# Optimization of Python-bindings

Wim T.L.P. Lavrijsen, LBNL
ROOT Meeting, CERN; 01/15/10

# Introductory words ...



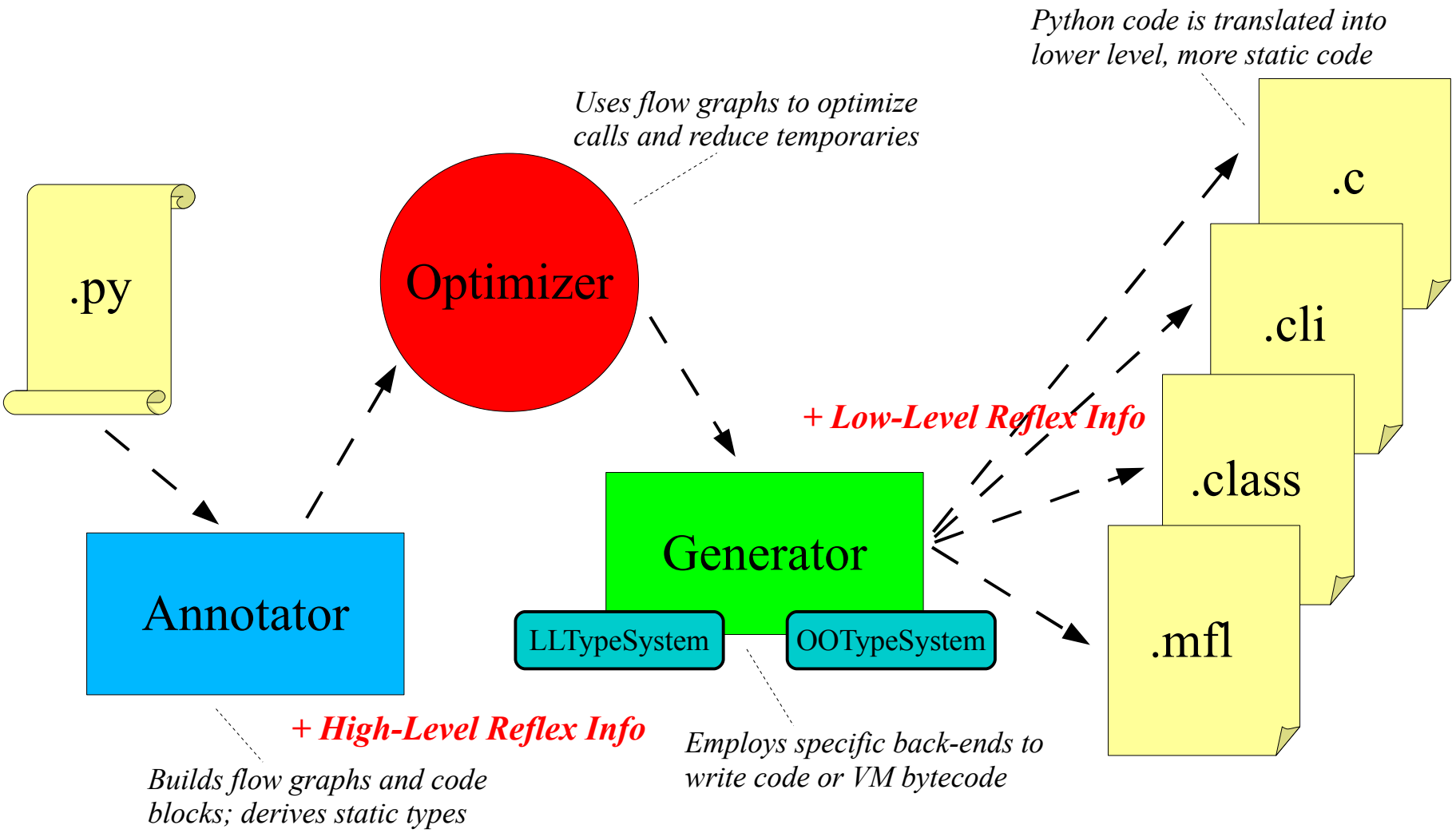
- More wish than plan
  - No clear idea of time expenditure this year
- Based on proposal hatched last Summer
  - Very low change of funding (< 1%)
  - Originally mainly looked to PyPy
    - Have prototype to play with
  - Unladen Swallow may be better direction
    - At least initially: less ambitious short term
    - No work done on it yet

# PyPy

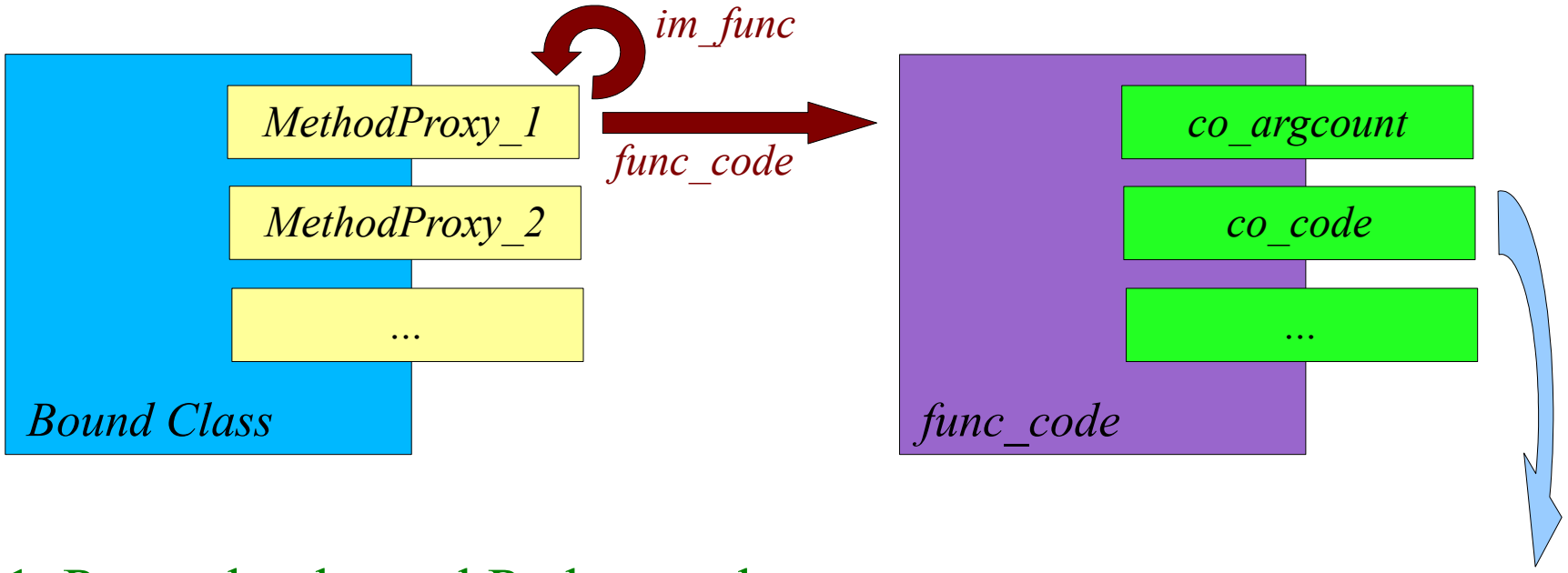


- "Python interpreter written in Python"
  - Makes Python code first-class objects
    - Allows for analysis and manipulation of code, including the full interpreter
- Full-fledged translation framework
  - Extensible with external types
  - Fully customizable with new back-ends
  - Transformations such as "stackless"
- Python as high-level description of intent
  - Target multi-core, Green Flash, etc.

# PyPy Architecture



# Feeding the Annotator



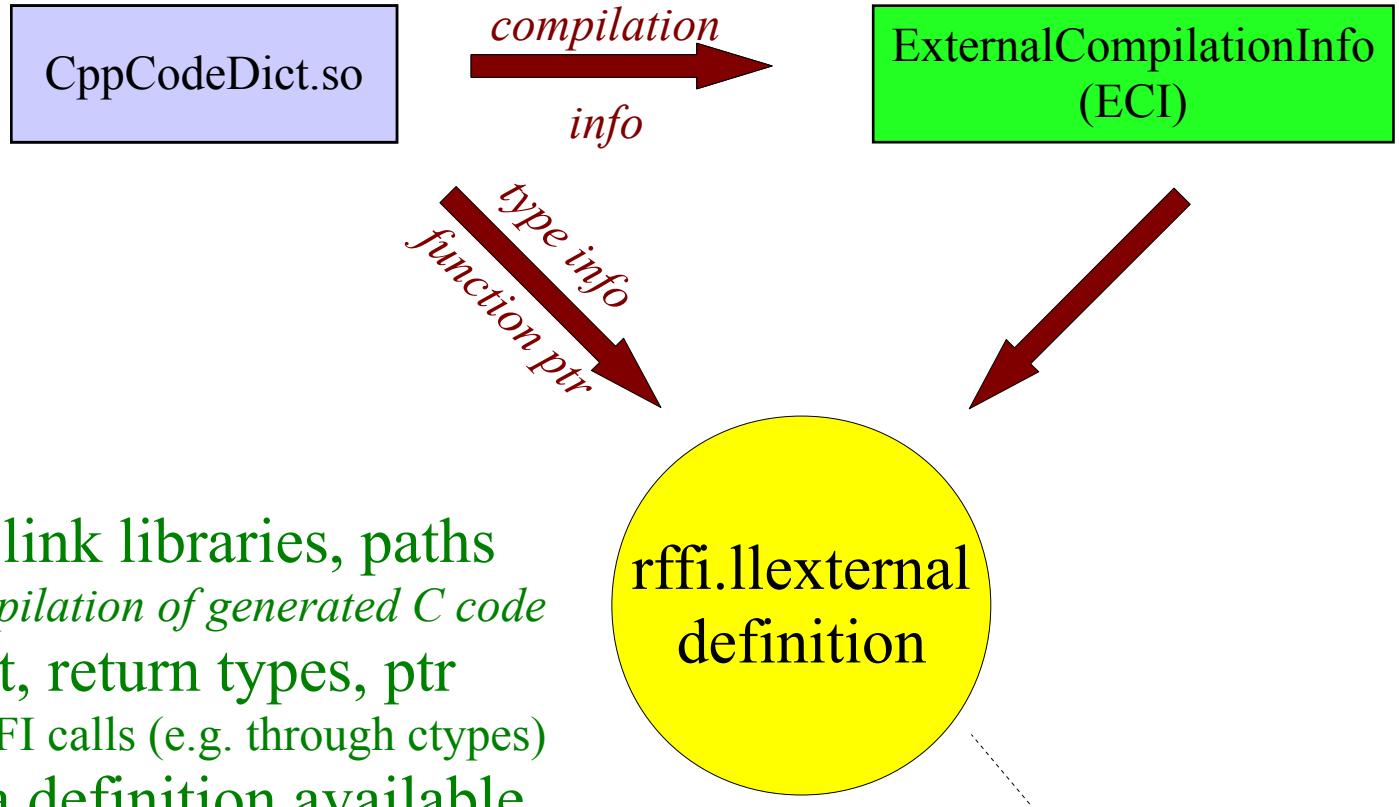

1. Pretend to be real Python code
2. Return emulated function
3. Construct appropriate bytecode
4. Allow annotator to analyse

*(all on-the-fly for minimal memory impact)*

Generated bytecode delivered:

```
def method_1( self, *args ):
    lvar = long(self)
    return ext_func( lvar, *args )
```

# Feeding the Generator




1. Get headers, link libraries, paths
   => *for compilation of generated C code*
2. Get argument, return types, ptr
   => for FFI calls (e.g. through ctypes)
3. Combine in a definition available to the generator.

# Unladen Swallow



- "Google-sponsored" project
  - Basically 2 Google engineers + OS bazaar
- Goal is to make Python 5x faster
  - Leverage LLVM and JIT technology
  - Get rid of Global Interpreter Lock
  - Remain compatible with CPython
- Three releases based on p2.6
- If CINT is LLVM, and Python is LLVM … :^)

# Bonus Material

# PyPy Annotation



- Start with known or given types
  - Constants and built-ins (known)
  - Function arguments (given [when called])
- Calculate flow graphs
  - Locate joint points
  - Consolidate code in blocks

} => **graph structure of possible flows and outcomes**

- Fill in all known types
  - Derived from initial known/given ones
  - Add information from dictionary

# Annotation Example



user

```
>>> def doFillMyHisto( h, val ):
...     x = ROOT.gRandom.Gaus() * val
...     return h.Fill( x )
...
```

behind the scenes

```
>>> t = Translation( doFillMyHisto )
>>> t.annotate( [ TH1F, int ] )
   -- type(h) is TH1F and type(val) is int
   -- type(x) is float, because:
        Gaus is TRandom::Gaus() which yields (C++)double
        mul( (python)float, int ) yields (python)float
   -- result is None, because:
        Fill is TH1F::Fill which yields (C++)void
```

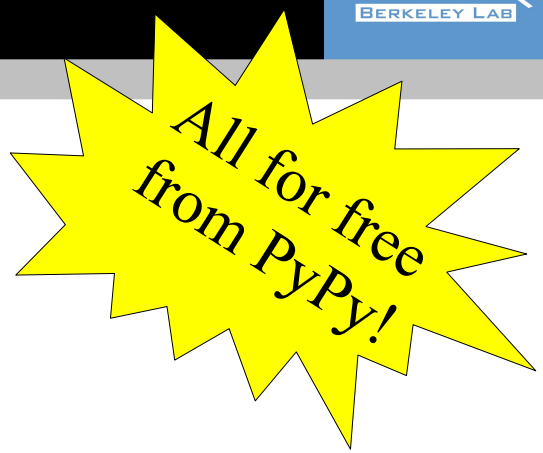
user

```
>>> doFillMyHisto = t.compile_c()
>>> h, val = TH1F('hpx','px',100,-4,4), 10
>>> doFillMyHisto( h, val )              # normal call
```

Explicitly in translation or at runtime; different (non-)choices can coexist

# PyPy Optimizer






- Function inlining
  - Equivalent to its C++ brother
  - May allow further optimizations
- Malloc removal
  - Use values rather than objects
    - Esp. for loop variables and iterators
  - Dict special case: remove method objects
- Escape analysis and stack allocation
  - Use stack for scope-lifetime objects

# PyPy Generation



- Two type systems
  - "Low Level" and "Object Oriented"

```
result = method( self, *args )
result = self.method( *args )
```

- Two kinds of back-ends
  - Code generation (e.g. C, JS, Lisp)
  - VM bytecodes (e.g. CLI/.Net and Java)
- Customizable
  - Covers cross-language calls (FFI)
  - Add information from dictionary

# Current Limitations



- PyPy is rather slow in run/use
  - Code mgmt needed of compiled functions
  - Develop/compile/run cycle unwanted anyway
- Loss of type information => loss of offsets
  - No virtual inheritance
  - No support of heterogeneous containers
  - In need of a solution ...
- No overloading resolution
  - Can be (partly) resolved with Python types
  - Complication of implicit conversions