PyPy: Dynamic Language Compilation Framework

Jim Huang (黃敬群) <jserv@0xlab.org>

Developer, 0xlab

June 18, 2012

Rights to copy

contact@0xlab.org

Corrections, suggestions, contributions and translations are welcome!

Latest update: June 18, 2012

creative commons
COMMONS DEED

Attribution – ShareAlike 3.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

Agenda

(1) Interpreters and Compilers

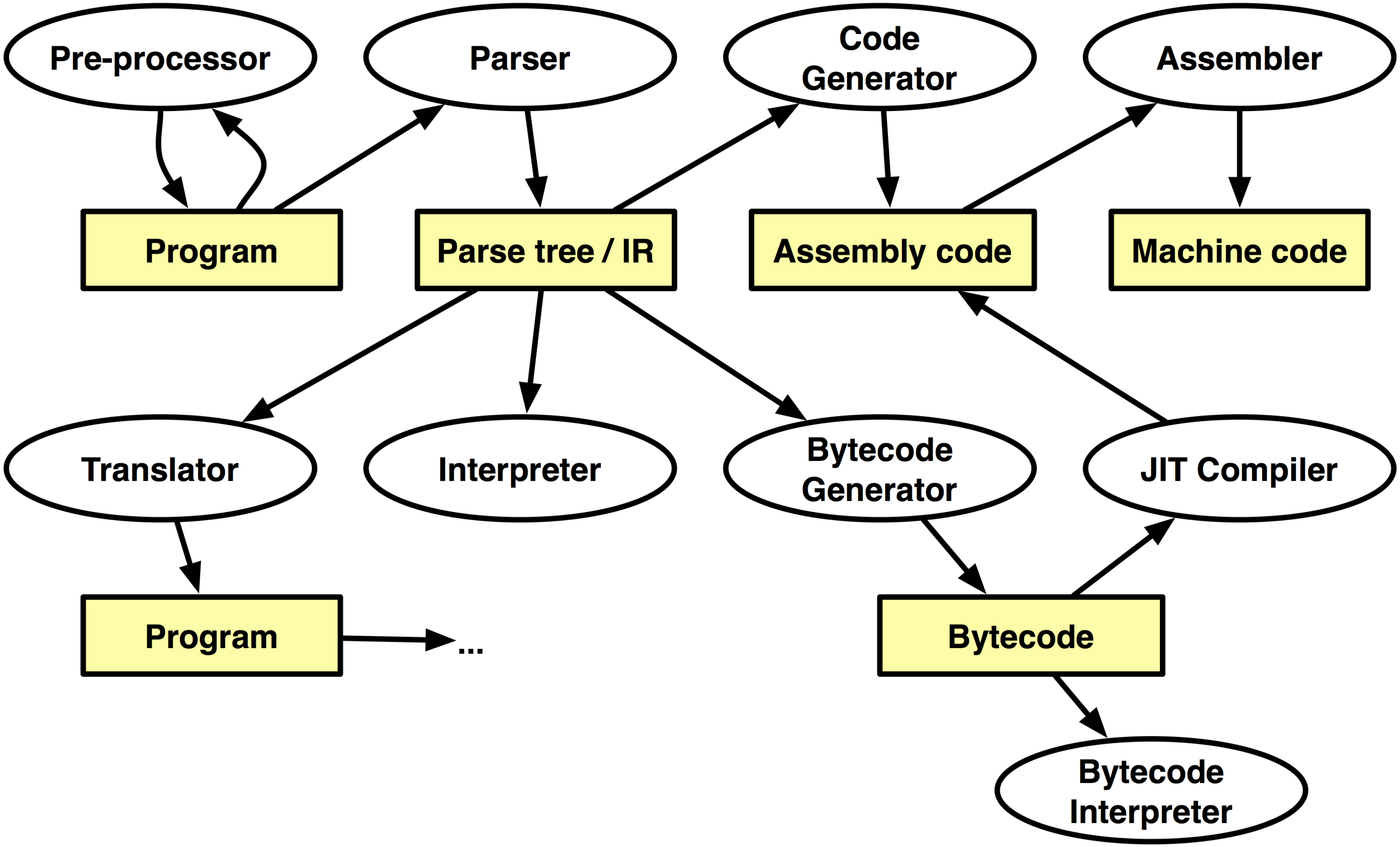
(2) PyPy components

(3) Translation Toolchain

A compiler was originally a program that "compiled" subroutines [a link-loader]. When in 1954 the combination "algebraic compiler" came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

– Bauer and Eickel [1975]

Compiler / Interpreter



Pre-processor

Parser

Code Generator

Assembler

Program

Parse tree / IR

Assembly code

Machine code

Translator

Interpreter

Bytecode Generator

JIT Compiler

Program

...

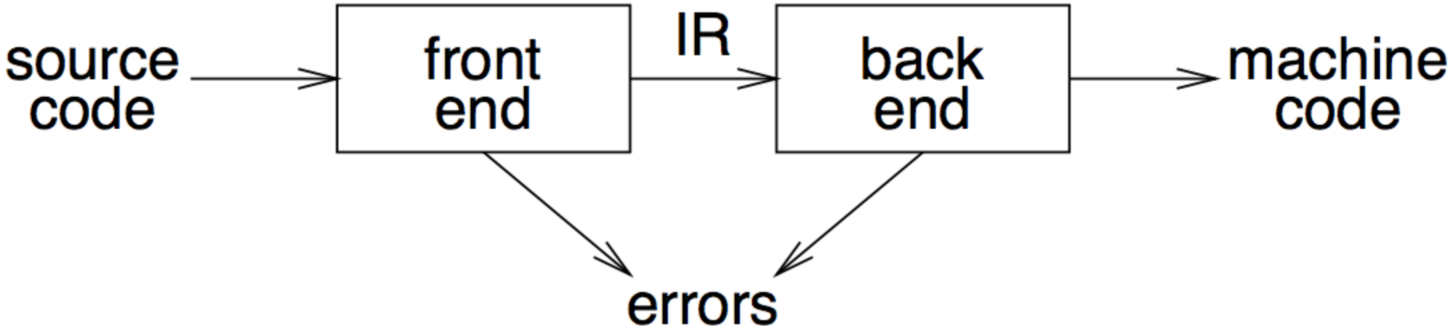
Bytecode

Bytecode Interpreter

Source: Compiler Construction, Prof. O. Nierstrasz

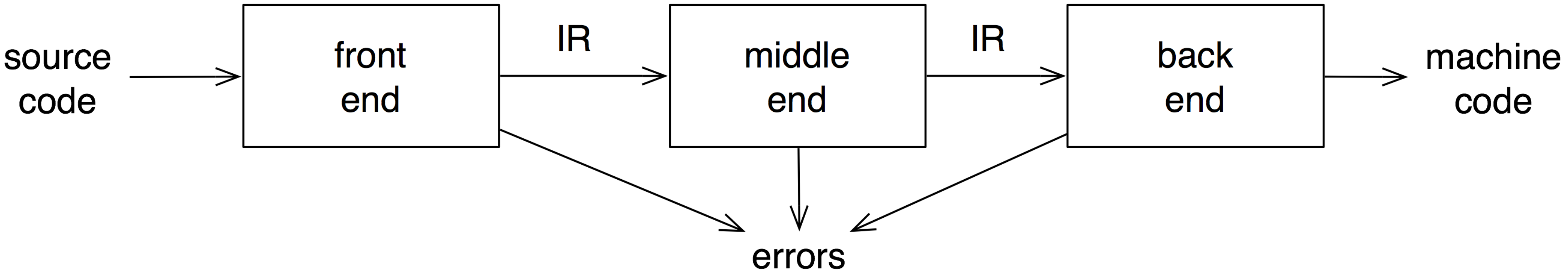
Traditional 2 pass compiler

- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes → better code



Traditional 3 pass compiler

- analyzes and changes IR
- goal is to reduce runtime
- must preserve values



source code

front end

IR

middle end

IR

back end

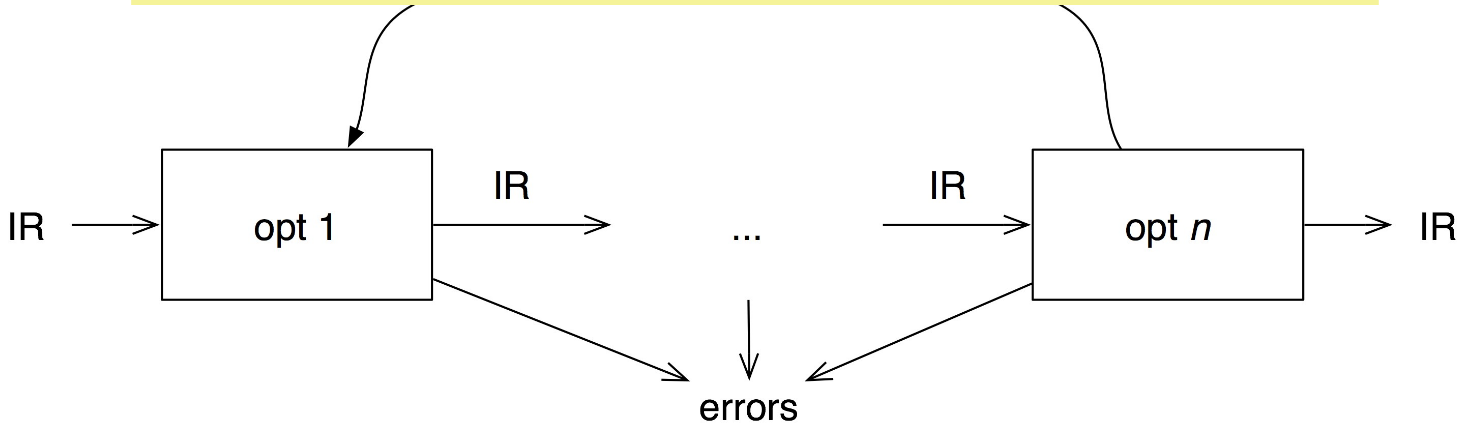
machine code

errors

Optimizer: middle end

- constant propagation and folding
- code motion
- reduction of operator strength
- common sub-expression elimination
- redundant store elimination
- dead code elimination

Modern optimizers are usually built as a set of passes



Optimization Challenges

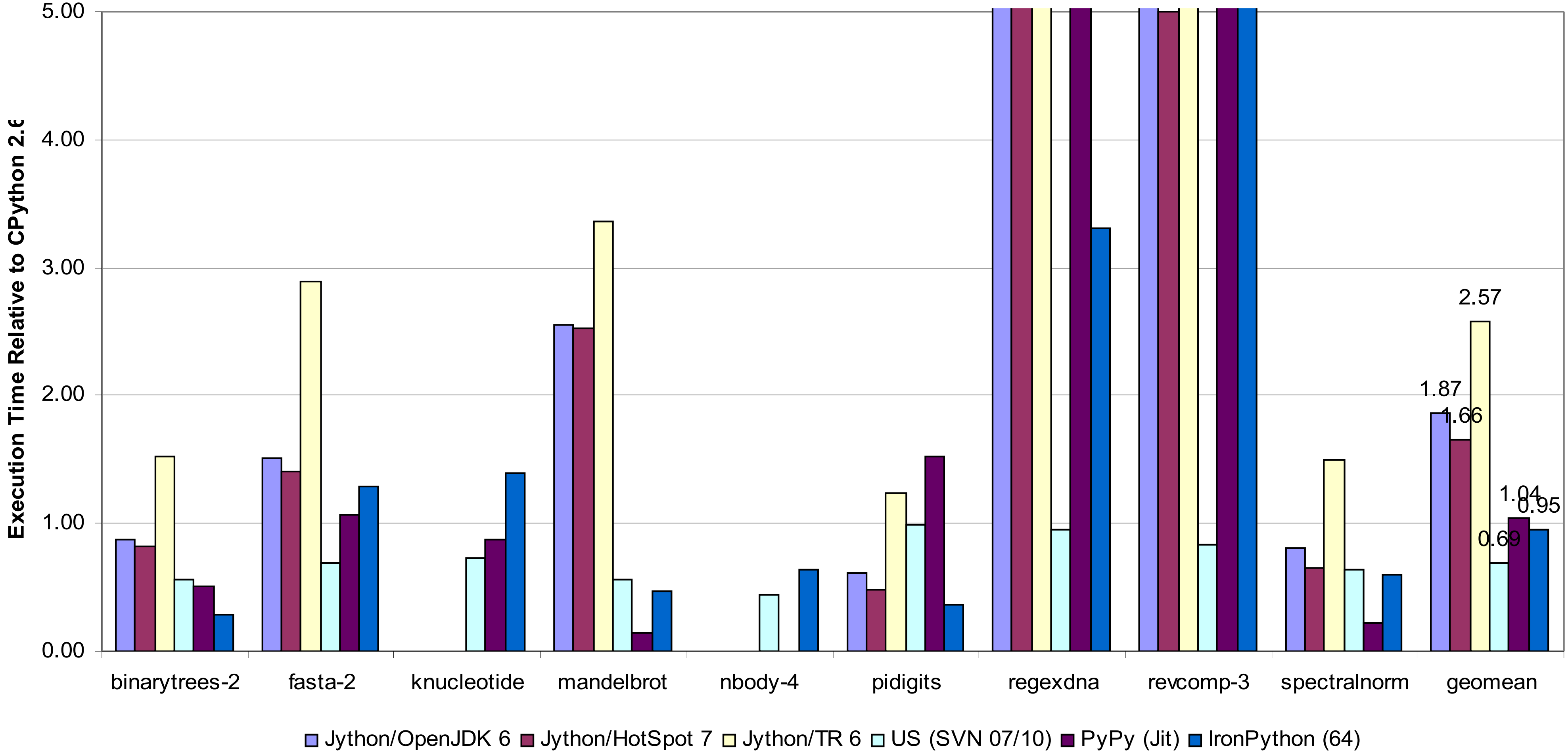
- Preserve language semantics
  - Reflection, Introspection, Eval
  - External APIs
- Interpreter consists of short sequences of code
  - Prevent global optimizations
  - Typically implemented as a stack machine
- Dynamic, imprecise type information
  - Variables can change type
  - Duck Typing: method works with any object that provides accessed interfaces
  - Monkey Patching: add members to “class” after initialization
- Memory management and concurrency
- Function calls through packing of operands in fat object

Python Compilers

- **Jython**: "Python over the JVM"; written in Java
  - Similar approaches: JRuby, Rhino, …
- **IronPython**: "Python over CLR/DLR"; written in C#
  - Open source effort led by Microsoft, Apache License
- **Unladen Swallow** compiler: "Extend the standard CPython interpreter with the LLVM JIT"
  - Open source effort led by Google,
  - Similar approaches: Rubinius, …
- **PyPy**: "Python on Python"
  - Open source effort (evolution of Psycho)
  - Tracing JIT; PYPY VM/JIT can target other languages

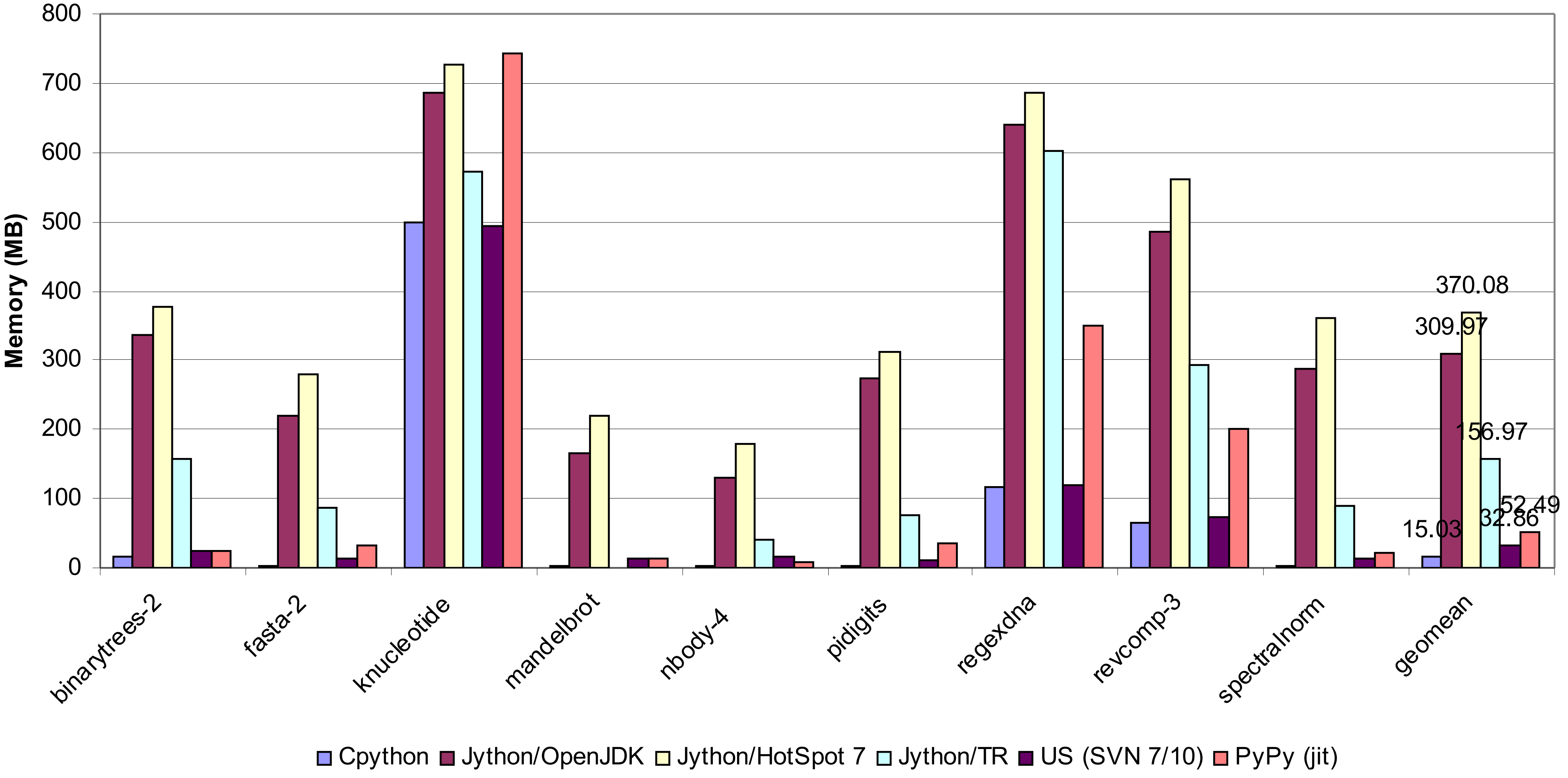
Python benchmark

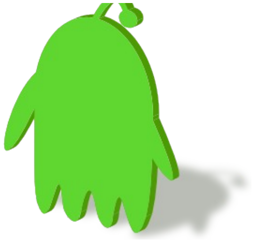
(IBM Research, 2010)



Memory Consumption: important for Parallelism

(IBM Research, 2010)

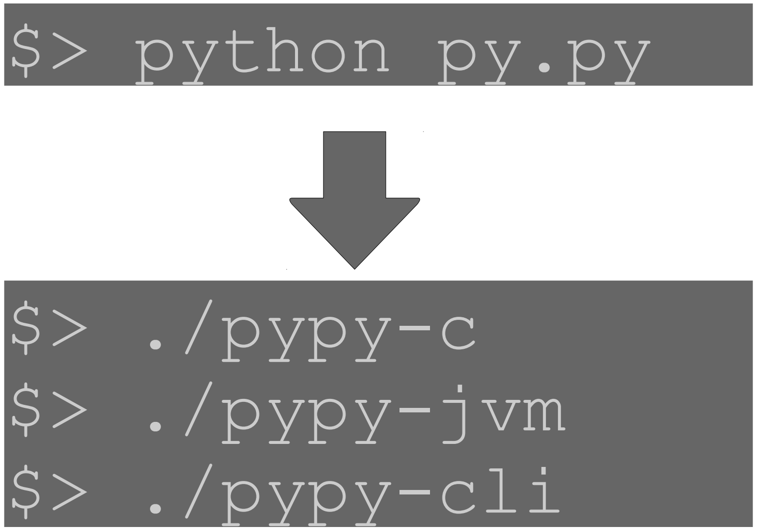




> “It's possible to take an interpreter and transform it into a compiler”
>
> – Yoshihiko Futamura (1971). "Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler"

What is PyPy?

- Reimplementation of Python in Python
- Framework for building interpreters and virtual machines **with Restricted Python**
- L * O * P configurations
  - L : dynamic languages
  - O : optimizations
  - P : platforms



```
$> python py.py
```

```
$> ./pypy-c
$> ./pypy-jvm
$> ./pypy-cli
```

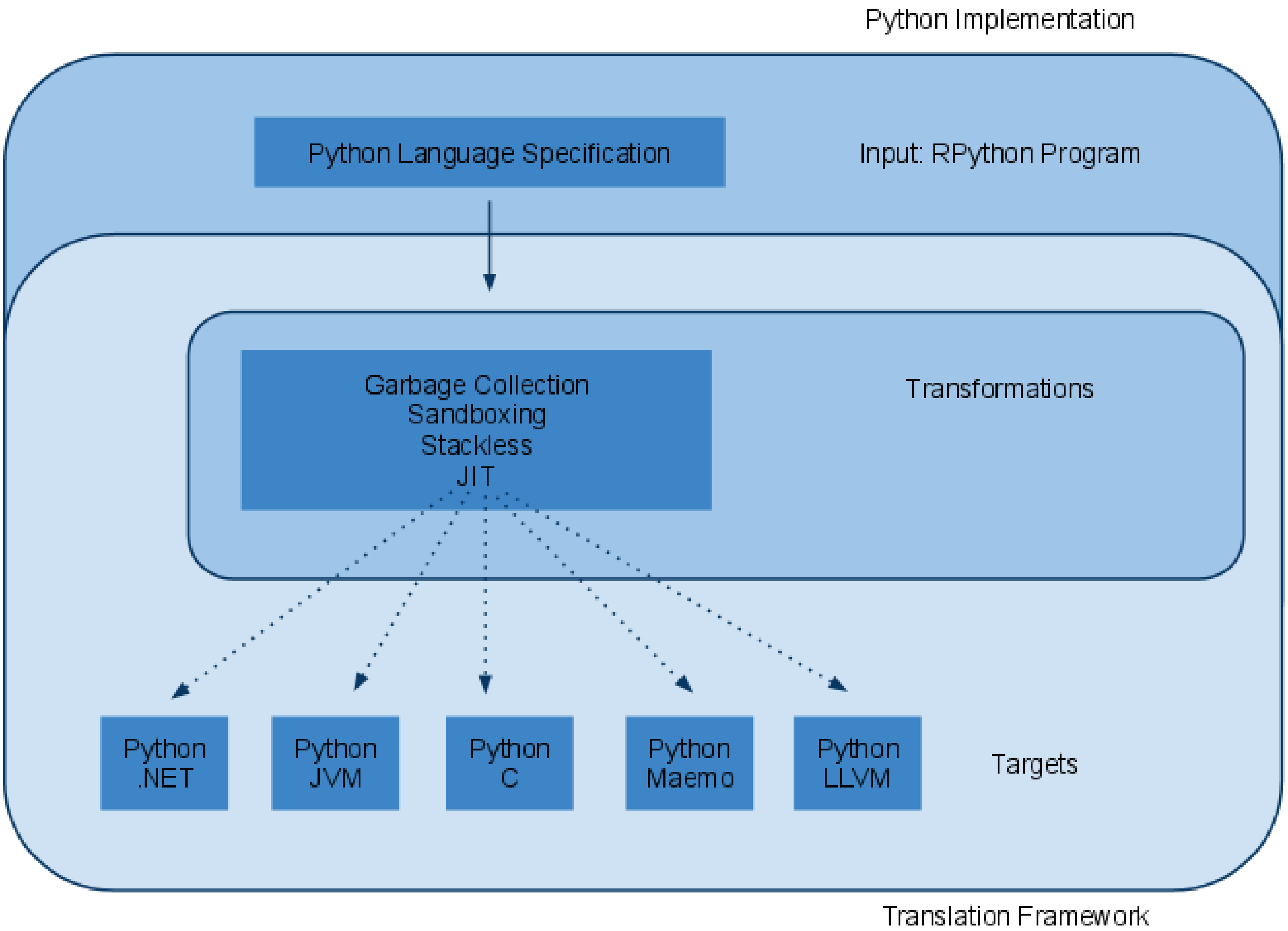
PyPy

- founded in 2003 by Holger Krekel and Armin Rigo
- 2004 - 2007 EU Project - Sixth Framework
- Open Source Project since 2007
- occasional Funding by Google
- Sprint-Driven Development

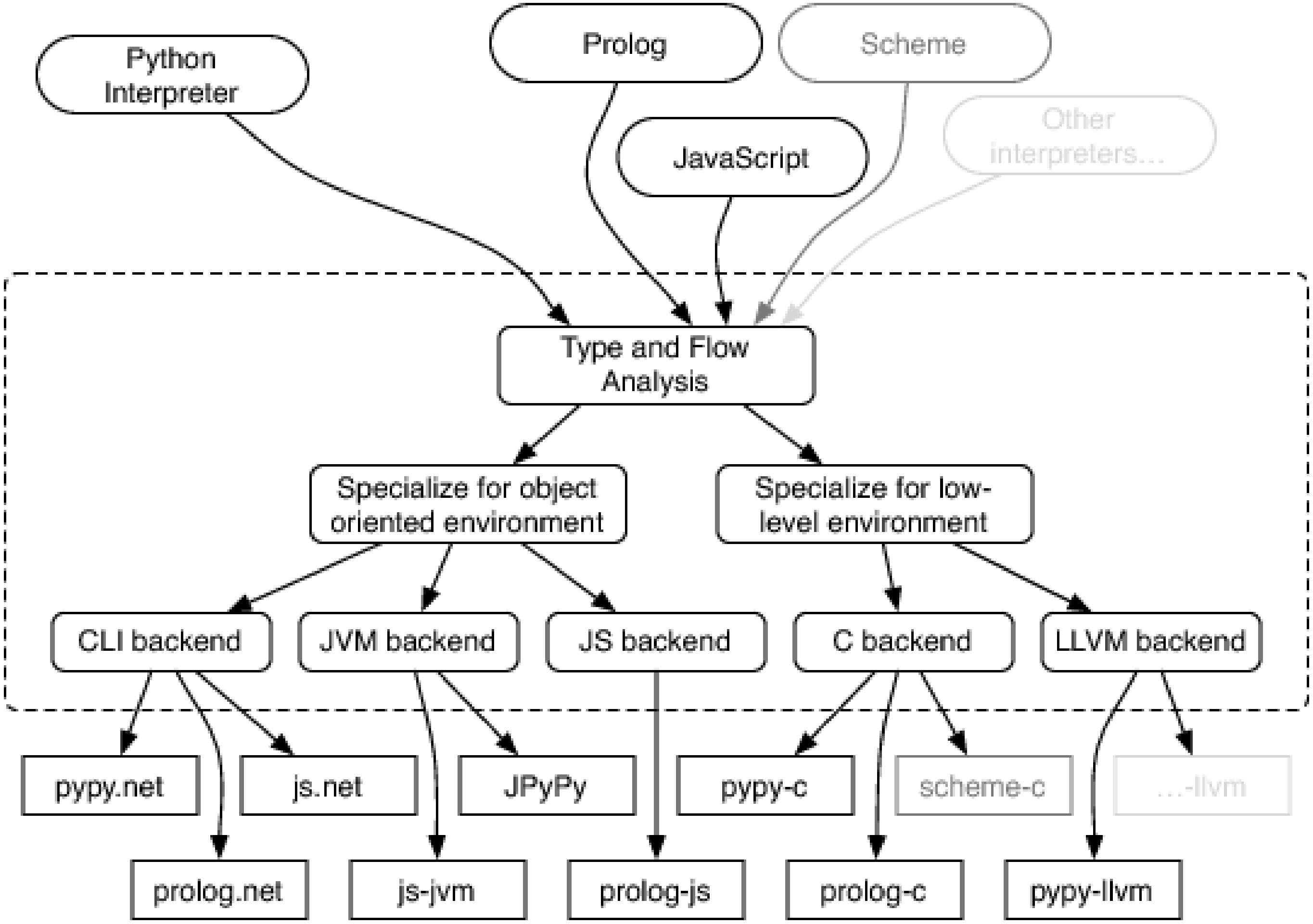
PyPy

- common use scenario is to translate the PyPy RPython code to a backend
    - C (and then standalone binary), CLI (.Net), JVM
- PyPy component
    - A Python interpreter with the ability to collects traces
    - A tracing JIT, derived from RPython
        - Tracing of loops in the user level programs, but recording exact operations executed inside the interpreter
    - Well defined points to enter and exit traces, and state that can be safely modified inside the trace

PyPy Architecture



PyPy Functional Architecture

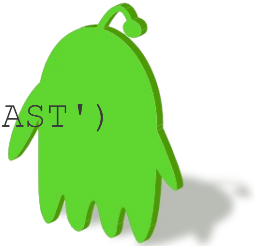


PyPy Abilities

- Use techniques similar to prototype languages (V8) to infer offsets of instance attributes
- Garbage collected
- Can interface with (most) standard CPython modules
  - Creates PyObject proxies to internal PyPy objects
- Limited concurrency because of GIL

PyPy Traces

```
[2bcbab384d062] {jit-log-noopt-loop
[p0, p1, p2, p3, p4, p5, p6, p7, p8]
debug_merge_point('<code object fioranoTest, file 'perf.py', line 2> #24 JUMP_IF_FALSE')
debug_merge_point('<code object fioranoTest, file 'perf.py', line 2> #27 POP_TOP')
debug_merge_point('<code object fioranoTest, file 'perf.py', line 2> #28 LOAD_FAST')
guard_nonnull(p8, descr=<ResumeGuardDescr object at 0xf6c4cd7c>)
debug_merge_point('<code object fioranoTest, file 'perf.py', line 2> #31 LOAD_FAST')
guard_nonnull(p7, descr=<ResumeGuardDescr object at 0xf6c4ce0c>)
debug_merge_point('<code object fioranoTest, file 'perf.py', line 2> #34 BINARY_ADD')
guard_class(p8, ConstClass(W_IntObject), descr=<ResumeGuardDescr object at 0xf6c4ce9c>)
guard_class(p7, ConstClass(W_IntObject), descr=<ResumeGuardDescr object at 0xf6c4cf08>)
guard_class(p8, ConstClass(W_IntObject), descr=<ResumeGuardDescr object at 0xf6c4cf74>)
guard_class(p7, ConstClass(W_IntObject), descr=<ResumeGuardDescr object at 0xf6c4cfe0>)
i13 = getfield_gc_pure(p8, descr=<SignedFieldDescr 8>)
i14 = getfield_gc_pure(p7, descr=<SignedFieldDescr 8>)
i15 = int_add_ovf(i13, i14)
guard_no_overflow(, descr=<ResumeGuardDescr object at 0xf6c4d0c8>)
p17 = new_with_vtable(ConstClass(W_IntObject))
setfield_gc(p17, i15, descr=<SignedFieldDescr 8>)
debug_merge_point('<code object fioranoTest, file 'perf.py', line 2> #35 STORE_FAST')
…
[2bcbab3877419] jit-log-noopt-loop}
```



RPython

- RPython = Restricted/Reduced Python
  - Restricted = most possible static subset of Python
- required to perform Type Inference
- input to the Translation Framework
- no real specification - just some hints
- used for writing interpreters
- *can* be used for writing extensions as well

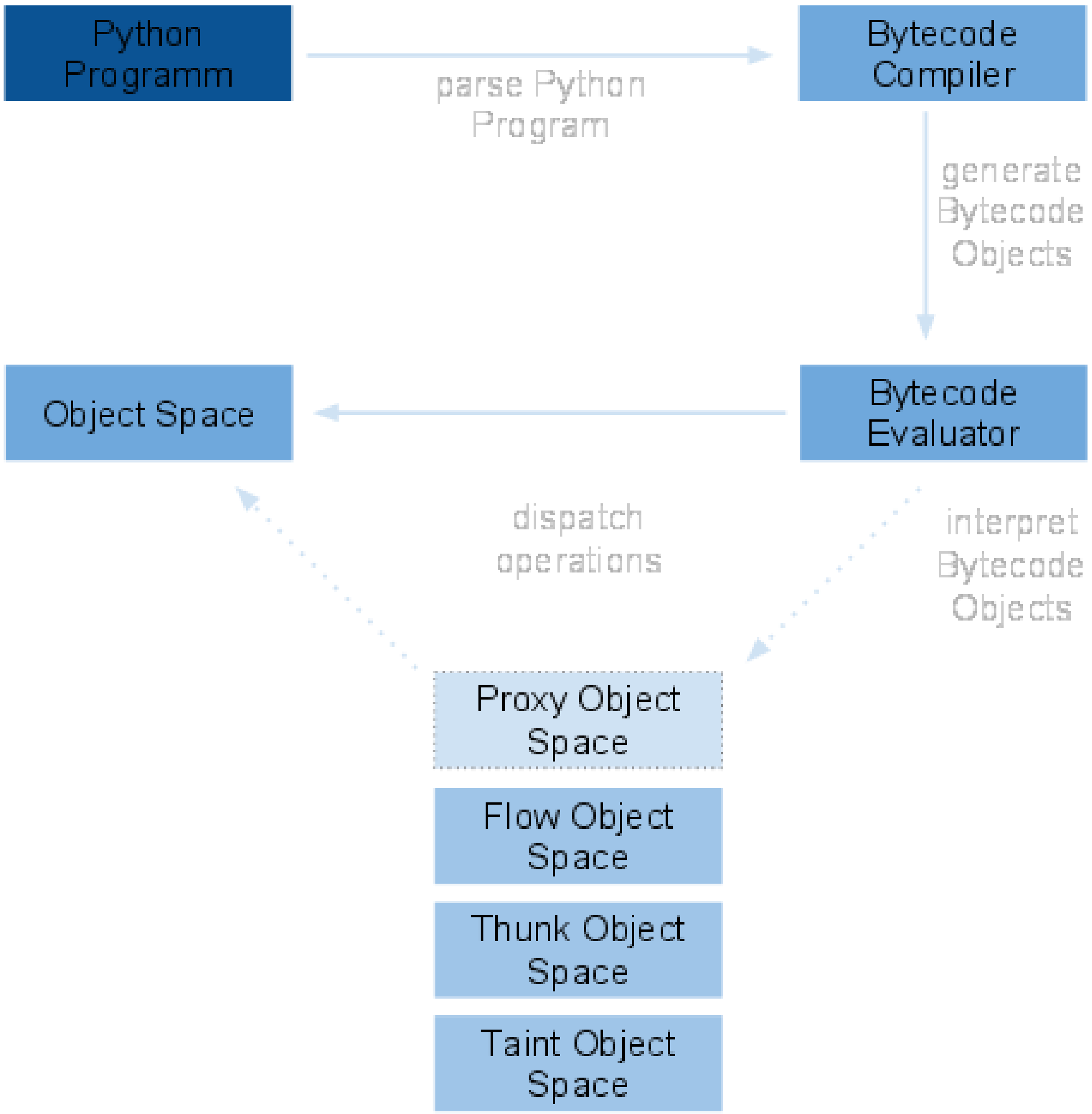
PyPy Interpreter

- written in Rpython
- Stack-based bytecode interpreter (like JVM)
  - bytecode compiler → generates bytecode
  - bytecode evaluator → interprets bytecode
  - object space → handles operations on objects

```
def f(x):
    return x + 1
```

```
>>> dis.dis(f)
2      0 LOAD_FAST       0 (x)
       3 LOAD_CONST      1 (1)
       6 BINARY_ADD
       7 RETURN_VALUE
```

PyPy Bytecode Interpreter



Python Programm

parse Python Program

Bytecode Compiler

generate Bytecode Objects

Bytecode Evaluator

Object Space

dispatch operations

interpret Bytecode Objects

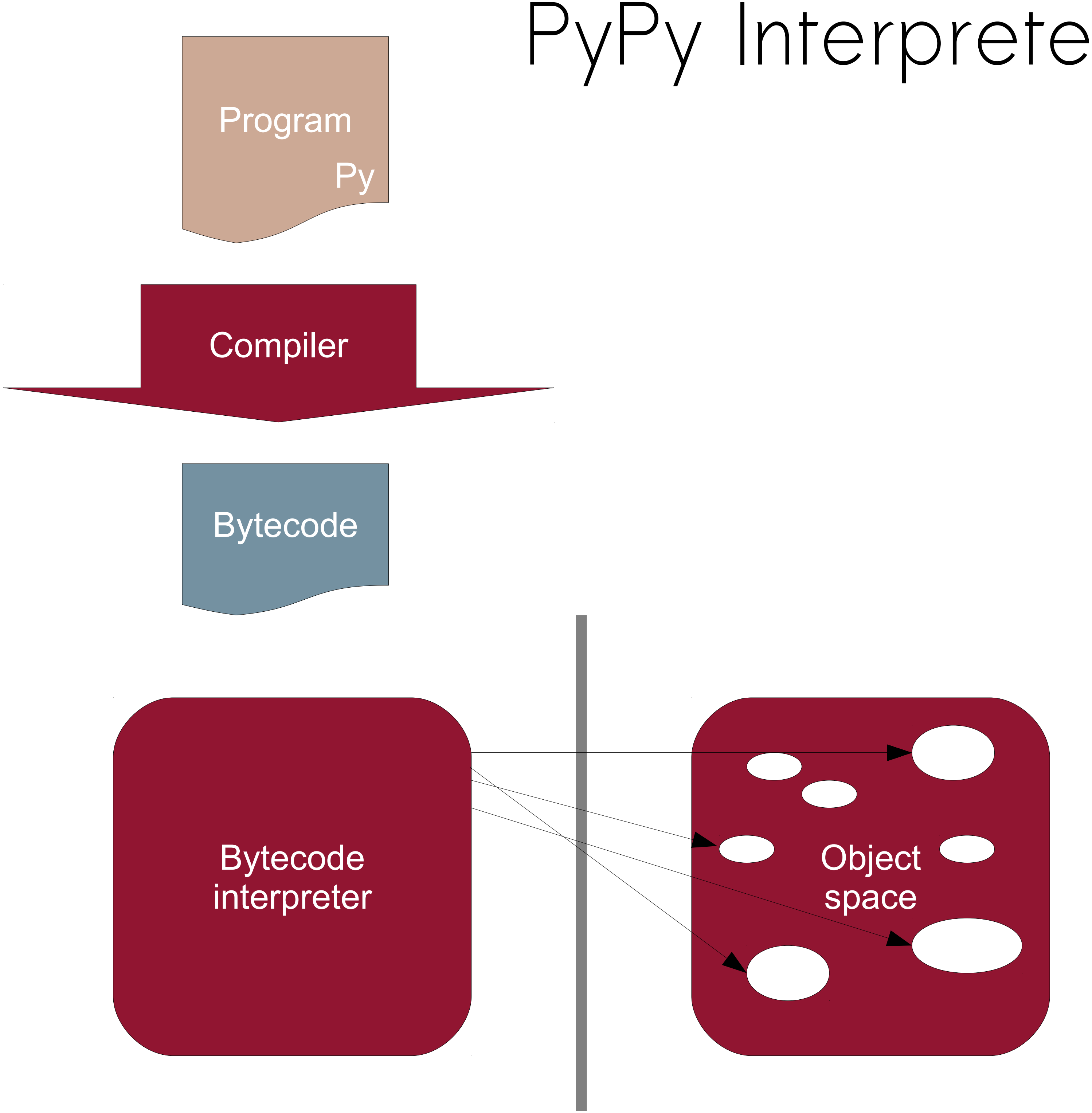
Proxy Object Space

Flow Object Space

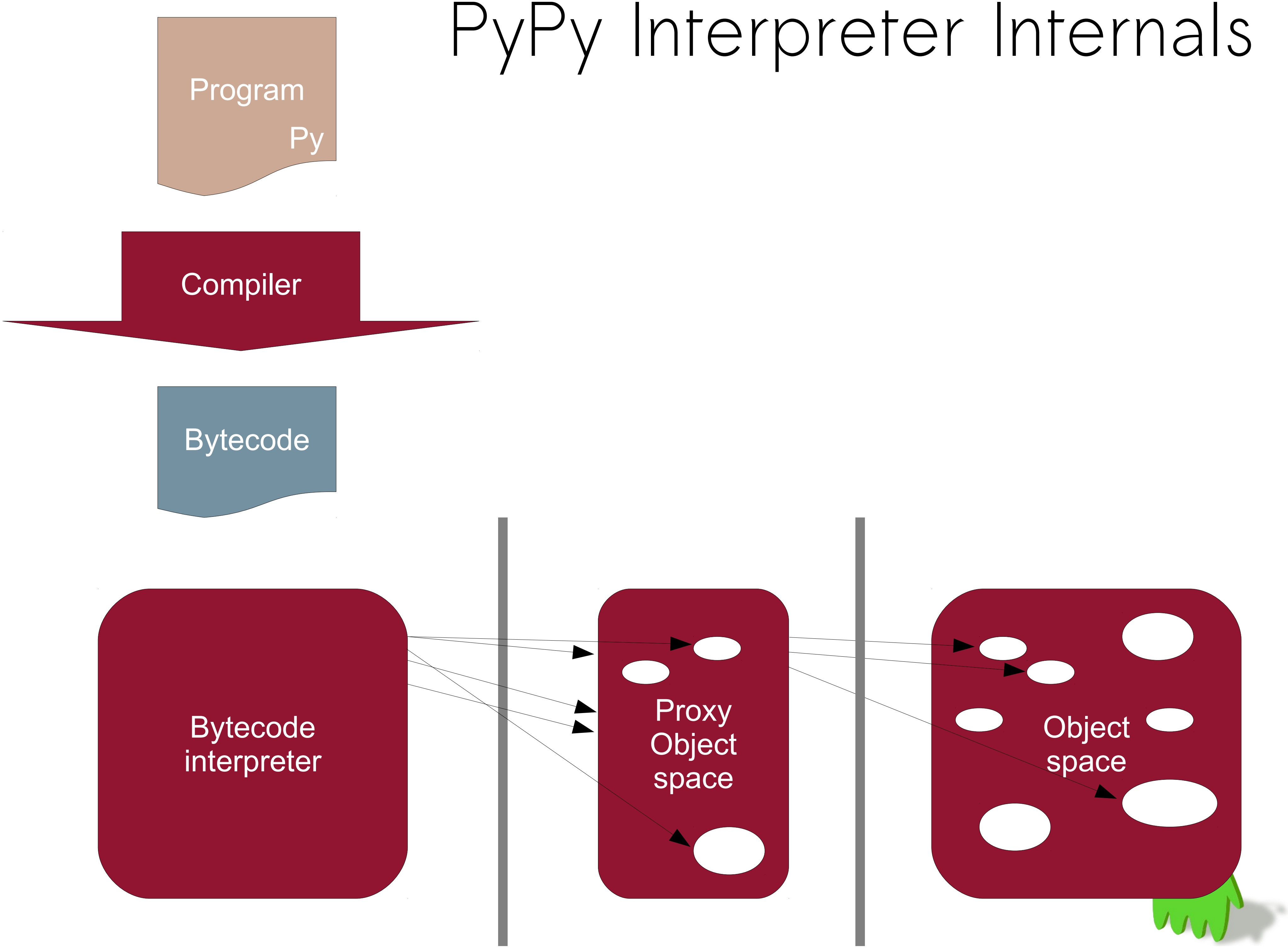
Thunk Object Space

Taint Object Space

PyPy Interpreter Internals



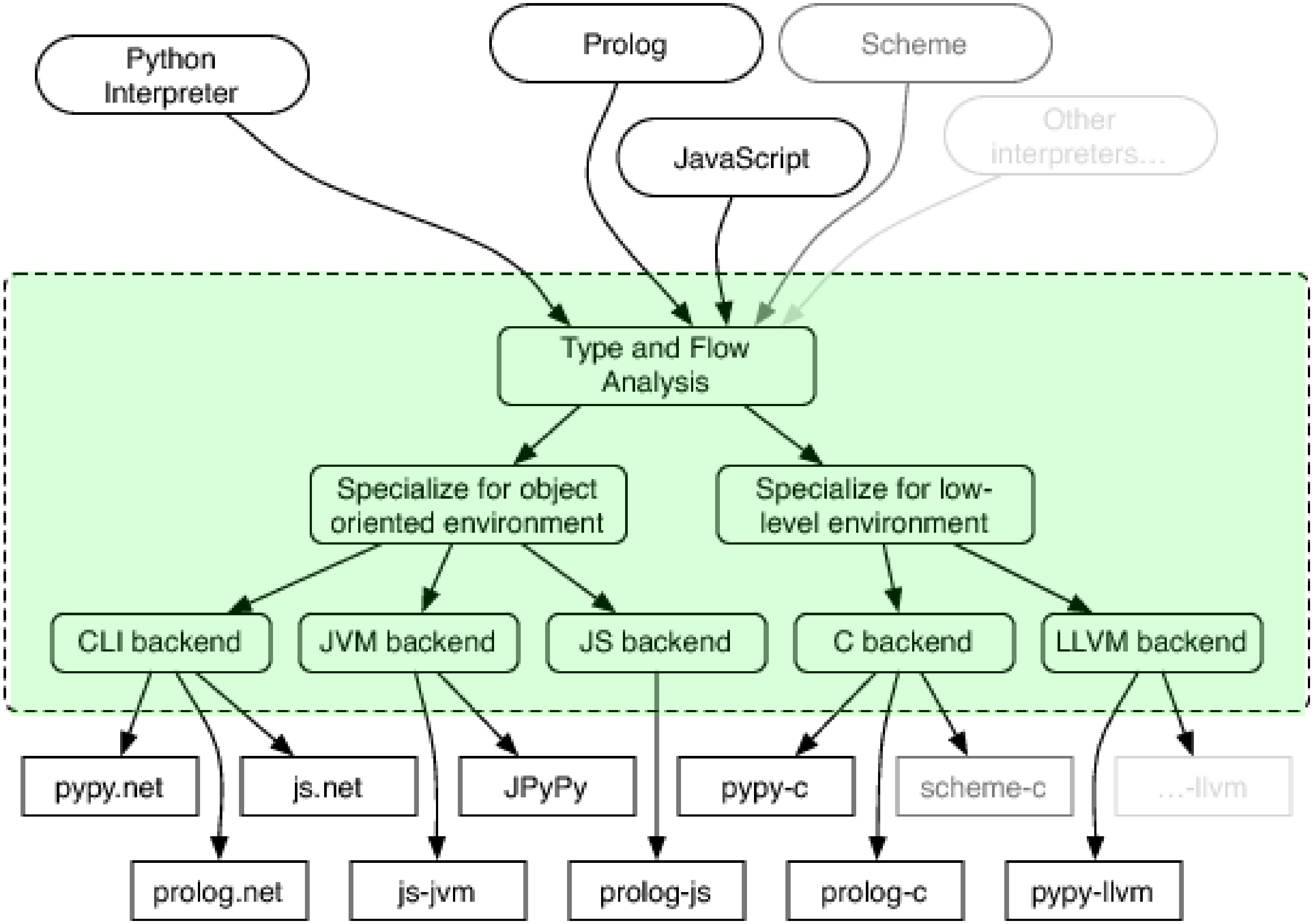
PyPy Interpreter Internals

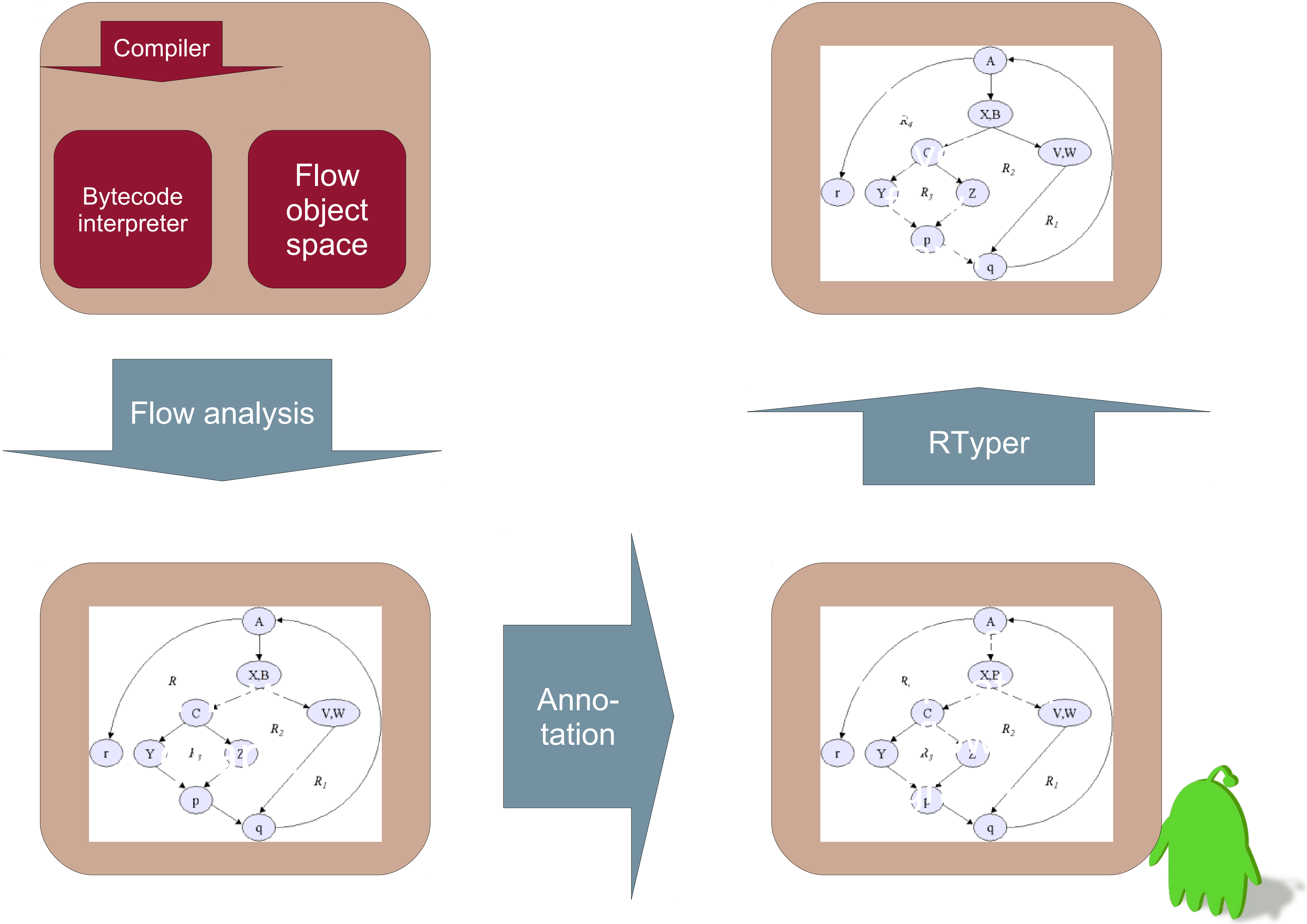


PyPy Translation Toolchain

- Model-driven interpreter (VM) development
  - Focus on language model rather than implementation details
  - Executable models (meta-circular Python)
- Translate models to low-level (LL) back-ends
  - Considerably lower than Python
  - Weave in implementation details (GC, JIT)
  - Allow compilation to different back-ends (OO, procedural)

PyPy Translation Toolchain





Compiler

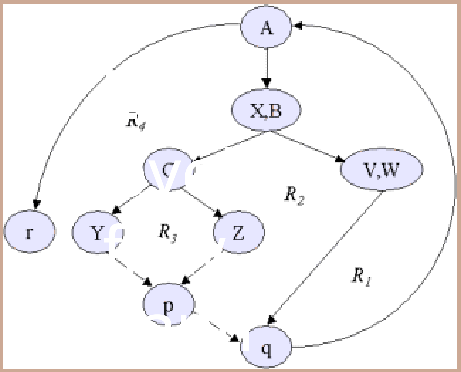
Bytecode interpreter

Flow object space

Flow analysis

Anno-tation

RTyper



Code generation

POSI
X
JVM
CLI

Garbage collector

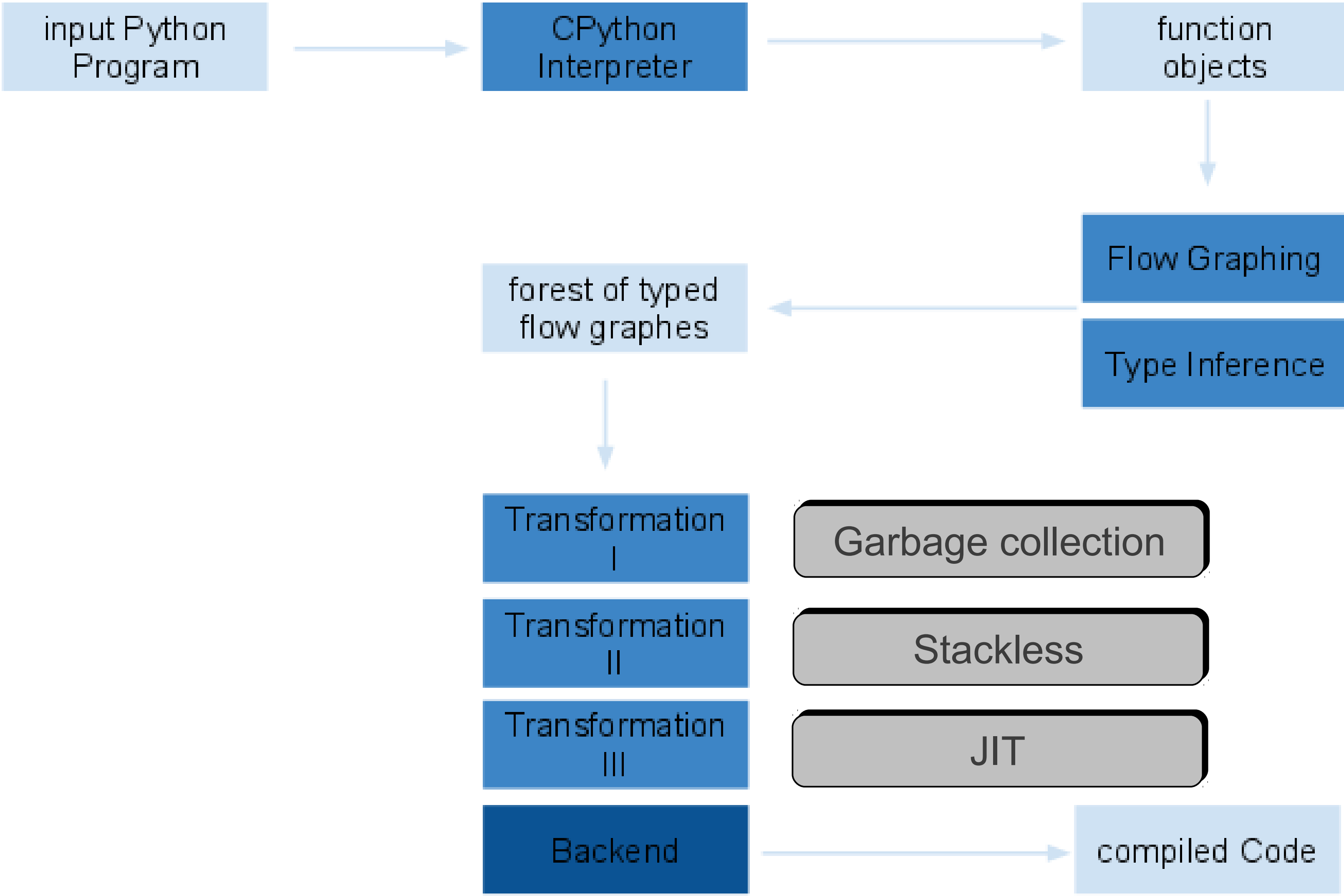
JIT generation

POSI
X
(JVM)
(CLI)

PyPy interpreter
with Just In Time
Optimizations

X

Translation Framework



input Python Program

CPython Interpreter

function objects

Flow Graphing

Type Inference

forest of typed flow graphes

Transformation I

Transformation II

Transformation III

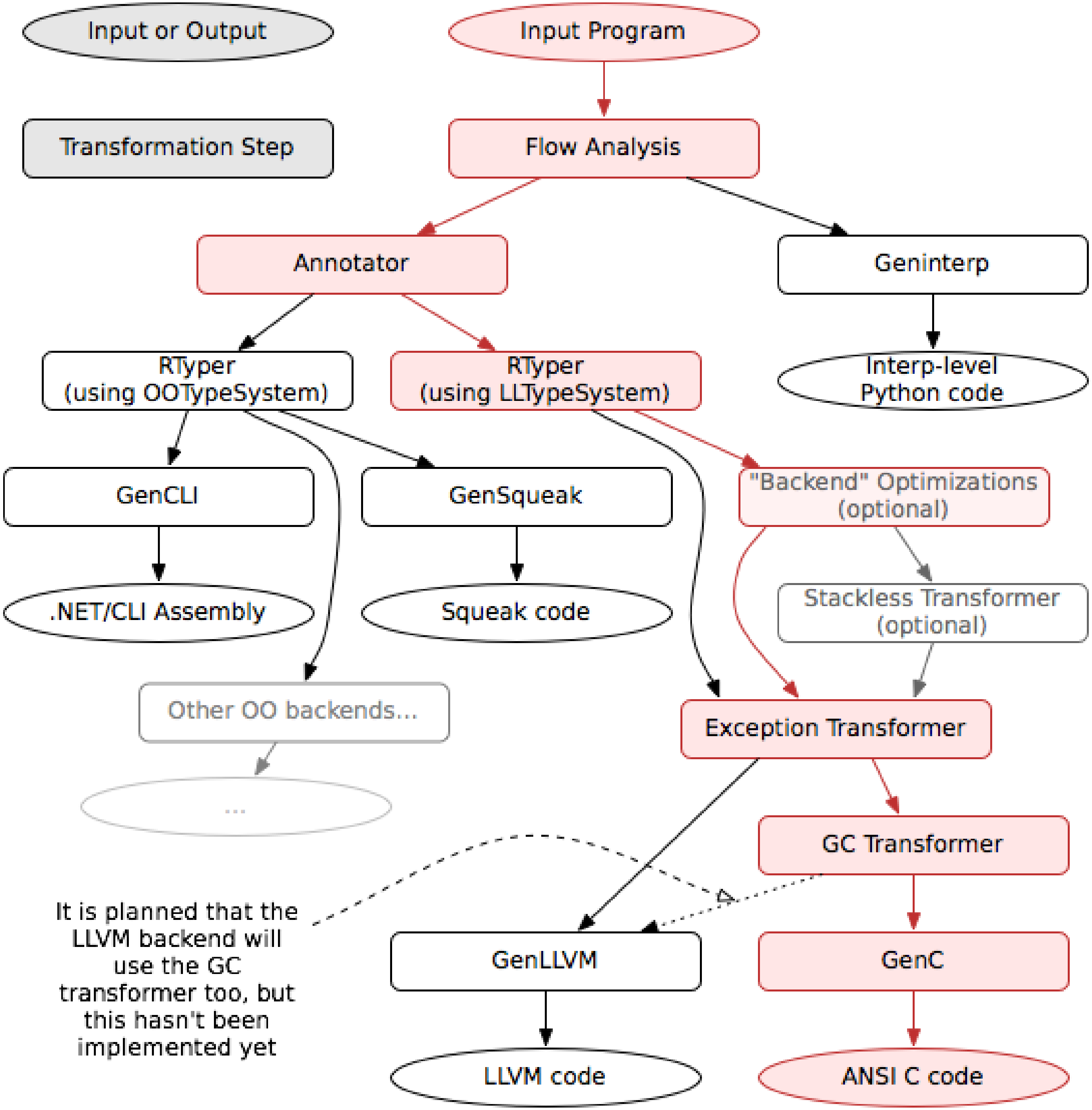
Backend

Garbage collection

Stackless

JIT

compiled Code



Input or Output

Transformation Step

Input Program

Flow Analysis

Annotator

Geninterp

RTyper
(using OOTypeSystem)

RTyper
(using LLTypeSystem)

Interp-level
Python code

GenCLI

GenSqueak

"Backend" Optimizations
(optional)

.NET/CLI Assembly

Squeak code

Stackless Transformer
(optional)

Other OO backends...

...

Exception Transformer

GC Transformer

It is planned that the
LLVM backend will
use the GC
transformer too, but
this hasn't been
implemented yet

GenLLVM

GenC

LLVM code

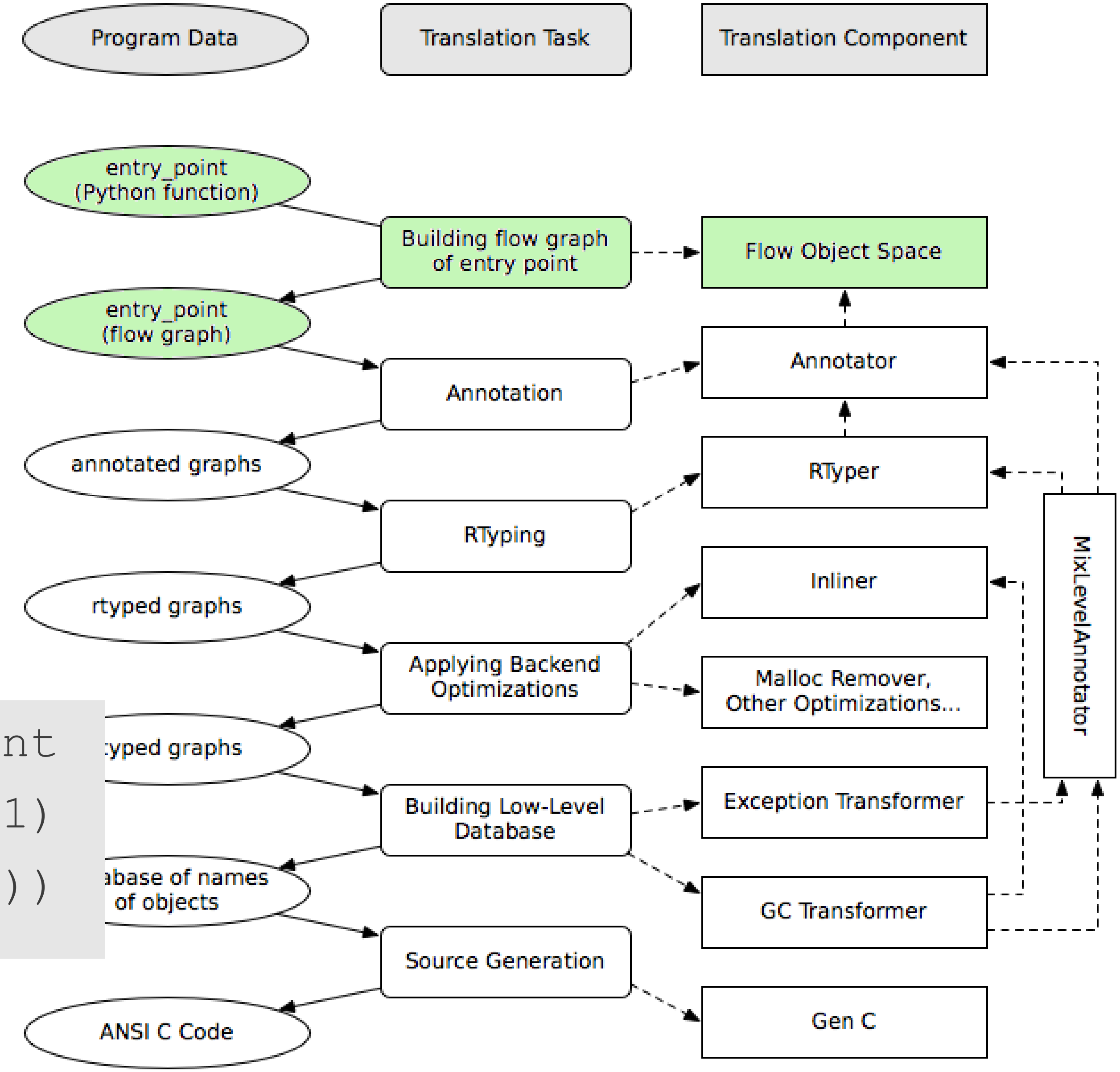
ANSI C code

CFG (Call Flow Graph)

- Consists of Blocks and Links
- Starting from *entry_point*
- “Single Static Information” form

```
def f(n):
    return 3 * n + 2
```

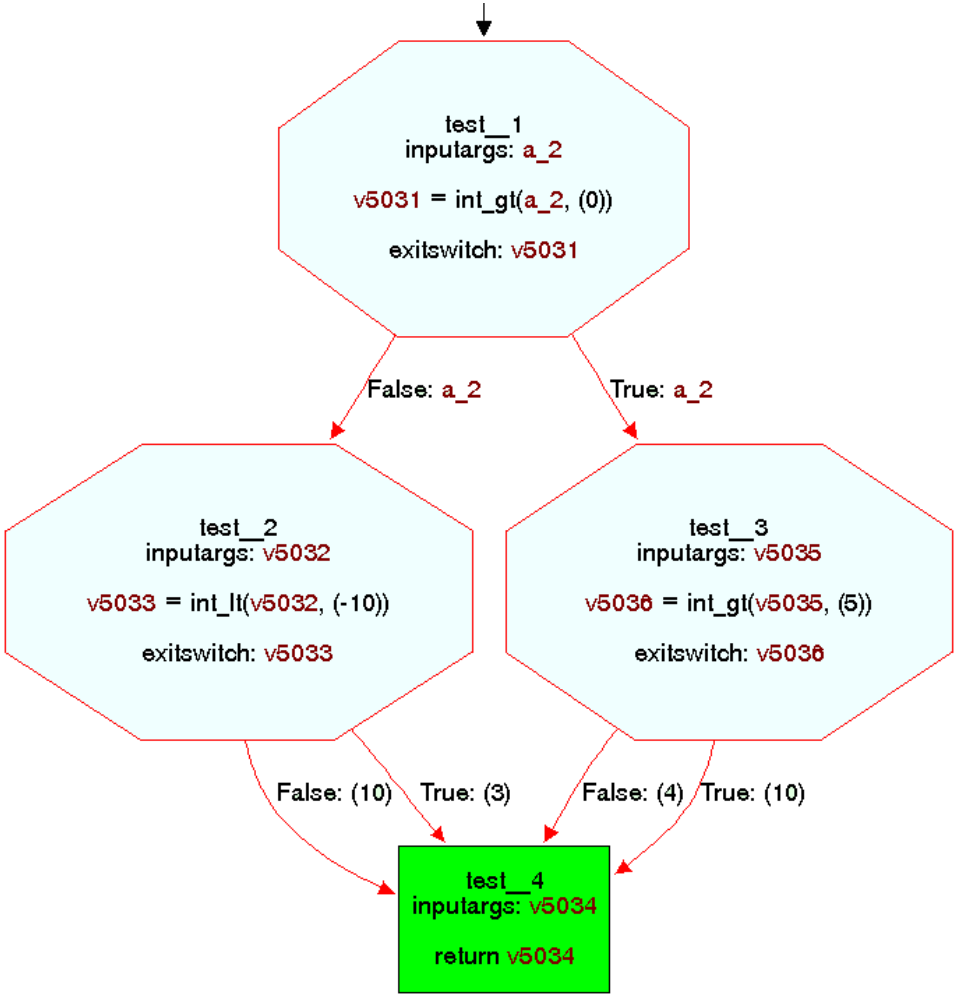
```
Block(v1): # input argument
  v2 = mul(Constant(3), v1)
  v3 = add(v2, Constant(2))
```



CFG: Static Single Information

- SSI: “PHIs” for all used variables
- Blocks as “functions without branches”

```
def test(a):
    if a > 0:
        if a > 5:
            return 10
        return 4
    if a < - 10:
        return 3
    return 10
```



test__1
inputargs: a_2
v5031 = int_gt(a_2, (0))
exitswitch: v5031

False: a_2
True: a_2

test__2
inputargs: v5032
v5033 = int_lt(v5032, (-10))
exitswitch: v5033

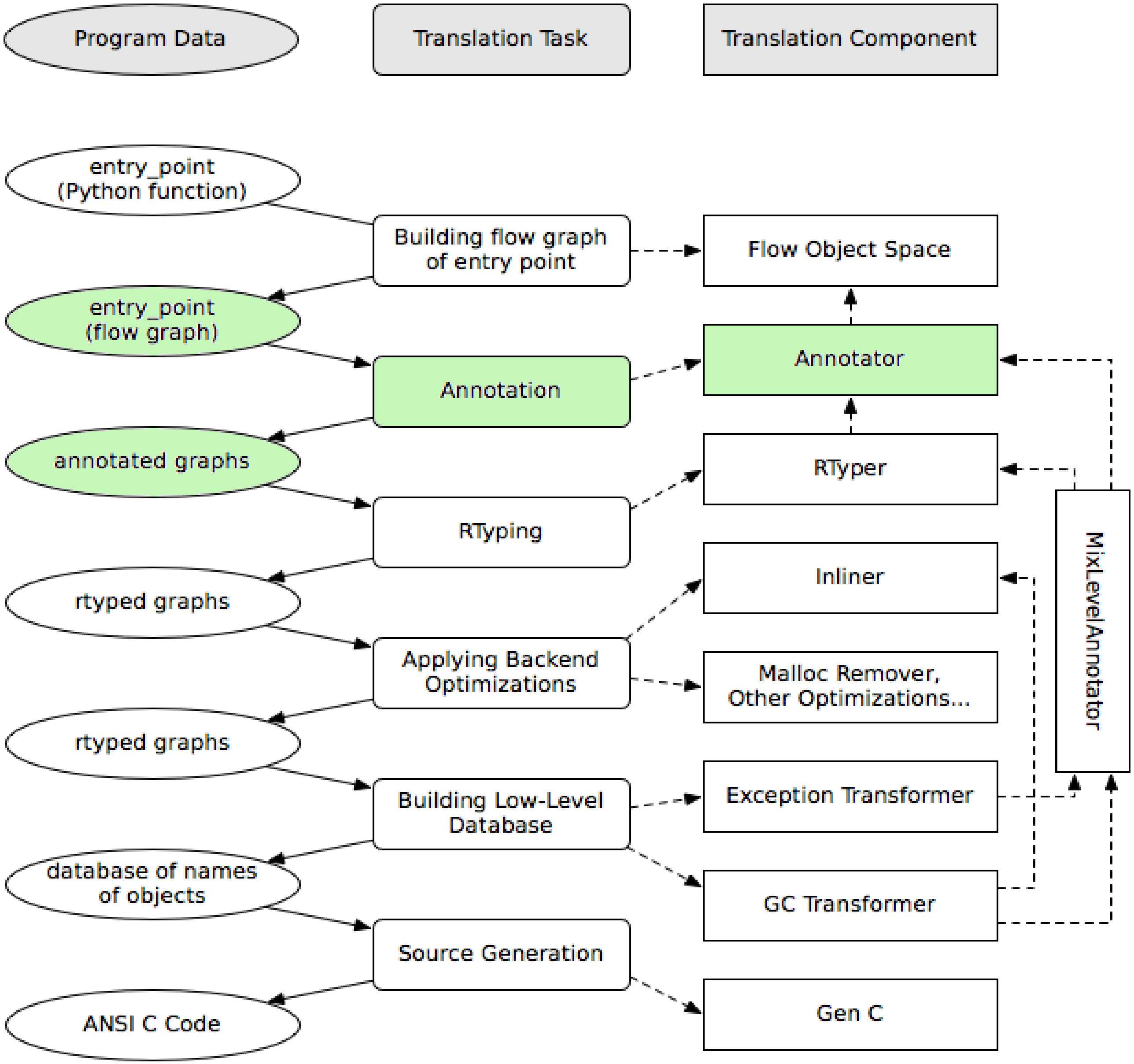
test__3
inputargs: v5035
v5036 = int_gt(v5035, (5))
exitswitch: v5036

False: (10)
True: (3)
False: (4)
True: (10)

test__4
inputargs: v5034
return v5034

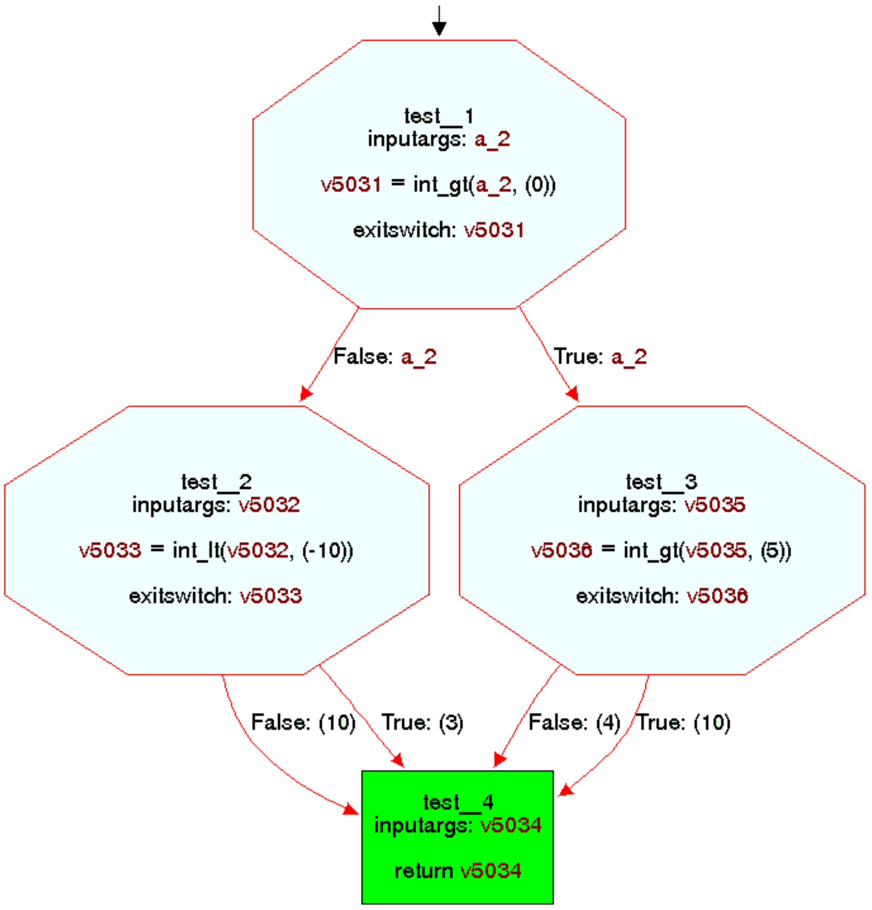
Type Inference

- Python is dynamically typed
- Translate to statically typed code for efficiency reasons



Type Inference

- Which to be inferred?
  - Type for every variable
  - Messages sent to an object must be defined in the compile-time type or a supertype
- How to infer types
  - Starting from *entry_point*
    - reach the whole program
    - type of arguments and return-value are known
  - Forward propagation
    - Iteratively, until all links in the CFG have been followed at least once
    - Results in a large dictionary mapping variables to types



test__1
inputargs: a_2
v5031 = int_gt(a_2, (0))
exitswitch: v5031

test__2
inputargs: v5032
v5033 = int_lt(v5032, (-10))
exitswitch: v5033

test__3
inputargs: v5035
v5036 = int_gt(v5035, (5))
exitswitch: v5036

test__4
inputargs: v5034
return v5034

Type inference restricts

- RRython is the subset of Python, which is type inferable
- Actually: type inferable stabilized bytecode
  - Allows load-time meta-programming
  - Messages sent to an object must be defined in the compile-time type or supertype

```
def plus(a, b):
   return a + b

def entry_point(arv=None):
   print plus(20, 22)
   print plus(“4”, “2”)
```

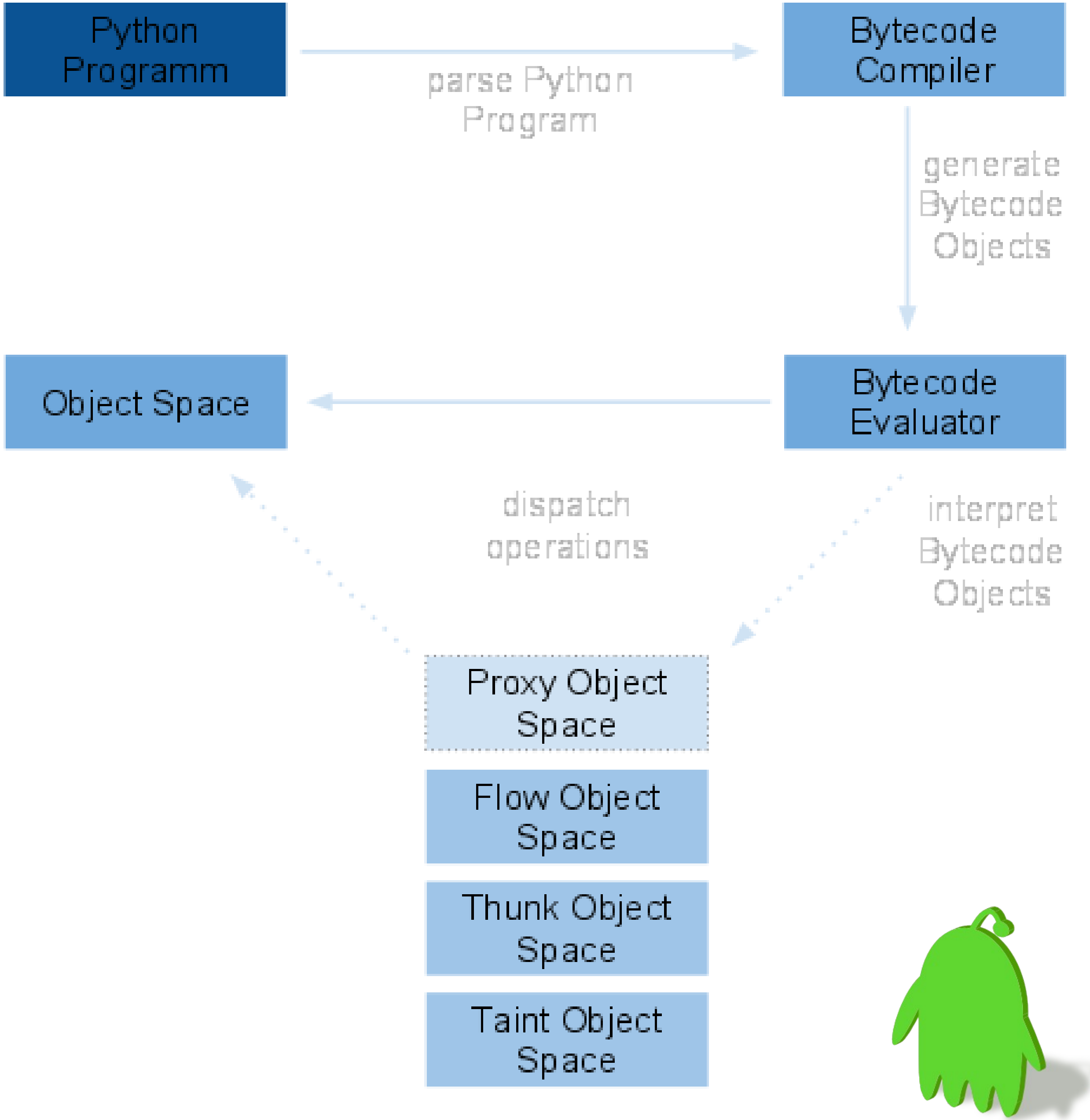
```
@objectmodel.specialize.argtype(0)
def plus(a, b):
   return a + b

def entry_point(arv=None):
   print plus(20, 22)
   print plus(“4”, “2”)
```

PyPy Advantages

- High Level Language Implementation
  - to implement new features: lazily computed objects and functions, plug-able garbage-collection, runtime replacement of live-objects, stackless concurrency
- JIT Generation
- Object space
- Stackless
  - infinite Recursion
  - Microthreads: Coroutines, Tasklets and Channels, Greenlets

Object Spaces in PyPy

- Flow ObjSpace
- Thunk ObjSpace
- Taint ObjSpace
- Dump ObjSpace
- Transparent Proxies



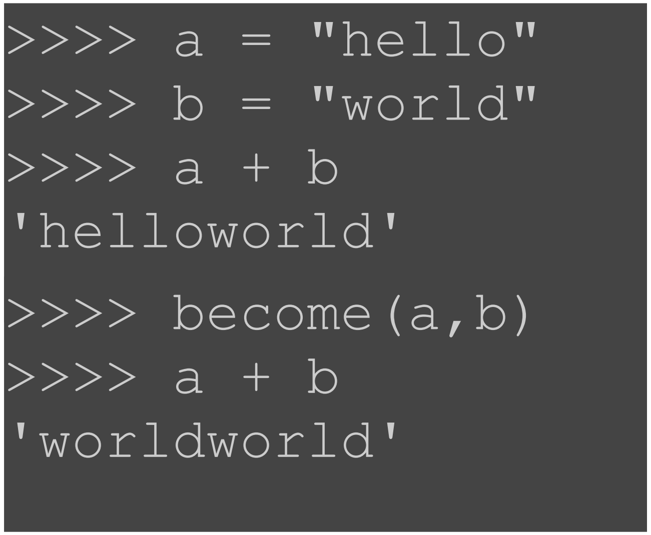
Object Spaces in PyPy

- Thunk ObjSpace

  lazily computed objects

  lazily computed functions

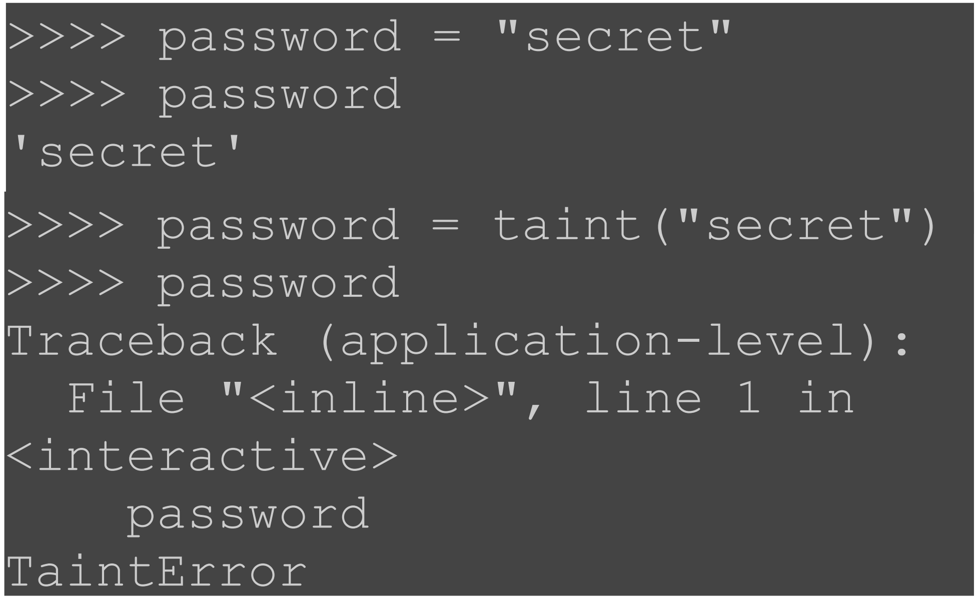
  globally replaceable objects



```
>>>> a = "hello"
>>>> b = "world"
>>>> a + b
'helloworld'
>>>> become(a,b)
>>>> a + b
'worldworld'
```

- Taint ObjSpace
  - provides protection **for**:

    sensitive data that should not leak
  - provides protection **from**:
  - Untrusted data that needs to be validated



```
>>>> password = "secret"
>>>> password
'secret'
>>>> password = taint("secret")
>>>> password
Traceback (application-level):
  File "<inline>", line 1 in
<interactive>
    password
TaintError
```

Reference

- PyPy Internals: http://codespeak.net/pypy/dist/pypy/doc/
- “Compiler Construction”, Prof. O. Nierstrasz, Fall Semester 2008
- “The PyPy translation tool chain”, Toon Verwaest
- “Compilers are from Mars, Dynamic Scripting Languages are from Venus, Jose Castanos”, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Takeshi Ogasawara, Akihiko Tozawa, Peng Wu (2010)

0xlab

http://0xlab.org