Wasm GC proposal:
Spec for Prototype Implementation v.6

Authors: jkummerow@chromium.org, manoskouk@chromium.org, 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.
It's currently (May 2023) looking like no breaking changes before the final opcode reshuffling will be needed, so v7 of this document will ~~probably~~ equal the final version of the proposal (🎉), including an **all-new binary encoding**.
(A draft document does not exist yet, and implementation work has not yet started.)
Update: the final binary encoding (per this PR) has been added to the tables below. No further behavior changes are planned.

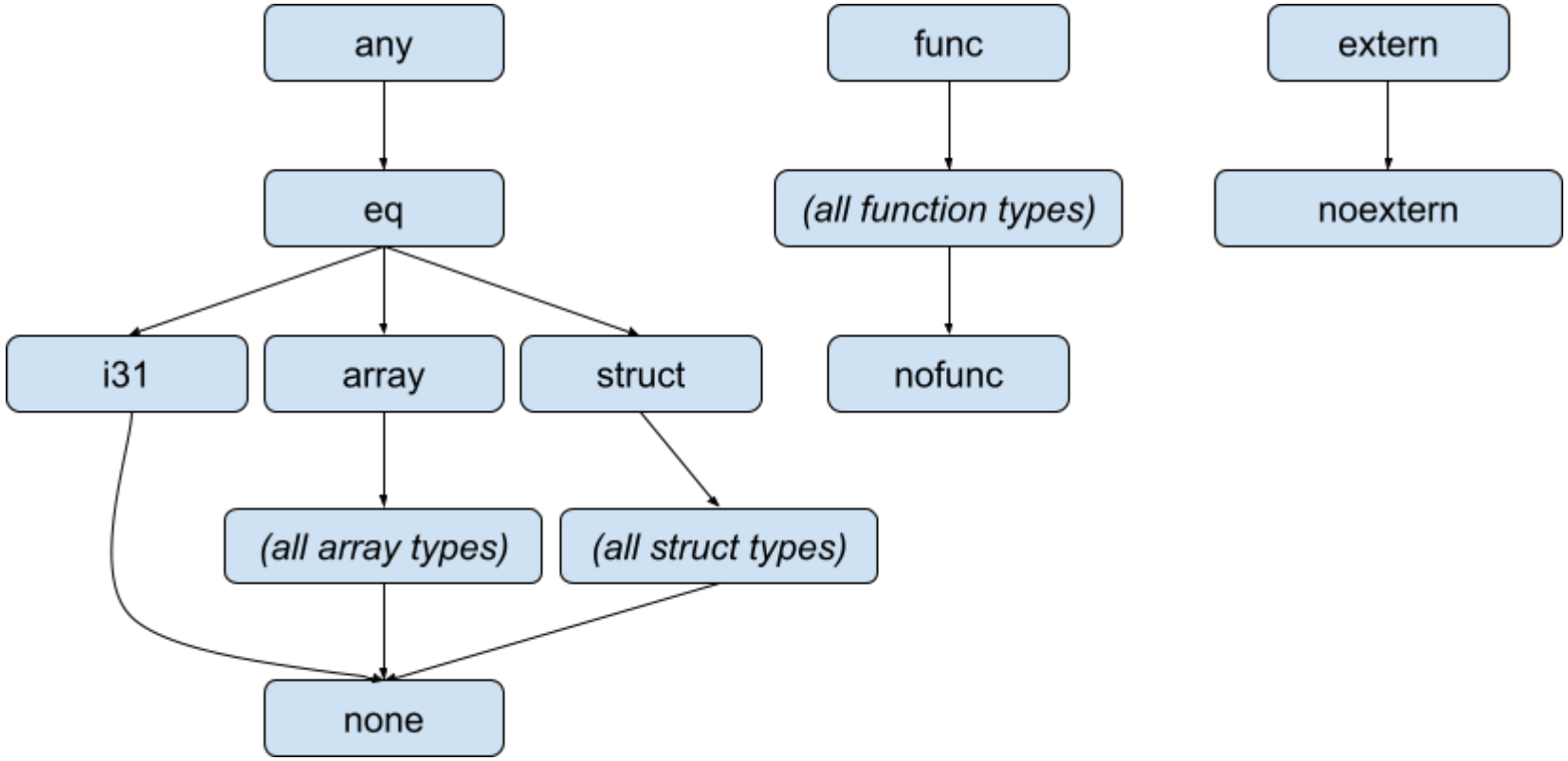
Overview

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 rtts 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	final +code	final -code	immediates	feature	Notes
i32	0x7f	-0x1					
i64	0x7e	-0x2					
f32	0x7d	-0x3					
f64	0x7c	-0x4					
v128	0x7b	-0x5				simd	
i8	0x7a	-0x6	**0x78**	**-0x8**		gc	packed struct/array fields only
i16	0x79	-0x7	**0x77**	**-0x9**		gc	packed struct/array fields only
nullfuncref	0x68	-0x18	**0x73**	**-0xd**		gc	Shorthand for (ref null nofunc)
nullexternref	0x69	-0x17	**0x72**	**-0xe**		gc	Shorthand for (ref null noextern)
nullref	0x65	-0x1b	**0x71**	**-0xf**		gc	Shorthand for (ref null none)
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	**0x6e**	**-0x12**		gc	
eqref	0x6d	-0x13	**0x6d**	**-0x13**		gc	Shorthand for (ref null eq)
i31ref	0x6a	-0x16	**0x6c**	**-0x14**		gc	Shorthand for (ref null i31)
structref	0x67	-0x19	**0x6b**	**-0x15**		gc	Shorthand for (ref null struct)

arrayref	0x66	-0x1a	**0x6a**	**-0x16**		gc	Shorthand for (ref null array)
ref $t	0x6b	-0x15	**0x64**	**-0x1c**	<heaptype> : s33	f-r	
ref null $t	0x6c	-0x14	**0x63**	**-0x1d**	<heaptype> : s33	f-r	
~~rtt $n $t~~	~~0x69~~	~~-0x17~~	**-**	**-**	~~u32 <typeidx>~~	~~gc~~	~~Deprecated alias for (rtt $t) (depth gets ignored), for backwards compatibility only; to be dropped in M6~~
~~rtt $t~~	~~0x68~~	~~-0x18~~	**-**	**-**	~~<typeidx>~~	~~gc~~	~~to be dropped in M6~~
~~dataref~~	~~0x67~~	~~-0x19~~	**-**	**-**		~~gc~~	~~Shorthand for (ref null data)~~

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
                   final encoding: 0x4e
                 | 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)
                 | 0x4e vec(u32) baseTypeDef     ; A final base type (same as 0x50…, but can't have further subtypes)
                   final encoding: 0x4f
                 | baseTypeDef                   ; A final base type without supertypes, i.e. 0x4e 0 <baseTypeDef>
```

```
baseTypeDef   ::= 0x60 funcTypeDef                ; final type!
                | 0x5f structTypeDef              ; final type!
                | 0x5e arrayTypeDef               ; final type!
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.
Note: In V8, enforcement of final types currently requires the `--wasm-final-types` flag. We'll enable that by default when module producers have caught up and are emitting final types correctly. In the meantime (without the flag), all types are treated as subtypable for compatibility. Use the flag manually to verify that your module is compatible with the final (hehe) spec.

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

[1] As a replacement for the 'let' instruction, locals may now be non-nullable, and retain initialized-ness until the end of the current block.
[2] Returned [funcref] per “ref” proposal, refined to [ref $t] per “f-r” proposal.

[3] 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**
funcidx	u32
heaptype	s33
labelidx	u32
segmentidx	u32
blocktype	0x40 | <value_type> | $t: u32, if $t: func_type
localdefs	vec(u32 <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~~ final version]

name	code	final code	immediates	stack signature	notes	C
struct.new	0xfb07	**0xfb00**	t : <typeidx>	[t'*] -> [(ref $t)]		✔
struct.new_default	0xfb08	**0xfb01**	t : <typeidx>	[] -> [(ref $t)]		✔
struct.get	0xfb03	**0xfb02**	<typeidx> <fieldidx>	[(ref null $t)] -> [t]		
struct.get_s	0xfb04	**0xfb03**	<typeidx> <fieldidx>	[(ref null $t)] -> [t]		
struct.get_u	0xfb05	**0xfb04**	<typeidx> <fieldidx>	[(ref null $t)] -> [t]		
struct.set	0xfb06	**0xfb05**	<typeidx> <fieldidx>	[(ref null $t) ti] -> []		
array.new	0xfb1b	**0xfb06**	t : <typeidx>	[t' i32] -> [(ref $t)]		✔
array.new_default	0xfb1c	**0xfb07**	t : <typeidx>	[i32] -> [(ref $t)]		✔
array.new_fixed*	0xfb1a	**0xfb08**	<typeidx> <u32>	[t^n] -> [(ref $t)]	Former array.init_static	✔
array.new_data*	0xfb1d	**0xfb09**	<typeidx> <segmentidx>	[i32 i32] -> [(ref $t)]	$t = (array t mutable?) t numeric type. Former array.init_from_data_static	?
array.new_elem*	0xfb1f	**0xfb0a**	<typeidx> <segmentidx>	[i32 i32] -> [(ref $t)]	$t = (array t mutable?) t reference type. Former array.init_from_elem_static	?
array.get	0xfb13	**0xfb0b**	<typeidx>	[(ref null $t) i32] -> [t]	index unsigned	

array.get_s	0xfb14	**0xfb0c**	<typeidx>	[(ref null $t) i32] -> [t]	index unsigned	
array.get_u	0xfb15	**0xfb0d**	<typeidx>	[(ref null $t) i32] -> [t]	index unsigned	
array.set	0xfb16	**0xfb0e**	<typeidx>	[(ref null $t) i32 t] -> []	index unsigned	
array.len	0xfb19	**0xfb0f**		[arrayref] -> [i32]		
array.fill*	0xfb0f	**0xfb10**	<typeidx>	[(ref null typeidx) i32 $t i32] -> []	typeidx = (array $t' mutable), t <: t'	
array.copy*	0xfb18	**0xfb11**	<typeidx1> <typeidx2>	[(ref null typeidx1) i32 (ref null typeidx2) i32 i32] -> []	indices unsigned	
array.init_data*	0xfb54	**0xfb12**	<typeidx> <segmentidx>	[(ref null typeidx) i32 i32 i32] -> []	typeidx = (array $t mutable)	
array.init_elem*	0xfb55	**0xfb13**	<typeidx> <segmentidx>	[(ref null typeidx) i32 i32 i32] -> []	typeidx = (array $t mutable)	
ref.test	0xfb40	**0xfb14**	<heaptype>	[(ref null ht)] -> [i32]	returns 0 for null	
ref.test null	0xfb48	**0xfb15**	<heaptype>	[(ref null ht)] -> [i32]	returns 1 for null	
ref.cast	0xfb41	**0xfb16**	<heaptype t2>	[(ref null? t1)] -> [(ref t2)]	traps on null	
ref.cast null	0xfb49	**0xfb17**	<heaptype t2>	[(ref null t1)] -> [(ref null t2)]	null is passed through	
br_on_cast	0xfb4e	**0xfb18**	<flags> <labelidx l> <heaptype t1> <heaptype t2>	[(ref null1 t1)] -> [(ref null1 t1)]	Flags: bit 0 = null1, bit 1 = null2. Branches on null if null2 = 1. Branch at l >: [(ref null2 t2)]	
br_on_cast_fail	0xfb4f	**0xfb19**	<flags>	[(ref null1 t1)] -> [(ref null2 t2)]	Flags: bit 0 = null1, bit 1 = null2.	

			<labelidx l> <heaptype t1> <heaptype t2>		Branches on null if null2 = 0. Branch at l >: [(ref null1 t1)]	
extern.internalize	0xfb70	**0xfb1a**		[(ref null? extern)] ->[(ref null? any)]		✔
extern.externalize	0xfb71	**0xfb1b**		[(ref null? any)] ->[(ref null? extern)]		✔
ref.i31	0xfb20	**0xfb1c**		[i32] -> [(ref i31)]	formerly i31.new	✔
i31.get_s	0xfb21	**0xfb1d**		[i31ref] -> [i32]	Note: i31ref was previously (ref i31) but is now (ref null i31)	
i31.get_u	0xfb22	**0xfb1e**		[i31ref] -> [i32]	Note: i31ref was previously (ref i31) but is now (ref null i31)	
~~struct.new_with_rtt~~	~~0xfb01~~	~~-~~	~~<typeidx>~~	~~[t'* (rtt $t)] -> [(ref $t)]~~	~~will disappear~~	~~✔~~
~~struct.new_default_with_rtt~~	~~0xfb02~~	~~-~~	~~<typeidx>~~	~~[(rtt $t)] -> [(ref $t)]~~	~~will disappear~~	~~✔~~
~~array.new_with_rtt~~	~~0xfb11~~	~~-~~	~~<typeidx>~~	~~[t' i32 (rtt $t)] -> [(ref $t)]~~	~~length argument interpreted as unsigned (u32), will disappear~~	
~~array.new_default_with_rtt~~	~~0xfb12~~	~~-~~	~~<typeidx>~~	~~[i32 (rtt $t)] -> [(ref $t)]~~	~~length unsigned, will disappear~~	
~~array.len~~	~~0xfb17~~	~~-~~	~~<typeidx>~~	~~[arrayref] -> [i32]~~	please use 0xfb19	
~~array.init~~	~~0xfb19~~	~~-~~	~~<typeidx> <u32>~~	~~[t'' (rtt $t)] -> [(ref $t)]~~	~~$t = (array t mutable)~~ ~~will disappear~~	~~✔~~
~~array.init_from_data~~	~~0xfb1e~~	~~-~~	~~<typeidx> <segmentidx>~~	~~[i32 i32 (rtt t)] -> [(ref $t)]~~	~~$t = (array t mutable?)~~ ~~t numeric, will disappear~~	~~✔~~

~~rtt.canon~~	~~0xfb30~~	-	~~<typeidx>~~	~~[] -> [(rtt $t)]~~	~~will disappear~~	✔
~~rtt.sub~~	~~0xfb31~~	-	~~<typeidx>~~	~~[(rtt n? t1)] -> [(rtt (n+1)? t2)]~~		✔
~~rtt.fresh_sub~~	~~0xfb32~~	-	~~<typeidx>~~	~~[(rtt n? t1)] -> [(rtt (n+1)? t2)]~~		?
~~ref.test~~	~~0xfb40~~	-		~~[(ref null t1) (rtt t2)] -> [i32]~~	~~will disappear~~	
ref.test	0xfb44	-	<typeidx t2>	[(ref null t1)] -> [i32]	deprecated, use 0xfb40 instead Returns 0 for null	
~~ref.cast~~	~~0xfb41~~	-		~~[(ref null? t1) (rtt t2)] -> [(ref null? t2)]~~	~~will disappear~~	
ref.cast	0xfb45	-	<typeidx t2>	[(ref null? t1)] -> [(ref null? t2)]	deprecated, use 0xfb41/0xfb49 instead null is passed through	
ref.cast_nop	0xfb4c	-	<typeidx 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)]~~	~~will disappear~~	
br_on_cast	0xfb42	-	<labelidx l> <heaptype t2>	[(ref null? t1)] -> [(ref null? t1)]	does not branch on null; branch at l must be >: [(ref t2)] Deprecated, use 0xfb4e instead	
br_on_cast null	0xfb4a	-	<labelidx l> <heaptype t2>	[(ref null? t1)] -> [(ref t1)]	branches on null; branch at l must be >: [(ref null? t2)] Deprecated, use 0xfb4e instead	
br_on_cast	0xfb46	-	<labelidx>	[(ref null? t1)] -> [(ref null? t1)]	deprecated, use 0xfb4e instead	

			<typeidx t2>			
~~br_on_cast_fail~~	~~0xfb43~~	-	~~<labelidx>~~	~~[(ref null? t1) (rtt' t2)] -> [(ref t2)]~~	~~will disappear~~	
br_on_cast_fail	0xfb43	-	<labelidx l> <heaptype t2>	[(ref null? t1)] -> [(ref t2)]	branches on null; branch at l must be >: [(ref null? t1)] Deprecated, use 0xfb4f instead	
br_on_cast_fail null	0xfb4b	-	<labelidx l> <heaptype t2>	[(ref null? t1)] -> [(ref null? t2)]	does not branch on null; branch at l must be >: [(ref t1)] Deprecated, use 0xfb4f instead	
br_on_cast_fail	0xfb47	-	<labelidx> <typeidx t2>	[(ref null? t1)] -> [(ref t2)]	deprecated, use 0xfb4f instead	
~~ref.is_func~~	~~0xfb50~~	-		~~[anyref] -> [i32]~~	~~will disappear~~	
ref.is_data	0xfb51	-		[anyref] -> [i32]	will disappear, use 0xfb40 instead	
ref.is_i31	0xfb52	-		[anyref] -> [i32]	will disappear, use 0xfb40 instead	
ref.is_array	0xfb53	-		[anyref] -> [i32]	will disappear, use 0xfb40 instead	
~~ref.as_func~~	~~0xfb58~~	-		~~[anyref] -> [(ref func)]~~	~~will disappear~~	
ref.as_data	0xfb59	-		[anyref] -> [dataref]	will disappear, use 0xfb41 instead	
ref.as_i31	0xfb5a	-		[anyref] -> [i31ref]	will disappear, use 0xfb41 instead	
ref.as_array	0xfb5b	-		[anyref] -> [arrayref]	will disappear, use 0xfb41 instead	
~~br_on_func~~	~~0xfb60~~	-	~~depth:<labelidx>~~	~~[t'* t] -> [t'* t] if t <: anyref~~	~~Branch at <depth> must be >: t'* (ref func);~~ ~~will disappear~~	

br_on_data	0xfb61	-	depth:<labelidx>	[t'* t] -> [t'* t] if t <: anyref	will disappear, use 0xfb42 instead	
br_on_i31	0xfb62	-	depth:<labelidx>	[t'* t] -> [t'* t] if t <: anyref	will disappear, use 0xfb42 instead	
br_on_array	0xfb66	-	depth:<labelidx>	[t'* t] -> [t'* t] if t <: anyref	will disappear, use 0xfb42 instead	
~~br_on_non_func~~	~~0xfb63~~	-	~~depth:<labelidx>~~	~~[t'* t] -> [t* (ref func)] if t <: anyref~~	~~Branch at <depth> must be >: t'* t;~~ will disappear	
br_on_non_data	0xfb64	-	depth:<labelidx>	[t'* t] -> [t* dataref] if t <: anyref	will disappear, use 0xfb43 instead	
br_on_non_i31	0xfb65	-	depth:<labelidx>	[t'* t] -> [t* i31ref] if t <: anyref	will disappear, use 0xfb43 instead	
br_on_non_array	0xfb67	-	depth:<labelidx>	[t'* t] -> [t* arrayref] if t <: anyref	will disappear, use 0xfb43 instead	

Encoding:

- <typeidx> : u32
- <heaptype> : 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 $typeidx = (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 $typeidx 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.
- Active segments count as empty

array.new_elem

- Takes an immediate array type index $typeidx = (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 $typeidx 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.

array.fill

- Takes an immediate array type index. The parameters on the stack are the array, index, value and length.
- Traps if (index + length) > array.len.
- Otherwise sets elements in the range [index, index+length) to value.

array.init_data/init.elem

- Takes an immediate $array_type index and a (data/element) $segment index.
- From the stack, it takes an $array, an $array_index, a $segment_offset and a $length.
- Validation: $array_index + $length <= $array.len
    - For data segments, $segment_offset + $length * sizeof($array_type.element_type) <= $segment.length.
    - For element segments, $segment_offset + $length <= $segment.length.
- Initializes $length elements of $array from $segment, starting at $array_index and $segment_offset. Bytes in data segments are

interpreted as full-length little-endian elements of the array type.
- 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 0x00 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.