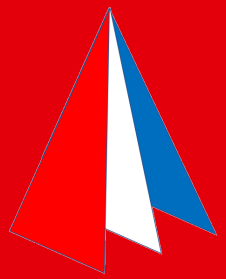
Quick Clean for the Web

Dante van Gemert

Thursday 5th February 2026



Radboud Universiteit

Web is where it’s at

E-mail, Google Docs, Typst, custom keyboard software, …

But: JavaScript 😬

Clean

Functional programming language

Clean

Functional programming language

iTasks

- Structure code with tasks
- Automagically generates web UI ✨

Clean

Functional programming language

iTasks

- Structure code with tasks
- Automagically generates web UI ✨
- Program runs on server
- Small bits of Clean code in browser for creating UI elements

Clean

Functional programming language

iTasks

- Structure code with tasks
- Automagically generates web UI ✨
- Program runs on server
- Small bits of Clean code in browser for creating UI elements
- Beneficial to run more in browser
  - Reduce latency, increase availability

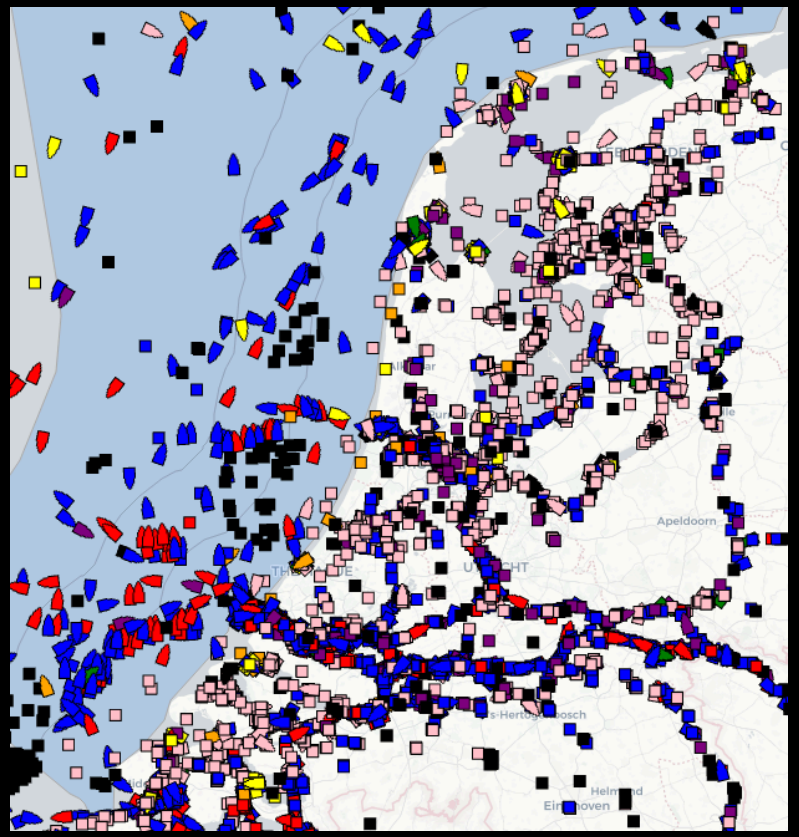
Clean

Functional programming language

iTasks

- Structure code with tasks
- Automagically generates web UI ✨
- Program runs on server
- Small bits of Clean code in browser for creating UI elements
- Beneficial to run more in browser
  - Reduce latency, increase availability

But: interpreting is too slow



VIIA

Clean-LLVM

Compile instead of interpret

Clean-LLVM

Compile instead of interpret

In short

- Clean web-app → iTasks
- Better performance → Clean-LLVM (instead of ABC interpreter)

Clean-LLVM

Compile instead of interpret

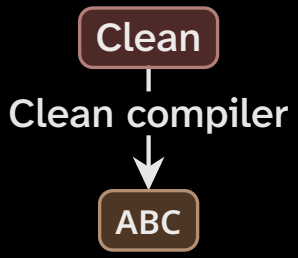
In short

- Clean web-app ⟶ iTasks
- Better performance ⟶ Clean-LLVM (instead of ABC interpreter)

But: don’t want to compile Clean code directly

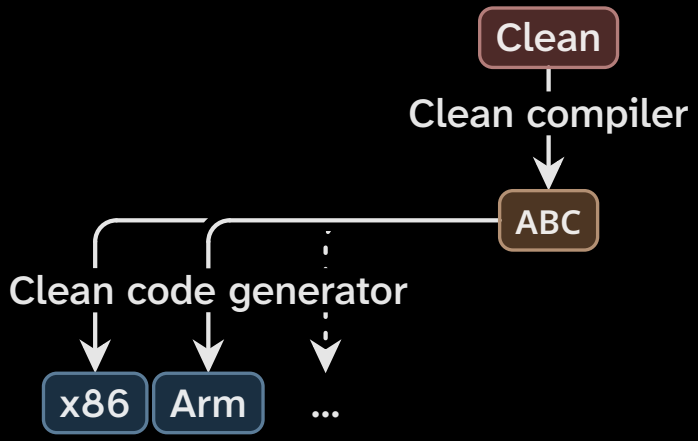
ABC

Use Clean compiler's ABC bytecode



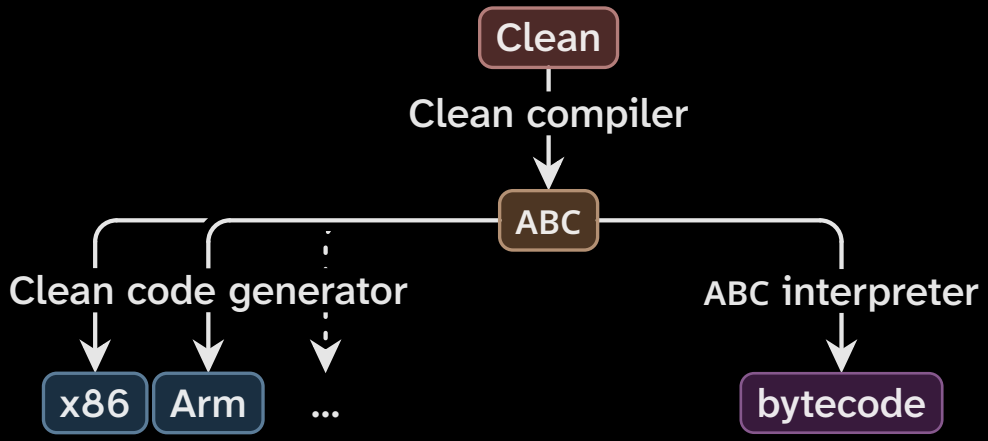
ABC

Use Clean compiler's ABC bytecode



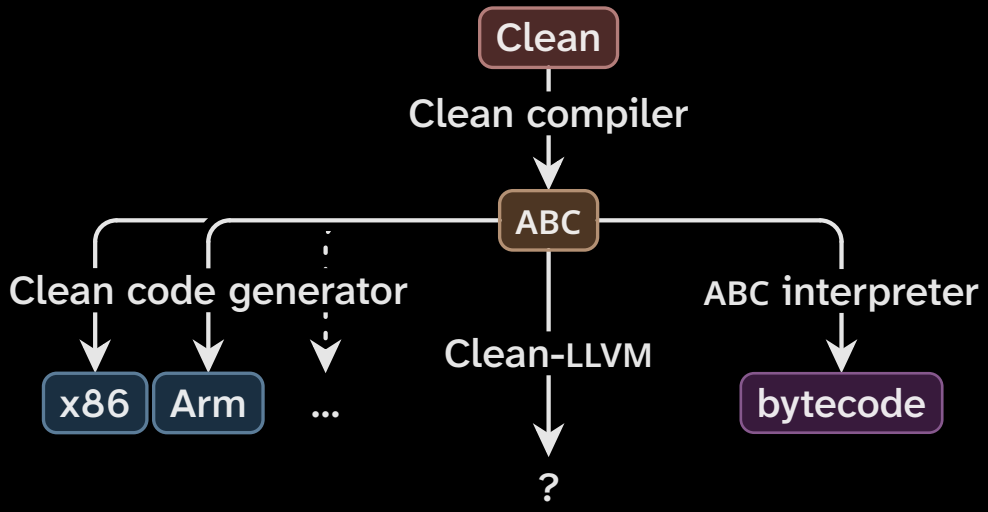
ABC

Use Clean compiler's ABC bytecode



ABC

Use Clean compiler's ABC bytecode



What do we compile **to**?

WebAssembly (Wasm)

Clean → ABC → ? → WebAssembly

- Compilation target for the web
- Stack based
- Safe & secure (→ restrictive)
- C(++), Rust, Go, Kotlin, Dart, …

WebAssembly (Wasm)

Clean → ABC → ? → WebAssembly

- Compilation target for the web
- Stack based
- Safe & secure (→ restrictive)
- C(++), Rust, Go, Kotlin, Dart, …

Originally focussed on imperative languages

- Recently: tail calls

WebAssembly (Wasm)

Clean → ABC → ? → WebAssembly

- Compilation target for the web
- Stack based
- Safe & secure (→ restrictive)
- C(++), Rust, Go, Kotlin, Dart, …

Originally focussed on imperative languages

- Recently: tail calls

But: we can do better than generating Wasm directly

LLVM

- Compiler toolkit
- Clang compiler for C & C++
- Intermediate representation (IR)
- Optimisation passes (inlining, dead code elimination, …)
- Rust, Zig, Julia, Swift, …

LLVM

- Compiler toolkit
- Clang compiler for C & C++
- Intermediate representation (IR)
- Optimisation passes (inlining, dead code elimination, ...)
- Rust, Zig, Julia, Swift, ...

Why?

- Generates Wasm, but also x86, Arm, RISC-V...
- Optimisations

LLVM

- Compiler toolkit
- Clang compiler for C & C++
- Intermediate representation (IR)
- Optimisation passes (inlining, dead code elimination, ...)
- Rust, Zig, Julia, Swift, ...

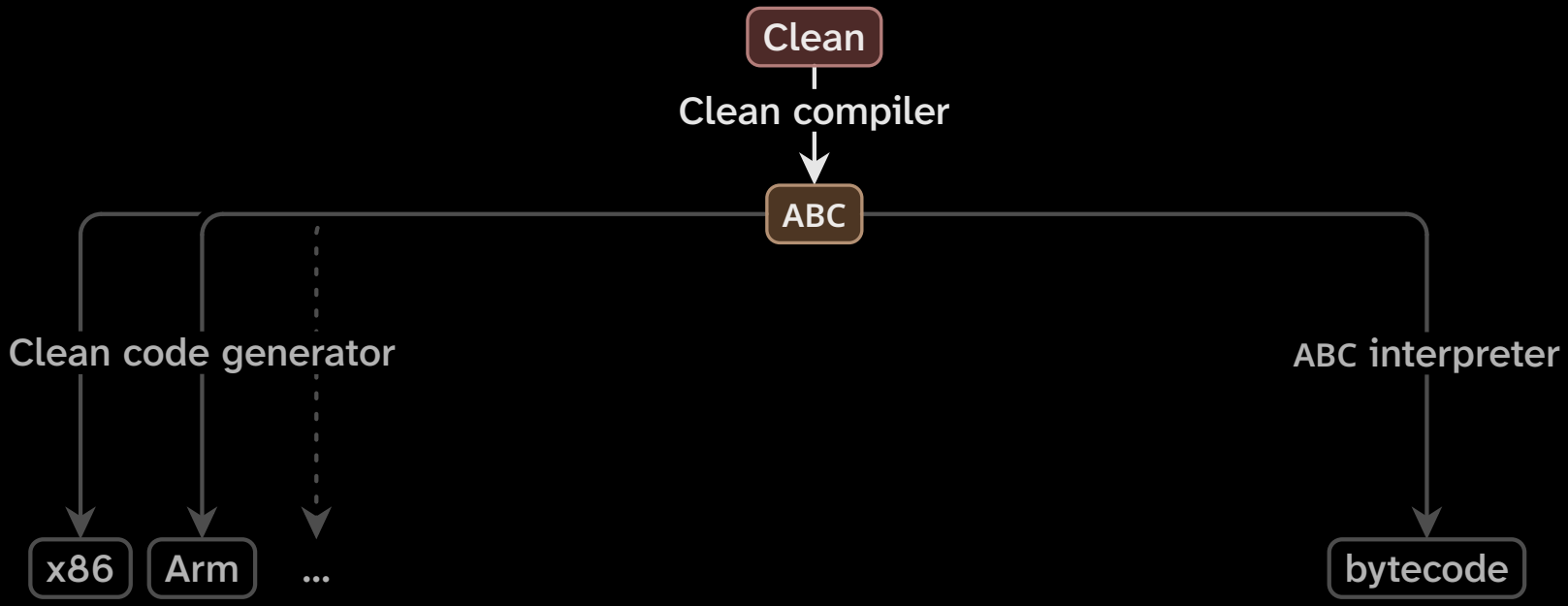
Why?

- Generates Wasm, but also x86, Arm, RISC-V...
- Optimisations

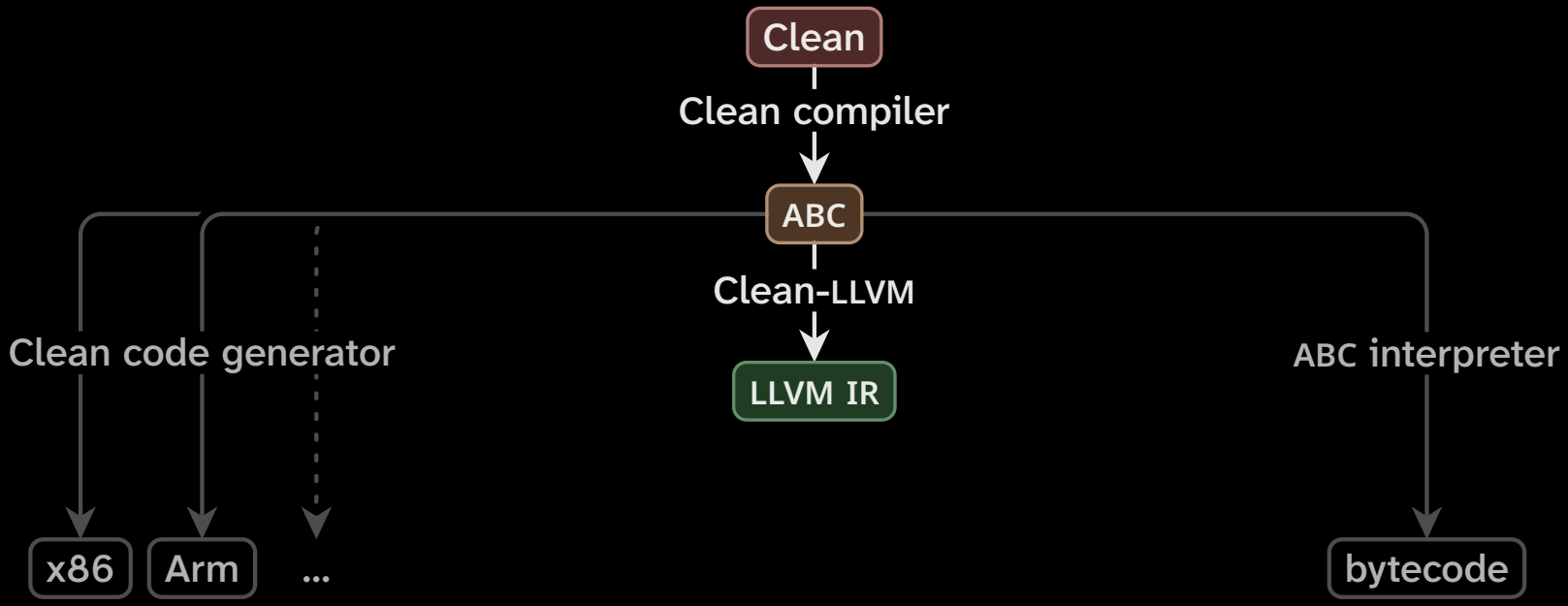
Wasm is not a standard target for LLVM

- Requires some creative solutions

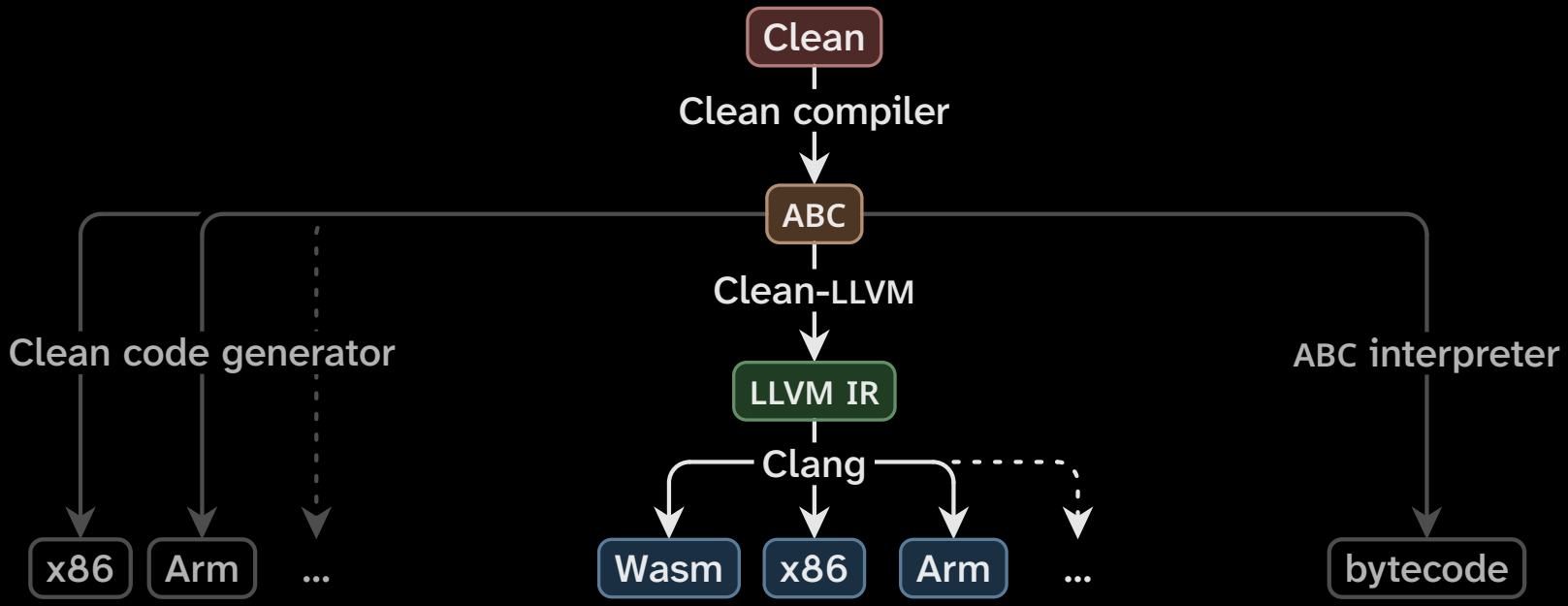
Overview


Clean
Clean compiler
ABC
Clean code generator
ABC interpreter
x86
Arm
...
bytecode

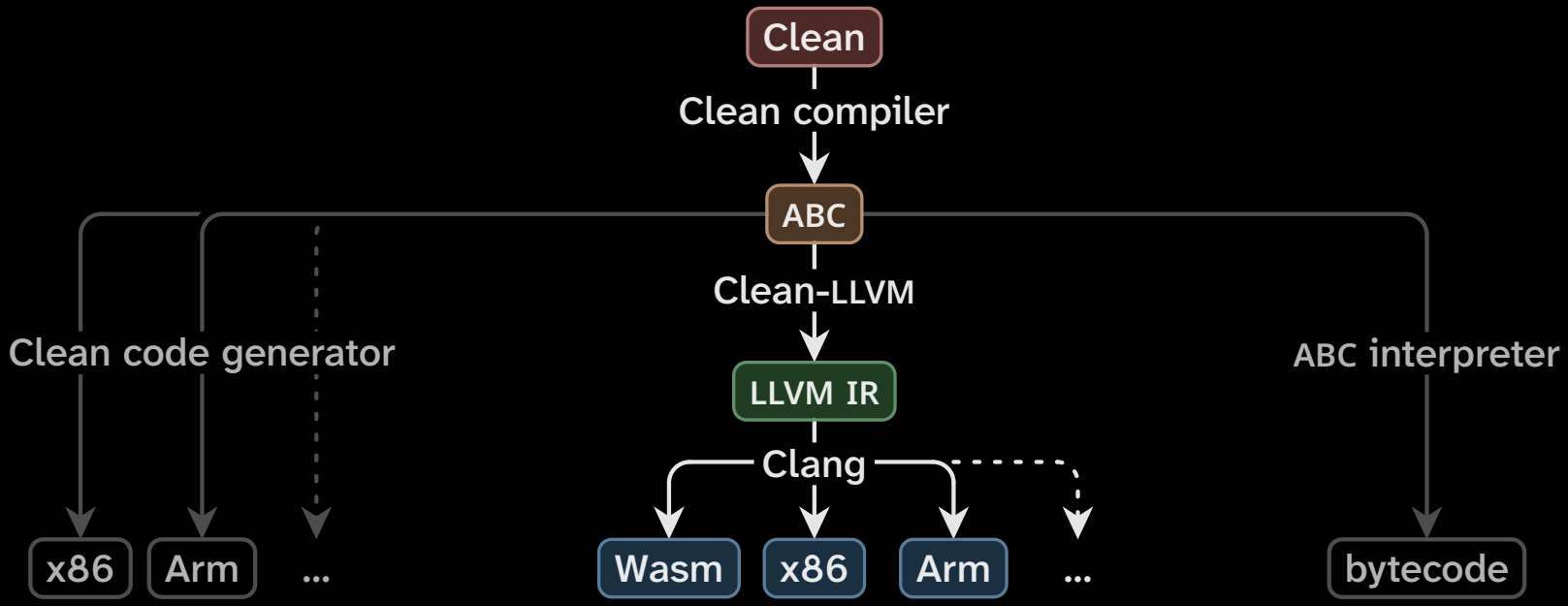
Overview


Clean
Clean compiler
ABC
Clean-LLVM
LLVM IR
Clean code generator
ABC interpreter
x86
Arm
...
bytecode

Overview


Clean
Clean compiler
ABC
Clean-LLVM
LLVM IR
Clang
Clean code generator
ABC interpreter
x86
Arm
...
Wasm
x86
Arm
...
bytecode

Overview



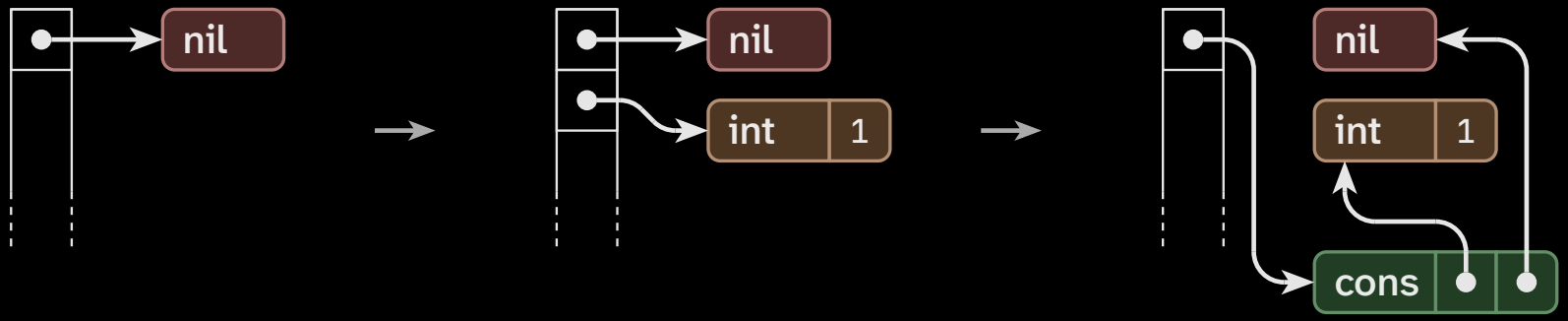
Focus on WebAssembly

GC Integration

Heap and A Stack

A stack: references to the heap (boxed values)
B stack: basic values (unboxed values: int, char, bool, ...)
C stack: control flow

(hence the name of ABC code)



Creation of the list `[1]`.

GC Integration

- A stack is on WebAssembly stack, for performance
- Can only access stack frame of current function

GC Integration

- **A** stack is on WebAssembly stack, for performance
- Can only access stack frame of current function

Shadow stack

1. **Spill** (write **A** stack to memory)
2. **Restore** (read back new addresses)

GC Integration

- **A** stack is on WebAssembly stack, for performance
- Can only access stack frame of current function

Shadow stack

1. **Spill** (write **A** stack to memory)
2. **Restore** (read back new addresses)

LLVM passes

- Run optimisations before generating spills/restores: inlining, dead code elimination
- Leads to less function calls → less spill locations

Shadow Stack Overview

mark → statepoints → `asp` arg → spill

1. Mark A stack (preparation)
2. Insert statepoints
3. Add shadow stack argument
4. Add spills

Preparation: Mark A Stack

- Which LLVM variables are on **A** stack?
- Mark them with special type

Preparation: Mark A Stack

- Which LLVM variables are on **A** stack?
- Mark them with special type

1. Mark heap references (**A** stack values)

```
ptr %a      →      heapptr %a
```

Note: simplified / imaginary syntax

LLVM Pass: Insert Statepoints

- Statepoint: point at which `GC` *can* run
- Only need to update spilled **A** stack across statepoints
  - spill before, restore after

LLVM Pass: Insert Statepoints

- Statepoint: point at which `GC` *can* run
- Only need to update spilled **A** stack across statepoints
  - spill before, restore after

2. Convert function calls to statepoints

```
%answer = call i64 @f(42)
```

LLVM Pass: Insert Statepoints

- Statepoint: point at which `GC` *can* run
- Only need to update spilled **A** stack across statepoints
  - spill before, restore after

2. Convert function calls to statepoints

```
%answer = call i64 @f(42)
```

↓

```
%s = call token @gc.statepoint(@f, 42) [ "gc-live"(heapptr %a) ]
```

LLVM Pass: Insert Statepoints

- Statepoint: point at which GC *can* run
- Only need to update spilled **A** stack across statepoints
  - spill before, restore after

2. Convert function calls to statepoints

```
%answer = call i64 @f(42)
```

↓

```
%s = call token @gc.statepoint(@f, 42) [ "gc-live"(heapptr %a) ]
%answer = call i64 @gc.result(%s)
```

LLVM Pass: Insert Statepoints

- Statepoint: point at which GC *can* run
- Only need to update spilled **A** stack across statepoints
  - spill before, restore after

2. Convert function calls to statepoints

```
%answer = call i64 @f(42)
```

↓

```
%s = call token @gc.statepoint(@f, 42) [ "gc-live"(heapptr %a) ]
%answer = call i64 @gc.result(%s)
%a.new = call i64 @gc.relocate(%s, 0)
```

LLVM Pass: Add Stack Top Argument

3. Add argument for pointer to spilled **A** stack

- `asp` = **A S**tack **P**ointer

```
define i64 @f(i64 %x)
```

↓

```
define { ptr, i64 } @f_asp(ptr %asp, i64 %x)
```

LLVM Pass: Add Spills

4. Transform statepoints to function calls, spills, restores

```
%s = call token @gc.statepoint(@f, 42) [ "gc-live"(heapptr %a) ]
%answer = call i64 @gc.result(%s)
```

LLVM Pass: Add Spills

4. Transform statepoints to function calls, spills, restores

```
%s = call token @gc.statepoint(@f, 42) [ "gc-live"(heapptr %a) ]
%answer = call i64 @gc.result(%s)
```

↓

```
store %a to %asp ; spill
{ %asp.new, %answer } = call i64 @f_asp(%asp, 42)
```

LLVM Pass: Add Spills

4. Transform statepoints to function calls, spills, restores

```
%a.new = call i64 @gc.relocate(%s, 0)
```

↓

```
%a.new = load %asp.new ; restore
```

LLVM Passes: All Together

mark → statepoints → `asp` arg → spill

```
%answer = call i64 @f(42)
```

↓

```
store %a to %asp ; spill
{ %asp.new, %answer } = call i64 @f_asp(%asp, 42)
%a.new = load %asp.new ; restore
```

LLVM Passes: All Together

mark → statepoints → `asp` arg → spill

```
%answer = call i64 @f(42)
```

↓

```
store %a to %asp ; spill
{ %asp.new, %answer } = call i64 @f_asp(%asp, 42)
%a.new = load %asp.new ; restore
```

Why statepoints?

Implementation for native x86 without shadow stack (theoretically)

Results

Benchmarks

Restriction: not all ABC instructions implemented yet

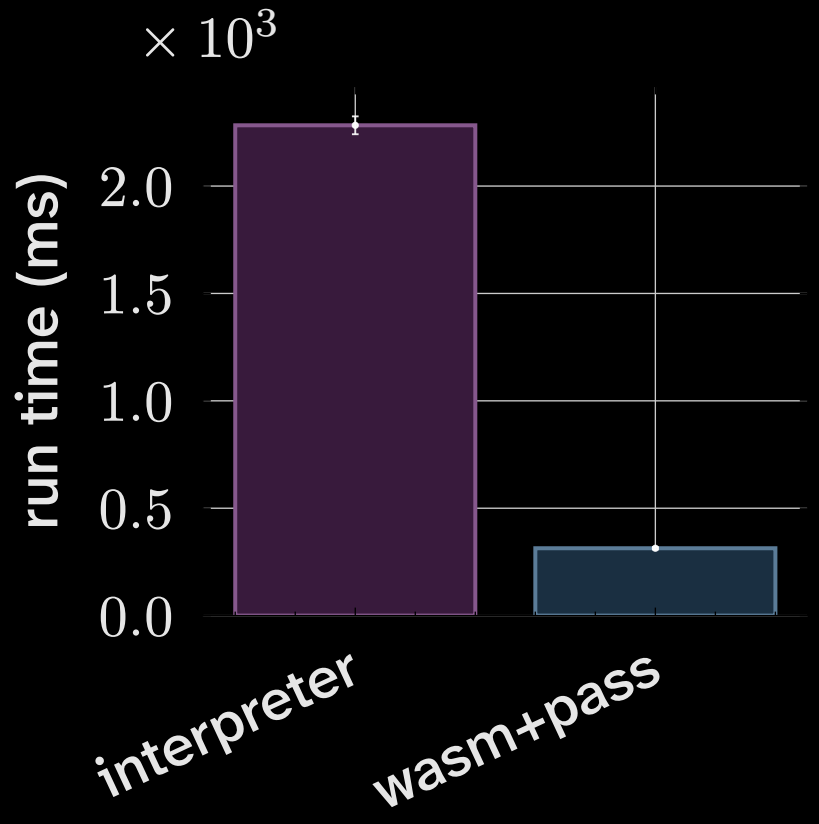
Benchmarks

Restriction: not all ABC instructions implemented yet

- List
  - Create a long list and sum it
- Astack
  - Create 16 heap nodes and pass them to multiple function calls
- **Binarytrees**
  - From the Computer Language Benchmark Game
  - Required additional ABC instructions

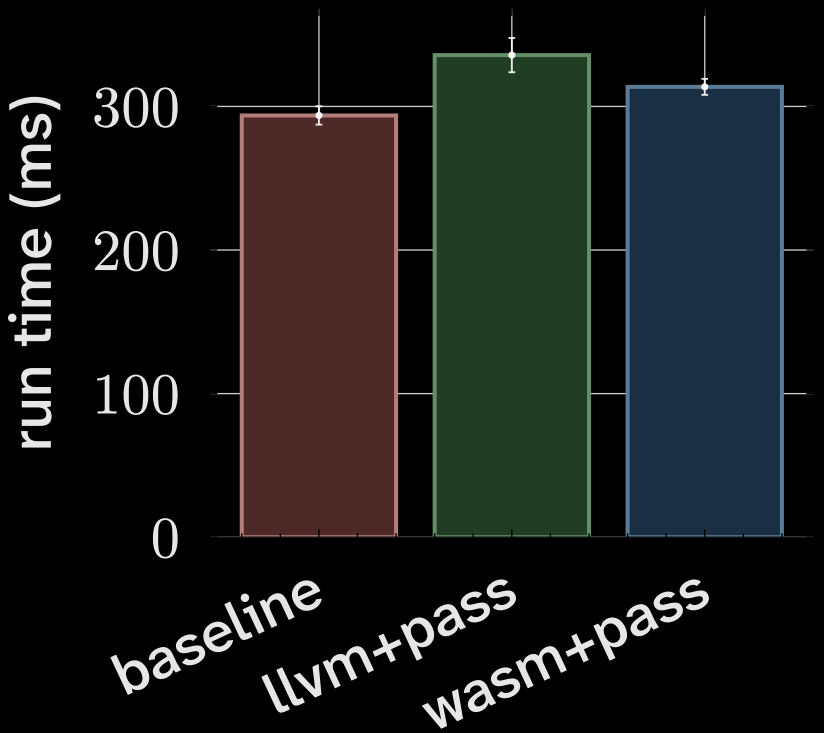
Benchmark: Binarytrees

Wasm: 7 times faster than ABC interpreter (2.3 s vs 0.3 s)



Benchmark: Binarytrees

Close to original Clean compiler's performance



Conclusion

- **Why?** More client-side iTasks code → interpreter too slow

Conclusion

- **Why?** More client-side iTasks code → interpreter too slow
- Clean → ABC → LLVM → WebAssembly

Conclusion

- **Why?** More client-side iTasks code → interpreter too slow
- Clean → ABC → LLVM → WebAssembly
- **GC integration** with shadow **A** stack: spill/restore

Conclusion

- **Why?** More client-side iTasks code → interpreter too slow
- Clean → ABC → LLVM → WebAssembly
- **GC integration** with shadow **A** stack: spill/restore
- **LLVM passes** with statepoints
  1. Mark **A** stack (not a pass, but preparation)
  2. Convert function calls to statepoints
  3. Add shadow **A** stack pointer argument
  4. Convert statepoints to spills/restores

Conclusion

- **Why?** More client-side iTasks code → interpreter too slow
- Clean → ABC → LLVM → WebAssembly
- **GC integration** with shadow **A** stack: spill/restore
- **LLVM passes** with statepoints
  1. Mark **A** stack (not a pass, but preparation)
  2. Convert function calls to statepoints
  3. Add shadow **A** stack pointer argument
  4. Convert statepoints to spills/restores
- **Optimisation:** inlining before generating statepoints

Conclusion

- **Why?** More client-side iTasks code → interpreter too slow
- Clean → ABC → LLVM → WebAssembly
- **GC integration** with shadow **A** stack: spill/restore
- **LLVM passes** with statepoints
  1. Mark **A** stack (not a pass, but preparation)
  2. Convert function calls to statepoints
  3. Add shadow **A** stack pointer argument
  4. Convert statepoints to spills/restores
- **Optimisation:** inlining before generating statepoints
- **Results:** 7 times faster than interpreter, close to original Clean

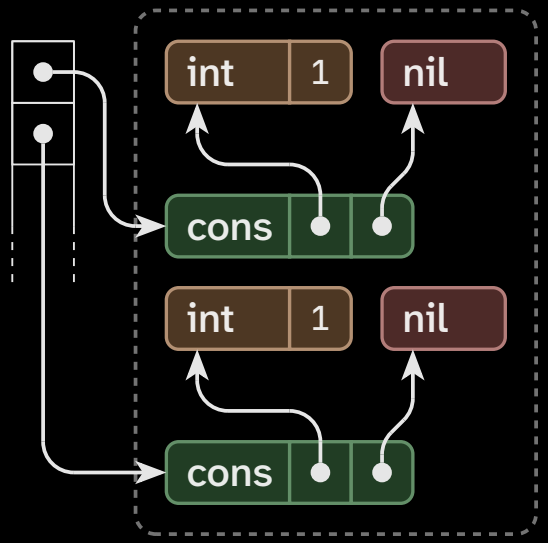
Conclusion

- **Why?** More client-side iTasks code → interpreter too slow
- Clean → ABC → LLVM → WebAssembly
- **GC integration** with shadow **A** stack: spill/restore
- **LLVM passes** with statepoints
  1. Mark **A** stack (not a pass, but preparation)
  2. Convert function calls to statepoints
  3. Add shadow **A** stack pointer argument
  4. Convert statepoints to spills/restores
- **Optimisation:** inlining before generating statepoints
- **Results:** 7 times faster than interpreter, close to original Clean

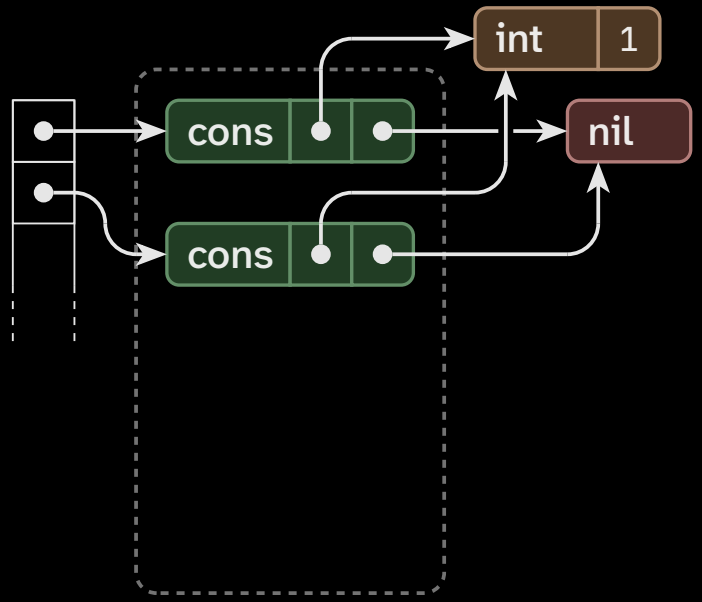
Future work: statepoints without shadow stack for native x86

Extra Slides

Optimisation: Static Nodes

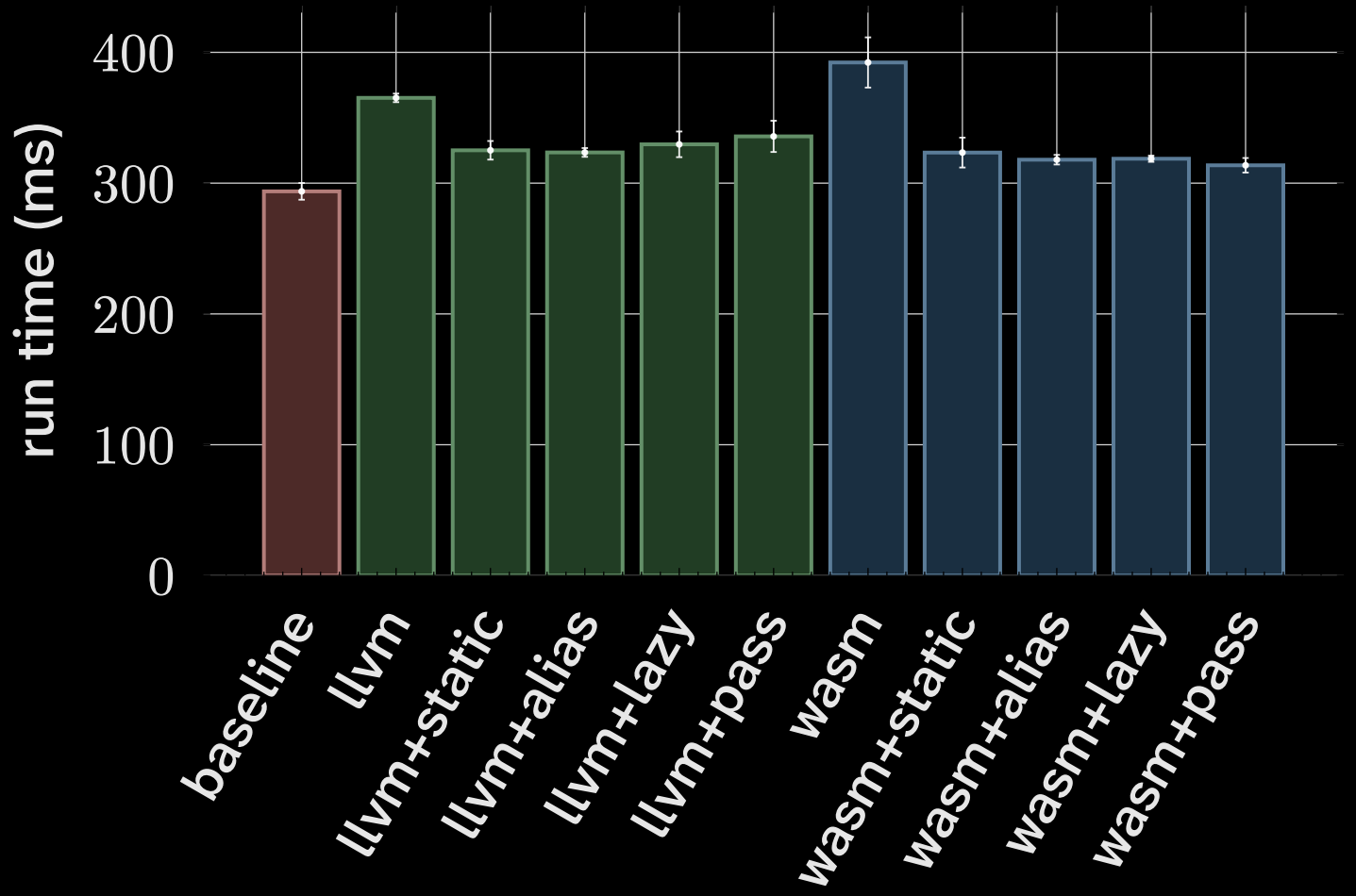


without static nodes

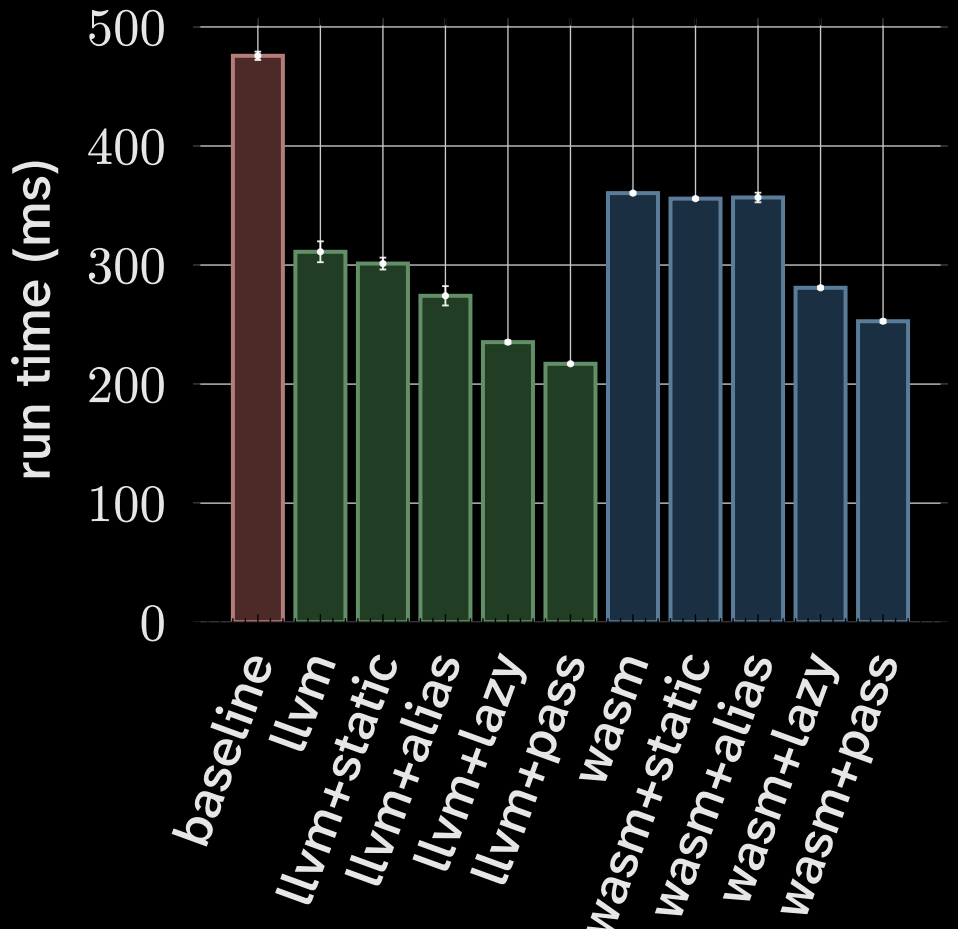


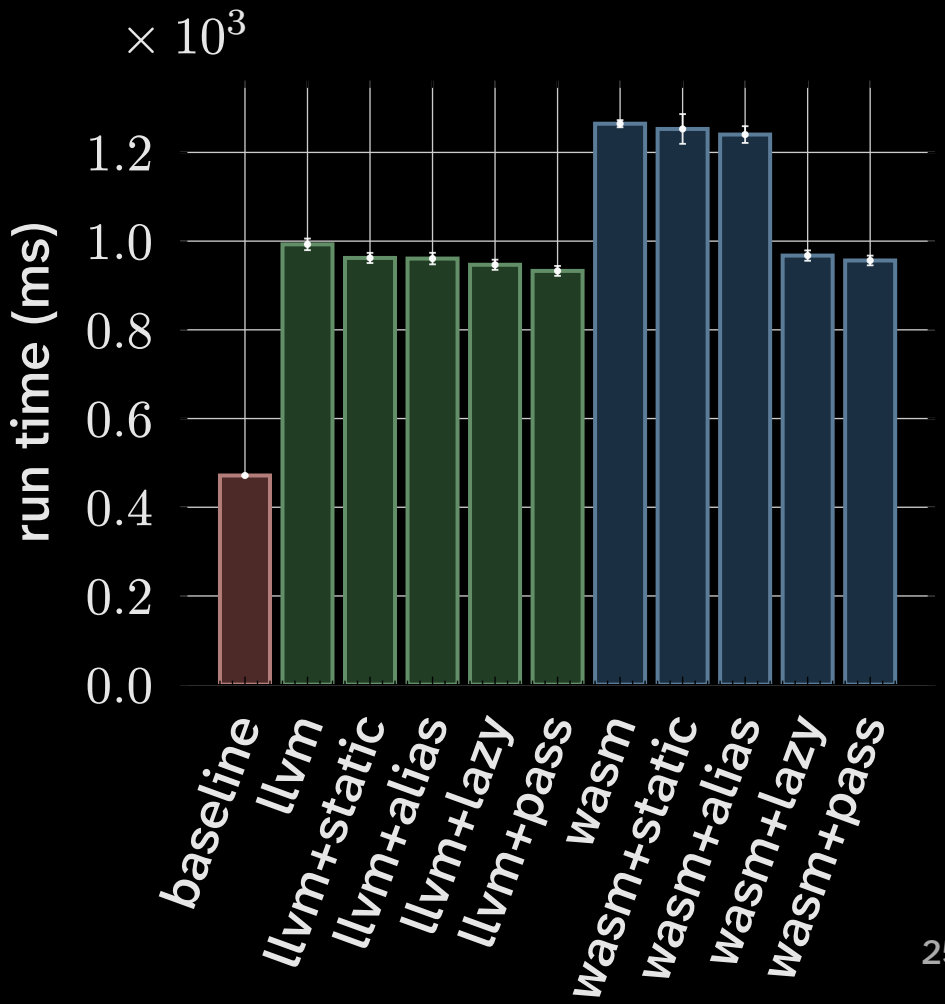
with static nodes

More Benchmarks: Binarytrees

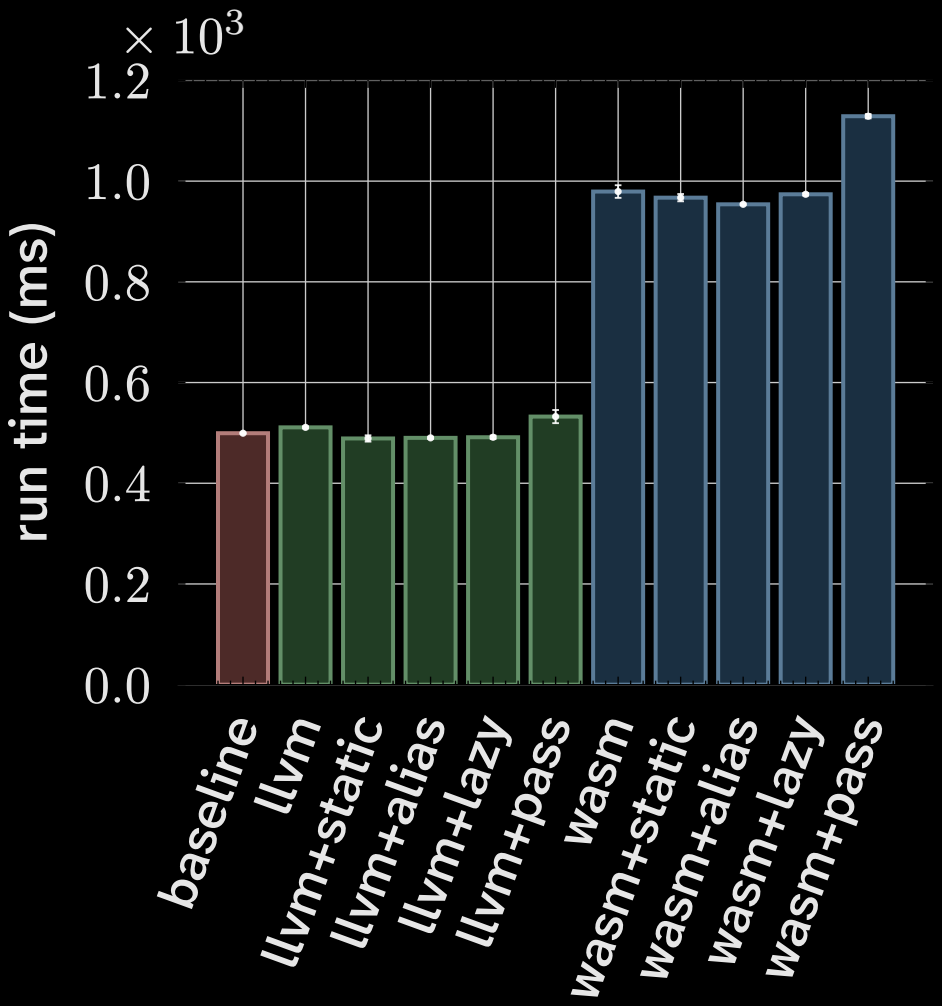


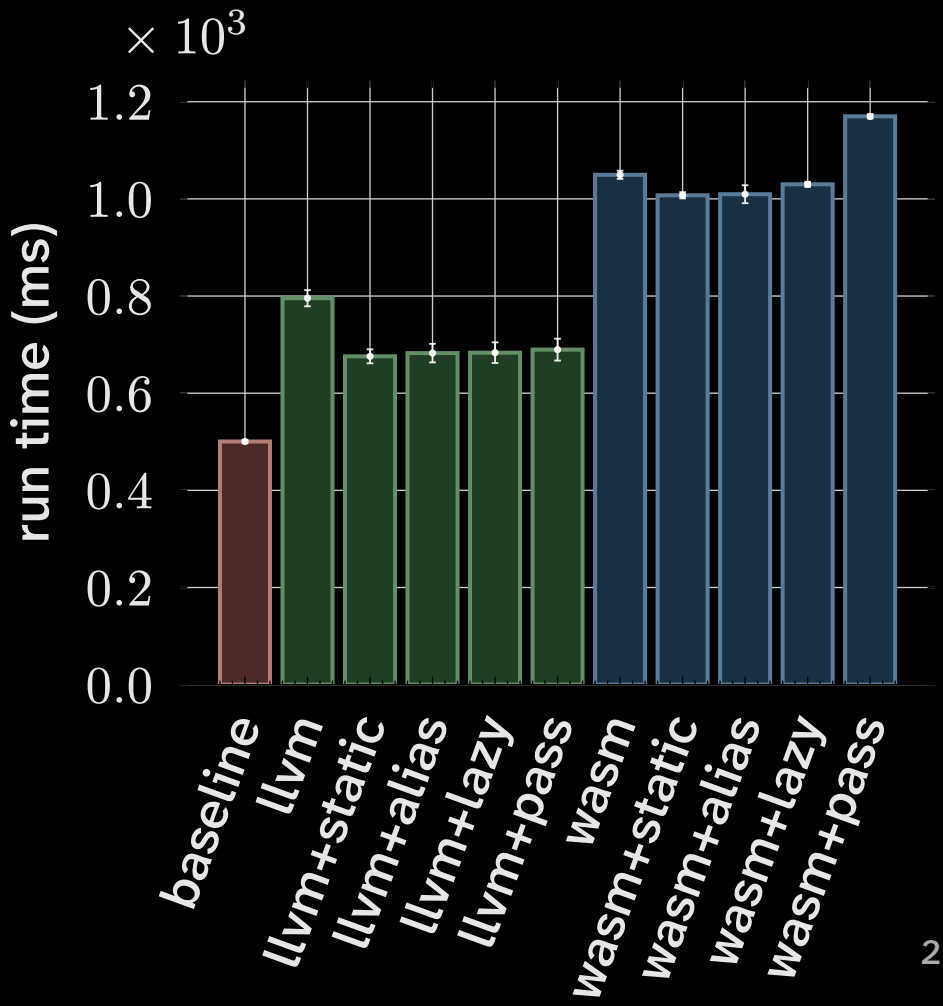
More Benchmarks: Astack





More Benchmarks: List





Questions

Why not Wasm GC?

- Not supported by LLVM (yet)
- Issues with Wasm's type safety when overwriting a thunk with its result