Interfacing Mathematical Computation Systems with Low-Level Libraries: a Survey of General Methods and Concrete Systems

Florian Rabe[1,2]

FAU Erlangen[1] and LRI Paris[2]

Abstract

1 Introduction and Related Work

2 Methods

Consider a high-level language H and a low-level language L. Our goal is to allow users to write H-programs in a way makes use of fast code in L. The latter may be a pre-existing library or co-developed by the same user.

We develop a set of criteria that allow classifying the various solutions.

2.1 Type: Relationship between High- and Low-Level

This criterion describes the relationship between H and L. Here we name these relationships from the perspective of the high-level language.

Extension: Calling L-Code from H Like every programming language, H must provide a set of built-in primitive types and functions. The simplest examples are the type of machine integers and the associated operations.

Usually, the implementation language of H already provides a systematic way of defining such primitive operations. If L is this implementation language, it is straightforward to expose this to H-users. This is often called *foreign function interface*.

Instantiating the foreign function interface may be as easy as supplying as declaring a fresh H-identifier h and annotating it with an existing L-identifier l. This is often called an H-*binding* for l. Given an L-library, it is then straightforward to write such H-declarations for all L-primitives that are to be exposed.
[1] EdN:1

[1] EdNote: drawbacks: writing the bindings can be cumbersome; difficult to instantiate polymorphic L-types with H-types

Reification: Operating on H-Code in L If an implementation (compiler or interpreter) of H in L is available, one can treat H-code as L-data. H-declarations and objects h can be built and manipulated as L-objects for the syntax tree of h. Moreover, a convenience function can be provided that takes h as an H-string and parses it into its syntax tree. A dual function, often called *evaluation* takes such a syntax tree and returns the result of running the H-code. These functions are often already part of an L-library that can be easily integrated into L-projects.

This can be used in two ways. Most obviously, one can call H-code from L: this just requires building the expression to evaluate and then calling the evaluation function.

But more importantly for our interest, it can also be used to use L-code in H: Here we build the L-object l representing the desired H-object and then writing H-code that links against l. Crucially, we are free to use objects l that are not the semantics of any object expressible in L-syntax.

Translation: Compiling H-Code to L If H-code is compiled to a target into which we can also compile L-code, it is possible allow L-snippets in H-code. When compiling the H-code, the L-snippets are compiled via different chain and spliced into the results. In the simplest case, H is compiled into L itself, in which case the L-snippets can be retained without changed.

This may require the user to be familiar with the way in which H is compiled in order to correctly refer to H-identifiers in the L-snippets.

2.2 Directionality of Function Calls

This criterion describes the direction of in which data flows between H and L.

Unidirectional In the simplest interfaces, H calls L-functions and L returns results. This requires

- mapping H-functions to L-functions
- translating H-objects (the function arguments) to L
- translating L-objects (the result) to H

The maps of H-function to L-functions can often be restricted to a fixed set of mappings of named H-functions to L-functions. Often the corresponding L-function arises from a named L-function through a simple operation, whose overlap in constant in the size in the arguments. Typical examples are reordering or wrapping arguments.

Bidirectional Often a more complex kind of interface is are needed, where H calls L-functions, which may call back to H. Seemingly, this is already possible using the requirements of unidirectional interfaces: all we have to do is map the callback to an L-function. However, while unidirectional calls often need only a fixed set of named L-functions, the callback is typically a new function (anonymous or user-defined) that is not covered by the fixed set of mappings.

The options for mapping callbacks depend crucially on the interface type. For example, if H is translated to L anyway, the callback function is already available available as an L-function. Thus, the callback can be called directly from L without having to translate the arguments to H and the result back to L.

2.3 Level: Set of Types and Objects that can be shared

Efficient interfacing requires that translations between H- and L-types and objects can be done efficiently at the memory level. Ideally, the necessary translation is the identity function. In many situations a constant-time translation, e.g., by wrapping, is also acceptable. However, a constant-time translation of basic objects may cause linear-time overhead in the translation of complex objects, e.g., when every element of a list has to be wrapped in order to translate a list.

Machine Types At the very least the types and functions provided by the underlying hardware machine (which are usually part of all programming languages anyway) can be shared between L and H.

Certain Named Types and Functions It is desirable to share also complex types and operations. Often a fixed set of named types and functions is shared by individually creating bindings for them.

Arbitrary Expressions Ideally, arbitrary expressions can be shared, e.g., by using a shallow embedding of H and into L.

Here a *shallow* embedding of A in B translates A-entities to B-entities of the same kind, e.g., A-types are represented as B-types and A-exceptions as B-exceptions. That distinguishes shallow embeddings from deep ones such as the ones induced by reification, which are much easier obtain.

Polymorphic Types A particular subtlety is the handling of polymorphic types. There are three ways to handle a polymorphic L-types $T(a)$:

- **preservation**: T is mapped to a polymorphic type in L. This is most desirable if L-supports polymorphism.
- **elaboration**: T is mapped to a non-polymorphic H-entity that mimics polymorphism, e.g., an H-function that takes a type tag representing a and returns a new type.
- **restriction to ground instances**: No binding is generated for T itself. Instead, for every ground instance $T(A)$ that is needed throughout the course of an H-program, a monomorphic binding is generated.

2.4 Binding Generation Time

Often interfaces must generate bindings that allow sharing an L-type or object with H. For example, if L is the C language, bindings may be generated from C-header files. This may require running an auxiliary tool that may use an L-parser or L-interpreter to work with L-source files.

This criterion describes when these bindings are created relative to the H-development work flow.

Static Creation With static creation, bindings are generated once and for all independent of development in H. Consequently, binding generation is limited to a fixed set of H-entities.

Dynamic Creation With dynamic creation, bindings are generated at the run-time of the H-code. This allows sharing types that are only created at run time (e.g., H-strings that are dynamically interpreted as L-classes). But H-code acquires a dependency on the auxiliary tool.

3 Systems for Interfacing Python and C/C++

[2] EdN:2

3.1 Extension-Based

3.1.1 DragonFFI

Foreign Function Interface to transparently call C functions from Python

DragonFFI is a Python library that lets a Python program analyze on the fly the C-header file, or shared library with debugging information, and then transparently call the C functions of that library:

```
import pydffi
F = pydffi.FFI()
CU = F.cdef(”include <stdlib.h>”)
print(float(CU.funcs.atof(”4.5”)))
```

One can also compile additional code:

```
CU = F.compile(”__attribute__((ms_abi)) int foo(int a, int b) { return a*b;
int(CU.funcs.foo(4,−7))

 Not in stable version
CU=F.from_dwarf(”/path/to/libarchive/build_clang_full_relwithdeb/./libarchi
a = funcs.archive_read_new()
...
```

Implementation:

Goal: Avoid C ABI chaos.

Clang AST is too high-level (includes irrelevant info like source code locations, no information about type sizes, struct padding, etc...), LLVM IR is too low-level (i.e.: structure coersing, can make structures ”disappear) dragonffi uses LLVM-IR + DWARF debug info to generate wrappers automatically from a header file, or even from a shared library (not yet windows), generated by clang or gcc.

Dependencies: * python pip package (wheel): everything statically compiled * compilation by hand: Clang/LLVM 5.0 (see https://github.com/aguinet/dragonffi/llvm5-compilation)

[2] EdNote: FR: the structure in this section is good; the text is just some notes copied from the pad to get started

3.1.2 libffi

reference impl of various C ABIs) -¿ cffi (higher-level Python interface based on pycparser)

Problem: libffi is hard to work on (hand-written ASM), cffi is limited.

3.1.3 cppyy

generates Python bindings for C++ supports many language features including templates, inheritance, exceptions

requires compiling the sources with clang to do introspection on the binary

Integration Solutions: Other CAS ¡-¿ C/C++

CyPari: Pari ¡-¿ C

Strategy: The code of Pari comes endowed with a description of each function:

TODO: insert example here

From this code, cypari autogenerates the following Cython binding which is then compiled to C:

TODO: insert example here

TODO: blurb about types: Pari has a dozen types of objects; conversions between Pari types and Sage types is written by hand.

libsemigroups: GAP ¡-¿ C

libsemigroups is a C++11 library for computational semigroups. It was written with three goals in mind: accelerate core code from GAP's semigroup package (low level language, parallelism, ...), include a modernized existing C library Semigroupe (parallelism, design issues), and make the result available to a larger community (GAP,) Distribution: conda and github but no wheel (supposedly not well suited)

GAP bindings

TODO: summary, pros, cons

Original version: handwritten

Conversion overhead but still faster than original computation (unlike numpy approach)

Second attempt: mimic pybind11 using compile time introspection (presumably available in both clang and g++).

Python bindings

Handwritten using Cython; totally incomplete and painful to maintain

Potential plan for writing a C API, because many languages (clojure, ...) can directly call C code

Difficulties - ”old” compilers don't quite support C++ 11 (old but still current in conda, GAP tests, and other systems, at that point) - code duplication - *not* tested on windows; would presumably work on cygwin and windows subsystem but not natively (relies on automake/autoconf)

Julia ¡-¿ C

Sebastian and Thomas)

Features: - Implemented as a GAP package JuliaInterface - works on top of vanilla GAP and Julia (some improvements will be submitted upstream GAP for tighter integration) - conversions between GAP and Julia objects - Manipulate Julia objects from GAP via handles - Manipulate GAP objects from Julia via handles - Run arbitrary Julia functions from GAP by name - Evaluate arbitrary

Julia commands as strings - Application: load Julia code that calls back to GAP; this can lead to 20x speed improvements, presumably due to compilation + use of native Julia data structures (lists, ...) that are faster than GAP's.

Implementation: - GAP and Julia cohabit in the same memory space, and interact through their C-API. - Currently each system uses its own garbage collector, with GAP keeping a list of Julia objects; this leads to loops, and therefore memory leaks. - The next step is to add a compilation option to GAP to make it optionally use Julia's garbage collector instead of it's own. Same for HPC-GAP? - Same strategy as cypari, ...: stay lazy and convert objects only if really needed

Demo of calling functionalities of Nemo, Arb, ...

Difficulties: - Cost of conversions; for example for big integers: the representation in gap isn't the same as in Julia =¿ costly translation (sometimes carried through strings!), question is when to do it? - Julia currently recompiles all loaded code (e.g. the interface to Singular) on the fly. This is slow and a big burden in practice. The Julia folks are working on caching.

- Would it make sense to - How comes that Singular is so slow to start (compared to launching a plain Singular): - Semantic handles: thinking about it; not clear best way to proceed

PolyMake (Perl) ¡-¿ C++

PolyMake is a combination of C++ code (for hard core calculation) and Perl code (for high level mathematical features, with a rule-based mechanism). It also includes many external C or C++ libraries. C++ and the Perl interpreter share the same memory space. In general it's mostly Perl calling C++ (with features like automatic instantiation of templates). But its also possible to call PolyMake from C++ (including all the rule-based Perl mechanism).

Mechanism to generate the Perl bindings to the C++ code? Kinda similar pybind11 And a declaration mechanism similar to Cython pxd's

The rule-based mechanism is essentially a method selection mechanism, filling a similar purpose to GAP's method selection mechanism (or Sage's categories). It has different features though; possibly more powerful. filling a similar purpose to GAP's method selection mechanism (or Sage's categories). It has different features though; possibly more powerful.

Various other presentations from the workshop

SageMath

Overview

- A general purpose Python based software for mathematics - Tens of libraries interfaced

For most libraries, users don't actually see the underlying software; they use SageMath objects, and it just turns out that some of them use internally data structures or functions from the library.

Exceptions: for Pari / GAP the interface gives full access to all the commands of the library.

Strategies

For most C/C++ libraries, SageMath uses Cython to write bindings.

Drawbacks:

This typically requires lots of boilerplate. Typically writing a Cython header file which is duplicating the library header file (or whatever subset is used). TODO: include here the example from Jeroen's talk. Also, for C++ libraries, something to be done for each method / ...

xtensor: The Lazy Tensor Algebra Expression System

Many scientific languages wrapping same core data structures (nd-array + data frames (e.g. Python with numpy&pandas, Julia with..., R with ...).

Goal: decouple the data storage from the computations, so that data could be in memory, on disk, in the cloud...

xtensor provides a lazy container with a semantic similar to Numpy arrays. However operations are not evaluated immediately. Instead they return lazy expressions that are only computed upon value access or assigning to a container (expression template inside)

xtensor can wrap numpy arrays, Julia arrays, R arrays ...

Bindings: - BLAS bindings for BLAS operations on xtensor expressions - fftw

Question: how does it relate to Apache Arrow

Upcoming: xframe: the analogue of xtensor for dataframes instead of arrays

xtensor cookie cutter to gather best practices in bundling native libraries and deploying them for julia / R / python when based on xtensor

Data analysis for High Energy Physics

Issues different from the typical HPC; the data is typically highly nested, and can't be flatten

Not new: splitting complex arrays into columns (Apache Arrow)

Sort of proxy: code holds empty arrays then fetches the columns JIT

New technique: Object Array mapping

Goal: translate user-written traditional object-oriented code into HPC-like array operations at runtime

Type checking through numba type extension system Extends Numba to generate the appropriate pointer arithmetics to reach the correct data when a specific type is used. Numba provides an extension mechanism to control the emitted LLVM bytecode for custom types.

xeus-cling

xeus: framework for building Jupyter kernels cling: C++ interpreter

A cool demo of interactive C++, with widgets, images, sound, interactive map drawing: https://beta.mybinder.org/v2/gh/QuantStack/xeus-cling/0.4.2?filepath=notebooks/xcpp.ipy

provide the conversion wrapper for pybind11 for their various data types

3.2 Reification-Based

3.2.1 CPython

3.2.2 Boost-Python

3.2.3 PyBind11

evolution/colmpetitor of Boost-Python

3.3 Translation-Based

3.3.1 Cython

Python compiler

3.3.2 Numba

JIT Python compiler tailored for fast numerical code (number crunching, Numpy arrays, ...). It runs on CPython and can compile to CPU (x86, x86-64) / GPU (NVidia). It's backed by llvm, and benefits from its optimizations. It takes as input a Python function (typically through a decorator), and works from its AST:

TODO: example

3.3.3 Pythran

Static compiler for numerical code (similar to numba), with a focus on high level code (vectorized, numpy style). Type annotation for functions can (must?) be provided in comments above.

Dependencies: a C++11 compliant compiler, numpy, networkx

Choices and consequences: Take from Serge's talk

4 Other Systems that Interface with C/C++

[3] EdN:3

4.1 Pari: CyPari

Strategy: The code of Pari comes endowed with a description of each function:

TODO: insert example here

From this code, cypari autogenerates the following Cython binding which is then compiled to C:

TODO: insert example here

TODO: blurb about types: Pari has a dozen types of objects; conversions between Pari types and Sage types is written by hand.

4.2 GAP: libsemigroups

libsemigroups is a C++11 library for computational semigroups. It was written with three goals in mind: accelerate core code from GAP's semigroup package (low level language, parallelism, ...), include a modernized existing C library Semigroupe (parallelism, design issues), and make the result available to a larger community (GAP,) Distribution: conda and github but no wheel (supposedly not well suited)

GAP bindings TODO: summary, pros, cons

Original version: handwritten

Conversion overhead but still faster than original computation (unlike numpy approach)

Second attempt: mimic pybind11 using compile time introspection (presumably available in both clang and g++).

[3]EdNote: FR: the structure in this section is good; the text is just some notes copied from the pad to get started

Python bindings Handwritten using Cython; totally incomplete and painful to maintain

Potential plan for writing a C API, because many languages (clojure, ...) can directly call C code

Difficulties - ”old” compilers don’t quite support C++ 11 (old but still current in conda, GAP tests, and other systems, at that point) - code duplication - *not* tested on windows; would presumably work on cygwin and windows subsystem but not natively (relies on automake/autoconf)

4.3 Julia

(Sebastian and Thomas)

Features: - Implemented as a GAP package JuliaInterface - works on top of vanilla GAP and Julia (some improvements will be submitted upstream GAP for tighter integration) - conversions between GAP and Julia objects - Manipulate Julia objects from GAP via handles - Manipulate GAP objects from Julia via handles - Run arbitrary Julia functions from GAP by name - Evaluate arbitrary Julia commands as strings - Application: load Julia code that calls back to GAP; this can lead to 20x speed improvements, presumably due to compilation + use of native Julia data structures (lists, ...) that are faster than GAP’s.

Implementation: - GAP and Julia cohabit in the same memory space, and interact through their C-API. - Currently each system uses its own garbage collector, with GAP keeping a list of Julia objects; this leads to loops, and therefore memory leaks. - The next step is to add a compilation option to GAP to make it optionally use Julia’s garbage collector instead of it’s own. Same for HPC-GAP? - Same strategy as cypari, ...: stay lazy and convert objects only if really needed

Demo of calling functionalities of Nemo, Arb, ...

Difficulties: - Cost of conversions; for example for big integers: the representation in gap isn’t the same as in Julia =¿ costly translation (sometimes carried through strings!), question is when to do it? - Julia currently recompiles all loaded code (e.g. the interface to Singular) on the fly. This is slow and a big burden in practice. The Julia folks are working on caching.

- Would it make sense to - How comes that Singular is so slow to start (compared to launching a plain Singular): - Semantic handles: thinking about it; not clear best way to proceed

4.4 PolyMake (Perl)

PolyMake uses Perl as H and C++ as L.

Perl is extended with entities defined in C++ by statically generating bindings. Polymorphic C-classes are handled by dynamically generating bindings for ground instances.

Extension with is not only used for efficiency and expressivity. Users also write parts of the library in L that benefit from the OO-style type hierarchies available in C++.

C++ and the Perl interpreter share the same memory space. The interaction is mostly Perl calling C++. But its also possible to call PolyMake from C++

(including all the rule-based Perl mechanism).[4] EdN:4

The rule-based mechanism is essentially a method selection mechanism, filling a similar purpose to GAP's method selection mechanism. Like in GAP, when determining a property of an object, one of multiple available methods is chosen based on conditions about the object. However, unlike in GAP, if these conditions are properties that have not been determined yet, they are determined through back-chaining. Instead of a simple priority-based choice between multiple applicable methods, PolyMake must choose between all applicable branches of the resulting search tree.

5 Conclusion and Future Work

[4]EdNote: How?