower limit on `a`.

What has gone wrong is that we started assuming not "zero information," but "total information." That is, we assumed we had seen all possible values of `i` on entry to each block, when in fact we had no reason, a priori, to assume that `i` was even defined.

- That is analogous to assuming initially that all definitions reach the beginning of each block in the RD problem.
- In RD, spurious definitions can then propagate themselves around loops. Here, our initial spurious assumption that we knew of values for `i` like -234 on entry to B_2 and B_3 propagates itself around the loop.

What is the Top Element?

Recall that the top element of V is the one that provides "no information," i.e., its meet with any element is that other element.

- In DFA problems seen previously, we start by assuming the top element enters each block. For example, in RD, top is the set of no definitions.
- But the set S that gives every variable the range $[-\infty, \infty]$ is *not* the top element. Just the opposite, $S \wedge X = S$ for any set of bounds X.
- For the range checking DFA framework, the top element is $\emptyset$. Note that $\emptyset \wedge X = X$ for any set X.

Thus, we really should have initialized IN[B] to $\emptyset$ in our previous example. That "works," but convergence is slow!

	IN	OUT	IN	OUT	IN	OUT
B_1	$\emptyset$	$[1,1]$	$\emptyset$	$[1,1]$	$\emptyset$	$[1,1]$
B_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$[2,2]$	$[2,2]$
B_3	$\emptyset$	$\emptyset$	$[1,1]$	$[2,2]$	$[1,2]$	$[2,3]$
B_4	$\emptyset$	$\emptyset$	$[2,2]$	$\emptyset$	$[2,3]$	$\emptyset$

B_1	$\emptyset$	$[1,1]$			$\emptyset$	$[1,1]$
B_2	$[2,3]$	$[2,3]$	$\cdots$	$\cdots$	$[2,100]$	$[2,100]$
B_3	$[1,3]$	$[2,4]$			$[1,100]$	$[2,101]$
B_4	$[2,4]$	$\emptyset$			$[2,101]$	$[101,101]$

Now, we get all the useful information: `i` has value 101 on exit from B_4, and the range of `i` at the statement `a[i] := 0` is $[1, 100]$, which is exactly what we hoped it would be.

A Revised Transfer Function to Speed Convergence

A useful trick is to assume that when we go through an assertion like $[[i \leq 100]]$, we have "reason to believe that" values of `i` up to 100 will eventually be seen. We can revise the function f in F that is associated with an assertion $[[A \leq c]]$, as follows.

Construct $T = f(S)$ by deleting any upper bound for `A` in S and inserting `A` $\leq c$ into T (unless there is a conflicting lower bound). Otherwise, T is the same as S, including any lower bound for `A`.

Lower bound assertions are treated similarly.

- Note that now a set can have an upper bound and no lower bound for a variable, or vice versa.

Example

For our running example we get

	IN	OUT	IN	OUT	IN	OUT
B_1	$\emptyset$	$[1,1]$	$\emptyset$	$[1,1]$	$\emptyset$	$[1,1]$
B_2	$\emptyset$	`i` ≤ 100	$\emptyset$	`i` ≤ 100	$[2,101]$	$[2,100]$
B_3	$\emptyset$	$\emptyset$	$[1,100]$	$[2,101]$	$[1,100]$	$[2,101]$
B_4	$\emptyset$	`i` ≥ 101	$[2,101]$	$[101,101]$	$[2,101]$	$[101,101]$

We get all the useful information about `i`, but convergence is much faster.

Monotonicity

For an iterative solution like those discussed to make sense, we need an additional axiom, called *monotonicity*.

As a preliminary, we need to define a partial order $\leq$ on V. Say $a \leq b$ if $a \wedge b = a$.

We can show $\leq$ is a partial order on V. For example, $\leq$ is transitive: $a \leq b$ and $b \leq c$ imply $a \leq c$.

Proof: $a \wedge b = a$ and $b \wedge c = b$ are given. Then

$$\begin{aligned} a \wedge c &= (a \wedge b) \wedge c \text{ (substitution for } a) \\ &= a \wedge (b \wedge c) \text{ (associativity)} \\ &= a \wedge b \text{ (substitution for } b \wedge c) \\ &= a \text{ (substitution for } a \wedge b) \end{aligned}$$

We say a framework is *monotone* if for every f in F and a and b in V,

$a \leq b$ implies $f(a) \leq f(b)$

Intuitively, when you feed f more information, it doesn't return less.

Equivalently, the framework is monotone if for all a, b, and f

$f(a \wedge b) \leq f(a) \wedge f(b)$

Example: Reaching Definitions

For RD, $\wedge$ is $\cup$, so $a \wedge b = a$ means a is a superset of b. Thus, $a \leq b$ means a is a *larger* set of definitions than b!

- The traditional interpretation of $\leq$ is "less ignorance," i.e., more information. That holds true here, if we regard information as definitions known to reach a certain point.

RD is monotone; in fact it obeys the stronger *distributivity* condition

$f(a \wedge b) = f(a) \wedge f(b)$

Proof: Let $f(S) = (S - K) \cup G$.

$$\begin{aligned} f(a \cup b) &= \big((a \cup b) - K\big) \cup G \\ &= \big((a - K) \cup G\big) \cup \big((b - K) \cup G\big) \\ &= f(a) \cup f(b) \end{aligned}$$

Example: Available Expressions

For AE, $\wedge$ is $\cap$, and $a \wedge b = a$ means that a is a subset of b. Thus, $\leq$ is set containment.

The functions for AE are of the same form as for RD and are easily shown distributive with respect to intersection.

Example: Range Computation

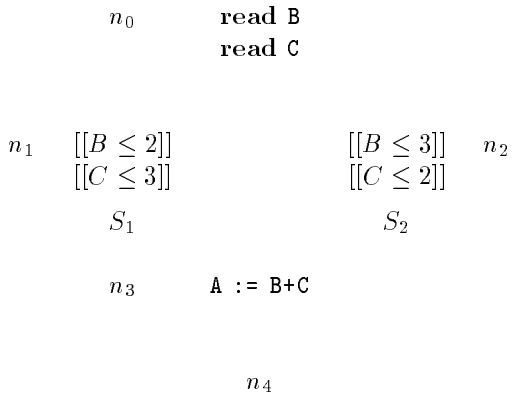
Under what conditions is $S \wedge T = S$, where S and T are sets of bounds? For every bound $\mathtt{A} \leq c$ in T, there must be a bound $\mathtt{A} \leq d$ in S, where $c \leq d$. Lower bounds are analogous.

That is, S gives at least as wide a range to $\mathtt{A}$ as T does. S may have a bound for T, even in situations where T does not know anything about $\mathtt{A}$.

- Thus, $S \leq T$ again means "less ignorance," or more possible values for more variables are known in S than in T.

For either of the two sets of functions F we discussed, RC is monotone. No proof is offered, but the intuition about wider ranges should help.

However, RC is not distributive. Consider the following flow graph.



Let S_1 be the set of bounds coming out of n_1, and S_2 be the set coming out of n_2. Then

$$S_1 = \{\mathtt{B} \leq 2, \mathtt{C} \leq 3\}$$

and $S_2 = \{\mathtt{B} \leq 3, \mathtt{C} \leq 2\}$ (we ignore lower bounds, which are all $-\infty$).

Then coming into n_3 we have

$$S_3 = S_1 \wedge S_2 = \{\mathtt{B} \leq 3, \mathtt{C} \leq 3\}$$

Let f be the transfer function for n_3. Then $f(S_3)$ has bound $\mathtt{A} \leq 6$.

However, $f(S_1)$ and $f(S_2)$ both have $\mathtt{A} \leq 5$, so $f(S_1) \wedge f(S_2)$ does also. Thus,

$$f(S_1 \wedge S_2) \neq f(S_1) \wedge f(S_2)$$

Fortunately, the discrepancy is in the right direction for monotonicity; we have $f(S_1 \wedge S_2) \leq f(S_1) \wedge f(S_2)$, since the former gives a broader range to $\mathtt{A}$ than does the latter.

DFA Problem Instances

Given a DFA framework $(V, F, \wedge)$, an *instance* is a flowgraph with an assignment of a (transfer) function in F to each node.

- We use f_n for the function assigned to node n.

Functions for Paths

If $P = n_0 \rightarrow n_1 \rightarrow \cdots \rightarrow n_k$ is a path from the start node n_0, the *function associated with* path P is the composition of the functions associated with all but the last node on the path. That is,

$$f_P = f_{k-1} \circ f_{k-2} \circ \cdots \circ f_0$$

Note that the last node has no effect; intuitively, f_P tells us about what happens when we enter the block corresponding to the last node.

Meet-Over-Paths

It turns out that the proper generalization of the standard DFA frameworks is to compute for each node n the value MOP(n) equal to the meet over all paths P from n_0 to n, of $f_P(Top)$, where Top is the top element of V.

Example: Reaching Definitions

For RD, the function f_P gives the contribution of path P, that is, the set of definitions generated along that path and not subsequently killed.

Thus, MOP(n) is exactly the set of definitions we have been computing.

Example: Range Computations

For RC, f_P is a set of ranges for variables that includes all the values for variables that are ever computed along path P, but may include other values that cannot be computed.

Computing the MOP

It turns out that when the framework is distributive, it is easy to compute the MOP by an iterative algorithm like those we have been using.

However, when the framework is only monotone, the best we can do is compute some solution IN such that IN(n) $\leq$ MOP(n) for all nodes n.

Safety of Solutions $\leq$ MOP

The set of paths from the start node to node n can be broken into two (possibly infinite) sets: the *real* paths and the *apparent* paths.

Real paths can be taken on some execution of the program, while apparent paths are graph-theoretic paths that happen not to be traversable by any execution of the program.

The meet of f_P over any set S of paths is always $\leq$ the meet taken over a subset of S. That is because

$$a_1 \wedge \cdots \wedge a_m \wedge b_1 \wedge \cdots \wedge b_k \leq a_1 \wedge \cdots \wedge a_m$$

- We want the meet over real paths only, but have been content with the meet over both real and apparent paths in all DFA applications discussed.
- The MOP is the meet over real and apparent paths, and is therefore $\leq$ the true solution.
- If a solution X $\leq$ the true solution is safe, then something $\leq$ X must also be safe, although less "accurate."

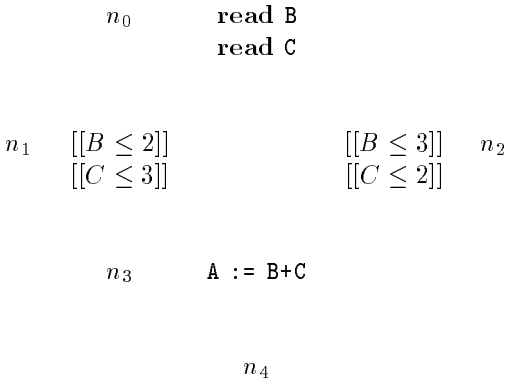
General Iterative Algorithm

```
for all nodes n do
    OUT(n) := f_n(Top);
while changes to any OUT occur do
    for all nodes n do begin
        /* use depth-first order? */
        IN(n) := meet, over all prede-
          cessors p of n, of OUT(p);
            /* meet over ∅ is Top */
        OUT(n) := f_n(IN(n))
    end
```

- The above algorithm need not converge (the while-loop could be infinite). If it does converge, and the framework is monotone, then IN(n) $\leq$ MOP(n) for all n.
- If the DFA framework is distributive and converges, then IN(n) = MOP(n).

Example Where IN Differs from MOP

Consider the RC instance discussed earlier,



We saw that on entrance to n_4, we shall only know A ≤ 6, while along both paths, and therefore, in the MOP solution, we know A ≤ 5.

Type Determination in Typeless Languages

Languages that do not require type declarations, e.g., Lisp, Prolog, are becoming more important.

Serious attempts to compile such languages into efficient code use DFA to infer types of variables. For example, code to add two integers is much more efficient than code to add objects of unknown and arbitrary type.

Our first guess might be that computing types is something like computing reaching definitions. We associate a set of types with each variable at each point.

The confluence operator is union on sets of types, since if variable `A` has set S_1 of possible types along one path and set S_2 of possible types along another, then `A` can have any of the types in $S_1 \cup S_2$ after confluence of the paths.

As control passes through a statement, we may be able to make some inferences about types of variables, based on the operator used in the statement and the possible types of arguments and result for that operator.

Unfortunately, there are at least two problems with that approach.

1. The set of possible types for a variable may be infinite.
2. Type determination usually requires both forward and backward propagation of information to obtain precise, safe estimates of the types of variables at points in the program.

We shall give only the sketchiest comments about these points. Much more is found in the Dragon Book.

Dealing With Infinite Type Sets

A language like SETL allows a statement like

```
A := {A}
```

to appear in a loop. At this point, `A` can assume any of the types integer, set of integers, set of sets of integers, and so on.

To handle this infiniteness in the type sets, we normally group types into a finite number of classes, with simpler types, e.g., integer and set-of-integers, by themselves, and more complex types grouped into one class, e.g., "set-of-thingees."

Example of Forward-Backward Inference

Consider the statements

```
I := A[J]
K := A[I]
```

Suppose at first we do not know anything about the types of variables `A`, `I`, `J`, or `K`. However, let us suppose that the array accessing operator `[]` requires an integer argument.

By examining the first statement, we infer that `J` is an integer at that point, and `A` is an array of elements of some type.

Then, the second statement tells us `I` is an integer there.

Now, propagate inferences backwards. If `I` was computed to be an integer in the first statement, then the type of the expression `A[J]` must be integer, so `A` must be an array of integers.

We can then reason forward again, to discover that the value assigned to `K` by the second statement must also be an integer.

- Note it is impossible to discover that elements of `A` are integers by reasoning only forwards or only backwards.