The use of value numbers in static analysis.

By Tucker Taft, Director of Language Research, AdaCore.

CodePeer uses value numbers as part of its static analysis. This is one of the keys to its power and its flexibility. This is a short note that explains how value numbers work in the context of CodePeer, and what advantages they provide.

Value Numbering process in CodePeer

Value numbering is a relatively old technique used in compiler optimizers, as a way of assigning a unique identifier to each computation in a function based on the value it will have at run-time. The basic rule is that if two computations are assigned the same value number, then they will produce the same value at run-time. The converse is not guaranteed; in other words, if two computations have different value numbers they might still compute the same value at run-time.

The original value-numbering technique was generally restricted to a single basic block within the control flow graph. Later, by first doing a static-single assignment (SSA) pass over the control flow graph, global value numbering became straightforward, where a single value numbering was applied to all of the computations throughout the function, covering multiple basic blocks. SSA introduces "phi" nodes, which are pseudo-operators which combine value numbers from multiple predecessors to a given basic block into a single value number, whose value reflects the particular path used to enter the basic block.

The initial SSA approaches were themselves limited to simple scalar variables, as they did not take aliasing into account. In later approaches, and in particular in CodePeer, SSA accomodates aliasing. In CodePeer this is done by introducing another pseudo-operator called a "Kappa" node (which originally stood for "kill"). They are introduced at points where an assignment to a given object might affect ("kill") the value of another distinctly named object. Whether or not the two objects are actually aliases is generally not known, but clearly the value of the possible alias is either its original value, or the new value assigned to the given object. A Kappa node captures these two possibilities, very much the way a Phi node can capture two possible paths entering a block. The Kappa node goes a bit further, by recording a conditional expression that determines whether the new value or the old value will be in the possible alias. This expression is generally equivalent to the equality comparison "Addr(Assigned-obj) == Addr(Potential-alias)", which if True means the possible alias has the new value, and if False means the possible alias preserves its old value. In some sense a Kappa node is simply capturing a small bit of control flow without the overhead of creating multiple basic blocks and multiple Phi nodes.

In CodePeer the Kappa nodes have been generalized to handle any sort of conditional expression, so what in C would be "X? Y: Z" can become a Kappa node where the condition is "X", the "new" value is "Y", and the "old" value is Z.

Note that in the presence of cycles in the control flow graph, some Phi nodes actually represent a *sequence* of values, rather than a single value. On each iteration of the loop, the Phi node takes on a new value in the sequence, generally as a function of its old value. Because of this, one must be careful when dealing with Phi nodes that one doesn't presume the value represented by a Phi on one iteration is necessarily the same as its value on some prior iteration. For example, in a simple loop like:

```
X := 1;
while X <= 10 loop
   X := X + 1;
   if X = Y then
      Print(X);
   end if;
end loop;
```

This would generate a Phi node for X of roughly:

```
X = phi(1, X + 1)
```

Clearly if somewhere along the way we assume X = X+1, we are in for trouble. Phi nodes that represent a sequence of values are generally called "sequence variables."

Usefulness of Value Numbers for static analysis

In CodePeer, rather than trying to track the values of variables, we track the "possible value set" of individual value numbers. What this means is that each distinct value number is treated separately and uniformly. That means that each value number associated with each of the expressions "X + Y", "Z", "A - B", "Y ** 2", etc., is treated distinctly, as though it were its own "variable." And of course "X + Y" and "Z" might have the same value number, and if so, there is only one rather than two distinct entities to track.

But in general value numbers don't change in value, so what does it mean to "track" the value of a value number? What changes is what we *know* about the possible run-time value of the value number. This is determined by two things, one are run-time checks, and the other are conditional jumps.

When we "check" the value of a value number at run-time, such as whether an index is within the bounds of an array, we are effectively restricting the range of the possible values of the value number. If the value of the value number is outside the allowed

range, an exception will be raised (if the run-time checks are present), or the computation will become undefined (if there are no run-time checks present). In any case, for subsequent uses in the same basic block, we may assume that only the "allowed" values are possible. If in fact the value might be out of range at the point of the check, the static analysis tool would (eventually) complain at that point (in its message-emitting phase).

Alternatively, when there is a conditional jump from the end of one basic block to the beginning of another, we know that along that path, the associated condition is True. That condition is itself a value number, and so we know the value of that value number is restricted to True. Knowing that, and knowing how that value number is defined (we keep track of what computation a value number represents in a "computation table"), we can propagate that restriction to "True" down to the constituents that make up the value number. For example, if the condition were "X in {0..inf}" then we know that the truth of that condition implies the value number "X" is in 0..inf along that path. At the beginning of a basic block, we union the possible-value-sets of each value number along each of the incoming paths, to produce a new possible-value-set for each value number in the new basic block.

Canonicalization

Since we are tracking the possible-value-sets of each value number, it is important to reduce the number of value numbers as much as possible. This brings up the issue of "canonicalization." If we know a particular operator is commutative, such as "+", then we want to treat "X+Y" and "Y+X" equivalently. So we come up with a canonical order for the operands of commutative operators, such as always making the operand with the lower value number the left operand, and the one with the higher value number the right operand. Note that "lower" and "higher" here just means the value number itself, not anything about its possible value at run time. In general value numbers are assigned sequentially as you proceed through the function, so "lower" would mean assigned earlier.

Another very important canonicalization performed by CodePeer is to change all numeric comparison operators into subtractions and a test for membership in a numeric set. For example, A >= B becomes A-B in {0..inf}. This makes it now trivial to union possible-value-set information we have about "A-B". We also factor out constant terms, and embed them in the set, so "A <= B+4" becomes "A-B in {-inf..4}" Now if we want to combine "A >= B" from one conditional jump/check with "A <= B+4" on a subsequent jump/check, we simply intersect the sets to produce "A-B in {0..4}". Alternatively if these checks are on two different paths into the same block, then we would be unioning, to produce "A-B in {-inf..+inf}". We also canonicalize the order of subtractions, even though "-" is not commutative, so that if we have "B >= A+2" that would become "-(A-B) in {2..+inf}" and then factoring out the "-" it would become "A-B in {-inf..-2}". So now it is easy to combine with the earlier bits of information we have gathered on "A-B".

One interesting result of this canonicalization of comparisons into subtractions is that the transitivity law reduces to a simple addition rule. If we know A >= B and B >= C, and we express these as "A-B in {0..inf}" and "B-C in {0..inf}" then if we simply add these two membership conditions we get "A-C in {0..inf}", or A >= C.

Propagation of information

As mentioned above, we keep a computation table which defines each value number in terms of its constituent value numbers. This is used during value-numbering to ensure that if the same computation (or an equivalent computation after canonicalization) is found, it will be given the same value number. But we also use the computation table to propagate information we learn about one value number to another. In particular if because of a check or a jump we learn that some value number X is in {1..10}, then we can look at how it is defined (e.g. as "A-B") as well as how it is used as a constituent of other value numbers (e.g. "X+Y"), and propagate this new smaller value set for X to other value numbers, giving us smaller (i.e. "tighter") bounds on the value numbers that are related to it. In general the goal is to shrink the size of the possible value sets, since the smaller are these sets, the more we know about whether a given check might fail, or whether a given jump is "dead."

Inter-procedural analysis

CodePeer does sophisticated inter-procedure analysis, also based on these value numbers. For every input to a function, including parameters, globals, components reached through pointers, etc., there is an "initial" value number assigned (actually called an "External VN" in CodePeer). For every output of a function (including OUT parameters, results, globals, and components reached through pointers, etc.), a "final" value number is computed. This final value number is in the computation table, and so it is defined in terms of other (intermediate) value numbers of the function, but ultimately will likely depend on some initial value numbers. The computation-table graph defining these final value numbers essentially defines the semantic effect of the function.

When CodePeer encounters a call on a function for which there is already a computation table linking final value numbers back to initial value numbers, it does the substitution. That is, for each of the outputs of the function call, a computation tree is built up by substituting into the initial value numbers the pre-call values of the inputs, to produce the post-call values of the outputs. This process uses the caller's computation table and the usual canonicalizations, so the net effect is a set of value numbers representing the result of the function call, as though the call had been inlined.

One limitation in this process of "inlining" function calls in CodePeer is that no new basic blocks are created. Outputs whose final values are defined by Phi nodes from inside the called function cannot be directly imported, as they depend on the control flow graph of the called function. CodePeer first attempts to convert all relatively simple Phi nodes into Kappa nodes, though that is not possible for Phi nodes representing sequence

variables (that is, that involved a loop). If the Phi node is simply too complex or involves a loop, then CodePeer creates a special "Call" value number, which has as its inputs all of the inputs to the function, but otherwise loses track of the actual computation involved. However, such a Call node is still uniquely defined, and if the same function is called again with the same value numbers as inputs, then the same Call nodes will result as outputs. Hence, CodePeer can handle complex functions such as "Sin(X)" and still learn something from "if Sin(X) > 0 then" to the extent of knowing that if within the "then" part Sin(X) is called again on the same value number X, it will produce a positive answer.