: AwkwardForth
deserialization DSL + Awkward-Array + ;

Jim Pivarski, Ianna Osborne, Pratyush Das, David Lange, and Peter Elmer

Princeton University – IRIS-HEP

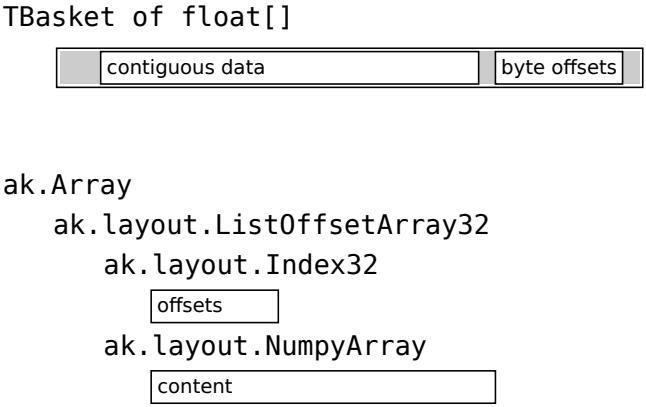
May 19, 2021

vCHEP 2021 contribution 4324131

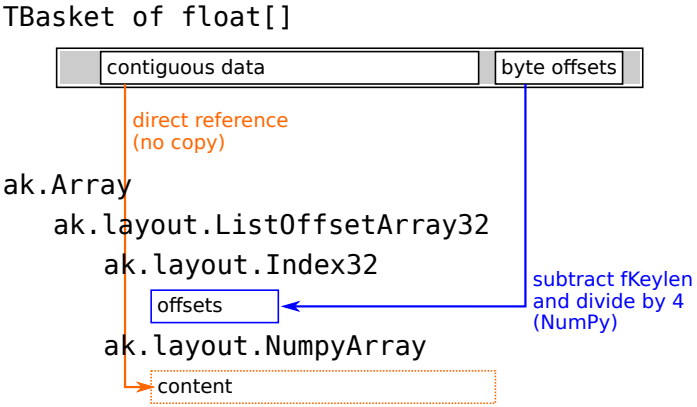
Deserializing *columnar* data can be very fast

`TBasket of float[]`

Deserializing *columnar* data can be very fast

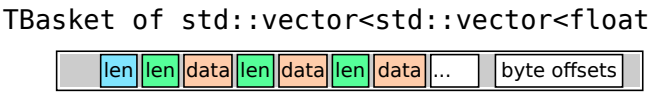


Deserializing *columnar* data can be very fast



"Deserialization" consists of $\mathcal{O}(1)$ metadata-only operations and $\mathcal{O}(n)$ vectorizable operations.

Deserializing *record-oriented* is more limited




```
ak.Array
   ak.layout.ListOffsetArray32
      ak.layout.Index32
         offsets 1
      ak.layout.ListOffsetArray32
         ak.layout.Index32
            offsets 2
         ak.layout.NumpyArray
            content
```

Record-oriented data, on the other hand, *must* be iterated sequentially, with control-flow decisions throughout.

For Uproot, this means that reading lists of lists of numbers (in Python) is 460× slower than reading numbers (NumPy cast).

The general problem of deserialization

1. Data types are not known until you examine the data's schema.
2. Knowing data types is essential for generating fast code.

The general problem of deserialization

1. Data types are not known until you examine the data's schema.
2. Knowing data types is essential for generating fast code.

Potential solutions

Generate deserialization code in a dynamic language, like Python.

Suitable for Uproot?

This is what Uproot does, and as we've seen, it's too slow.

The general problem of deserialization

1. Data types are not known until you examine the data's schema.
2. Knowing data types is essential for generating fast code.

Potential solutions

Generate deserialization code in a dynamic language, like Python.

Require a compilation step for every data file (like Protobuf).

Suitable for Uproot?

This is what Uproot does, and as we've seen, it's too slow.

That's a cumbersome workflow if you have several datasets.

The general problem of deserialization

1. Data types are not known until you examine the data's schema.
2. Knowing data types is essential for generating fast code.

Potential solutions	**Suitable for Uproot?**
Generate deserialization code in a dynamic language, like Python.	This is what Uproot does, and as we've seen, it's too slow.
Require a compilation step for every data file (like Protobuf).	That's a cumbersome workflow if you have several datasets.
JIT-compile the deserialization code (like ROOT, using Cling).	Adding LLVM as a dependency would undermine portability.

The general problem of deserialization

1. Data types are not known until you examine the data's schema.
2. Knowing data types is essential for generating fast code.

Potential solutions	**Suitable for Uproot?**
Generate deserialization code in a dynamic language, like Python.	This is what Uproot does, and as we've seen, it's too slow.
Require a compilation step for every data file (like Protobuf).	That's a cumbersome workflow if you have several datasets.
JIT-compile the deserialization code (like ROOT, using Cling).	Adding LLVM as a dependency would undermine portability.
Use a parser combinator library.	ROOT deserialization requires advanced language features.

The general problem of deserialization

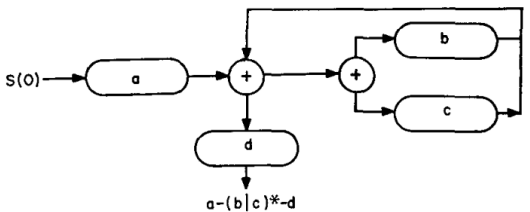
1. Data types are not known until you examine the data's schema.
2. Knowing data types is essential for generating fast code.

Potential solutions	Suitable for Uproot?
Generate deserialization code in a dynamic language, like Python.	This is what Uproot does, and as we've seen, it's too slow.
Require a compilation step for every data file (like Protobuf).	That's a cumbersome workflow if you have several datasets.
JIT-compile the deserialization code (like ROOT, using Cling).	Adding LLVM as a dependency would undermine portability.
Use a parser combinator library.	ROOT deserialization requires advanced language features.
Create a lightweight/specialized virtual machine.	That's the subject of this talk.

“Lightweight” virtual machines?

The most numerous virtual machines are not Java, VirtualBox, Xen, etc., but regex string-matching.

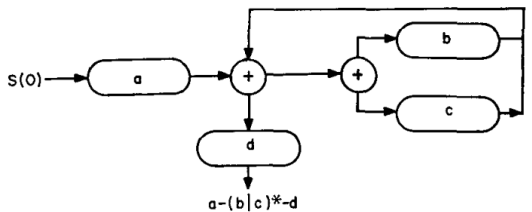
Ken Thompson, *Regular Expression Search Algorithm*, 1968.



“Lightweight” virtual machines?

The most numerous virtual machines are not Java, VirtualBox, Xen, etc., but regex string-matching.

Ken Thompson, *Regular Expression Search Algorithm*, 1968.

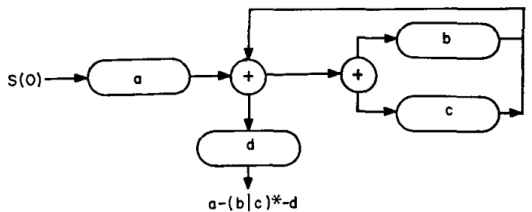


A regex string like `"a(b|c)*d"` gets compiled into a finite state machine for fast execution. Limiting the scope of the machine provides opportunities for optimization.

“Lightweight” virtual machines?

The most numerous virtual machines are not Java, VirtualBox, Xen, etc., but regex string-matching.

Ken Thompson, *Regular Expression Search Algorithm*, 1968.



A regex string like `"a(b|c)*d"` gets compiled into a finite state machine for fast execution. Limiting the scope of the machine provides opportunities for optimization.

Python’s `struct` module (bytestring-parsing) and `numexpr` (math) are similar.

Another issue: Uproot and Awkward Array must be independent

uproot

Specializes in ROOT file (de)serialization.

Python-only.

Awkward Array

Should know nothing about ROOT files.

Has a compiled C++ part.

Another issue: Uproot and Awkward Array must be independent

Specializes in ROOT file (de)serialization.

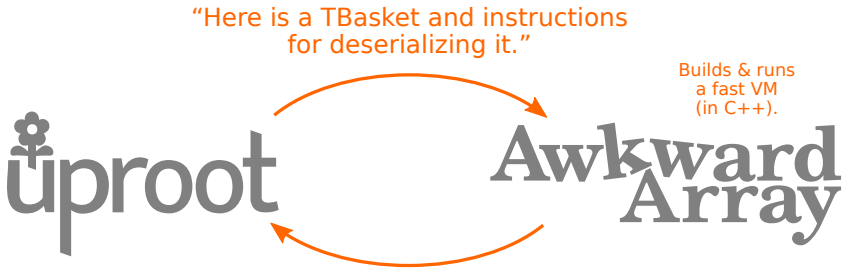
Python-only.

Should know nothing about ROOT files.

To build *fast* VMs.

Has a compiled C++ part.

Another issue: Uproot and Awkward Array must be independent



Specializes in ROOT file (de)serialization.

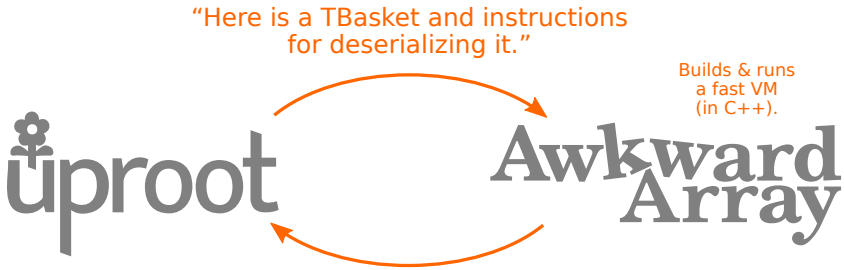
Python-only.

Should know nothing about ROOT files.

To build *fast* VMs.

Has a compiled C++ part.

Another issue: Uproot and Awkward Array must be independent



Specializes in ROOT file (de)serialization.

Python-only.

Should know nothing about ROOT files.

To build *fast* VMs.

Has a compiled C++ part.

Uproot needs a language to express how to deserialize a TBasket, but it doesn't need to be a human language.

So, how about Forth?

I heard about it through CollapseOS, which targets underpowered but ubiquitous hardware (Z80 chips).

So, how about Forth?

I heard about it through CollapseOS, which targets underpowered but ubiquitous hardware (Z80 chips).

Forth was invented in 1970 to control radio astronomy telescopes and was popular in early PCs because of the limited hardware.

- First Mac couldn't run a compiler; Forth was the only developer environment to make applications.

So, how about Forth?

I heard about it through CollapseOS, which targets underpowered but ubiquitous hardware (Z80 chips).

Forth was invented in 1970 to control radio astronomy telescopes and was popular in early PCs because of the limited hardware.

- First Mac couldn't run a compiler; Forth was the only developer environment to make applications.

Almost no grammar, easy to generate. The Postscript language is a Forth.

What does Forth look like?

```
: fibonacci    ( pops n -- pushes nth-fibonacci-number )
  dup
  1 > if
    1- dup 1- fibonacci
    swap fibonacci
    +
  then
;

( pushes [0 1 1 2 3 5 8 13 21 34 55 89 144 233 377] onto the stack )
15 0 do
  i fibonacci
loop
```

There's a global stack, words like "`1`" push a number on the stack, and words like "`>`" and "`if`" pop values off the stack, apply operations, and push the result.

Other than control flow like "`: ... ;`" and "`do ... loop`," it goes left to right.

AwkwardForth: a dialect of Forth with built-in parsing

Deserializing `std::vector<std::vector<float>>` from a ROOT TBasket:

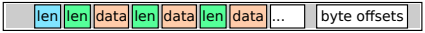
```
0 offsets0 <- stack             ( offsets start at zero )
0 offsets1 <- stack
0 offsets2 <- stack

begin
  byte_offsets i-> stack        ( get a position from the byte offsets )
  6 + data seek                 ( seek to it plus a 6-byte header )
  data !i-> stack               ( get the std::vector size )
  dup offsets0 +<- stack        ( add it to the offsets )
  0 do                          ( and use it as the loop counter )
    data !i-> stack             ( same for the inner std::vector )
    dup offsets1 +<- stack
    0 do
      data !i-> stack           ( and the innermost std::vector )
      dup offsets2 +<- stack
      data #!f-> content        ( finally, the floating point values )
    loop
  loop
again                           ( ends with a "seek beyond" exception )
```

Provides a way for Uproot to talk to Awkward Array

- Knowledge of ROOT I/O stays in Uproot.
- Uproot generates an AwkwardForth program as a string (loose coupling).
- Awkward Array builds and runs the machine to get an Awkward Array as output.
- No humans need to read or write the Forth code (except for debugging).

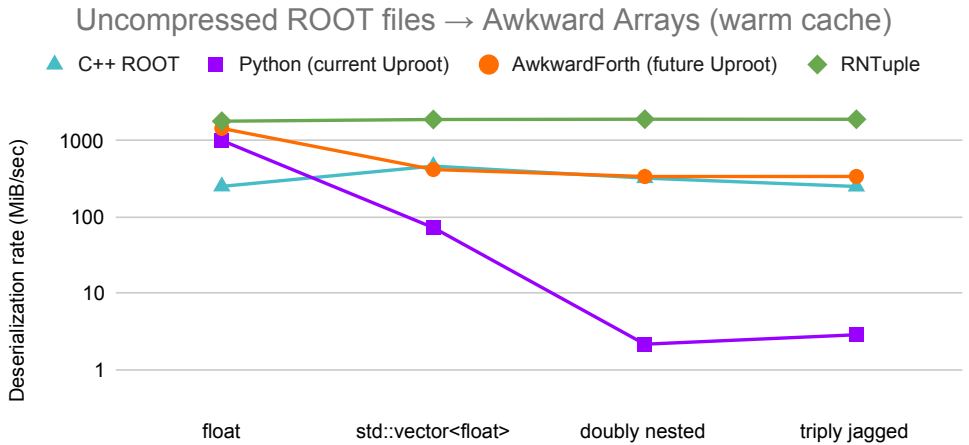
```
0 offsets0 <- stack            ( offsets start at zero )
0 offsets1 <- stack
0 offsets2 <- stack

begin
  byte_offsets i-> stack       ( get a position from the byte offsets )
  6 + data seek                ( seek to it plus a 6-byte header )
  data !i-> stack              ( get the std::vector size )
  dup offsets0 +<- stack       ( add it to the offsets )
  0 do                         ( and use it as the loop counter )
    data !i-> stack            ( same for the inner std::vector )
    dup offsets1 +<- stack
    0 do
      data !i-> stack          ( and the innermost std::vector )
      dup offsets2 +<- stack
      data #!f-> content       ( finally, the floating point values )
    loop
  loop
again                          ( ends with a "seek beyond" exception )
```

TBasket of std::vector<std::vector<float>>



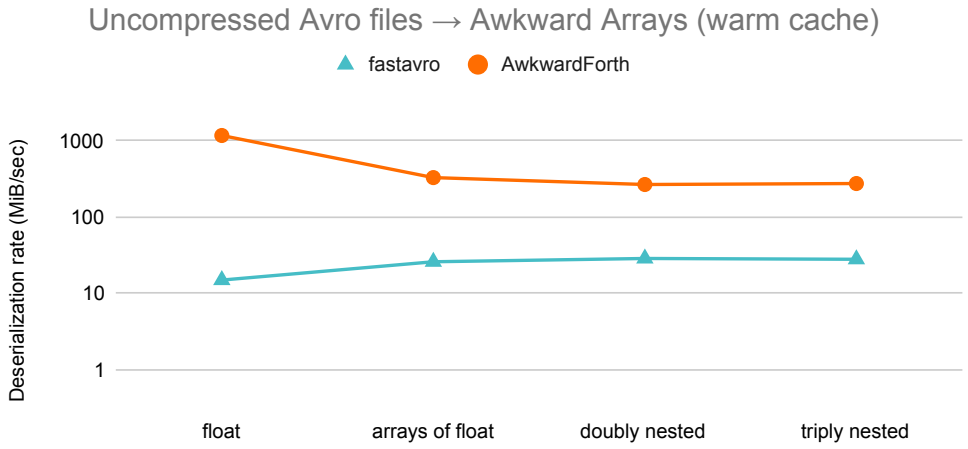

```
ak.Array
    ak.layout.ListOffsetArray32
        ak.layout.Index32
            offsets 1
        ak.layout.ListOffsetArray32
            ak.layout.Index32
                offsets 2
            ak.layout.NumpyArray
                content
```

Performance for ROOT deserialization (higher is better)



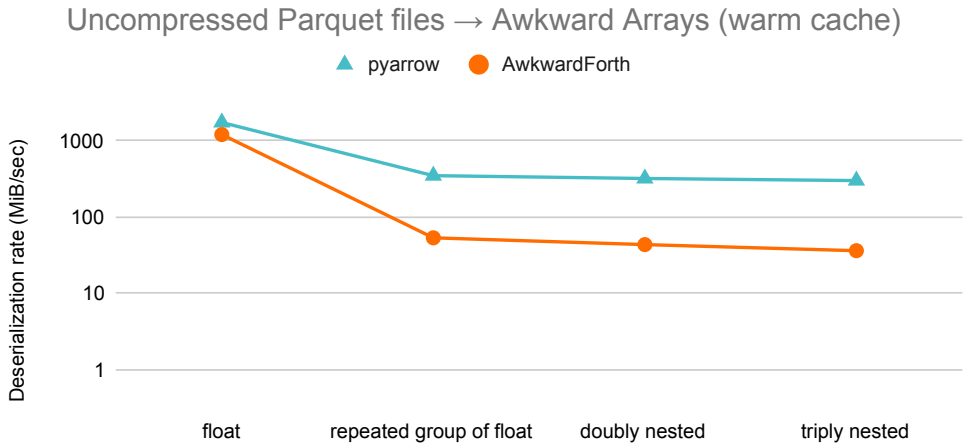
AwkwardForth is several times slower than compiled C++, but on par when data throughput is included. Compare also Python (current Uproot) and RNTuple.

Performance for Avro deserialization (higher is better)



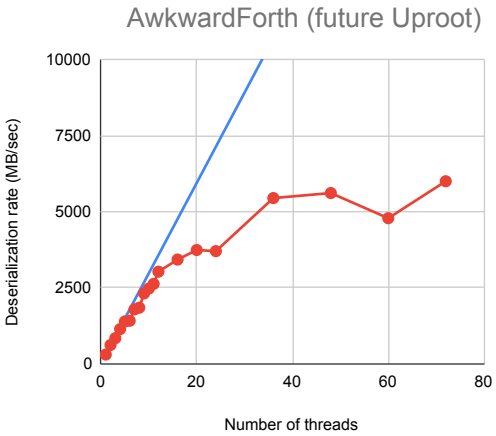
Since the language can handle any parsing problem, consider other formats like Avro. The `fastavro` library is C code, but doesn't know data types in advance.

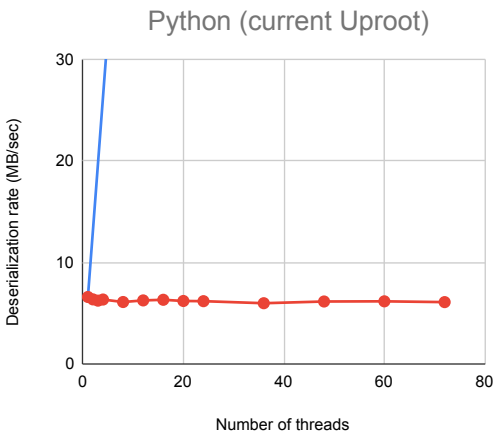
Performance for Parquet deserialization (higher is better)



Keep going: consider Parquet. Here, AwkwardForth doesn't do as well as `pyarrow`'s C++ parser because Parquet is a columnar format (like RNTuple).

Parallel processing performance of AwkwardForth and Python





AwkwardForth machines are lightweight: we can make one per thread. Python is inhibited by the GIL. Scales linearly up to RAM access ceiling (5 GB/sec).

Closing remarks

- See the paper for more details, including application to `TypedArrayBuilder`.

Closing remarks

- See the paper for more details, including application to `TypedArrayBuilder`.
- A complete Forth implementation is small ($<$ 5k lines of C++) and fast (5 ns per instruction).

Closing remarks

- See the paper for more details, including application to `TypedArrayBuilder`.
- A complete Forth implementation is small ($<$ 5k lines of C++) and fast (5 ns per instruction).
- Particularly useful for communicating algorithms between software libraries with restricted (sandboxed) runtimes.

Closing remarks

- See the paper for more details, including application to `TypedArrayBuilder`.
- A complete Forth implementation is small ($<$ 5k lines of C++) and fast (5 ns per instruction).
- Particularly useful for communicating algorithms between software libraries with restricted (sandboxed) runtimes.
- Uproot's Python-generating routines must now be supplemented by Forth-generating routines; targeting the end of this year.

Open IRIS-HEP fellow projects

This page lists a number of known software R&D projects of interest to IRIS-HEP researchers. (This page will be updated from time to time, so check back and reload to see if new projects have been added.) Contact the mentors for more information about any of these projects! Be sure you have read the guidelines.

- **Accelerating Uproot with AwkwardForth**: Uproot is a Python library that reads and writes ROOT files, the file format for nearly all particle physics data. (Over an exabyte of data is stored in the ROOT format.) As described in this talk, Uproot can only read data types that have a columnar layout *quickly*; data types with a record-oriented layout are hundreds of times slower. The same talk describes a solution: generating AwkwardForth code to read the data, rather than generating Python code to read the data, where AwkwardForth is a dialect of Forth, specialized for deserializing record-oriented data into columnar data. A successful candidate would add routines to generate AwkwardForth code in Python to deserialize C++ objects into Awkward Arrays—a very multilingual experience! The successful candidate would also monitor performance: adding these routines is expected to speed up deserialization of types like `std::vector<std::vector<float>>` by over 100× (see talk and the accompanying paper). (Contact(s): Jim Pivarski Ianna Osborne)