Iris-Wasm: A Higher-Order Mechanised Program Logic For WebAssembly

Xiaojia Rao
Imperial College London

Joint work with

Aïna Linn Georges, Maxime Legoupil, Jean Pichon-Pharabod, Lars Birkedal	**Aarhus University**
Conrad Watt	**Cambridge University**
Philippa Gardner	**Imperial College London**

WebAssembly (Wasm)

What is WebAssembly?

- A modern bytecode language supported by all major browsers
- An efficient compilation target for low-level languages (e.g. C/C++)
- Specified through formal semantics from the start
- Code distributed in modules — the unit of compilation

WebAssembly (Wasm)

Related Previous Work

- Wasm formal semantics (Haas et al, PLDI 2017)
- Isabelle mechanisation (Watt, CPP 2018)
- First-order encapsulated Wasm Program Logic (Watt et al, ECOOP 2018)
- Wasm 1.0, official W3C-Recommendation (2019)
- Isabelle and Coq mechanisation of Wasm 1.0 (Watt et al, FM 2021)

Iris-Wasm: Goals

- A mechanised program logic of Wasm using Iris
  - Based on the previous faithful representation of the Wasm semantics
  - Robust safety examples via logical relation defined on the language…
    - But unknown code is only a thing where multiple modules are involved
- A lightweight host language that supports module instantiation
  - Also enables building modular specification for Wasm modules

Talk Overview

- Program Logic for Native WebAssembly
- Modules and Host Language

Program Logic for Native WebAssembly

WebAssembly Overview

- Stack-based language
  - Instruction stack and value stack
- Small-step operational semantics
- Syntactic type-check (validation) before execution

Program Logic for Native WebAssembly

Basic Wasm Definitions

<table>
<tr><td>Value</td><td>v</td><td rowspan="2">32/64-bit integers and floats</td></tr>
<tr><td>Value type</td><td>t ::= i32 | i64 | f32 | f64</td></tr>
<tr><td>Function type</td><td>ft ::= [t*] -> [t*]</td><td rowspan="2">Each instruction has a static function type for validation</td></tr>
<tr><td>Instruction</td><td>e ::= t.const v | t.add |
block ft e* | loop ft e* | br n |
call n | call_indirect n | return |
load … | store … | …</td></tr>
</table>

(Trivial) Embedding in Iris

<table>
<tr><td>Iris val</td><td>iris_v ::= v*</td></tr>
<tr><td>Iris expr</td><td>iris_e ::= e*</td></tr>
</table>

Program Logic for Native WebAssembly

Example: Numeric Instructions

(i32.const 2)
(i32.const 10)
(i32.const 30)
(i32.add)
(i32.add)

(i32.const 2)
(i32.const 40)
(i32.add)

(i32.const 42)

- Following the official specification, value stack is not implemented explicitly
- Leading list of constants is interpreted as the value stack
- A very simple wp rule demonstrating the semantic behaviour of addition:

$$\frac{\triangleright\Phi(v_1 +_t v_2)}{\texttt{wp}\ \ [\texttt{t.const}\ \ v_2; \texttt{t.const}\ \ v_1; \texttt{t.add}]\ \{v.\Phi(v)\}}\ \texttt{wp\_add}$$

Program Logic for Native WebAssembly

Example: Control Flow

Continuation of a label: Instructions to execute if targeted by a br (empty for blocks)

```
(Block ...
    [(Block ...
        [(i32.const 2)
        (i32.const 40)
        (i32.add)
        (br 0)]
    )]
)
```

Intermediate steps omitted

```
(Label ... []
    [(Label ... []
        [(i32.const 2)
        (i32.const 40)
        (i32.add)
        (br 0)]
    )]
)
```

```
(Label ... []
    [(Label ... []
        [(i32.const 42)
        (br 0)]
    )]
)
```

```
(Label ... []
        [(i32.const 42)]
)
```

- Blocks reduce to Label instructions (Labels are somewhat similar to evaluation contexts)
- br breaks out of the corresponding Label, taking (some) values out of Label, and pushes the continuation of Label to the instruction stack and continue from there

Program Logic for Native WebAssembly

Example: Control Flow

```
(Block …
    [(Block …
        [(i32.const 2)
        (i32.const 40)
        (i32.add)
        (br 1)]
    )]
)
```

Intermediate steps omitted

```
(Label … []
    [(Label … []
        [(i32.const 2)
        (i32.const 40)
        (i32.add)
        (br 1)]
    )]
)
```

```
(Label … []
    [(Label … []
        [(i32.const 42)
        (br 1)]
    )]
)
```

```
(i32.const 42)
```

Program Logic for Native WebAssembly

Example: Control Flow

- Cannot treat labels as simple evaluation contexts due to *br*
- Forget about this idea of context and treat labels as normal expressions?
  - Hard to craft the traditional *bind* rules for labels
  - Awkward to apply the resulting wp rules to actual programs
- Solution:
  - Consider a ***group*** of nested labels as an evaluation context
  - Extend the definition of values to include ***stuck*** *br*

Program Logic for Native WebAssembly

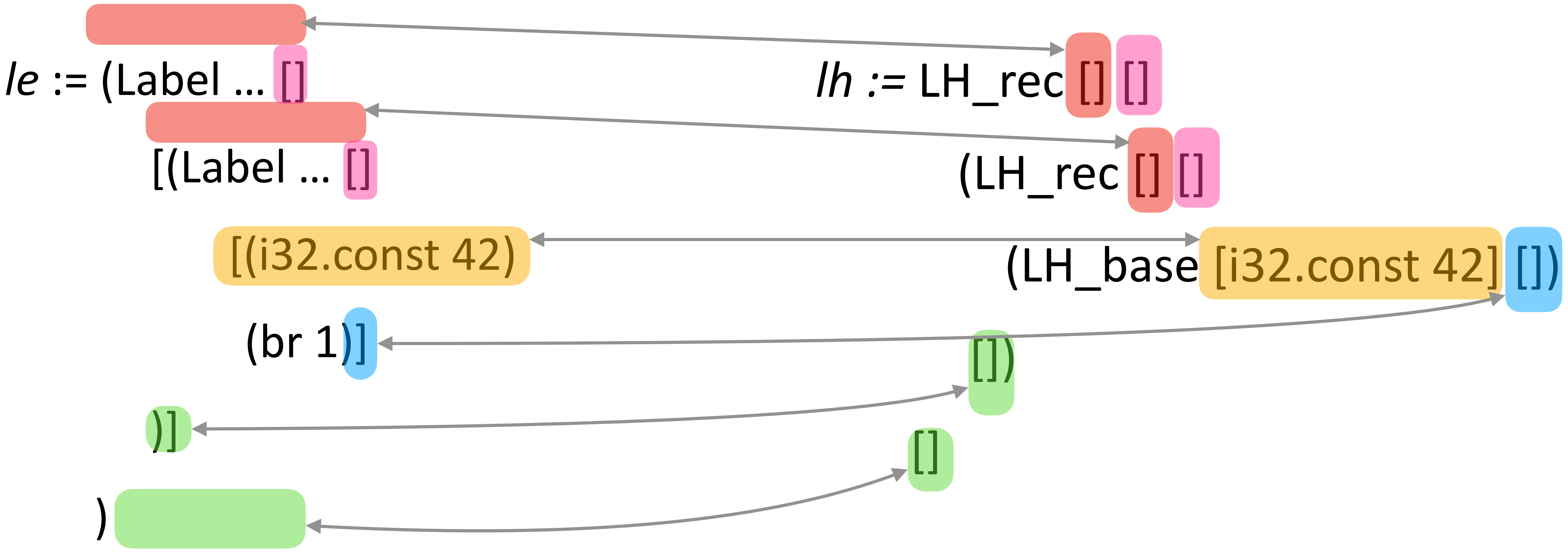
Example: Control Flow

Iris val	*iris_v* ::= immV *v** | brV *n lh* | …
Label hole context	*lh* ::= LH_base *v* e** | LH_rec *v* e*_cont lh e*_exec*

- *lh* describes a (nested) label context surrounding a hole
- In the brV constructor of *iris_v*, *lh* is required to be shallow enough for (*br n*) to get stuck

- Filling a *lh* context with an expression:

  lh_fill *lh* (br 1) = *le* = of_val (brV 1 *lh*)



Program Logic for Native WebAssembly

Example: Control Flow

- An auxiliary notation of context-wp is defined for dealing with contexts more easily

$$\texttt{wp\_ctx}\ e\ lh\ \{\Phi\} ::= \texttt{wp}\ (\texttt{lh\_fill}\ lh\ e)\ \{\Phi\}$$

- A number of rules proved to handle context manipulation

$$\frac{\ldots\ *\ \texttt{wp}\ e\ \{w.\ \texttt{wp\_ctx}\ \ (\texttt{of\_val}\ \ w)\ lh\ \{v.\Phi(v)\}\}}{\texttt{wp\_ctx}\ \ e\ lh\ \{v.\Phi(v)\}}\ \texttt{wp\_ctx\_bind}$$

- Together with a rule for *br* within an appropriate context, this allowed the spec of the previous example to be proved

$$\frac{\ldots\ *\ lh = \texttt{LH\_rec}\ \ vs'\ \ e_c\ \ lh'\ \ e\ *\ \texttt{depth}(lh) = (i+1)\ *\ \texttt{wp}\ (vs' \mathbin{++} vs \mathbin{++} e_c \mathbin{++} e)\ \{v.\Phi(v)\}}{\texttt{wp\_ctx}\ \ (vs \mathbin{++} (\texttt{br}\ \ i))\ lh\ \{v.\Phi(v)\}}\ \texttt{wp\_br\_ctx}$$

```
(Label … []
   [(Label … []
      [(i32.const 2)
      (i32.const 40)
      (i32.add)
      (br 1)]
   )]
)
```

le := (Label … []
   [(Label … []
      [(i32.const 42)
      (br 1)]
)]
)

(i32.const 42)

Program Logic for Native WebAssembly

Example: Wasm State and Function Call

- Wasm state consisting of two records:
  - The global store *S,* collecting all resources allocated by instantiation of modules
  - The local frame (local environment) *F*, which itself consists of:
    - A list of local variables *locs*;
    - A local runtime instance *inst* containing function types used, and addresses to each field of the global store *S*.
- Each component of the store is modelled by an individual heap in the memory model
- Frame cannot be split or shared in anyway, so is modelled by a unit resource

$$S := \left\{ \begin{array}{ll} \texttt{funcs} & := func^* \\ \texttt{tabs} & := tab^* \\ \texttt{mems} & := mem^* \\ \texttt{globs} & := glob^* \end{array} \right\}$$

$$F := \left\{ \begin{array}{ll} \texttt{locs} & := v^* \\ \texttt{inst} & := \left\{ \begin{array}{ll} \texttt{types} & := ft^* \\ \texttt{funcaddr} & := addr^* \\ \texttt{tabaddr} & := addr^* \\ \texttt{memaddr} & := addr^* \\ \texttt{globaddr} & := addr^* \end{array} \right\} \end{array} \right\}$$

$$(0 \overset{wf}{\longmapsto} fn) * (0 \overset{wm}{\longmapsto}_{42} (\texttt{0x61})) * (1 \overset{wg}{\longmapsto} gv) * \ldots$$

$$() \overset{frame}{\hookrightarrow} F$$

$$(S, F, e) \hookrightarrow (S', F', e')$$

Program Logic for Native WebAssembly

Example: Wasm State and Function Call

- We start with a state *(S, F)* given by:
- And execute [(i32.const *42*); (call *0*)] under this state.

$$S := \left\{ \begin{array}{ll} funcs & := [f_0, f_1, f_2] \\ \dots & \end{array} \right\}$$

$$F := \left\{ \begin{array}{ll} locs & := [] \\ inst & := \left\{ \begin{array}{ll} \dots & \\ funcaddr & := [1] \end{array} \right\} \end{array} \right\}$$

$$f_1 := \left\{ \begin{array}{ll} ft & := [\mathtt{i32}] \to [\mathtt{i32}] \\ body & := [(\mathtt{get\_local\ 0});\ (\mathtt{get\_local\ 0});\ (\mathtt{i32.add});\ (\mathtt{return})] \\ \dots & \end{array} \right\}$$

$$F' := \left\{ \begin{array}{ll} locs & := [\mathtt{i32.const\ 42}] \\ inst & := \dots \end{array} \right\}$$

[(i32.const 42);

(call 0)]

↪

[(i32.const 42);

(invoke 1)]

↪

[(Local ***F'***

[(get_local 0);

(get_local 0);

(i32.add);

(return)

]

↪

...

$$\frac{(S, F, es) \hookrightarrow (S', F', es')}{(S, F_0, [\mathtt{Local}\ \ F\ es]) \hookrightarrow (S', F_0, [\mathtt{Local}\ \ F'\ es'])}\ \mathtt{Local\_lift}$$

Program Logic for Native WebAssembly

Limitation of Native Wasm Code

- No native Wasm instructions can modify the list of function closures or function tables
  - Calling static code in the store only, no real higher-order functions
- Function closures and function tables are results of resource allocations during module instantiation
  - A host supporting instantiation would allow more interesting examples
  - Host can also choose to directly provide operations with more expressive power (e.g. Wasm-JS API)

Talk Overview

- Program Logic for Native WebAssembly
- Modules and Host Language

Modules and Host Language

WebAssembly Module

- A Wasm module *M* is a large record:
  - *types* collect the function types used in the module
  - *funcs, tabs, mems, globs* contain declarations of the corresponding resources of the module
  - *elem, data* are initialisers for tables and memories
  - *import* states the type of the imports expected
  - *export* states which resources declared by the modules are exposed to be used by other modules
  - *start* optionally chooses one of the functions declared to be executed immediately after instantiating the module

$$M := \left\{ \begin{array}{ll} \texttt{types} & := \ldots \\ \texttt{funcs} & := \ldots \\ \texttt{tabs} & := \ldots \\ \texttt{mems} & := \ldots \\ \texttt{globs} & := \ldots \\ \texttt{elem} & := \ldots \\ \texttt{data} & := \ldots \\ \texttt{import} & := \ldots \\ \texttt{export} & := \ldots \\ \texttt{start} & := \ldots \end{array} \right\}$$

Modules and Host Language

Example: Stack Module (Fragment)

- Defines one function of type [] -> [i32] which declares one local variable, with a function body consisting of native Wasm code
- Defines a memory with initial size 0 and no maximum limit
- Exports the 0th function and name it “new_stack”

```
Definition stack_module :=
  {|
    types := [
      Tf [] [T_i32] ; ...
    ] ;
    funcs := [
      {|
        modfunc_type := Mk_typeidx 0 ; (* Function type *)
        modfunc_locals := [T_i32] ; (* Type of local variables needed to be declared *)
        modfunc_body := new_stack (* Function body *)
      |} ; ...
    ] ;
    mems := [
    (* Declare a memory with minimal and initial size of 0 and no maximum size *)
      {| lim_min := 0%N ; lim_max := None |}
    ] ;
    ...
    exports := [
      {|
      (* Export the 0th function of the module and call it "new_stack". *)
        modexp_name := list_byte_of_string "new_stack" ;
        modexp_desc := MED_func (Mk_funcidx 0)
      |} ; ...
    ]
  |}.
```

Modules and Host Language

Example: Stack Module (Fragment)

- Function body of new_stack:
  - attempts to allocate 1 new page in the memory (64KB);
  - if successful, maintain an abstract stack data structure on that continuous segment of memory and return the address of the starting byte.

```
Definition new_stack :=
  [ (* First, try to grow the memory by 1 page (64KB) *)
    i32.const 1 ;
    grow_memory ;
(* The above pushes a i32.(-1) to the value stack if failed, else the original memory size
(* We save a copy of the result to the local variable 0 *)
    tee_local 0 ;
    i32.const (-1) ;
(* Is the result -1? *)
    i32.eq;
    if [
(* It is -- well unlucky, we return -1 as well *)
    i32.const (-1)
    ]
    [
(* Push the previously stored local 0 to the value stack *)
        get_local 0 ;
        i32.const 65536 ;
(* Multiplying by 64K, this gives us the number of bytes in the original memory,
which is the starting byte of the newly allocated memory. Call it x *)
        i32.mul;
(* Save a copy of x to local 0 *)
        tee_local 0 ;
(* Get another one to the stack, essentially duplicating it *)
        get_local 0 ;
        i32.const 4;
(* Add 4 (=sizeof(i32)) to the value *)
        i32.add;
(* At this point the stack contais two values: x + 4 and x.*)
(* Store x + 4 to location x of the memory *)
        i32.store;
(* Retrieve a copy of x stored in local 0 for return *)
        get_local 0
        ]
  ].
```

Modules and Host Language

Module Instantiation

- Allocates each resource declared by the module and add to the current global store *S*
- Initialisation of resources
- …
- Returns:
    - A list of exports that can be imported by other modules later
    - The resulting Wasm global store *S'* after instantiation
- Essentially like ‘importing’ the module into our global context *S*

$$
\begin{array}{rcl}
\mathrm{instantiate}(S, \mathit{module}, \mathit{externval}^n) & = & S'; F; \begin{array}[t]{l} (\mathsf{init\_elem}\ \mathit{tableaddr}\ \mathit{eo}\ \mathit{elem}.\mathsf{init})^* \\ (\mathsf{init\_data}\ \mathit{memaddr}\ \mathit{do}\ \mathit{data}.\mathsf{init})^* \\ (\mathsf{invoke}\ \mathit{funcaddr})^? \end{array} \\
& (\mathrm{if} & \vdash \mathit{module} : \mathit{externtype}_{\mathrm{im}}^n \rightarrow \mathit{externtype}_{\mathrm{ex}}^* \\
& \wedge & (S \vdash \mathit{externval} : \mathit{externtype})^n \\
& \wedge & (\vdash \mathit{externtype} \leq \mathit{externtype}_{\mathrm{im}})^n \\
& \wedge & \mathit{module}.\mathsf{globals} = \mathit{global}^* \\
& \wedge & \mathit{module}.\mathsf{elem} = \mathit{elem}^* \\
& \wedge & \mathit{module}.\mathsf{data} = \mathit{data}^* \\
& \wedge & \mathit{module}.\mathsf{start} = \mathit{start}^? \\
& \wedge & S', \mathit{moduleinst} = \mathrm{allocmodule}(S, \mathit{module}, \mathit{externval}^n, \mathit{val}^*) \\
& \wedge & F = \{\mathsf{module}\ \mathit{moduleinst}, \mathsf{locals}\ \epsilon\} \\
& \wedge & (S'; F; \mathit{global}.\mathsf{init} \hookrightarrow^* S'; F; \mathit{val}\ \mathsf{end})^* \\
& \wedge & (S'; F; \mathit{elem}.\mathsf{offset} \hookrightarrow^* S'; F; \mathsf{i32.const}\ \mathit{eo}\ \mathsf{end})^* \\
& \wedge & (S'; F; \mathit{data}.\mathsf{offset} \hookrightarrow^* S'; F; \mathsf{i32.const}\ \mathit{do}\ \mathsf{end})^* \\
& \wedge & (\mathit{eo} + |\mathit{elem}.\mathsf{init}| \leq |S'.\mathsf{tables}[\mathit{tableaddr}].\mathsf{elem}|)^* \\
& \wedge & (\mathit{do} + |\mathit{data}.\mathsf{init}| \leq |S'.\mathsf{mems}[\mathit{memaddr}].\mathsf{data}|)^* \\
& \wedge & (\mathit{tableaddr} = \mathit{moduleinst}.\mathsf{tableaddrs}[\mathit{elem}.\mathsf{table}])^* \\
& \wedge & (\mathit{memaddr} = \mathit{moduleinst}.\mathsf{memaddrs}[\mathit{data}.\mathsf{data}])^* \\
& \wedge & (\mathit{funcaddr} = \mathit{moduleinst}.\mathsf{funcaddrs}[\mathit{start}.\mathsf{func}])^?)
\end{array}
$$

$$
\begin{array}{rcl}
S; F; \mathsf{init\_elem}\ a\ i\ \epsilon & \hookrightarrow & S; F; \epsilon \\
S; F; \mathsf{init\_elem}\ a\ i\ (x_0\ x^*) & \hookrightarrow & S'; F; \mathsf{init\_elem}\ a\ (i+1)\ x^* \\
& & (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{tables}[a].\mathsf{elem}[i] = F.\mathsf{module}.\mathsf{funcaddrs}[x_0]) \\
S; F; \mathsf{init\_data}\ a\ i\ \epsilon & \hookrightarrow & S; F; \epsilon \\
S; F; \mathsf{init\_data}\ a\ i\ (b_0\ b^*) & \hookrightarrow & S'; F; \mathsf{init\_data}\ a\ (i+1)\ b^* \\
& & (\mathrm{if}\ S' = S\ \mathsf{with}\ \mathsf{mems}[a].\mathsf{data}[i] = b_0)
\end{array}
$$

Modules and Host Language

Example: Instantiating the Stack Module (Fragment)

- Given any existing store *S*, instantiating this module:
    - Pushes an additional function closure corresponding for new_stack to the end of *S*.(funcs);
    - Pushes a new memory (initially empty) to the end of *S*.(mems);
    - Generates an export corresponding to the new_stack function to the host language; the host should store it for potential future use by other modules;
    - …

```
Definition stack_module :=
  {|
    types := [
      Tf [] [T_i32] ; ...
    ] ;
    funcs := [
      {|
        modfunc_type := Mk_typeidx 0 ; (* Function type *)
        modfunc_locals := [T_i32] ; (* Type of local variables needed to be declared *)
        modfunc_body := new_stack (* Function body *)
      |} ; ...
    ] ;
    mems := [
    (* Declare a memory with minimal and initial size of 0 and no maximum size *)
      {| lim_min := 0%N ; lim_max := None |}
    ] ;
    ...
    exports := [
      {|
      (* Export the 0th function of the module and call it "new_stack". *)
        modexp_name := list_byte_of_string "new_stack" ;
        modexp_desc := MED_func (Mk_funcidx 0)
      |} ; ...
    ]
  |}.
```

Modules and Host Language

Host Language

- Implemented a host language handling module instantiation
- Crafted a wp rule characterising the behaviour of instantiation
  - Used it to verify an example stack module with a higher order map function
- Notable features:
  - Host memory is a superset of the Wasm memory
    - An additional heap that stores instantiated exports
  - A separated wp to reason about host programs and resources
  - Host wp depends on the Wasm one due to the presence of start functions which can call Wasm code
  - 2 languages with 2 dependent but different wps, working on a similar set of memory model

Modules and Host Language

Example: Stack Module

- 6 functions, all exported
  - new_stack/is_empty/is_full/pop/push/stack_map
- 1 function table, exported
  - 'Interface' for client modules to feed functions to use the higher-order stack map
- 1 memory, not exported
  - Encapsulation property will guarantee that the memory can only be accessed through the interfaces we exposed
- Modular specification for instantiation verified using the current instantiation wp rule
  - Provide specifications of exported functions in the post
- A client module that tests our module, also verified

```
Definition stack_module :=
  {|
    mod_types := [
      Tf [] [T_i32] ;
      Tf [T_i32] [T_i32] ;
      Tf [T_i32 ; T_i32] []
    ] ;
    mod_funcs := [
      {|
        modfunc_type := Mk_typeidx 0 ;
        modfunc_locals := [T_i32] ;
        modfunc_body := mk_basic_expr new_stack
      |} ;
      {|
        modfunc_type := Mk_typeidx 1 ;
        modfunc_locals := [] ;
        modfunc_body := mk_basic_expr is_empty
      |} ;
      {|
        modfunc_type := Mk_typeidx 1 ;
        modfunc_locals := [] ;
        modfunc_body := mk_basic_expr is_full
      |} ;
      {|
        modfunc_type := Mk_typeidx 1 ;
        modfunc_locals := [T_i32] ;
        modfunc_body := mk_basic_expr pop
      |} ;
      {|
        modfunc_type := Mk_typeidx 2 ;
        modfunc_locals := [T_i32] ;
        modfunc_body := mk_basic_expr push
      |} ;
      {|
        modfunc_type := Mk_typeidx 2 ;
```

```
        modfunc_locals := [T_i32 ; T_i32] ;
        modfunc_body := mk_basic_expr stack_map
      |}
    ] ;
    mod_tables := [ {| modtab_type := {| tt_limits := {| lim_min := 1%N ; lim_max := None
                                         tt_elem_type := ELT_funcref |} |} ] ;
    mod_mems := [
      {| lim_min := 0%N ; lim_max := None |}
    ] ;
    mod_globals := [] ;
    mod_elem := [] ;
    mod_data := [] ;
    mod_start := None ;
    mod_imports := [] ;
    mod_exports := [
      {|
        modexp_name := list_byte_of_string "new_stack" ;
        modexp_desc := MED_func (Mk_funcidx 0)
      |} ;
      {|
        modexp_name := list_byte_of_string "is_empty" ;
        modexp_desc := MED_func (Mk_funcidx 1)
      |} ;
      {|
        modexp_name := list_byte_of_string "is_full" ;
        modexp_desc := MED_func (Mk_funcidx 2)
      |} ;
      {|
        modexp_name := list_byte_of_string "pop" ;
        modexp_desc := MED_func (Mk_funcidx 3)
      |} ;
      {|
        modexp_name := list_byte_of_string "push" ;
        modexp_desc := MED_func (Mk_funcidx 4)
      |} ;
      {|
        modexp_name := list_byte_of_string "stack_map" ;
        modexp_desc := MED_func (Mk_funcidx 5)
      |} ;
      {|
        modexp_name := list_byte_of_string "table" ;
        modexp_desc := MED_table (Mk_tableidx 0)
      |}
    ]
  |}.
```

Robust Safety

- What if unknown code is present — e.g. a module import functions from an unknown module?
  - Encapsulation of resources: external code has no access to resources in the module unless exported
- Defined a logical relation over the entire program logic
  - Large relation due to size of the language, but canonical
  - Proved examples demonstrating the robust safety property
  - The imported function $*f* from the unknown module $*adv* cannot modify the encapsulated memory and global variable

$$\{\ldots * \$ret \overset{wg}{\longmapsto} - \;*\; f.\mathtt{mem} \overset{wm}{\longmapsto}_0 -\}\ \mathtt{main}\ \{\ldots * \$ret \overset{wg}{\longmapsto} 42 \;*\; f.\mathtt{mem} \overset{wm}{\longmapsto}_0 42\}$$

```
(module
  (type ...)
  (import "adv" "f" (func $f))
  (table 1)
  (memory 1)

  (func $main
    i32.const 0
    i32.const 42
    store
    i32.const 0
    call $f
    global.set $ret))
```

Future Work

- Verify some real world code in Wasm
- Wasm is still an evolving language
    - Wasm 2.0 (currently a candidate draft)
    - Additional language features to Wasm, e.g. capability