C2 IR Graph and Nodes

Input edges

Node inputs (use-def edges) is an ordered collection. Every node has a control input that has index 0. The value of it can be null, in which case it is not dependent on any control. The control edge is typically referenced by its index **node->in(0)**, however many nodes define a constant to refer to it symbolically as in **node->in(NodeType::Control)**, where **NodeType** is a particular type of a node.

Here is an example of a method that adds two integers passed as parameters to a function (the following print out of the graph can be obtained by specifying -XX:+PrintIdeal command-line option):

```
class X {
    private static int foo1(int a, int b) {
        return a + b;
    }
}
```

```
11 Parm ===  3  [[ 23 ]] Parm1: int !jvms: X::foo1 @ bci:-1
10 Parm ===  3  [[ 23 ]] Parm0: int !jvms: X::foo1 @ bci:-1
3 Start ===  3  0  [[ 3  5  6  7  8  9  10  11 ]]  #{0:control, 1:abIO, 2:memory, 3:rawptr:BotPTR, 4:
return_address, 5:int, 6:int}
23 AddI === _  10  11  [[ 24 ]]  !jvms: X::foo1 @ bci:2
9 Parm ===  3  [[ 24 ]] ReturnAdr !jvms: X::foo1 @ bci:-1
8 Parm ===  3  [[ 24 ]] FramePtr !jvms: X::foo1 @ bci:-1
7 Parm ===  3  [[ 24 ]] Memory  Memory: @BotPTR *+bot, idx=Bot; !jvms: X::foo1 @ bci:-1
6 Parm ===  3  [[ 24 ]] I_O !jvms: X::foo1 @ bci:-1
5 Parm ===  3  [[ 24 ]] Control !jvms: X::foo1 @ bci:-1
24 Return ===  5  6  7  8  9 returns 23  [[ 0 ]]
0 Root ===  0  24  [[ 0  1  3 ]] inner
```

blocked URL

Notice that **AddI** has the "_" placeholder, which means its control input is null.

Nodes that access memory or otherwise depend of memory state express that dependency by having a memory state edge. It can encoded as edge with index 1 for subclasses of **MemNode** (**Loads** and **Stores**), or at index 2 for subclasses of the **SafePointNode** (which are all the calls and the safepoint). It is a good idea refer to the index symbolically - **MemNode::Memory** for **MemNode** subclasses and **TypeFunc::Memory** for **SafePointNode** subclasses. When traversing the memory graph one has to dispatch on the type of the node to get the correct index of the memory edge. Store nodes produce memory directly, Call nodes need projection nodes to extract their memory effect result.

Some nodes (like calls) require dependency on the state that is different from memory, which is called an IO dependency (input with index 1 for **SafePoint Node** subclasses) and can be refered to as **NodeType::I_O**, for example **TypeFunc::I_O.** The rest of the input edges depend on the type of node, in the example above AddI has two inputs for its left and right argument.

Whatever the input edges are the rules of the IR evaluation require that dependencies (all inputs) be evaluated before the given node (in general the evaluation model requires a depth-first traversal over input edges starting from the **Return** node). The graph itself encodes the program in essentially a functional form composing pure operations with monadic operations (that depend on control, memory and i/o side effects).

Output edges

Node's outgoing edges (def-use edges) is an unordered collection linking the node to every other node that has input edges pointing to it. In the example graph above the output edges are listed in double square brackets. Typically the out edges are iterated over using the following idiom:

```
for (DUIterator i = x->outs(); x->has_out(i); i++) {
  Node* y = x->out(i);
  ...
}
```

or (which is faster but disallows insertion to the outs array, while iterating):

```
for (DUIterator_Fast imax, i = x->fast_outs(imax); i < imax; i++) {
  Node* y = x->fast_out(i);
  ...
}
```

Having def-use edges is essential to facilitate constant-time node replacement.

Projections

Result extraction from nodes producing multiple values is achieved by using projection nodes, which may be viewed as method to extract an element from the node returning a tuple. One of the important types of projections are control projections (subclasses of the **CProjNode** node): **IfTrue** and **IfFalse** that are used to split the control flow as a result of the **If** operation.

Consider the example below:

```
class X {
    private static int foo1(int a, int b) {
        return a + b;
    }
    private static int foo2(int a, int b) {
        if (a < 0) {
            return foo1(a, b);
        }
        return 0;
    }
}
```

In the following subgraph of the **foo2** the test of **a < 0** is performed and the control flow is split using **If** and its projections:

blocked URL

Another example of projection nodes usage is extracting side effects and return values from **Call** nodes:

```
public class X {
    private static int foo1(int a, int b) {
        return a + b;
    }
    private static int foo2(int a, int b) {
        return foo1(a, b);
    }
}
```

In this case node 28 selects the return value (an int), node 26 gets the memory effect, node 25 gets the i/o effect, node 24 is the exception edge.

blocked URL

Catch projections are used to select the exception handler.It is passed in the bci of the target handler (node 32, that transfers control to a rethrow), or no_handler_bci (node 31 in the example) in case the projection doesn't lead to an exception handler.

blocked URL

Region and Phi

There are special nodes for merging values and side-effects.

Region nodes are used to merge multiple control inputs. It's first input is special and is a self-reference, other inputs refer to the CFG nodes being merged. **Phi** nodes merge data flow and other side effects (memory and i/o) that are dependent on control flow. The first input (index 0) of a **Phi** node always links it to the **Region** node. Each input of a **Phi** corresponds to the input of the **Region** with the same index. **Phi** takes the value of one of its inputs depending on which control edge is used to reach the **Region** node that it's coupled with.

Consider the following example:

```
public class X {
    private static int foo3(int a, int b, int c) {
        if (a > b) {
            return a + c;
        } else {
            return a + b;
        }
    }
}
```

blocked URL

Here the **Phi** (node 19) take the value returned by node 31 if control takes the **IfFalse** path, and value of node 32 if it's the **IfTrue** path. Also note that in this particular case **AddI** nodes don't have explicit control dependencies (since they are pure), which allows them to float freely and be easily value-numbered as early as possible. Formal IR evaluation works independent for the traversal order - one can first evaluate the control dependency of the **Phi** and then evaluate on of the data dependencies or first evaluate the data dependencies and then choose the resulting value depending on the control input.

MergeMem

For the purposes of alias analysis (and the consequent operation reordering) memory effects are spit into alias classes - “memory slices”. Each slice represents a location or a set of locations in memory.

```
public class X {
    private int f1;
    private int f2;
    private void foo4(int a, int b) {
        f1 = a;
        f2 = b;
    }
}
```

```
29 StoreI ===  5  7  28  12  [[ 18 ]]  @X+16 *, name=f2, idx=5;  Memory: @X:NotNull+16 *, name=f2, idx=5; !jvms:
X::foo4 @ bci:7
26 StoreI ===  5  7  25  11  [[ 18 ]]  @X+12 *, name=f1, idx=4;  Memory: @X:NotNull+12 *, name=f1, idx=4; !jvms:
X::foo4 @ bci:2
7 Parm ===  3  [[ 18  29  26 ]] Memory  Memory: @BotPTR *+bot, idx=Bot; !jvms: X::foo4 @ bci:-1
18 MergeMem === _  1  7  1  26  29  [[ 32 ]]  { - N26:X+12 * N29:X+16 * }  Memory: @BotPTR *+bot, idx=Bot;
32 Return ===  5  6  18  8  9  [[ 0 ]]
```

In the example above there are two stores to the fields on an object. Node 7 represents the state of memory that is received by the method, it has a **bottom** type that means “all memory”. Each store (nodes 26 and 29) get this memory state as an input and each produce separate memory slices that have types **X:NotNull+12** and **X:NotNull+16** respectively. The memory effect of a store is essentially to cut the slice from the input memory state if necessary, modify it, and pass it to the next consumer. The **Return** node requires all memory slices as an input, and the way to express it is to make a union of all visible memory side effects at that point by using a **MergeMem** node. The inputs of the MergeMem are the effects of the two stores (that are independent) and the rest of the memory. There is no way to do set subtraction and represent a memory state equivalent to **bottom - {X:NotNull+12} - {X:NotNull+16}** so we just use the **bottom** type.

blocked URL

Methods of Node

Identity()

Returns an existing node which computes the same function as this node.The optimistic combined algorithm requires to return a Node which is a small number of steps away (e.g., one of the inputs). For example for AddI, if one of the inputs is zero, return the other input.

Ideal()

The Ideal call almost arbitrarily reshapes the graph rooted at the given node. The reshape has to be monotonic to allow iterative convergence. If any change is made returns the root of the reshaped graph - even if the root is the same Node, otherwise it returns null. Example: swapping the inputs to an AddINode gives the same answer and same root, but you still have to return the 'this' pointer instead of null. It is not allowed to return an old Node, except for the 'this' pointer. Example: AddINode::Ideal must check for add of zero; in this case it returns NULL instead of doing any graph reshaping. Ideal cannot modify any old Nodes except for the 'this' pointer. Due to sharing there may be other users of the old Nodes relying on their current semantics. Modifying them will break the other users. Example: when reshape "(X+3)+4" into "X+7" you must leave the Node for "X+3" unchanged in case it is shared.

Value()

The Value call is used to compute a type of the node based on the type of its inputs. Example: int types are encoded as intervals [lo, hi] (if lo == hi, that’s a constant). Example: In case of AddINode::Value() it returns an TyepInt with the range computed as: [lo1,hi1] + [lo2,hi2] = if (!overflow(lo1 + lo2, hi1 + hi2)) [lo1 + lo2,hi1 + hi2] else [min_int,max_int].