BAT2C++

Calling Julia from C++

A. Khudyakov

12 Nov 2018

Using BAT2 in C++ programs

> You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.
>
> *Joe Armstrong*

Our jungle

Banana: using BAT2

1. We simply want use BAT2 as library

Our jungle

Banana: using BAT2

1. We simply want use BAT2 as library

Gorilla: mapping API

1. Mapping Julia API of BAT2 onto C++

Our jungle

Banana: using BAT2

1. We simply want use BAT2 as library

Gorilla: mapping API

1. Mapping Julia API of BAT2 onto C++

Jungle: Julia/C++ interoperability

1. Memory management.
2. Marshalling data between C++ & Julia.
3. Multithreading.

Jungle

Memory management

C++

- Manual memory management
- Stack allocation
- RAII

Programmer manages memory all by himself.

Julia

- Garbage collection

Julia runtime allocates and frees data.

We have two conflicting strategies for memory management. How can we reconcile them?

C++ view on Julia's heap

Julia objects:

```
jl_value_t* val
```

1. Opaque pointers
2. Could be reclaimed by GC at any moment

C++ view on Julia's heap

Julia objects:

```
jl_value_t* val
```

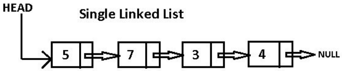
1. Opaque pointers
2. Could be reclaimed by GC at any moment

Values could be protected by placing them into GC root

GC root

```
struct gc_root {
  int            size;
  jl_values_t** objects[];
};
```

- Roots arranged as linked list



- Roots are normally allocated on stack
- *Lifetime of rooted object is duration of function call!*

Keeping Julia values alive

GC root allocated on stack:

Problems:

1. Values returned from function are not protected
2. Size of GC root is static!

- OK for hand-rolled code
- Unacceptable for high level API

Keeping Julia values alive

GC root allocated on stack:

Problems:

1. Values returned from function are not protected
2. Size of GC root is static!

- OK for hand-rolled code
- Unacceptable for high level API

Solution:

Create C++ wrapper class which will protect Julia values from GC.

1. Allocate 1-element GC root on heap during Julia initialization.
2. It contains `IdDict{Any,Int64}()` acting as reference counter
3. We increment counter on creation of new wrapper.
4. We decrement on destruction.

Keeping Julia values alive

Wrapper object

```
class Value {
public:
  explicit Value(jl_value_t*);
  jl_value_t* juliaValue() const;
private:
  std::shared_ptr<GCBarrier> m_value;
};

struct GCBarrier : non_copyable {
  jl_value_t* m_val;
};
```

Conversion between data types

Conversion is generally easy if somewhat verbose *unless*:

1. Callbacks are involved

   OR

2. High performance is desired

Conversion between data types

Conversion is generally easy if somewhat verbose *unless*:

1. Callbacks are involved

OR

2. High performance is desired

Just copy data and be happy about it:

```
template<>
jl_value_t* toJulia<T>(T x) {
    ...

template<>
T fromJulia<T>(jl_value_t* x) {
    ...
```

Callbacks

Say we want to pass as callback to Julia:

```
double foo(int, const std::vector<double>&)
```

Before calling `foo`:

1. Unpack 1st argument as `int`
2. Convert 2nd argument from Julia's `Array` to `std::vector`
3. If conversion fails abort

After calling `foo`:

1. Wrap `double` into Julia value

Lets make a closure!

Make wrapper function:

```
jl_value_t* foo(void* f, jl_value_t* a, jl_value_t* b)
```

- f — function we want to call
- a,b — parameters coming from Julia

Function body:

```
auto fun = (double(*)(int, const std::vector&))(f);
int                 v_a = fromJulia(a)
std::vector<double> v_b = fromJulia(a);
double              r   = fun(v_a, v_b);
return toJulia(r);
```

Making callbacks

Of course we'll template everything so in the end API will look like:

```
template<T>
jl_value_t* wrapFunction( T(*)() );

template<T,A>
jl_value_t* wrapFunction( T(*)(A) );

template<T,A,B>
jl_value_t* wrapFunction( T(*)(A,B) );
```

To do:

- Calling object methods
- Making closures (but in C++ it again means calling object methods)

Gorilla

Designing API

No C++ wrappers for BAT2 so far

Design guidelines

- We can map Julia values onto C++ objects
- Julia inheritance of abstract types maps to C++ inheritance
- No direct correspondence for multimethods

- Use Julia introspection to avoid handwritten boilerplate?
- How to expose generation of samples?

No banana yet

Conclusions

What we have:

- Embedding of Julia is mostly understood.
- We have half done C++ library for embedding Julia.

And what we don't:

- Any working program

Conclusions

What we have:

- Embedding of Julia is mostly understood.
- We have half done C++ library for embedding Julia.

And what we don't:

- Any working program

What should we do?

- Try to use embedded Julia for something!

Usage should guide requirement. Library writer without users is blind.