How an Awkward Array/Julia bridge can introduce HEP to Julia

Jim Pivarski

Princeton University – IRIS-HEP

September 27, 2021

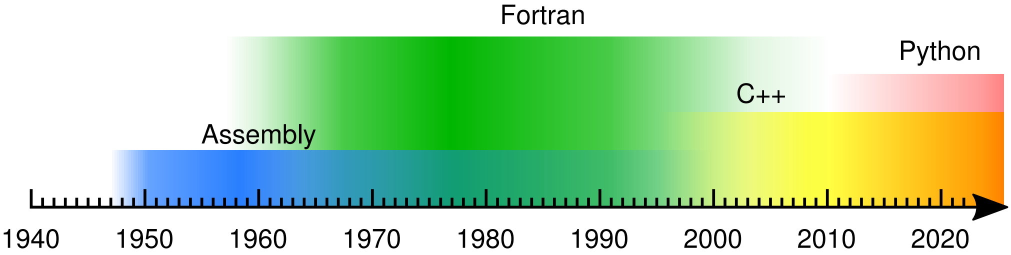
Language transitions

Physicists have to overcome an “activation energy” to switch programming languages (as anyone would).

Language transitions

Physicists have to overcome an "activation energy" to switch programming languages (as anyone would).

On a large scale, it has only happened a few times.



Language transitions

The benefits have to be major and palpable: not a laundry list of little features.

Language transitions

The benefits have to be major and palpable: not a laundry list of little features.

Assembly $\rightarrow$ Fortran: **readable math, hardware independence.**

Language transitions

The benefits have to be major and palpable: not a laundry list of little features.

Assembly $\rightarrow$ Fortran: **readable math, hardware independence.**

Fortran $\rightarrow$ C++: **nested data structures.**

"Data bank" libraries like ZEBRA and BOS addressed this problem in pre-Fortran 90, but with rough edges that **`struct`**/**`class`** avoid.

Language transitions

The benefits have to be major and palpable: not a laundry list of little features.

Assembly → Fortran: **readable math, hardware independence.**

Fortran → C++: **nested data structures.**

"Data bank" libraries like ZEBRA and BOS addressed this problem in pre-Fortran 90, but with rough edges that **`struct`**/**`class`** avoid.

C++ → Python: **dynamic interactivity, introspection.**

CINT, TClass, and now Cling address this problem in C++, but with rough edges that a "ground up" dynamic language avoids.

Language transitions

The benefits have to be major and palpable: not a laundry list of little features.

Assembly → Fortran: **readable math, hardware independence.**

Fortran → C++: **nested data structures.**

"Data bank" libraries like ZEBRA and BOS addressed this problem in pre-Fortran 90, but with rough edges that **`struct`**/**`class`** avoid.

C++ → Python: **dynamic interactivity, introspection.**

CINT, TClass, and now Cling address this problem in C++, but with rough edges that a "ground up" dynamic language avoids.

C++/Python mix → Julia: **built-in JIT? autodiff?**

Cling-in-Python (PyROOT/cppyy) and Numba address JIT now; JAX addresses autodiff. Are the rough edges bad enough to drive physicists to a new language?

Numba compiles statically typed Python, but the Python language was not designed to be statically typed.

Numba compiles statically typed Python, but the Python language was not designed to be statically typed.

- Yes, Python has type annotations/mypy now, but the type granularity is mostly for correctness-checking, not compilation.

Numba compiles statically typed Python, but the Python language was not designed to be statically typed.

- Yes, Python has type annotations/mypy now, but the type granularity is mostly for correctness-checking, not compilation.

item type	`numpy.typing.NDArray[numpy.float64]`
item type, ndims	`Array{Float64, 3}` **(Julia)**
item type, ndims, stride layout	`numba.types.Array(3, numba.float64, "C")`
item type, ndims with lengths	`jax.ShapedArray((2, 3, 5), numpy.float64)`

Numba's rough edges

Numba compiles statically typed Python, but the Python language was not designed to be statically typed.

- Yes, Python has type annotations/mypy now, but the type granularity is mostly for correctness-checking, not compilation.

item type	`numpy.typing.NDArray[numpy.float64]`
item type, ndims	`Array{Float64, 3}` **(Julia)**
item type, ndims, stride layout	`numba.types.Array(3, numba.float64, "C")`
item type, ndims with lengths	`jax.ShapedArray((2, 3, 5), numpy.float64)`

- Adding fields to existing objects or changing an object's type are basic parts of the Python language, but can never be allowed in statically compiled Numba.

Numba's rough edges

Numba compiles statically typed Python, but the Python language was not designed to be statically typed.

- Yes, Python has type annotations/mypy now, but the type granularity is mostly for correctness-checking, not compilation.

item type	`numpy.typing.NDArray[numpy.float64]`
item type, ndims	`Array{Float64, 3}` **(Julia)**
item type, ndims, stride layout	`numba.types.Array(3, numba.float64, "C")`
item type, ndims with lengths	`jax.ShapedArray((2, 3, 5), numpy.float64)`

- Adding fields to existing objects or changing an object's type are basic parts of the Python language, but can never be allowed in statically compiled Numba.
- Any library that Numba doesn't recognize can't be used in its `@nb.jit` functions.

Numba's rough edges

Supported Python features

Apart from the Language part below, which applies to both object mode and nopython mode, this page only lists the features supported in nopython mode.

> **Warning**
>
> Numba behavior differs from Python semantics in some situations. We strongly advise reviewing Deviations from Python Semantics to become familiar with these differences.

Language

Constructs

Numba strives to support as much of the Python language as possible, but some language features are not available inside Numba-compiled functions. Below is a quick reference for the support level of Python constructs.

Supported constructs:

- conditional branch: `if .. elif .. else`
- loops: `while`, `for .. in`, `break`, `continue`

Supported NumPy features

One objective of Numba is having a seamless integration with NumPy. NumPy arrays provide an efficient storage method for homogeneous sets of data. NumPy dtypes provide type information useful when compiling, and the regular, structured storage of potentially large amounts of data in memory provides an ideal memory layout for code generation. Numba excels at generating code that executes on top of NumPy arrays.

NumPy support in Numba comes in many forms:

- Numba understands calls to NumPy ufuncs and is able to generate equivalent native code for many of them.
- NumPy arrays are directly supported in Numba. Access to Numpy arrays is very efficient, as indexing is lowered to direct memory accesses when possible.
- Numba is able to generate ufuncs and gufuncs. This means that it is possible to implement ufuncs and gufuncs within Python, getting speeds comparable to that of ufuncs/gufuncs implemented in C extension modules using the NumPy C API.

The following sections focus on the Numpy features supported in nopython mode, unless otherwise stated.

Awkward Array, Vector, and soon Hist (PR #293) in Numba

```python
@nb.jit                                              # input Awkward Arrays
def delta_r_matching(array_reco, array_gen, builder):
    for reco_event, gen_event in zip(array_reco, array_gen):
        builder.begin_list()                         # output Awkward Array
        for reco in reco_event:                      # nested list
            best_i = -1
            best_dr = -1.0
            for i, gen in enumerate(gen_event):      # nested list
                dr = reco.deltaR(gen)                # Vector!
                if best_i < 0 or dr < best_dr:
                    best_i = i
                    best_dr = dr
            if best_i < 0:
                builder.append(None)
            else:
                builder.append(gen_event[best_i])
        builder.end_list()
    return builder
```

Awkward Array/Numba interface is designed for quick excursions

1. No memory management: Awkward Arrays passed to `@nb.jit` functions as borrowed references and cannot be created in the `@nb.jit` function.

Awkward Array/Numba interface is designed for quick excursions

1. No memory management: Awkward Arrays passed to `@nb.jit` functions as borrowed references and cannot be created in the `@nb.jit` function.

2. Therefore, the `ak.*` functions can't be called in any `@nb.jit` functions. Only iteration (nested **`for`** loops) is allowed.

Awkward Array/Numba interface is designed for quick excursions

1. No memory management: Awkward Arrays passed to `@nb.jit` functions as borrowed references and cannot be created in the `@nb.jit` function.

2. Therefore, the `ak.*` functions can't be called in any `@nb.jit` functions. Only iteration (nested `for` loops) is allowed.

3. Runtime representation of every Awkward Array in `@nb.jit` is (roughly)

```
template <typename AwkwardNodeType>
struct AwkwardArrayView {
    size_t pos;              // nesting level (index in arrayptrs)
    size_t start, stop;      // view within this nesting level
    void** arrayptrs;        // pointers to actual array data
    void** sharedptrs;       // workaround for C++ memory management
    PyObject* pylookup;      // keep borrowed references in scope
};                           // total: 48 bytes
```

with type-specific code generated for each `AwkwardNodeType`.

Imagine doing the same thing in Julia, with the same scope

```
>>> from julia import Julia    # PyJulia
>>> jl = Julia(compiled_modules=False)
>>> jl.eval("""
... function delta_r_matching(array_reco, array_gen, builder)
...     for (reco_event, gen_event) in zip(array_reco, array_gen)
...         builder.begin_list()
...         for reco in reco_event
...             (best_i, best_dr) = (nothing, nothing)
...             for (i, gen) in enumerate(gen_event)
...                 dr = reco.deltaR(gen)
...                 if isnothing(best_i) || dr < best_dr
...                     (best_i, best_dr) = (i, dr)
...                 end
...             end
...             builder.append(isnothing(best_i) ? nothing : gen_event[best_i])
...         end
...         builder.end_list()
...     end
... end
... """)
>>> # array_reco and array_gen are Awkward Arrays
>>> builder = jl.delta_r_matching(array_reco, array_gen, ak.ArrayBuilder())
>>> result = builder.snapshot()
```

Fast iteration over Awkward Arrays in Julia

- would be a reasonably small-scope project (3 months?)
- would offer an alternative to Numba with the advantages of Julia
- would be an incentive for physicists to take quick excursions into Julia.

Perspective of a busy physicist:

My iterative algorithm really can't be columnar.

I could try to write a Numba function for it, but I keep running into huge type error messages for code that works in uncompiled Python.

Or I could try writing a Julia function. The syntax is a little different and needs to be in a separate file or string, but the error messages make more sense, and it always works if it's valid Julia code.

This Julia's not bad. I could do more of my analysis in it. I should pester Jim to rewrite Awkward Array in Julia...

Conclusion/proposal

1. Teach PyJulia to recognize Awkward Arrays (ArrayBuilders?) as arguments.

Conclusion/proposal

1. Teach PyJulia to recognize Awkward Arrays (ArrayBuilders?) as arguments.
2. Minimal runtime representation, e.g.

```
struct AwkwardArrayView{AwkwardNodeType}
  pos::UInt64                       # nesting level (index in arrayptrs)
  start::UInt64 stop::UInt64        # view within this nesting level
  arrayptrs::Ptr{Ptr{Cvoid}}        # to be cast with 'unsafe_wrap'
end                                 # total: 32 bytes
```

Conclusion/proposal

1. Teach PyJulia to recognize Awkward Arrays (ArrayBuilders?) as arguments.
2. Minimal runtime representation, e.g.

```julia
struct AwkwardArrayView{AwkwardNodeType}
  pos::UInt64                           # nesting level (index in arrayptrs)
  start::UInt64 stop::UInt64            # view within this nesting level
  arrayptrs::Ptr{Ptr{Cvoid}}            # to be cast with 'unsafe_wrap'
end                                     # total: 32 bytes
```

3. Generate `AwkwardNodeType`-dependent code *only for iteration.*

Conclusion / proposal

1. Teach PyJulia to recognize Awkward Arrays (ArrayBuilders?) as arguments.
2. Minimal runtime representation, e.g.
```
struct AwkwardArrayView{AwkwardNodeType}
  pos::UInt64                        # nesting level (index in arrayptrs)
  start::UInt64 stop::UInt64         # view within this nesting level
  arrayptrs::Ptr{Ptr{Cvoid}}         # to be cast with 'unsafe_wrap'
end                                  # total: 32 bytes
```
3. Generate `AwkwardNodeType`-dependent code *only for iteration.*
4. Let physicists choose between **Numba** (same as the surrounding language) and **Julia** (more consistent, powerful) for non-columnar algorithms.

Conclusion / proposal

1. Teach PyJulia to recognize Awkward Arrays (ArrayBuilders?) as arguments.
2. Minimal runtime representation, e.g.
```
struct AwkwardArrayView{AwkwardNodeType}
  pos::UInt64                          # nesting level (index in arrayptrs)
  start::UInt64 stop::UInt64           # view within this nesting level
  arrayptrs::Ptr{Ptr{Cvoid}}           # to be cast with 'unsafe_wrap'
end                                    # total: 32 bytes
```
3. Generate `AwkwardNodeType`-dependent code *only for iteration.*
4. Let physicists choose between **Numba** (same as the surrounding language) and **Julia** (more consistent, powerful) for non-columnar algorithms.
5. This may be a “gateway” to Julia: strongly motivated (speed), small up-front commitment; encourages physicists to use it more if they like it.

Conclusion/proposal

1. Teach PyJulia to recognize Awkward Arrays (ArrayBuilders?) as arguments.
2. Minimal runtime representation, e.g.

```
struct AwkwardArrayView{AwkwardNodeType}
  pos::UInt64                        # nesting level (index in arrayptrs)
  start::UInt64 stop::UInt64         # view within this nesting level
  arrayptrs::Ptr{Ptr{Cvoid}}         # to be cast with 'unsafe_wrap'
end                                  # total: 32 bytes
```

3. Generate `AwkwardNodeType`-dependent code *only for iteration.*
4. Let physicists choose between **Numba** (same as the surrounding language) and **Julia** (more consistent, powerful) for non-columnar algorithms.
5. This may be a "gateway" to Julia: strongly motivated (speed), small up-front commitment; encourages physicists to use it more if they like it.

Anyone interested?