PyPy – a progress report

ACCU 2006/Python UK, Oxford

Michael Hudson mwh@python.net
Heinrich-Heine-Univeristät Düsseldorf

What is PyPy?

- PyPy is:
  - An implementation of Python, written in Python
  - An open source project (MIT license)
  - A STREP ("Specific Targeted REsearch Project"), partially funded by the EU
  - A lot of fun!

Demo

- We can currently produce a binary that looks very much like CPython to the user
- It's fairly slow (around the same speed as Jython)
- Can also produce binaries that are more capable than CPython -- stackless, thunk, ...

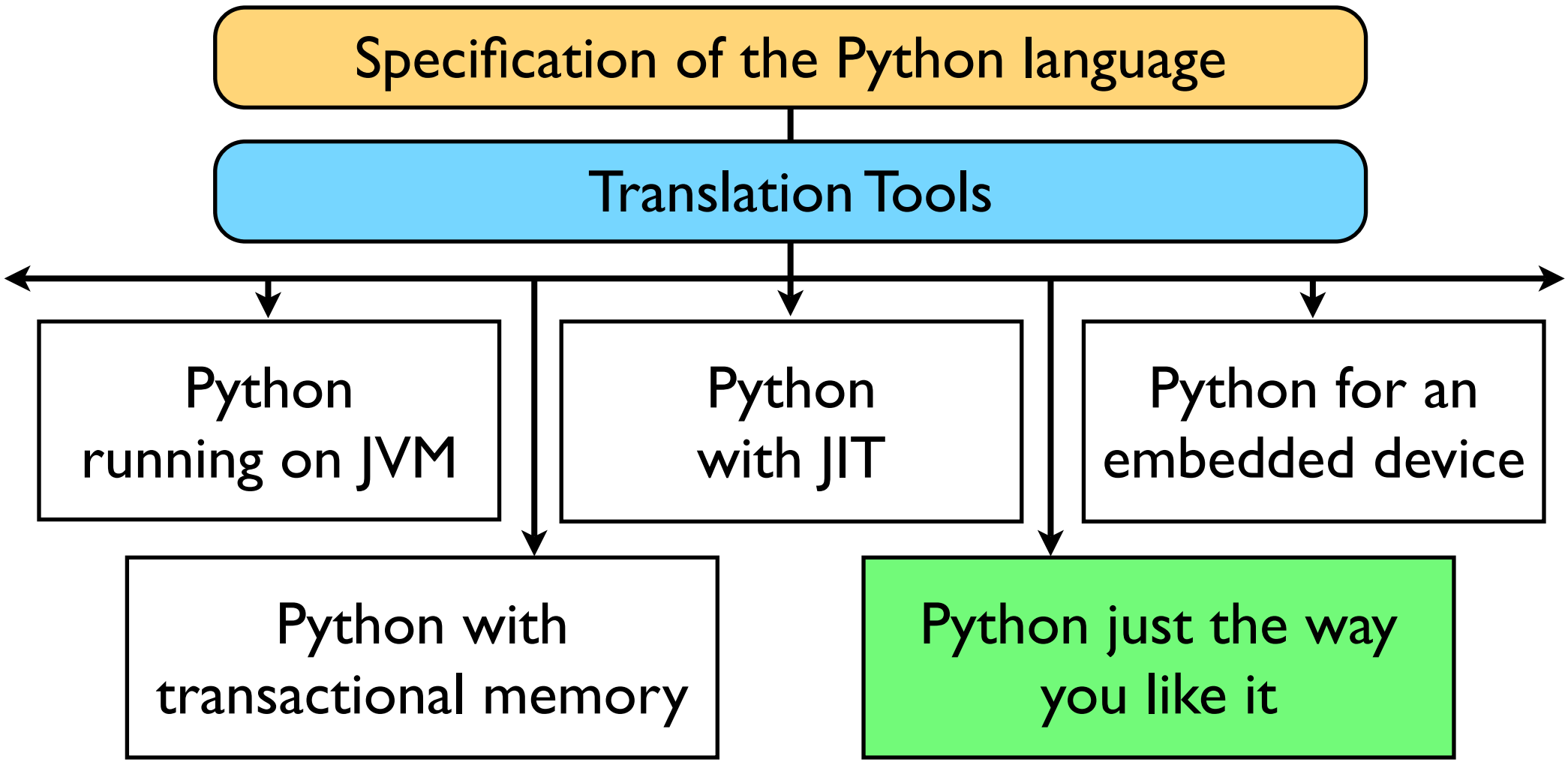
Motivation

- PyPy grew out of a desire to modify/extend the *implementation* of Python, for example to:
  - increase performance (psyco-style JIT compilation, better garbage collectors)
  - add expressiveness (stackless-style coroutines, logic programming)
  - ease porting (to new platforms like the JVM or CLI or to low memory situations)

Lofty goals, but first...

- CPython is hardly a bad implementation of Python but:
  - it's written in C, which makes porting to, for example, the CLR hard
  - while psyco and stackless exist, they are very hard to maintain as Python evolves
  - some implementation decisions would be very hard to change (e.g. refcounting)

Enter the PyPy platform



How do you specify the Python language?

- The way we did it was to write an interpreter for Python in *RPython* – a subset of Python that is amenable to analysis
- This lets us write unit tests for our specification/implementation that run on top of CPython
- Can also test entire specification/ implementation in same way

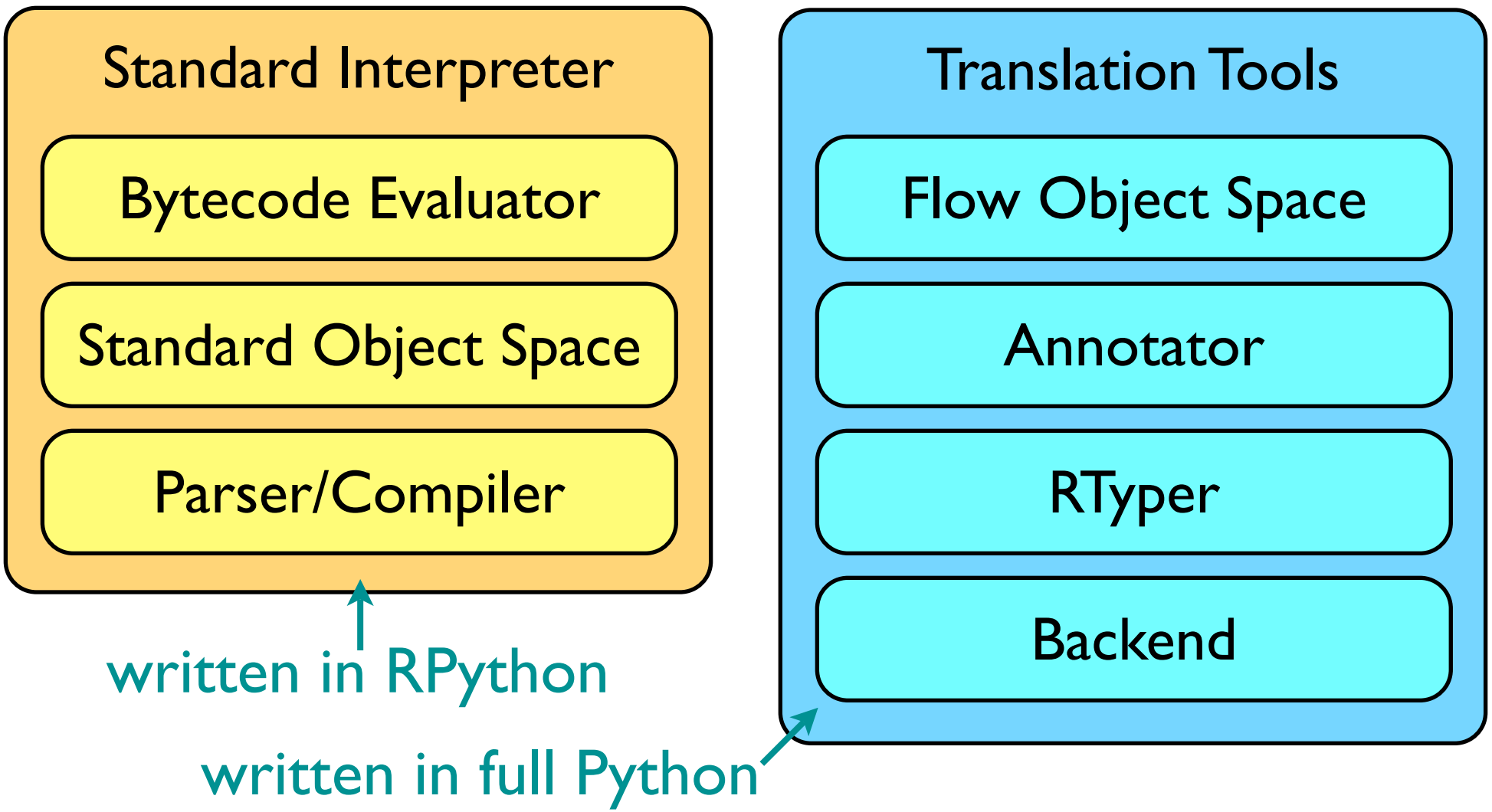
The "What is RPython?" question

- Restricted Python, or RPython, first and foremost it *is* Python
- It is a subset of Python that is static enough – *after initialization code has run* – for our analysis tools to cope with
- Somewhat Java-like – classes, methods, no pointers, no operator overloading

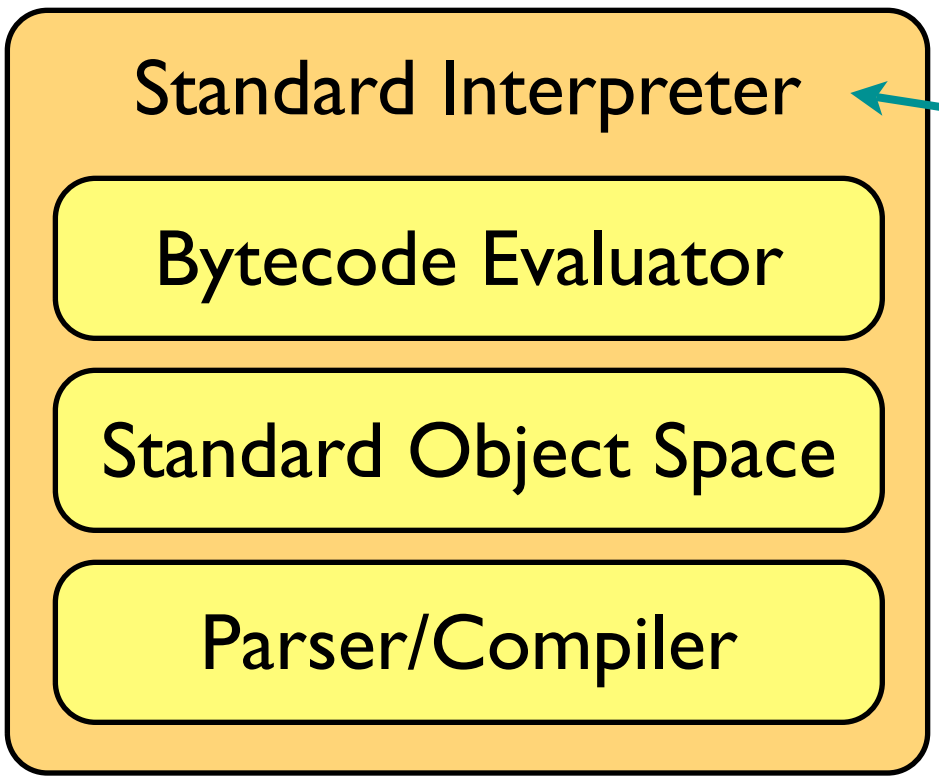
The "What is RPython?" question

- The property of "being RPython" belongs to entire programs and not, say, functions or modules because the annotator performs a global analysis
- The definition of RPython is basically "what our tools accept" – so changes (slowly) as toolchain does

In more detail…



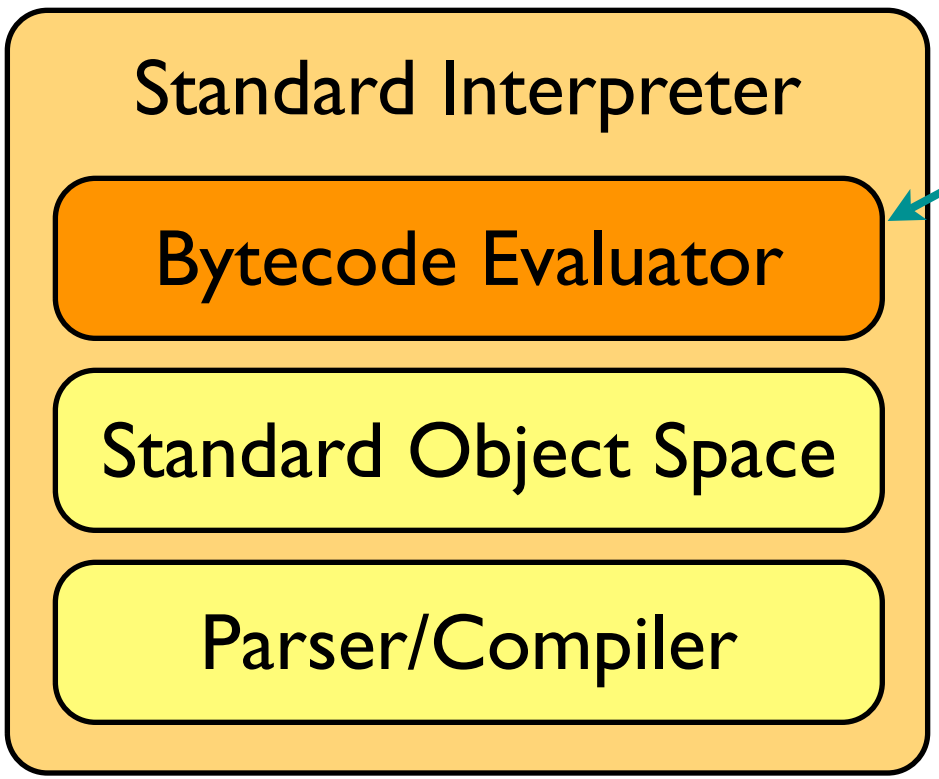
The Standard Interpreter



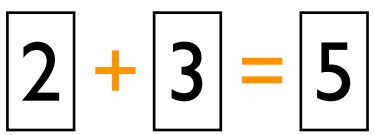
The standard interpreter does roughly speaking the same job as CPython does, and is split into three chunks

CPython can be split along the same lines with enough imagination – hardly a coincidence!

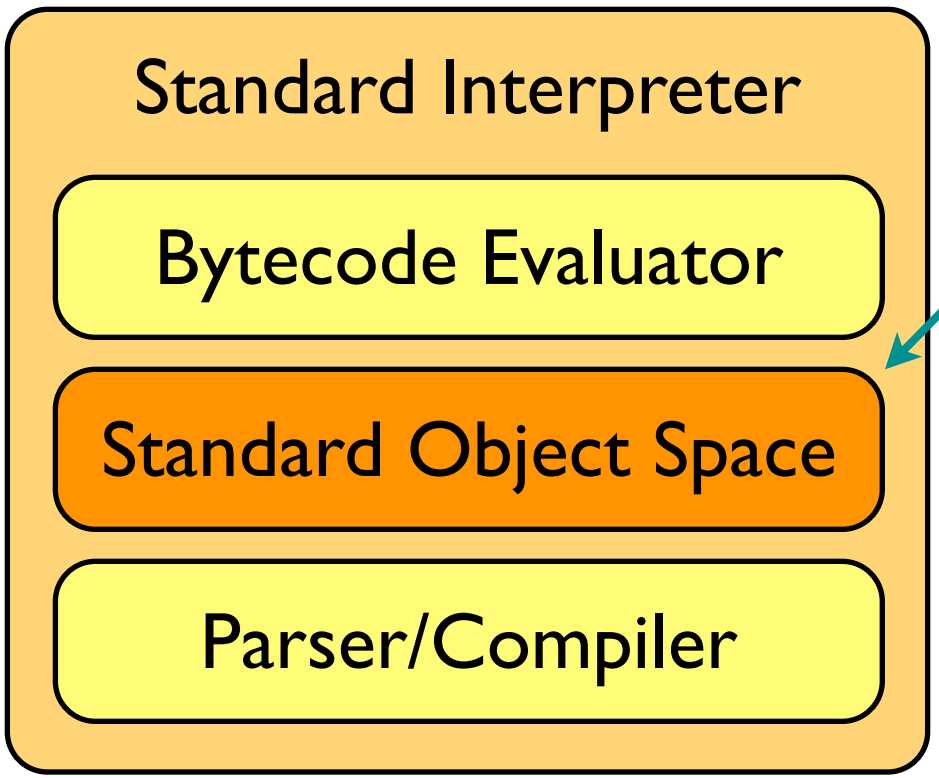
The Standard Interpreter



The bytecode evaluator evaluates the same bytecodes as CPython but treats objects as black boxes – it doesn't care if they are Python-like values, abstract `Variables` or even fruit



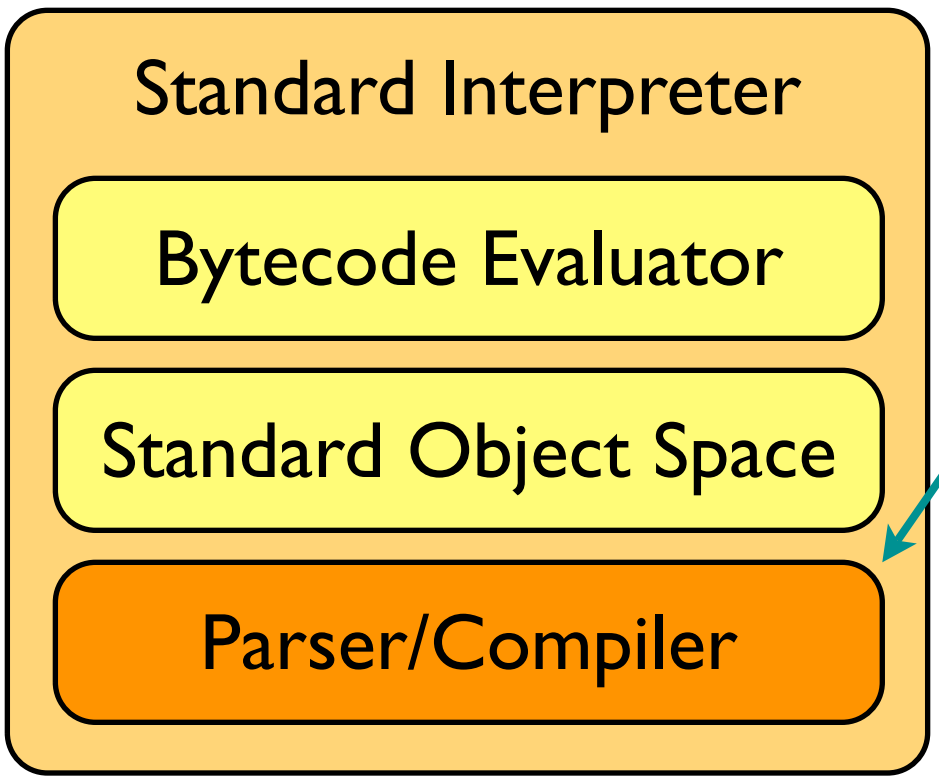
The Standard Interpreter



The Standard Object Space implements objects that look very much like CPython's – integers, lists, dictionaries, classes, etc

(it's a bit different on the inside though)

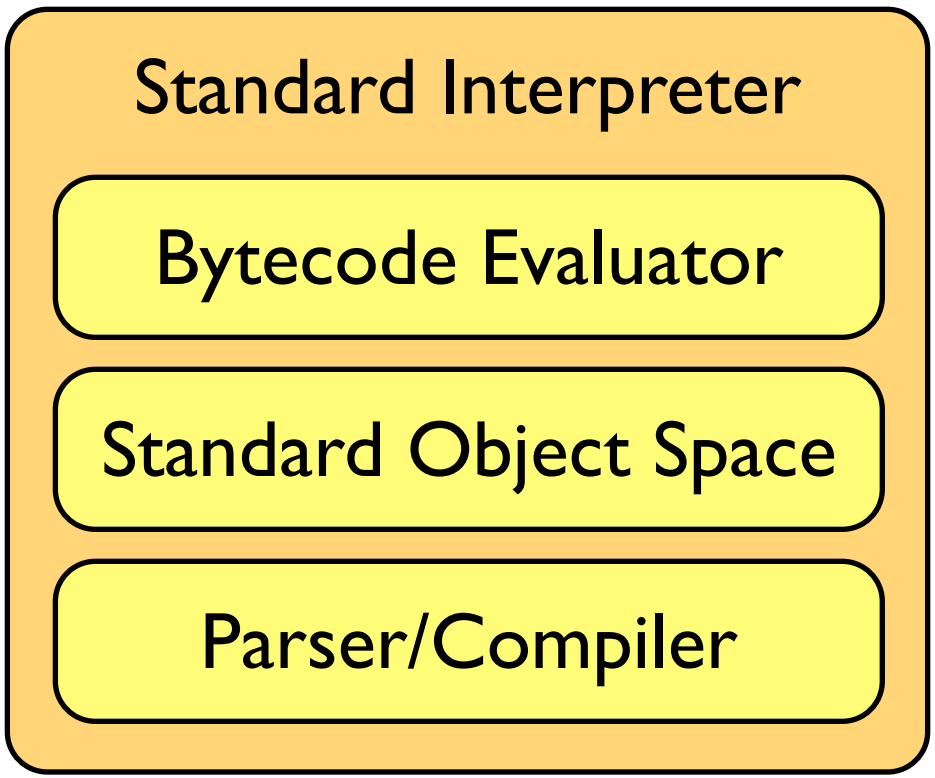
The Standard Interpreter



The parser and compiler, perhaps predictably, parses Python code and compiles it – to the same bytecode as CPython uses

Will sometime soon allow runtime modification of the grammar of the language

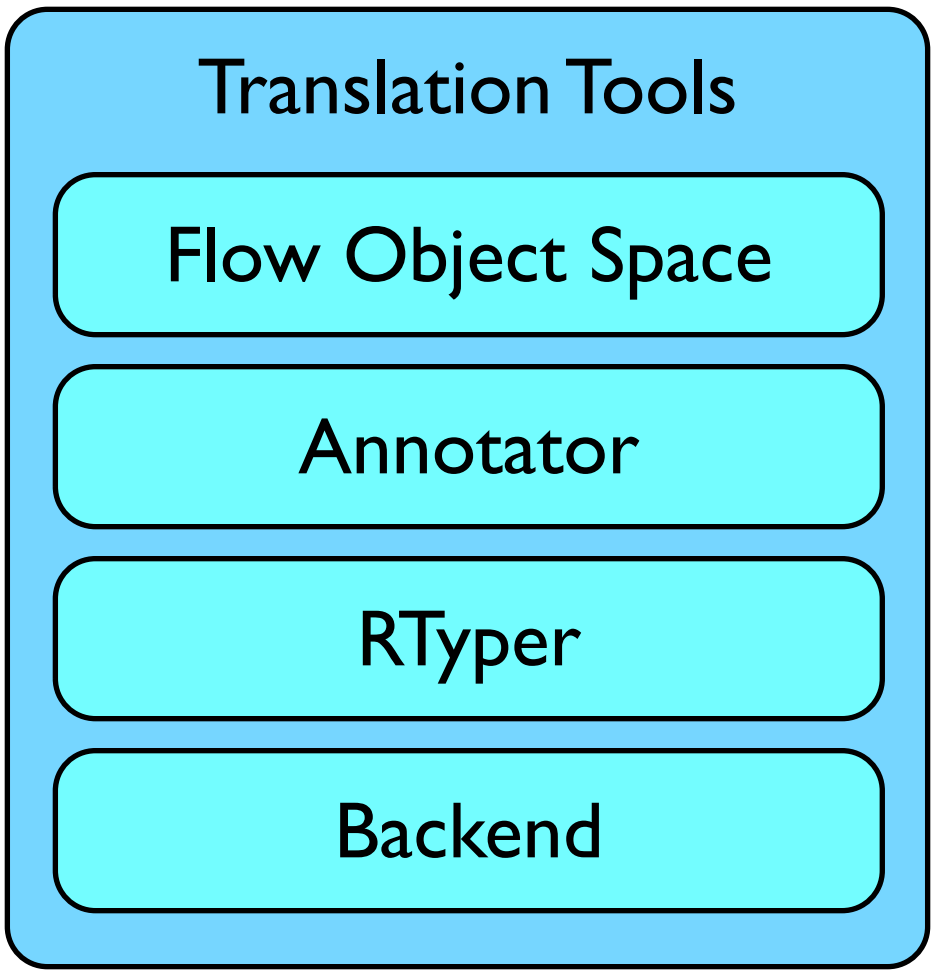
The Standard Interpreter



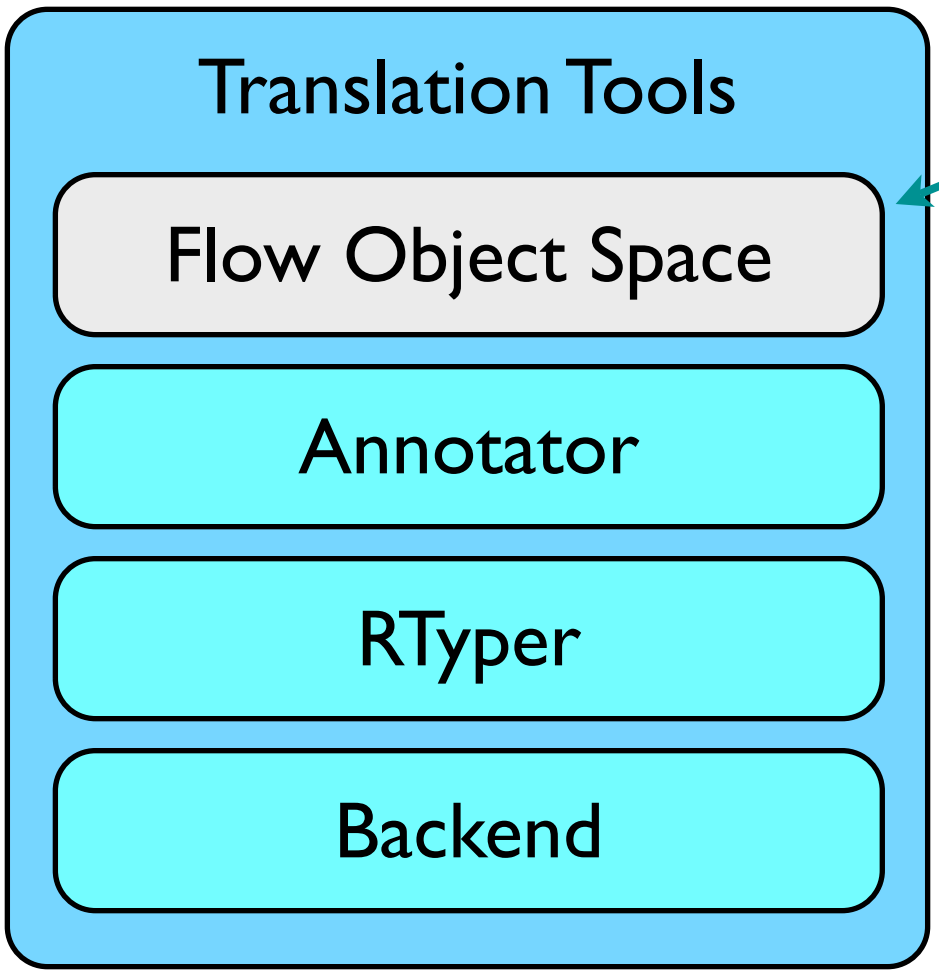
The standard interpreter is pretty stable now, implementing Python 2.4.3 (and some 2.5 features),

Some work to come on the parser/compiler and logic variable integration

Translation Tools



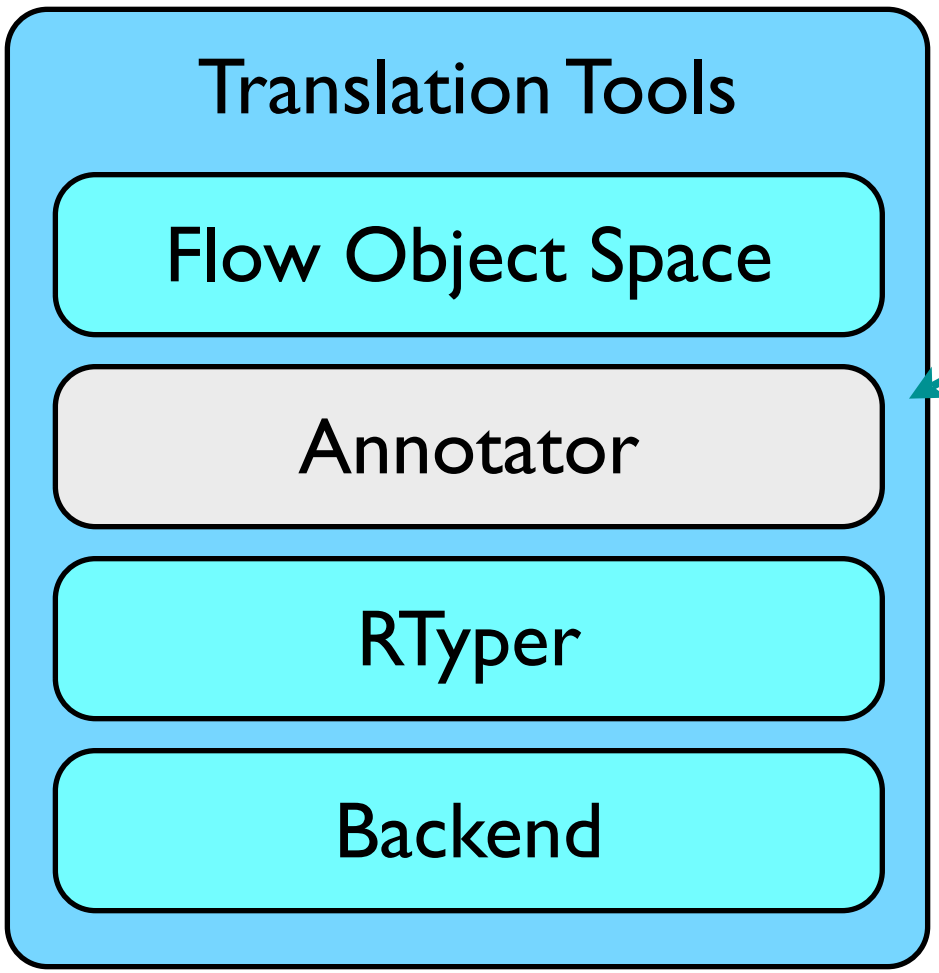
Translation Tools



Analyzes a single code object to deduce control flow

We have a funky pygame flow graph viewer that we use to view these flow graphs
(demo)

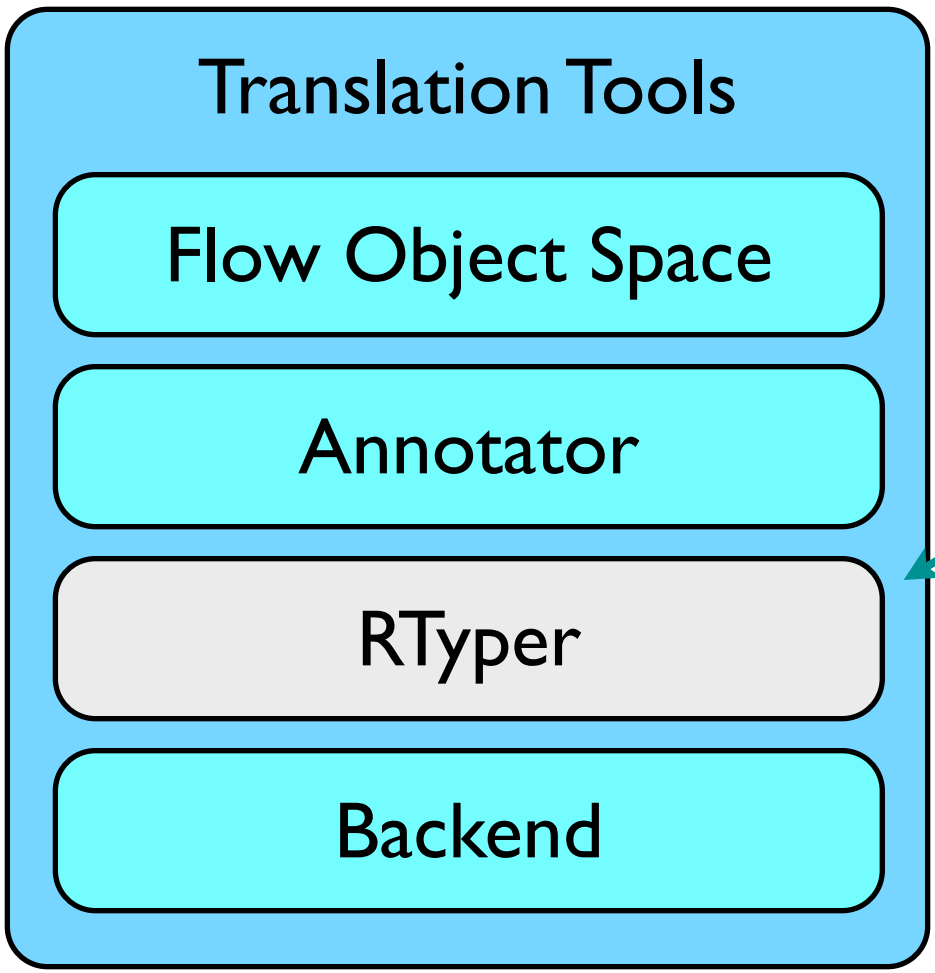
Translation Tools



Analyzes an *entire program* to deduce type and other information

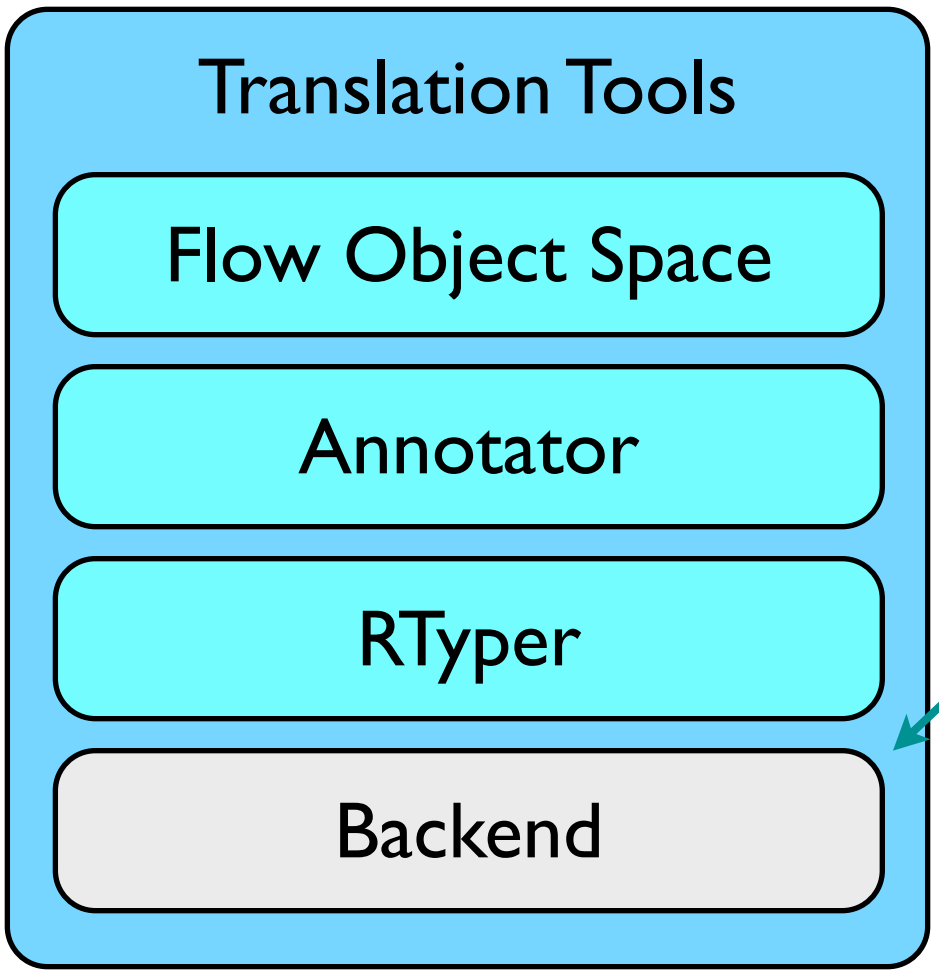
Uses abstract interpretation, rescheduling and other funky stuff

Translation Tools



Uses the information found by the annotator to decide how to lay out the types used by the input program in memory, and translates high level operations to lower level more pointer-ish operations

Translation Tools



Translates low level operations and types from the RTyper to (currently) C, JavaScript or LLVM code

Sounds like it should be easy, in fact a bit painful

The Flow Model

- Without going into details of how the Flow Object Space works, it produces a control flow graph of a code object
- Values are either `Variable`s or `Constant`s
- Operations are described by `SpaceOperation`s like `"add"`
- `SpaceOperation`s live in `Block`s which are connected by `Link`s

The Flow Model

- `SpaceOperation`s have an `opname`, a `result` variable and a list of `args`
- A couple of examples:
  - `z = x.y` → `SpaceOperation("getattr", [v_x, Constant("y")], v_z)`
  - `c = a + b` → `SpaceOperation("add", [v_a, v_b], v_c)`

The Annotator

- Type annotation is a fairly widely known concept – it associates variables with information about which values they might take at run time
- An unusual feature of PyPy's approach is that the annotator works on live objects
- This means it never sees initialization code, so that can use `exec` and other insane tricks

The Annotator

- Works by abstractly interpreting the control flow graphs produced by the flow analysis
- Annotation starts at a given entry point and discovers as it proceeds which functions may be called by the input program
- Read "Compiling dynamic language implementations" on the web site for more than is on these slides

The Annotator

- Does not modify the graphs; end result is essentially a big dictionary mapping `Variable`s to instances of a subclass of `SomeObject`
- Important subclasses are `SomeInteger`, `SomeList`, `SomeInstance`, `SomePBC` ("some pre-built constant", includes classes and functions)

The RTyper

- An apology: "RTyper" is a pretty bad name – just treat it as a random atomic identifier
- Performs "representation selection" and converts high-level operations to low-level
- Potentially can target a C-ish, pointer-using language or an OO language like Java or Smalltalk with classes and instances (OO backend not yet complete)

Representation Selection

- The fact that the annotator performs a global analysis gives us a novel opportunity
- For example, in:

```
l = range(10)
for x in l: print l
```

can represent the return value of range as just start/stop/step, but if we know the return value of range() is going to be mutated we just return a normal list

lltypes

- In PyPy, an *instance* of (a subclass of) the class `LowLevelType` describes a C-like *type* – a structure or array type, a pointer or a primitive type such as integer or float
- The RTyper attaches a `concretetype` to each `Variable` and `Constant` in each annotated control flow graph

Translating High Level to Low Level

- Many high level operations apply to different types; for example you can `"add"` strings, floats or integers and continually having to distinguish is annoying
- Better to have monomorphic operations `int_add`, `float_add`, `str_add` (well...)
- Some operations are more complex, e.g. instantiation of a class

Translating High Level to Low Level

- We saw that the code "`z = x + y`" becomes "`SpaceOperation("add", [v_x, v_y], v_z)`"
- Assuming that `v_x` and `v_y` (and thus `v_z`) are annotated as SomeInteger, then:
  - `v_x`, `v_y`, `v_z` will get a `concretetype` of `Signed`
  - the `"add"` operation will be replaced with an `"int_add"` operation

The Backend(s)

- Maintained backends: C, JavaScript(!) and LLVM (Smalltalk and CLI on the way)
- All proceed in two phases:
  - Traverse the forest of rtyped graphs, computing names for everything
  - Spit out the code

Status – what works

- The Standard Interpreter works well
- The translation tools work
  - C and LLVM backends well supported
  - JavaScript backend works, but not for all of PyPy
- The C backend supports ‘stackless’ features – coroutines and tasklets

Status – what works

- The C backend supports three garbage collection strategies:
  - reference counting,
  - using the conservative Boehm-Demers-Weiser collector
  - a precise mark and sweep collector we wrote

What we're working on now

- The Just-In-Time compiler – early stages, works for a very simple language
- More home-grown GCs (e.g. a semispace copying collector) and more GCs for LLVM
- Logic programming – some working code, interface and integration in progress

What we're working on now

- "rctypes", a uniform way of calling external functions based on the now-standard "ctypes" module for CPython
- CLI (.NET) and Smalltalk backends
- supporting stackless features in other backends