

# Wasm GC proposal:

## Spec for Prototype Implementation v.6

Authors: [jkummerow@chromium.org](mailto:jkummerow@chromium.org), [manoskouk@chromium.org](mailto:manoskouk@chromium.org), [tlively@google.com](mailto:tlively@google.com)

Link to this doc: <https://bit.ly/3cWcm6Q>

**Status: final**

**This is version 6** of this document. Version 5 is archived [here](#).

### Major changes since version 5:

- Nominal types have been removed.
- Rtts and rtt-consuming instructions have been removed.
- The `let` instruction has been removed. It is replaced by non-nullable locals, which must be written before they can be read (and then count as initialized until the end of the current control flow block).
- The abstract type hierarchy has been refactored: We decided to go with a 3-pronged type hierarchy, with func, extern, and any as the top types of each one. "dataref" has been replaced with "structref".
- Refactored type check/cast operations.

### Preview of version 7:

The cast instruction refactoring has left behind a number of deprecated instructions, which will be removed; see the table below for details. (A draft document does not exist yet, and implementation work has not yet started.)

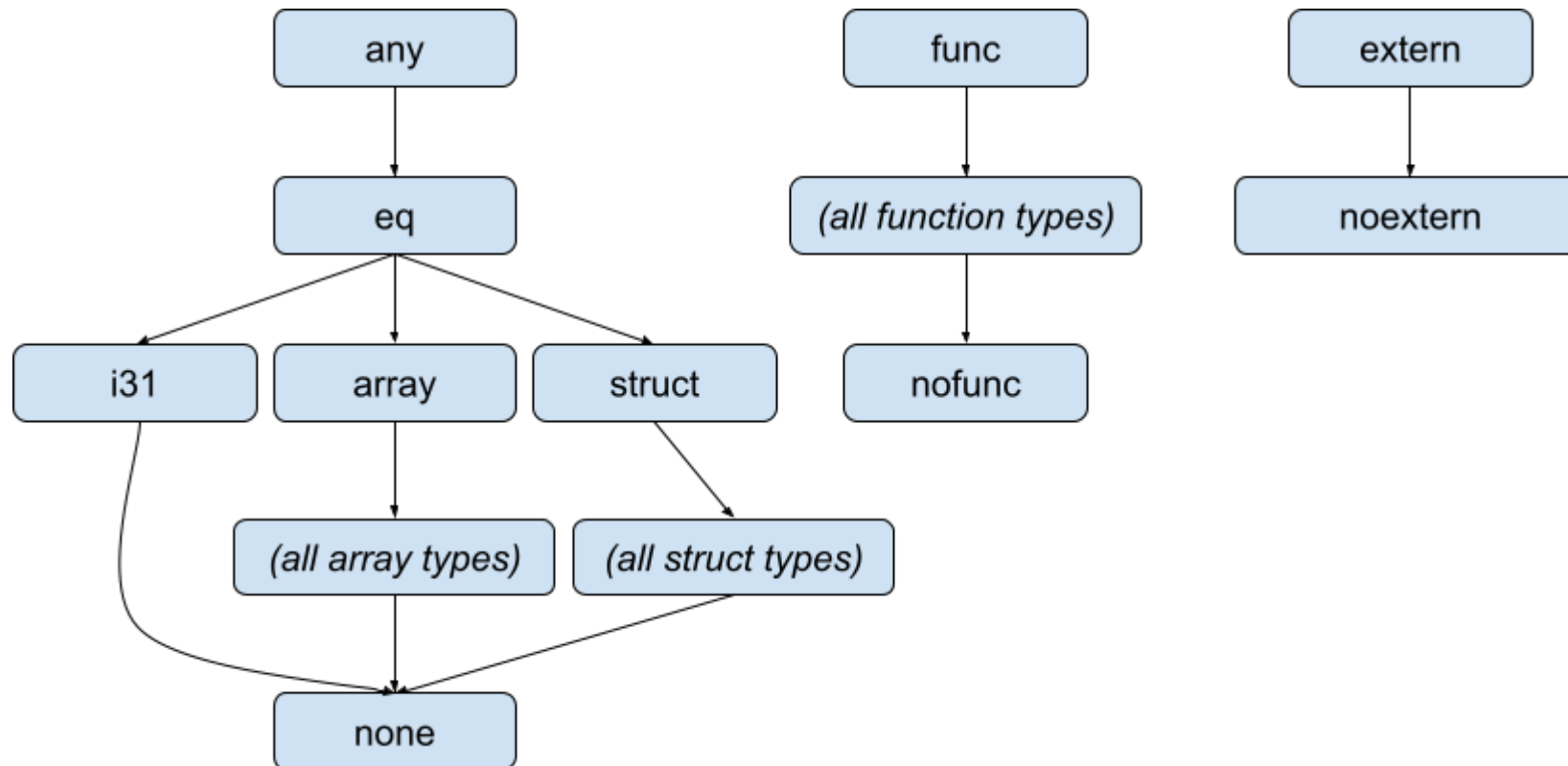
## Overview

See "major changes" above.

# Types

## Abstract type hierarchy

It has been decided that there will be three distinct type hierarchies: external references, functions, and “internal” or data types. All hierarchies include a bottom type.



Note that `any` includes references introduced by the host, which belong to none of `any`'s subtypes.

## Encoding

The type encoding does not change other than dropping rtt's with depth (see rationale below).

References: [gc proposal](#), [f-r proposal](#)

[Color codes: white = old, green = milestone 5, yellow = updated in milestone 6]

name	+code	-code	immediates	feature	Notes
i32	0x7f	-0x1			
i64	0x7e	-0x2			
f32	0x7d	-0x3			
f64	0x7c	-0x4			
v128	0x7b	-0x5		simd	
i8	0x7a	-0x6		gc (*)	packed struct/array fields only
i16	0x79	-0x7		gc (*)	packed struct/array fields only
funcref	0x70	-0x10		ref	Shorthand for (ref null func)
externref	0x6f	-0x11		ref	Shorthand for (ref null extern); for some time, this was known as anyref
anyref	0x6e	-0x12		gc	
eqref	0x6d	-0x13		gc	Shorthand for (ref null eq)

ref null \$t	0x6c	-0x14	<heaptypes> : <a href="#">s33</a>	f-r	
ref \$t	0x6b	-0x15	<heaptypes> : <a href="#">s33</a>	f-r	
i31ref	0x6a	-0x16		gc	Shorthand for (ref null i31)
<del>rtt \$n \$t</del>	<del>0x69</del>	<del>-0x17</del>	<del>u32 &lt;<a href="#">typeid</a>&gt;</del>	<del>ge</del>	<del>Deprecated alias for (rtt \$t) (depth gets ignored), for backwards compatibility only; to be dropped in M6</del>
<del>rtt \$t</del>	<del>0x68</del>	<del>-0x18</del>	<del>&lt;<a href="#">typeid</a>&gt;</del>	<del>ge</del>	<del>to be dropped in M6</del>
nullexternref	0x69	-0x17		gc	Shorthand for (ref null noextern)
nullfuncref	0x68	-0x18		gc	Shorthand for (ref null nofunc)
dataref	<del>0x67</del>	<del>-0x19</del>		<del>ge</del>	<del>Shorthand for (ref null data)</del>
structref	0x67	-0x19		gc	Shorthand for (ref null struct)
arrayref	0x66	-0x1a		gc	Shorthand for (ref null array)
nullref	0x65	-0x1b		gc	Shorthand for (ref null none)

## Type definitions

Find below the formal grammar for the binary format of an isorecursive type section. An isorecursive type section consists of recursive type groups of type which can reference each other (and themselves). Each type in a group is an optional subtype definition followed by a base type definition of a struct, array, or function.

## Isorecursive module

```
typeSection    ::= 0x01 vec(recGroupDef)
recGroupDef    ::= 0x4f vec(subtypeDef)           ; A rec. group with a specified #elements
                | subtypeDef                     ; A type outside a rec. group
subtypeDef     ::= 0x50 vec(u32) baseTypeDef      ; A base type with a number of explicitly specified supertypes
                                                        (note: restricted to 1 for now)
                | baseTypeDef                   ; A base type without supertypes, i.e. 0x50 0 <baseTypeDef>
baseTypeDef    ::= 0x60 funcTypeDef
                | 0x5f structTypeDef
                | 0x5e arrayTypeDef
funcTypeDef    ::= vec(type) vec(type) ; A function type with parameter and return types
structTypeDef  ::= vec(fieldDef)       ; a struct with fields
arrayTypeDef   ::= fieldDef            ; an array with an element type and mutability
fieldDef       ::= storageType [0|1]   ; a storage type (value type including i8 and i16) and a mutability
```

Note: Types are encoded as specified by the previous section.

## Nominal module

*Nominal modules are dropped in "Milestone 6".*

## Type Canonicalization

Isorecursive type groups are canonicalized (across all Wasm modules instantiated in an engine at the same time) as long as the entire group has identical structure (i.e. same types, same subtyping relationships between them).

(TODO: Does this need to be fleshed out more?)

# Instructions

[Color codes: white = old, green = milestone 5, yellow = new or updated in milestone 6]

## Unprefixed

name	code	immediates	stack signature	feat.
call_ref	0x14	<typeid> <sup>3</sup>	[t1* (ref null \$t)] -> [t2*]	f-r
return_call_ref	0x15	<typeid>	[t1* (ref null \$t)] -> [t2*]	f-r
let <sup>4</sup>	0x17	<blocktype> <localdefs>	[t* t1*] -> [t2*]	f-r
call_ref <sup>5</sup>	0x17	<typeid> <sup>5</sup>	[t1* (ref null \$t)] -> [t2*]	x <sup>5</sup>
...	...	...	...	...
ref.null	0xd0	<heaptyp>	[] -> [ref null \$t]	ref
ref.is_null	0xd1		[ref null \$t] -> [i32]	ref
ref.func <sup>2</sup>	0xd2	<funcidx>	[] -> [(ref \$t)]	ref
ref.as_non_null	0xd3		[(ref null \$t)] -> [(ref \$t)]	f-r
br_on_null	0xd4	<labelidx>	[t* (ref null \$t)] -> [t* (ref \$t)]	f-r
ref.eq	0xd5		[eqref eqref] -> [i32]	gc
br_on_non_null	0xd6	<labelidx>	[t* (ref null \$t)] -> [t*]	f-r

<sup>1</sup> As a replacement for the 'let' instruction, locals [may now be non-nullable, and retain initialized-ness until the end of the current block](#).

<sup>2</sup> Returned [funcref] per “ref” proposal, refined to [ref \$t] per “f-r” proposal.

<sup>3</sup> The call\_ref instruction (0x14) is [getting a type immediate](#). Since that's a backwards-incompatible change, we're doing a multi-step dance across V8 and Binaryen to provide an incremental transition: we temporarily produce/accept 0x17+immediate, will then change the 0x14 encoding to require this immediate, and will finally drop the 0x17 encoding again (with several weeks between each step).

Progress:

- as of [r82839](#) (Aug 31), V8 accepts 0x17 + type immediate.
- as of PR [5079](#) (Sep 23), Binaryen emits 0x17 + immediate.
- as of [r83900](#) (Oct 25), V8 requires a type immediate for 0x14 (and considers 0x17 deprecated).
- as of PR [5246](#) (Nov 15), Binaryen emits 0x14 + immediate
- as of [r85223](#) (Jan 11), V8 no longer accepts 0x17

name	encoding
<a href="#">funcidx</a>	<a href="#">u32</a>
<a href="#">heaptypes</a>	<a href="#">s32</a>
<a href="#">labelidx</a>	<a href="#">u32</a>
segmentidx	u32
blocktype	0x40   <value_type>   \$t: <a href="#">u32</a> , if \$t: func_type
localdefs	<a href="#">vec</a> ( <a href="#">u32</a> <value_type> )

## New prefix (0xfb)

The "C" column indicates whether instructions are considered "constant instructions", i.e., are usable outside the code section. A '?' there means: might make sense to be supported, raise your metaphorical hand if you'd like to have it.

[Color codes: white = old, green = milestone 5, yellow = new or updated in milestone 6, pink = slated to change in milestone 7]

name	code	immediates	stack signature	notes	C
struct.new_with_rtt	0xfb01	<typeidx>	<del>[t'* (rtt \$t)] -&gt; [(ref \$t)]</del>	<del>will disappear</del>	✓
struct.new_default_with_rtt	0xfb02	<typeidx>	<del>[(rtt \$t)] -&gt; [(ref \$t)]</del>	<del>will disappear</del>	✓
struct.new	0xfb07	t : <typeidx>	[t'*] -> [(ref \$t)]		✓
struct.new_default	0xfb08	t : <typeidx>	[] -> [(ref \$t)]		✓
struct.get	0xfb03	<typeidx> <fieldidx>	[(ref null \$t)] -> [t]		
struct.get_s	0xfb04	<typeidx> <fieldidx>	[(ref null \$t)] -> [t]		
struct.get_u	0xfb05	<typeidx> <fieldidx>	[(ref null \$t)] -> [t]		
struct.set	0xfb06	<typeidx> <fieldidx>	[(ref null \$t) ti] -> []		
array.new_with_rtt	0xfb14	<typeidx>	<del>[t' i32 (rtt \$t)] -&gt; [(ref \$t)]</del>	length argument interpreted as unsigned (u32), <del>will disappear</del>	
array.new_default_with_rtt	0xfb12	<typeidx>	<del>[i32 (rtt \$t)] -&gt; [(ref \$t)]</del>	length unsigned, <del>will disappear</del>	
array.new	0xfb1b	t : <typeidx>	[t' i32] -> [(ref \$t)]		✓
array.new_default	0xfb1c	t : <typeidx>	[i32] -> [(ref \$t)]		✓



array.get	0xfb13	<typeid>	[(ref null \$t) i32] -> [t]	index unsigned	
array.get_s	0xfb14	<typeid>	[(ref null \$t) i32] -> [t]	index unsigned	
array.get_u	0xfb15	<typeid>	[(ref null \$t) i32] -> [t]	index unsigned	
array.set	0xfb16	<typeid>	[(ref null \$t) i32 t] -> []	index unsigned	
array.len	0xfb17	<typeid>	[arrayref] -> [i32]	please use 0xfb19	
array.len	0xfb19		[arrayref] -> [i32]		
array.copy*	0xfb18	<typeid1> <typeid2>	[(ref null typeid1) i32 (ref null typeid2) i32 i32] -> []	indices unsigned	
array.init	0xfb19	<typeid> <u32>	[t <sup>n</sup> (rtt \$t)] -> [(ref \$t)]	\$t = (array t mutable) will disappear	✓
array.new_fixed*	0xfb1a	<typeid> <u32>	[t <sup>n</sup> ] -> [(ref \$t)]	Former array.init_static	✓
array.init_from_data	0xfb1e	<typeid> <segmentidx>	[i32 i32 (rtt t)] -> [(ref \$t)]	\$t = (array t mutable?) t numeric, <u>will disappear</u>	✓
array.new_data*	0xfb1d	<typeid> <segmentidx>	[i32 i32] -> [(ref \$t)]	\$t = (array t mutable?) t numeric type. Former array.init_from_data_static	?
array.new_elem*	0xfb1f	<typeid> <segmentidx>	[i32 i32] -> [(ref \$t)]	\$t = (array t mutable?) t reference type. Former array.init_from_elem_static	?
i31.new	0xfb20		[i32] -> [i31ref]		✓
i31.get_s	0xfb21		[i31ref] -> [i32]	Note: i31ref was previously (ref i31)	

				but is now (ref null i31)	
i31.get_u	0xfb22		[i31ref] -> [i32]	Note: i31ref was previously (ref i31) but is now (ref null i31)	
rtt.canon	0xfb30	<typeid>	[] -> [(rtt \$t)]	will disappear	✓
rtt.sub	0xfb34	<typeid>	[(rtt n? t1)] -> [(rtt (n+1)? t2)]		✓
rtt.fresh_sub	0xfb32	<typeid>	[(rtt n? t1)] -> [(rtt (n+1)? t2)]		?
ref.test	0xfb40		[(ref null t1) (rtt t2)] -> [i32]	<a href="#">will disappear</a>	
ref.test	0xfb40	<heaptypes>	[(ref null ht)] -> [i32]	returns 0 for null	
ref.test null	0xfb48	<heaptypes>	[(ref null ht)] -> [i32]	returns 1 for null	
ref.test	0xfb44	<typeid t2>	[(ref null t1)] -> [i32]	<a href="#">deprecated</a> , use 0xfb40 instead Returns 0 for null	
ref.cast	0xfb44		[(ref null? t1) (rtt t2)] -> [(ref null? t2)]	<a href="#">will disappear</a>	
ref.cast	0xfb41	<heaptypes t2>	[(ref null? t1)] -> [(ref t2)]	traps on null	
ref.cast null	0xfb49	<heaptypes t2>	[(ref null t1)] -> [(ref null t2)]	null is passed through	
ref.cast	0xfb45	<typeid t2>	[(ref null? t1)] -> [(ref null? t2)]	<a href="#">deprecated</a> , use 0xfb41/0xfb49 instead null is passed through	
ref.cast_nop	0xfb4c	<typeid t2>	[(ref null? t1)] -> [(ref null? t2)]	unsafe, temporary, only for experimenting, in V8 needs -experimental-ref-cast-nop	

br_on_cast	0xfb42	<labelidx>	[(ref null? t1) (rtt t2)] → ¬[(ref null? t1)]	<a href="#">will disappear</a>	
br_on_cast	0xfb42	<labelidx l> <heaptypes t2>	[(ref null? t1)] → [(ref null? t1)]	does not branch on null; branch at l must be >: [(ref t2)]	
br_on_cast null	0xfb4a	<labelidx l> <heaptypes t2>	[(ref null? t1)] → [(ref t1)]	branches on null; branch at l must be >: [(ref null? t2)]	
br_on_cast	0xfb46	<labelidx> <typeid t2>	[(ref null? t1)] → [(ref null? t1)]	<a href="#">deprecated</a> , use 0xfb42 instead	
br_on_cast_fail	0xfb43	<labelidx>	[(ref null? t1) (rtt t2)] → ¬[(ref t2)]	<a href="#">will disappear</a>	
br_on_cast_fail	0xfb43	<labelidx l> <heaptypes t2>	[(ref null? t1)] → [(ref t2)]	branches on null; branch at l must be >: [(ref null? t1)]	
br_on_cast_fail null	0xfb4b	<labelidx l> <heaptypes t2>	[(ref null? t1)] → [(ref null? t2)]	does not branch on null; branch at l must be >: [(ref t1)]	
br_on_cast_fail	0xfb47	<labelidx> <typeid t2>	[(ref null? t1)] → [(ref t2)]	<a href="#">deprecated</a> , use 0xfb43 instead	
ref.is_fun	0xfb50		[anyref] → [i32]	<a href="#">will disappear</a>	
ref.is_data	0xfb51		[anyref] → [i32]	<a href="#">will disappear</a> , use 0xfb40 instead	
ref.is_i31	0xfb52		[anyref] → [i32]	<a href="#">will disappear</a> , use 0xfb40 instead	
ref.is_array	0xfb53		[anyref] → [i32]	<a href="#">will disappear</a> , use 0xfb40 instead	
ref.as_fun	0xfb58		[anyref] → [(ref fun)]	<a href="#">will disappear</a>	

ref.as_data	0xfb59		[anyref] -> [dataref]	<a href="#">will disappear</a> , use 0xfb41 instead	
ref.as_i31	0xfb5a		[anyref] -> [i31ref]	<a href="#">will disappear</a> , use 0xfb41 instead	
ref.as_array	0xfb5b		[anyref] -> [arrayref]	<a href="#">will disappear</a> , use 0xfb41 instead	
br_on_func	0xfb60	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* t] if t <: anyref	Branch at <depth> must be >: t* (ref func); <a href="#">will disappear</a>	
br_on_data	0xfb61	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* t] if t <: anyref	<a href="#">will disappear</a> , use 0xfb42 instead	
br_on_i31	0xfb62	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* t] if t <: anyref	<a href="#">will disappear</a> , use 0xfb42 instead	
br_on_array	0xfb66	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* t] if t <: anyref	<a href="#">will disappear</a> , use 0xfb42 instead	
br_on_non_func	0xfb63	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* (ref func)] if t <: anyref	Branch at <depth> must be >: t* t; <a href="#">will disappear</a>	
br_on_non_data	0xfb64	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* dataref] if t <: anyref	<a href="#">will disappear</a> , use 0xfb43 instead	
br_on_non_i31	0xfb65	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* i31ref] if t <: anyref	<a href="#">will disappear</a> , use 0xfb43 instead	
br_on_non_array	0xfb67	depth:< <a href="#">labelidx</a> >	[t* t] -> [t* arrayref] if t <: anyref	<a href="#">will disappear</a> , use 0xfb43 instead	
extern.internalize	0xfb70		[(ref null? extern)] -> [(ref null? any)]		✓
extern.externalize	0xfb71		[(ref null? any)] -> [(ref null? extern)]		✓

#### Encoding:

- <[typeid](#)> : [u32](#)

- <heaptyp> : [s33](#)
- <fieldidx> : [u32](#)
- <labelidx> : [u32](#)

#### **array.copy:**

- two immediate arguments: type index of destination array, type index of source array.
- stack arguments (in order): destination array, destination index, source array, source index, length.
- Semantics: copy 'length' elements of the source array, starting at source index, into the destination array, starting at destination index.
- Traps if either array is null
- Traps if either of the ranges falls outside the bounds of the respective array.
- If the two arrays coincide and the ranges overlap, the copy will happen as if the elements are first copied into an intermediate array.
- Note: It is not guaranteed that this instruction will be in the MVP.

#### **array.new\_fixed:**

- two immediate arguments: index of the array type, array length
- Semantics: returns an array initialized from a statically known number of arguments. Reads from the stack n arguments compatible with the array type.

#### **array.new\_data**

- Takes an immediate array type index \$typeid = (array \$t mutable?) and an immediate segment index \$segmentidx, and an \$offset and \$length from the stack. \$t has to be a numeric type.
- If \$segmentidx has length at least \$offset + \$length \* sizeof(\$t), the instruction returns an array of type \$typeid and \$length, initialized with the contents of segment \$segmentidx starting at \$offset. Otherwise, it traps. Bytes in memory are interpreted as little-endian full-length values (not LEB-128) of type \$t.

#### **array.new\_elem**

- Takes an immediate array type index \$typeid = (array \$t mutable?) and an immediate segment index \$segmentidx, and an \$offset and \$length from the stack. \$t has to be a reference type. The type of \$segmentidx has to be a subtype of \$t.
- If \$segmentidx has length at least \$offset + \$length, the instruction returns an array of type \$typeid and \$length, initialized with the contents of segment \$segmentidx starting at \$offset. Otherwise, it traps.
- This instruction does not take dropping of segments into account, i.e., all passive segments count as non-dropped.
- Active segments count as empty.

# Tables

Tables can now be declared with all reference types, including non-nullable ones. Since non-nullable types do not have a default value, such tables need an initializer. Therefore the following table form is introduced:

```
table ::= 0x40 tabletype constexpr
```

## JavaScript interaction

NEW: there is now a draft for JS interop, for details see [this doc](#). In short, you can pass Wasm structs/arrays to JavaScript, where they'll appear as frozen empty objects. You can pass them back to Wasm in one of two ways:

1. Typed as ``externref``, which makes the crossing of the boundary free, but to get the Wasm types back, you have to execute ``extern.internalize`` and ``ref.cast`` explicitly.
2. Typed as a more specific type, in which case the equivalents of ``extern.internalize`` and ``ref.cast`` will be performed implicitly.

Aside from function parameters/results, objects can also be passed via (exported) tables and globals.

V8 already implements this, and no longer uses wrapper objects internally.

## V8/Binaryen Implementation Backlog

### V8

- ☐ [back compat] The spec does not include `array.copy`

### Binaryen

- ☐ Does not implement `arrayref` or the `array heap` type
- ☐ Does not implement `array.init_from_data` or `array.init_from_data_static`

