Julia SSA-form IR

Background

Beginning in Julia 0.7, parts of the compiler use a new SSA-form intermediate representation. Historically, the compiler used to directly generate LLVM IR, from a lowered form of the Julia AST. This form had most syntactic abstractions removed, but still looked a lot like an abstract syntax tree. Over time, in order to facilitate optimizations, SSA values were introduced to this IR and the IR was linearized (i.e. a form where function arguments may only be SSA values or constants). However, non-SSA values (slots) remained in the IR due to the lack of Phi nodes in the IR (necessary for back-edges and re-merging of conditional control flow), negating much of the usefulfulness of the SSA form representation to perform middle end optimizations. Some heroic effort was put into making these optimizations work without a complete SSA form representation, but the lack of such a representation ultimately proved prohibitive.

New IR nodes

With the new IR representation, the compiler learned to handle four new IR nodes, Phi nodes, Pi nodes as well as PhiC nodes and Upsilon nodes (the latter two are only used for exception handling).

Phi nodes and Pi nodes

Phi nodes are part of generic SSA abstraction (see the link above if you're not familiar with the concept). In the Julia IR, these nodes are represented as:

```
struct PhiNode
    edges::Vector{Int}
    values::Vector{Any}
end
```

where we ensure that both vectors always have the same length. In the canonical representation (the one handled by codegen and the interpreter), the edge values indicate come-from statement numbers (i.e. if edge has an entry of `15`, there must be a `goto`, `gotoifnot` or implicit fall through from statement `15` that targets this phi node). Values are either SSA values or constants. It is also possible for a value to be unassigned if the variable was not defined on this path. However, undefinedness checks get explicitly

inserted and represented as booleans after middle end optimizations, so code generators may assume that any use of a Phi node will have an assigned value in the corresponding slot. It is also legal for the mapping to be incomplete, i.e. for a Phi node to have missing incoming edges. In that case, it must be dynamically guaranteed that the corresponding value will not be used.

PiNodes encode statically proven information that may be implicitly assumed in basic blocks dominated by a given pi node. They are conceptually equivalent to the technique introduced in the paper ABCD: Eliminating Array Bounds Checks on Demand or the predicate info nodes in LLVM. To see how they work, consider, e.g.

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    # use x
else
    # use x
end
```

We can perform predicate insertion and turn this into:

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    %x_int = PiNode(x, Int)
    # use %x_int
else
    %x_float = PiNode(x, Float64)
    # use %x_float
end
```

Pi nodes are generally ignored in the interpreter, since they don't have any effect on the values, but they may sometimes lead to code generation in the compiler (e.g. to change from an implicitly union split representation to a plain unboxed representation). The main usefulness of PiNodes stems from the fact that path conditions of the values can be accumulated simply by def-use chain walking that is generally done for most optimizations that care about these conditions anyway.

PhiC nodes and Upsilon nodes

Exception handling complicates the SSA story moderately, because exception handling introduces additional control flow edges into the IR across which values must be tracked. One approach to do so, which is followed by LLVM is to make calls which may throw exceptions into basic block terminators and add an explicit control flow edge to the catch handler:

```
invoke @function_that_may_throw() to label %regular unwind to %catch

regular:
# Control flow continues here

catch:
# Exceptions go here
```

However, this is problematic in a language like julia where at the start of the optimization pipeline, we do not know which calls throw. We would have to conservatively assume that every call (which in julia is every statement) throws. This would have several negative effects. On the one hand, it would essentially recuce the scope of every basic block to a single call, defeating the purpose of having operations be performed at the basic block level. On the other hand, every catch basic block would have `n*m` phi node arguments (`n`, the number of statements in the critical region, `m` the number of live values through the catch block). To work around this, we use a combination of `Upsilon` and `PhiC` (the C standing for `catch`, written φ^c in the IR pretty printer, because unicode subscript c is not available) nodes. There are several ways to think of these nodes, but perhaps the easiest is to think of each `PhiC` as a load from a unique store-many, read-once slot, with `Upsilon` being the corresponding store operation. The `PhiC` has an operand list of all the upsilon nodes that store to its implicit slot. The `Upsilon` nodes however, do not record which `PhiC` node they store to. This is done for more natural integration with the rest of the SSA IR. E.g. if there are no more uses of a `PhiC` node, it is safe to delete it, and the same is true of an `Upsilon` node. In most IR passes, `PhiC` nodes can be treated like `Phi` nodes. One can follow use-def chains through them, and they can be lifted to new `PhiC` nodes and new `Upsilon` nodes (in the same places as the original `Upsilon` nodes). The result of this scheme is that the number of `Upsilon` nodes (and `PhiC` arguments) is proportional to the number of assigned values to a particular variable (before SSA conversion), rather than the number of statements in the critical region.

To see this scheme in action, consider the function

```
@noinline opaque() = invokelatest(identity, nothing) # Something opaque
function foo()
    local y
    x = 1
    try
        y = 2
        opaque()
        y = 3
        error()
    catch
    end
    (x, y)
end
```

The corresponding IR (with irrelevant types stripped) is:

```
1 —       nothing::Nothing
2 — %2  = $(Expr(:enter, #4))
3 — %3  = Y (false)
│   %4  = Y (#undef)
│   %5  = Y (1)
│   %6  = Y (true)
│   %7  = Y (2)
│         invoke Main.opaque()::Any
│   %9  = Y (true)
│   %10 = Y (3)
│         invoke Main.error()::Union{}
└──       $(Expr(:unreachable))::Union{}
4 ┄ %13 = φᶜ (%3, %6, %9)::Bool
│   %14 = φᶜ (%4, %7, %10)::Core.Compiler.MaybeUndef(Int64)
│   %15 = φᶜ (%5)::Core.Compiler.Const(1, false)
└──       $(Expr(:leave, 1))
5 —       $(Expr(:pop_exception, :(%2)))::Any
│         $(Expr(:throw_undef_if_not, :y, :(%13)))::Any
│   %19 = Core.tuple(%15, %14)
└──       return %19
```

Note in particular that every value live into the critical region gets an upsilon node at the top of the critical region. This is because catch blocks are considered to have an invisible control flow edge from outside the function. As a result, no SSA value dominates the catch blocks, and all incoming values have to come through a `φᶜ` node.

Main SSA data structure

The main `SSAIR` data structure is worthy of discussion. It draws inspiration from LLVM and Webkit's B3 IR. The core of the data structure is a flat vector of statements. Each statement is implicitly assigned an SSA value based on its position in the vector (i.e. the result of the statement at idx 1 can be accessed using `SSAValue(1)` etc). For each SSA value, we additionally maintain its type. Since, SSA values are definitionally assigned only once, this type is also the result type of the expression at the corresponding index. However, while this representation is rather efficient (since the assignments don't need to be explicitly encoded), it of course carries the drawback that order is semantically significant, so reorderings and insertions change statement numbers. Additionally, we do not keep use lists (i.e. it is impossible to walk from a def to all its uses without explicitly computing this map–def lists however are trivial since you can look up the corresponding statement from the index), so the LLVM-style RAUW (replace-all-uses-with) operation is unavailable.

Instead, we do the following:

- We keep a separate buffer of nodes to insert (including the position to insert them at, the type of the corresponding value and the node itself). These nodes are numbered by their occurrence in the insertion buffer, allowing their values to be immediately used elesewhere in the IR (i.e. if there are 12 statements in the original statement list, the first new statement will be accessible as `SSAValue(13)`).
- RAUW style operations are performed by setting the corresponding statement index to the replacement value.
- Statements are erased by setting the corresponding statement to `nothing` (this is essentially just a special-case convention of the above.
- If there are any uses of the statement being erased, they will be set to `nothing`.

There is a `compact!` function that compacts the above data structure by performing the insertion of nodes in the appropriate place, trivial copy propagation, and renaming of uses to any changed SSA values. However, the clever part of this scheme is that this compaction can be done lazily as part of the subsequent pass. Most optimization passes need to walk over the entire list of statements, performing analysis or modifications along the way. We provide an `IncrementalCompact` iterator that can be used to iterate over the statement list. It will perform any necessary compaction and return the new index of the node, as well as the node itself. It is legal at this point to walk def-use chains, as well as make any modifications or deletions to the IR (insertions are disallowed however).

The idea behind this arrangement is that, since the optimization passes need to touch the corresponding memory anyway and incur the corresponding memory access penalty, performing the extra housekeeping should have comparatively little overhead (and save the overhead of maintaining these data structures during IR modification).

« Inference Static analyzer annotations for GC correctness in C code »