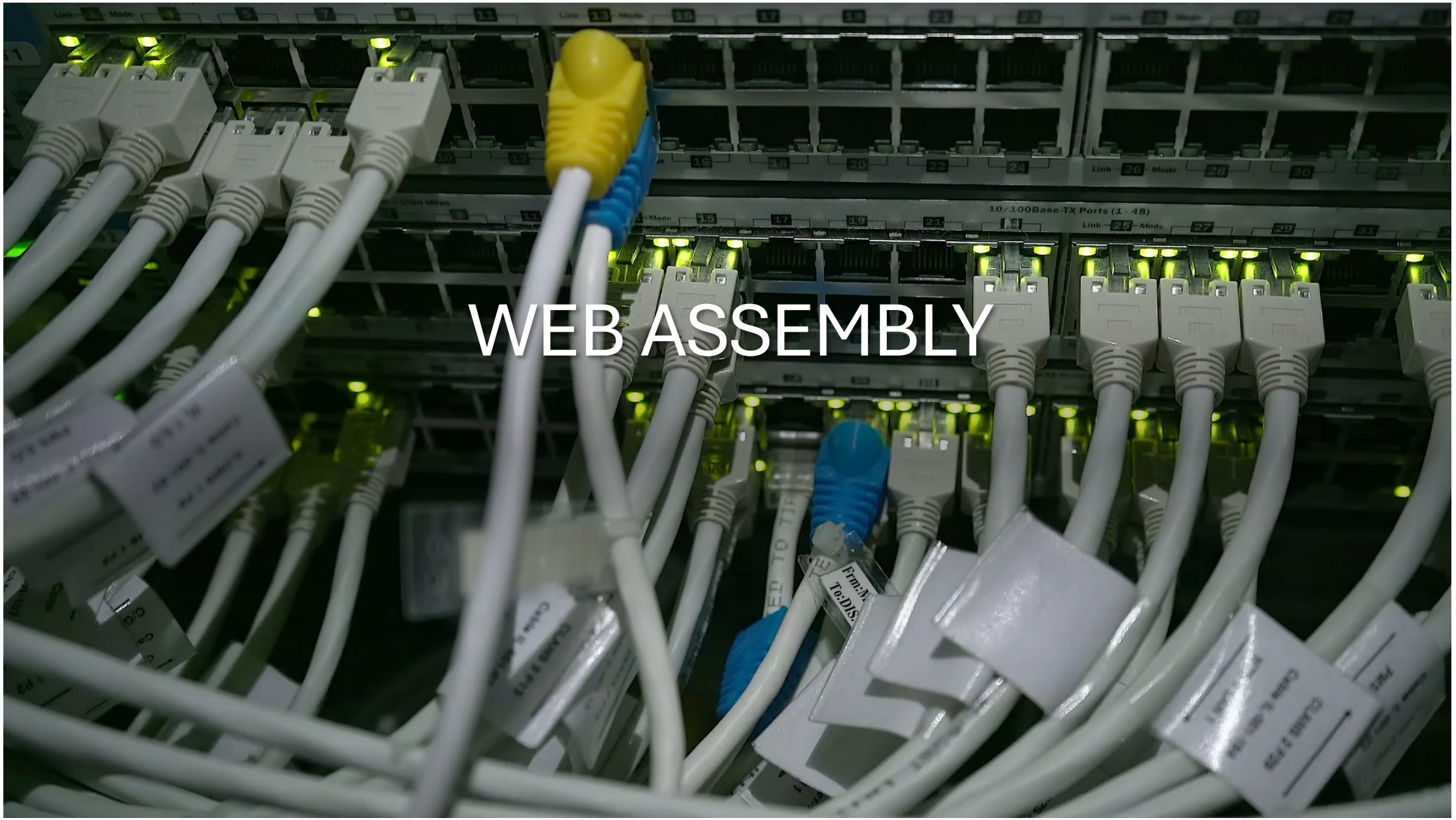

WEB ASSEMBLY

What is WebAssembly?

Why WebAssembly?

The origin and motivation of WebAssembly is described well in the introduction of a paper

Bringing the Web up to Speed with WebAssembly (PLDI2017) Prog. language design and implementation

jointly authored by employees at Google, Microsoft, Mozilla, and Apple that laid out the design principles for the language.

Bringing the Web up to Speed with WebAssembly

Andreas Haas Andreas Rossberg Derek L. Schuff* Ben L. Titzer

Google GmbH, Germany / *Google Inc, USA
{ahaas,rossberg,dschuff,titzer}@google.com

Michael Holman

Microsoft Inc, USA
michael.holman@microsoft.com

Dan Gohman Luke Wagner Alon Zakai

Mozilla Inc, USA
{sunfishcode,luke,azakai}@mozilla.com

JF Bastien

Apple Inc, USA
jfbastien@apple.com

Many people working in industries

Quoting the Introduction of PLDI2017 paper

The Web began as a simple document exchange network but has now become the most ubiquitous application platform ever, accessible across a vast array of operating systems and device types. By historical accident, JavaScript is the only natively supported programming language on the Web, its widespread usage unmatched by other technologies available only via plugins like ActiveX, Java or Flash. Because of JavaScript's ubiquity, rapid performance improvements in modern VMs, and perhaps through sheer necessity, it has become a compilation target for other languages. …. WebAssembly addresses the problem of safe, fast, portable low-level code on the Web.

presence

To make Web App standard by de facto was Javascript

Try to have common standard that is: ①

Quoting the Introduction of PLD2017

Previous attempts at solving it, from ActiveX to Native Client to asm.js, have fallen short of properties that a low-level compilation target should have:

① Safe, fast, and portable semantics:

- safe to execute
- fast to execute
- language-, hardware-, and platform-independent
- deterministic and easy to reason about
- simple interoperability with the Web platform

Safe and efficient representation:

- compact and easy to decode
- easy to validate and compile
- easy to generate for producers
- streamable and parallelizable

There was a need for a common language for Web Apps across different browsers. It is at the level of intermediate representation (like bytecode). A bit higher than machine language ①

What is WebAssembly?

(BYTECODE)

WebAssembly (WASM) is a **binary instruction format** designed as a **portable compilation target**. It allows code written in languages like **C, C++, Rust** to run efficiently in web browsers and beyond.

- **Key Characteristics:**
  - Low-level, **stack-based** abstract machine Low level AM: AM closer to the hardware representation
  - Executable in **web browsers** alongside JavaScript ②
  - Runs at **near-native speed** by leveraging hardware capabilities
  - **Secure** by design with a sandboxed execution model
  - ③ **Platform-independent**, meaning the same WASM code can run across different environments

Think of WASM as a way to run high-performance code (like Rust) inside the browser—without needing JavaScript to do the heavy lifting.

① We compile to this intermediate representation.
② There is a direct correspondance between
There's possibility to put together JS and WASM.
Very close at efficiency you have with direct implementation on hardware
③ Ensures portability among different architecture, as long as they provide implementation of WASM AM.

(my personal) Remark

WebAssembly's design follows the **Security by Design** approach, ensuring that all its features have been **formally specified**, making it a robust and well-defined execution environment.

Overall, WebAssembly is an excellent ”real” example of the approach taken in this course on **programming languages and security**.

We want formal methods as method to achieve security. Formal specification on one side, security by design on the other.

Our goal:
Introduce you to WebAssembly …

- …. And we hope you will get a sense of
  - what the language is,
  - how it works,
  - how you could program it,
  - how its semantics is formalized.

Key features

Portability

- Runs on **any device and platform** with a WASM runtime (browsers, servers, embedded devices).
- Independent of CPU architecture (works on x86, ARM, RISC-V, etc.).

Speed & Efficiency

- Near-native execution speed (avoids JavaScript's overhead).
- Optimized for fast startup and low memory usage.

Security (by design)

- **Sandboxed environment** prevents direct access to system resources.
- Memory isolation makes it resistant to common attacks like buffer overflows.

Key Features

Interoperability with JavaScript:

- Can be loaded and executed using the **WebAssembly JavaScript API**.
- Supports passing data between WASM and JavaScript seamlessly.

Expanding Beyond the Browser:

- **Server-side execution** (Cloudflare Workers, Fastly's Compute@Edge)
- **IoT & Embedded Systems**
- **Blockchain smart contracts** (Ethereum, Polkadot)
- **Gaming Engines** (Unity WebAssembly, Unreal Engine)

The **WebAssembly JavaScript API** is a set of JavaScript functions and objects that allow web applications to interact with and execute WebAssembly (WASM) modules in a browser or other JavaScript environments.

Key Features:

- **Loading and Compiling WASM Modules**: JavaScript can fetch, compile, and instantiate WebAssembly binaries.
- **Interfacing Between JavaScript and WASM**: JavaScript can call WebAssembly functions and pass data to them.
- **Memory Management**: WASM modules use a linear memory model that JavaScript can read from and write to.
- **Performance Optimization**: WebAssembly runs at near-native speed, and the API allows JavaScript to work efficiently with it.

WASM Sandbox?

- A **sandbox** is a security mechanism that isolates code execution to prevent **malicious or unintended interactions** with the host system.
- WebAssembly **does not have direct access to system resources** (files, network, memory, or hardware).
- WASM code runs in a **restricted virtual machine**, separate from the host environment. It can only interact with the outside world via **explicit imports and APIs provided by the host (e.g., JavaScript).**
- **WebAssembly sandbox supports execution of complex operations, but it can't step outside the boundaries without explicit permission.**

Stack Machine

A **stack machine** is a type of **abstract machine** that exploits a **stack data structure** to evaluate expressions and execute instructions.

Instead of using registers for storing intermediate values, a stack machine operates by **pushing** and **popping** values onto and from the stack.

The **SECD machine** is a **stack-based virtual machine**, specifically designed to execute **functional programming languages**

Key features

Uses a Stack for Execution

- Operands are pushed onto the stack before operations.
- Instructions take their arguments from the stack and push results back.

Implicit Operand Handling

- Unlike register-based machines, where operands are explicitly referenced, a stack machine assumes that operands are on the top of the stack.

Compact and Simple Instruction Set

- Instructions are often short and efficient, leading to a more compact bytecode representation.

WASM Abstract Machine

- **WebAssembly** exploits a *stack machine* model of computation, where at runtime a program modifies a separate stack of values.
- **The Java Virtual Machine (JVM)** – Executes Java bytecode using a stack-based computational model.
- **Why?:** Stack machines simplify instruction decoding and reduce bytecode size, making them ideal for **abstract machines** and **interpreters** like WebAssembly and the JVM.

A running example

WebAssembly program that implements the function $f(x)=2*x^2+1$.

```
(func $f (param $x i32) (result i32)
                                ;; stack: []
  (get_local $x)                ;; stack: [x]
  (get_local $x)                ;; stack: [x, x]
  (i32.mul)                     ;; stack: [x*x]
  (i32.const 2)                 ;; stack: [2, x*x]
  (i32.mul)                     ;; stack: [2*x*x]
  (i32.const 1)                 ;; stack: [1, 2*x*x]
  )i32.add)                     ;; stack: [2*x*x+1]
  )
```

integer with 32 bits

- $ to identify an identifier

① At the beginning stack is empty, no intermediate value

$x is interpreted as a local variable

bytecode op to manage local parameter. We push it to the stack.

i32.mul across integers, we have basic operation at level of abstract machine

A running example

```
(func $f (param $x i32) (result i32)
                                ;; stack: []
(get_local $x)                  ;; stack: [x]
(get_local $x)                  ;; stack: [x, x]
(i32.mul)                       ;; stack: [x*x]
(i32.const 2)                   ;; stack: [2, x*x]
(i32.mul)                       ;; stack: [2*x*x]
(i32.const 1)                   ;; stack: [1, 2*x*x]
)i32.add)                       ;; stack: [2*x*x+1]
)
```

Representation of a program

WebAssembly text format is derived from S-expressions, (borrowed from Lisp).

An S-expression is either an "atom" (e.g. get_local $x), or a list of S-expressions surrounded by parentheses (e.g. (get_local $x), (param $x i32)).

A running example

```
(func $f (param $x i32) (result i32)
                                ;; stack: []
(get_local $x)                  ;; stack: [x]
(get_local $x)                  ;; stack: [x, x]
(i32.mul)                       ;; stack: [x*x]
(i32.const 2)                   ;; stack: [2, x*x]
(i32.mul)                       ;; stack: [2*x*x]
(i32.const 1)                   ;; stack: [1, 2*x*x]
)i32.add)                       ;; stack: [2*x*x+1]
)
```

Instructions of WebAssembly modify the value stack in some way.

The instructions

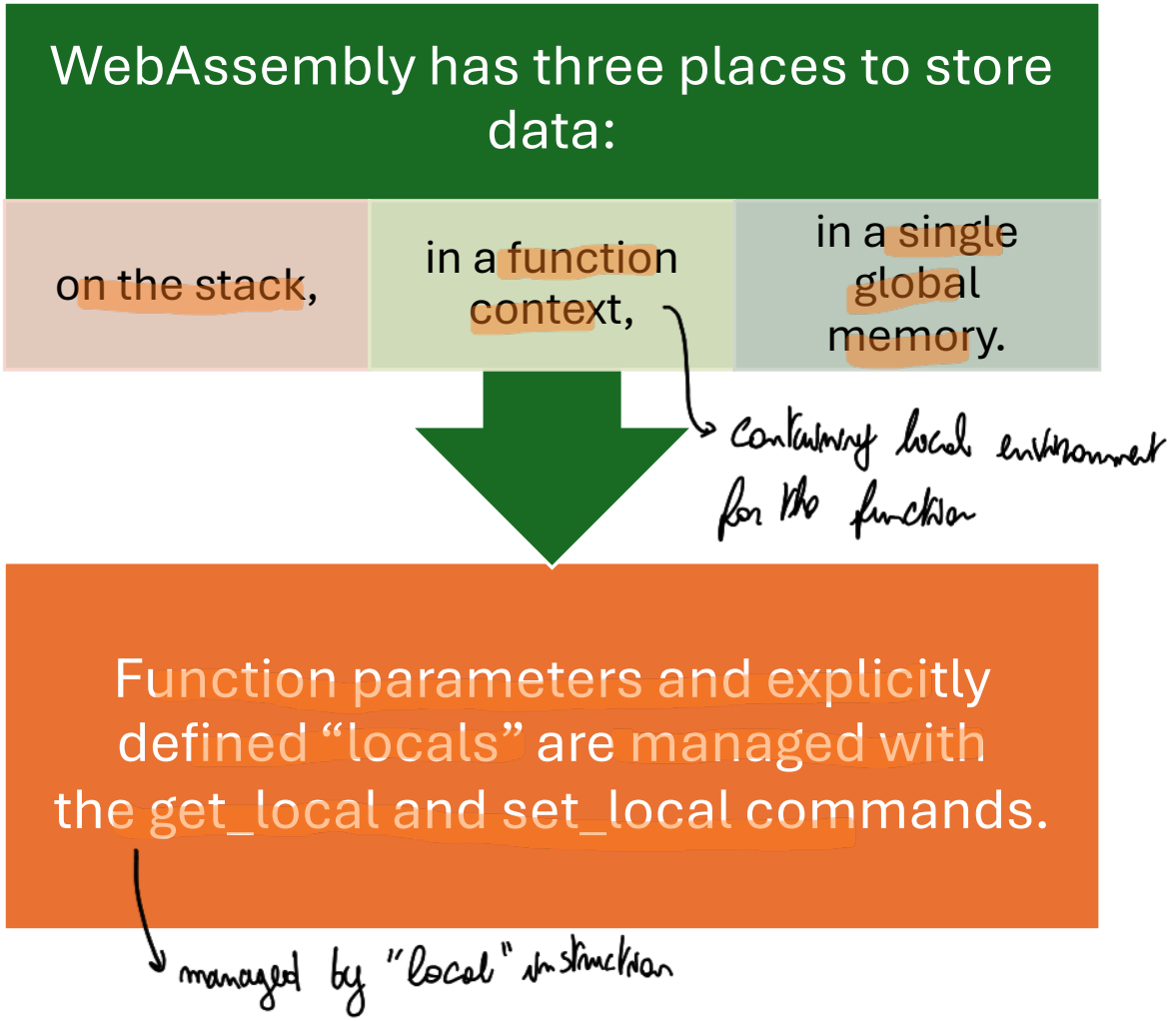
- **get_local** pushes the parameter **x** onto the stack
- **i32.mul** pops two values from the stack, multiplies them, and pushes the result back onto the stack.
- **i32.const n** pushes the value **n** onto the stack.

A running example

```
(func $f (param $x i32) (result i32)
                                ;; stack: []
(get_local $x)                  ;; stack: [x]
(get_local $x)                  ;; stack: [x, x]
(i32.mul)                       ;; stack: [x*x]
(i32.const 2)                   ;; stack: [2, x*x]
(i32.mul)                       ;; stack: [2*x*x]
(i32.const 1)                   ;; stack: [1, 2*x*x]
)i32.add)                       ;; stack: [2*x*x+1]
)
```

The function implicitly returns the value at the top of the stack.

Storing values



An example

```c
int f(int x) {
  int i = x + 1;
  return i/x;
}
```

C

```
(func $f (param $x i32) (local $i i32) (result i32)
  (get_local $x)      [x]
  (i32.const 1)       [1,x]
  (i32.add)           [x+1]
  (set_local $i)      [] : implementation of assignment: v = top of stack
                           pop it and the local memory

  (get_local $x)      [x]
  (get_local $i)      [i,x]
  (i32.div_s))        [i/x]
```

modify memory location associated to variable $i.

WASM

An example

(local $i i32) declares a local variable within the function f

```c
int f(int x) {
  int i = x + 1;
  return i/x;
}
```

C

```
(func $f (param $x i32) (local $i i32) (result i32)
  (get_local $x)
  (i32.const 1)
  (i32.add)
  (set_local $i)

  (get_local $x)
  (get_local $i)
  (i32.div_s))
```

WASM

An example

The load and store instructions read and write to memory. There is a single memory, so load/store are not parameterized by which memory they are accessing.

```c
void incr(int* x) {
  *x = *x + 1;
}
```

C

```
(func $incr (param $x i32)
  (get_local $x)
  (i32.load)

  (i32.const 1)
  (i32.add)

  (get_local $x)
  (i32.store))
```

WASM

Control flow (register-based abstract machine)

Control flow in **register-based** abstract machines works by maintaining a ***program counter***.

Program counter acts as a pointer to the current instruction.

After executing an instruction, the program counter increments by 1 pointing to the next instruction, but jump instructions allow the program counter to be set an arbitrary value.

Goto are harmful

- **Arbitrary jumps** are a bad idea: they are an unmitigated disaster.
- **Dijkstra** knew this back in **1968** when he wrote the (by now) classic paper **Go To Statement Considered Harmful**.
- Years of buffer overflows and ROP attacks have only confirmed his (early) intuition. ↳ Modify control flow!
- Today, no higher-level language (even C) allows arbitrary jumps/gotos.

17

Go To Statement Considered Harmful

E. W. Dijkstra

Key Words and Phrases

go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

***CR* Categories**

4.22, 5.23, 5.24

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

Originally published in Communications of the ACM, Volume 11 / Number 3 / March, 1968.

WASM: Arbitrary Jumps are Harmful

- WebAssembly adopts a **structured control flow** mechanism where the language only allows jumps to specific points in the program. (cannot jump to arbitrary code)
- Example: the WebAssembly program that infinitely increments a counter.

```
(func $inf_incr (local $x i32)
  ;; x = 0
  (i32.const 0)          [0]
  (set_local $x)         []

  (loop $incr_loop
    ;; x++
    (get_local $x)       [x]
    (i32.const 1)        [1,x]
    (i32.add)            [x+1]
    (set_local $x)       []

    (br $incr_loop)))
```

2 labels: 1 at the beginning of the loop, one at the end when we jump.

- When I declare a loop, I declare a label (at beginning, so when I jump at beginning I only repeat loop) and body of the loop.

WASM: Arbitraty Jumps are Harmful

- The **loop** construct defines a. new *label* (also called **break point**) **$incr_loop**.
- The **br $label** instruction unconditionally jumps (or break) to *label* (but only to that label)
- **incr_loop** is at the *top* of the loop, so jumping to it restarts the loop sequence.
- The loop construct is parameterized by a list of instructions. This is because the loop defines a *label scope*, i.e. a range within the **$incr_loop** label can be accessed.
- You can't, execute **br $incr_loop** outside of the loop.
  ↳ you only see the label from inside

```
(func $inf_incr (local $x i32)
  ;; x = 0
  (i32.const 0)
  (set_local $x)

  (loop $incr_loop
    ;; x++
    (get_local $x)
    (i32.const 1)
    (i32.add)
    (set_local $x)

    (br $incr_loop)))
```

WASM: Arbitraty Jumps are Harmful

- The complementary control flow construct to **loop** is **block**, which defines a label associated at the end of an instruction sequence.
- **br_if** instruction allows us to program *conditional* jumps.

```
(func $inf_incr (local $x i32)
  ;; x = 0
  (i32.const 0)
  (set_local $x)

  (block $incr_loop_break
    (loop $incr_loop
      ;; x++
      (get_local $x)
      (i32.const 1)
      (i32.add)
      (set_local $x)

      ;; if (x == 10) { break; }
      (get_local $x)
      (i32.const 10)
      (i32.eq)
      (br_if $incr_loop_break);

      (br $incr_loop))))
```

[0]
[ ]

- A block groups together as a unit entity a set of statements
- You group together statements. A block is characterized by a label.

Jumping at label of the block, that is the last instruction of the block, because we are in a loop.

C like syntax: 0 = false, otherwise we assume true.

Structured Control Flow in WASM

In WebAssembly (WASM), **labels** are an essential concept for **structured control flow**.

They are primarily used in **block-based control structures** like loops to define **jump targets** and manage control flow efficiently.

Unlike traditional assembly languages, WASM enforces a **structured control flow** model, preventing arbitrary jumps (e.g., goto) and ensuring **security and verifiability**.

What Are Labels in WASM?

Labels are implicit **identifiers** associated with **blocks** of code.

They define **where execution should resume** when breaking out of a block.

At runtime we don't have names and labels at runtime. We represent labels and variables with base + offset approach: low level!

WASM: anonymity

- When we write parameters, locals, and loops, we will always use names to increase the clarity of the program.
- WebAssembly also supports anonymity through automatic numbering of otherwise named things.

```
(block
 i32.const 10
 i32.const 20
 ....
) ;; Implicit label at the end of the block
```

Blocks write like this. Label automatically derived by the program

The **label is automatically placed** at the end of the block.
If a **br** (break) instruction targets this block, execution jumps to the end.

↳ br targets this label

WASM: anonymity

- When we write parameters, locals, and loops, we will always use names to increase the clarity of the program.
- WebAssembly also supports anonymity through automatic numbering of otherwise named things.

```
(loop
 i32.const 1
 ... ;; do_something
 br 0  ;; Jumps to the beginning of the loop (label 0)
)
```

The **label “0” is automatically placed** at the beginning of the loop.
br 0 refers to the label of the **innermost loop**, effectively continuing the loop.
Prevents unstructured jumps, enforcing **safe execution flow**.

* Index representation - Like a vector!

WASM Anonymity

Identify locals and param. with an array like fashion

- **Parameters and locals** are numbered left to right as they occur in the function definition, e.g. x, y, i maps to **0, 1, 2**. We have x corresponding to index 0, y to 1, 2 to variable 1.*
- **Labels** are numbered in increasing order from the closest scope to the outermost scope, e.g. **loop_start is 0 and loop_end is 1** in the inner scope in the example we discussed before.
- Remark (Foundations) this is identical to the idea of de Bruijn indices for unambiguous representation of variables in the lambda calculus!

Oh! So they are not absolute label, correct? If i do br 0 i jump at the closest label "going upwards" that i find, correct? I cannot jump "down"

Exactly! WebAssembly labels are **not absolute**, and `br <N>` always jumps **upward** in the control flow hierarchy. You can think of it as searching for the **Nth enclosing label while moving outward** from the current instruction.

Key Rules:

✅ `br 0` jumps to the **innermost enclosing control construct** (i.e., the closest `block`, `loop`, or `if`).
✅ `br 1` jumps to the **second enclosing control construct** (skipping the first).
✅ `br 2` jumps to the **third enclosing control construct**, and so on.
🚫 You **cannot** jump "down" into an inner block or loop—you can only jump outward.

↓

1. WebAssembly Uses a Control Stack for Labels

Instead of absolute jump addresses, WebAssembly maintains a **control stack** that keeps track of active blocks, loops, and function calls.

Each structured construct (`block`, `loop`, `if`) is **pushed onto the control stack** when execution enters it. The `br <N>` instruction tells WebAssembly to look at the **N-th item on the control stack** and jump there.

---

2. How `br N` Works at Runtime

- When executing `br N`, WebAssembly:
  1. **Counts upward through the control stack** to find the N-th enclosing control construct.
  2. **Jumps to that construct** based on its type:
     - If it's a `block`, execution **exits** the block and resumes **after** it.
     - If it's a `loop`, execution **restarts** at the beginning of the loop.

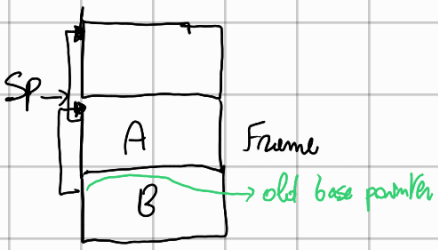
↓

Control Stack: The Hidden Stack for Structured Control Flow

WebAssembly has an **internal "Control Stack"**, which is **not the same as the operand stack**. This stack is used **exclusively** for tracking structured control flow (`block`, `loop`, `if`, `else`, and function calls).

Each time WebAssembly enters a new structured control construct, it **pushes a frame onto the control stack**. Each time a construct exits, WebAssembly **pops that frame**.

System automatically creates indexes to manage the loop.



WASM: anonymity

- When we write parameters, locals, and loops, we will always use names to increase the clarity of the program.
- WebAssembly also supports anonymity through automatic numbering of otherwise named things.

```
(func $foo2 (param i32) (param i32) (local i32)
  (block
    (loop
      (get_local 0)
      (get_local 1)
      (i32.add)
      (set_local 2)

      (get_local 2)
      (br_if 1)

      (br 0))))
```

Real syntax

```
(func $foo1 (param $x i32) (param $y i32) (local $i i32)
  (block $loop_end
    (loop $loop_start
      (get_local $x)
      (get_local $y)
      (i32.add)
      (set_local $i)

      (get_local $i)
      (br_if $loop_end)

      (br $loop_start))))
```

• Si può usare do loop senza blocchi? Come facciamo a sapere quanti bl. annestati abbiamo?

```
block 3
    (loop 2
        (
            block 1
                (loop 0
                    (
                        ...
                    )
                )
            )
        )
```

How many steps u need to make to reach position of name/label.

At the beginning the level of nesting is 0 and 1 is beyond

Level of nesting is generated by counting how many nested blocks I traverse.

But with branching you step out of the nested block

Understanding Blocks and Labels

```
(func $example
  (block $outer  ;; Label 0
    (block $inner  ;; Label 1
      br 1
    ;; This code runs if `br 1` is not taken
  )
)
```

block $outer creates a block, and it **implicitly** gets label **0** (innermost)
block $inner is nested inside $outer, and it is assigned label **1**

When a br N is executed:N refers to the **nesting level of blocks**.

br 0 would jump out of the **innermost** block ($inner).
br 1 jumps out of the **second** enclosing block ($outer).

Remark: the nesting level of blocks resamable the static chain!!!

Understanding Blocks and Labels

```
(func $example
 (block $outer  ;; Label 0
  (block $inner  ;; Label 1
                 :
   br 1
  ;; This code runs if `br 1` is not taken
  )
 )
)
```

Executes the instruction outside $outer

Understanding Blocks and Labels

```
(func $example
  (block $outer  ;; Label 0
    (block $inner  ;; Label 1
                :
      br 1
    ;; This code runs if `br 1` is not taken
    )
  )
)
```

Step 1: Enter block $outer

- Execution starts in the **outer block**.
- It enters the **inner block** immediately after.

Step 2: Enter block $inner

- Execution enters the block $inner, the **most nested block**.
- The br 1 instruction is encountered.

Step 3: Execute br 1

- br 1 means **break to the second enclosing block** (counting outwards):
  - 0 is **$inner**, **1 is $outer**.
- The program **jumps out of both blocks** and **skips all remaining instructions** inside them.

Step 4: Exit the Blocks

- The function **exits early** because br 1 bypasses everything after the block $inner.

What happens if …

```
(func $example
 (block $outer
  (block $inner
   br 0   ;; Now jumps to the end of $inner only
  )
 )
)
```

What happens if ...

```
(func $example
 (block $outer
  (block $inner
   br 0   ;; Now jumps to the end of $inner only
  )
  ;; This code still executes!
 )
)
```

Labels and Security

- WebAssembly's use of labels ensures:
- **Structured control flow** – No arbitrary jumps or goto behavior.
- **Prevention of stack corruption** – Labels enforce safe stack-based execution.①
- **Formal verification** – Since control flow is structured, security tools can verify WASM execution paths. You can identify paths corresponding to legal execution

① Side effect; with buffer overflow attacks you could modify the return point of a function. A structured control flow provides protection.

WASM: call and return

- Functions in WebAssembly are top-level constructs (no nested functions).
- Functions can declare local variables only accessible inside the function.
- Similar to register-machines, arguments to a function are pushed onto the value stack. The top value on the stack becomes the first argument to the function.
- Parameters are pushed onto the stack by the caller, the callee must access them through **get_local** just like local variables.
- A function's return value is implicitly the value at the top of the stack when the function exits.

```
(func $incr (param $x i32) (result i32)
  (get_local $x)
  (i32.const 1)
  (i32.add))

(func $f (result i32)
  (i32.const 1)
  (call $incr))
```

WASM SEMANTICS

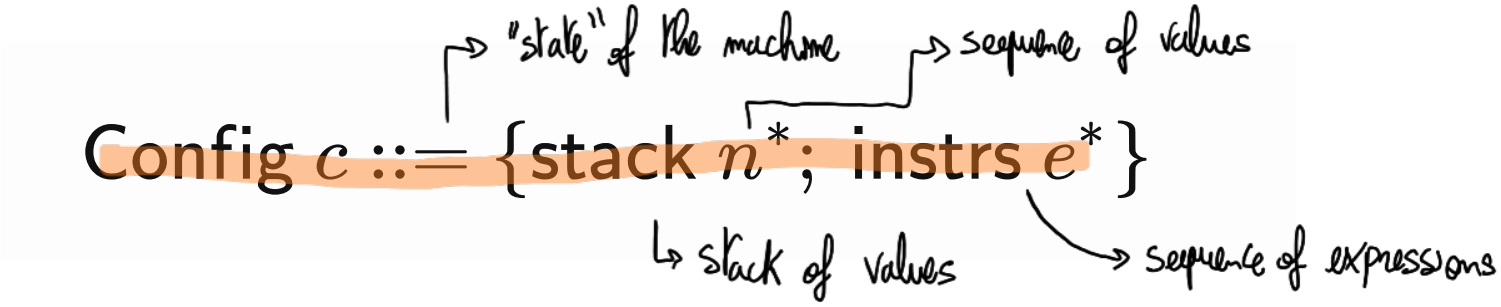
The wasm semantics will be illustrated in the context of *the formalisms seen so far in the lecture*.

WASM: a small subset

- We first consider a subset of WebAssembly where a program is just a sequence of instructions, no modules or functions.

Expression e ::=		i32.const n	constant
	|	i32. *binop*	binary operators
	|	i32. *relop*	relational operators
Binop *binop* ::=		add	addition
	|	sub	subtraction
	|	mul	multiplication
	|	div_s	division
Relop *relop* ::=		eq	equals
	|	lt_s	less than
	|	gt_s	greater than

Run-Time Configurations



A configuration is a record consisting of a **value stack** and a **sequence** of instructions.

The operational rules

$$\frac{}{\{\text{stack } n^*;\ \text{instrs } (\text{i32.const } n'), e^*\} \mapsto \{\text{stack } n', n^*;\ \text{instrs } e^*\}}\ (\text{D-Const})$$

Rule name

Declaration of const

Initial config

Pushing n' on the stack

The rule reads as follows:
given some value **stack n∗,** when the first instruction in our program is **i32.const n′** followed by some **e∗**,
then put that value on the stack and remove the instruction from the sequence

Are we satisfied?

↑ takes full structure of a state ①

- The operational rule is on the entire configuration of the WASM abstract machine.
- It would be better to have rules that chatacterize only *changes* to the configuration.
- The simplified rule will take the form

$$\frac{}{\{\text{instrs i32.const } n\} \hookrightarrow \{\text{stack } n\}}\ (\text{D-Const})$$

② ① But we simply put a value on top of the stack. So:

② When we find an instruction for a constant, we simply put it on top of the stack.

D-Const rule

$$\frac{}{\{\mathsf{instrs}\ \mathsf{i32.const}\ n\} \mapsto \{\mathsf{stack}\ n\}}\ (\text{D-Const})$$

The **i32.const n** instruction does not appear on the right side: we can assume that has been removed.

D-Const rule

$$\frac{}{\{\mathsf{instrs}\ \mathsf{i32.const}\ n\} \mapsto \{\mathsf{stack}\ n\}}\ (\text{D-Const})$$

The **i32.const n** instruction does not appear on the right side: we can assume that has been removed.

The **stack** does not appear on the left side, the value **n** is being pushed onto whatever existing values are on the stack.

Evaluation context

This type of execution rules, we informally applied, are so common in the theory of programming languages.

Evaluation context

imagine it as a program with a hole. You want to understand the behaviour when you fill it with specific instructions

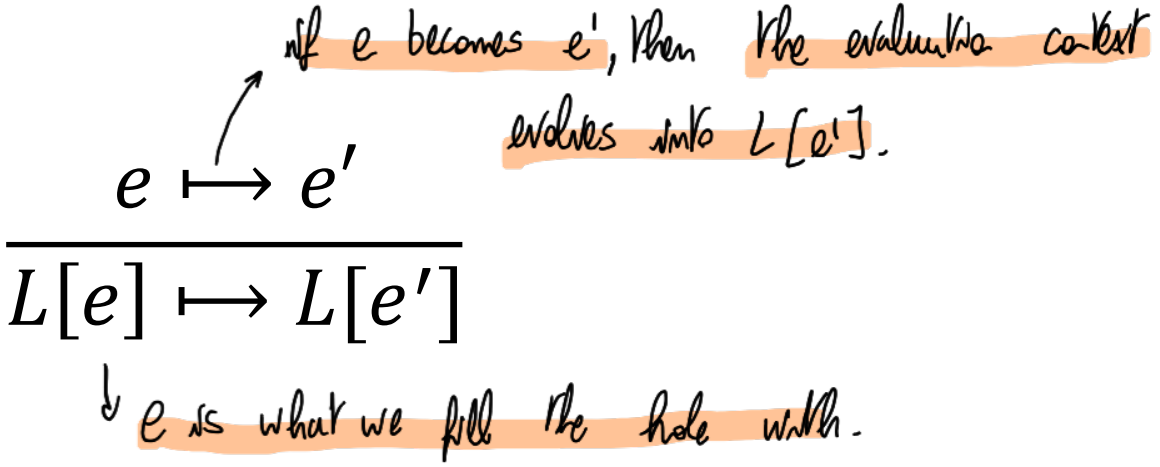
Evaluation Context $L ::= v * [-] e *$

↳ sequence of values already produced

Evaluation contexts are **incomplete programs** with a **hole** [—] inside

A **complete program** is created if the hole is filled with a **complete program** e

Context Rule



Intuition

Find the instruction(s) e
Rewrite e using the rules
Replace the result in the original program.

Details

The PLDI2017 paper on WASM design available on TEAMS.

Rules for Binary Operations

$$\frac{n_3 = n_1\ binop\ n_2}{\{\mathsf{stack}\ n_2, n_1;\ \mathsf{instrs}\ \mathsf{i32}.binop\} \mapsto \{\mathsf{stack}\ n_3\}}\ (\text{D-Binop})$$

$$\frac{}{\{\text{stack } n^*;\ \text{instrs } (\text{i32.const } n'), e^*\} \mapsto \{\text{stack } n', n^*;\ \text{instrs } e^*\}}\ (\text{D-Const})$$

↳ apply implementation of underlying machine of binary operation

$$\frac{n_3 = n_1\ binop\ n_2}{\{\text{stack } n_2, n_1;\ \text{instrs i32.}\ binop\} \mapsto \{\text{stack } n_3\}}\ (\text{D-Binop})$$

↳ n_1 XOR n_2, we find n_1 first and push it, so n_2 is at the top and is the second variable

Using the operational we presented rules, we can prove the correctness of the following trace

$$\begin{aligned}
& \{\text{stack } \varepsilon;\ \text{instrs i32.const } 3,\ \text{i32.const } 5,\ \text{i32.sub}\} \\
\mapsto\ & \{\text{stack } 3;\ \text{instrs i32.const } 5,\ \text{i32.sub}\} && (\text{D-Const}) \\
\mapsto\ & \{\text{stack } 5, 3;\ \text{instrs i32.sub}\} && (\text{D-Const}) \\
\mapsto\ & \{\text{stack } -2;\ \text{instrs } \varepsilon\} && (\text{D-Binop})
\end{aligned}$$

$$\frac{}{\{\mathsf{stack}\ n^*;\ \mathsf{instrs}\ (\mathsf{i32.const}\ n'), e^*\} \mapsto \{\mathsf{stack}\ n', n^*;\ \mathsf{instrs}\ e^*\}}\ (\text{D-Const})$$

$$\frac{n_3 = n_1\ binop\ n_2}{\{\mathsf{stack}\ n_2, n_1;\ \mathsf{instrs}\ \mathsf{i32}.\ binop\} \mapsto \{\mathsf{stack}\ n_3\}}\ (\text{D-Binop})$$

$$\begin{aligned}
&\{\mathsf{stack}\ \varepsilon;\ \mathsf{instrs}\ \mathsf{i32.const}\ 3,\ \mathsf{i32.const}\ 5,\ \mathsf{i32.sub}\} \\
\mapsto\ &\{\mathsf{stack}\ 3;\ \mathsf{instrs}\ \mathsf{i32.const}\ 5,\ \mathsf{i32.sub}\} && (\text{D-Const}) \\
\mapsto\ &\{\mathsf{stack}\ 5, 3;\ \mathsf{instrs}\ \mathsf{i32.sub}\} && (\text{D-Const}) \\
\mapsto\ &\{\mathsf{stack}\ -2;\ \mathsf{instrs}\ \varepsilon\} && (\text{D-Binop})
\end{aligned}$$

	$(stack\ \epsilon, (\mathbf{i32.const\ 3}, i32.const\ 5, i32.sub))$	
$\mapsto$	$(stack\ \mathbf{3}, (i32.const\ 5, i32.sub))$	D-Const

$stack\ 3$

$$\frac{}{\{\text{stack } n^*;\ \text{instrs } (\text{i32.const } n'), e^*\} \mapsto \{\text{stack } n', n^*;\ \text{instrs } e^*\}}\ (\text{D-Const})$$

$$\frac{n_3 = n_1\ binop\ n_2}{\{\text{stack } n_2, n_1;\ \text{instrs i32.}\ binop\} \mapsto \{\text{stack } n_3\}}\ (\text{D-Binop})$$

$$\begin{array}{ll} & \{\text{stack } \varepsilon;\ \text{instrs i32.const } 3,\ \text{i32.const } 5,\ \text{i32. sub}\} \\ \mapsto & \{\text{stack } 3;\ \text{instrs i32.const } 5,\ \text{i32. sub}\} \quad (\text{D-Const}) \\ \mapsto & \{\text{stack } 5, 3;\ \text{instrs i32. sub}\} \quad (\text{D-Const}) \\ \mapsto & \{\text{stack } -2;\ \text{instrs } \varepsilon\} \quad (\text{D-Binop}) \end{array}$$

	$(stack\ \epsilon, (i32.const\ 3, i32.const\ 5, i32.sub))$	
$\mapsto$	$(stack\ 3, (\mathbf{i32.const\ 5}, i32.sub))$	D-Const
$\mapsto$	$(stack\ \mathbf{5}, 3\ (i32.sub))$	D-Const

$stack\ 5$

$$\frac{}{\{\text{stack } n^*;\ \text{instrs } (\text{i32.const } n'), e^*\} \mapsto \{\text{stack } n', n^*;\ \text{instrs } e^*\}}\ (\text{D-Const})$$

$$\frac{n_3 = n_1\ binop\ n_2}{\{\text{stack } n_2, n_1;\ \text{instrs i32.}\ binop\} \mapsto \{\text{stack } n_3\}}\ (\text{D-Binop})$$

$$\begin{array}{lll}
 & \{\text{stack } \varepsilon;\ \text{instrs i32.const } 3,\ \text{i32.const } 5,\ \text{i32.sub}\} & \\
\mapsto & \{\text{stack } 3;\ \text{instrs i32.const } 5,\ \text{i32.sub}\} & (\text{D-Const}) \\
\mapsto & \{\text{stack } 5, 3;\ \text{instrs i32.sub}\} & (\text{D-Const}) \\
\mapsto & \{\text{stack } -2;\ \text{instrs } \varepsilon\} & (\text{D-Binop})
\end{array}$$

	$(stack\ \epsilon, (i32.const\ 3, i32.const\ 5, i32.sub))$	
$\mapsto$	$(stack\ 3, (i32.const\ 5, i32.sub))$	D-Const
$\mapsto$	$(stack\ 5, 3\ (\boldsymbol{i32.sub}))$	D-Const
$\mapsto$	$(stack\ -\mathbf{2}, \epsilon)$	D-Binop

$$\frac{3 - 5 = -2}{stack\ -2}$$

Memory

The value stack represents a separate data store from a WASM program, the program can modify the value stack indirectly through instructions.

Memory can be accessed/modified directly through the load and store instructions.

Our simplified model of WebAssembly, the memory is viewed as a sequence of 32-bit integers, initially of length zero.

We assume the memory like a sort of array with length

Memory instructions

Expression e ::=	...		
	|	i32.load	load from memory
	|	i32.store	store to memory
	|	memory. size	get memory size
	|	memory. grow	increase memory size

Configurations include memory

$$\text{Config } c ::= \{\text{mem } n^*_{\text{mem}};\ \text{stack } n^*_{\text{stack}};\ \text{instrs } e^*\}$$

Notation:

m[i] is used to represent the i-th number in the sequence m,

m with i=n is used to represent the sequence m with the i-th number changed to n

|m| indicates the length of the sequence m.

Operanting over the memory

$$\frac{0 \leq i < |n^*_{\mathsf{mem}}|}{\{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ i;\ \mathsf{instrs\ i32.load}\} \mapsto \{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ n^*_{\mathsf{mem}}[i]\}}\ \text{(D-Load-Valid)}$$

This rule says that
if the **requested index i is within the bounds of memory**,
then a load instruction from an index i (retrieved from the stack)
pushes m[i] to the stack with the **memory unchanged.**

Memory Instructions

↳ we are trying to get a value from an invalid address

$$\frac{i \geq |n^*_{\mathsf{mem}}|}{\{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ i;\ \mathsf{instrs}\ \mathsf{i32.load}\} \mapsto \{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{instrs}\ \mathsf{trapping}\}} \text{ (D-Load-Invalid)}$$

↳ abnormal behavior

$$\frac{0 \leq i < |n^*_{\mathsf{mem}}|}{\{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ n, i;\ \mathsf{instrs}\ \mathsf{i32.store}\} \mapsto \{\mathsf{mem}\ n^*_{\mathsf{mem}}\ \mathsf{with}\ i = n\}} \text{ (D-Store-Valid)}$$

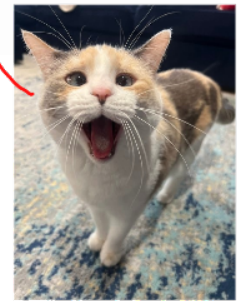
$$\frac{i \geq |n^*_{\mathsf{mem}}|}{\{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ n, i;\ \mathsf{instrs}\ \mathsf{i32.store}\} \mapsto \{\mathsf{mem}\ n^*_{\mathsf{mem}}; \mathsf{instrs}\ \mathsf{trapping}\}} \text{ (D-Store-Invalid)}$$

$$\frac{}{\{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{instrs}\ \mathsf{memory.size}\} \mapsto \{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ |n^*_{\mathsf{mem}}|\}} \text{ (D-Size)}$$

$$\frac{}{\{\mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{instrs}\ \mathsf{memory.grow}\} \mapsto \{\mathsf{mem}\ n^*_{\mathsf{mem}}, 0^{16384}\}} \text{ (D-Grow)}$$

↓ you cannot specify how much memory grows

↓ Values are put to a default which is 0.



CONTROL FLOW

Loops and Blocks

Rewiew: Using **blocks** and **loops**,
we can create nested instruction sequences
that can **break** to defined **labels** (**jump points**).

```
void inf_incr() {
  int x = 0;
  while (true) {
    x += 1;
    if (x == 10) { break; }
  }
}
```

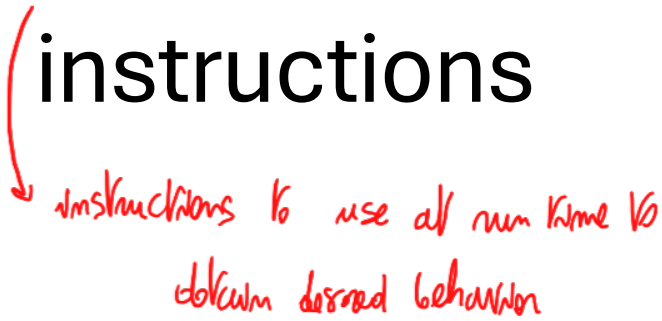
from C to WASM

```
(func $inf_incr (local $x i32)
  ;; x = 0
  (i32.const 0)
  (set_local $x)

  (block $incr_loop_break
    (loop $incr_loop
      ;; x++
      (get_local $x)
      (i32.const 1)
      (i32.add)
      (set_local $x)

      ;; if (x == 10) { break; }
      (get_local $x)
      (i32.const 10)
      (i32.eq)
      (br_if $incr_loop_break);

      (br $incr_loop))))
```

Control flow instructions

Expression e ::=	...	
|	block e^*	label at end
|	loop e^*	label at start
|	br i	branch
|	br_if i	conditional branch

A trick: administrative instructions



To formally encode the semantics of jumps, we need to add another language construct: labels, or jump/branch points.

But what about labels?

We express labels as an expression that is part of the language, but not meant to be written by the user, only used by the runtime to encode the state of the interpreter.

The interpreter adopts the nameless code

You have to work precisely with labels

Branch point

Expression $e ::= \ldots$

$\mid$ label $\{e^*_{\text{cont}}\}\ (n^*;\ e^*_{\text{body}})$ branch point

introduce construct at runtime

local stack

called

e continuation

body

$\mathcal{E}$ if you have a block

Intuition

A label defines a *local stack* n^*, which is a value stack only **accessible in code executing inside the scope of the label**.

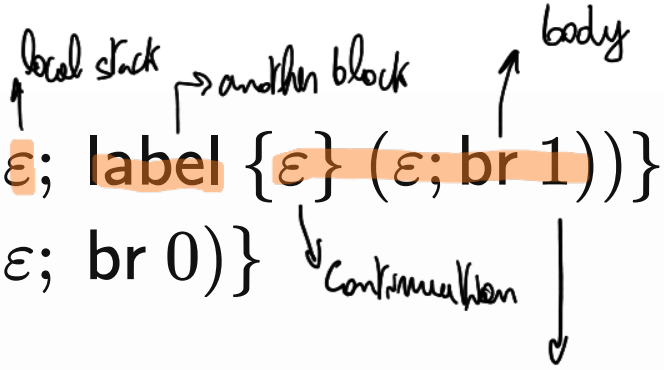
The codes comes with two sequences of instructions: e^*_{body} which is the current set of instructions being executed, and e^*_{cont} which is the "continuation" of label, or the sequence of instructions that will be run if the label is jumped to

Local stack for local variables declaration

Example

label $\{e^*_{\text{cont}}\}$ $(n^*;\ e^*_{\text{body}})$ branch point

$$
\begin{aligned}
& \{\text{stack } \varepsilon;\ \text{instrs label } \{\text{i32.const } 2\}\ (\varepsilon;\ \text{label } \{\varepsilon\}\ (\varepsilon;\ \text{br } 1))\} \\
\mapsto\ & \{\text{stack } \varepsilon;\ \text{instrs label } \{\text{i32.const } 2\}\ (\varepsilon;\ \text{br } 0)\} \\
\mapsto\ & \{\text{stack } \varepsilon;\ \text{instrs i32.const } 2\} \\
\mapsto\ & \{\text{stack } 2;\ \text{instrs } \varepsilon\}
\end{aligned}
$$



execution of inner block. br 1 is for outer block. I decrement 1 to 0 and execute br 0. We do not execute cont. of inner block. We move out and do br 0.

The operational rules (1)

$$\frac{i > 0}{\{\text{instrs label } \{e^*_{\text{cont}}\}\ (\_;\ (\text{br } i), \_)\} \mapsto \{\text{instrs br } i - 1\}}\ \text{(D-Breaking-Continue)}$$

e^*_{cont}: continuation

$\_$: doesn't matter. branch does not need any next expression or local stack.

The rule says that when the first break instruction under a label is a found, then check what number is in the branch.
If it is greater than zero, we toss away the current label and continue breaking up the label stack.

The Operational Rules (2)

$$\frac{}{\{\mathsf{instrs\ label}\ \{e^*_{\mathsf{cont}}\}\ (\_;\ (\mathsf{br}\ 0), \_)\} \mapsto \{\mathsf{instrs}\ e^*_{\mathsf{cont}}\}}\ (\text{D-Breaking-Done})$$

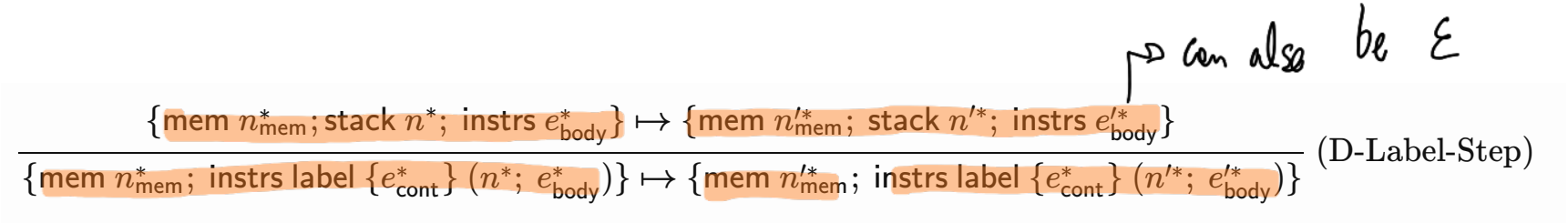
The rule says when when the first break instruction under a label is a found, then check what number is in the branch. If it is **zero** then we step into the continuation.

Summary (1+2)

The two rules say that when the first break instruction under a label is found, then check what number is in the branch.

If it is greater than zero, we toss away the current label and continue breaking up the label stack. When we reach the label we are breaking to (number equal zero), then we step into the continuation.

The Operational Rules (3)



The rule describes what happens when a label is executing any other instruction besides a break.
The rule rule says to step the inside of the label, but replacing the stack with the label's custom private stack.

The Operational Rules (4)

$$\frac{}{\{\text{instrs label } \{\_\}\ (\_;\ \varepsilon)\} \mapsto \{\}}\ (\text{D-Label-Done})$$

↳ we go to continuation only if you branch out

The rule says that if we execute all the instructions in a label without breaking, then we throw away the label.

The Administrative Encoding

$$\frac{}{\{\text{instrs block } e^*\} \mapsto \{\text{instrs label } \{\varepsilon\}\ (\varepsilon;\ e^*)\}}\ (\text{D-Block})$$

$$\frac{}{\{\text{instrs loop } e^*\} \mapsto \{\text{instrs label } \{\text{loop } e^*\}\ (\varepsilon;\ e^*)\}}\ (\text{D-Loop})$$

$$\frac{n = 0}{\{\text{stack } n;\ \text{instrs br\_if } i\} \mapsto \{\}}\ (\text{D-BrIf-False}) \qquad \frac{n \neq 0}{\{\text{stack } n;\ \text{instrs br\_if } i\} \mapsto \{\text{instrs br } i\}}\ (\text{D-BrIf-True})$$

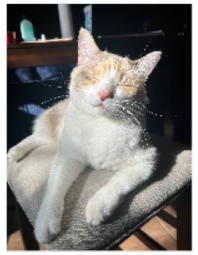
The translation of loops/blocks/conditional breaks into the core “label” primitives

$$\frac{n = 0}{\{\text{stack } n;\ \text{instrs br\_if } i\} \mapsto \{\}}\ (\text{D-BrIf-False}) \qquad \frac{n \neq 0}{\{\text{stack } n;\ \text{instrs br\_if } i\} \mapsto \{\text{instrs br } i\}}\ (\text{D-BrIf-True})$$

Would be better to have booleans! But whatever

Weak point. We know you want to be close to machine, but ye

No comment???



Remark

$$\frac{}{\{\mathsf{instrs\ loop}\ e^*\} \mapsto \{\mathsf{instrs\ label}\ \{\mathsf{loop}\ e^*\}\ (\varepsilon;\ e^*)\}}\ (\text{D-Loop})$$

When we encounter a loop, the rule allow to move to a label where the continuation is the *loop itself*, meaning jumping back to a label generated by a loop permits infinite recursion.

Functions

Expression e ::=	...	
|	get_local i	get frame local
|	set_local i	set frame local
|	call i	function call
|	return	function return
Function f ::=	{params i; locals j; body e^*}	
Module m ::=	{funcs f^*}	

global entries. Not possible to declare a funct. in another function

Modules & Configurations

↑ all the functions declared

- A module is a sequence of functions, and a function has some number of parameters and locals along with a function body, which is a sequence of instructions.
- Since the persistent memory belongs to runtime configuration, we also need to have functions in our configuration as well. This leads to our full, final configuration:

$$\text{Config } c ::= \quad \{\text{module } m;\ \text{mem } n^*_{\text{mem}};\ \text{locals } n^*_{\text{locals}};\ \text{stack } n^*_{\text{stack}};\ \text{instrs } e^*\}$$

↑ Main might need parameters as well

Configurations

$$\text{Config } c ::= \quad \{\text{module } m;\ \text{mem } n^*_{\text{mem}};\ \text{locals } n^*_{\text{locals}};\ \text{stack } n^*_{\text{stack}};\ \text{instrs } e^*\}$$

The **runtime configurations** consist of

- **module** (set of functions),
- **memory** (sequence of numbers),
- **locals** (also a sequence of numbers),
- **value stack**,
- **the current instruction sequence**.

Administrative instructions

In the same way we introduced a label construct to represent a stack of labels at runtime, we introduce two administrative instruction for dealing with functions.

$$
\begin{array}{lll}
\text{Expression } e ::= & \ldots & \\
\mid & \text{frame } c & \text{Function call frame} \\
\mid & \text{returning } n & \text{Active return}
\end{array}
$$

Function Call

$$\frac{m.\,\mathsf{funcs}[i] = \{\mathsf{params}\ j;\ \mathsf{locals}\ k;\ \mathsf{body}\ e^*\} \qquad c = \{\mathsf{module}\ m;\ \mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{locals}\ \mathsf{rev}(n^j_{\mathsf{stack}}), 0^k;\ \mathsf{stack}\ \varepsilon;\ \mathsf{instrs}\ e^*\}}{\{\mathsf{module}\ m;\ \mathsf{mem}\ n^*_{\mathsf{mem}};\ \mathsf{stack}\ n^j_{\mathsf{stack}};\ \mathsf{instrs}\ \mathsf{call}\ i\} \mapsto \{\mathsf{module}\ m;\ \mathsf{instrs}\ \mathsf{frame}\ c\}}\ (\text{D-Call})$$

parameters are put in reverse order of declaration. 0 is the default value we have

PROLOGUE

Intuition.
when we call a function **i** we have:

- look up the corresponding function definition in the module, ***m.funcs[i]***,
- create a new configuration ***c*** with the same module and memory, but with a new set of local (the j-parameters taken from the stack),
- and k 0-values for local variables.

Start execution with an empty value stack in this new configuration, and a set of instructions taken from the function definition.
Place this configuration into a call frame instruction.

Remark: we reverse the arguments, by function “rev”, from the stack, as the top element of the stack is the last argument of the function.

Managing function frames

$$\frac{c \mapsto c' \quad \text{frame evolves from } c \text{ to } c'}{\{\text{mem } \_;\ \text{instrs frame } c\} \mapsto \{\text{mem } c'.\text{mem};\ \text{instrs frame } c'\}}\ \text{(D-Frame-Step)}$$

$$\frac{}{\{\text{instrs frame } \{\text{stack } n;\ \text{instrs } \varepsilon\}\} \mapsto \{\text{stack } n\}}\ \text{(D-Frame-Done)}$$

During function execution, if the function body still has work to do , then step its configuration (Rule D-Frame-Step).
Once the function has no more work to do, i.e. the instruction sequence is empty, then take the return value at the top of the inner configuration and put it on the stack of the outer configuration.

Early Returns from Functions

$$\frac{}{\{\text{stack } n;\ \text{instrs return}\} \mapsto \{\text{instrs returning } n\}}\ \text{(D-Return)}$$

$$\frac{}{\{\text{instrs label } \{\_\}\ (\_;\ \text{returning } n)\} \mapsto \{\text{instrs returning } n\}}\ \text{(D-Label-Returning)}$$

$$\frac{}{\{\text{instrs frame } \{\text{stack } \_;\ \text{instrs returning } n\}\} \mapsto \{\text{instrs frame } \{\text{stack } n;\ \text{instrs } \varepsilon\}\}}\ \text{(D-Frame-Returning)}$$

Why???

We need separate “return” and “returning” instructions because the returning instruction has the return value packaged in the instruction, and it can hold onto it across many stacks that it might propagate across.

Discussion

Kernel

- We covered the core of WebAssembly. However, the semantics presented both **exclude** some features and **illustrtes** others in a different way than the standard specification.
- The complete specs is on TEAMS
- The operational rulese presented have been inspired by the official WebAssembly interpreter (written in OCaml!)
- https://github.com/WebAssembly/spec/tree/main/interpreter

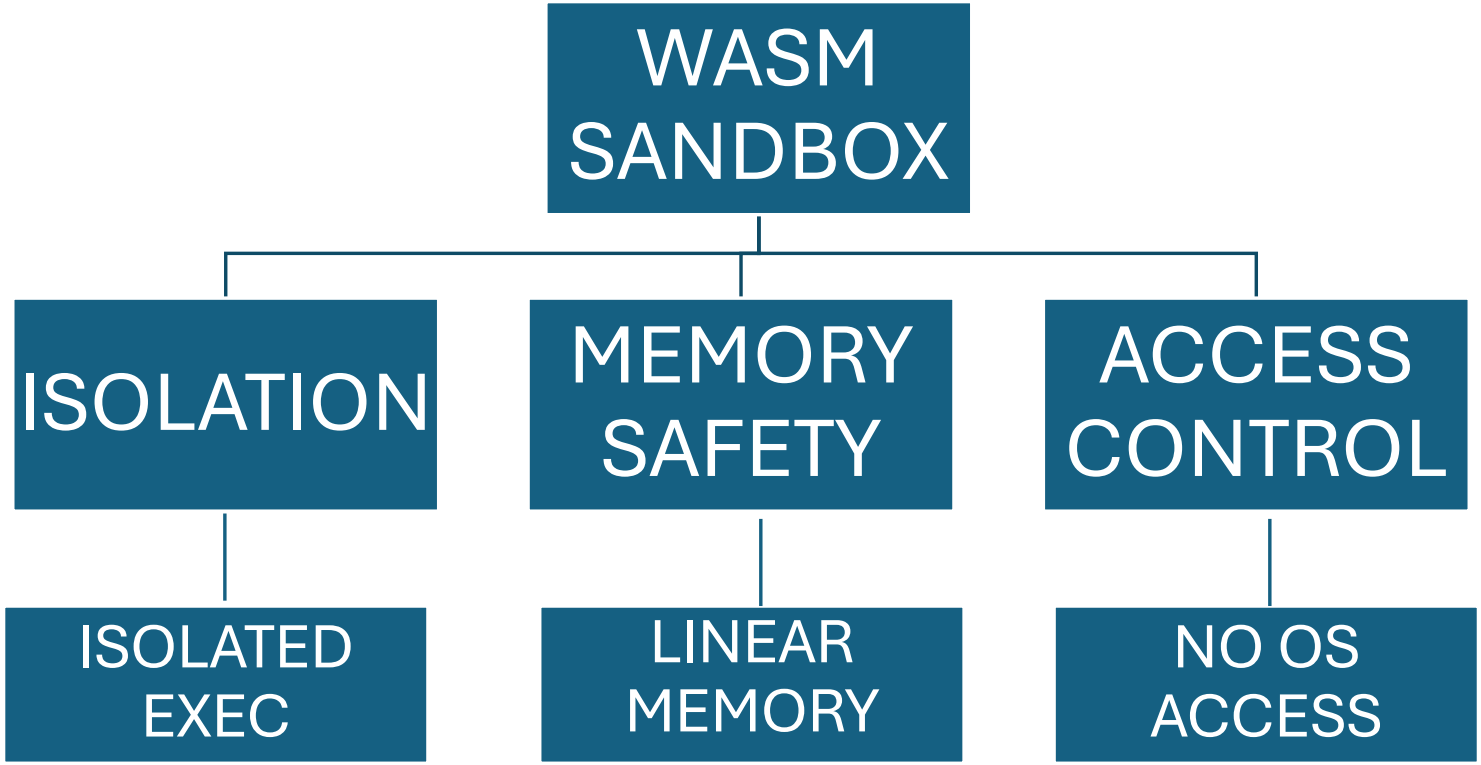
WASM vs Native Code

	WebAssembly (WASM)	Native Code
Memory Safety	Yes Load and store enforce it	No
Sandboxing	Yes	No
Direct OS Access	No Forbidden	Yes
Performance	Near-native	✅ High

WASM Security

- **Isolated Execution**
  - Ensures no direct interference with the host system.
- **Linear Memory Model**
  - Prevents buffer overflows and unauthorized access.
- **Capability-Based Security**
  - Restricts API and system call access.
- **No Direct OS Access**
  - Prevents malicious system-level interactions.

WASM SECURITY



Evolution

- **WASI (WebAssembly System Interface)** is a standardized API that extends WebAssembly beyond the browser, enabling it to interact securely with the underlying system. Interface to program at level of systems
- WASI provides a **sandboxed POSIX-like interface** that allows WASM programs to run **safely on servers, edge devices, and operating systems** without compromising security.

WASI

	Description
Sandboxed Execution	Limits access to host system resources to prevent security risks.
Capability-Based Security	Programs can only access explicitly granted resources (e.g., files, sockets).
Portable & Cross-Platform	Runs across OS environments (Linux, macOS, Windows).
Efficient	No need for full system emulation, improving performance.
Modular API Design	Expands functionality over time with controlled access to new system features.

WASI vs SYSTEM CALLS

	Traditional OS (POSIX)	WASI
System Access	Full access	Limited, permission-based
Security	High risk of exploits	Sandboxed execution
Portability	OS-dependent	Runs anywhere with WASI support
Performance	Varies	Optimized for lightweight execution