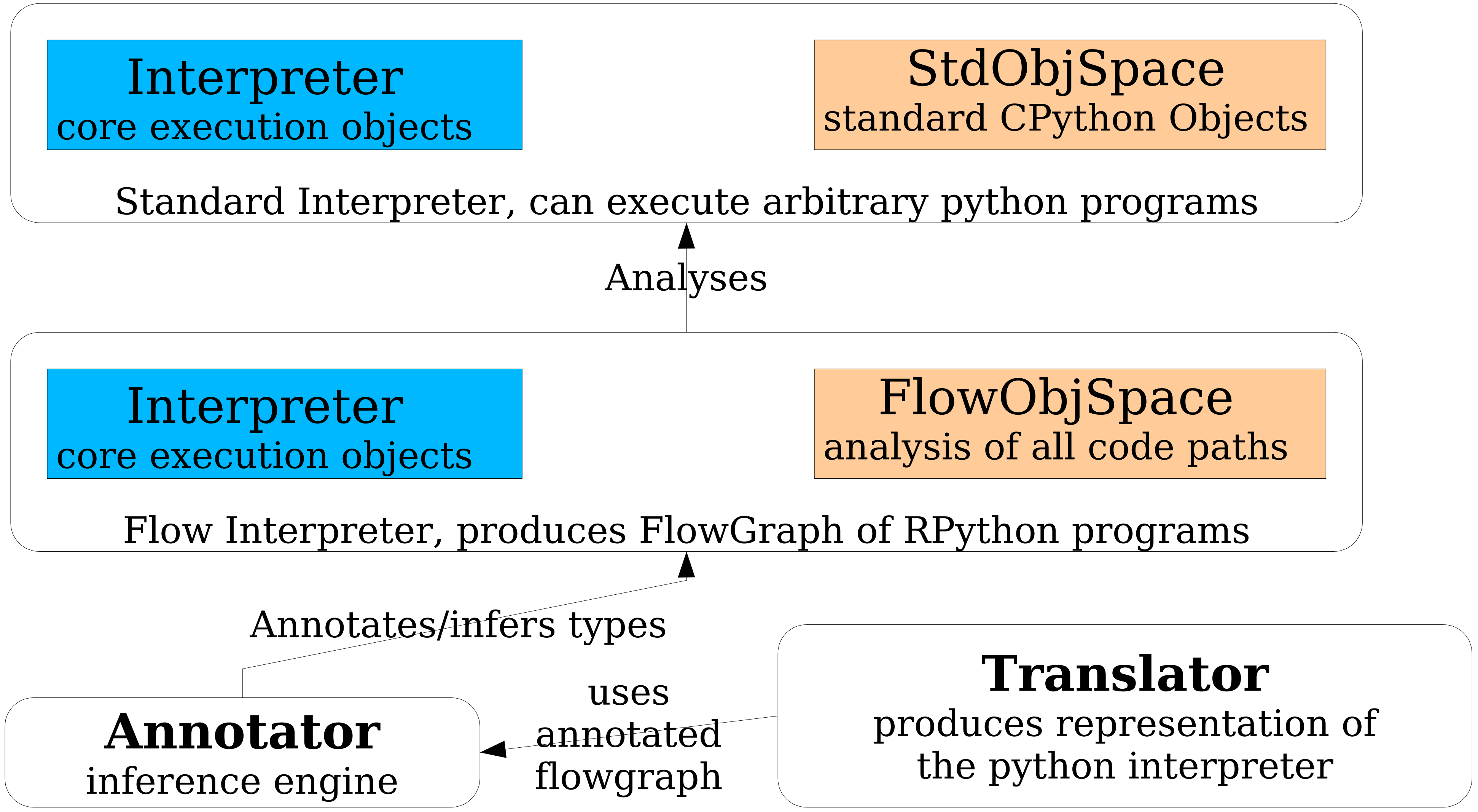
PyPy

Architecture overview

Holger Krekel
14th December 2003

Current “Big Picture”



The “plain” interpreter

- Has no knowledge about implementation, layout, methods or other details of application level objects, it passes them as black boxes to an ObjectSpace who is responsible for working and performing operations with them.
- **The plain interpreter provides core execution objects such as Functions, Frame and Code Objects which are exposed to application level code (introspection).**
- **Gateway** instances unify invocation of interp-level and app-level code objects. Gateways unify two basic kinds of Code objects:
  - PyCode (for running application-level functions/code)
  - BuiltinCode (for running interpreter-level functions)
- A Frame instance reads bytecodes and invokes their implementations which in turn dispatch operations to a frame's ObjectSpace

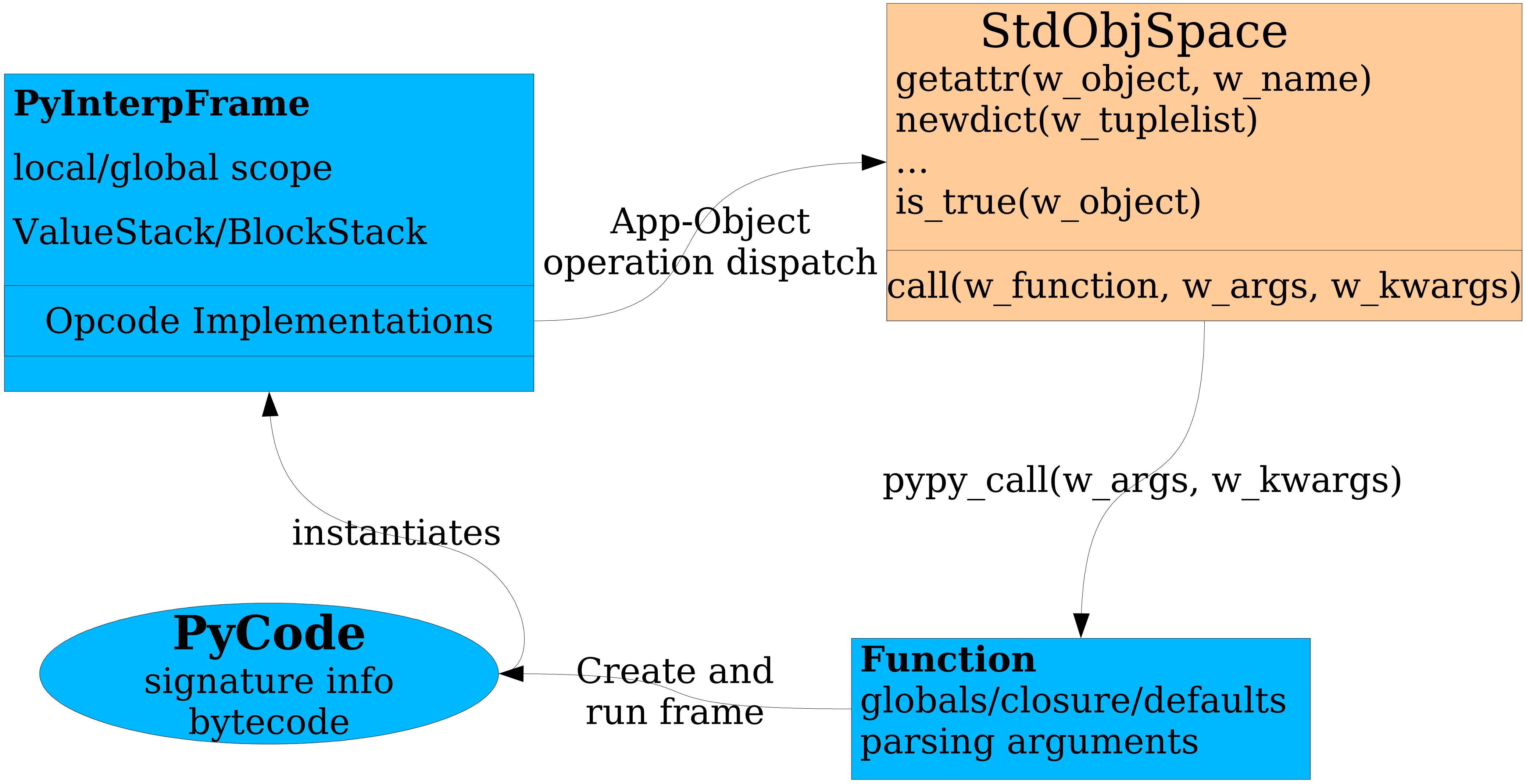
Standard Objectspace

- Implements standard object/behaviour as per language specification
- leverages a “multimethod” mechanism to allow the interpreter uniform access to object-operations. Remember that the interpreter doesn't know anything about applevel objects and especially not about their “types”.
- Does not rely on “type” objects to dispatch and implement operations. Each multimethod like getattr/call has a registry for (interpreter-level) **implementation classes** (more on that from Armin in the types.py proposal)
- All multimethods take and return app-level objects
- Only **is_true(w_obj)** delivers an interpreter-level boolean to enable the interpreter to test conditions and do branches based on the state of application level objects
- --> Interactive Session (Functions, Gateways, Code, ...)

A word about frames

- Frames generally encapsulate the complete execution of a function. They manage the valuestack, blockstack, scopes and interpretation of a function's bytecode
- In PyPy Frames are created by code objects because a frame needs to interpret a code object's bytecodes
- **PyInterpFrame** contains logic for the standard set of bytecodes
- Generators and nested scope aspects can be mixed into an PyInterpFrame and don't affect the rest of the code
- --> look into PyCode.create_frame()

Interpreting Applevel-functions



Applevel / Interplevel

- Application-Level objects are represented
  - "plainly" in application level code
  - as wrapped objects on interpreter level code ("w_*")
- Interpreter-Level "core execution" objects such as Function, Code, Module objects are represented
  - "plainly" in interpreter level code
  - "special-wrapped" on application-level, i.e. An ObjectSpace is required to dispatch operations on wrapped interp-level objects to their pypy_* protocol. (Note that we still need to fix introspection/access to attributes in such a way that "dis.dis(dis.dis)" works).
- Application level Exceptions are currently represented at intepreter-level as "OperationErrors" containing a wrapped app-level Exception. This allows unified catching of application-level errors (note pending renaming!)

Applevel is preferable (until it's not)

- Parts of the interpreter and objectspaces are written with application-level functions! Of course there is an implicit restriction here ...
- **The “involved-code-closure” of interpreting an app-level function should not contain “itself”, otherwise:infinite recursion**
- Writing parts of the interpreter and object spaces at applevel is less tedious to write but sometimes harder to debug (but then again, applevel code tends to have less bugs)
- Note that the implementation of the objectspaces freely “mixes” interp-level and applevel code.
- So **the terms and concepts behind “interpreter-level and application-level functions/objects” are pervasive througout the whole pypy codebase and refer to certain views of a running/interpreted program**

FlowObjectSpace

- And now for something completely different ...
- FlowObjSpace Implements (limited) abstract/symbolic interpretation of an RPython program
- reuses the normal PyInterpFrame for interpreting code objects (dispatching to ObjectSpace operations)
- Records and transforms objectspace operations into a flowgraph representation (without affecting the interpreter)
- Follows all possible execution pathes at least once. This is done by “splitting the world” at each **is_true()** and returning True in one world and False in another. A “world” is implemented by continuationish mechanisms.
- When the “FlowInterpreter” analyses the “StdInterpreter” execution it will generate a flowgraph which can be annotated and translated to other (low-level) languages.
- --> Interactive Session (graph creation/display, annotation)

Annotation

- Currently works on the flowgraph, but may later get interweaved with the FlowInterpreter (because the FlowObjSpace can not always make the best decisions regarding specizializing/generalizing frame-states).
- Values flowing through a program are “naturally” annotated with space-operations so we don't need to come up with yet another type-model but can just reuse existing terms …
- Currently we have a hackish but working annotation in translator/ann*.py and a half-done but more general “annotation engine” in annotation/*.py
- More on this later in the week :-)

Translation

- First we aim to implement static translation to C-sourcecode (or another static language) by doing the following steps:
  - Analyze the StdInterpreter (Interpreter+StdObjSpace) with the help of FlowInterpreter + Annotation/type inference
  - Generate a low-level representation of the StdInterpreter from the annotated flow model
  - Note: we probably need a small “native” minimal runtime library which the generated code will adapt to in order to provide IO access to the host system ...
- Design goal: low-level aspects such as memory management, threading models, speed-memory tradeoffs are separate concerns which don't change our “StdInterpreter” at all.

That's it ...

- Not yet :-)

Coding Style / Testing

- Make sure you read doc/dev/coding-style.txt, especially our naming conventions (namespace/directory/filenames are all singular, functions lowercase, classes CamelCase, don't put “unneccessary underscores” in your function names and never in class or directory names)
- Try to write sensible tests for everything you change in PyPy, this is especially important because we constantly improve and refactor the architecture which would become impossible and quite unpleasant without proper tests!
- Yes, we know there are a lot of areas that seem somewhat vague (like bootstrapping, handling exceptions, translation) but fortunately Python lets us stay vague in some areas and still allows us to get our basic concepts running :-)