Irreducible Control Flow in WebAssembly

Web Engines Hackfest

Conrad Watt

This talk

- What is irreducible control flow?
- Why do we care that WebAssembly can’t directly express it?
- How would we extend WebAssembly?

What is reducible control flow

- Exactly the (intra-function) control flow that is directly expressible using blocks, loops, conditionals, and labelled break/continue.

- Programs/languages using only these constructs are called “semi-structured” (e.g. Java, JavaScript).

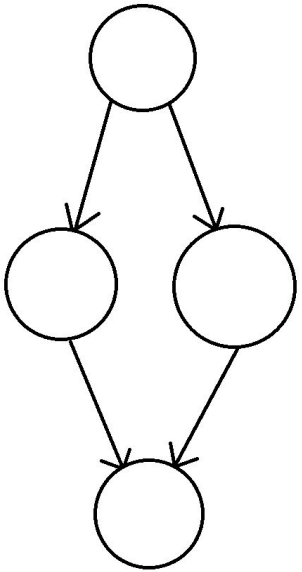
What is reducible control flow

- Formally characterised by conditions on the control flow graph:
  - Can partition all edges into “forward” and “backward” sets s.t.
    - Forward edges form a rooted DAG
    - For all backward edges (A,B), node B dominates node A

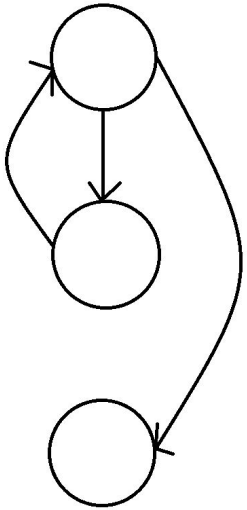
- Defines where the “loops” are in the CFG, and restricts all loops to be single-entry

What is irreducible control flow

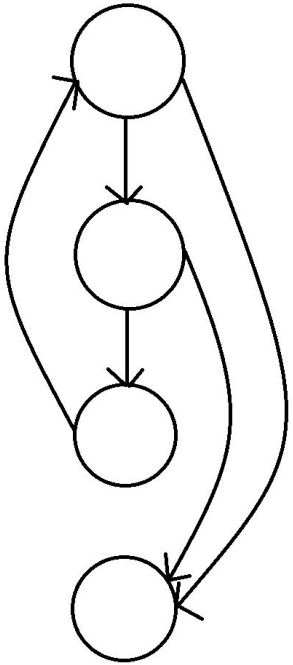
- Everything else
- Programs with fancy uses of **`goto`** which cannot be directly expressed using semi-structured control constructs
- Can be characterised in terms of the existence of *multi-entry loops* in the CFG



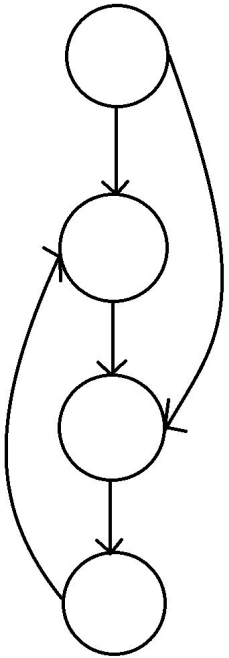
(a) reducible control flow
if-else (structured)



(b) reducible control flow
simple loop (structured)



(c) reducible control flow
loop with break
(semi-structured)



(d) irreducible control flow
`goto` into middle of loop body
(unstructured)

Limitations for Wasm

- Wasm only has semi-structured control flow constructs (**block**, **loop**, **if**, **br**)
- Therefore, can only be directly targeted by CFGs which are reducible
- When compiling a program with irreducible control flow, need to use an inefficient indirection

Limitations for Wasm

```
…
goto a; // x
…
goto b; // y
…
goto c; // z
```

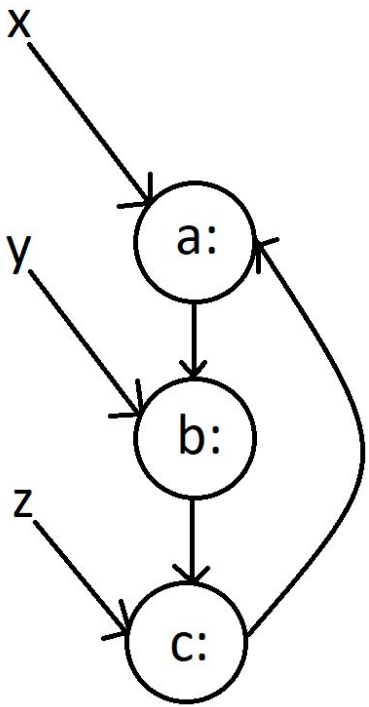
```
a:
while(true) {
  …
  b:
  …
  c:
  …
}
```

Limitations for Wasm

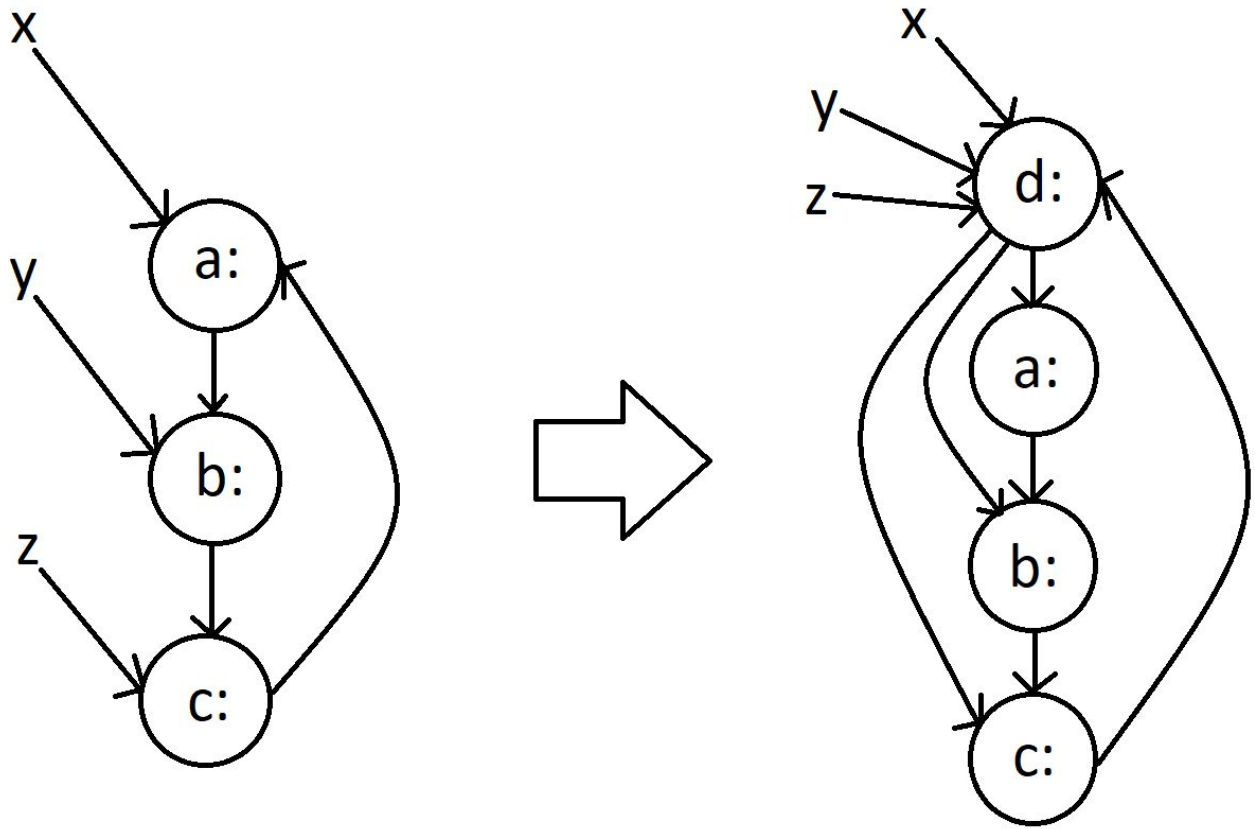
```
…
goto a; // x

…
goto b; // y

…
goto c; // z
```

```
a: …


b: …


c: …

goto a;
```

Limitations for Wasm



Limitations for Wasm



```
…
goto a; // x

…
goto b; // y

…
goto c; // z
```

```
a: …

b: …

c: …

goto a;
```

```
…
dispatch = 0;
goto d; // x
…
dispatch = 1;
goto d; // y
…
dispatch = 2;
goto d; // z
```

```
d:
switch (dispatch) {
  case == 0: goto a;
  case == 1: goto b;
  case == 2: goto c; }
a: …
b: …
c: …
dispatch = 0;
goto d;
```

State of the world

- Most real user programs inherently have reducible control flow, even if they use **`goto`**
  - e.g. “**`goto finalize`**” is fine - no indirections needed

- Irreducible control flow can appear as a result of
  - Hyper-hand-optimised code (e.g. in a standard library)
  - Implementing async/resumable functions with green threads (e.g. Goroutines)
  - Compiler IR optimisations, even of semi-structured code

State of the toolchain art

- Three state-of-the-art implementations
  - LLVM FixIrreducibleControlFlow + CFGStackify
  - Cheerp Stackifier
    - Good rundown blogpost here:
      “Solving the structured control flow problem once and for all”
      https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2
  - Binaryen Relooper (with 2016 control stack optimisations)

Why would we want irreducible CF?

- Producer ergonomics
  - Having to implement CFG transformation/Wasmification is a major speed bump

- Programs with lots of irreducible control flow might be inefficient
  - Current approaches make irreducible control flow relatively “pay-as-you-go”

How would we add irreducible CF?

- Constraints:
  - One-pass validation and codegen
  - Must compose with existing control flow operations
  - Engines must be able to optimise

- The funclets proposal was a first draft of this (https://github.com/WebAssembly/funclets/blob/master/proposals/funclets/Overview.md)

`multiloop` - iterating on funclets

- Within the body of a regular Wasm **`loop`**, you can jump back to the start of the body using **`br`**.
  - Works like a higher-level language's **`continue`** (semi-structured).

- **`multiloop`** - a loop with multiple bodies. Within any body, you can jump to the start of any body using **`br`**.
  - To enable one pass validation and compilation, in general the type signatures of *all* bodies must be forward declared before the code of *any* body.

`multiloop`

Abstract syntax:

$$\texttt{multiloop}\ tf^{\,n}\ (e\texttt{* end})^{n}$$

Forgetting exceptions for a second, any function CFG can be represented as a single **`multiloop`** with one body for each basic block.

There is a decent amount of subtlety around exception handling

`multiloop`

Abstract syntax:

multiloop *tf* ^{n} (*e** **end**)^{n}

Bikeshedding (not necessarily for now):

- Efficient body type declarations (e.g. in the binary format)
- Should bodies have fallthrough semantics?
  (I'd argue yes)
- Unifying **`loop`**, **`block`**, and **`multiloop`** in the formal semantics

Composition with existing control flow

- Multiloops can be arbitrarily nested within each-other, and within regular **`block`** and **`loop`** constructs.

- To calculate the **`br`** index, count through each **`multiloop`** body in turn.

Composition with existing control flow

For each *k* = n, where does the inner **(br *k*)** target?

```
loop ([]->[]) k = 3:
    multiloop ([]->[]), ([]->[])
        k = 1: <multiloop first body>
    end
        k = 2: (block ([]->[]) (br k) ... end k = 0:)
    end
end
```

Composition with existing control flow

For each `k = n`, where does the inner `(br k)` target?

```
loop ([]->[]) k = 3:
    multiloop ([]->[]), ([]->[])
        k = 1: <multiloop first body>
    end
        k = 2: (block ([]->[]) (br k) ... end k = 0:)
    end
end
```

Composition with existing control flow

For each `k = n`, where does the inner `(br k)` target?

```
loop ([]->[]) k = 3:
    multiloop ([]->[]), ([]->[])
        k = 1: <multiloop first body>
    end
        k = 2: (block ([]->[]) (br k) ... end k = 0:)
    end
end
```

Composition with existing control flow

For each `k = n`, where does the inner `(br k)` target?

```
loop ([]->[]) k = 3:
    multiloop ([]->[]), ([]->[])
        k = 1: <multiloop first body>
    end
        k = 2: (block ([]->[]) (br k) ... end k = 0:)
    end
end
```

Composition with existing control flow

For each `k = n`, where does the inner `(br k)` target?

```
loop ([]->[]) k = 3:
    multiloop ([]->[]), ([]->[])
        k = 1: <multiloop first body>
    end
        k = 2: (block ([]->[]) (br k) ... end k = 0:)
    end
end
```

Composition with existing control flow

For each `k = n`, where does the inner `(br k)` target?

```
loop ([]->[]) k = 3:
    multiloop ([]->[]), ([]->[])
        k = 1: <multiloop first body>
    end
        k = 2: (block ([]->[]) (br k) ... end k = 0:)
    end
end
```

Exception handling

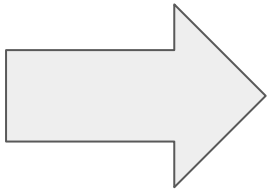
- How can **`multiloop`** compose with the Wasm **`try/catch`** proposal?
- Obvious semantics - if an uncaught exception is thrown in any body of the **`multiloop`**, the **`multiloop`** as a whole is broken out of.
- Is this semantics-preserving for source programs?

Exception handling

```
try {
  f_maythrow();
  a:
  g_maythrow();
} catch () {
  h();
  goto a;
}
```

Exception handling

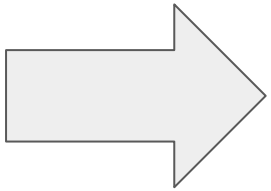
```
try {
  f_maythrow();
  a:
  g_maythrow();
} catch () {
  h();
  goto a;
}
```



```
try
  ???
catch
  (call $h)
  ???
end
```

Exception handling

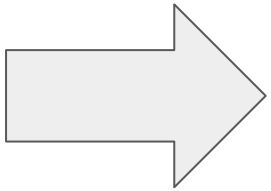
```
try {
  f_maythrow();
  a:
  g_maythrow();
} catch () {
  h();
  goto a;
}
```



```
try
  (call $f)
  (call $g)
catch
  (call $h)
  ???
end
```

Exception handling

```
try {
  f_maythrow();
  a:
  g_maythrow();
} catch () {
  h();
  goto a;
}
```



```
multiloop <2>
  try
    (call $f)
  catch
    (call $h)
    (br 1)
  end
end
  try
    (call $g)
  catch
    (call $h)
    (br 1)
  end
end
```

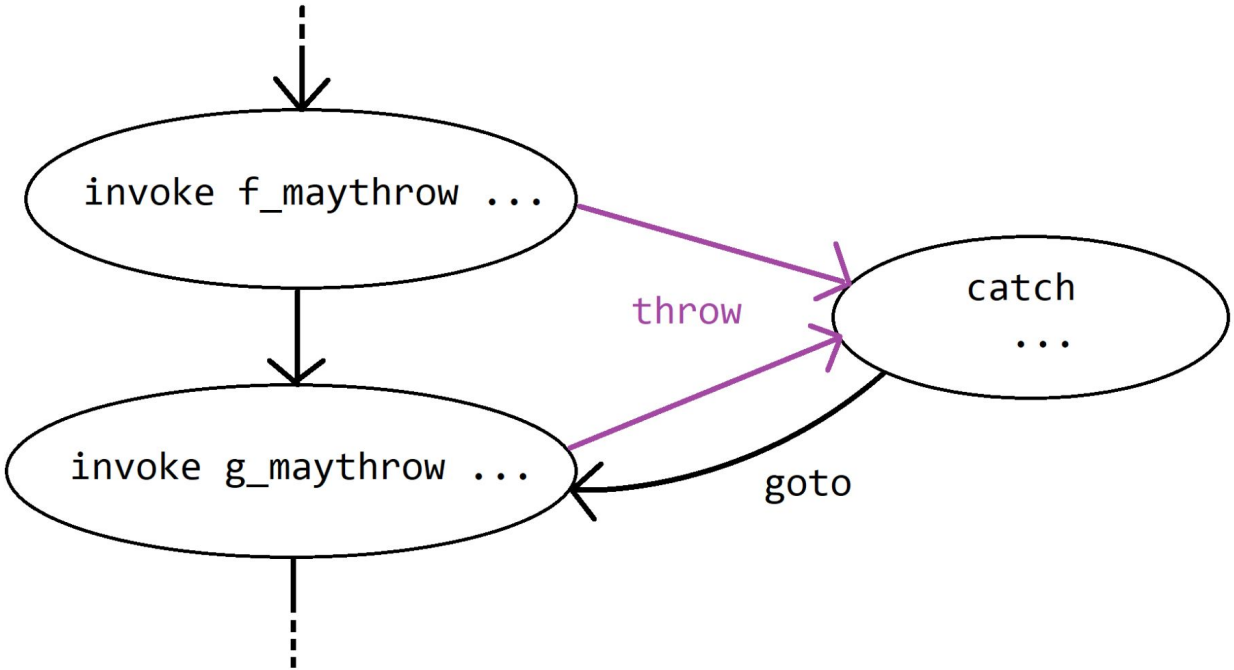
C++ standard to the rescue!

- “A `goto` or `switch` statement shall not be used to transfer control into a try block or into a handler.” (N4713 18.3)
  - explicitly forbids jumping from a `catch` back into a `try` body
- So this code is actually not legal C++!
- Due to this restriction, we can always translate a C++ try/catch into a Wasm **`multiloop`** and outer **`try/catch`** - we never need to handle jumping back into the **`multiloop`** from the **`catch`**

Except...

- LLVM’s exception-handling is more general
- No explicit try scope, just (effectively) floating catch blocks which can be thrown to from anywhere in the function
- While the example isn’t valid C++, an analogous example *is* valid LLVM IR.

LLVM-style basic blocks



LLVM

- We can’t write a C++ program that will generate this LLVM directly

- We *can* craft a C++ program that LLVM will optimise into this CFG

```
int x = 1:
a:
try {
  if (x) {
    f_maythrow();
  }
  g_maythrow();
} catch () {
  h();
  x = 0;
  goto a;
}
```

```
int x = 1:
a:
try {
  if (x) {
    f_maythrow();
  }
  g_maythrow();
} catch () {
  h();
  x = 0;
  goto a;
}
```

```
int x = 1:
a:
try {
  if (x) {
    f_maythrow();
  }
  g_maythrow();
} catch () {
  h();
  x = 0;
  goto a;
}
```

```
int x = 1:
a:
try {
  if (x) {
    f_maythrow();
  }
  g_maythrow();
} catch () {
  h();
  x = 0;
  goto a;
}
```

Consequences

- LLVM would still need some minor code duplication/transformation even with **`multiloop`**
  - Still much less than is currently necessary!
  - Could be fully supported through a more generalised **`multiloop`** with **`catch`** blocks.

- A toolchain that preserves source try/catch nesting could use **`multiloop`** and Wasm **`try/catch`** to implement C++-style exceptions without issue

Implementation

- **`multiloop`** should cause no issues for validation, or baseline compilers
  - One-pass compilers already need to deal with compiling jumps out of blocks they haven’t seen the end of yet
  - The important thing is that target type signatures need to be available - hence forward declaration

- I believe the main blocker is that some optimising compilers would need to be re-engineered to handle the more general control flow

Implementation

- Some optimising compilers would need to be re-engineered to handle the more general control flow

- My main concern would be a JavaScript Proper Tail Calls scenario - some engines can easily optimise **`multiloop`**, and some perhaps can’t. Consensus seeking might become painful.

- Maybe **`multiloop`** can initially be part of a toolchain-only “pre-Wasm” which must be transformed before deployment to the Web?
  - Solves the producer erg issue, but not the performance issue

How to build a case for irreducible CF?

- Immediately visceral: find an existing program compiled to Wasm that is slowed down by the insertion of lots of CF indirections
    - Appear to be some pretty compelling Go examples, although these could potentially be addressed by better async support (e.g. continuations)
    - Would be good to find a C/C++ example

How to build a case for irreducible CF?

- Hard to quantify “road not taken” impact
    - Compiler Wasm targets abandoned/never attempted due to perceived complexity
    - Wasm deployments abandoned due to (relevant) performance issues
- Can we collect anecdata?

Write-up (including exception-handling)

- https://tinyurl.com/multiloop
  - https://gist.github.com/conrad-watt/6a620cb8b7d8f0191296e3eb24dffdef

- Special thanks to Heejin Ahn, Ross Tate, and Alon Zakai for some incredibly helpful conversations!

Analogy to funclets (extra)

- The top-level **`multiloop`** is like a **`funclet_region`**
- The forward-declaration of body types is like a naive **`funclet_sig`**
  - As mentioned, we could bikeshed more efficient representations
- The existing **`br`** instructions work like **`funclet_call`**, **`funclet_call_if`**, etc
- Difference: a funclet nested inside another funclet can't contain jumps to the outer funclet.