CRD

COMPUTATIONAL RESEARCH DIVISION

Recent Performance Results with PyPy

High-speed Python for data analysis

Wim Lavrijsen

ATLAS Software Week, 11/29-12/03-10

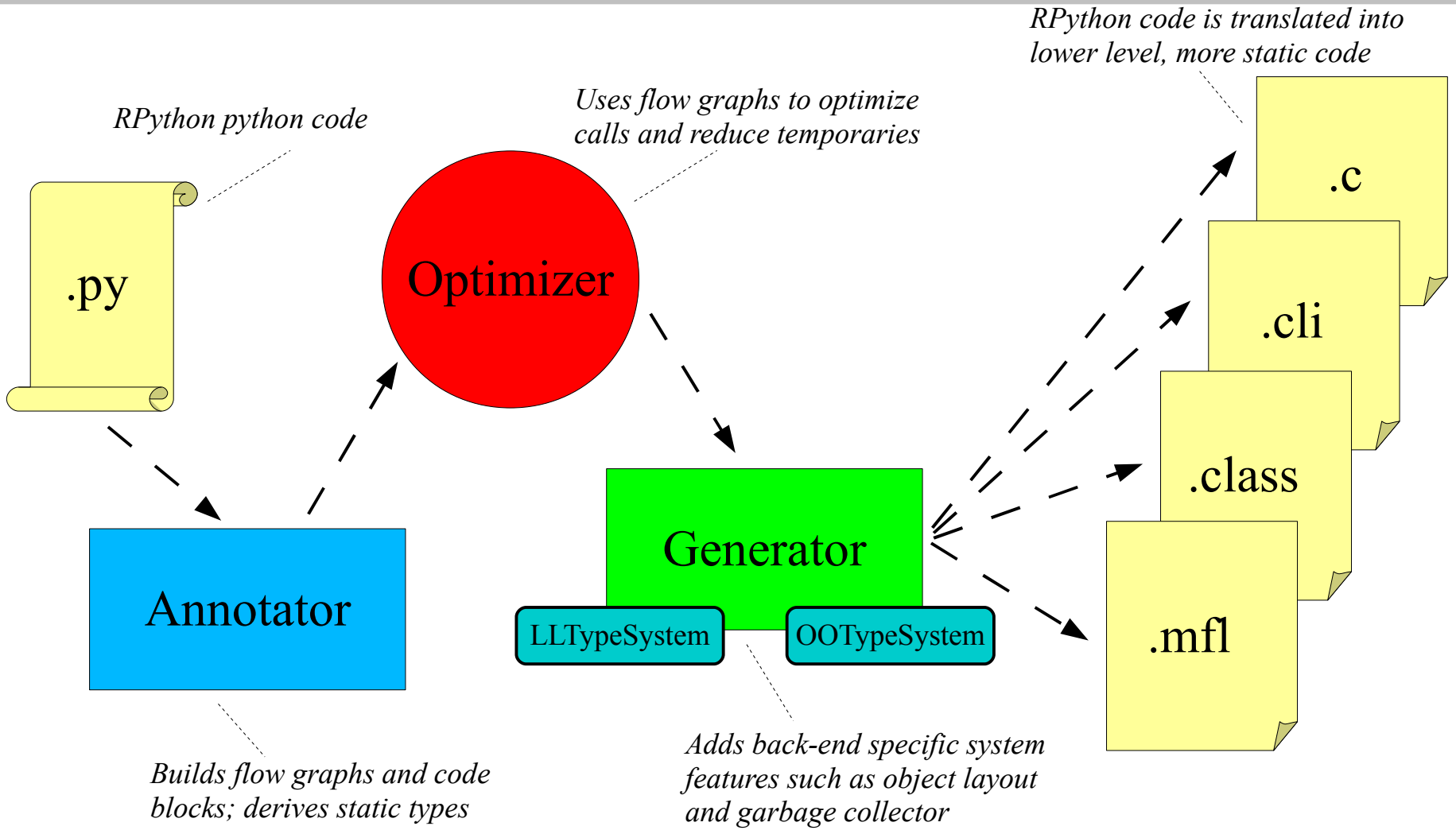
December 1, CERN

Office of Science
U.S. DEPARTMENT OF ENERGY

PyPy

- Dynamic language development framework
  - Framework is implemented in Python
  - One language thus developed is Python
    - CPython alternative
    - Makes it “Python written in Python”
- Translation tool-chain w/ several back-ends
  - E.g. Py → C to get pypy-c
- JIT generator as part of the toolchain
  - Operates on the interpreter level

PyPy Toolchain



Old: Python Optimization

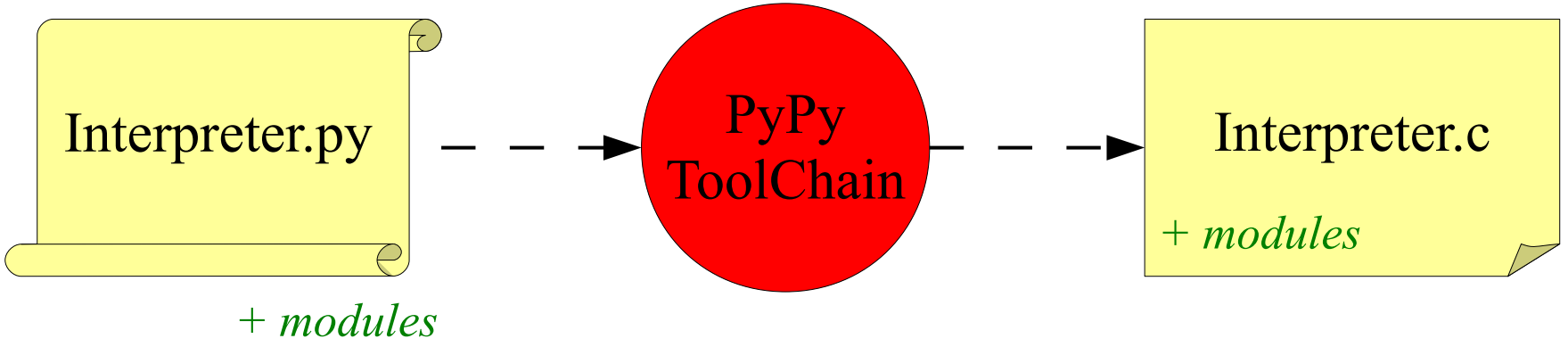
- Optimize algorithmic parts of analysis
  - Pull through PyPy toolchain → Compiled C
  - Several problems:
    - (Too) Restrictive in dynamic typing usage
    - Slow translation/compilation process
    - Difficult to understand error messages
      - Even several bugs in error reporting
- Turns out: no more productive than C++
  - Better approach now exists
  - Completely given up on this idea ...

New: Python Optimization

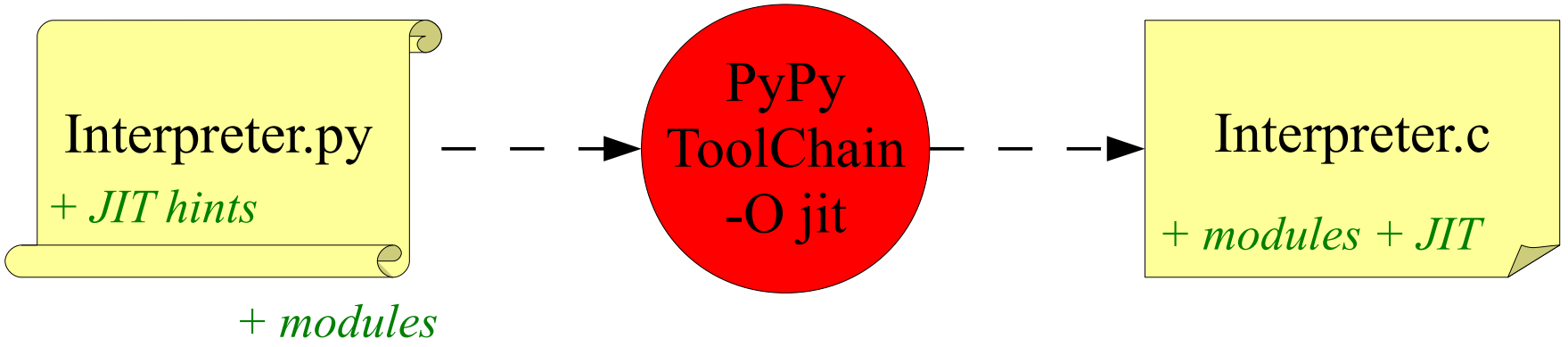
- Fully rely on PyPy's JIT
  - Implement Python/C++ at interpreter level
  - Supply JIT with hints for bindings
  - Integrate Reflex features directly
- Opens development path to the future
  - Support ROOT I/O at interpreter level
    - Automatic selection of best practices
  - New types of parallel processing
    - No need for GIL or manage it easily
    - Able to reuse memory between threads

Interpreter, JIT in PyPy

Without JIT:



With JIT:



PyPy's Generated JIT

- JIT applied on the interpreter level
  - Optimizes Interpreter.c for a given input (where input is the user source code)
  - Combines light-weight profiling and tracing JIT: especially effective for loopy code
- Can add core features at interpreter level
  - Provide hints to the JIT through JIT API
    - Including libffi type information
  - JIT developer deals with platform details
  - Completely transparent for end-user

Add Bindings to Python

With CPython:

+ *modules*
+ *Bindings.py*

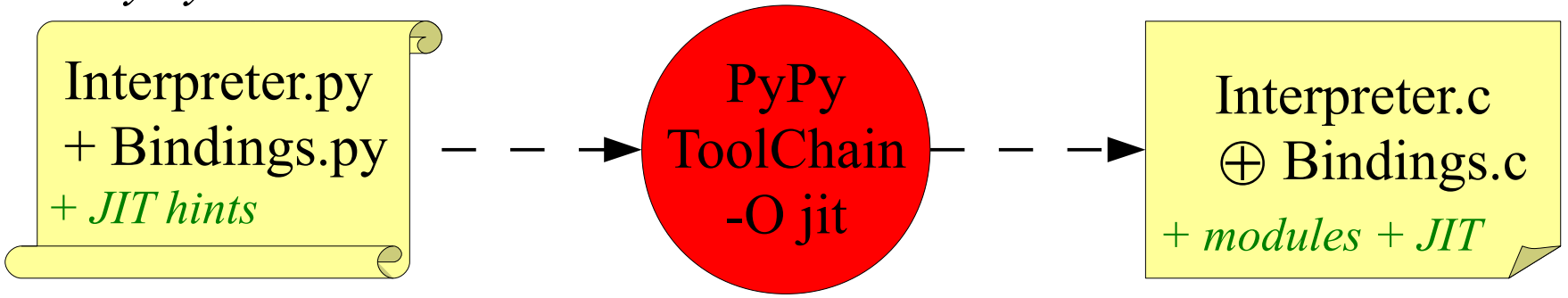
→ CPython is blind to the bindings

→ *No optimizations for GIL, types, direct calls, etc.*

→ PyPy+JIT know the bindings natively

→ *Full optimizations possible*

With PyPy:



+ *modules*

Current Work

- Reflex-support branch with module cppyy
  - Reflex-based bindings
    - Note the limitation of usefulness b/c of this
    - Best results with minor patch to Reflex
  - Limited feature-set, but growing quickly
  - Goal: data access and support for legacy
  - Get users to pypy-c b/c of speed-up, in particular for Python analysis codes
- Prototype available on afs soonish ...
  - /afs/cern.ch/....

Install from SVN

- SVN repository on PyPy server:
  - https://codespeak.net/viewvc/pypy
- Steps to check-out and install:

```
$ svn co http://codespeak.net/svn/pypy/branch/reflex-support pypy-reflex
$ cd pypy-reflex/pypy/translator/goal
$ python translate.py -O jit targetpypystandalone.py --withmod-cppyy
$ <setup or install ROOT; or an ATLAS release>
$ [opt: patch pypy-reflex/pypy/module/cppyy/genreflex-methptrgetter.patch]
```

- Result is `pypy-c` in work directory

Note PyPy is self-hosting, so for 2nd build can use:

```
$ pypy-c translate.py -O jit targetpypystandalone.py –withmod-cppyy
```

Prototype Usage

- Setup ATLAS release 16.3.0
  - For ROOT, compiler, etc.
- Start pypy-c executable, use like CPython

```
>>>> print “Hello World”
Hello World
>>>> import cppyy
>>>> cppyy.load_lib( “libMyClassDict.so” )
>>>> inst = cppyy.gbl.MyClass()
>>>> inst.MyFunc()
```

Current Results

- Benchmark measuring bindings overhead:
  - PyROOT: 48.6 (1000x)
  - PyCintex: 50.2 (1000x)
  - pypy-c-jit: 5.5 (110x)
  - pypy-c-jit-fp: 0.41 (8x)
  - pypy-c-jit-fp-py: 3.46 (70x)
  - C++: 0.05 (1x)

Notes:
1) "overhead" is the price to pay when calling a C++ function
2) bindings overhead matters less the larger the C++ function body
3) "-fp" is "fast path" and requires Reflex patch
4) "-py" is the pythonified (made python-looking) version, which still needs to be made JIT-friendly

(Near) Future Plans

- Build out functionality
  - Most needed is dealing with destructors
    - PyPy has a GC, so no dtor on scope-exit
- Support for CINT dictionaries
  - Probably easier to work with for end-users
  - But no fast-path options possible
    - Still a factor of 100 improvement expected
- Support for CLANG PCH
  - New direction taken by ROOT/CINT

Wanted

- Users willing to try out the prototype
  - Confirm improvement results
  - Measure memory requirements
  - Determine workability, set priorities
- From PAT: “candle analysis”
  - Both in Python and C++
    - Need to be realistic, need not be optimal
  - To offer a real-world benchmark
  - Work-bench for determining priorities