A C/C++ API for Wasm

Proposal

Andreas Rossberg

Dfinity

Motivation & Goals

API for embedding Wasm VM in other apps

… "black box" embedding; white box non-goal for now

Enable 3rd language APIs via FFI bindings

… provide plain C API

Compatibility

… agnostic to VM specifics

… binary compatibility: can swap VMs at link time

Standardise similar to JS API

C vs C++

C preferable for foreign language interop

... avoid difficult features (varargs, val structs)

... explicit management of interface objects

C++ preferable for safe native use

... convenience and readability

... smart pointers

Currently proposal defines both

... but focus on C for now

API Outline

Engine & Store

Modules & Instances

Functions, Globals, Tables, Memories

Values & References

Type Representations

Auxiliaries

Engine

```
struct config_t;
struct engine_t;

engine_t* engine_new();
engine_t* engine_new_with_config(config_t*);
```

Stores

```
struct store_t;

store_t* store_new(engine_t*);
```

Modules

```
struct module_t;

store_t* module_new(store_t*, const byte_vec_t*);

void module_imports(const module_t*, importtype_vec_t*);
void module_exports(const module_t*, exporttype_vec_t*);

void module_serialize(const module_t*, byte_vec_t*);
module_t* module_deserialize(store_t*, const byte_vec_t*);

shared_module_t* module_share(const module_t*);
module_t* module_obtain(store_t*, const shared_module_t*);
```

Values

```
union val_t {
    int32_t i32;
    int64_t i64;
    float32_t f32;
    float64_t f64;
    ref_t* ref;
};
```

References

```
struct ref_t;

void ref_delete(ref_t*);
```

Functions

```
struct func_t;

trap_t* func_call(const func_t*, const val_t args[], val_t res[]);

typedef trap_t* (*func_callback_t)(
    const val_t args[], val_t res[]);
typedef trap_t* (*func_callback_with_env_t)(
    void* env, const val_t args[], val_t res[]);

func_t* func_new(
    store_t*, functype_t*, func_callback_t);
func_t* func_new_with_env(
    store_t*, functype_t*, func_callback_with_env_t);

functype_t* func_type(const func_t*);
```

Globals

```
struct global_t;

global_t* global_new(store_t*, globaltype_t*, val_t);

void global_get(const global_t*, const val_t*);
void global_set(global_t*, val_t*);

globaltype_t* global_type(const global_t*);
```

Tables

```
struct table_t;

table_t* table_new(store_t*, tabletype_t*, );

ref_t* table_get(const table_t*, uint32_t);
void table_set(table_t*, uint32_t, ref_t*);
uint32_t table_size(const table_t*);
uint32_t table_grow(table_t*, uint32_t, ref_t*);

tabletype_t* table_type(const table_t*);
```

Memories

```
struct memory_t;

memory_t* memory_new(store_t*, memorytype_t*, );

byte_t* memory_data(memory_t*);
uint32_t memory_size(const memory_t*);
uint32_t memory_grow(memory_t*, uint32_t);

memorytype_t* memory_type(const memory_t*);
```

Instances

```
struct instance_t;

instance_t* instance_new(
    store_t*, const module_t*, const extern_t const[]);

void instance_exports(
    const instance_t*, extern_vec_t*)
```

Externals

```
struct extern_t;

extern_t* func_as_extern(func_t*);
extern_t* global_as_extern(global_t*);
extern_t* table_as_extern(table_t*);
extern_t* memory_as_extern(memory_t*);

func_t* extern_as_func(extern_t*);
global_t* extern_as_global(extern_t*);
table_t* extern_as_table(extern_t*);
memory_t* extern_as_memory(extern_t*);

externtype_t* extern_type(const extern_t*);
```

Types

…see proposal for details

V8 Prototype

Implemented on top of V8 API (plus a few hacks)

C API implemented on top of C++ API

Some debugging aids (tracing of API allocations)

Limitations

… external calls have to go through JS (slow)

… cannot call i64 or multi-result functions

… host functions & globals created through aux modules

Open Questions

High performance call interface

… seems to require platform-specific impl

… post-MVP?

Tagged val_t or not?

Simplify use of vectors?