Language interoperability

Jim Pivarski

Presentation: will redirect to notebooks

Organization of this tutorial

1. Overview of languages
2. Moving functions across languages
3. Moving data across languages
4. Projects: form pairs or small groups and work on one of seven problems

The Big Three: a basis set of features

C/C++	Java/JVM	Python
compiles to machine code	compiles to VM; optimizes at runtime	interpreted bytecode; runtime type-checks

The Big Three: a basis set of features

C/C++	Java/JVM	Python
compiles to machine code	compiles to VM; optimizes at runtime	interpreted bytecode; runtime type-checks
fastest; reproducible performance	slow start and intermittent pauses, but relatively high throughput	quick to start, but terrible throughput

The Big Three: a basis set of features

C/C++	Java/JVM	Python
compiles to machine code	compiles to VM; optimizes at runtime	interpreted bytecode; runtime type-checks
fastest; reproducible performance	slow start and intermittent pauses, but relatively high throughput	quick to start, but terrible throughput
bare metal and high abstractions coexist	choice of language, some high, some low	generally only one way to do things

The Big Three: a basis set of features

C/C++	Java/JVM	Python
compiles to machine code	compiles to VM; optimizes at runtime	interpreted bytecode; runtime type-checks
fastest; reproducible performance	slow start and intermittent pauses, but relatively high throughput	quick to start, but terrible throughput
bare metal and high abstractions coexist	choice of language, some high, some low	generally only one way to do things
manual memory management (including smart pointers)	generational garbage collectors, tunable to improve throughput	simple reference counting garbage collector

The Big Three: a basis set of features

C/C++	Java/JVM	Python
compiles to machine code	compiles to VM; optimizes at runtime	interpreted bytecode; runtime type-checks
fastest; reproducible performance	slow start and intermittent pauses, but relatively high throughput	quick to start, but terrible throughput
bare metal and high abstractions coexist	choice of language, some high, some low	generally only one way to do things
manual memory management (including smart pointers)	generational garbage collectors, tunable to improve throughput	simple reference counting garbage collector
language superset of C, others through FFI	hard to break out of the VM	excellent interface to other languages

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).
- Go: like C++ without the history (cruft) and an emphasis on modern concurrency (coroutines, mostly).

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).
- Go: like C++ without the history (cruft) and an emphasis on modern concurrency (coroutines, mostly).
- .NET: multilingual ecosystem like the JVM that learned from Java's mistakes. Mostly Windows, though.

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).
- Go: like C++ without the history (cruft) and an emphasis on modern concurrency (coroutines, mostly).
- .NET: multilingual ecosystem like the JVM that learned from Java's mistakes. Mostly Windows, though.
- R: like Python but specialized for statistical analysis; has a huge library (hint: use rpy2 to run R from Python).

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).
- Go: like C++ without the history (cruft) and an emphasis on modern concurrency (coroutines, mostly).
- .NET: multilingual ecosystem like the JVM that learned from Java's mistakes. Mostly Windows, though.
- R: like Python but specialized for statistical analysis; has a huge library (hint: use rpy2 to run R from Python).
- Julia: like Python but JIT-compiles each function before use; intended for serious number-crunching (hint: see Numba).

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).
- Go: like C++ without the history (cruft) and an emphasis on modern concurrency (coroutines, mostly).
- .NET: multilingual ecosystem like the JVM that learned from Java's mistakes. Mostly Windows, though.
- R: like Python but specialized for statistical analysis; has a huge library (hint: use rpy2 to run R from Python).
- Julia: like Python but JIT-compiles each function before use; intended for serious number-crunching (hint: see Numba).
- Javascript: embedded in the GUI toolkit that absolutely everyone has installed on their laptops/phones/watches.

Other languages, notable for being *different* from these

- Rust: like C++ without the history (cruft) and a typesystem that tracks ownership (inescapable smart pointers).
- Go: like C++ without the history (cruft) and an emphasis on modern concurrency (coroutines, mostly).
- .NET: multilingual ecosystem like the JVM that learned from Java's mistakes. Mostly Windows, though.
- R: like Python but specialized for statistical analysis; has a huge library (hint: use rpy2 to run R from Python).
- Julia: like Python but JIT-compiles each function before use; intended for serious number-crunching (hint: see Numba).
- Javascript: embedded in the GUI toolkit that absolutely everyone has installed on their laptops/phones/watches.
- Haskell: too pure and beautiful to touch. It can only be sullied by our unclean hands.

Moving functions across languages

Degrees of cohabitation

Same process

C/C++, Java, and Python can live in the same process, single or multiple threads, calling each other's functions (with care).

Statically compiled or dynamically loaded.

Degrees of cohabitation

Same process

C/C++, Java, and Python can live in the same process, single or multiple threads, calling each other's functions (with care).

Statically compiled or dynamically loaded.

Same machine

It's safer and easier to use multiple processes, but now all communication is asynchronous.

Shared memory (fast and dangerous), pipes, localhost sockets, or even intermediate files (don't, though).

Degrees of cohabitation

Same process

C/C++, Java, and Python can live in the same process, single or multiple threads, calling each other's functions (with care).

Statically compiled or dynamically loaded.

Same machine

It's safer and easier to use multiple processes, but now all communication is asynchronous.

Shared memory (fast and dangerous), pipes, localhost sockets, or even intermediate files (don't, though).

Same planet

Communicating over a network allows for flexible scale-out, though at a cost of performance.

Raw sockets, ZeroMQ sockets, HTTP, message queuing systems.

Same process: static and dynamic

Python

Ships with a C API that is very good: nearly all high-performance libraries for Python use it.

Some high-level wrappers on the C API: Cython, Boost::Python, SWIG, TPython (ROOT).

Also has a builtin ctypes library to dynamically load .so files; a little-known gem. [ctypes.ipynb]

Same process: static and dynamic

Python

Ships with a C API that is very good: nearly all high-performance libraries for Python use it.

Some high-level wrappers on the C API: Cython, Boost::Python, SWIG, TPython (ROOT).

Also has a builtin ctypes library to dynamically load .so files; a little-known gem. [ctypes.ipynb]

Java/JVM

Ships with a Java Native Interface (JNI) to statically compile native functions into JAR files (Java shared libraries). Not often used in the Java community, prone to error.

But the Java Native Access (JNA) library is a good way to dynamically load .so files (native shared library). [jna.ipynb]

Shared object files: the least common denominator

Unless compiled with the DEBUG flag, .so files only contain function names and their bytecode.

- no number of arguments
- no argument types
- no return types

That's why you need .h files to compile a C program. :)

Shared object files: the least common denominator

Unless compiled with the DEBUG flag, .so files only contain function names and their bytecode.

- no number of arguments
- no argument types
- no return types

That's why you need .h files to compile a C program. :)

Worse still, most of the C++ concepts that extend beyond C are either encoded in mangled names or not at all.

Shared object files: the least common denominator

Unless compiled with the DEBUG flag, .so files only contain function names and their bytecode.

- no number of arguments
- no argument types
- no return types

That's why you need .h files to compile a C program. :)

Worse still, most of the C++ concepts that extend beyond C are either encoded in mangled names or not at all.

All languages (including Javascript in Nodejs) can run C functions, given the type info, but very few can run C++. Notably,

- Python (through Cython)
- Julia (through Cxx.jl)
- R (through Rcpp)

Moving data across languages

The easy case: raw arrays

Nearly every language has some construct for a contiguous buffer of fixed-width values.

The easy case: raw arrays

Nearly every language has some construct for a contiguous buffer of fixed-width values.

Python

The builtin “array” module is not widely used, but Numpy is the basis for the whole scientific Python ecosystem, including all the machine learning libraries.

Numpy arrays can allocate their own buffers or wrap arbitrary pointers to view any data anywhere. [numpy.ipynb]

The easy case: raw arrays

Nearly every language has some construct for a contiguous buffer of fixed-width values.

Python

The builtin “array” module is not widely used, but Numpy is the basis for the whole scientific Python ecosystem, including all the machine learning libraries.

Numpy arrays can allocate their own buffers or wrap arbitrary pointers to view any data anywhere. [numpy.ipynb]

Java/JVM

Called “off-heap memory,” meaning data not accessible to the garbage collector, this was an unintended hole in the Java specification.

Widely used for performance (Spark, OpenHFT). [offheap.ipynb]

The hard case: arbitrary objects

Different languages have different type systems. When moving structured data across languages, some sort of translation is necessary.

- language A knows language B's conventions and converts
- bring everything down to the lowest common denominator
- define a common meta-typesystem

The hard case: arbitrary objects

Different languages have different type systems. When moving structured data across languages, some sort of translation is necessary.

- language A knows language B's conventions and converts
- bring everything down to the lowest common denominator
- define a common meta-typesystem

The least common denominator is C's structs.

Numpy handles this case as well. [structs.ipynb]

Stuck this here because it's the best place it fits

GPUs!

Communication between CPU and GPU is largely through arrays, possibly arrays of structs. Numpy is just general enough to cover this case.

[pycuda.ipynb]

Sharing more complex datatypes between languages

Transfer protocols such as Google Protobuf, Thrift, and Avro each defines a language-neutral typesystem with only the essentials (arbitrary-length lists, structs, unions, etc.). Dozens of languages know how to read and write these protocols.

A project to keep an eye on: Apache Arrow seeks to define a standard *in-memory* format so that R DataFrames, Pandas DataFrames, and Spark DataFrames can share a pointer to the same memory. (Hopefully ROOT's TDataFrame will follow suit!)

Projects

Form pairs or small groups to solve one

1. JNA has callbacks, pointers, and structs just like ctypes. Repeat the GSL root-finding example with JNA.
2. Do the "time" and "gsl_deriv_central" examples with Cython, rather than ctypes.
3. Numba compiles Python code into native bytecode so that it will run faster. But that also gives us a *native* function pointer to pass to C code. Open [numba.ipynb] and try it.
4. `multiprocessing.RawArray(ctypes.c_double, N)` creates an array of N doubles that can be shared among multiple processes (if you fork them as instances of `multiprocessing.Process`). Cast that memory as Numpy in the shared processes to do work in parallel.
5. If we have access to McMillan's KNLs, try using *numa_alloc_local(nbytes)* in "libnuma.so" to allocate Numpy arrays on MCDRAM.