PyPy
Crash Course/Sprint Intro

Michael Hudson, based on Holger's old intro

25 August 2006

Something to look at if I'm too boring

- "getting started" has a lot of good stuff, including where to get the source, links to subversion clients and entry points:

http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

What is PyPy?

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

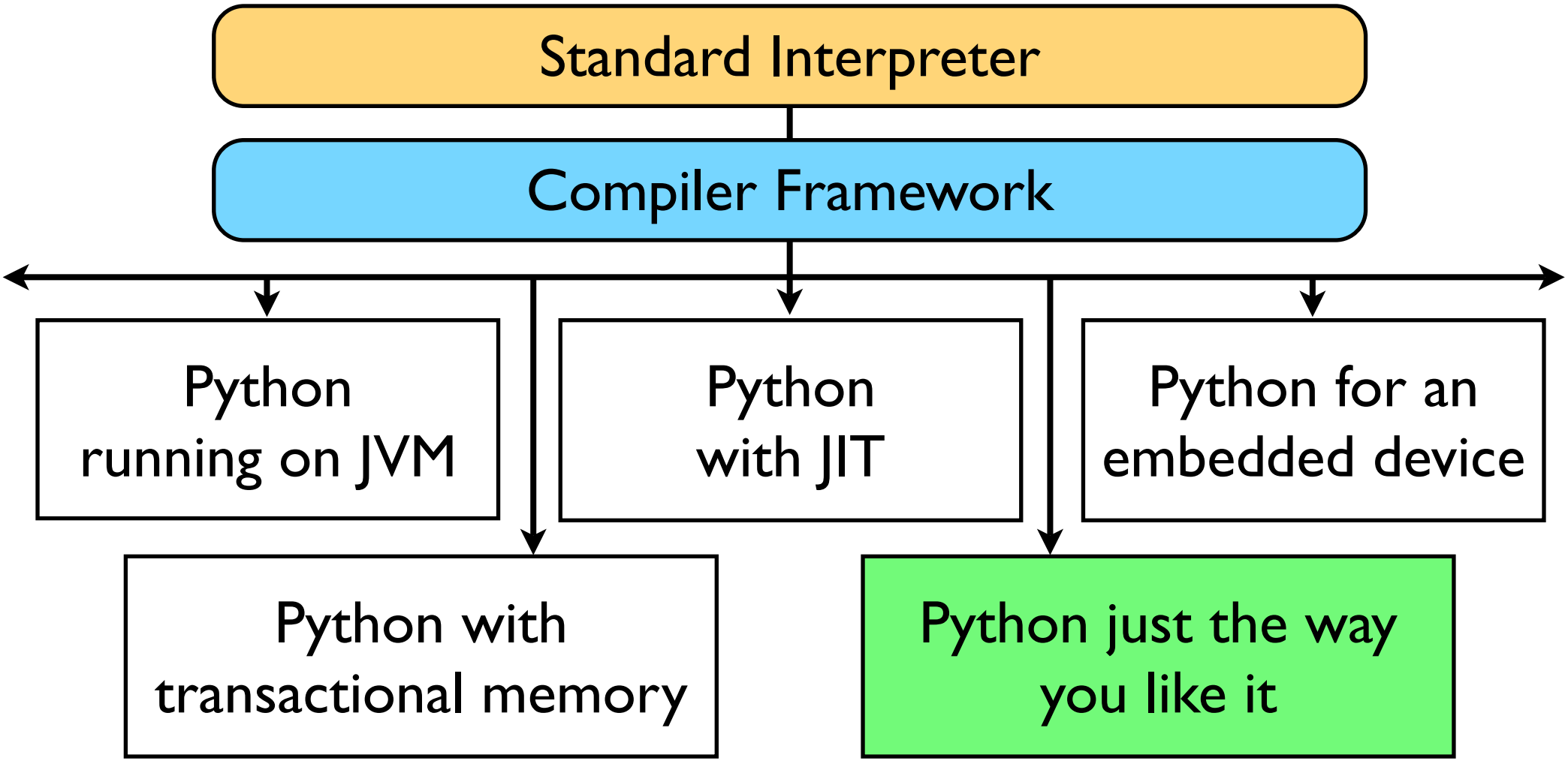
Motivation

- 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)

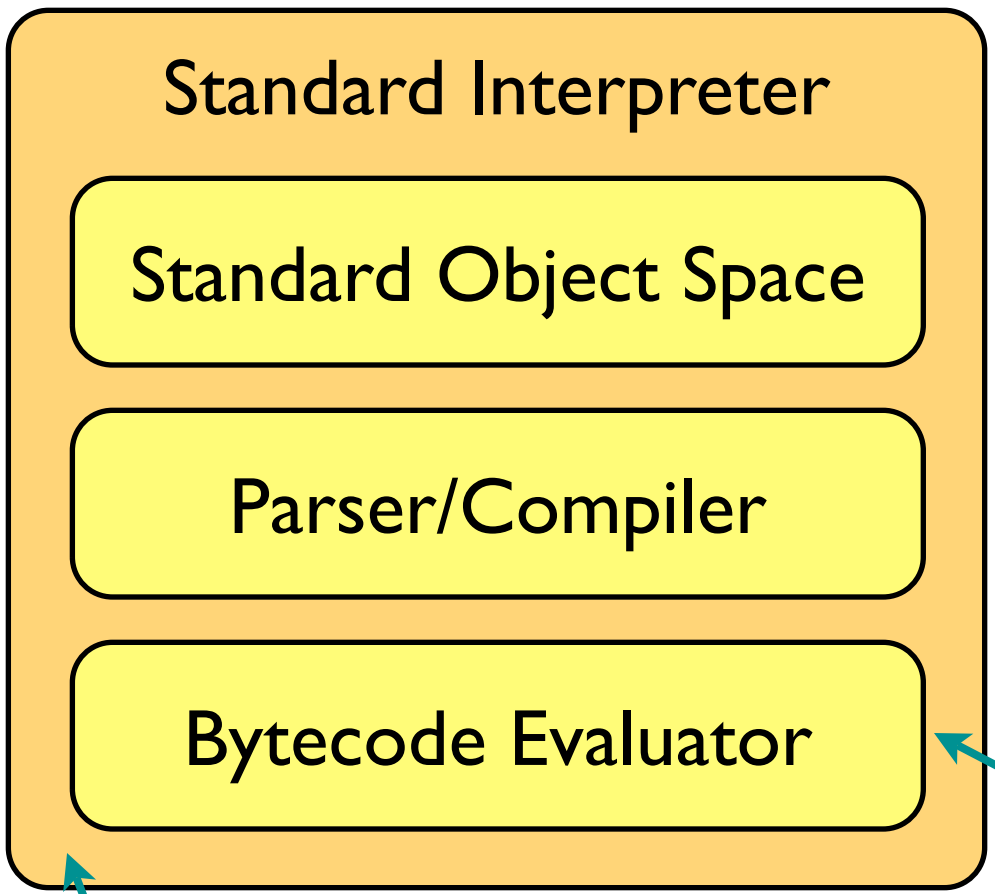
Lofty goals, but first...

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

Enter the PyPy platform



Standard Interpreter



The Standard Interpreter does roughly the same job as CPython, which can be divided into the same parts with sufficient imagination – which is hardly a coincidence

independent of object space implementation, which is important

all written in RPython

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

Some Jargon

- There are many levels in PyPy
- Two of the more important, defined in terms of running PyPy on top of CPython:
  - “interp-level”: code that will be executed by CPython (and maybe get translated to C)
  - “app-level”: code that will be executed by PyPy’s bytecode evaluator, not CPython’s

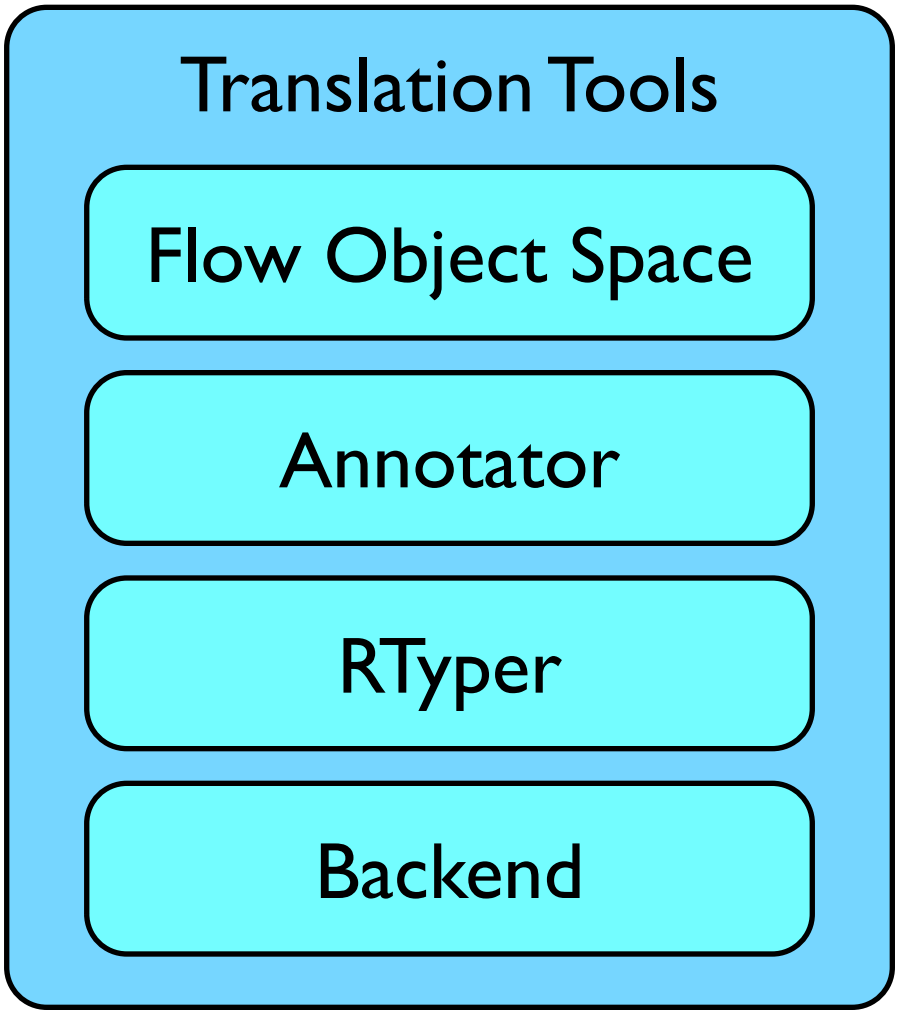
Interp/App-level

- The standard interpreter is written in a mixture of app-level and interp-level code
- Can call from one to the other
- Advantages of app-level: can use full power of Python, less code
- Advantages of interp-level: faster, closer to the metal, can run "early"

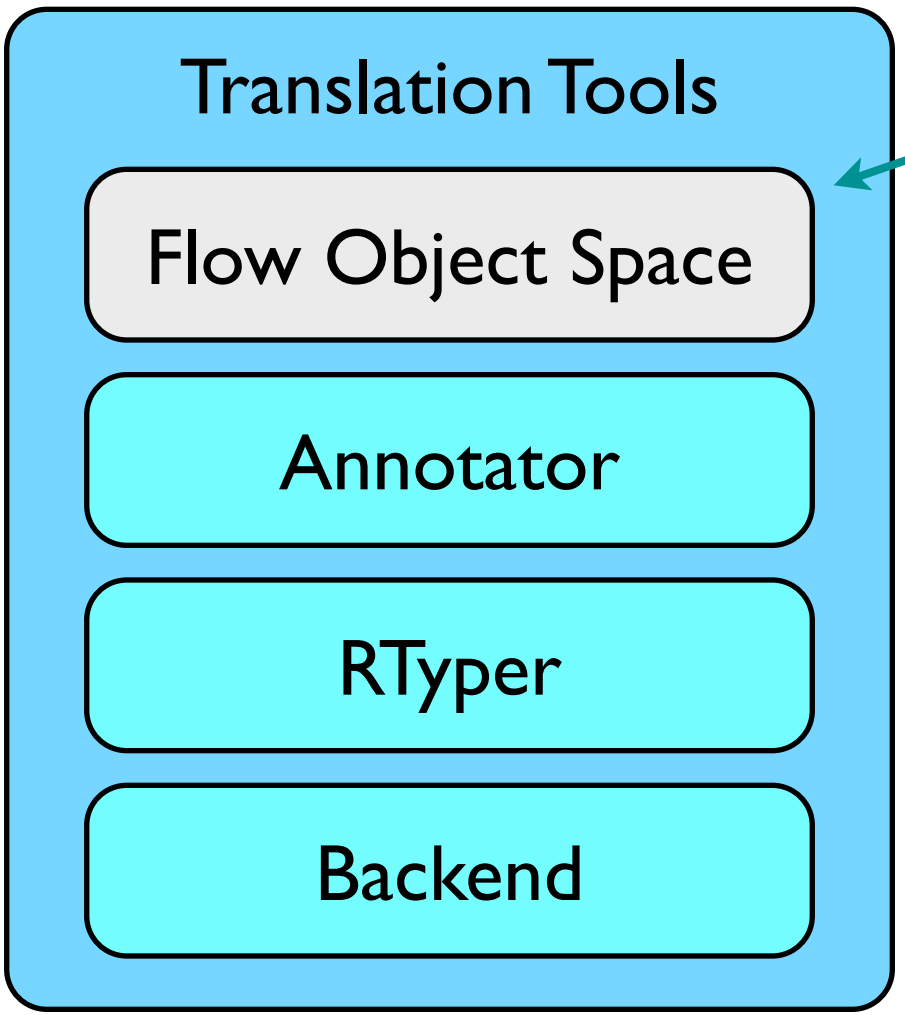
Status

- Standard Interpreter very complete, passes a large majority (>95%) of CPython's core regression tests
- Standard Object Space and bytecode evaluator now very stable
- Only remaining work: make it faster! (not *necessarily* by working on its code, though)

Translation Tools



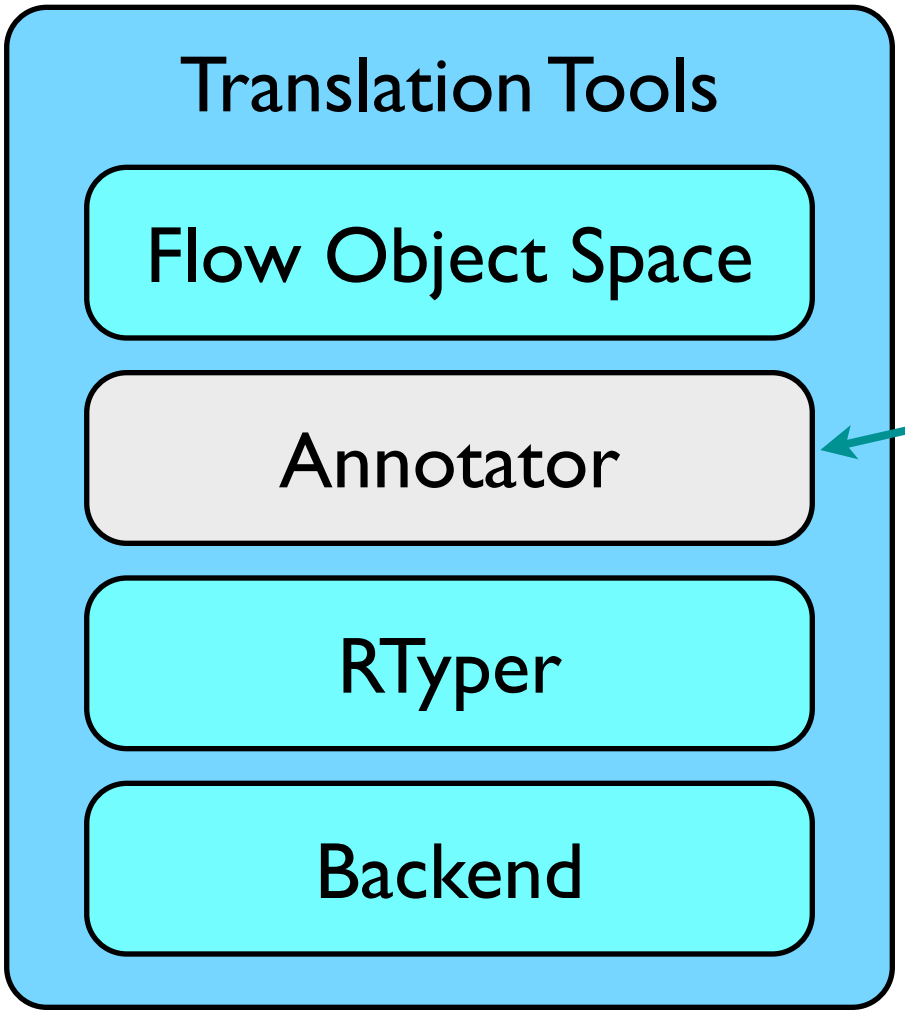
Translation Tools



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)

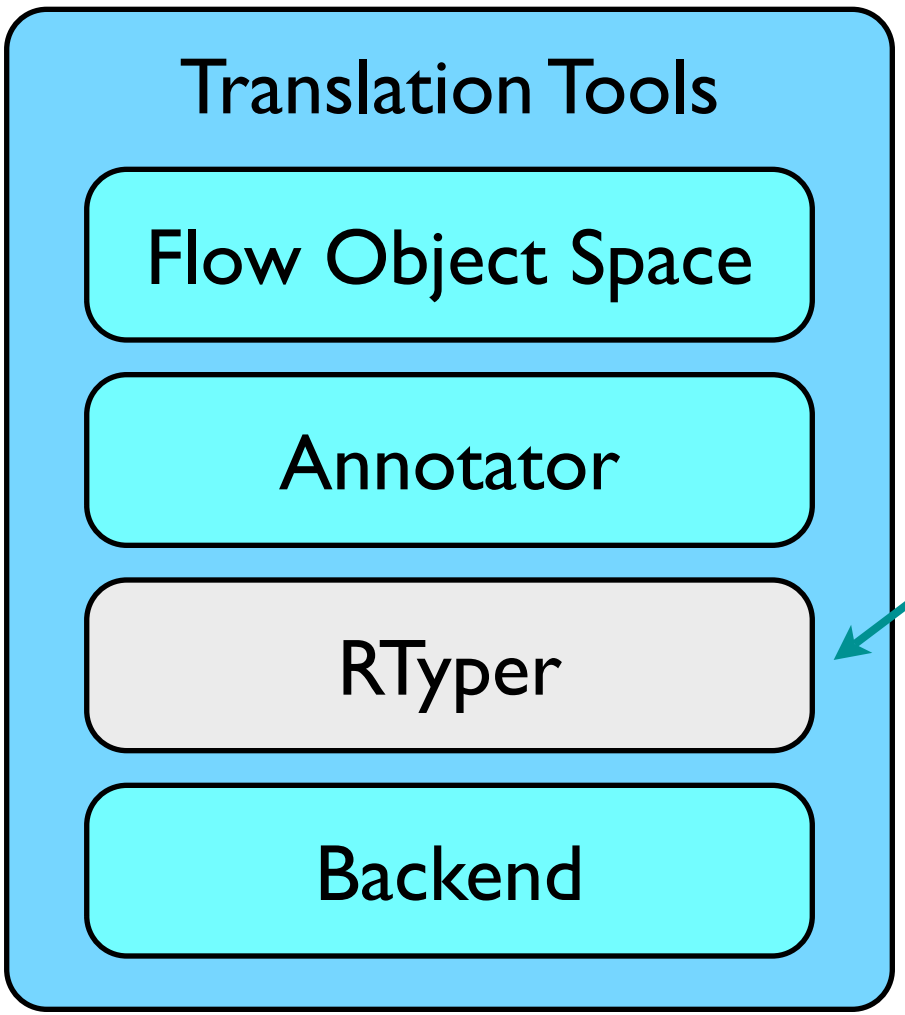
Translation Tools



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

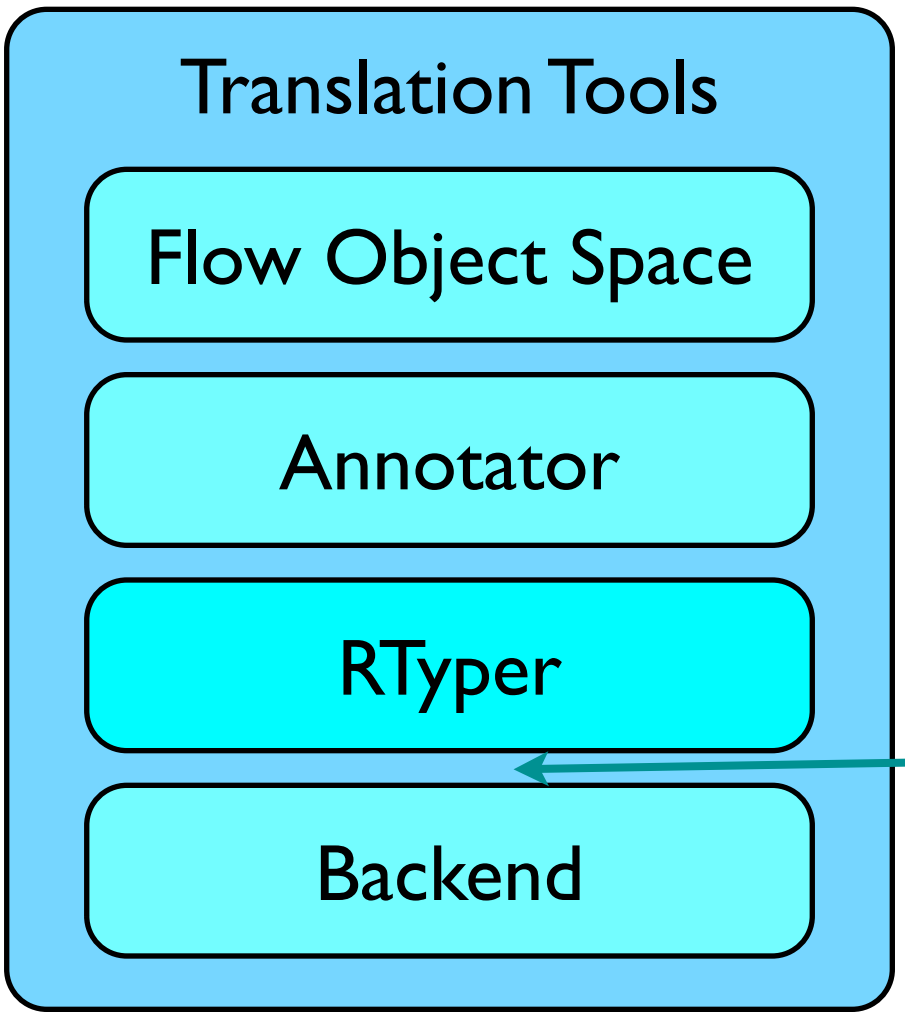
Uses abstract interpretation, rescheduling and other funky stuff

Translation Tools



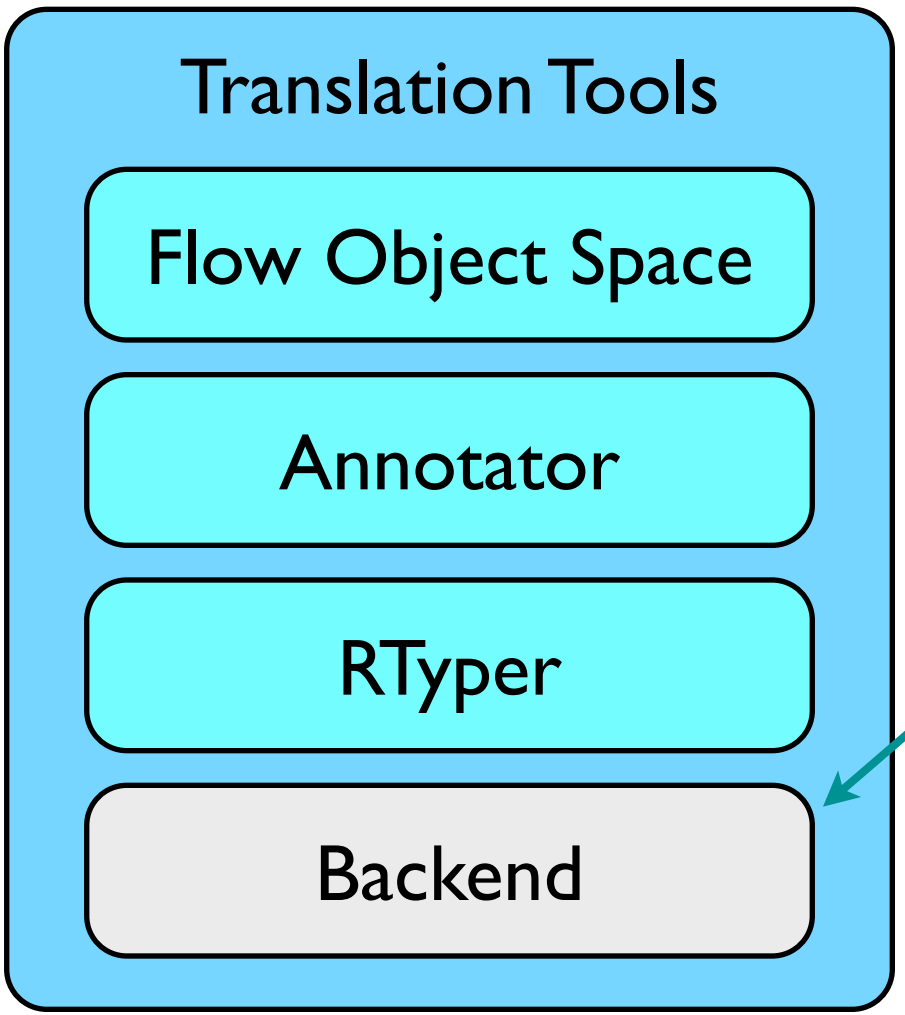
Uses the information found by the annotator to decide how to lay out the types used by the input program in memory, and translates high level operations to lower level more pointer-ish operations

Translation Tools



"Stuff" happens here – optional optimizations, the "stackless transform", insertion of explicit exception handling and memory management/GC code

Translation Tools



Translates low level operations and types from the RTyper to C, LLVM, CLI, JavaScript, … code

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

Flow Analysis

- Control flow graphs are built by abstractly interpreting the code object using the standard interpreter's bytecode evaluator in an *abstract domain* – the Flow Object Space
- Uses state-saving tricks to consider both parts of a branch
- Don't worry about how it works – what it produces is much more relevant today

Flow Analysis

- Control flow graphs are built by abstractly interpreting the code object using the standard interpreter's bytecode evaluator in an *abstract domain* – the Flow Object Space
- Uses state-saving tricks to consider both parts of a branch
- Don't worry about how it works – what it produces is much more relevant today

The Flow Model

- All defined in `pypy.objspace.flow.model`
- Values are either `Variable`s or `Constant`s
- A function's control flow graph is described by a `FunctionGraph`
- This contains `Block`s and `Link`s
- Blocks contain a list of `SpaceOperation`s

The Flow Model

- `SpaceOperation`s have an `opname`, a `result` variable and a list of `args`.
- The graph is naturally produced in *Static Single Information* form, which is very handy for later analysis
- This means each `Variable` is used in exactly one block and is renamed if it needs to be passed across a link

The Flow Model

- Some examples:
  - `z=x.y` → `SpaceOperation("getattr", [v_x, Constant("y")], v_z)`
  - `c=a+b` → `SpaceOperation("add", [v_a, v_b], v_c)`
  - `t=f(u)` → `SpaceOperation("simple_call", [Constant(f), v_u], v_t)`

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 (a popular phrase :) the control flow graphs produced by the flow analysis
- Annotation starts at an entry point and discovers as it proceeds which functions are needed
- Read "Compiling dynamic language implementations" on the web site for more than is on these slides

The Annotator

- Works a block at a time, maintaining a pile of blocks that need analysis
- As analysis proceeds, the information about a block may get invalidated – in other words, the annotation reschedules as needed
- A fix-point approach:

```
while work_to_do: do_work()
```

The Annotation Model

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

The RTyper

- RTyper takes as input an annotated RPython program (e.g. our Python implementation)
- Performs "representation selection" and converts high-level operations to low-level
- Can target a C-ish language or an object oriented platform such as .NET/CLI or Squeak (for real) or the JVM (would be nice)

The RTyper

- Originally we tried to do the job the RTyper does at the same time as source generation
- Failed
- Miserably
- It does a job that's not part of the standard "Introduction to Compilers 101" course

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

- `pypy.rpython.lltypesystem.lltype` contains a collection of Python classes that describe (and implement!) a C-like memory model with `Struct`s, `Array`s, `Pointer`s, `Signed`s (integers), `GcStruct`s, `Float`s... all subclasses of `LowLevelType`
- Convention is that variables holding instances of lltypes are in ALLCAPS:

```
TYPE = GcStruct("T", ("x", Signed))
```

Representation Selection

- The RTyper attaches an attribute "`concretetype`" containing an lltype to all `Constant`s and `Variable`s
- During the process of RTyping, however, an instance of `pypy.rpython.rmodel.Repr` is created and associated with each `Variable`'s annotation, which knows how to translate operations involving the `Variable`

Translating High Level to Low Level

- The high level operations such as "`add`" apply to different types; 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

- For each operation:
  - an instance of a subclass of `Repr` is created/found for each argument's annotation
  - and these are asked what low-level operation(s) the high level operation should be replaced with

Translating High Level to Low Level, example

- Start with, say, `SpaceOperation("add", [v_x, v_y], v_z)` with `v_x` and `v_y` (and `v_z`) all annotated as `SomeInteger()`
- `rtyper.getrepr(v_x)` returns an instance of `IntegerRepr` (same for `v_y`)
- We end up calling a method `rtype_add` on a "`pairtype`" which makes a "`int_add`" operation

Source Generation

- Maintained backends: C, LLVM, JavaScript, .NET/CLI, Squeak, Common Lisp (in roughly descending order of maintainedness)
- All proceed in two phases:
  - Traverse the forest of rtyped graphs, computing names for everything
  - Spit out the code

Dealing with the world

- One of Python's strengths is the ease with which external C libraries can be wrapped
- Existing C extensions cannot work with PyPy
- But don't need C any more for this job – ctypes!

rctypes

- rctypes is our name for the code which allows RPython programs to use ctypes (in a sufficiently static way)
- Example:

```
time_t = ctypes_platform.getsimpletype(
    'time_t', '#include <time.h>', c_long)
time = libc.time
time.argtypes = [POINTER(time_t)]
time.restype = time_t
```

The Extension Compiler

- In addition to using this to wrap external libraries for PyPy, it can be used to do the same for CPython
- This is `bin/compilemodule`
- It's not yet "the nicest thing in the world"

Things I haven't talked about

- Garbage collection policies
- Stackless
- Backend optimizations
- The JIT
- Exceptions in the flow model
- Specialization of functions/annotation policies
- Low level helpers
- Constraint solving
- Geninterp
- Pairtypes
- Multimethods in the Standard Object Space

Coding Issues

- See http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html
- But in summary: PEP 8, test driven development (using py.test)
- Away from sprints, much talking in #pypy

Thank you for your time!

Subversion awaits!