Inlining

Inlining

- Put the body of a function we're *calling* inside the body of the *caller*
- Naïve assumption: Saves a call instruction
- Reality: Who cares about one instruction
- *Inlining allows us to propagate knowledge over function boundaries*

Catalysts

- Inlining is a *catalyst*: It is barely an optimization on its own, but helps other optimizations
- As such, its effect is hard to measure… but trust me, it's *huge*

```
function doubler(x) {
    return x+x;
}

function quadler(x) {
    return doubler(x)+doubler(x);
}
```

This looks fine… let's introduce types!

doubler:

```
1: PARAM 1
3: ADD 1, 1
4: RETURN 3
```

quadler:

```
1: PARAM 1
3: (function "quadler")
4: ARG 1
5: CALL 3
7: ARG 1
8: CALL 3
10:ADD 5,8
11:RETURN 10
```

doubler:

```
1: PARAM 1: blob
3: ADD 1, 1: blob
4: RETURN 3: void
```

quadler:

```
1: PARAM 1: blob
3: (function "quadler"): func
4: ARG 1: void
5: CALL 3: blob
7: ARG 1: void
8: CALL 3: blob
10:ADD 5,8: blob
11:RETURN 10: void
```

OK, types did nothing... speculation?

doubler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: ADD 2, 2: int
4: RETURN 3: void
```

quadler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: (function "quadler"): func
4: ARG 2: void
5: CALL 3: blob
6: SPECULATE 5: int
7: ARG 2: void
8: CALL 3: blob
9: SPECULATE 8: int
10:ADD 6,9: int
11:RETURN 10: void
```

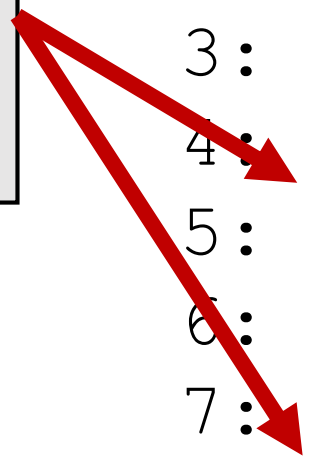
All we've done is make the type-checking explicit!
No way for us to know that (6) and (9) will never fail!

doubler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: ADD 2, 2: int
4: RETURN 3: void
```

quadler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: (function "quadler"): func
4: ARG 2: void
5: CALL 3: blob
6: SPECULATE 5: int
7: ARG 2: void
8: CALL 3: blob
9: SPECULATE 8: int
10:ADD 6,9: int
11:RETURN 10: void
```



Let's inline!

Inlining 101

- SSA makes inlining very easy:
  - Make a new symbol table, populated with the arguments,
  - compile the target function body inline, with a new inline-return pair for the jump out,
  - union multiple returns into the result.

doubler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: ADD 2, 2: int
4: RETURN 3: void
```

quadler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: (function "quadler"): func
    a1: ADD 2,2: int
    a2: INLINE_RETURN a1: int
4: INLINE_RETURN_TARGET a2: v
    b1: ADD 2,2: int
    b2: INLINE_RETURN b1: int
7: INLINE_RETURN_TARGET b2: v
10:ADD a2,b2: int
11:RETURN 10: void
```

BAM! And the type checking's gone! (Mostly)

It ain’t that easy

- We glossed over some critical details:
  - I can’t inline an unknown function
  - It’s not clear when inlining is actually beneficial
  - It’s not clear how inlining and profiling interact

It ain’t that easy

- We glossed over some critical details:
  - I can’t inline an unknown function
  - It’s not clear when inlining is actually beneficial
  - It’s not clear how inlining and profiling interact

Everything's related

- Inlining in a dynamic language demands speculation
  - Profile which functions a call dispatches to,
  - speculate that the call will dispatch to the same function in the future,
  - handle speculation failures at runtime.
- [This is just as true of Java!]

This is Hell, and also the second half of this lecture

doubler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: ADD 2, 2: int
4: RETURN 3: void
```

quadler:

```
1: PARAM 1: blob
2: SPECULATE 1: int
3: (function “quadler”): func
    a1: ADD 2,2: int
    a2: INLINE_RETURN a1: int
INLINE_RETURN_TARGET a2: v
    b1: ADD 2,2: int
INLINE_RETURN b1: int
10:ADD a2,b2: int
11:RETURN 10: void
```

Really, this needs to be something like:

```
3: TOP: object
4: GETMEMBER 3,”quadler”: blob
5: SPECULATE_FUNC 4, (the function we’ve seen before)
```

BAM! And the type checking’s gone! (Mostly)

It ain’t that easy

- We glossed over some critical details:
  - I can’t inline an unknown function
  - It’s not clear when inlining is actually beneficial
  - It’s not clear how inlining and profiling interact

This is the slide where I would tell you the good news if there was any.

Inlining and brown numbers

- Inlining is *almost always* beneficial in runtime
- Inlining leads to more compile time, more memory use
- VM's set an arbitrary limit on function size to inline/function size after inlining
- … there is no further logic.

Demo

It ain’t that easy

- We glossed over some critical details:
  - I can’t inline an unknown function
  - It’s not clear when inlining is actually beneficial
  - It’s not clear how inlining and profiling interact

The problem

- Profiling tells me it’s always the same func
- I inline it! Do I…
  - Inline its profiling results, which apply to *all* calls to that function, or…
  - restart profiling with the inlined version?
- If the latter, how does profiling end?

Multi-stage JIT to the rescue

- “Middle” stages of a JIT must be able to profile+specialize
- Goal is fix-point *or* brown number limit on number of passes

Ossification

- Yup.

Function Replacement

Look at me. I am your function now.

What's going on here?

- Our ossification example didn't ossify on Firefox because of inlining
- But… there are only two functions, and one of them only runs once
- It actually replaced the outer function *while it was already executing*

The problem

- I’m recompiling a function…
- while running that function.
- Need to jump from one version to the other
- As you can imagine, this… is difficult

Breaking it down

- Smaller problems:
  - Egress
  - Ingress
  - On-stack replacement
  - Garbage collection

Note: The whole concept is often called “on-stack replacement”, but *really* that’s one part

The easy solution

- Easiest solution: Don’t
- Usually, the old function works fine!

Board