# WebAssembly Component Model

***Release 0.0 (Draft 2023-01-10)***

**Authors of the Webassembly Component Model Specification**

**Jan 10, 2023**

# CONTENTS:

CHAPTER

ONE

# INTRODUCTION

TODO: Introduction

CHAPTER

TWO

# STRUCTURE

## 2.1 Conventions

The WebAssembly component specification defines a language for specifying components, which, like the WebAssembly core language, may be represented by multiple complete representations (e.g. the *binary format* and the *text format*). In order to avoid duplication, the static and dynamic semantics of the WebAssembly component model are instead defined over an abstract syntax.

The following conventions are adopted in defining grammar rules for abstract syntax.

- Terminal symbols (atoms) are written in sans-serif font: i32, end.
- Nonterminal symbols are written in italic font: $valtype$, $instr$.
- $A^n$ is a sequence of $n \geq 0$ iterations of $A$.
- $A^*$ is a possibly empty sequence of iterations of $A$. (This is a shorthand for $A^n$ used where $n$ is not relevant.)
- $A^+$ is a non-empty sequence of iterations of $A$. (This is a shorthand for $A^n$ where $n \geq 1$.)
- $A^?$ is an optional occurrence of $A$. (This is a shorthand for $A^n$ where $n \leq 1$.)
- Productions are written $sym ::= A_1 \mid \dots \mid A_n$.
- Large productions may be split into multiple definitions, indicated by ending the first one with explicit ellipses, $sym ::= A_1 \mid \dots$, and starting continuations with ellipses, $sym ::= \dots \mid A_2$.
- Some productions are augmented with side conditions in parentheses, "$(\mathrm{if}\ condition)$", that provide a shorthand for a combinatorial expansion of the production into many separate cases.
- If the same meta variable or non-terminal symbol appears multiple times in a production, then all those occurrences must have the same instantiation. (This is a shorthand for a side condition requiring multiple different variables to be equal.)

## 2.2 Types

The component model introduces two new kinds of types: value types, which are used to classify shared-nothing interface values, and definition types, which are used to characterize the core and component modules, instances, and functions which form part of a a component's interface.

### 2.2.1 Value types

A *value type* classifies a component-level abstract value. Unlike for Core WebAssembly values, no specified abstract syntax of component values exist; they serve simply to define the interface of lifted component functions (which currently may be produced only via canonical definitions).

Value types are further divided into primitive value types, which have a compact representation and can be found in most places where types are allowed, and defined value types, which must appear in a type definition before they can be used (via a $typeidx$ into the type index space):

$$
\begin{array}{llll}
primvaltype & ::= & \mathsf{bool} \\
& | & \mathsf{s8} \mid \mathsf{u8} \mid \mathsf{s16} \mid \mathsf{u16} \mid \mathsf{s32} \mid \mathsf{u32} \mid \mathsf{s64} \mid \mathsf{u64} \\
& | & \mathsf{float32} \mid \mathsf{float64} \\
& | & \mathsf{char} \mid \mathsf{string} \\
& | & \\
defvaltype & ::= & \mathsf{prim}\ primvaltype \\
& | & \mathsf{record}\ record\_field^+ \\
& | & \mathsf{variant}\ variant\_case^+ \\
& | & \mathsf{list}\ valtype \\
& | & \mathsf{tuple}\ valtype^* \\
& | & \mathsf{flags}\ name^* \\
& | & \mathsf{enum}\ name^+ \\
& | & \mathsf{union}\ valtype^+ \\
& | & \mathsf{option}\ valtype \\
& | & \mathsf{result}\ valtype^?\ valtype^? \\
& | & \mathsf{own}\ typeidx \\
& | & \mathsf{borrow}\ typeidx \\
valtype & ::= & primvaltype \mid typeidx
\end{array}
$$

$$
\begin{array}{lll}
record\_field & ::= & \{\mathsf{name}\ name, \mathsf{type}\ valtype\} \\
variant\_case & ::= & \{\mathsf{name}\ name, \mathsf{type}\ valtype, \mathsf{refines}\ u32^?\}
\end{array}
$$

### 2.2.2 Resource types

$$
resourcetype \quad ::= \quad \{\mathsf{rep}\ \mathtt{i32}, \mathsf{dtor}\ funcidx\}
$$

### 2.2.3 Function types

A component-level shared-nothing function is classified by the types of its parameters and return values. Such a function may take as parameters zero or more named values, and will return as results zero or more namde values. If a function takes a single parameter, or returns a single result, said parameter or result may be unnamed:

$$
functype \quad ::= \quad resulttype \to resulttype
$$

The input or output of a function is classified by a result type:

$$
\begin{array}{lll}
resulttype & ::= & valtype \\
& | & \{\mathsf{name}\ name, \mathsf{type}\ valtype\}^* \quad n
\end{array}
$$

### 2.2.4 Instance types

A component instance is conceptually classified by the types of its exports. However, an instance's type is concretely represented as a series of *declarations* manipulating index spaces (particular to the instance type; these index spaces are entirely unrelated to both the index spaces of any instance which has this type and those of any instance importing or exporting something of this type). This allows for better type sharing and, in the future, uses of private types from parent components.

$$
\begin{array}{lcl}
\mathit{instancetype} & ::= & \mathit{instancedecl}^* \\
\mathit{instancedecl} & ::= & \mathsf{alias}\ \mathit{alias} \\
 & | & \mathsf{core\_type}\ \mathit{core{:}type} \\
 & | & \mathsf{type}\ \mathit{deftype} \\
 & | & \mathsf{export}\ \mathit{exportdecl} \\
\mathit{externdesc} & ::= & \mathsf{type}\ \mathit{typebound} \\
 & | & \mathsf{core\_module}\ \mathit{core{:}typeidx} \\
 & | & \mathsf{func}\ \mathit{typeidx} \\
 & | & \mathsf{value}\ \mathit{valtype} \\
 & | & \mathsf{instance}\ \mathit{typeidx} \\
 & | & \mathsf{component}\ \mathit{typeidx} \\
\mathit{typebound} & ::= & \mathsf{EQ}\ \mathit{typeidx} \\
 & | & \mathsf{SUB}\ \mathsf{resource} \\
 & | & \dots \\
\mathit{exportdecl} & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathit{externdesc}\}
\end{array}
$$

### 2.2.5 Component types

A component is conceptually classified by the types of its imports and exports. However, like instances, this is concretely represented as a series of declarations; in particular, a similar set of declarations allowing also for imports.

$$
\begin{array}{lcl}
\mathit{componenttype} & ::= & \mathit{componentdecl}^* \\
\mathit{componentdecl} & ::= & \mathit{instancedecl} \\
 & | & \mathsf{import}\ \mathit{importdecl} \\
\mathit{importdecl} & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathit{externdesc}\}
\end{array}
$$

### 2.2.6 Definition types

A type definition may name a value, resource, function, component, or instance type:

$$
\begin{array}{lcl}
\mathit{deftype} & ::= & \mathit{defvaltype} \\
 & | & \mathit{resourcetype} \\
 & | & \mathit{functype} \\
 & | & \mathit{componenttype} \\
 & | & \mathit{instancetype}
\end{array}
$$

### 2.2.7 Core definition types

The component module specification also defines an expanded notion of what a core type is, which may eventually be subsumed by a core module linking extension.

$$
\begin{array}{lll}
core{:}deftype & ::= & core{:}functype \\
 & | & core{:}moduletype \\
core{:}moduletype & ::= & coremoduledecl^* \\
coremoduledecl & ::= & core{:}importdecl \\
 & | & core{:}deftype \\
 & | & core{:}alias \\
 & | & core{:}exportdecl \\
core{:}alias & ::= & \{\mathsf{sort}\ core{:}sort, \mathsf{target}\ corealiastarget\} \\
corealiastarget & ::= & \mathsf{outer}\ u32\ u32 \\
core{:}importdecl & ::= & core{:}import \\
core{:}exportdecl & ::= & \{\mathsf{name}\ name, \mathsf{desc}\ core{:}importdesc\}
\end{array}
$$

## 2.3 Components

### 2.3.1 Sorts

A component's definitions define objects, each of which is of one of the following *sorts*:

$$
\begin{array}{lll}
core{:}sort & ::= & \mathsf{func} | \mathsf{table} | \mathsf{memory} | \mathsf{global} | \mathsf{type} | \mathsf{module} | \mathsf{instance} \\
sort & ::= & \mathsf{core}\ core{:}sort \\
 & | & \mathsf{func} | \mathsf{value} | \mathsf{type} | \mathsf{component} | \mathsf{instance}
\end{array}
$$

### 2.3.2 Indices

Each object defined by a component exists within an *index space* made up of all objects of the same sort. Unlike in Core WebAssembly, a component definition may only refer to objects that were defined prior to it in the current component. Future definitions refer to past definitions by means of an *index* into the appropriate index space:

$$
\begin{array}{lll}
core{:}moduleidx & ::= & u32 \\
core{:}instanceidx & ::= & u32 \\
componentidx & ::= & u32 \\
instanceidx & ::= & u32 \\
funcidx & ::= & u32 \\
core{:}funcidx & ::= & u32 \\
valueidx & ::= & u32 \\
typeidx & ::= & u32 \\
core{:}typeidx & ::= & u32
\end{array}
$$

$$
\begin{array}{lll}
core{:}sortidx & ::= & \{\mathsf{sort}\ core{:}sort, \mathsf{idx}\ u32\} \\
sortidx & ::= & \{\mathsf{sort}\ sort, \mathsf{idx}\ u32\}
\end{array}
$$

### 2.3.3 Definitions

Each object within a component is defined by a *definition*, of which there are several kinds:

$$\begin{array}{lcl} \mathit{definition} & ::= & \mathsf{core\_module}\ \mathit{core{:}module} \\ & | & \mathsf{core\_instance}\ \mathit{core{:}instance} \\ & | & \mathsf{core\_type}\ \mathit{core{:}deftype} \\ & | & \mathsf{component}\ \mathit{component} \\ & | & \mathsf{instance}\ \mathit{instance} \\ & | & \mathsf{alias}\ \mathit{alias} \\ & | & \mathsf{type}\ \mathit{deftype} \\ & | & \mathsf{canon}\ \mathit{canon} \\ & | & \mathsf{start}\ \mathit{start} \\ & | & \mathsf{import}\ \mathit{import} \\ & | & \mathsf{export}\ \mathit{export} \end{array}$$

### 2.3.4 Core instances

A core instance may be defined either by instantiating a core module with other core instances taking the place of its first-level imports, or by creating a core module from whole cloth by combining core definitions already present in our index space:

$$\begin{array}{lcl} \mathit{core{:}instance} & ::= & \mathsf{instantiate}\ \mathit{core{:}moduleidx}\ \mathit{core{:}instantiatearg}^* \\ & | & \mathsf{exports}\ \mathit{core{:}export}^* \\ \mathit{core{:}instantiatearg} & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{instance}\ \mathit{core{:}instanceidx}\} \\ \mathit{core{:}export} & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{def}\ \mathit{core{:}sortidx}\} \end{array}$$

### 2.3.5 Components

A component is merely a sequence of definitions:

$$\mathit{component} \quad ::= \quad \mathit{definition}^*$$

### 2.3.6 Instances

Component-level instance declarations are nearly identical to core-level instance declarations, with the caveat that more sorts of definitions may be supplied as imports:

$$\begin{array}{lcl} \mathit{instance} & ::= & \mathsf{instantiate}\ \mathit{componentidx}\ \mathit{instantiatearg}^* \\ & | & \mathsf{exports}\ \mathit{export}^* \\ \mathit{instantiatearg} & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{arg}\ \mathit{sortidx}\} \end{array}$$

### 2.3.7 Aliases

An alias definition copies a definition from some other module, component, or instance into an index space of the current component:

$$\begin{array}{lcl} \mathit{alias} & ::= & \{\mathsf{sort}\ \mathit{sort}, \mathsf{target}\ \mathit{aliastarget}\} \\ \mathit{aliastarget} & ::= & \mathsf{export}\ \mathit{instanceidx}\ \mathit{name} \\ & | & \mathsf{core\_export}\ \mathit{core{:}instanceidx}\ \mathit{name} \\ & | & \mathsf{outer}\ \mathit{u32}\ \mathit{u32} \end{array}$$

### 2.3.8 Canonical definitions

Canonical definitions are the only way to convert between Core WebAssembly functions and component-level shared-nothing functions which produce and consume values of type $valtype$. A *canon lift* definition converts a core WebAssembly function into a component-level function which may be exported or used to satisfy the imports of another component; a *canon lower* definition converts an lifted function (often imported) into a core function.

$$\begin{array}{lll} canon & ::= & \mathsf{lift}\ core{:}funcidx\ canonopt^*\ typeidx \\ & | & \mathsf{lower}\ funcidx\ canonopt^* \\ canonopt & ::= & \mathsf{string\_encoding\_utf8} \\ & | & \mathsf{string\_encoding\_utf16} \\ & | & \mathsf{string\_encoding\_latin1{+}utf16} \\ & | & \mathsf{memory}\ core{:}memidx \\ & | & \mathsf{realloc}\ core{:}funcidx \\ & | & \mathsf{post\_return}\ core{:}funcidx \end{array}$$

### 2.3.9 Start definitions

A start definition specifies a component function which this component would like to see called at instantiation type in order to do some sort of initialization.

$$start \quad ::= \quad \{\mathsf{func}\ funcidx, \mathsf{args}\ valueidx^*\}$$

### 2.3.10 Imports

Since an imported value is described entirely by its type, an actual import definition is effectively the same thing as an import declaration:

$$import \quad ::= \quad importdecl$$

### 2.3.11 Exports

An export definition is simply a name and a reference to another definition to export:

$$export \quad ::= \quad \{\mathsf{name}\ name, \mathsf{def}\ sortidx\}$$

CHAPTER

# THREE

# VALIDATION

## 3.1 Conventions

As in Core WebAssembly, a *validation* stage checks that a component is well-formed, and only valid components may be instantiated.

Similarly to Core WebAssembly, a *type system* over the abstract syntax of a component is used to specify which modules are valid, and the rules governing the validity of a component are given in both prose and formal mathematical notation.

### 3.1.1 Contexts

Validation rules for individual definitions are interpreted within a particular *context*, which contains the information about the surrounding component and environment needed to validae a particular definition. The validation contexts used in the component model contain the types of every definition in every index space currently accessible (including the index spaces of parent components, which may be accessed via outer aliases).

Concretely, a validation context is defined as a record with the following abstract syntax:

$$
\begin{array}{lcll}
\Gamma_c & ::= & \{ & \mathsf{types}\ \mathit{core}{:}\mathit{deftype}_e{}^*, \\
 & & & \mathsf{funcs}\ \mathit{core}{:}\mathit{functype}^*, \\
 & & & \mathsf{modules}\ \mathit{core}{:}\mathit{moduletype}_e{}^*, \\
 & & & \mathsf{instances}\ \mathit{core}{:}\mathit{instancetype}_e{}^*, \\
 & & & \mathsf{tables}\ \mathit{core}{:}\mathit{tabletype}^*, \\
 & & & \mathsf{mems}\ \mathit{core}{:}\mathit{memtype}^*, \\
 & & & \mathsf{globals}\ \mathit{core}{:}\mathit{globaltype}^* \} \\
\Gamma & ::= & \{ & \mathsf{parent}\ \Gamma, \\
 & & & \mathsf{core}\ \Gamma_c, \\
 & & & \mathsf{uvars}\ \mathit{boundedtyvar}^*, \\
 & & & \mathsf{evars}\ (\mathit{boundedtyvar}, \mathit{deftype}_e)^* \\
 & & & \mathsf{rtypes}\ \mathit{resourcetype}_e^*, \\
 & & & \mathsf{types}\ \mathit{deftype}_e{}^*, \\
 & & & \mathsf{components}\ \mathit{componenttype}_e{}^*, \\
 & & & \mathsf{instances}\ \mathit{instancetype}_e{}^{\dagger *}, \\
 & & & \mathsf{funcs}\ \mathit{functype}_e{}^*, \\
 & & & \mathsf{values}\ \mathit{valtype}_e^{?}{}^*, \}
\end{array}
$$

### 3.1.2 Notation

Both the formal and prose notation share a number of constructs:

- When writing a value of the abstract syntax, any component of the abstract syntax which has the form $nonterminal^n$, $nonterminal^*$, $nonterminal^+$, or $nonterminal^?$, we may write $\overline{\ldots_i}^n$ to mean that this position is filled by a series of $n$ abstract values, named $\ldots_1$ to $\ldots_n$.

## 3.2 Types

During validation, the abstract syntax types described above are *elaborated* into types of a different structure, which are easier to work with. Elaborated types are different from the original abstract syntax types in three major aspects:

- They do not contain any indirections through type index spaces: since recursive types are explicitly not permitted by the component model, it is possible to simply inline all such indirections.
- Due to the above, instance and component types do not contain any embedded declarations; the type sharing that necesstated the use of type alias declarations is replaced with explicit binders and type variables.
- Value types have been *despecialised*: the value type constructors tuple, flags, enum, option, union, result, and string have been replaced by equivalent types.

This elaboration also ensures that the type definitions themselves have valid structures, and so may be considered as validation on types.

### 3.2.1 Primitive value types

Any $primvaltype$, $defvaltype$, or $valtype$ elaborates to a a $valtype_e$. The syntax of $valtype_e$ is specified by parts over the next several sections, as it becomes relevant.

$$\begin{array}{lcl} valtype_e & ::= & \mathsf{bool} \\ & | & \mathsf{s8}|\mathsf{u8}|\mathsf{s16}|\mathsf{u16}|\mathsf{s32}|\mathsf{u32}|\mathsf{s64}|\mathsf{u64} \\ & | & \mathsf{float32}|\mathsf{float64} \\ & | & \mathsf{char} \\ & | & \mathsf{list}\ valtype_e \\ & | & \ldots \end{array}$$

Because values are used linearly, values in the context must be associated with information about whether they are alive or dead. This is accomplished by assigning them types from $valtype_e^?$:

$$\begin{array}{lcl} valtype_e^? & ::= & valtype_e \\ & | & {valtype_e}^\dagger \end{array}$$

string

- The primitive value type string elaborates to the $valtype_e$ of list char.

$$\frac{}{\Gamma \vdash \mathsf{string} \rightsquigarrow \mathsf{list}\ \mathsf{char}}$$

#### $\mathit{primvaltype}$ other than string

- Any $\mathit{primvaltype}$ other than string elaborates to the $\mathit{valtype}_e$ of the same name.

$$\frac{\mathit{primvaltype} \neq \mathsf{string}}{\Gamma \vdash \mathit{primvaltype} \leadsto \mathit{primvaltype}}$$

### 3.2.2 Record fields

Any *record_field* elaborates to a $\mathit{record\_field}_e$ with the following abstract syntax:

$$\mathit{record\_field}_e \quad ::= \quad \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \mathit{valtype}_e\}$$

- The type of the record field must elaborate to some $\mathit{valtype}_e$
- Then the record field elaborates to an $\mathit{record\_field}_e$ of the same name with the type $\mathit{valtype}_e$.

$$\frac{\Gamma \vdash \mathit{valtype} \leadsto \mathit{valtype}_e}{\Gamma \vdash \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \mathit{valtype}\} \leadsto \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \mathit{valtype}_e\}}$$

### 3.2.3 Variant cases

Because validation must ensure that a variant case which refines another case has a compatible type, a variant case elaborates to an $\mathit{variant\_case}_e$ in a special context $\mathit{vcctx}$:

$$\begin{array}{lcl} \mathit{vcctx} & ::= & \{\mathsf{ctx}\ \Gamma, \mathsf{cases}\ \mathit{variant\_case}_e{}^*\} \\ \mathit{variant\_case}_e & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \mathit{valtype}_e{}^?, \mathsf{refines}\ \mathit{u32}^?\} \end{array}$$

- If the variant case contains a type, it must elaborate to some $\mathit{valtype}_e$.
- If an index $i$ is present in the refines record of the variant case type, then $\mathit{vcctx}.\mathsf{cases}[i]$ must be present, and:
  - If the variant case does not contain a type, $\mathit{vcctx}.\mathsf{cases}[i]$ must not contain a type.
  - If the variant case contains a type, then $\mathit{vcctx}.\mathsf{cases}[i]$ must also contain an elaborated type, and the elaborated form of the cases' type must be a subtype of that type.
- Then the variant case elaborates to an $\mathit{record\_field}_e$ of the same name, with:
  - If the variant case does not contain a type, then no type.
  - If the variant case does contain a type, then the $\mathit{valtype}_e$ to which it elaborates.
  - If the variant case does not contain a refines index, then no refines name.
  - If the variant case does contain a refines index $i$, then a refines name of $\mathit{vcctx}.\mathsf{cases}[i].\mathsf{name}$.

$$\frac{\begin{array}{c}\forall i, \mathit{vcctx}.\mathsf{ctx} \vdash \mathit{valtype}_i \leadsto \mathit{valtype}_{ei} \\ \forall j, \mathit{vcctx}.\mathsf{cases}[\mathit{u32}_j] = \{\mathsf{name}\ \mathit{name}_j, \mathsf{type}\ \overline{\mathit{valtype}_e{}'_k}, \dots\} \wedge \forall i, \mathit{valtype}_{ei} \preccurlyeq \mathit{valtype}_e{}'_i\end{array}}{\mathit{vcctx} \vdash \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \overline{\mathit{valtype}_i}, \mathsf{refines}\ \overline{\mathit{u32}_j}\} \leadsto \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \overline{\mathit{valtype}_{ei}}, \mathsf{refines}\ \overline{\mathit{name}_j}\}}$$

### 3.2.4 Definition value types

A definition value type elaborates to a $valtype_e$. The syntax of $valtype_e$ is broader than shown earlier:

$$\begin{array}{lcl} valtype_e & ::= & \dots \\ & | & \mathsf{record}\ record\_field_e{}^+ \\ & | & \mathsf{variant}\ variant\_case_e{}^+ \\ & | & \mathsf{own}\ deftype_e \end{array}$$

$\mathsf{ref}\ scope\ deftype_e$

$\mathsf{prim}\ primvaltype$

- The primitive value type $primvaltype$ must elaborate to some $valtype_e$.
- Then the definition value type $\mathsf{prim}\ primvaltype$ elaborates to the the the same $valtype_e$.

$$\frac{\Gamma \vdash primvaltype \rightsquigarrow valtype_e}{\Gamma \vdash \mathsf{prim}\ primvaltype \rightsquigarrow valtype_e}$$

$\mathsf{record}\ record\_field^+$

- Each record field declaration $record\_field_i$ must elaborate to some $record\_field_{ei}$.
- The names of the $record\_field_{ei}$ must all be distinct.
- Then the definition value type $\mathsf{record}\ \overline{record\_field_i}^n$ elaborates to $\mathsf{record}\ \overline{record\_field_{ei}}^n$.

$$\frac{\forall i, \Gamma \vdash record\_field_i \rightsquigarrow record\_field_{ei} \quad \forall ij, record\_field_{ei}.\mathsf{name} = record\_field_{ej}.\mathsf{name} \Rightarrow i = j}{\Gamma \vdash \mathsf{record}\ \overline{record\_field_i}^n \rightsquigarrow \mathsf{record}\ \overline{record\_field_{ei}}^n}$$

$\mathsf{variant}\ variant\_case^+$

- Each variant case declaration $variant\_case_i$ must elaborate to some $variant\_case_{ei}$, in a variant-case context $vcctx_i$ where:
  - $vcctx_i.\mathsf{ctx} = \Gamma$
  - $vcctx_i.\mathsf{cases} = variant\_case_{e1}, \dots, variant\_case_{ei-1}$
- The names of the $variant\_case_{ei}$ must all be distinct.
- Then the definition value type $\mathsf{variant}\ \overline{variant\_case_i}^n$ elaborates to $\mathsf{variant}\ \overline{variant\_case_{ei}}^n$.

$$\frac{\forall i, \{\mathsf{ctx}\ \Gamma, \mathsf{cases}\ variant\_case_{e1}, \dots, variant\_case_{ei-1}\} \vdash variant\_case_i \rightsquigarrow variant\_case_{ei} \quad \forall i, j\, variant\_case_{ei}.\mathsf{name} = variant\_case_{ej}.\mathsf{name} \Rightarrow i = j}{\Gamma \vdash \mathsf{variant}\ \overline{variant\_case_i}^n \rightsquigarrow \mathsf{variant}\ \overline{variant\_case_{ei}}^n}$$

list $valtype$

- The list element type $valtype$ must elaborate to some $valtype_e$.
- Then the definition value type list $valtype$ elaborates to list $valtype_e$.

$$\frac{\Gamma \vdash valtype \rightsquigarrow valtype_e}{\Gamma \vdash \mathsf{list}\ valtype \rightsquigarrow \mathsf{list}\ valtype_e}$$

tuple $\overline{valtype_i}$

- Each tuple element type $valtype_i$ must elaborate to some $valtype_{ei}$.
- Then the definition value type tuple $\overline{valtype_i}$ elaborates to record $\overline{\{\mathsf{name}\ ''i'', \mathsf{type}\ valtype_{ei}\}}$.

$$\frac{\forall i, \Gamma \vdash valtype_i \rightsquigarrow valtype_{ei}}{\Gamma \vdash \mathsf{tuple}\ \overline{valtype_i} \rightsquigarrow \mathsf{record}\ \overline{\{\mathsf{name}\ ''i'', \mathsf{type}\ valtype_{ei}\}}}$$

flags $\overline{name_i}$

- The definition value type flags $\overline{name_i}$ elaborates to record $\overline{\{\mathsf{name}\ name_i, \mathsf{type}\ \mathsf{bool}\}}$

$$\frac{}{\Gamma \vdash \mathsf{flags}\ \overline{name_i} \rightsquigarrow \mathsf{record}\ \overline{\{\mathsf{name}\ name_i, \mathsf{type}\ \mathsf{bool}\}}}$$

enum $\overline{name_i}$

- The definition value type enum $\overline{name_i}$ elaborates to variant $\overline{\{\mathsf{name}\ name_i\}}$.

$$\frac{}{\Gamma \vdash \mathsf{enum}\ \overline{name_i} \rightsquigarrow \mathsf{variant}\ \overline{\{\mathsf{name}\ name_i\}}}$$

option $valtype$

- The type contained in the option $valtype$ must elaborate to some $valtype_e$.
- Then the definition value type option $valtype$ elaborates to variant $\{\mathsf{name}\ ''none''\}\ \{\mathsf{name}\ ''some'', \mathsf{type}\ valtype_e\}$.

$$\frac{\Gamma \vdash valtype \rightsquigarrow valtype_e}{\Gamma \vdash \mathsf{option}\ valtype \rightsquigarrow \mathsf{variant}\ \{\mathsf{name}\ ''none''\}\ \{\mathsf{name}\ ''some'', \mathsf{type}\ valtype_e\}}$$

union $\overline{valtype_i}$

- Each value type *valtype_i* must elaborate to some $valtype_{ei}$.
- Then the definition value type union $\overline{valtype_i}$ elaborates to variant $\overline{\{\mathsf{name}\ ''i'', \mathsf{type}\ valtype_{ei}\}}$.

$$\frac{\forall i, \Gamma \vdash valtype_i \rightsquigarrow valtype_{ei}}{\Gamma \vdash \mathsf{union}\ \overline{valtype_i} \rightsquigarrow \mathsf{variant}\ \overline{\{\mathsf{name}\ ''i'', \mathsf{type}\ valtype_{ei}\}}}$$

$\mathsf{result}\ \overline{valtype_i}\ \overline{valtype'_j}$

- Each value type $valtype_i$ must elaborate to some $valtype_{e\,i}$.
- Each value type $valtype'_j$ must elaborate to some $valtype_{e\,j}'$.
- Then the definition value type $\mathsf{result}\ \overline{valtype_i}\ \overline{valtype'_j}$ elaborates to $\mathsf{variant}\ \{\mathsf{name}\ "ok", \mathsf{type}\ \overline{valtype_{e\,i}}\}\ \{\mathsf{name}\ "error", \mathsf{type}\ \overline{valtype_e{'}_j}\}$.

$$\frac{\forall i, \Gamma \vdash valtype_i \rightsquigarrow valtype_{e\,i} \qquad \forall j, \Gamma \vdash valtype'_j \rightsquigarrow valtype_e{'}_j}{\Gamma \vdash \mathsf{result}\ \overline{valtype_i}\ \overline{valtype'_j} \rightsquigarrow \mathsf{variant}\ \{\mathsf{name}\ "ok", \mathsf{type}\ \overline{valtype_{e\,i}}\}\ \{\mathsf{name}\ "error", \mathsf{type}\ \overline{valtype_e{'}_j}\}}$$

$\mathsf{own}\ typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context, and must be a subtype of resource.
- Then the definition value type $\mathsf{own}\ typeidx$ elaborates to $\mathsf{own}\ valtype_e$.

$$\frac{\Gamma.\mathsf{types}[typeidx] = valtype_e \qquad valtype_e \preccurlyeq \mathsf{resource}}{\Gamma \vdash \mathsf{own}\ typeidx \rightsquigarrow \mathsf{own}\ valtype_e}$$

$\mathsf{borrow}\ typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context, and must be a subtype of resource.
- Then the definition value type $\mathsf{borrow}\ typeidx$ elaborates to $\mathsf{ref}\ \mathsf{call}\ valtype_e$.

$$\frac{\Gamma.\mathsf{types}[typeidx] = valtype_e \qquad valtype_e \preccurlyeq \mathsf{resource}}{\Gamma \vdash \mathsf{borrow}\ typeidx \rightsquigarrow \mathsf{ref}\ \mathsf{call}\ valtype_e}$$

### 3.2.5 Value types

$primvaltype$

- A value type of the form $primvaltype$ must be a $primvaltype$ which elaborates to some $valtype_e$.
- Then the value type elaborates to the same $valtype_e$.

$$\frac{\Gamma \vdash primvaltype \rightsquigarrow valtype_e}{\Gamma \vdash primvaltype \rightsquigarrow valtype_e}$$

$typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context.
- Then the value type $typeidx$ elaborates to $\Gamma.\mathsf{types}[typeidx]$.

$$\frac{}{\Gamma \vdash typeidx \rightsquigarrow \Gamma.\mathsf{types}[typeidx]}$$

### 3.2.6 Value type well-formedness

Since certain value types cannot appear in certain places (most notably, ref call may not appear anywhere save a function parameter type), we define a family of well-formedness judgments. Each context which may require a $valtype_e$ uses one of these well-formedness judgments to ensure that it is of correct form.

Note that the variable scoping constraints should already be enforced by earlier elaboration stages, which never generate free type variables, but they are included here for completeness.

We define a formal syntax of the position parameters which may be used:

$$\begin{array}{lcl} \rho & ::= & \epsilon \\ & | & \mathsf{p} \end{array}$$

bool

- In any context and any position, bool is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{bool}}$$

s8

- In any context and any position, s8 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{s8}}$$

u8

- In any context and any position, u8 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{u8}}$$

s16

- In any context and any position, s16 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{s16}}$$

### u16

- In any context and any position, u16 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{u16}}$$

### s32

- In any context and any position, s32 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{s32}}$$

### u32

- In any context and any position, u32 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{u32}}$$

### s64

- In any context and any position, s64 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{s64}}$$

### u64

- In any context and any position, u64 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{u64}}$$

### float32

- In any context and any position, float32 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{float32}}$$

### float64

- In any context and any position, float64 is well-formed.

$$\frac{}{\Gamma \vdash_\rho \mathsf{float64}}$$

char

- In any context and any position, char is well-formed.

$$\overline{\Gamma \vdash_\rho \text{char}}$$

list $valtype_e$

- In any context and any position, if $valtype_e$ is well-formed, then list $valtype_e$ is well-formed.

$$\frac{\Gamma \vdash_\rho valtype_e}{\Gamma \vdash_\rho \text{list } valtype_e}$$

record $record\_field_e{}^*$

- In any context and any position, if each $name_i$ is distinct, and each $valtype_{ei}$ is well-formed, then record $\overline{\text{name } name_i, \text{type } valtype_{ei}}$ is well-formed.

$$\frac{\forall ij, i \neq j \Rightarrow name_i \neq name_j \qquad \forall i, \Gamma \vdash_\rho valtype_{ei}}{\Gamma \vdash_\rho \text{record } \overline{\text{name } name_i, \text{type } valtype_{ei}}}$$

variant $variant\_case_e{}^+$

- In any context and any position, if each $name_i$ is distinct, and each $valtype_{ei}$ is well-formed, and each $u32^?_i$ does not refer to a non-existent or self-referential case, then variant $\overline{\text{name } name_i, \text{type } valtype_e{}^?{}_i, \text{refines}}$ is well-formed.

$$\frac{\forall ij, i \neq j \Rightarrow name_i \neq name_j \qquad \forall i, \forall valtype_e, valtype_e{}^?{}_i = valtype_e \Rightarrow \Gamma \vdash_\rho valtype_e}{\Gamma \vdash_\rho \text{variant } \overline{\text{name } name_i, \text{type } valtype_e{}^?{}_i, \text{refines } u32^?_i}^n}$$

own resource $rtidx$

- The resource type $\Gamma.\text{rtypes}[rtidx]$ must be defined in the context.
- Then in any position, own resource $rtidx$ is well-formed.

$$\frac{\exists resourcetype_e, \Gamma.\text{rtypes}[rtidx] = resourcetype_e}{\Gamma \vdash_\rho \text{own resource } rtidx}$$

own $\alpha$

- The type variable $\alpha$ must be defined in the context with a bound of sub resource.
- Then in any position, own $\alpha$ is well-formed.

$$\frac{(\alpha : \text{sub resource}) \in \Gamma.\text{uvars} \vee \exists resourcetype_e, (\alpha : \text{sub resource}, resourcetype_e) \in \Gamma.\text{evars}}{\Gamma \vdash_\rho \text{own } \alpha}$$

$\mathsf{ref}\ \mathit{deftype}_e$

- The value type $\mathsf{own}\ \mathit{deftype}_e$ must be well-formed in the context in parameter position.
- Then $\mathsf{ref}\ \mathsf{call}\ \mathit{deftype}_e$ is well-formed in parameter position.

$$\frac{\Gamma \vdash_{\mathsf{p}} \mathsf{own}\ \mathit{deftype}_e}{\Gamma \vdash_{\mathsf{p}} \mathsf{ref}\ \mathsf{call}\ \mathit{deftype}_e}$$

### 3.2.7 Result types

Because a *resulttype* may appear in a parameter position or in a return position, its elaboration is parametrized by which position it appears in.

Any *resulttype* elaborates to a $\mathit{resulttype}_e$ with the following abstract syntax:

$$\begin{array}{lcl} \mathit{resulttype}_e & ::= & \mathit{valtype}_e \\ & | & \{\mathsf{name}\ \mathit{name}, \mathsf{type}\ \mathit{valtype}_e\}^* \end{array}$$

$\mathit{valtype}$

- $\mathit{valtype}$ must elaborate to some $\mathit{valtype}_e$
- $\mathit{valtype}_e$ must be valid in the appropriate position.
- Then the result type $\mathit{valtype}$ elaborates to $\mathit{valtype}_e$.

$$\frac{\Gamma \vdash \mathit{valtype} \rightsquigarrow \mathit{valtype}_e \qquad \Gamma \vdash_\rho \mathit{valtype}_e}{\Gamma \vdash_\rho \mathit{valtype} \rightsquigarrow \mathit{valtype}_e}$$

$\overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_i\}}$

- Each $\mathit{valtype}_i$ must elaborate to some $\mathit{valtype}_{ei}$.
- Then the result type $\overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_i\}}$ elaborates to $\overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_{ei}\}}$.

$$\frac{\forall i, \Gamma \vdash \mathit{valtype}_i \rightsquigarrow \mathit{valtype}_{ei} \qquad \forall i, \Gamma \vdash_\rho \mathit{valtype}_{ei}}{\Gamma \vdash_\rho \overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_i\}} \rightsquigarrow \overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_{ei}\}}}$$

### 3.2.8 Function types

Any *functype* elaborates to a $\mathit{functype}_e$ with the following abstract syntax:

$$\mathit{functype}_e ::= \mathit{resulttype}_e \to \mathit{resulttype}_e$$

$resulttype_1 \to resulttype_2$

- $resulttype_1$ must elaborate in parameter position to some $resulttype_{e1}$.
- $resulttype_2$ must elaborate to some $resulttype_{e2}$.
- Then the function type $resulttype_1 \to resulttype_2$ elaborates to $resulttype_{e1} \to resulttype_{e2}$.

$$\frac{\Gamma \vdash_{\mathsf{p}} resulttype_1 \leadsto resulttype_{e1} \qquad \Gamma \vdash resulttype_2 \leadsto resulttype_{e2}}{\Gamma \vdash resulttype_1 \to resulttype_2 \leadsto resulttype_{e1} \to resulttype_{e2}}$$

### 3.2.9 Type bound

A type bound elaborates to a $typebound_e$ with the following abstract syntax:

$$\begin{array}{lcl} typebound_e & ::= & \mathsf{eq}\ deftype_e \\ & | & \mathsf{sub\ resource} \end{array}$$

$\mathsf{EQ}\ typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context.
- Then the type bound $\mathsf{EQ}\ typeidx$ elaborates to $\mathsf{eq}\ \Gamma.\mathsf{types}[typeidx]$.

$$\frac{}{\Gamma \vdash \mathsf{EQ}\ typeidx \leadsto \mathsf{eq}\ \Gamma.\mathsf{types}[typeidx]}$$

$\mathsf{SUB\ resource}$

- The type bound $\mathsf{SUB\ resource}$ elaborates to $\mathsf{sub\ resource}$.

$$\frac{}{\Gamma \vdash \mathsf{SUB\ resource} \leadsto \mathsf{sub\ resource}}$$

### 3.2.10 Instance types

An elaborated instance type is nothing more than a list of its exports behind existential quantifiers for exported types:

$$\begin{array}{lcl} instancetype_e & ::= & \exists boundedtyvar^*.externdecl_e^* \\ boundedtyvar & ::= & (\alpha : typebound_e) \\ externdecl_e & ::= & \{\mathsf{name}\ name, \mathsf{desc}\ externdesc_e\} \\ externdesc_e & ::= & \mathsf{core\_module}\ core{:}moduletype_e \\ & | & \mathsf{func}\ functype_e \\ & | & \mathsf{value}\ valtype_e \\ & | & \mathsf{type}\ deftype_e \\ & | & \mathsf{instance}\ instancetype_e \\ & | & \mathsf{component}\ componenttype_e \end{array}$$

Because instance value exports must be used linearly in the context, instances in the contexts are, by analogy with $valtype_e^?$, assigned types from $instancetype_e^?$.

$$\begin{array}{lcl} instancetype_e^? & ::= & \exists boundedtyvar^*.externdecl_e^{?*} \\ externdecl_e^? & ::= & externdecl_e \\ & | & externdecl_e^\dagger \end{array}$$

### Notational conventions

- We write $instancetype_e \oplus instancetype_e{}'$ to mean the instance type formed by the concatonation of the export declarations of $instancetype_e$ and $instancetype_e{}'$.
- We write $\bigoplus_i instancetype_{ei}$ to mean the instance type formed by $instancetype_{e1} \oplus \cdots \oplus instancetype_{en}$.

### Finalize: $\langle\!\langle instancetype_e \rangle\!\rangle$

Finalizing an instance type eliminates unnecessary type variables with equality constraints, ensures that all type variables are well-scoped, and that all quantified types are exported.

- Each type variable existentially quantified in $instancetype_e$ must either be exported or have an equality type bound.
- Then the finalized version of $instancetype_e$ is that type, with each type variable which is not exported replaced by the type that it is equality-bounded to.

$$\frac{\begin{array}{c}\text{defined}(\alpha) = \begin{cases} deftype_e & \text{if } \exists i, \alpha_i = \alpha \wedge typebound_{ei} = \mathsf{eq}\ deftype_e \\ \bot & \text{otherwise}\end{cases} \\ \text{externed}(\alpha) = \begin{cases} \top & \text{if } \exists i, \alpha_i = \alpha \wedge \exists name, \{\mathsf{name}\ name, \mathsf{desc}\ \mathsf{type}\ \alpha\} \in \overline{externdecl_{ej}} \\ \bot & \text{otherwise}\end{cases} \\ \forall i, \text{defined}(\alpha_i) \vee \text{externed}(\alpha_i) \\ \delta(\alpha) = \begin{cases} \text{defined}(\alpha) & \text{if } \neg\text{externed}(\alpha) \\ \bot & \text{otherwise}\end{cases} \\ \bar{i} = \{i \mid \text{externed}(\alpha_i)\}\end{array}}{\begin{array}{c}\langle\!\langle \exists \overline{(\alpha_i : typebound_{ei})}.\overline{externdecl'_{ej}} \rangle\!\rangle \\ = \delta(\exists \overline{(\alpha_i : typebound_{ei})}^{i \in \bar{i}}.\overline{externdecl'_{ej}})\end{array}}$$

### $\overline{instancedecl_i}$

- $instancedecl_1$ must elaborate to some $instancetype_{e1}$ in the context $\{\mathsf{parent}\ \Gamma\}$.
- For each $i > 1$, the instance declarator $instancedecl_i$ must elaborate in the context produced by the elaboration of $instancedecl_{i-1}$ to some $instancetype_{ei}$.
- Then the instance type $\overline{instancedecl_i}$ elaborates to $\bigoplus_i instancetype_{ei}$.

$$\frac{\begin{array}{c}\Gamma_0 = \{\mathsf{parent}\ \Gamma\} \\ \forall i, \Gamma_{i-1} \vdash instancedecl_i \rightsquigarrow instancetype_{ei} \dashv \Gamma_i\end{array}}{\Gamma \vdash \overline{instancedecl_i} \rightsquigarrow \langle\!\langle \bigoplus_i instancetype_{ei} \rangle\!\rangle}$$

## 3.2.11 Instance declarators

Each instance declarator elaborates to a (partial) $instancetype_e$.

alias $alias$

- The $alias$.sort must be type.
- The $alias$.target must be of the form outer $u32_o$ $u32_i$.
- The type $\Gamma$.parent[$u32_o$].types[$u32_i$] must be defined in the context.
- The type $\Gamma$.parent[$u32_o$].types[$u32_i$] must not be of the form resource $i$ for any $i$.
- Then the instance declarator alias $alias$ elaborates to the empty list of exports, and sets types in the context to the original $\Gamma$.types followed by $\Gamma$.parent[$u32_o$].types[$u32_i$].

$$\frac{alias.\mathsf{sort} = \mathsf{type} \quad alias.\mathsf{target} = \mathsf{outer}\ u32_o\ u32_i \quad \forall i, \Gamma.\mathsf{parent}[u32_o].\mathsf{types}[u32_i] \neq \mathsf{resource}\ i}{\Gamma \vdash \mathsf{alias}\ alias \rightsquigarrow \exists\varnothing.\varnothing \dashv \Gamma \oplus \{\mathsf{types}\ \Gamma.\mathsf{parent}[u32_o].\mathsf{types}[u32_i]\}}$$

core_type $core{:}type$

- The core type definition $core{:}type$ must elaborate to some elaborated core type $core{:}deftype_e$.
- Then the instance declarator core_type $core{:}type$ elaborates to the empty list of exports, and sets core.types in the context to the original $\Gamma$.core.types followed by the $core{:}deftype_e$.

$$\frac{\Gamma \vdash core{:}type \rightsquigarrow core{:}deftype_e}{\Gamma \vdash \mathsf{core\_type}\ core{:}type \rightsquigarrow \exists\varnothing.\varnothing \dashv \Gamma \oplus \{\mathsf{core\ types}\ core{:}deftype_e\}}$$

type $deftype$

- The definition type $deftype$ must elaborate to some elaborated definition type $deftype_e$.
- Let $\alpha$ be a fresh type variable.
- Then the instance declarator type $deftype$ elaborates to the empty list of exports behind an existential quantifier associating $\alpha$ with $deftype_e$, and sets types in the context to the original $\Gamma$.types followed by the $\alpha$.

$$\frac{\Gamma \vdash deftype \rightsquigarrow deftype_e}{\Gamma \vdash \mathsf{type}\ deftype \rightsquigarrow \exists(\alpha : \mathsf{eq}\ deftype_e).\varnothing \dashv \Gamma \oplus \{\mathsf{evars}\ (\alpha : \mathsf{eq}\ deftype_e, deftype_e), \mathsf{types}\ \alpha\}}$$

- Notice that because this type variable is equality-bounded and not exported, it will always be inlined by $\langle\langle instancetype_e \rangle\rangle$.

export $exportdecl$

- The extern descriptor $exportdecl$.desc must elaborate to some $\forall boundedtyvar^*.externdesc_e$.
- Then the instance declarator export $exportdecl$ elaborates to the singleton list of exports containing {name $exportdecl$.name, desc $externdesc_e$} and quantified by $boundedtyvar$, and adds an appropriately typed entry to the context.

$$\frac{\Gamma \vdash exportdecl.\mathsf{desc} \rightsquigarrow \forall boundedtyvar^*.externdesc_e}{\Gamma \vdash exportdecl \rightsquigarrow \exists boundedtyvar^*.\{\mathsf{name}\ exportdecl.\mathsf{name}, \mathsf{desc}\ externdesc_e\} \dashv \Gamma \oplus \{\mathsf{uvars}\ boundedtyvar^*, externdesc_e\}}$$

### 3.2.12 Extern descriptors

An extern descriptor elaborates to a quantified $externdesc_e$ with the following abstract syntax:

type $typebound$

- The $typebound$ must elaborate to some $typebound_e$.
- Let $\alpha$ be a fresh type variable.
- Then the import descriptor type $typebound$ elaborates to $\forall(\alpha : typebound_e).\mathsf{type}\ \alpha$.

$$\frac{\Gamma \vdash typebound \rightsquigarrow typebound_e}{\Gamma \vdash \mathsf{type}\ typebound \rightsquigarrow \forall(\alpha : typebound_e).\mathsf{type}\ \alpha}$$

core_module $core{:}typeidx$

- The type $\Gamma.\mathsf{core}.\mathsf{types}[core{:}typeidx]$ must be defined in the context, and must be of the form $core{:}moduletype_e$.
- Then the import descriptor core_module $core{:}typeidx$ elaborates to $\forall\varnothing.\mathsf{core\_module}\ core{:}moduletype_e$.

$$\frac{\Gamma.\mathsf{core}.\mathsf{types}[core{:}typeidx] = core{:}moduletype_e}{\Gamma \vdash \forall\varnothing.\mathsf{core\_module}\ core{:}typeidx \rightsquigarrow \mathsf{core\_module}\ core{:}moduletype_e}$$

func $typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context, and must be of the form $functype_e$.
- Then the import descriptor func $typeidx$ elaborates to $\forall\varnothing.\mathsf{func}\ functype_e$

$$\frac{\Gamma.\mathsf{types}[typeidx] = functype_e}{\Gamma \vdash \mathsf{func}\ typeidx \rightsquigarrow \forall\varnothing.\mathsf{func}\ functype_e}$$

value $typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context, and must be of the form to some $valtype_e$.
- $valtype_e$ must be well-formed.
- Then the import descriptor value $typebound$ elaborates to $\forall\varnothing.\mathsf{value}\ valtype_e$

$$\frac{\Gamma.\mathsf{types}[typeidx] = valtype_e \qquad \Gamma \vdash valtype_e}{\Gamma \vdash \mathsf{value}\ typeidx \rightsquigarrow \mathsf{value}\ valtype_e}$$

instance $typeidx$

- The type $\Gamma.\mathsf{types}[typeidx]$ must be defined in the context, and must be of the form $\exists boundedtyvar^*.externdecl_e^*$.
- Then the import descriptor instance $typeidx$ elaborates to $\forall boundedtyvar^*.\mathsf{instance}\ \exists\varnothing.externdecl_e^*$

$$\frac{\Gamma.\mathsf{types}[typeidx] = \exists boundedtyvar^*.externdecl_e^*}{\Gamma \vdash \mathsf{instance}\ typeidx \rightsquigarrow \forall boundedtyvar^*.\mathsf{instance}\ \exists\varnothing.externdecl_e^*}$$

$\mathsf{component}\ \mathit{typeidx}$

- The type $\Gamma.\mathsf{types}[\mathit{typeidx}]$ must be defined in the context, and must be of the form $\mathit{componenttype}_e$.
- Then the import descriptor $\mathsf{component}\ \mathit{typeidx}$ elaborates to $\forall\varnothing.\mathsf{component}\ \mathit{componenttype}_e$

$$\frac{\Gamma.\mathsf{types}[\mathit{typeidx}] = \mathit{componenttype}_e}{\Gamma \vdash \mathsf{component}\ \mathit{typeidx} \rightsquigarrow \forall\varnothing.\mathsf{component}\ \mathit{componenttype}_e}$$

### 3.2.13 Component types

In a similar manner to instance types above, component types change significantly upon elaboration: an elaborated component type is described as a mapping from a quantified list of imports to the type of the instance that it will produce upon instantiation:

$$\mathit{componenttype}_e \quad ::= \quad \forall \mathit{boundedtyvar}^*.\mathit{externdecl}_e^* \rightarrow \mathit{instancetype}_e$$

#### Notational conventions

- Much like with instance types above, we write $\mathit{componenttype}_e \oplus \mathit{componenttype}_e{}'$ to mean the combination of two component types; in this case, the component type whose imports are the concatenation of the import lists of $\mathit{componenttype}_e$ and $\mathit{componenttype}_e{}'$ and whose instantiation result (instance) type is the result of applying $\oplus$ to the instantiation result (instance) types of $\mathit{componenttype}_e$ and $\mathit{componenttype}_e{}'$.

#### Finalize: $\langle\!\langle \mathit{componenttype}_e \rangle\!\rangle$

As with instance types above, finalizing a component type eliminates unnecessary type variables with equality constraints, ensures that all type variables are well-scoped, and that all quantified types are imported or exported.

- Each type variable universally quantified in $\mathit{componenttype}_e$ must either be imported (either directly or as a type export of an imported instance) or have an equality type bound.
- Each type variable existentially quantified in $\mathit{componenttype}_e$ must either be exported or have an equality type bound.
- Each type variable existentially quantified in $\mathit{componenttype}_e$ that is exported must not be present in the type of any import.
- Then the finalized version of $\mathit{componenttype}_e$ is that type, with each type variable which is not imported or exported replaced by the type that it is equality-bounded to.

$$\text{defined}(\alpha) = \begin{cases} \mathit{deftype}_e & \text{if } \exists i, \alpha_i = \alpha \wedge \mathit{typebound}_{ei}^{\alpha} = \mathsf{eq}\ \mathit{deftype}_e \\ \mathit{deftype}_e & \text{if } \exists k, \beta_k = \alpha \wedge \mathit{typebound}_{ek}^{\beta} = \mathsf{eq}\ \mathit{deftype}_e \\ \bot & \text{otherwise} \end{cases}$$

$$\text{externed}(\alpha) = \begin{cases} \top & \text{if } \exists i, \alpha_i = \alpha \wedge \exists \mathit{name}, \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathsf{type}\ \alpha\} \in \overline{\mathit{externdecl}_{ej}} \\ \top & \text{if } \exists j, \mathit{externdecl}_{ej} = \exists \overline{\alpha''}.\overline{\mathit{externdecl}''_e} \wedge \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathsf{type}\ \alpha\} \in \overline{\mathit{externdecl}''_e} \\ \top & \text{if } \exists i, \beta_k = \alpha \wedge \exists \mathit{name}, \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathsf{type}\ \alpha\} \in \overline{\mathit{externdecl}'_{ek}} \\ \bot & \text{otherwise} \end{cases}$$

$$\forall i, \text{defined}(\alpha_i) \vee \text{externed}(\alpha_i)$$
$$\forall k, \text{defined}(\beta_k) \vee \text{externed}(\beta_k)$$
$$\forall k, \text{externed}(\beta_k) \Rightarrow \beta_k \notin \text{free\_tyvars}(\overline{\mathit{externdecl}_{ej}})$$
$$\delta(\alpha) = \begin{cases} \text{defined}(\alpha) & \text{if } \neg\text{externed}(\alpha) \\ \bot & \text{otherwise} \end{cases}$$
$$\bar{i} = \{i \mid \text{externed}(\alpha_i)\}$$
$$\bar{k} = \{k \mid \text{externed}(\beta_k)\}$$

$$\overline{\phantom{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}$$

$$\langle\!\langle \forall \overline{(\alpha_i : \mathit{typebound}_{ei}^{\alpha})}.\overline{\mathit{externdecl}_{ej}} \rightarrow \exists \overline{(\beta_k : \mathit{typebound}_{ek}^{\beta})}.\overline{\mathit{externdecl}'_{el}} \rangle\!\rangle$$
$$= \delta(\forall \overline{(\alpha_i : \mathit{typebound}_{ei}^{\alpha})}^{i \in \bar{i}}.\overline{\mathit{externdecl}_{ej}} \rightarrow \exists \overline{(\beta_k : \mathit{typebound}_{ek}^{\beta})}^{k \in \bar{k}}.\overline{\mathit{externdecl}'_{el}})$$

$\overline{\mathit{componentdecl}_i}$

- $\mathit{componentdecl}_1$ must elaborate to some $\mathit{componenttype}_{e1}$ in the context $\{\mathsf{parent}\ \Gamma\}$.
- For each $i > 1$, the component declarator $\mathit{componentdecl}_i$ must elaborate in the context produced by the elaboration of $\mathit{componentdecl}_{i-1}$ to some $\mathit{componenttype}_{ei}$.
- Then the component type $\overline{\mathit{componentdecl}_i}$ elaborates to the type produced by finalizing $\bigoplus_i \mathit{componenttype}_{ei}$.

$$\frac{\Gamma_0 = \{\mathsf{parent}\ \Gamma\} \qquad \forall i, \Gamma_{i-1} \vdash \mathit{componentdecl}_i \rightsquigarrow \mathit{componenttype}_{ei} \dashv \Gamma_i}{\Gamma \vdash \overline{\mathit{componentdecl}_i} \rightsquigarrow \langle\!\langle \bigoplus_i \mathit{componenttype}_{ei} \rangle\!\rangle}$$

## 3.2.14 Component declarators

Each component declarator elaborates to a (partial) $\mathit{componenttype}_e$.

$\mathit{instancedecl}$

- The instance declarator $\mathit{instancedecl}$ must elaborate to some instance type $\mathit{instancetype}_e$ (and may affect the context).
- Then the component declarator $\mathit{instancedecl}$ elaborates to the component type $\forall\varnothing.\varnothing \rightarrow \mathit{instancetype}_e$ and alters the context in the same way.

$$\frac{\Gamma \vdash \mathit{instancedecl} \rightsquigarrow \mathit{instancetype}_e \dashv \Gamma'}{\Gamma \vdash \mathit{instancedecl} \rightsquigarrow \forall\varnothing.\varnothing \rightarrow \mathit{instancetype}_e \dashv \Gamma'}$$

*importdecl*

- The extern descriptor $importdecl.\mathsf{desc}$ must elaborate to some $\forall boundedtyvar^*.externdesc_e$.
- Then the component declarator $importdecl$ elaborates to the component type with no results, the same quantifiers, and a singleton list of imports containing $\{\mathsf{name}\ importdecl.\mathsf{name}, \mathsf{desc}\ externdesc_e\}$, and updates the context with $externdesc_e$.

$$\frac{\Gamma \vdash importdecl.\mathsf{desc} \rightsquigarrow \forall boundedtyvar^*.externdesc_e}{\begin{array}{l}\Gamma \vdash importdecl \\ \quad \rightsquigarrow \forall boundedtyvar^*.\{\mathsf{name}\ importdecl.\mathsf{name}, \mathsf{desc}\ externdesc_e\} \to \varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{uvars}\ boundedtyvar^*, externdesc_e\}\end{array}}$$

### 3.2.15 Definition types

A *deftype* elaborates to a $deftype_e$ with the following abstract syntax:

$$\begin{array}{lcl} deftype_e & ::= & \alpha \\ & | & \mathsf{resource}\ rtidx \\ & | & valtype_e \\ & | & functype_e \\ & | & componenttype_e \\ & | & instancetype_e \end{array}$$

In general, a *deftype* of the form *resourcetype* does not elaborate to any $deftype_e$; however, the component type declarator generates a new context entry for the resource in question and produces an appropriate resource type.

*defvaltype*

- The definition value type *defvaltype* must elaborate to some $valtype_e$.
- Then the definition type *defvaltype* elaborates to $valtype_e$.

$$\frac{\Gamma \vdash defvaltype \rightsquigarrow valtype_e}{\Gamma \vdash defvaltype \rightsquigarrow valtype_e}$$

*functype*

- The function type *functype* must elaborate to some $functype_e$.
- Then the definition type *functype* elaborates to $functype_e$.

$$\frac{\Gamma \vdash functype \rightsquigarrow functype_e}{\Gamma \vdash functype \rightsquigarrow functype_e}$$

*componenttype*

- The component type $componenttype$ must elaborate to some $componenttype_e$.
- Then the definition type $componenttype$ elaborates to $componenttype_e$.

$$\frac{\Gamma \vdash componenttype \rightsquigarrow componenttype_e}{\Gamma \vdash componenttype \rightsquigarrow componenttype_e}$$

*instancetype*

- The instance type $instancetype$ must elaborate to some $instancetype_e$.
- Then the definition type $instancetype$ elaborates to $instancetype_e$.

$$\frac{\Gamma \vdash instancetype \rightsquigarrow instancetype_e}{\Gamma \vdash instancetype \rightsquigarrow instancetype_e}$$

### 3.2.16 Core instance types

Although there are no core instance types present at the surface level, it is useful to define the abstract syntax of (elaborated) core instance types, as they will be needed to characterise the results of instantiationg core modules. As with a component instance type, an (elaborated) core instance type is nothing more than a list of its exports:

$$core{:}instancetype_e \quad ::= \quad core{:}exportdecl^*$$

#### Notational conventions

- We write $core{:}instancetype_e \oplus core{:}instancetype_e{}'$ to mean the instance type formed by the concationation of the export declarations of $core{:}instancetype_e$ and $core{:}instancetype_e{}'$.

### 3.2.17 Core module types

Core module types are defined much like component types above: as a mapping from import descriptions to the type of the instance that will be produced upon instantiating the module:

$$core{:}moduletype_e \quad ::= \quad core{:}importdecl^* \rightarrow core{:}exportdecl^*$$

#### Notational conventions

- Much like with core instance types above, we write $core{:}moduletype_e \oplus core{:}moduletype_e{}'$ to mean the combination of two module types; in this case, the module type whose imports are the concatenation of the import lists of $core{:}moduletype_e$ and $core{:}moduletype_e{}'$ and whose instantiation result (instance) type is the result of applying $\oplus$ to the instantiation result (instance) types of $core{:}moduletype_e$ and $core{:}moduletype_e{}'$.

$\overline{coremoduledecl_i}$

- $coremoduledecl_1$ must elaborate to some $core{:}moduletype_{e1}$ in the context $\{\mathsf{parent}\ \Gamma\}$.
- For each $i > 1$, the core module declarator $coremoduledecl_i$ must elaborate in the context produced by the elaboration of $coremoduledecl_{i-1}$ to some $core{:}moduletype_{ei}$.
- Then the core module type $\overline{coremoduledecl_i}$ to $\bigoplus_i core{:}moduletype_{ei}$.

$$\frac{\Gamma_0 = \{\mathsf{parent}\ \Gamma\} \quad \forall i, \Gamma_{i-1} \vdash coremoduledecl_i \rightsquigarrow core{:}moduletype_{ei} \dashv \Gamma_i}{\Gamma \vdash \overline{coremoduledecl_i} \rightsquigarrow \bigoplus_i core{:}moduletype_{ei}}$$

### 3.2.18 Core module declarators

Each core module declarator elaborates to a (partial) $core{:}moduletype_e$.

$core{:}importdecl$

- The core module declarator $core{:}importdecl$ elaborates to the core module type with no results and a singleton list of imports containing $core{:}importdecl$, and does not modify the context.

$$\frac{}{\Gamma \vdash core{:}importdecl \rightsquigarrow core{:}importdecl \to \varnothing \dashv \Gamma}$$

$core{:}deftype$

- The core definition tyep $core{:}deftype$ must elaborate to some elaborated core definition type $core{:}deftype_e$.
- Then the core module declarator $core{:}deftype$ elaborates to the empty core module type, and sets $\mathsf{core.types}$ in the context to the original $\Gamma.\mathsf{core.types}$ followed by the $deftype_e$.

$$\frac{\Gamma \vdash core{:}deftype \rightsquigarrow core{:}deftype_e}{\Gamma \vdash core{:}deftype \rightsquigarrow \varnothing \to \varnothing \dashv \Gamma \oplus \{\mathsf{core.types}\ core{:}deftype_e\}}$$

$core{:}alias$

- The $core{:}alias.\mathsf{sort}$ must be $\mathsf{type}$.
- The $core{:}alias.$target must be of the form $\mathsf{outer}\ u32_o\ u32_i$.
- The type $\Gamma.\mathsf{parent}[u32_o].\mathsf{core.types}[u32_i]$ must be defined in the context.
- Then the core module declarator $core{:}alias$ elaborates to the empty core module type and sets $\mathsf{core.types}$ in the context to the original $\Gamma.\mathsf{core.types}$ followed by $\Gamma.\mathsf{parent}[u32_o].\mathsf{core.types}[u32_i]$.

$$\frac{core{:}alias.\mathsf{sort} = \mathsf{type} \quad core{:}alias.\mathsf{target} = \mathsf{outer}\ u32_o\ u32_i}{\Gamma \vdash alias \rightsquigarrow \varnothing \to \varnothing \dashv \Gamma \oplus \{\mathsf{core.types}\ \Gamma.\mathsf{parent}[u32_o].\mathsf{core.types}[u32_i]\}}$$

$core{:}exportdecl$

- The core module declarator $core{:}exportdecl$ elaborates to the core module type with no imports and a singleton list of exports containing $core{:}exportdecl$, and does not modify the context.

$$\frac{}{\Gamma \vdash core{:}exportdecl \rightsquigarrow \varnothing \to core{:}exportdecl \dashv \Gamma}$$

### 3.2.19 Core definition types

A core definition type elaborates to a $core{:}deftype_e$ with the following abstract syntax:

$$\begin{array}{lcl} core{:}deftype_e & ::= & core{:}functype \\ & | & core{:}moduletype_e \end{array}$$

$core{:}functype$

- The core definition type $core{:}functype$ elaborates to $core{:}functype$.

$$\frac{}{\Gamma \vdash core{:}functype \rightsquigarrow core{:}functype}$$

$core{:}moduletype$

- The core module type $core{:}moduletype$ must elaborate to some $core{:}moduletype_e$.
- Then the core definition type $core{:}moduletype$ elaborates to $core{:}moduletype_e$.

$$\frac{\Gamma \vdash core{:}moduletype \rightsquigarrow core{:}moduletype_e}{\Gamma \vdash core{:}moduletype \rightsquigarrow core{:}moduletype_e}$$

## 3.3 Subtyping

Subtyping defines when a value of one type may be used when a value of another type is expected.

TODO: This is not complete, pending further discussion, especially in re the special treatment that may or may not be required or specialized value types.

### 3.3.1 Value types

#### Reflexivity

- Any value type is a subtype of itself

$$\frac{}{valtype_e \preccurlyeq valtype_e}$$

### Numeric types

- s8 is a subtype of s16, s32, and s64.
- s16 is a subtype of s32 and s64.
- s32 is a subtype of s64.
- u8 is a subtype of u16, u32, u64, s16, s32, and s64.
- u16 is a subtype of u32, u64, s32, and s64.
- u32 is a subtype of u64 and s64.
- float32 is a subtype of float64.

$$\frac{m > n}{\mathsf{s}n \preccurlyeq \mathsf{s}m}$$

$$\frac{m > n}{\mathsf{u}n \preccurlyeq \mathsf{u}m}$$

$$\frac{m > n}{\mathsf{u}n \preccurlyeq \mathsf{s}m}$$

$$\frac{}{\mathsf{float32} \preccurlyeq \mathsf{float64}}$$

### Records

- A type record $\overline{record\_field_{ei}}$ is a subtype of a type record $\overline{record\_field_{e}'{}_{j}}$ if, for each named field of the latter type, a field with the same name is present in the former, and the type of the field in the former is a subtype of the type of the field in the latter.

Todo: We may need to move despecialization later because of subtyping?

$$\frac{\forall j, \exists i, record\_field_{ei}.\mathsf{name} = record\_field_{e}'{}_{j}.\mathsf{name} \quad \wedge record\_field_{ei}.\mathsf{type} \preccurlyeq record\_field_{ej}.\mathsf{type}}{\mathsf{record}\ \overline{record\_field_{ei}} \preccurlyeq \mathsf{record}\ \overline{record\_field_{e}'{}_{j}}}$$

### Variants

- A type variant $\overline{variant\_case_{ei}}$ is a subtype of a type variant $\overline{variant\_case_{e}'{}_{j}}$ if, or each named case of the former type, either:
  - A case of the same name exists in the latter type, such that the type of the field in the former is a subtype of the type of the field in the latter; or
  - No case of the same name exists in the latter type, and the case in the former contains a refines.

$$\frac{\begin{array}{c}\forall i, (\exists j, variant\_case_{e}'{}_{j}.\mathsf{name} = variant\_case_{ei}.\mathsf{name} \\ \wedge variant\_case_{ei} \preccurlyeq variant\_case_{e}'{}_{j}) \\ \vee(\forall j, variant\_case_{e}'{}_{j}.\mathsf{name} \neq variant\_case_{ei}.\mathsf{name} \\ \wedge \exists name, variant\_case_{ei}.\mathsf{refines} = name)\end{array}}{\mathsf{variant}\ \overline{variant\_case_{ei}} \preccurlyeq \mathsf{variant}\ \overline{variant\_case_{e}'{}_{j}}}$$

#### Lists

- A type $\mathsf{list}\ \mathit{valtype}_e$ is a subtype of a type $\mathsf{list}\ \mathit{valtype}_e{}'$ if $\mathit{valtype}_e$ is a subtype of $\mathit{valtype}_e{}'$

$$\frac{\mathit{valtype}_e \preccurlyeq \mathit{valtype}_e{}'}{\mathsf{list}\ \mathit{valtype}_e \preccurlyeq \mathsf{list}\ \mathit{valtype}_e{}'}$$

### 3.3.2 Result types

- A result type of the form $\mathit{valtype}_e$ is a subtype of a result type of te form $\mathit{valtype}_e{}'$ if $\mathit{valtype}_e$ is a subtype of $\mathit{valtype}_e{}'$.

$$\frac{\mathit{valtype}_e \preccurlyeq \mathit{valtype}_e{}'}{\mathit{valtype}_e \preccurlyeq \mathit{valtype}_e{}'}$$

- A result type of the form $\overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_{e\,i}\}}$ is a subtype of a result type of the form $\overline{\{\mathsf{name}\ \mathit{name}'_j, \mathsf{type}\ \mathit{valtype}_e{}'_j\}}$ when:
  - For each $\mathit{name}'_j$, there is some $i$ such that $\mathit{name}'_j = \mathit{name}_i$ and $\mathit{valtype}_{e\,i} \preccurlyeq \mathit{valtype}_{e\,j}'$.

$$\frac{\forall j, \exists i, \mathit{name}_i = \mathit{name}'_j \wedge \mathit{valtype}_{e\,i} \preccurlyeq \mathit{valtype}_{e\,j}'}{\overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{type}\ \mathit{valtype}_{e\,i}\}} \preccurlyeq \overline{\{\mathsf{name}\ \mathit{name}'_j, \mathsf{type}\ \mathit{valtype}_e{}'_j\}}}$$

### 3.3.3 Function types

- A function type $\mathit{resulttype}_{e1} \rightarrow \mathit{resulttype}_{e2}$ is a subtype of a function $\mathit{resulttype}_e{}'_1 \rightarrow \mathit{resulttype}_e{}'_2$ if $\mathit{resulttype}_e{}'_1 \preccurlyeq \mathit{resulttype}_{e1}$ and $\mathit{resulttype}_{e2} \preccurlyeq \mathit{resulttype}_e{}'_2$.

$$\frac{\mathit{resulttype}_e{}'_1 \preccurlyeq \mathit{resulttype}_{e1} \qquad \mathit{resulttype}_{e2} \preccurlyeq \mathit{resulttype}_e{}'_2}{\mathit{resulttype}_{e1} \rightarrow \mathit{resulttype}_{e2} \preccurlyeq \mathit{resulttype}_e{}'_1 \rightarrow \mathit{resulttype}_e{}'_2}$$

### 3.3.4 Type bound

$\mathsf{eq}\ \mathit{deftype}_e$

- A type bound $\mathsf{eq}\ \mathit{deftype}_e$ is a subtype of $\mathsf{eq}\ \mathit{deftype}_e{}'$ if $\mathit{deftype}_e$ is a subtype of $\mathit{deftype}_e{}'$.

$$\frac{\mathit{deftype}_e \preccurlyeq \mathit{deftype}_e{}'}{\mathsf{eq}\ \mathit{deftype}_e \preccurlyeq \mathsf{eq}\ \mathit{deftype}_e{}'}$$

### 3.3.5 Extern descriptors

$\mathsf{core\_module}\ \mathit{core}{:}\mathit{moduletype}_e$

- A extern descriptor $\mathsf{core\_module}\ \mathit{core}{:}\mathit{moduletype}_e$ is a subtype of $\mathsf{core\_module}\ \mathit{core}{:}\mathit{moduletype}_e{}'$ if $\mathit{core}{:}\mathit{moduletype}_e$ is a subtype of $\mathit{core}{:}\mathit{moduletype}_e{}'$.

$$\frac{\mathit{core}{:}\mathit{moduletype}_e{}' \preccurlyeq \mathit{core}{:}\mathit{moduletype}'}{\mathsf{core\_module}\ \mathit{core}{:}\mathit{moduletype}_e \preccurlyeq \mathsf{core\_module}\ \mathit{core}{:}\mathit{moduletype}_e{}'}$$

func $functype_e$

- An extern descriptor func $functype_e$ is a subtype of func $functype_e{}'$ if $functype_e$ is a subtype of $functype_e{}'$.

$$\frac{functype_e \preccurlyeq functype_e{}'}{\mathsf{func}\ functype_e \preccurlyeq \mathsf{func}\ functype_e{}'}$$

value $valtype_e$

- An extern descriptor value $valtype_e$ is a subtype of value $valtype_e{}'$ if $valtype_e$ is a subtype of $valtype_e{}'$.

$$\frac{valtype_e \preccurlyeq valtype_e{}'}{\mathsf{value}\ valtype_e \preccurlyeq \mathsf{value}\ valtype_e{}'}$$

type $typebound_e$

- An extern descriptor type $typebound_e$ is a subtype of type $typebound_e{}'$ if $typebound_e$ is a subtype of $typebound_e{}'$.

$$\frac{typebound_e \preccurlyeq typebound_e{}'}{\mathsf{type}\ typebound_e \preccurlyeq \mathsf{type}\ typebound_e{}'}$$

instance $instancetype_e$

- An extern descriptor instance $instancetype_e$ is a subtype of instance $instancetype_e{}'$ if $instancetype_e$ is a subtype of $instancetype_e{}'$.

$$\frac{instancetype_e \preccurlyeq instancetype_e{}'}{\mathsf{instance}\ instancetype_e \preccurlyeq \mathsf{instance}\ instancetype_e{}'}$$

component $componenttype_e$

- An extern descriptor component $componenttype_e$ is a subtype of component $componenttype_e{}'$ if $componenttype_e$ is a subtype of $componenttype_e{}'$.

$$\frac{componenttype_e \preccurlyeq componenttype_e{}'}{\mathsf{component}\ componenttype_e \preccurlyeq \mathsf{component}\ componenttype_e{}'}$$

### 3.3.6 Instance types

- An instance type $\overline{externdecl_{ei}}$ is a subtype of an instance type $\overline{externdecl'_{ej}}$ if:
  - For each $j$, there exists some $i$ such that $externdecl_{ei}.\mathsf{name} = externdecl'_{ej}.\mathsf{name}$ and $externdecl_{ei}.\mathsf{desc} \preccurlyeq externdecl'_{ej}.\mathsf{desc}$.

$$\frac{\forall j, \exists i, externdecl_{ei}.\mathsf{name} = externdecl'_{ej}.\mathsf{name} \wedge externdecl_{ei}.\mathsf{desc} \preccurlyeq externdecl'_{ej}.\mathsf{desc}.}{\overline{externdecl_{ei}} \preccurlyeq \overline{externdecl'_{ej}}}$$

### 3.3.7 Component types

- A component type $\overline{externdecl_{ei}} \to instancetype_e$ is a subtype of a $\overline{externdecl'_{ej}} \to instancetype_e'$ if:
  - For each $i$, there exists some $j$, such that $externdecl'_{ej}.\mathsf{name} = externdecl_{ei}.\mathsf{name}$ and $externdecl'_{ej}.\mathsf{desc} \preccurlyeq externdecl_{ei}.\mathsf{desc}$; and
  - $instancetype_e \preccurlyeq instancetype_e'$

$$\frac{\forall i, \exists j, externdecl'_{ej}.\mathsf{name} = externdecl_{ei}.\mathsf{name} \land externdecl'_{ej}.\mathsf{desc} \preccurlyeq externdecl_{ei}.\mathsf{desc} \qquad instancetype_e \preccurlyeq instancetype_e'}{\overline{externdecl_{ei}} \to instancetype_e \preccurlyeq \overline{externdecl'_{ej}} \to instancetype_e'}$$

## 3.4 Components

### 3.4.1 No live values in context: $\dashv^{\mathsf{lv}} \Gamma$

- There must be no live values in $\Gamma.\mathsf{parent}$.
- Every type in $\Gamma.\mathsf{values}$ must be of the form $valtype_e{}^{\dagger}$.
- For each instance in $\Gamma.\mathsf{instances}$, every extern declaration which is not dead must have a descriptor which is not of the form $\mathsf{value}\ valtype_e$.
- Then there are no live values in the context $\Gamma$.

$$\frac{\dashv^{\mathsf{lv}} \Gamma.\mathsf{parent} \qquad \forall i, \exists valtype_e, \Gamma.\mathsf{values}[i] = valtype_e{}^{\dagger} \qquad \begin{array}{c}\forall i, \exists \overline{externdecl^{?}_{ej}}, \Gamma.\mathsf{values}[i] = externdecl^{?}_e \\ \land \forall j, \neg\exists valtype_e, externdecl^{?}_{ej} = \mathsf{value}\ valtype_e\end{array}}{\dashv^{\mathsf{lv}} \Gamma}$$

### 3.4.2 $\overline{definition_i}$

- $definition_1$ must have some type $componenttype_{e1}$ in context $\{\mathsf{parent}\ \Gamma\}$.
- For each $i > 1$, $definition_i$ must have some type $componenttype_{ei}$ in the context produced by typechecking $definition_{i-1}$.
- There must be no live values in the final context.
- Then the component $\overline{definition_i}$ has the type produced by finalizing $\bigoplus_i componenttype_{ei}$.

$$\frac{\Gamma_0 = \{\mathsf{parent}\ \Gamma\} \qquad \forall i, \Gamma_{i-1} \vdash definition_i : componenttype_{ei} \dashv \Gamma_i \qquad \dashv^{\mathsf{lv}} \Gamma_n}{\Gamma \vdash \overline{definition_i}^{n} : \langle\!\langle \bigoplus \overline{componenttype_{ei}} \rangle\!\rangle}$$

### 3.4.3 Core sort indices: $\Gamma \vdash \mathit{core{:}sortidx} : \mathit{core{:}importdesc}$

### 3.4.4 Instantiate/export arguments: $\Gamma \vdash \mathit{sortidx} : \mathit{externdesc}_e$.

#### Core modules

- If the type $\Gamma.\mathsf{core.modules}[i]$ exists in the context and is a subtype of $\mathit{core{:}moduletype}_e$, then $\{\mathsf{sort\ core\ module}, \mathsf{idx}\ i\}$ is valid with respect to extern descriptor $\mathsf{core\_module}\ \mathit{core{:}moduletype}_e$.

$$\frac{\Gamma \vdash \Gamma.\mathsf{core.modules}[i] \preccurlyeq \mathit{core{:}moduletype}_e}{\Gamma \vdash \{\mathsf{sort\ core\ module}, \mathsf{idx}\ i\} : \mathsf{core\_module}\ \mathit{core{:}moduletype}_e}$$

#### Functions

- If the type $\Gamma.\mathsf{funcs}[i]$ exists in the context and is a subtype of $\mathit{functype}_e$, then $\{\mathsf{sort\ func}, \mathsf{idx}\ i\}$ is valid with respect to extern descriptor $\mathsf{func}\ \mathit{functype}_e$.

$$\frac{\Gamma \vdash \Gamma.\mathsf{funcs}[i] \preccurlyeq \mathit{functype}_e}{\Gamma \vdash \{\mathsf{sort\ func}, \mathsf{idx}\ i\} : \mathsf{func}\ \mathit{functype}_e}$$

#### Values

- If the type $\Gamma.\mathsf{values}[i]$ exists in the context and is a subtype of $\mathit{valtype}_e$
- And $\mathit{valtype}_e$ is well-formed.
- Then $\{\mathsf{sort\ value}, \mathsf{idx}\ i\}$ is valid with respect to extern descriptor $\mathsf{value}\ \mathit{valtype}_e$.

$$\frac{\Gamma \vdash \Gamma.\mathsf{values}[i] \preccurlyeq \mathit{valtype}_e \qquad \Gamma \vdash \mathit{valtype}_e}{\Gamma \vdash \{\mathsf{sort\ value}, \mathsf{idx}\ i\} : \mathsf{value}\ \mathit{valtype}_e}$$

#### Types

- If the type $\Gamma.\mathsf{types}[i]$ exists in the context and is a subtype of $\mathit{deftype}_e$, then $\{\mathsf{sort\ type}, \mathsf{idx}\ i\}$ is valid with respect to extern descriptor $\mathsf{type}\ \mathit{deftype}_e$.

$$\frac{\Gamma \vdash \Gamma.\mathsf{types} \preccurlyeq \mathit{deftype}_e}{\Gamma \vdash \{\mathsf{sort\ type}, \mathsf{idx}\ i\} : \mathsf{type}\ \mathit{deftype}_e}$$

#### Instances

- If the type $\Gamma.\mathsf{instances}[i]$ exists in the context and is a subtype of $\mathit{instancetype}_e$, then $\{\mathsf{sort\ instance}, \mathsf{idx}\ i\}$ is valid with respect to extern descriptor $\mathsf{instance}\ \mathit{instancetype}_e$.

$$\frac{\Gamma \vdash \Gamma.\mathsf{instances}[i] \preccurlyeq \mathit{instancetype}_e}{\Gamma \vdash \{\mathsf{sort\ instance}, \mathsf{idx}\ i\} : \mathsf{instance}\ \mathit{instancetype}_e}$$

#### Components

- If the type $\Gamma.\mathsf{components}[i]$ exists in the context and is a subtype of $componenttype_e$, then $\{\mathsf{sort}\ \mathsf{component}, \mathsf{idx}\ i\}$ is valid with respect to extern descriptor $\mathsf{component}\ componenttype_e$.

$$\frac{\Gamma \vdash \Gamma.\mathsf{components}[i] \preccurlyeq componenttype_e}{\Gamma \vdash \{\mathsf{sort}\ \mathsf{component}, \mathsf{idx}\ i\} : \mathsf{component}\ componenttype_e}$$

### 3.4.5 Start arguments $\Gamma \vdash \overline{valueidx_i} : resulttype$

#### Single argument

$$\frac{\Gamma \vdash \Gamma.\mathsf{values}[i] \preccurlyeq valtype_e \lor \exists deftype_e, \Gamma.\mathsf{values}[i] = \mathsf{ref}\ \mathsf{call}\ deftype_e \land \Gamma \vdash \mathsf{own}\ deftype_e \preccurlyeq valtype_e}{\Gamma \vdash \overline{valueidx_i} : valtype_e}$$

#### Multiple arguments

$$\frac{\forall i, \Gamma \vdash valueidx_i : valtype_{ei}}{\Gamma \vdash \overline{valueidx_i} : \overline{\mathsf{name}\ name_i, \mathsf{type}\ valtype_{ei}}}$$

### 3.4.6 Definitions

$\mathsf{core\_module}\ core{:}module$

- The core module $core{:}module$ must be valid (as per Core WebAssembly) with respect to the elaborated core module type $core{:}moduletype_e$.
- Then $\mathsf{core\_module}\ core{:}module$ is valid with respect to the empty component type, and sets $\mathsf{core}.\mathsf{modules}$ in the context to the orginal $\Gamma.\mathsf{core}.\mathsf{modules}$ followed by $core{:}moduletype_e$.

$$\frac{\vdash core{:}module : core{:}moduletype_e}{\begin{array}{l}\Gamma \vdash \mathsf{core\_module}\ core{:}module \\ \quad : \forall \varnothing.\varnothing \to \exists \varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{core}.\mathsf{modules}\ core{:}moduletype_e\}\end{array}}$$

$\mathsf{core\_instance}\ \mathsf{instantiate}\ core{:}moduleidx\ \overline{core{:}instantiatearg_i}$

- No two instantiate arguments may have identical $\mathsf{name}$ members.
- The type $\Gamma.\mathsf{core}.\mathsf{modules}[core{:}moduleidx]$ must exist in the context, and for each $core{:}importdecl$ in that type:
  - There must exist an instantiate argument whose $\mathsf{name}$ member matches its core:module member, such that:
    * If the argument's $\mathsf{instance}$ member is $core{:}instanceidx$, then the type $\Gamma.\mathsf{core}.\mathsf{instances}[core{:}instanceidx]$ must exist in the context, and furthermore, must contain an export whose core:name member matches the import declarations core:name member, and whose core:desc member is a subtype of the import declaration's core:desc member.

$$\frac{\begin{array}{c}\Gamma.\mathsf{core.modules}[core{:}moduleidx] = \overline{core{:}importdecl_j} \to \overline{core{:}instancetype_e} \\ \forall j, \exists i, core{:}instantiatearg_i.\mathsf{name} = core{:}importdecl_j.\mathsf{core{:}module} \\ \wedge \Gamma.\mathsf{core.instances}[core{:}instantiatearg_i.\mathsf{instance}] = \overline{core{:}exportdecl_l} \\ \wedge \exists l, core{:}exportdecl_l.\mathsf{core{:}name} = core{:}importdecl_j.\mathsf{core{:}name} \\ \wedge core{:}exportdecl_l.\mathsf{core{:}desc} \preccurlyeq core{:}importdecl_j.\mathsf{core{:}desc} \\ \forall i, \forall i', core{:}instantiatearg_i.\mathsf{name} = core{:}instantiatearg_{i'}.\mathsf{name} \Rightarrow i = i'\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{core\_instance\ instantiate}\ core{:}moduleidx\ \overline{core{:}instantiatearg_i} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{core.instances}\ core{:}instancetype_e\}\end{array}}$$

core_instance exports $\overline{\{\mathsf{name}\ name_i, \mathsf{def}\ core{:}sortidx_i\}}$

- Each $name_i$ must be distinct.
- Each $core{:}sortidx_i$ must be valid with respect to some $core{:}importdesc_i$.
- Then core_instance exports $\overline{\{\mathsf{name}\ name_i, \mathsf{def}\ core{:}sortidx_i\}}$ is valid with respect to the empty module type, and sets core.instances in the context to the original core.instances followed by $\overline{\{\mathsf{name}\ name_i, \mathsf{desc}\ core{:}importdesc_i\}}$.

$$\frac{\begin{array}{c}\forall i, \Gamma \vdash core{:}sortidx_i : core{:}importdesc_i \\ \forall ij, name_i = name_j \Rightarrow i = j\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{core\_instance\ exports}\ \overline{\{\mathsf{name}\ name_i, \mathsf{def}\ core{:}sortidx_i\}} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{core.instances}\ \overline{\{\mathsf{name}\ name_i, \mathsf{desc}\ core{:}importdesc_i\}}\}\end{array}}$$

core_type $core{:}deftype$

- The type $core{:}deftype$ must elaborate to some $core{:}deftype_e$.
- Then the definition core_type $core{:}deftype$ is valid with respect to the empty module type, and sets core.types in the context to the original $\Gamma$.core.types followed by $core{:}deftype_e$.

$$\frac{\Gamma \vdash core{:}deftype \leadsto core{:}deftype_e}{\Gamma \vdash \mathsf{core\_type}\ core{:}deftype : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \dashv \Gamma \oplus \{\mathsf{core.types}\ core{:}deftype_e\}}$$

component $component$

- It must be possible to split the context $\Gamma$ such that the component $component$ is valid for some type $componenttype_e$ in the first portion of the context
- Then the definition component $component$ is valid with respect to the empty component type, and sets the context to the second portion of the aforementioned split of the context, further updated by setting components to the original $\Gamma_2$.components followed by $componenttype_e$.

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \boxplus \Gamma_2 \\ \Gamma_1 \vdash component : componenttype_e\end{array}}{\Gamma \vdash component : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \dashv \Gamma_2 \oplus \{\mathsf{components}\ componenttype_e\}}$$

$\mathsf{instance}\ \mathsf{instantiate}\ \mathit{componentidx}\ \overline{\mathit{instantiatearg}_i}$

- The type $\Gamma.\mathsf{components}[\mathit{componentidx}]$ must exist in the context, and for each $\mathit{externdecl}_e$ in that type:
  - There must exist an instantiate argument whose $\mathsf{name}$ member matches its name member and whose $\mathsf{arg}$ is valid with respect to its desc.
- Then $\mathsf{instance}\ \mathsf{instantiate}\ \mathit{componentidx}\ \overline{\mathit{instantiatearg}_i}$ is valid with respect to the empty module type, and sets $\mathsf{instances}$ in the context to the original $\Gamma.\mathsf{instances}$ followed by $\mathit{instancetype}_e$ of $\Gamma.\mathsf{components}[\mathit{componentidx}]$, and marks as dead in the context any values present in $\overline{\mathit{instantiatearg}_i}$.

$$\frac{\begin{array}{c}\Gamma.\mathsf{components}[\mathit{componentidx}] = \forall\overline{\mathit{boundedtyvar}_j}.\overline{\mathit{externdecl}_{ek}} \to \mathit{instancetype}_e \\ \forall j, \exists \mathit{deftype}_{ej}, \mathit{deftype}_{ej} \preccurlyeq \mathit{boundedtyvar}_j \\ \overline{\mathit{externdecl}'_{ek}} \to \exists\overline{\mathit{boundedtyvar}'_o}\mathit{instancetype}_e{}' = (\overline{\mathit{externdecl}_{ek}} \to \mathit{instancetype}_e)[\overline{\mathit{deftype}_{ej}/\mathit{boundedtyvar}_j}] \\ \forall k, \exists i, \mathit{instantiatearg}_i.\mathsf{name} = \mathit{externdecl}'_{ek}.\mathsf{name} \\ \wedge \Gamma \vdash \mathit{instantiatearg}_i.\mathsf{arg} : \mathit{externdecl}'_{ek}.\mathsf{desc} \\ \forall l, \mathit{valtype}^?_{el} = \begin{cases} \Gamma.\mathsf{values}[l]^\dagger & \text{if } \begin{array}{l}\exists i, \mathit{instantiatearg}_i.\mathsf{arg}.\mathsf{sort} = \mathsf{value} \\ \wedge \mathit{instantiatearg}_i.\mathsf{arg}.\mathsf{idx} = k\end{array} \\ \Gamma.\mathsf{values}[l] & \text{otherwise}\end{cases} \\ \forall m, \mathit{instancetype}^?_{em} = \begin{cases} \mathit{instancetype}_e{}' & \text{if } m = \|\Gamma.\mathsf{instances}\| \\ \exists \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^\dagger_{en}} & \text{if } \begin{array}{l}\exists i, \mathit{instantiatearg}_i.\mathsf{arg}.\mathsf{sort} = \mathsf{component} \\ \wedge \mathit{instantiatearg}_i.\mathsf{arg}.\mathsf{idx} = m \\ \wedge \Gamma.\mathsf{instances}[m] = \exists \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^?_{en}}\end{array} \\ \Gamma.\mathsf{instances}[m] & \text{otherwise}\end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{instance}\ \mathsf{instantiate}\ \mathit{componentidx}\ \overline{\mathit{instantiatearg}_i} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma' \ominus \{\mathsf{values}, \mathsf{instances}\} \oplus \{\mathsf{uvars}\ \overline{\mathit{boundedtyvar}'_o}, \mathsf{instances}\ \overline{\mathit{instancetype}^?_{em}}, \mathsf{values}\ \overline{\mathit{valtype}^?_{el}}\}\end{array}}$$

$\mathsf{instance}\ \mathsf{exports}\ \overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{def}\ \mathit{sortidx}_i\}}$

- Each $\mathit{name}_i$ must be distinct.
- Each $\mathit{sortidx}_i$ must be valid with respect to some $\mathit{externdesc}_{ei}$.
- Then $\mathsf{instance}\ \mathsf{exports}\ \overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{def}\ \mathit{sortidx}_i\}}$ is valid with respect to the empty module type, and sets $\mathsf{instances}$ in the context to the original $\Gamma.\mathsf{instances}$ followed by $\langle\!\langle\exists(\Gamma.\mathsf{evars}).\overline{\mathsf{name}\ \mathit{name}_i, \mathsf{desc}\ \mathit{externdesc}_{ei}}\rangle\!\rangle$, and marks as dead in the context any values present in $\overline{\mathit{sortidx}_i}$.
- TODO: What is the right way to choose which type variables to put into the existential here?

$$
\frac{\begin{array}{c}\forall i, \Gamma \vdash \mathit{sortidx}_i : \mathit{externdesc}_{ei} \\ \forall ij, \mathit{name}_i = \mathit{name}_j \Rightarrow i = j \\ \forall j, \mathit{valtype}^?_{ej} = \begin{cases} \Gamma.\mathsf{values}[j]^\dagger & \text{if } \begin{array}{l}\exists i, \mathit{sortidx}_i.\mathsf{sort} = \mathsf{value} \\ \wedge \mathit{sortidx}_i.\mathsf{idx} = j\end{array} \\ \Gamma.\mathsf{values}[j] & \text{otherwise} \end{cases} \\ \mathit{instancetype}_e = \langle\!\langle \exists(\Gamma.\mathsf{evars}).\overline{\mathsf{name}\ \mathit{name}_i, \mathsf{desc}\ \mathit{externdesc}_{ei}} \rangle\!\rangle \\ \forall k, \mathit{instancetype}^?_{ek} = \begin{cases} \mathit{instancetype}_e & \text{if } k = \|\Gamma.\mathsf{instances}\| \\ \exists \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^\dagger_{el}} & \text{if } \begin{array}{l}\exists i, \mathit{sortidx}_i.\mathsf{sort} = \mathsf{instance} \\ \wedge \mathit{sortidx}_i.\mathsf{idx} = k \\ \wedge \Gamma.\mathsf{instances}[k] = \forall \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^?_{el}}\end{array} \\ \Gamma.\mathsf{instances}[k] & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{instance\ exports}\ \overline{\{\mathsf{name}\ \mathit{name}_i, \mathsf{def}\ \mathit{sortidx}_i\}} \\ \quad : \forall \varnothing.\varnothing \to \exists \varnothing.\varnothing \\ \quad \dashv \Gamma \ominus \{\mathsf{instances}, \mathsf{values}\} \oplus \{\mathsf{instances}\ \overline{\mathit{instancetype}^?_{ek}}, \mathsf{values}\ \overline{\mathit{valtype}^?_{ej}}\}\end{array}}
$$

alias {sort *sort*, target export *instanceidx name*}

- This rule applies of $\mathit{sort} \neq \mathsf{instance}$.
- The type $\Gamma.\mathsf{instances}[\mathit{instanceidx}]$ must exist in the context.
- Some extern descriptor with a matching *name* and some desc *desc* must exist within $\Gamma.\mathsf{instances}[\mathit{instanceidx}]$.
- Then alias {sort *sort*, target export *instanceidx name*} is valid with respect to the empty component type, and sets index_space(*sort*) to the original $\Gamma$.index_space(*sort*) followed by *desc*.

$$
\frac{\begin{array}{c}\Gamma.\mathsf{instances}[\mathit{instanceidx}] = \overline{\mathit{externdecl}^?_{ei}} \\ \exists i, \mathit{externdecl}^?_{ei}.\mathsf{name} = \mathit{name} \\ \forall j, \mathit{externdecl}^{?\prime}_{e\ j} = \begin{cases} \mathit{externdecl}^{?\dagger}_{ej} & \text{if } \mathit{sort} = \mathsf{value} \wedge j = i \\ \mathit{externdecl}^?_{ej} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{alias}\ \{\mathsf{sort}\ \mathit{sort}, \mathsf{target}\ \mathsf{export}\ \mathit{instanceidx}\ \mathit{name}\} \\ \quad : \forall \varnothing.\varnothing \to \exists \varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathrm{index\_space}(\mathit{sort})\ \mathit{externdecl}^?_{ei}.\mathsf{desc}, \mathsf{instances}[i]\ \overline{\mathit{externdecl}^{?\prime}_{e\ j}}\}\end{array}}
$$

alias {sort instance, target export *instanceidx name*}

- The type $\Gamma.\mathsf{instances}[\mathit{instanceidx}]$ must exist in the context.
- Some extern descriptor with a matching *name* and a *desc* of the form instance $\forall\overline{\mathit{boundedtyvar}_l}.\overline{\mathit{externdecl}^?_{em}}w$ must exist within $\Gamma.\mathsf{instances}[\mathit{instanceidx}]$.
- Then alias {sort instance, target export *instanceidx name*} is valid with respect to the empty component type, and sets instances to the original instances followed by $:\overline{\mathit{externdecl}^?_{em}}$, and sets uvars to the original uvars followed by $\overline{\mathit{boundedtyvar}_l}$.

$$\frac{\begin{array}{c}\Gamma.\mathsf{instances}[\mathit{instanceidx}] = \overline{\mathit{externdecl}^?_{e\,i}} \\ \exists i, \mathit{externdecl}^?_{e\,i}.\mathsf{name} = \mathit{name} \\ \mathit{externdecl}^?_{e\,i}.\mathsf{desc} = \mathsf{instance}\ \forall \overline{\mathit{boundedtyvar}_l}.\overline{\mathit{externdecl}^?_{e\,m}} \\ \forall j, \mathit{externdecl}^{?\prime}_{e\ j} = \begin{cases} \forall \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^\dagger_{e\,k}} & \text{if } j = i \\ & \quad \wedge \mathit{externdecl}^?_e = \forall \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^?_{e\,k}} \\ \mathit{externdecl}^?_{e\,j} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{alias}\ \{\mathsf{sort}\ \mathit{sort}, \mathsf{target}\ \mathsf{export}\ \mathit{instanceidx}\ \mathit{name}\} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{uvars}\ \overline{\mathit{boundedtyvar}_l} \mathsf{instances}[i]\ \overline{\mathit{externdecl}^{?\prime}_{e\ j}\ \overline{\mathit{externdecl}^?_{e\,m}}}\}\end{array}}$$

alias {sort *sort*, target core_export *core:instanceidx name*}

- The type $\Gamma$.core.instances[*core:instanceidx*] must exist in the context.
- *sort* must be core *core:sort*.
- Some export declarator with a matching *name* and some desc *desc* must exist within $\Gamma$.instances[*instanceidx*].
- Then alias {sort *sort*, target core_export *core:instanceidx name*} is valid with respect to the empty component type, and sets index_space(*sort*) to the original $\Gamma$.index_space(*sort*) followed by *desc*.

$$\frac{\begin{array}{c}\mathit{sort} = \mathsf{core}\ \mathit{core{:}sort} \\ \Gamma.\mathsf{core.instances}[\mathit{core{:}instanceidx}] = \overline{\mathit{core{:}exportdecl}_i} \\ \mathit{core{:}exportdecl}_i.\mathsf{name}\ \mathit{name}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{alias}\ \{\mathsf{sort}\ \mathit{sort}, \mathsf{target}\ \mathsf{core\_export}\ \mathit{core{:}instanceidx}\ \mathit{name}\} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathrm{index\_space}(\mathit{sort})\ \mathit{core{:}exportdecl}_i.\mathsf{desc}\}\end{array}}$$

alias {sort *sort*, target outer $u32_o$ $u32_i$}

- *sort* must be one of component, core module, type, or core type.
- $\Gamma$.parent[$u32_o$].index_space(*sort*)[$u32_i$] must exist in the context.
- If *sort* is *STYPE*, then $\Gamma$.parent[$u32_o$].types[$u32_i$] must not be of the form resource $i$ for any $i$.
- Then alias {sort *sort*, target outer $u32_o$ $u32_i$} is valid with respect to the empty component type, and sets index_space(*sort*) in the context to the original $\Gamma$.index_space(*sort*) followed by $\Gamma$.parent[$u32_o$].index_space(*sort*)[$u32_i$].

$$\frac{\begin{array}{c}\mathit{sort} \in \{\mathsf{component}, \mathsf{core\ module}, \mathsf{type}, \mathsf{core\ type}\} \\ \mathit{sort} = \mathsf{type} \Rightarrow \forall i.\Gamma.\mathsf{parent}[u32_o].\mathsf{types}[u32_i] \neq \mathsf{resource}\ i\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{alias}\ \{\mathsf{sort}\ \mathit{sort}, \mathsf{target}\ \mathsf{outer}\ u32_o\ u32_i\} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\mathrm{index\_space}(\mathit{sort})\ \Gamma.\mathsf{parent}[u32_o].\mathrm{index\_space}(\mathit{sort})[u32_i]\}\end{array}}$$

$\mathsf{type}~\mathit{deftype}$

- The type $\mathit{deftype}$ must elaborate to some $\mathit{deftype}_e$.
- Then $\mathsf{type}~\mathit{deftype}$ is valid with respect to the empty component type, and sets $\mathsf{types}$ in the context to the original $\Gamma.\mathsf{types}$ followed by $\mathit{deftype}_e$.

$$\frac{\Gamma \vdash \mathit{deftype} \leadsto \mathit{deftype}_e \qquad \mathrm{fresh}(\alpha)}{\begin{array}{l}\Gamma \vdash \mathsf{type}~\mathit{deftype} \\ \quad : \forall\varnothing.\varnothing \rightarrow \varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{types}~\mathit{deftype}_e\}\end{array}}$$

$\mathsf{type}~\{\mathsf{rep}~\mathtt{i32}, \mathsf{dtor}~\mathit{funcidx}\}$

- $\Gamma.\mathsf{funcs}[\mathit{funcidx}]$ must exist.
- Then $\mathsf{type}~\{\mathsf{rep}~\mathtt{i32}, \mathsf{dtor}~\mathit{funcidx}\}$ is valid with respect to the empty component type, and appends $\{\mathsf{rep}~\mathtt{i32}, \mathsf{dtor}~\mathit{funcidx}\}$ to $\mathsf{rtypes}$ in the context, and sets $\mathsf{types}$ in the context to the original $\Gamma.\mathsf{types}$ followed by $\mathsf{resource}~\mathrm{length}(\Gamma.\mathsf{rtypes})$.

$$\frac{}{\Gamma \vdash \mathsf{type}~\{\mathsf{rep}~\mathtt{i32}, \mathsf{dtor}~\mathit{funcidx}\} : \forall\varnothing.\varnothing \rightarrow \varnothing \dashv \Gamma \oplus \{\mathsf{rtypes}~\{\mathsf{rep}~\mathtt{i32}, \mathsf{dtor}~\mathit{funcidx}\}, \mathsf{types}~\mathsf{resource}~\mathrm{length}(\Gamma.\mathsf{rtypes})\}}$$

$\mathsf{canon}~\mathsf{lift}~\mathit{core{:}funcidx}~\overline{\mathit{canonopt}_i}~\mathit{typeidx}$

- $\Gamma.\mathsf{types}[\mathit{typeidx}]$ must exist and be a $\mathit{functype}_e$.
- $\mathrm{canon\_lower\_type}(\mathit{functype}_e, \overline{\mathit{canonopt}_i})$ must be equal to $\Gamma.\mathsf{core}.\mathsf{funcs}[\mathit{core{:}funcidx}]$.
- Then $\mathsf{canon}~\mathsf{lift}~\mathit{core{:}funcidx}~\overline{\mathit{canonopt}_i}~\mathit{typeidx}$ is valid with respect to the empty component type, and sets $\mathsf{funcs}$ in the context to the original $\Gamma.\mathsf{funcs}$ followed by $\mathit{functype}_e$.

$$\frac{\Gamma.\mathsf{types}[\mathit{typeidx}] = \mathit{functype}_e \qquad \Gamma.\mathsf{core}.\mathsf{funcs}[\mathit{core{:}funcidx}] = \mathrm{canon\_lower\_type}(\mathit{functype}_e, \overline{\mathit{canonopt}_i})}{\Gamma \vdash \mathsf{canon}~\mathsf{lift}~\mathit{core{:}funcidx}~\overline{\mathit{canonopt}_i}~\mathit{typeidx} : \varnothing \rightarrow \varnothing \dashv \Gamma \oplus \{\mathsf{funcs}~\mathit{functype}_e\}}$$

$\mathsf{canon}~\mathsf{lower}~\mathit{funcidx}~\overline{\mathit{canonopt}_i}$

- The type $\Gamma.\mathsf{funcs}[\mathit{funcidx}]$ must exist in the context.
- $\mathrm{canon\_lower\_type}(\Gamma.\mathsf{funcs}[\mathit{funcidx}], \overline{\mathit{canonopt}_i})$ must be defined (to be some $\mathit{core{:}functype}$.
- Then $\mathsf{canon}~\mathsf{lower}~\mathit{funcidx}~\overline{\mathit{canonopt}_i}$ is valid with respect to the empty component type, and sets $\mathsf{core}.\mathsf{funcs}$ in the context to the original $\Gamma.\mathsf{core}.\mathsf{funcs}$ followed by that $\mathit{core{:}functype}$.

$$\frac{}{\begin{array}{l}\Gamma \vdash \mathsf{canon}~\mathsf{lower}~\mathit{funcidx}~\overline{\mathit{canonopt}_i} \\ \quad : \forall\varnothing.\varnothing \rightarrow \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \oplus \{\Gamma.\mathsf{core}.\mathsf{funcs}~\mathrm{canon\_lower\_type}(\Gamma.\mathsf{funcs}[\mathit{funcidx}], \overline{\mathit{canonopt}_i})\}\end{array}}$$

$\mathsf{start}~\{\mathsf{func}~\mathit{funcidx}, \mathsf{args}~\overline{\mathit{valueidx}_i}\}$

- The type $\Gamma.\mathsf{funcs}[\mathit{funcidx}]$ must be defined in the context.
- The arguments $\overline{\mathit{valueidx}_i}$ must be valid with respect to the parameter list of the function.
- Then $\mathsf{start}~\{\mathsf{func}~\mathit{funcidx}, \mathsf{args}~\overline{\mathit{valueidx}_i}\}$ is valid with respect to the empty component type, and sets $\mathsf{values}$ in the context to the original $\Gamma.\mathsf{values}$ followed by the types of the return values of the function.

$$\frac{\begin{array}{c}\Gamma.\mathsf{funcs}[\mathit{funcidx}] = \mathit{resulttype}_e \to \mathit{resulttype}_e{}' \\ \Gamma \vdash \overline{\mathit{valueidx}_i} : \mathit{resulttype}_e \\ n = \mathrm{length}(\Gamma.\mathsf{values}) \\ \forall j, \mathit{valtype}^{?\prime}_{ej} = \begin{cases} \Gamma.\mathsf{values}[j]^\dagger & \text{if } \exists i\, \forall \mathit{deftype}, j < n \land j = \mathit{valueidx}_i \\ & \quad \land \mathit{resulttype}_{ei}.\mathsf{type} \neq \mathsf{ref~call}~\mathit{deftype}_e \\ \Gamma.\mathsf{values}[j] & \text{if } j < n \land j \notin \overline{\mathit{valueidx}_i} \\ \mathit{resulttype}'_{e\,j-n} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{start}~\{\mathsf{func}~\mathit{funcidx}, \mathsf{args}~\overline{\mathit{valueidx}_i}\} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\varnothing \\ \quad \dashv \Gamma \ominus \{\mathsf{values}\} \oplus \{\mathsf{values}~\overline{\mathit{valtype}^{?\prime}_{ej}}\}\end{array}}$$

$\mathsf{import}~\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}\}$

- The $\mathit{externdesc}$ must elaborate to some $\forall \mathit{boundedtyvar}^*.\mathit{externdesc}_e$.
- Then the definition $\mathsf{import}~\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}\}$ is valid with respect to the component type whose export list is empty and whose import list is the singleton containing $\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}_e\}$, and updates the context with desc.

$$\frac{\Gamma \vdash \mathit{externdesc} \rightsquigarrow \forall \mathit{boundedtyvar}^*.\mathit{externdesc}_e}{\begin{array}{l}\Gamma \vdash \mathsf{import}~\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}\} \\ \quad : \forall \mathit{boundedtyvar}^*.\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}_e\} \to \varnothing \\ \quad \dashv \Gamma \oplus \{\mathsf{uvars}~\mathit{boundedtyvar}^*, \mathit{externdesc}_e\}\end{array}}$$

$\mathsf{export}~\{\mathsf{name}~\mathit{name}, \mathsf{def}~\mathit{sortidx}\}$

- This rule applies when when $\mathit{sortidx}.\mathsf{sort}$ is not *STYPE*.
- The $\mathit{sortidx}$ must be valid with respect to some $\mathit{externdesc}_e$.
- Then the definition $\mathsf{export}~\{\mathsf{name}~\mathit{name}, \mathsf{def}~\mathit{sortidx}\}$ is valid with respect to the component type whose import list is empty and whose export list is the singleton containing $\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}_e\}$

$$\frac{\begin{array}{c}\mathit{sortidx}.\mathsf{sort} \neq \mathsf{type} \\ \Gamma \vdash \mathit{sortidx} : \mathit{externdesc}_e \\ \forall j, \mathit{valtype}^{?\prime}_{ej} = \begin{cases} \Gamma.\mathsf{values}[j]^\dagger & \text{if } \mathit{sortidx}.\mathsf{sort} = \mathsf{value} \land \mathit{sortidx}.\mathsf{idx} = j \\ \Gamma.\mathsf{values}[j] & \text{otherwise} \end{cases} \\ \forall k, \mathit{instancetype}^{?\prime}_{ek} = \begin{cases} \forall \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^\dagger_{el}} & \text{if } \mathit{sortidx}.\mathsf{sort} = \mathsf{component} \land \mathit{sortidx}.\mathsf{idx} = j \\ & \quad \land \Gamma.\mathsf{instances}[j] = \forall \mathit{boundedtyvar}^*.\overline{\mathit{externdecl}^?_{el}} \\ \Gamma.\mathsf{instances}[j] & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{export}~\{\mathsf{name}~\mathit{name}, \mathsf{def}~\mathit{sortidx}\} \\ \quad : \forall\varnothing.\varnothing \to \exists\varnothing.\{\mathsf{name}~\mathit{name}, \mathsf{desc}~\mathit{externdesc}_e\} \\ \quad \dashv \Gamma \ominus \{\mathsf{values}, \mathsf{instances}\} \oplus \{\mathsf{values}~\overline{\mathit{valtype}^{?\prime}_{ej}}, \mathsf{instances}~\overline{\mathit{instancetype}^{?\prime}_{ek}}\}\end{array}}$$

$\mathsf{export}\ \{\mathsf{name}\ name, \mathsf{def}\ \{\mathsf{sort}\ \mathsf{type}, \mathsf{idx}\ typeidx\}\}$

- Then the definition $\mathsf{export}\ \{\mathsf{name}\ name, \mathsf{def}\ sortidx\}$ is valid with respect to the component type whose import list is empty and whose export list is the singleton containing $\{\mathsf{name}\ name, \mathsf{desc}\ externdesc_e\}$

$$\frac{\begin{array}{c} sortidx.\mathsf{sort} \neq \mathsf{type} \\ \mathrm{fresh}(\alpha) \\ \Gamma.\mathsf{types}[typeidx] = deftype_e \\ typebound_e = \begin{cases} \mathsf{sub}\ \mathsf{resource} & \text{if } \exists i, deftype_e = \mathsf{resource}\ i \\ \mathsf{eq}\ deftype_e & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} \Gamma \vdash \mathsf{export}\ \{\mathsf{name}\ name, \mathsf{def}\ sortidx\} \\ \quad : \forall\varnothing.\varnothing \to \exists(\alpha : typebound_e).\{\mathsf{name}\ name, \mathsf{desc}\ \mathsf{type}\ \alpha\} \\ \quad \dashv \Gamma \oplus \{\mathsf{evars}\ (\alpha : typebound_e, deftype_e), \mathsf{types}\ \alpha\} \end{array}}$$

CHAPTER

FOUR

# EXECUTION

TODO: Describe the execution semantics of a component

# CHAPTER FIVE

# BINARY FORMAT

TODO: Formal write-up of the binary format.

CHAPTER

SIX

# TEXT FORMAT

TODO: Formal write-up of the text format.

# CHAPTER SEVEN

# APPENDIX

CHAPTER

EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX