WebAssembly GC subgroup meeting

Ts2wasm: Compiling TypeScript to WasmGC

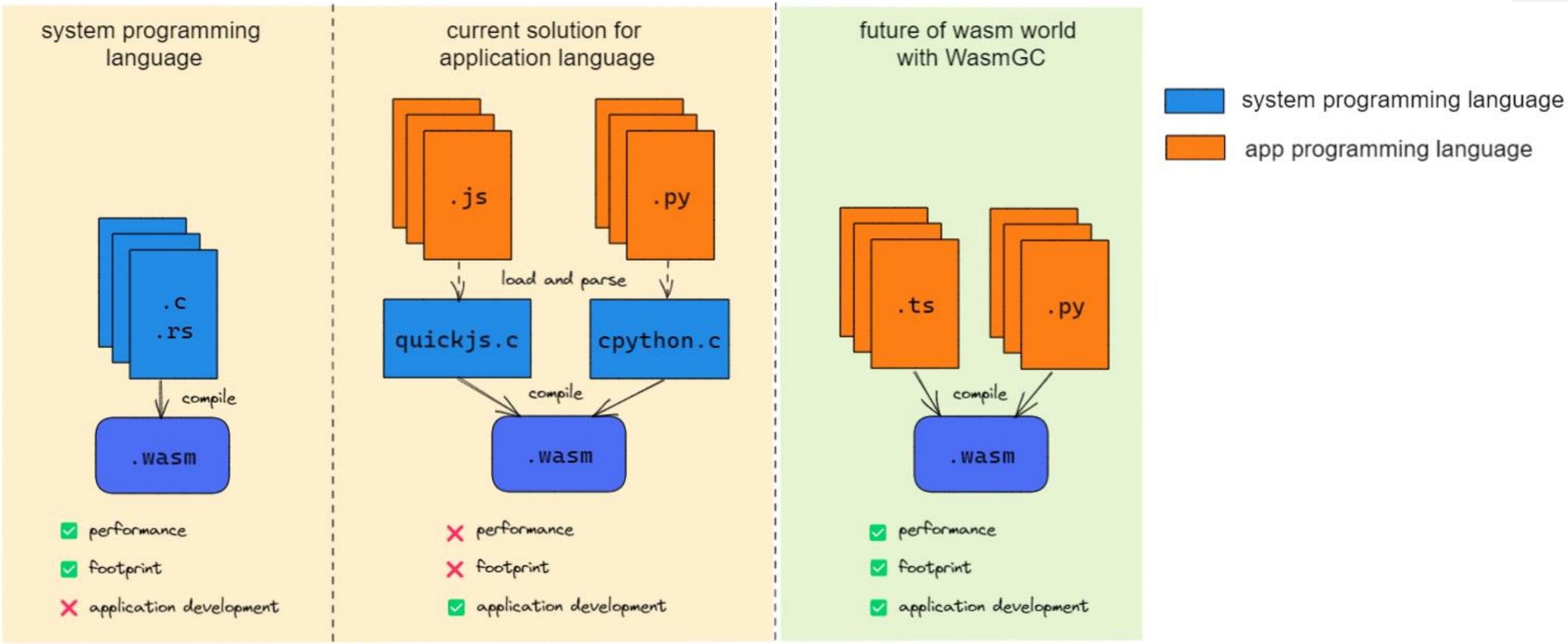
Jun Xu, Intel

Agenda

- Background
- Features work well on WasmGC
  - Primitive
  - Class
  - Function / closure
- Features require extended capabilities
  - Any
  - Interface
- Summary

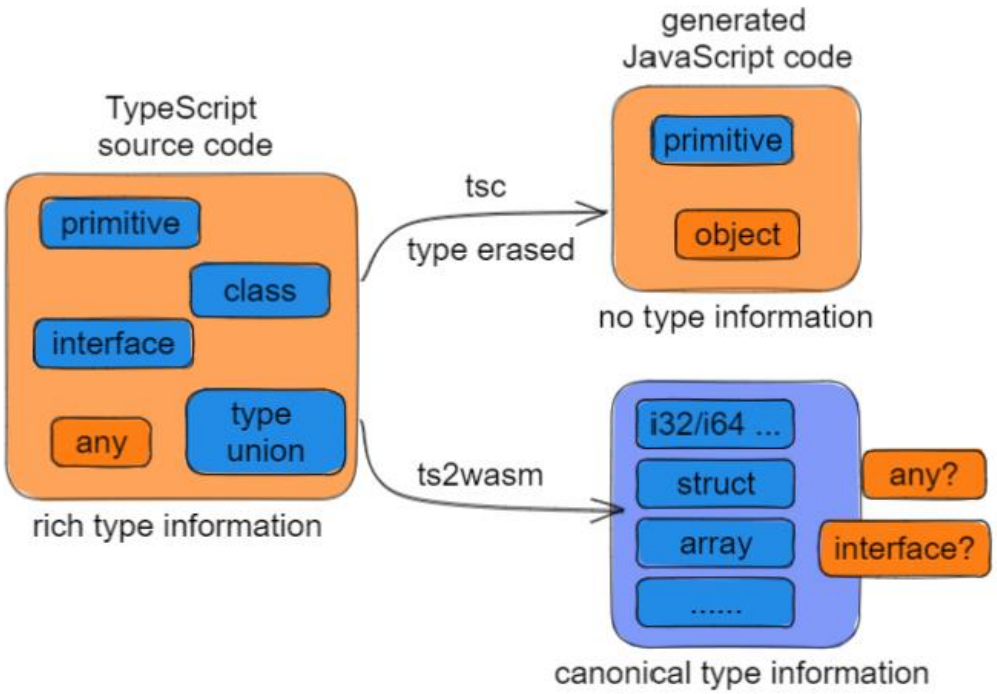
Background

- WebAssembly lacks application programming languages



Background

- TypeScript is a good start because:
  - It can likely run faster on WebAssembly if the type information is well used
  - It is very popular and has rich ecosystem



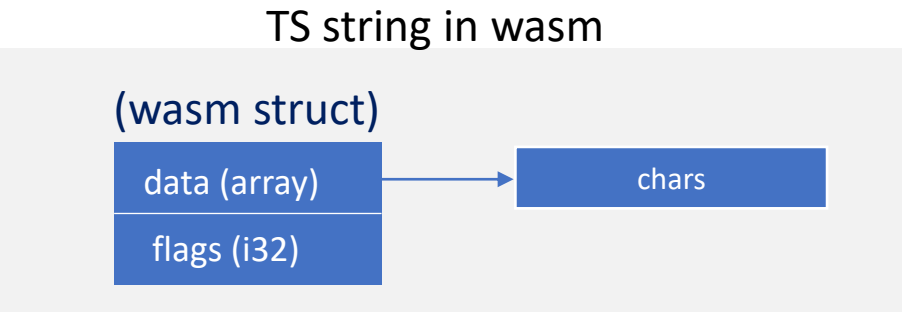
- TypeScript is **gradually typed**, but currently all type information is **lost** during compiling to JavaScript

- Static type information is represented by WasmGC types
- Need special design for **any (dynamic type)** and **interface (duck typing)**

Feature works well on WasmGC :Primitive

- number is represented as f64
- boolean is represented as i32
- string is represented by struct + array (may use stringref in the future)

TypeScript type	Wasm type	Possible optimization
number	f64	
Boolean	i32	treat as i8 for field or element
string	ref (struct (ref array (i8)) (i32))	use stringref proposal



Feature works well on WasmGC :Class

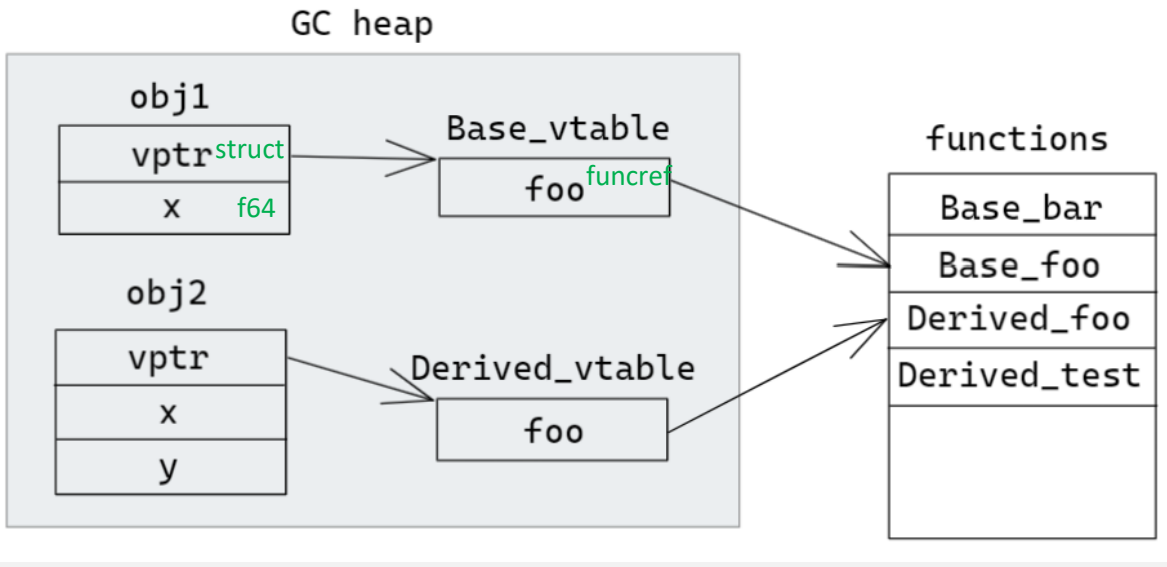
- Instance and vtable are represented as **struct**
- Static fields are in a global struct
- Vtable based inheritance

TS

```
class Base {
    x: number = 0
    foo() {
        return this.x + 1;
    }
    bar(x: number) {}
}
```

TS

```
class Derived extends Base
{
    y: number = 1
    foo() {
        return this.y - 1;
    }
    test() {}
}
```



Explicit sub typing relationship

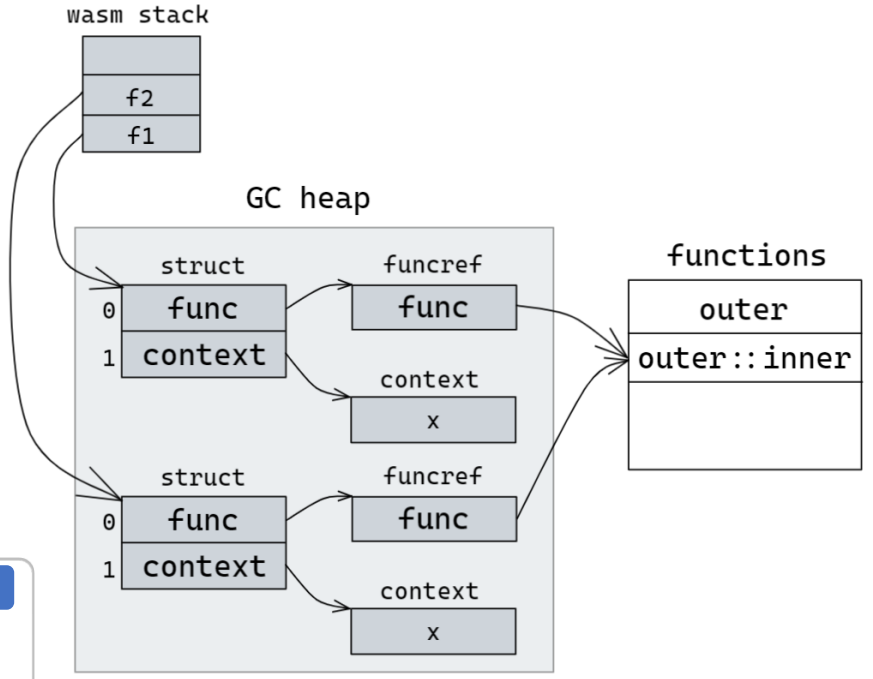
Overwrite method receives base type as “this”
And cast back to concrete type

wasm

```
(type $Base_foo (func ((ref $Base)) (f64)))
(type $Base_bar (func ((ref $Base) ($x f64)) (f64)))
(type $Derived_foo (func ((ref $Base)) (f64)))
(type $Derived_test (func ((ref $Derived))()))

(type $Base-vt (struct (ref $Base_foo)))
(type $Derived-vt (sub $Base-vt (struct (ref
                    $Base_foo))))
(type $Base (struct (ref $Base-vt)($x f64)))
(type $Derived (sub $Base (struct
                    (ref $Derived-vt)
                    ($x f64) ($y f64))))

func Derived_foo($this (ref $Base)) {
    local.get $this
    ref.cast $Derived
    struct.get $Derived 2    ;; Derived.y
    f64.const 1
    f64.sub
}


// call non-virtual function
call $Base_bar($obj1, f64.const 1)

// call virtual function
$Derived_vtable = struct.get ($obj2, 0)
$foo = struct.get ($Derived_vtable, 0)
call_ref $foo($obj2)
```

Feature works well on WasmGC: Function / closure

- Function is represented as wasm function
- First parameter reserved for closure's context
- Closure is a struct with ref to its context and a funcref



TS

```
function outer() {
    let x: number;
    let y: number;

    x = 10;
    y = 10;

    function inner(y: number) {
        return x + y;
    }

    return inner;
}

let f1 = outer();
let f2 = outer();
f1(5);
```

x is closed by inner, y is still a local var

wasm

```
type $outer-context struct ($x f64)
type $closure struct (funcref $inner) (anyref)

func outer($clos-ctx anyref) (ref $closure) {
    $ctx = struct.new $outer-context
    struct.set $outer-context 0 (10)        // x = 10
    local.set $y (10)                       // y = 10
    $clos = struct.new $closure
    struct.set $clos 0 (ref.func $inner)
    struct.set $clos 1 (ref.cast anyref ($ctx))
    return ($clos)
}

func inner($clos-ctx anyref, $y f64) (f64) {
    ref.cast $outer-context ($clos-ctx)
    $x = struct.get $outer-context 0
    return (f64.add ($x, local.get $y))
}
```

Create context because there are closed variable (x)

Closed var go to context

Cast back to context type

wasm

```
// continued
$f1 = call $outer ({})
$f1_func = struct.get $f1 0
$f1_context = struct.get $f1 1
callref $f1_func ($f1_context, f64.const 5)
```

Feature requires extended capabilities: Any

- In TypeScript, “any” is equivalent to a JavaScript object, the actual type is unknown at compile time (dynamic typing)
- It’s not possible to represent any-objects using WasmGC types, we introduced a “libdyntype” to manage the any-objects

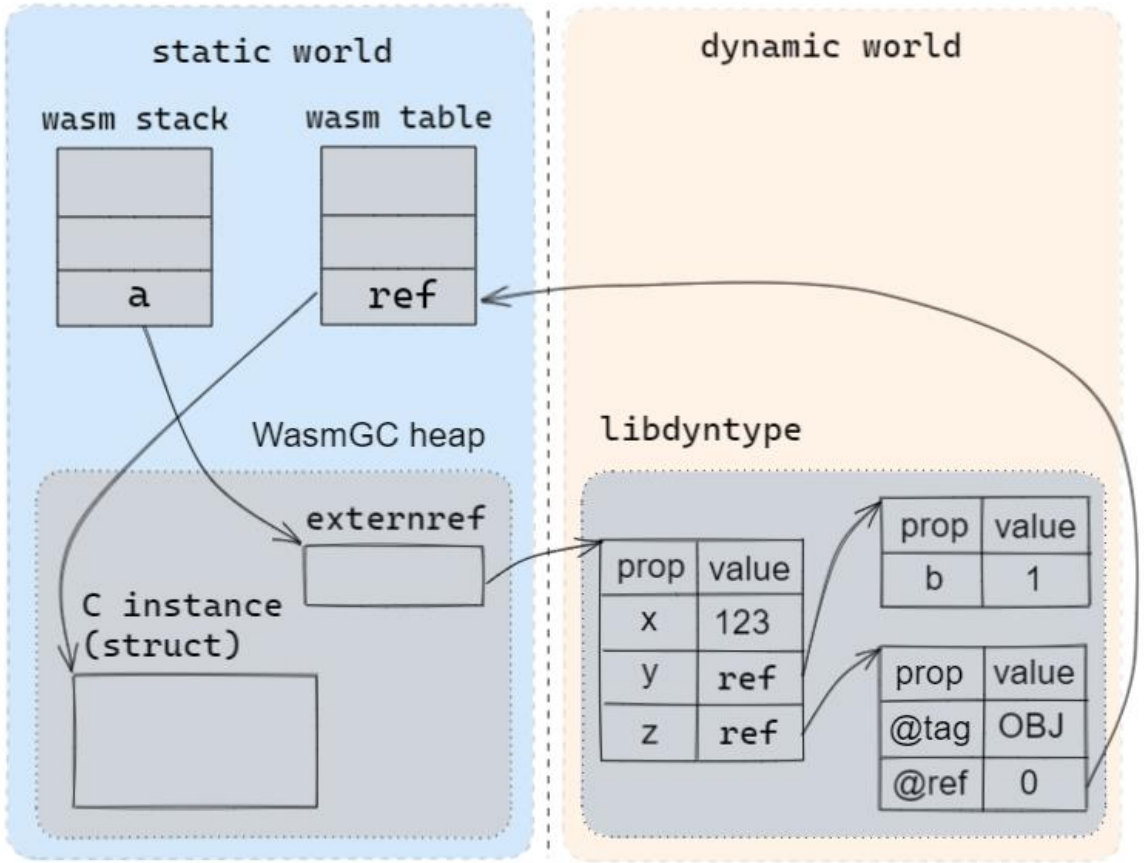
```
class C {…}

let a : any = {
    x: 123,
    y: { b: 1 }
}
```

```
(local.set $a.y
  (call dyntype_new_object)
)
(call dyntype_set_property
  (local.get $a.y)
  (i32.const 1024) ;; ‘b’
  (call dyntype_new_number
    (f64.const 1))
)
```

……

```
a.z = new C();
```

```
(table.set
  (struct.new $C)
  (i32.const 0))
(call dyntype_set_property
  (local.get $a)
  (i32.const 1026) ;; ‘z’
  (call dyntype_new_extref
    (i32.const 0) ;; tbl idx
    (i32.const 0) ;; flag OBJ
  ))
```



Libdyntype can be implemented as:

1. Wasm module/component using linear memory
2. Host APIs

Feature requires extended capabilities: Interface

- In TypeScript, “interface” doesn’t introduce a new type, it’s just a contract to promise least required fields and their types
- Type checking is based on **unordered collections** of field names and types, which means interface doesn’t define a fixed shape for the variable
- We introduced an “interface description table (itable)”, and require an extended opcode to access field through runtime calculated index

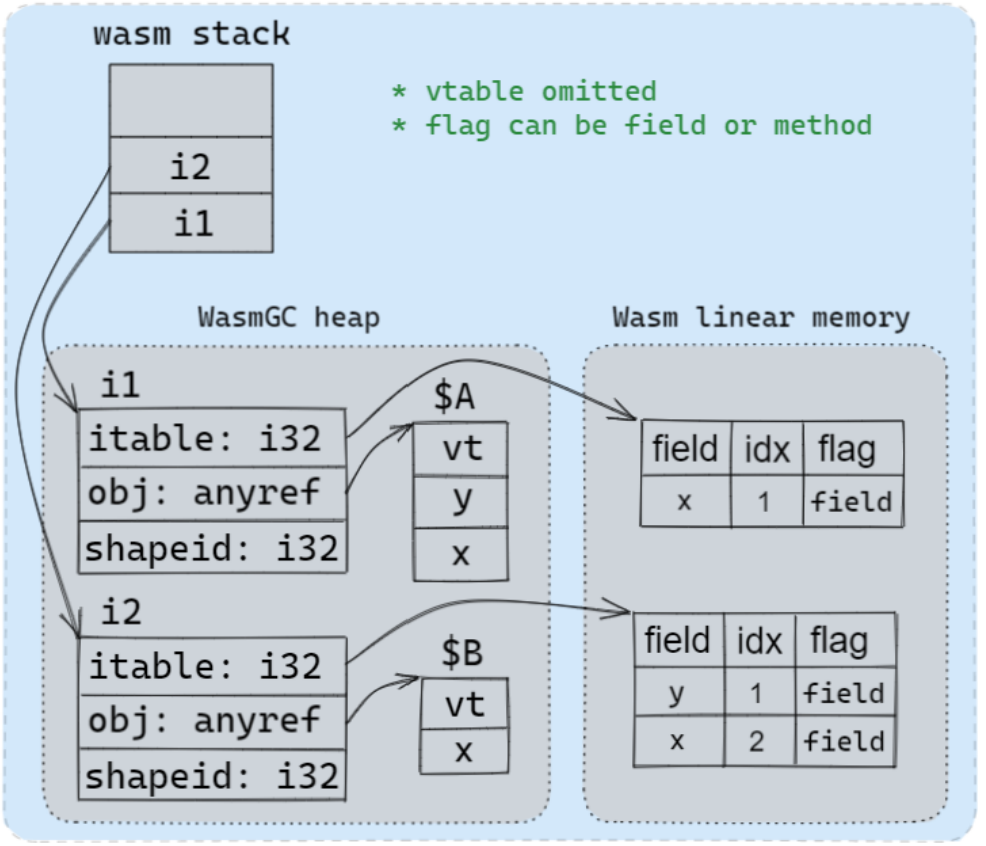
```
interface I {
  x: number
}
class A {
  y: string
  x: number
}
class B {
  x: number
}

let i1 : I = new A()
let i2 : I = new B()

i1.x
i2.x
```

All interface have the same type $ifc

```
(local.set $i1
  (struct.new $ifc
    (i32.const 2048)
    (struct.new $A ……)
    (i32.const 1))) ;; shape id for I and A is 1
```

$i1.xxx = (struct.get <$ifc> <n> (local.get $i1))

```
(if
  (i32.eq
    ($i1.shapeid)
    (i32.const 1))  ;; shape seen by compiler
  (then
    (struct.get <$I> <1>
      (ref.cast <$I> ($i1.obj))))   i2.x
  (else
    (struct.get_indirect <f64>
      ($i1.obj)
      (call find_index
        ($i1.itable)
        (i32.const 1024))))  ;; x   i1.x
)
```



Proposed new opcode: struct.get/set_indirect

Current opcode:

```
struct.get $t i : [(ref null $t)] -> [ti]
```

struct type, **immediate**
field index, **immediate**
ref to struct, **operand**
result

- The type and index is hardcoded
- Only work for specified concrete type

proposed opcode:

```
struct.get_indirect $ti : [anyref, i32] -> [ti]
```

result type, **immediate**
ref to struct, **operand**
field index, **operand**
result

- Runtime should check:
  1. ref is a reference to struct
  2. index is valid for that type
  3. field type is $ti

Similar for:
- struct.get_u/struct.get_s
- struct.set/struct.set_u/struct.set_s

Summary

- WasmGC works well for most of the static types, but need some extensions to allow runtime to enable limited dynamic features
- Our proposed libdyntype API is a better approach to support dynamic languages than compiling their whole language runtime into WebAssembly, because:

    1. Recent languages tends to be gradually typed (e.g. TypeScript, python with type annotation), some parts may be able to be statically compiled

    2. Compiling whole runtime into WebAssembly introduce large footprint, reduce performance and not friendly for FFI and further component model integration

- Our proposed opcode can provide the mechanism to access WasmGC struct fields through index calculated during runtime without influencing original GC design


intel

Libdyntype APIs

Object Creation

```
dyntype_new_number
dyntype_new_boolean
dyntype_new_string
dyntype_new_undefined
dyntype_new_null
dyntype_new_object
dyntype_new_array
dyntype_new_extref
```

Field access

```
dyntype_set_property
dyntype_define_property
dyntype_get_property
dyntype_has_property
dyntype_delete_property
```

Prototype

```
dyntype_new_object_with_proto
dyntype_set_prototype
dyntype_get_prototype
dyntype_get_own_property
dyntype_instanceof
```

Runtime type check

```
dyntype_is_undefined
dyntype_is_null
dyntype_is_bool
dyntype_to_bool
dyntype_is_number
dyntype_to_number
dyntype_is_string
dyntype_to_cstring
dyntype_free_cstring
dyntype_is_object
dyntype_is_array
dyntype_is_extref
dyntype_to_extref
dyntype_typeof
dyntype_type_eq
```

Semantic of the proposed opcode

name	immediates	stack signature
`struct.get_indirect_<sx>?`	`<ti>`	`[anyref, i32] → [ti]`
`struct.set_indirect_<sx>?`	`<ti>`	`[anyref, i32, ti] → []`

Checking rule:

- Trap if ref is null
- Trap if ref is not a struct
- Trap if index is invalid in the struct
- Trap if type of the accessing field is not ti