



IST FP6-004779

PYPY

**Researching a Highly Flexible and Modular Language Platform and
Implementing it by Leveraging the Open Source Python Lanugage and
Community**

STREP

IST Priority 2

**D05.1: Compiling Dynamic Language
Implementations**

Due date of deliverable: August 2005

Actual Submission date: 23th December 2005

Start date of Project: 1st December 2005

Duration: 2 years

Lead Contractor of this WP: HHU

Authors: Armin Rigo (HHU), Michael Hudson (HHU), Samuele Pedroni (AB Strakt)

Revision: final

**Project co-funded by the European Commission within the Sixth Framework
Programme (2002-2006)**

Dissemination Level: PU (Public)



Abstract

This document presents PyPy's analysis and compilation toolchain, which is used to translate RPython programs like PyPy's standard interpreter into stand-alone efficient executables.

Modern dynamic languages pose difficulties to program analysis; we debate them and introduce our basic approach. We then give an extended theoretical description of our toolchain and its design, motivated by advanced flexibility goals: we can indeed target a wide range of run-time platforms, inserting the necessary low-level details (e.g. memory management) as part of the translation process; and for each target we can experiment with a number of additional translation aspects like execution models (e.g. green microthreads and coroutines).



Contents

1	Executive Summary	5
2	Introduction	5
2.1	Purpose of this Document	5
2.2	Scope of this Document	5
2.3	Related Documents	5
3	The analysis of dynamic languages	6
3.1	No Declarations	6
3.2	The analysis of live programs	7
4	Concrete and abstract interpretation	8
4.1	Object Spaces	8
4.2	Abstract interpretation	8
4.3	The PyPy analysis toolchain	9
4.4	Motivating our architecture	9
5	Flow Object Space	11
5.1	Construction of flow graphs	11
5.2	Branching	12
5.3	Dynamic merging	13
5.4	Geninterp	14
6	Annotator	16
6.1	Static model	16
6.2	Flow graph model	17
6.3	Annotation model	17
6.4	Rules	20
6.5	Mutable objects	22
6.6	Pre-built constants	24
6.7	Classes and instances	25
6.8	Calls	26
6.9	Termination and soundness	27
6.9.1	Generalisation	27
6.9.2	Termination	30
6.9.3	Soundness	31



6.9.4 Complexity	34
6.10 Non-static aspects and extensions	34
6.10.1 Specialization	34
6.10.2 Concrete mode execution	35
6.10.3 Constant propagation	35
6.10.4 Narrowing	36
6.10.5 Termination with non-static aspects	37
7 Code Generation	38
7.1 RTyper	38
7.1.1 Low-level flow graphs	38
7.1.2 Representations	39
7.1.3 Helpers and LLPython	39
7.2 The back-ends	40
8 Conclusion	41
8.1 Limits of Static analysis	41
8.2 Test-driven development	41
9 Glossary of Abbreviations	43



1 Executive Summary

PyPy's tool-chain is able to translate a subset of the Python language, RPython, by performing whole-program type inference and then specializing to other language backends. In particular, the tool-chain allows translation of our full Python implementation, a "high-level description" of the Python language, into lower-level targets.

Translating our high-level Python description - itself expressed in RPython and Python - provides increased flexibility, because the implementation is analysable, amenable to transformation, optimisations and retargettable. Lower-level decisions can be postponed and their choice kept independently interchangeable, and a feature like making recursion only limited by the amount of memory can be added without touching the high level implementation.

Our approach to translating and subsetting of dynamic languages for system programming and their own implementation leverages abstract interpretation and strikes a good balance between simplicity and expressiveness. It reuses part of the bytecode interpreter for the analysis and in our experience gives a compact approach and framework that is reasonably malleable and maintainable.

This document presents a comprehensive theoretical account of our approach so far.

2 Introduction

2.1 Purpose of this Document

This document describes and motivates in technical and theoretical detail our approach to the translation of Python - more precisely a subset thereof - to low-level targets for use in its own implementation and for system programming.

2.2 Scope of this Document

This document describes our approach to translation as implemented by the tool-chain shipped with PyPy releases 0.7 and 0.8 and which enabled with 0.7 to achieve a self-contained PyPy able to run without CPython as a native executable. The details of our interpreter implementation and how low-level aspects are woven through translation are explained in other documents, D4.2, D4.3 and D5.3, D5.4 respectively.

2.3 Related Documents

Although this document essentially presents an independant theoretical point of view on our approach, cross-reading the following deliverables and documentation is recommended:

- D04.2 Complete Python implementation running on top of CPython
- D04.4 Release PyPy as a research tool for experimental language enhancements
- D05.2 A compiled, self-contained version of PyPy
- D05.3 Publish on implementation with translation aspects
- D05.4 Publish on encapsulating low level language aspects



3 The analysis of dynamic languages

Dynamic languages are definitely not new on the computing scene. However, new conditions like increased computing power and designs driven by larger communities have enabled the emergence of new aspects in the recent members of the family, or at least made them more practical than they previously were. The following aspects in particular are typical not only of Python but of most modern dynamic languages:

- The driving force is not minimalist elegance. It is a balance between elegance and practicality, and rather un-minimalist -- the feature sets built into languages tend to be relatively large and grow (to some extent, depending on the community driving the evolution of the language).
- High abstractions and theoretically powerful low-level primitives are generally ruled out in favour of a larger number of features that try to cover the most common use cases. In this respect, one could even regard these languages as mere libraries on top of some simpler (unspecified) language.
- Language design is no longer driven by a desire to enable high performance; any feature straightforward enough to implement in an interpreter is a candidate for being accepted. As a result, compilation and most kinds of static inference are made impossible due to this dynamism (or at best tedious, due to the complexity of the language).

3.1 No Declarations

The notion of "declaration", central in compiled languages, is entirely missing in Python. There is no aspect of a program that must be declared; the complete program is built and run by executing statements. Some of these statements have a declarative look and feel; for example, some appear to be function or class declarations. Actually, they are merely statements that, when executed, build a function or class object. A reference to the new object is then stored in a namespace from where it can be accessed. Units of programs -- modules, whose source is one file each -- are similarly mere objects in memory, built on demand by some other module executing an `import` statement. Any such statement -- class construction or module import -- can be executed at any time during the execution of a program.

This point of view should help explain why analysis of a program is theoretically impossible: there is no declared structure to analyse. The program could for example build a class in completely different ways based on the results of NP-complete computations or external factors. This is not just a theoretical possibility but a regularly used feature: for example, the standard Python module `os.py` provides some OS-independent interface to OS-specific system calls, by importing internal OS-specific modules and completing it with substitute functions, as needed by the OS on which `os.py` turns out to be executed. Many large Python projects use custom import mechanisms to control exactly how and from where each module is loaded, by tampering with import hooks or just emulating parts of the `import` statement manually.

In addition, there are of course classical (and only partially true) arguments against compiling dynamic languages (there is an `eval` function that can execute arbitrary code, and introspection can change anything at run-time), but we consider the argument outlined above as more fundamental to the nature of dynamic languages.



3.2 The analysis of live programs

How can we perform some static analysis on a program written in a dynamic language while keeping to the spirit of [No Declarations](#), i.e. without imposing that the program be written in a static way in which these declarative-looking statements would actually *be* declarations?

The approach of (PyPy) is, first of all, to perform analysis on live programs in memory instead of dead source files. This means that the program to analyse is first fully imported and initialised, and once it has reached a state that is deemed advanced enough, we limit the amount of dynamism that is allowed *after this point* and we analyse the program's objects in memory. In some sense, we use the full Python as a preprocessor for a subset of the language, called RPython. Informally, RPython is Python without the operations and effects that are not supported by our analysis toolchain (e.g. class creation, and most non-local effects).

Of course, putting more efforts into the toolchain would allow us to support a larger subset of Python. We do not claim that our toolchain -- which we describe in the sequel of this paper -- is particularly advanced. To make our point, let us assume as given an analysis tool, which supports a given subset of a language. Then:

- Analysing dead source files is equivalent to giving up all dynamism (as far as unsupported by this tool). This is natural in the presence of static declarations.
- Analysing a frozen memory image of a program that we loaded and initialised is equivalent to giving up all dynamic after a certain point in time. This is natural in image-oriented environments like Smalltalk, where the program resides in memory and not in files in the first place.

Our approach goes further and analyses *live* programs in memory: the program is allowed to contain fully dynamic sections, as long as these sections are entered a *bounded* number of times. For example, the source code of the PyPy interpreter, which is itself written in this bounded-dynamism style, makes extensive use of the fact that it is possible to build new classes at any point in time -- not just during an initialisation phase -- as long as the number of new classes is bounded. For example, [interpreter/gateway.py](#) builds a custom wrapper class corresponding to each function that a particular variable can reference. There is a finite number of functions in total, so this can only create a finite number of extra wrapper classes. But the precise set of functions that need such a wrapper class is difficult to manually compute in advance. It would also be redundant to do so: indeed, it is part of the type inference tool's job to discover all functions that can reach each point in the program. In this case, whenever it discovers that a new function could reach the particular variable mentioned above, the analysis tool itself will invoke the class-building code in [interpreter/gateway.py](#) as part of the inference process. This triggers the building of the necessary wrapper class, implicitly extending the set of classes that need to be analysed.

This approach is derived from dynamic analysis techniques that can support unrestricted dynamic languages by falling back to a regular interpreter for unsupported features (e.g. (Psyco)). The above arguments should have shown why we think that being similarly able to fall back to regular interpretation for parts that cannot be understood is a central feature of the analysis of dynamic languages.



4 Concrete and abstract interpretation

4.1 Object Spaces

The semantics of Python can be roughly divided in two groups: the syntax of the language, which focuses on control flow aspects, and the object semantics, which define how various types of objects react to various operations and methods. As it is common in all languages of the family, both the syntactic elements and the object semantics are complex and at times complicated (as opposed to more classical languages that tend to subsume one aspect to the other: for example, Lisp's execution semantics are almost trivial).

This observation led us to the concept of *Object Space*. An interpreter can be divided in two non-trivial parts: one for handling compilation to and interpretation of pseudo-code (control flow aspects) and one implementing the object library's semantics. The former, called *bytecode interpreter*, considers objects as black boxes; any operation on objects requested by the bytecode is handled over to the object library, called *object space*. The point of this architecture is, precisely, that neither of these two components is trivial; separating them explicitly, with a well-defined interface in-between, allows each part to be reused independently. This is a major flexibility feature of PyPy: we can for example insert proxy object spaces in front of the real one, as in the [Thunk Object Space](#) which adds lazy evaluation of objects.

Note that the term "object space" has already been reused for other dynamic language implementations, e.g. such as this [post on the Perl 6 compiler mailing list](#).

4.2 Abstract interpretation

In the sequel of this paper, we will consider another application of the object space separation. The analysis we perform in PyPy is whole-program type inference. The analysis of the non-dynamic parts themselves is based on their [abstract interpretation](#). PyPy has an alternate object space called the [Flow Object Space](#), whose objects are empty placeholders. The over-simplified view is that to analyse a function, we bind its input arguments to such placeholders, and execute the function -- i.e. let the interpreter follow its bytecode and invoke the object space for each operations, one by one. The Flow object space records each operation when it is issued, and returns a new placeholder as a result. At the end, the list of recorded operations, along with the involved placeholders, gives an assembler-like view of what the function performs.

The global picture is then to run the program while switching between the flow object space for static enough functions, and a standard, concrete object space for functions or initialisations requiring the full dynamism.

If, for example, the placeholders are endowed with a bit more information, e.g. if they carry a type information that is propagated to resulting placeholders by individual operations, then our abstract interpretation simultaneously performs type inference. This is, in essence, executing the program while abstracting out some concrete values and replacing them with the set of all values that could actually be there. If the sets are broad enough, then after some time we will have seen all potential value sets along each possible code paths, and our program analysis is complete. (Note that this is not exactly how [the PyPy analysis toolchain](#) does type inference: see below.)

An object space is thus an *interpretation domain*; the Flow Object Space is an *abstract interpretation domain*. We are thus interpreting the program while switching dynamically between several abstraction levels. This is possible because our design allows the *same* interpreter to work with a concrete or an abstract object space.



Following parts of the program at the abstract level allows us to deduce general information about the program, and for parts that cannot be analysed we switch to the concrete level. The restrictions placed on the program to statically analyse are that to be crafted in such a way that this process eventually terminates; from this point of view, more abstract is better (it covers whole sets of objects in a single pass). Thus the compromises that the author of the program to analyse faces are less strong but more subtle than a rule forbidding most dynamic features. The rule is, roughly speaking, to use dynamic features sparingly enough.

4.3 The PyPy analysis toolchain

We developed above a theoretical point of view that differs significantly from what we have implemented, for many reasons. The devil is in the details. Our toolchain is organised in three main passes, each described in its own chapter in the sequel:

- the [Flow Object Space](#) chapter describes how we turn Python bytecode objects into control flow graphs by performing its abstract interpretation;
- the [Annotator](#) chapter describes our type inference model and process, by doing abstract interpretation of the control flow graphs;
- the [Code Generation](#) chapter gives an overview about how we turn annotated flow graphs into low-level code.

The third pass is further divided into turning the control flow graphs into low-level ones, generating (e.g.) C source code for a C compiler, and invoking the C compiler to actually produce the executable.

4.4 Motivating our architecture

Before we start, we need a word of motivation to explain the reasons behind the rather complicated architecture that we describe in the sequel.

First of all, the overall picture of PyPy as described in ([ARCH](#)) is as follows: PyPy is an interpreter for the complete Python language, but it is itself written in the RPython subset. This is done in order to allow our analysis toolchain to apply to PyPy itself. Indeed, the primary goal is to allow us to implement the full Python language only once, as an interpreter, and derive interesting tools from it; doing so requires this interpreter to be amenable to analysis, hence the existence of RPython. The RPython language and our whole toolchain, despite their potential attraction, are so far meant as an internal detail of the PyPy project. The programs that we are deriving or plan to derive from PyPy include versions that run on very diverse platforms (from C to Java/.NET to Smalltalk), and also versions with modified execution models (from microthreads/coroutines to just-in-time compilers). This is why we have split the process in numerous interrelated phases, each at its own abstraction level. By enabling changes to the appropriate level, this opens the door to a wide range of retargetings of various kinds.

Focusing on the analysis toolchain again, here is how the existence of each component is justified (see below for *how* each component reaches the claimed goals):

- the [Flow Object Space](#) is a short but generic plug-in component for the Python interpreter of PyPy: it builds control flow graphs of functions by recording the operations issued by the *unmodified* interpreter. This means that it is independent of most language details. Changes in syntax or in bytecode format or opcode semantics only need to



be implemented once, in the standard Python interpreter. In effect, the Flow Object Space enables an interpreter for *any* language to work as a front-end for the rest of the toolchain.

- the [Annotator](#) performs type inference. This part is best implemented separately from other parts because it is based on a fixed point search algorithm. It is mostly this part that defines and restricts what RPython exactly is. After annotation, the control flow graphs still contain all the original relatively-high-level RPython operations; the inferred information is only used in the next step.
- the RType (TR) is the first component involved in [Code Generation](#). It does not emit any source code itself: it only replaces all RPython-level operations with lower-level operations in all control flow graphs. Each replacement is based on the type information collected by the annotator. In some sense the RType is the central bridge between the analysed program, written in RPython, and the target language and platform, which has different and usually lower-level operations, requirements and libraries. This RType is written in a modular way that allows it to be retargeted to various environments: for example, to target C-like languages we produce graphs that contain C-like operations, e.g. pointer manipulations; on the other hand, to target OO languages we need to produce graphs with operations like method calls.
- in the end, a translation back-end is responsible for generating actual source code from the flow graphs it receives. Given that the flow graphs are already at the correct level, the only remaining problems are at the level of difficulties with or limitations of the target language. This part depends strongly on the details of the target language, so little code can be shared between the different back-ends (even between back-ends taking as input the same low-level flow graphs, e.g. the C and the [LLVM](#) back-ends). The back-end is also responsible for integrating with some of the most platform-dependent aspects like memory management and exception model, as well as for generating alternate styles of code for different execution models like coroutines.



5 Flow Object Space

In our bytecode-interpreter design evaluation responsibilities are split between the Object Space, frames and the so-called execution context. The latter two object kinds are properly part of the interpretation engine, while the object space implements all operations on values which are treated as black boxes by the engine.

The Object Space plays the role of a factory for execution contexts. The base implementation of execution contexts is supplied by the engine, and exposes hooks triggered when frames are entered and left and before each bytecode, allowing a trace of the execution to be gathered. Frames have run/resume methods which embed the interpretation loop and invoke the hooks at the appropriate times.

One of our Object Spaces is the Flow Object Space, or “Flow Space” for short. Its role is to construct the control flow graph for a single function using abstract interpretation. The domain on which the Flow Space operates comprises variables and constant objects. They are stored as such in the frame objects without problems because by design the interpreter engine treats them as black boxes.

5.1 Construction of flow graphs

Concretely, the Flow Space plugs itself in the interpreter as an object space and supplies a derived execution context implementation. It also wraps a fix-point loop around invocations of the frame resume method. In our current design, this fix-point searching is implemented by interrupting the normal interpreter loop in the frame after every bytecode, and comparing the state with previously-seen states. These states describe the execution state for the frame at a given point. They are synthesised out of the frame by the Flow Space; they contain position-dependent data (current bytecode index, current exception handlers stack) as well as a flattened list of all variables and constants currently handled by the frame.

The Flow Space constructs the flow graph, operation after operation, as a side effect of seeing these operations performed by the interpretation of the bytecode. During construction, the operations are grouped in basic blocks that all have an associated frame state. The Flow Space starts from an empty block with a frame state corresponding to a freshly initialised frame, with a new variable for each input argument of the analysed function. It proceeds by recording the operations into this fresh block, as follows: when an operation is delegated to the Flow Space by the frame interpretation loop, either a constant result is produced -- in the case of constant arguments to an operation with no side-effects -- or a fresh new variable is produced. In the latter case, the operation (together with its input variables and constant arguments, and its output variable) is recorded in the current block and the new variable is returned as result to the frame interpretation loop.

When a new bytecode is about to be executed, as signalled by the bytecode hook, the Flow Space considers the frame state corresponding to the current frame contents. This state is compared with the existing states attached to the blocks produced so far. If the state was not seen before, the Flow Space creates a new block in the graph. If the same state was already seen before, then a backlink to the previous block is inserted, and the abstract interpretation stops here. If only a “similar enough” state was seen so far, then the current and the previous states are merged to produce a more general state.

In more details, “similar enough” is defined as having the same position-dependent part, the so-called “non-mergeable frame state”, which mostly means that only frame states corresponding to the same bytecode position can ever be merged. This process thus produces basic blocks that are generally in one-to-one correspondence with the bytecode positions seen



so far¹. The exception to this rule is in the rare cases where frames from the same bytecode position have a different non-mergeable state, which typically occurs during the “finally” part of a “try: finally:” construct, where the details of the exception handler stack differs according to whether the “finally” part was entered normally or as a result of an exception.

If two states have the same non-mergeable part, they can be merged using a “union” operation: only two equal constants unify to a constant of the same value; all other combinations (variable-variable or variable-constant) unify to a fresh new variable.

In summary, if some previously associated frame state for the next bytecode can be unified with the current state, then a backlink to the corresponding existing block is inserted; additionally, if the unified state is strictly more general than the existing one, then the existing block is cleared, and we proceed with the generalised state, reusing the block. (Reusing the block avoids the proliferation of over-specific blocks. For example, without this, all loops would typically have their first pass unrolled with the first value of the counter as a constant; instead, the second pass through the loop that the Flow Space does with the counter generalised as a variable will reuse the same entry point block, and any further blocks from the first pass are simply garbage-collected.)

5.2 Branching

Branching on conditions by the engine usually involves querying the truth value of a object through the `is_true` space operation. When this object is a variable, the result is not statically known; this needs special treatment to be able to capture both possible flow paths. In theory, this would require continuation support at the language level so that we can pretend that `is_true` returns twice into the engine, once for each possible answer, so that the Flow Space can record both outcomes. Without proper continuations in Python, we have implemented a more explicit scheme (described below) where the execution context and the object space collaborate to emulate this effect. (The approach is related to the one used in [Psyco](#), where regular continuations would be entirely impractical due to the need of huge amounts of them -- as described in the [ACM SIGPLAN 2004 paper](#).)

At any point in time, multiple pending blocks can be scheduled for abstract interpretation by the Flow Space, which proceeds by picking one of them and reconstructing a frame from the frame state associated with the block. This frame reconstruction is actually delegated to the block, which also returns a so-called “recorder” through which the Flow Space will append new space operations to the block. The recorder is also responsible for handling the `is_true` operation.

A normal recorder simply appends the space operations to the block from which it comes from. However, when it sees an `is_true` operation, it creates and schedules two special blocks (one for the outcome `True` and one for the outcome `False`) which do not have an associated frame state. The previous block is linked to the two new blocks with conditional exits. At this point, abstract interpretation stops (i.e. an exception is raised to interrupt the engine).

The special blocks have no frame state and thus cannot be used to setup a fresh frame. The reason is that while normal blocks correspond to the state of the engine between the execution of two bytecodes, the special blocks correspond to a call to `is_true` issued by the engine. The details of the engine state (internal call stack and local variables) are not available at this point.

¹this creates many small basic blocks; for convenience, a post-processing phase merges them into larger blocks when possible.



However, it is still possible to put the engine back into the state where it was calling `is_true`. This is what occurs later on, when one of the special block is scheduled for further execution: the block considers its previous block, and possibly its previous block's previous block, and so on up to the first normal block. As we can see, these blocks form a binary tree of special blocks with a normal block at the root. A special block thus corresponds to a branch in the tree, whose path is described by a list of outcomes -- a list of boolean values. We can thus restore the state of any block by starting from the root and asking the engine to replay the execution from there; intermediate `is_true` calls issued by the engine are answered according to the list of outcomes until we reach the desired state.

This is implemented by having a special blocks (called `EggBlocks` internally, whereas normal blocks are `SpamBlocks`²) return a chain of recorders: one so-called "replaying" recorder for each of the parent blocks in the tree, followed by a normal recorder for the block itself. When the engine replays the execution from the root of the tree, the intermediate recorders check (for consistency) that the same operations as the ones already recorded are issued again, ending in a call to `is_true`; at this point, the replaying recorder gives the answer corresponding to the branch to follow, and switches to the next recorder in the chain.

This mechanism ensures that all flow paths are considered, including different flow paths inside the engine and not only flow paths that are explicit in the bytecode. For example, an `UNPACK_SEQUENCE` bytecode in the engine iterates over a sequence object and checks that it produces exactly the expected number of values; so the single bytecode `UNPACK_SEQUENCE n` generates a tree with $n+1$ branches corresponding to the $n+1$ times the engine asks the iterator if it has more elements to produce. A simpler example is a conditional jump, which will generate a pair of special blocks for the `is_true`, each of which consisting only in a jump to the normal block corresponding to the next bytecode -- either the one following the conditional jump, or the target of the jump, depending on whether the replayer answered `False` or `True` to the `is_true`.

Note a limitation of this mechanism: the engine cannot use an unbounded loop to implement a single bytecode. All *loops* must still be explicitly present in the bytecodes. The reason is that the Flow Space can only insert backlinks from the end of a bytecode to the beginning of another one.

5.3 Dynamic merging

For simplicity, we have so far omitted a point in the description of how frame states are associated to blocks. In our implementation, there is not necessarily a block corresponding to each bytecode position (or more precisely each non-mergeable state): we avoid creating blocks at all if they would stay empty. This is done by tentatively running the engine on a given frame state and seeing if it creates at least one operation; if it does not, then we simply continue with the new frame state without having created a block for the previous frame state. The previous frame state is discarded without having even tried to compare it with already-seen state to see if it merges.

The effect of this is that merging only occurs at the beginning of a bytecode that actually produces an operation. This allows some amount of constant-folding: for example, the two functions below produce the same flow graph:

```
def f(n):                def g(n):
    if n < 0:              if n < 0:
        n = 0              return 1
```

²"spam, spam, spam, egg and spam" -- references to Monty Python are common in Python :-)



```
return n+1                else:
                           return n+1
```

because the two branches of the condition are not merged where the `if` statement syntactically ends: the `True` branch propagates a constant zero in the local variable `n`, and the following addition is constant-folded and does not generate a residual operation.

Note that this feature means that the Flow Space is not guaranteed to terminate. The analysed function can contain arbitrary computations on constant values (with loops) that will be entirely constant-folded by the Flow Space. A function with an obvious infinite loop will send the Flow Space following the loop ad infinitum. This means that it is difficult to give precise conditions for when the Flow Space terminates and which complexity it has. Informally, “reasonable” functions should not create problems: it is uncommon for a function to perform non-trivial constant computations at run-time; and the complexity of the Flow Space can more or less be bound by the run-time complexity of the constant parts of the function itself, if we ignore pathological cases -- e.g. a function containing some infinite loops that cannot be reached at run-time for reasons unknown to the Flow Space.

However, barring extreme examples we can disregard pathological cases because of testing -- we make sure that the code that we send to the Flow Space is first well-tested. This philosophy will be seen again.

5.4 Geninterp

Introducing [Dynamic merging](#) can be seen as a practical move: it does not, in practice, prevent even large functions from being analysed reasonably quickly, and it is useful to simplify the flow graphs of some functions. This is especially true for functions that are themselves automatically generated.

In the PyPy interpreter, for convenience, some of the more complex core functionalities are not directly implemented in the interpreter. They are written as “application-level” Python code, i.e. helper code that needs to be interpreted just like the rest of the user program. This has, of course, a performance hit due to the interpretation overhead. To minimise this overhead, we automatically turn some of this application-level code into interpreter-level code, as follows. Consider the following trivial example function at application-level:

```
def f_app(n):
    return n+1
```

Interpreting it, the engine just issues an `add` operation to the object space, which means that it is mostly equivalent to the following interpreter-level function:

```
def f_interp(space, wrapped_n):
    return space.add(wrapped_n, wrapped_1)
```

The translation from `f_app` to `f_interp` can be done automatically by using the Flow Space as well: we produce the flow graph of `f_app` using the techniques described above, and then we turn the resulting flow graph into `f_interp` by generating for each operation a call to the corresponding method of `space`.

This process loses the original syntactic structure of `f_app`, though; the flow graph is merely a collection of blocks that jump to each other. It is not always easy to reconstruct the structure from the graph (or even possible at all, in some cases where the flow graph does not exactly follow the bytecode). So, as is common for code generators, we use a workaround to the absence of explicit `gotos`:



```
def f_interp(...):
    next_block = 0
    while True:

        if next_block == 0:
            ...
            next_block = 1

        if next_block == 1:
            ...
```

This produces Python code that is particularly sub-efficient when it is interpreted; however, if it is further re-analysed by the Flow Space, dynamic merging will ensure that `next_block` will always be constant-folded away, instead of having the various possible values of `next_block` be merged at the beginning of the loop.

For more information see [The Interlevel Back-End](#) in the reference documentation.



6 Annotator

The annotator is the type inference part of our toolchain. The annotator infers *types* in the following sense: given a program considered as a family of control flow graphs, it assigns to each variable of each graph a so-called *annotation*, which describes the possible run-time objects that this variable can contain. Note that in the literature such an annotation is usually called a type, but we prefer to avoid this terminology to avoid confusion with the Python notion of the concrete type of an object. An annotation is a set of possible values, and such a set is not always the set of all objects of a specific Python type.

We will first describe a simplified, static model of how the annotator works, and then hint at some differences between the model and the reality.

6.1 Static model

The annotator can be considered as taking as input a finite family of functions calling each other, and working on the control flow graphs of each of these functions as built by the [Flow Object Space](#). Additionally, for a particular “entry point” function, each input argument is given a user-specified annotation.

The goal of the annotator is to find the most precise annotation that can be given to each variable of all control flow graphs while respecting the constraints imposed by the operations in which these variables are involved.

More precisely, it is usually possible to deduce information about the result variable of an operation given information about its arguments. For example, we can say that the addition of two integers must be an integer. Most programming languages have this property. However, Python -- like many languages not specifically designed with type inference in mind -- does not possess a type system that allows much useful information to be derived about variables based on how they are *used*; only on how they were *produced*. For example, a number of very different built-in types can be involved in an addition; the meaning of the addition and the type of the result depends on the type of the input arguments. Merely knowing that a variable will be used in an addition does not give much information per se. For this reason, our annotator works by flowing annotations forward, operation after operation, i.e. by performing abstract interpretation of the flow graphs. In a sense, it is a more naive approach than the one taken by type systems specifically designed to enable more advanced inference algorithms. For example, Hindley-Milner (([Milner](#)), ([DaMi](#))) type inference works in an inside-out direction, by starting from individual operations and propagating type constraints outwards.

Naturally, simply propagating annotations forward requires the use of a fixed point algorithm in the presence of loops in the flow graphs or in the inter-procedural call graph. Indeed, we flow annotations forward from the beginning of the entry point function into each block, operation after operation, and follow all calls recursively. During this process, each variable along the way gets an annotation. In various cases, e.g. when we close a loop, the previously assigned annotations can be found to be too restrictive. In this case, we generalise them to allow for a larger set of possible run-time values, and schedule the block where they appear for reflowing. The more general annotations can generalise the annotations of the results of the variables in the block, which in turn can generalise the annotations that flow into the following blocks, and so on. This process continues until a fixed point is reached.

We can consider that all variables are initially assigned the “bottom” annotation corresponding to the empty set of possible run-time values. Annotations can only ever be generalised, and the model is simple enough to show that there is no infinite chain of generalisation, so that this process necessarily terminates.



6.2 Flow graph model

For the purpose of the sequel, an informal description of the data model used to represent flow graphs will suffice (a [precise description](#) can be found in the reference documentation).

The flow graphs are in Static Single Information (SSI) form, an extension of Static Single Assignment (SSA): each variable is only used in exactly one basic block. All variables that are not dead at the end of a basic block are explicitly carried over to the next block and renamed. Instead of the traditional phi functions of SSA we use a minor variant, parameter-passing style: each block declares a number of *input variables* playing the role of input arguments to the block; each link going out of a block carries a matching number of variables and constants from the previous block into the target block's input arguments.

We use the following notation for an *operation* recorded in a block of the flow graph of a function:

$$z = \text{opname}(x_1, \dots, x_n) | z'$$

where x_1, \dots, x_n are the arguments of the operation (either variables defined earlier in the block, or constants), z is the variable into which the result is stored (each operation introduces a new fresh variable as its result), and z' is a fresh extra variable called the "auxiliary variable" which we will use in particular cases (which we omit from the notation when it is irrelevant).

Let us assume that we are given a user program, which for the purpose of the model we assume to be fully known in advance. Let us define the set V of all variables as follows:

- V contains all the variables that appear in operations, in any flow graph of any function of the program, as described above;
- in addition, for each class C of the user program and each possible attribute name $attr$, we add to V a variable called $v_{C.attr}$.

For a function f of the user program, we call $arg_{f_1}, \dots, arg_{f_n}$ the variables bound to the input arguments of f (which are actually the input variables of the first block in the flow graph of f) and $returnvar_f$ the variable bound to the return value of f (which is the single input variable of a special empty "return" block ending the flow graph).

Note that the complete knowledge of the operations and classes that appear in the user program allow us to bound the size of V . Indeed, the set of possible attribute names can be defined as all names that appear in a `getattr` or `setattr` operation; no other name will play a role during annotation.

6.3 Annotation model

As in the [formal definition](#) of Abstract Interpretation, the model for our annotation forms a [lattice](#), although we only use its structure of [join-semilattice](#).

The set A of annotations is defined as the following formal terms:

- Bot, Top -- the minimum and maximum elements (corresponding to "impossible value" and "most general value");
- $Int, NonNegInt, Bool$ -- integers, known-non-negative integers, booleans;



- *Str, Char* -- strings, characters (which are strings of length 1);
- *Inst(class)* -- instance of *class* or a subclass thereof (there is one such term per *class*);
- *List(v)* -- list; *v* is a variable summarising the items of the list (there is one such term per variable);
- *Pbc(set)* -- where the *set* is a subset of the (finite) set of all [Pre-built Constants](#), defined below. This set includes all the callables of the user program: functions, classes, and methods.
- *None* -- stands for the singleton `None` object of Python.

More details about the annotations will be introduced in due time. In addition, some of the annotations have a corresponding “nullable” twin, which stands for “either the object described or `None`”. We use it to propagate knowledge about which variable, after translation to C, could ever contain a NULL pointer. (More precisely, there are a *NullableStr*, nullable instances, and nullable Pbc’s, and all lists are implicitly assumed to be nullable).

Each annotation corresponds to a family of run-time Python object; the ordering of the lattice is essentially the subset order. Formally, it is the partial order generated by:

- $Bot \leq a \leq Top$ -- for any annotation *a*;
- $Bool \leq NonNegInt \leq Int$;
- $Char \leq Str$;
- $Inst(subclass) \leq Inst(class)$ -- for any class and subclass;
- $Pbc(subset) \leq Pbc(set)$;
- $a \leq b$ -- for any annotation *a* with a nullable twin *b*;
- $None \leq b$ -- for any nullable annotation *b*.

It is left as an exercise to show that this partial order makes *A* a lattice.

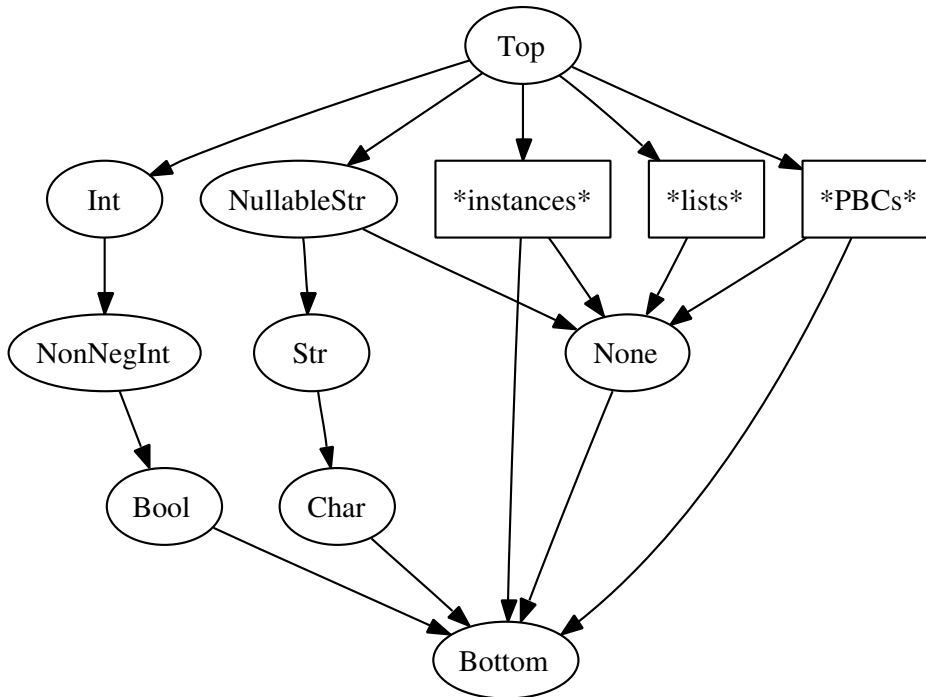


Figure 1: the lattice of annotations.

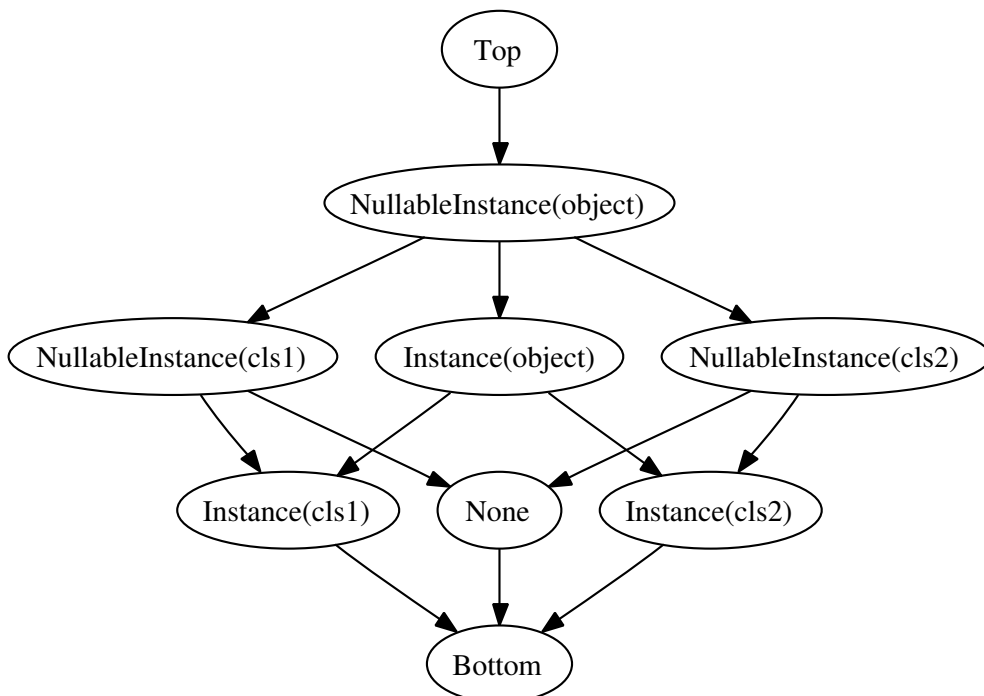


Figure 2: The part about instances and nullable instances, assuming a simple class hierarchy with only two direct subclasses of `object`.

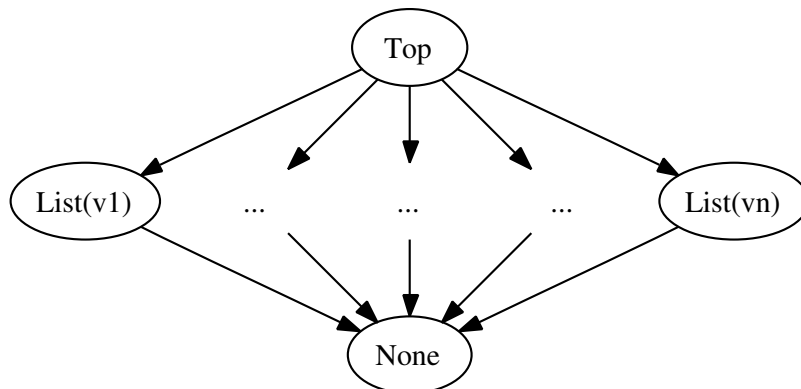


Figure 3: All list terms for all variables are unordered.

The Pbc's form a classical finite set-of-subsets lattice. In practice, we consider `None` as a de-generated pre-built constant, so the `None` annotation is actually $Pbc(\{None\})$.

We should mention (but ignore for the sequel) that all annotations also have a variant where they stand for a single known object; this information is used in constant propagation. In addition, we have left out a number of other annotations that are irrelevant for the basic description of the annotator and straightforward to handle: *Dictionary*, *Tuple*, *Float*, *UnicodePoint*, *Iterator*, etc. The complete list is defined and documented in [pypy/annotation/model.py](#) and described in the [annotator reference documentation](#) (TR).

6.4 Rules

In the sequel, we will use the following notations:

- A is the lattice defined above;
- V is the set of variable;
- $E, E', E'' \dots$ are equivalence relations on V ; where unambiguous, we write $v \sim v'$ to mean that v and v' are identified by E .
- $b, b', b'' \dots$ are maps from V to A .

We call *state* a pair (b, E) . We say that a state (b', E') is more general than a state (b, E) if for all variables v we have $b'(v) \geq b(v)$ and E' includes at least all relations of E . There is:

- a most general state, with $b_{max}(v) = Top$ for all v and E_{max} identifying all variables with each other;
- a least general state, with $b_{min}(v) = Bottom$ for all v and E_{min} containing no relation (apart from the mandatory $v \sim v$).

Based on the "union" operator \vee of the lattice A , we can compute the union of two states as follows: $(b_1, E_1) \vee (b_2, E_2) = (b_3, E_3)$ where $b_3(x) = b_1(x) \vee b_2(x)$ for all x , and $E_3 = E_1 \cup E_2$.

The goal of the annotator is to find the least general (i.e. most precise) state that is sound (i.e. correct for the user program). The algorithm used is a fixed point search: we start from the least



general state and consider the conditions repeatedly; if a condition is not met, we generalise the state incrementally to accommodate for it. This process continues until all conditions are satisfied.

The conditions are presented as a set of rules. A rule is a functional operation that, when applied, maps a state (b, E) to a possibly more general state (b', E') that satisfies the condition represented by the rule. *Soundness* is formally defined as a state in which all the conditions are already satisfied, i.e. none of the rules would produce a strictly more general state.

Basically, each operation in the flow graphs of the user program generates one such rule. The rules are conditional on the annotations bound to the operation's input argument variables, in a way that mimics the ad-hoc polymorphic nature of most Python operations. We will not give all rules in the sequel, but focus on representative examples. An `add` operation generates the following rules (where x, y and z are replaced by the variables that really appear in each particular `add` operation in the flow graphs of the user program):

$$\frac{z = \text{add}(x, y), b(x) = \text{Int}, \text{Bool} \leq b(y) \leq \text{Int}}{b' = b \text{ with } (z \rightarrow \text{Int})}$$

$$\frac{z = \text{add}(x, y), \text{Bool} \leq b(x) \leq \text{Int}, b(y) = \text{Int}}{b' = b \text{ with } (z \rightarrow \text{Int})}$$

$$\frac{z = \text{add}(x, y), \text{Bool} \leq b(x) \leq \text{NonNegInt}, \text{Bool} \leq b(y) \leq \text{NonNegInt}}{b' = b \text{ with } (z \rightarrow \text{NonNegInt})}$$

$$\frac{z = \text{add}(x, y), \text{Char} \leq b(x) \leq \text{NullableStr}, \text{Char} \leq b(y) \leq \text{NullableStr}}{b' = b \text{ with } (z \rightarrow \text{Str})^1}$$

The rules are read as follows: for the operation $z = \text{add}(x, y)$, we consider the bindings of the variables x and y in the current state (b, E) ; if the bindings satisfy the given conditions, then the rule is applicable. Applying the rule means producing a new state (b', E') derived from the current state -- here by changing the binding of the result variable z .

Note that for conciseness we omitted the $E' = E$ (none of these rules modify E).

¹ Also note that we do not generally try to prove the correctness and safety of the user program, preferring to rely on test coverage for that. This is apparent in the last rule above, which considers concatenation of two potentially "nullable" strings, i.e. strings that the annotator could not prove to be non-None. Instead of reporting an error, we take it as a hint that the two strings will not actually be None at run-time and proceed.

In the sequel, a lot of rules will be based on the following `merge` operator. Given an annotation a and a variable x , "merge $a \Rightarrow x$ " modifies the state as follows:

```
merge a ⇒ x :
  if a = List(v) and b(x) = List(w) :
    b' = b
    E' = E ∪ {v ~ w}
  else :
    b' = b with (x → a ∨ b(x))
    E' = E
```



where \vee is the union in the lattice A .

The above operator is first of all used to propagate bindings of variables across links between basic block in the control flow graphs. For every link mapping a variable x in the source block to a variable y in the target block, we generate the following rule (ϕ is not a normal operation in our [Flow graph model](#); we abuse the notation):

$$\frac{y = \phi(x)}{\text{merge } b(x) \Rightarrow y}$$

The purpose of the equivalence relation E is to force two identified variables to keep the same binding. The rationale for this is explained in the [Mutable objects](#) section below. It is enforced by the following family of rules (one for each pair (x, y)):

$$\frac{(x \sim y) \in E}{\begin{array}{l} \text{merge } b(x) \Rightarrow y \\ \text{merge } b(y) \Rightarrow x \end{array}}$$

Note that a priori, all rules should be tried repeatedly until none of them generalises the state any more, at which point we have reached a fixed point. However, the rules are well suited to a simple meta-rule that tracks a small set of rules that can possibly apply. Only these “scheduled” rules are tried. The meta-rule is as follows:

- when an identification $x \sim y$ is added to E , then the rule $(x \sim y) \in E$ is scheduled to be considered;
- when a binding $b(x)$ is modified, then all rules about operations that have x as an input argument are (re-)scheduled. This includes the rules $(x \sim y) \in E$ for each y that E identifies to x . This also includes the cases where x is the auxiliary variable of an operation (see [Flow graph model](#)).

These rules and meta-rule favour a forward propagation: the rule corresponding to an operation in a flow graph typically modifies the binding of the operation’s result variable which is used in a following operation in the same block, thus scheduling the following operation’s rule for consideration. The actual process is very similar to – and actually implemented as – abstract interpretation on the flow graphs, considering each operation in turn in the order they appear in the block. Then for simplicity we reschedule whole blocks instead of single operations.

6.5 Mutable objects

Tracking mutable objects is the difficult part of our approach. RPython contains two types of mutable objects that need special care: lists (Python’s vectors) and instances of user-defined classes. The current section focuses on lists. [Classes and instances](#) will be described in their own section. (The complete definition of RPython also allows for dictionaries, which are similar to lists.)

For lists, we try to derive a homogeneous annotation for all items of the list. In other words, RPython does not support heterogeneous lists. The approach is to consider each list-creation



point as building a new type of list and following the way the list is used to derive the union type of its items.

Note that we are not trying to be more precise than finding a single item type for each list. Flow-sensitive techniques could be potentially more precise by tracking different possible states for the same list at different points in the program and in time. But in any case, a pure forward propagation of annotations is not sufficient because of aliasing: it is possible to take a reference to a list at any point, and store it somewhere for future access. If a new item is inserted into a list in a way that generalises the list's type, all potential aliases must reflect this change -- this means all references that were "forked" from the one through which the list is modified.

To solve this, each list annotation -- $List(v)$ -- contains an embedded variable, called the "hidden variable" of the list. It does not appear directly in the flow graphs of the user program, but abstractedly stands for "any item of the list". The annotation $List(v)$ is propagated forward as with other kinds of annotations, so that all aliases of the list end up being annotated as $List(v)$ with the same variable v . The binding of v itself, i.e. $b(v)$, is updated to reflect generalisation of the list item's type; such an update is instantly visible to all aliases. Moreover, the update is described as a change of binding, which means that the meta-rules will ensure that any rule based on the binding of this variable will be reconsidered.

The hidden variable comes from the auxiliary variable syntactically attached to the operation that produces a list:

$$\frac{z = \text{new_list}()|z'}{b' = b \text{ with } (z \rightarrow List(z'))}$$

Inserting an item into a list is done by merging the new item's annotation into the list's hidden variable (y is the index in the list x and z is the new item):

$$\frac{\text{setitem}(x, y, z), b(x) = List(v)}{\text{merge } b(z) \Rightarrow v}$$

Reading an item out of a list requires care to ensure that the rule is rescheduled if the binding of the hidden variable is generalised. We do so by identifying the hidden variable with the current operation's auxiliary variable. The identification ensures that the hidden variable's binding will eventually propagate to the auxiliary variable, which -- according to the meta-rule -- will reschedule the operation's rule:

$$\frac{z = \text{getitem}(x, y)|z', b(x) = List(v)}{E' = E \cup \{z' \sim v\}} \\ b' = b \text{ with } (z \rightarrow b(z'))$$

We cannot directly set $z \rightarrow b(v)$ because that would be an "illegal" use of a binding, in the sense explained above: it would defeat the meta-rule for rescheduling the rule when $b(v)$ is modified. (In the source code, the same effect is actually achieved by recording on a case-by-case basis at which locations the binding $b(v)$ has been read; in the theory we use the equivalence relation E to make this notion explicit.)

If you consider the definition of [merge](#) again, you will notice that merging two different lists (for example, two lists that come from different creation points via different code paths) identifies



the two hidden variables. This effectively identifies the two lists, as if they had the same origin. It makes the two list annotations aliases for each other, allowing any storage location to contain lists coming from any of the two sources indifferently. This process gradually builds a partition of all lists in the program, where two lists are in the same part if they are combined in any way.

As an example of further list operations, here is the addition (which is the concatenation for lists):

$$\frac{z = \text{add}(x, y), b(x) = \text{List}(v), b(y) = \text{List}(w)}{E' = E \cup \{v \sim w\}} \\ b' = b \text{ with } (z \rightarrow \text{List}(v))$$

As with `merge`, it identifies the two lists.

6.6 Pre-built constants

The *Pbc* annotations play a special role in our approach. They group in a single family all the constant user-defined objects that exist before the annotation phase. This includes the functions and classes defined in the user program, but also some other objects that have been built while the user program was initialising itself.

The presence of the latter kind of object -- which come with a number of new problems to solve -- is a distinguishing property of the idea of analysing a live program instead of static source code. All the user objects that exist before the annotation phase are divided in two further families: the "pre-built instances" and the "frozen pre-built constants".

1. Normally, all instances of user-defined classes have the same behaviour, independently of whether they exist before annotation or are built dynamically by the program after annotation and compilation. Both correspond to the *Inst(C)* annotation. Instances that are pre-built will simply be compiled into the resulting executable as pre-built data structures.
2. However, as an exception to 1., the user program can give a hint that forces the annotator to consider such an object as a "frozen pre-built constant" instead. The object is then considered as an *immutable* container of attributes. It loses its object-oriented aspects and its class becomes irrelevant. It is not possible to further instantiate its class at run-time.

In summary, the pre-built constants are:

- all functions *f* of the user program (including the ones appearing as methods);
- all classes *C* of the user program;
- all frozen pre-built constants.

For convenience, we add the following objects to the above set:

- for each function *f* and class *C*, a "potential bound method" object written *C.f*, used below to handle method calls;
- the singleton `None` object (a special case of frozen pre-built constant).



The annotation $Pbc(set)$ stands for an object that belongs to the specified set of pre-built constant objects, which is a subset of all the pre-built constant objects.

In practice, the set of all pre-built constants is not fixed in advance, but grows while annotation discovers new functions and classes and frozen pre-built constants; in this way we can be sure that only the objects that are still alive will be included in the set, leaving out the ones that were only relevant during the initialisation phase of the program.

6.7 Classes and instances

Remember that Python has no notion of classes declaring attributes and methods. Classes are merely hierarchical namespaces: an expression like `obj.attr` (a `getattr` operation) means that the `attr` attribute is looked up in the class that `obj` is an instance of at run-time, and all parent classes. Expressions like `obj.method()` that look like method calls are actually grouped as `(obj.method)()`: they correspond to two operations, a `getattr` followed by a `call`. The intermediate object returned by `obj.method` is a bound method.

As the goal of annotator is to derive some static type information about the user program, it must reconstruct a static structure for each class in the hierarchy. It does so by observing the usage patterns of the classes and their instances, by propagating annotations of the form $Inst(cls)$ -- which stands for "an instance of the class cls or any subclass". Instance fields are attached to a class whenever we see that the field is being written to an instance of this class. If the user program manipulates instances polymorphically, the variables holding the instances will be annotated $Inst(cls)$ with some abstract base class cls ; accessing attributes on such generalised instances lifts the inferred attribute declarations up to cls . The same technique works for inferring the location of both fields and methods.

We assume that the classes in the user program are organised in a single inheritance tree rooted at the `object` base class. (Python supports multiple inheritance, but the annotator is limited to single inheritance plus simple mix-ins.) We also assume that polymorphic instance usage is "bounded" in the sense that all instances that can reach a specific program point are instances of a user-defined common base class, i.e. not `object`.

Remember from the [definition of V](#) that we have a variable $v_{C.attr}$ for each class C and each possible attribute name $attr$. The annotation state (b, E) gives the following meaning to these variables:

- the annotation $b(v_{C.attr})$ is the inferred type of the values that can result from reading the attribute $attr$ out of an instance of C ;
- to account for the inheritance between classes, the equivalence relation E identifies some of these variables as follows: if an attribute $attr$ is found to belong to a base class C , then all variables $v_{D.attr}$ for all subclasses D of C are identified with $v_{C.attr}$. This ensures that the instances of all the subclasses are given the same generic attribute defined in C .

Formally:

$$\frac{z = \text{getattr}(x, attr) | z', b(x) = Inst(C)}{E' = E \cup \{v_{C.attr} \sim v_{D.attr} \text{ for all } D \text{ subclass of } C\} \cup \{z' \sim v_{C.attr}\} \\ b' = b \text{ with } (z \rightarrow \text{lookup_filter}(b(z'), C))}$$



$$\frac{\text{setattr}(x, \text{attr}, z), b(x) = \text{Inst}(C)}{E' = E \cup \{v_{C.\text{attr}} \sim v_{D.\text{attr}} \text{ for all } D \text{ subclass of } C\}}$$

check $b(z)$ for the absence of potential bound method objects
merge $b(z) \Rightarrow v_{C.\text{attr}}$

Note the similarity with the `getitem` and `setitem` of lists, in particular the usage of the auxiliary variable z' . Also note that we still allow real bound methods to be handled and passed around in the way that is quite unique to Python: if meth is the name of a method of x , then $y = x.\text{meth}$ is allowed, and the object y can be passed around and stored in data structures. However, in our case we do not allow such objects to be stored directly back into other instances (it is the purpose of the check in the rule for `setattr`). This would create a confusion between class-level and instance-level attributes in a subsequent `getattr`. It is a limitation of our annotator to not distinguish these two levels -- there is only one set of $v_{C.\text{attr}}$ variables for both.

The purpose of `lookup_filter` is to avoid losing precision in method calls. Indeed, if attr names a method of the class C then the binding $b(v_{C.\text{attr}})$ is initialised to $Pbc(m)$, where m is the following set:

- for each subclass D of C , if the class D introduces a method attr implemented as, say, the function f , then the "potential bound method" object $D.f$ belongs to m .

However, because of the possible identification between the variable $v_{C.\text{attr}}$ and the corresponding variable $v_{B.\text{attr}}$ of a superclass, the set m might end up containing potential bound methods of other unrelated subclasses of B , even when performing a `getattr` on what we know is an instance of C . The `lookup_filter` reverses this effect. Its definition reflects where a method lookup can actually find a method if it is performed on an instance of an unspecified subclass of C : either in one of these subclasses, or in C or the closest parent class that defines the method. If the method is also defined in further parents, these definitions are hidden. More precisely:

$$\text{lookup_filter}(Pbc(m), C) = Pbc(\text{newset})$$

$$\text{lookup_filter}(NonPbcAnnotation, C) = NonPbcAnnotation$$

where the *newset* is a subset of the set m with the following objects:

- all the objects of m that are not potential bound method objects; plus
- the potential bound method objects of m that are of the form $D.f$, where D is a strict subclass of C ; plus
- among the potential bound method objects (if any) whose class is C or a superclass of C , we select the ones whose class is the most derived class thus represented. In other words, if C inherits from B which inherits from A , then potential bound method objects $A.f$ would be included in the *newset* if and only if there is no $B.g$ or $C.h$ in m .

6.8 Calls

A call in the user program is represented by a `simple_call` operation whose first argument is the object to call. Here is the corresponding rule -- regrouping all cases because a single $Pbc(\text{set})$ annotation could theoretically mix several kinds of callables:



$$z = \text{simple_call}(x, y_1, \dots, y_n) | z', b(x) = Pbc(\text{set})$$

```

for each  $c \in \text{set}$  :
  if  $c$  is a function :
     $E' = E \cup \{z' \sim \text{returnvar}_c\}$ 
     $b' = b$  with  $(z \rightarrow b(z'))$ 
    merge  $b(y_1) \Rightarrow \text{arg}_{c_1}$ 
    ...
    merge  $b(y_n) \Rightarrow \text{arg}_{c_n}$ 
  if  $c$  is a class :
    let  $f = c.\_\_\text{init}\_\_$  # the constructor
    merge  $\text{Inst}(c) \Rightarrow z$ 
    merge  $\text{Inst}(c) \Rightarrow \text{arg}_{f_1}$ 
    merge  $b(y_1) \Rightarrow \text{arg}_{f_2}$ 
    ...
    merge  $b(y_n) \Rightarrow \text{arg}_{f_{n+1}}$ 
  if  $c$  is a method :
     $c$  is of the form  $\text{cls}.f$ 
     $E' = E \cup \{z' \sim \text{returnvar}_f\}$ 
     $b' = b$  with  $(z \rightarrow b(z'))$ 
    merge  $\text{Inst}(\text{cls}) \Rightarrow \text{arg}_{f_1}$ 
    merge  $b(y_1) \Rightarrow \text{arg}_{f_2}$ 
    ...
    merge  $b(y_n) \Rightarrow \text{arg}_{f_{n+1}}$ 

```

Calling a class returns an instance and flows the annotations into the constructor `__init__` of the class. Calling a method inserts the instance annotation as the first argument of the underlying function (the annotation is exactly `Inst(C)` for the class `C` in which the method is found).

6.9 Termination and soundness

As the annotation process is a fix-point search, we should prove for completeness that it is, in some sense yet to be defined, well-behaved. Given the approach we have taken, none of the following proofs is “deep”: the intended goal of the whole approach is to allow the development of an intuitive understanding of why annotation works. However, despite their straightforwardness the following proofs are quite technical; they are oriented towards the more mathematically-minded reader.

6.9.1 Generalisation

We first have to check that during the annotation process each rule can only turn a state (b, E) into a state (b', E') that is either identical or more general. Clearly, E' can only be generalised – applying a rule can only add new identifications, not remove existing ones. What is left to



check is that the annotation $b(v)$ of each variable, when modified, can only become more general (i.e. be increased, in the lattice order). We prove it in the following order:

1. the annotations of the input variables of blocks;
2. the annotations $b(v_{C.attr})$ of variables corresponding to attributes on classes;
3. the annotations of the auxiliary variable of operations;
4. the annotations of the input and result variables of operations.

Proof:

1. Input variables of blocks

The annotation of these variables are only modified by the `phi` and `simple_call` rules, which are based on `merge`. The `merge` operation trivially guarantees the property of generalisation because it is based on the union operator \vee of the lattice.

2. Auxiliary variables of operations

The binding of an auxiliary variable z' of an operation is never directly modified: it is only ever identified with other variables via E . So $b(z')$ is only updated by the rule $(z' \sim v) \in E$, which is based on the `merge` operator as well.

3. Variables corresponding to attributes of classes

The annotation of such variables can only be modified by the `setattr` rule (with a `merge`) or as a result of having been identified with other variables via E . We conclude as above.

4. Input and result variables of operations

By construction of the flow graphs, the input variables of any given operation must also appear before in the block, either as the result variable of a previous operation, or as an input variable of the block itself. So assume for now that the input variables of this operation can only get generalised; we claim that in this case the same holds for its result variable. If this holds, then we conclude by induction on the number of operations in the block: indeed, starting from point 1 above for the input variables of the block, it shows that each result variable -- so also all input arguments of the next operation -- can only get generalised.

To prove our claim, first note that none of these input and result variables is ever identified with any other variable via E : indeed, the rules described above only identify auxiliary variables or attribute-of-class variables with each other (the variables that appear in `List` annotations are always auxiliary variables too). It means that the only way the result variable z of an operation can be modified is directly by the rule or rules specific to that operation. This allows us to check the property of generalisation on a case-by-case basis.

Most cases are easy to check. Cases like $b' = b$ with $(z \rightarrow b(z'))$ where z' is an auxiliary variable are based on point 2 above. The only non-trivial case is in the rule for `getattr`:

$$b' = b \text{ with } (z \rightarrow \text{lookup_filter}(b(z'), C))$$

For this case, we need the following lemma:



Let $v_{C.attr}$ be an attribute-of-class variable. Let (b, E) be any state seen during the annotation process. Assume that $b(v_{C.attr}) = Pbc(set)$ where set is a set containing potential bound method objects. Call m the family of potential bound method objects appearing in set . Then m has the following shape: it is “regular” in the sense that it contains only potential bound method objects $D.f$ such that f is exactly the function found under the name $attr$ in some class D ; moreover, it is “downwards-closed” in the sense that if it contains a $D.f$, then it also contains all $E.g$ for all subclasses E of D that override the method (i.e. have a function g found under the same name $attr$).

Proof:

As we have seen in [Classes and instances](#) above, the initial binding of $v_{C.attr}$ is regular and downwards-closed by construction. Moreover, the `setattr` rule explicitly checks that it is never adding any potential bound method object to $b(v_{C.attr})$, so that the only way such objects can be added to $b(v_{C.attr})$ is via the identification of $v_{C.attr}$ with other $v_{B.attr}$ variables, for the same name $attr$ -- which implies that the set m will always be regular. Moreover, the union of two downwards-closed sets of potential bound method objects is still downwards-closed. This concludes the proof.

Let us consider the rule for `getattr` again:

$$b' = b \text{ with } (z \rightarrow \text{lookup_filter}(b(z'), C))$$

The only interesting case is when the binding $b(z')$ is a $Pbc(set)$ -- more precisely, we are interested in the part m of set that is the subset of all potential bound method objects; the function `lookup_filter` is the identity on the rest. Given that $b(z')$ comes from z' being identified with various $v_{C.attr}$ for a fixed name $attr$, the set m is regular and downwards-closed, as we can deduce from the lemma.

The class C in the rule for `getattr` comes from the annotation $Inst(C)$ of the first input variable of the operation. So what we need to prove is the following: if the binding of this input variable is generalised, and/or if the binding of z' is generalised, then the annotation computed by `lookup_filter` is also generalised (if modified at all):

$$\begin{array}{ll} \text{if} & b(x) = Inst(C) \leq b'(x) = Inst(C') \\ \text{and} & b(z') = Pbc(set) \leq b'(z') = Pbc(set') \end{array}$$

$$\text{then } \text{lookup_filter}(b(z'), C) \leq \text{lookup_filter}(b'(z'), C')$$

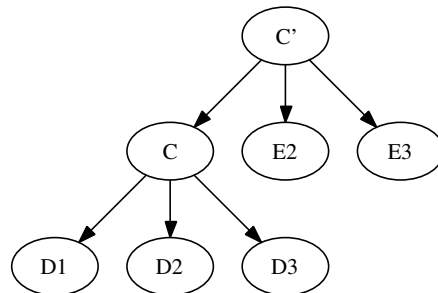
Proof:

Call respectively m and m' the subsets of potential bound method objects of set and set' , as above. Call l the subset of m as computed by `lookup_filter`, i.e.: l contains all objects $D.f$ of m for strict subclasses D of C , plus the single $B.g$ coming for the most derived non-strict superclass $B \geq C$ which appears in m . (Note that as m is regular, it cannot actually contain several potential bound method objects with the same class.) Similarly for l' computed from m' and C' .



By hypothesis, m is contained in m' , but we need to check that l is contained in l' . This is where we will use the fact that m is downwards-closed. Let $D.f$ be an element of l .

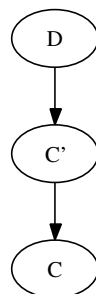
Case 1: if D is a strict subclass of C , then it is also a strict subclass of C' . In this case, $D.f$, which also belongs to m and thus to m' , also belongs to l' . Graphically:



(In this diagram, as far as they correspond to real methods, potential bound method objects $D1.f1$, $D2.f2$ and $D3.f3$ all belong to both l and l' . The family l' additionally contains $C.g$, $E2.h2$ and $E3.h3$. Both families l and l' also contain one extra item coming from the second part of their construction rule.)

Case 2: D is instead the most derived non-strict superclass of C which appears in m ; assume that D is still a strict subclass of C' . In this case, $D.f$ belongs to l' as previously. (For example, if there is a $C.g$ in m , then $D = C$ and as seen in the above diagram $C.g$ is indeed in l' .)

Case 3: D is still the most derived non-strict superclass of C which appears in m , but this time D is not a strict subclass of C' . The situation is as follows:



where extra intermediate classes could be present too. To be able to conclude that $D.f$ is in l' , we now have to prove that D is also the most derived non-strict superclass of C' that appears in m' . Ab absurdo, assume that this is not the case. Then there is a class B , strict superclass of C' but strict subclass of D , such that m' contains an element $B.g$. But m contains $D.f$ and it is downwards-closed, so it must also contain $B.g$. This contradicts the original hypothesis on D : this B would be another more derived superclass of C that appears in m . QED.

6.9.2 Termination

Each basic step (execution of one rule) can lead to the generalisation of the state. If it does, then other rules may be scheduled or re-scheduled for execution. The state can only be generalised a finite number of times because both the lattice A and the set of variables V



of which E is an equivalence relation are finite. If a rule does not lead to any generalisation, then it does not trigger re-scheduling of any other rule. This ensures that the process eventually terminates.

The extended lattice used in practice is a priori not finite. As we did not describe this lattice formally here, we have to skip the (easy) proof that it still contains no infinite ascending chain (an ascending chain is a sequence where each item is strictly larger than the previous one).

6.9.3 Soundness

We define an annotation state to be *sound* if none of the rules would lead to further generalisation. To define this notion more formally, we will use the following notation: let $Rules$ be the set of all rules (for the given user program). If r is a rule, then it can be considered as a (mathematical) function from the set of states to the set of states, so that “applying” the rule means computing $(b', E') = r((b, E))$. If the guards of the rule r are not satisfied then $r((b, E)) = (b, E)$. To formalise the meta-rule describing rescheduling of rules, we introduce a third component in the state: a subset S of the $Rules$ which stands for the currently scheduled rules. Finally, for any variable v we write $Rules_v$ for the set of rules that have v as an input or auxiliary variable. The rule titled $(x \sim y) \in E$ is called $r_{x \sim y}$ for short, and it belongs to $Rules_x$ and $Rules_y$.

The meta-rule can be formalised as follows: we start from the initial “meta-state” (S_0, b_0, E_0) , where $S_0 = Rules$ and (b_0, E_0) is the initial state; then we apply the following meta-rule that computes a new meta-state $(S_{i+1}, b_{i+1}, E_{i+1})$ from a meta-state (S_i, b_i, E_i) :

- pick a random r_i in the set of scheduled rules S_i
- compute $(b_{i+1}, E_{i+1}) = r_i((b_i, E_i))$
- let

$$S_{i+1} = \left(\begin{array}{l} S_i \quad - \quad \{r_i\} \\ \cup \quad Rules_v \text{ for all } v \text{ for which } b_{i+1}(v) \neq b_i(v) \\ \cup \quad \{r_{x \sim y} \text{ for all } (x \sim y) \in E_{i+1} - E_i\} \end{array} \right)$$

The meta-rule is applied repeatedly, giving rise to a sequence of meta-states $(S_0, b_0, E_0), (S_1, b_1, E_1), \dots, (S_n, b_n, E_n)$. The sequence ends when S_n is empty, at which point annotation is complete. The informal argument of the [Termination](#) paragraph shows that this sequence is necessarily of finite length. In the [Generalisation](#) paragraph we have also seen that each state (b_{i+1}, E_{i+1}) is equal to or more general than the previous state (b_i, E_i) – more generally, that applying any rule r to any state seen in the sequence leads to generalisation, or in formal terms $r((b_i, E_i)) \geq (b_i, E_i)$.

We define an annotation state (b, E) to be *sound* if for all rules r we have $r((b, E)) = (b, E)$. We say that (b, E) is *degenerated* if there is a variable v for which $b(v) = Top$. We will show the following propositions:

1. The final state (b_n, E_n) is sound.
2. If we assume that there exists (at all) a *non-degenerated sound* state (b, E) which is at least as general as (b_0, E_0) , then there is a unique minimal one among all such states, and this global minimum is exactly (b_n, E_n) .

Proof:



1. The final state (b_n, E_n) is sound.

The proof is based on the fact that the “effect” of any rule only depends on the annotation of its input and auxiliary variables. This “effect” is to merge some bindings and/or add some identifications; it can formalised by saying that $r((b, E)) = (b, E) \vee (b^f, E^f)$ for a certain (b^f, E^f) that contains only the new bindings and identifications.

More precisely, let r be a rule. Let V_r be the set of input and auxiliary variables of r , i.e.:

$$V_r = \{v \mid r \in Rules_v\}$$

Let (b, E) be a state. Then there exists a state (b^f, E^f) representing the “effect” of the rule on (b, E) as follows:

$$r((b, E)) = (b, E) \vee (b^f, E^f)$$

and the same (b^f, E^f) works for any (b', E') which is equal to (b, E) on V_r :

$$r((b', E')) = (b', E') \vee (b^f, E^f)$$

This is easily verified on a case-by-case basis for each kind of rule presented above. The details are left to the reader.

To show the proposition, we proceed by induction on i to show that each of the meta-states (S_i, b_i, E_i) has the following property: for each rule r which is not in S_i we have $r((b_i, E_i)) = (b_i, E_i)$. The result will follow from this claim because the final S_n is empty.

The claim is trivially true for $i = 0$. Let us assume that it holds for some $i < n$ and prove it for $i + 1$: let r be a rule not in S_{i+1} . By definition of S_{i+1} , the input/auxiliary variables of r have the same bindings at the steps i and $i + 1$, i.e. (b_i, E_i) is equal to (b_{i+1}, E_{i+1}) on V_r . Let (b^f, E^f) be the effect of r on (b_i, E_i) as above. We have:

$$\begin{aligned} r((b_i, E_i)) &= (b_i, E_i) \vee (b^f, E^f) \\ r((b_{i+1}, E_{i+1})) &= (b_{i+1}, E_{i+1}) \vee (b^f, E^f) \end{aligned}$$

Case 1: r is in S_i . As it is not in S_{i+1} it must be precisely r_i . In this case $r((b_i, E_i)) = (b_{i+1}, E_{i+1})$, so that:

$$\begin{aligned} r((b_{i+1}, E_{i+1})) &= (b_{i+1}, E_{i+1}) \vee (b^f, E^f) \\ &= r((b_i, E_i)) \vee (b^f, E^f) \\ &= (b_i, E_i) \vee (b^f, E^f) \vee (b^f, E^f) \\ &= (b_i, E_i) \vee (b^f, E^f) \\ &= r((b_i, E_i)) \\ &= (b_{i+1}, E_{i+1}). \end{aligned}$$

which concludes the induction step.

Case 2: r is not in S_i . By induction hypothesis $(b_i, E_i) = r((b_i, E_i))$.



$$\begin{aligned}
 (b_i, E_i) &= (b_i, E_i) \vee (b^f, E^f) \\
 (b_i, E_i) &\geq (b^f, E^f) \\
 (b_{i+1}, E_{i+1}) &\geq (b_i, E_i) \geq (b^f, E^f) \\
 (b_{i+1}, E_{i+1}) &= (b_{i+1}, E_{i+1}) \vee (b^f, E^f) \\
 (b_{i+1}, E_{i+1}) &= r((b_{i+1}, E_{i+1})).
 \end{aligned}$$

This concludes the proof.

2. (b_n, E_n) is the minimum of all sound non-degenerated states.

Let (b^s, E^s) be any sound non-degenerated state such that $(b_0, E_0) \leq (b^s, E^s)$. We will show by induction that for each $i \leq n$ we have $(b_i, E_i) \leq (b^s, E^s)$. The conclusion follows from the case $i = n$.

The claim holds for $i = 0$ by hypothesis. Let us assume that it is true for some $i < n$ and prove it for $i + 1$. We need to consider separate cases for each of the kind of rules that r_i can be. We only show a few representative examples and leave the complete proof to the reader. These examples show why it is a key point that (b^s, E^s) is not degenerated: most rules no longer apply if an annotation degenerates to *Top*, but continue to apply if it is generalised to anything below *Top*. The general idea is to turn each rule into a step of the complete proof, showing that if a sound state is at least as general as (b_i, E_i) then it must be at least as general as (b_{i+1}, E_{i+1}) .

Example 1. The rule r_i is:

$$\frac{z = \text{add}(x, y), b(x) = \text{Int}, \text{Bool} \leq b(y) \leq \text{Int}}{b' = b \text{ with } (z \rightarrow \text{Int})}$$

In this example, assuming that the guards are satisfied, b_{i+1} is b_i with $z \rightarrow \text{Int}$. We must show that $b^s(z) \geq \text{Int}$. We know from $(b_i, E_i) \leq (b^s, E^s)$ that $b^s(x) \geq \text{Int}$ and $b^s(y) \geq \text{Bool}$. As b^s is not degenerated, we have more precisely $b^s(x) = \text{Int}$, $\text{Bool} \leq b^s(y) \leq \text{Int}$. Moreover, by definition $r((b^s, E^s)) = (b^s, E^s)$. We conclude that $b^s(z) = \text{Int}$.

Example 2. The rule r_i is:

$$\frac{y = \text{phi}(x)}{\text{merge } b(x) \Rightarrow y}$$

We must show that $b^s(y) \geq b_{i+1}(y)$. We need to subdivide this example in two cases: either $b(x)$ and $b(y)$ are both *List* annotations or not.

If they are not, we know that $b_{i+1}(y) = b_i(y) \vee b_i(x)$ and $b^s(y) \geq b_i(y)$, so that we must show that $b^s(y) \geq b_i(x)$. We know that $r((b^s, E^s)) = (b^s, E^s)$ so that $b^s(y) = b^s(x) \vee b^s(y)$, i.e. $b^s(y) \geq b^s(x)$. We conclude by noting that $b^s(x) \geq b_i(x)$.

On the other hand, if $b(x) = \text{List}(v)$ and $b(y) = \text{List}(w)$, then b_{i+1} is b_i but E_{i+1} is E_i with $v \sim w$. We must show that $(v \sim w) \in E^s$. As (b^s, E^s) is at least as general as (b_i, E_i) but not degenerated we know that $b^s(x) = \text{List}(v)$ and $b^s(y) = \text{List}(w)$ as well. Again, because $r((b^s, E^s)) = (b^s, E^s)$ we conclude that $(v \sim w) \in E^s$.



6.9.4 Complexity

The lattice is finite, although its size depends on the size of the program. The List part has the same size as V , and the Pbc part is exponential on the number of pre-built constants. However, in this model a chain of annotations cannot be longer than:

$$\max(5, \text{number-of-pbcs} + 3, \text{depth-of-class-hierarchy} + 3).$$

In the extended lattice used in practice it is more difficult to compute an upper bound. Such a bound exists -- some considerations can even show that a finite subset of the extended lattice suffices -- but it does not reflect any practical complexity considerations. It is simpler to prove that there is no infinite ascending chain, which is enough to guarantee termination.

We will not present a formal bound on the complexity of the algorithm. Worst-case scenarios would expose severe theoretical problems. In practice, these scenarios are unlikely. Empirically, when annotating a large program like PyPy consisting of some 20'000 basic blocks from 4'000 functions, the whole annotation process finishes in 5 minutes on a modern computer. This suggests that our approach scales quite well. We also measured how many times each rule is re-applied; the results change from run to run due to the non-deterministic nature of the meta-rule -- we pick a random next rule to apply at each step -- but seems to be consistently between 20 and 40, which suggests an $n \log(n)$ practical complexity.

Moreover, we will have to explore modular annotation in the near future for other reasons -- to make the compiled PyPy interpreter modular, which is an important strength of CPython. We plan to do this by imposing the annotations at selected interface boundaries and annotating each part independently.

6.10 Non-static aspects and extensions

In practice, the annotation is much less "static" than the theoretical model presented above. All functions and classes are discovered while annotating, not in advance. In addition, as explained above, annotation occasionally reverts to concrete mode execution to force lazy objects to be computed or to fill more caches. We describe below some of these aspects.

6.10.1 Specialization

The type system used by the annotator does not include polymorphism support beyond object-oriented polymorphism with subclasses and overriding and parametric polymorphism for builtin containers (lists, ...). In this respect we opted for simplicity, considering this in most cases sufficient for the kind of system programming RPython is aimed at and matching our main targets.

Not all of our target code or our needs for expressiveness fit into this model. The fact that we allow unrestricted dynamism at bootstrap helps a great deal, but in addition we also support the explicit flagging of certain functions or classes as requiring special treatment. One such special treatment is support for parametric polymorphism. If this were supported for all callables, it would lead to an explosion of function implementations and likely the need for some kind of target specific type erasure and coalescing. Instead, the user-provided flag instructs the annotator to only create a new copy of a few specific functions for each annotation seen for a specific argument.



Another special treatment is more outright special casing (black-boxing): the user can provide code to explicitly compute the annotation information for a given function, without letting the flow object space and annotator abstractly interpret the function's bytecode.

In more details, the following special-cases are supported by default (more advanced specializations have been implemented specifically for PyPy):

- specializing a function by the annotation of a given argument
- specializing a function by the value of a given argument (requires all calls to the function to resolve the argument to a constant value)
- ignoring -- the function call is ignored. Useful for writing tests or debugging support code that should be removed during translation.
- by arity -- for functions taking a variable number of (non-keyword) arguments via a `*args`, the default specialization is by the number of extra arguments. (This follows naturally from the fact that the extended annotation lattice we use has annotations of the form $Tuple(a_1, \dots, a_n)$ representing a heterogeneous tuple of length n whose items are respectively annotated with a_1, \dots, a_n , but there is no annotation for tuples of unknown length.)
- `ctr_location` -- for classes. A fresh independent copy of the class is made for each program point that instantiate the class. This is a simple (but potentially over-specializing) way to obtain class polymorphism for the couple of container classes we needed in PyPy (e.g. Stack).
- memo -- the calls to such functions are fully computed at annotation time. This requires that each call site finds each argument to be either constant or element of a finite set (either $Pbc(m)$ or $Bool$). All possible call patterns are tried at annotation time, and the return annotation is the union of all possible results. Such functions are then compiled as memo table lookups -- their implementation is neither analysed nor translated.

6.10.2 Concrete mode execution

The *memo specialization* is used at key points in PyPy to obtain the effect described in the introduction (see [Abstract interpretation](#)): the memo functions and all the code it invokes is concretely executed during annotation. There is no staticness restriction on that code -- it will typically instantiate classes, creating more pre-built instances, and sometimes even build new classes and functions; this possibility is used quite extensively in PyPy.

The input arguments to a memo function are not known in advance: they are discovered by the annotator, typically as a $Pbc(m)$ annotation where the set m grows over time when rules are re-applied. In this sense, switching to concrete mode execution is an integral part of our annotation process.

6.10.3 Constant propagation

The extended lattice of annotations used in practice differs from the one presented above in that almost any annotation can also optionally carry a constant value. For example, there are annotations like $NonNegInt(const = 42)$ for all integers; these annotations are between *Bottom* and *NonNegInt*. The annotator carries the constant tag across simple operations whenever



possible. The main effect we are researching here is not merely constant propagation (low-level compilers are very good at this already), but dead code removal. Indeed, when a branch condition is $Bool(const = True)$ or $Bool(const = False)$, then the annotator will only follow one of the branches -- or in the above formalism: the $y = \text{phi}(x)$ rules that propagate annotations across links between basic blocks are guarded by the condition that the switch variable carries an annotation of either $Bool(const = \langle \text{link case} \rangle)$ or $Bool$.

The dead code removal effect is used in an essential way to hide bootstrap-only code from the annotator where it could not analyse such code. For example, some frozen pre-built constants force some of their caches to be filled when they are frozen (which occurs the first time the annotator discovers such a constant). This allows the regular access methods of the frozen pre-built constant to contain code like:

```
if self.not_computed_yet:
    self.compute_result()
return self.result
```

As the annotator only sees frozen `self` constants with `not_computed_yet=False`, it annotates this attribute as $Bool(const = False)$ and never follows the call to `compute_result()`.

6.10.4 Narrowing

Conditional branching has sometimes the effect of "narrowing" the annotation of the variables involved in the check. For example:

```
if isinstance(obj, MySubClass):
    ...positive case...
else:
    ...negative case...
```

In the basic block at the beginning of the positive case, the input block variable corresponding to the source-level `obj` variable is annotated as $Inst(MySubClass)$. Similarly, in:

```
if x > y:
    ...positive case...
else:
    ...negative case...
```

If y is annotated as $NonNegInt$, then the annotation corresponding to x is narrowed from (typically) Int to $NonNegInt$.

This is implemented by introducing an extended family of annotations for boolean values:

$$Bool(v_1 : (t_1, f_1), v_2 : (t_2, f_2), \dots)$$

where the v_n are variables and t_n and f_n are annotations. The result of a check is typically annotated with such an extended $Bool$. The meaning of the annotation is as follows: if the run-time value of the boolean is `True`, then we know that each variable v_n has an annotation at most as general as t_n ; and if the boolean is `False`, then each variable v_n has an annotation at most as general as f_n . This information is propagated from the check operation to the exit of the block via such an extended $Bool$ annotation, and the conditional exit logic uses it to trim the annotation it propagates.

More formally, one of the rules for (say) the comparison operation `greater_than` is:



$$\frac{z = \text{greater_than}(x, y), b(x) = \text{Int}, b(y) = \text{NonNegInt}}{b' = b \text{ with } (z \rightarrow \text{Bool}(x : (\text{NonNegInt}, \text{Int})))}$$

Then if v_{cond} is a boolean variable used as the exit condition of a block, we can describe the above process as being based on a more complicated “phi” rule. For each variable x that exits the current block along the “positive” link and enters the next block as a variable y , we have:

$$\frac{y = \text{phi}(x), b(v_{cond}) = \text{Bool}(\dots x : (t, f) \dots)}{\text{merge } (b(x) \wedge t) \Rightarrow y}$$

and similarly with f along the “negative” link. Here \wedge stands for the intersection operation in the annotation lattice.

It is possible to define an appropriate lattice structure that includes the extended *Bool* annotations and show that all soundness properties described above still hold. (The tricky point is to get the rules to still respect the [Generalisation](#) property if we also have constant annotations, as mentioned at the end of the [Annotation model](#). It requires constant *Bool* annotations -- i.e. known to be True or known to be False -- that are nevertheless extended as above, even though it seems redundant, just in case the annotation needs to be generalised to a non-constant extended annotation. See for example `builtin_isinstance()` in [pypy/annotation/builtin.py](#).)

6.10.5 Termination with non-static aspects

The non-static aspects, and concrete mode execution more particularly, makes it impossible to prove that annotation terminates in general. It could be the case that a memo function builds and returns a new class for each class that it receives as argument, and the result of this memo function could be fed back to its input. However, what can be proved is that annotation terminates under some conditions on the user program. A typical sufficient condition (which is true for PyPy) is that there must be a computable bound on the number of functions and classes that can ever exist in the user program at run-time.

For annotation to terminate -- and anyway for translation to a low-level language like C to have any chance of being reasonably straightforward to do -- it is up to the user program to satisfy such a condition. (It is similar to, but more “global” than, the flow object space’s restriction to terminate only if fed functions that do not obviously go into infinite loops.)



7 Code Generation

The actual generation of low-level code from the information computed by the annotator is not the central subject of the present report, so we will only skim it and refer to the reference documentation when appropriate.

The main difficulty with turning annotated flow graphs into, say, C code is that the RPython definition is still quite large. It supports a lot of the built-in data structures of Python, with most of their methods. Some of these data structures require either tedious or non-trivial implementations (e.g. dictionaries). Additionally, to use the type information computed by the annotator, we need some kind of polymorphic implementation (e.g. dictionaries with integer keys are not the same as dictionaries with string keys). Various approaches have been tried out, including writing a lot of template C code that gets filled with concrete types.

The approach eventually selected is different. We proceed in two steps:

- the [RTyper](#) rewrites the annotated graphs so that each RPython-level operation is replaced by one or a few low-level operations (or a call to a helper for more complex operations);
- a back-end generates code for the target language and environment based on the low-level flow graphs thus obtained.

We can currently generate C-like low-level flow graphs and turn them into either C or [LLVM](#) code; or experimentally, PowerPC machine code or [JavaScript](#). If we go through slightly less low-level flow graphs instead, we can also interface with an experimental back-end generating [Squeak](#) and in the future Java and/or .NET.

7.1 RTyper

The first step is called "RTyping", short for "RPython low-level typing". It turns general high-level operations into low-level C-like operations between variables with C-like types. This process is driven by the information computed by the annotator, and it produces a globally consistent family of low-level flow graphs by assuming that the annotation state is sound. It is described in more details in the [RTyper reference \(TR\)](#).

7.1.1 Low-level flow graphs

The purpose of the RTyper is to produce control flow graphs that contain a different set of variables and operations. At this level, the variables are typed, and the operations between them are constrained to be well-typed. The exact set of types and operations depends on the target environment's language; currently, we have defined two such sets:

- [lltype](#): a set of C-like types ([TR](#)). Besides primitives (integers, characters, and so on) it contains structures, arrays, functions and "opaque" (i.e. externally-defined) types. All the non-primitive types can only be manipulated via pointers. Memory management is still partially implicit: the back-end is responsible for inserting either reference counting or other forms of garbage collecting for some kinds of structures and arrays. Structures can directly contain substructures as fields, a feature that we use to implement instances in the presence of subclassing -- an instance of a class *B* is a structure whose first field is a substructure corresponding to the parent class *A*.



The operations are: arithmetic operations between primitives, pointer casts, reading/writing a field from/to a structure via a pointer, and reading/writing an array item via a pointer to the array.

- ootype: a set of low-level but object-oriented types. It mostly contains classes and instances and ways to manipulate them, as needed for RPython.

Besides the same arithmetic operations between primitives, the operations are: creating instances, calling methods, accessing the fields of instances, and some limited amount of run-time class inspection.

7.1.2 Representations

While the back-end only sees the typed variables and operations in the resulting flow graphs, the RTyper uses internally a powerful abstraction: *representation* objects. The representations are responsible for mapping the RPython-level types, as produced by the annotator, to the low-level types.

One representation is created for each used annotation. The representation maps a low-level type to each annotation in a way that depends on information discovered by the annotator. For example, the representation of *Inst* annotations are responsible for building the low-level type -- nested structures and vtable pointers, in the case of `lltype`. In addition, the representation objects' central role is to know precisely how, on a case-by-case basis, to turn the high-level RPython operations into operations on the low-level type -- e.g. how to map the `getattr` operation to the appropriate "fishing" of a field within nested substructures.

As another example, the annotator records which RPython lists are resized after their creations, and which ones are not. This allows the RTyper to select one of two different representations for each list annotation: the resizable lists need an extra indirection level when implemented as C arrays, while the fixed-size lists can be implemented more efficiently. A more extreme example is that lists that are discovered to be the result of a `range()` call and never modified get a very compact representation whose low-level type only stores the start and the end of the range of numbers.

7.1.3 Helpers and LPython

A noteworthy point of the RTyper is that for each operation that has no obvious C-level equivalent, we write a helper function in Python; each usage of the operation in the source (high-level) annotated flow graph is replaced by a call to this function. The function in question is implemented in terms of "simpler" operations. The function is then fed back into the flow object space and the annotator and the RTyper itself, so that it gets turned into another low-level control flow graph. At this point, the annotator runs with a different set of default specializations: it allows several copies of the helper functions to be automatically built, one for each low-level type of its arguments. We do this by default at this level because of the intended purpose of these helpers: they are usually methods of a polymorphic container.

This approach shows that our annotator is versatile enough to accommodate different kinds of sub-languages at different levels: it is straightforward to adapt it for the so-called "low-level Python" language in which we constrain ourselves to write the low-level operation helpers. Automatic specialization was a key point here; the resulting language feels like a basic C++ without any type or template declarations.



7.2 The back-ends

So far, all data structures (flow graphs, pre-built constants...) manipulated by the translation process only existed as objects in memory. The last step is to turn them into an external representation. This step, while basically straightforward, is messy in practice for various reasons including the limitations, constraints and irregularities of the target language (particularly so if it is C). Additionally, the back-end is responsible for aspects like memory management and exception model, as well as for generating alternate styles of code for different execution models like coroutines.

We will give as an example an overview of the GenC back-end ([TR](#)). The [LLVM back-end](#) works at the same level. The (undocumented) Squeak back-end takes ootyped graphs instead, as described above, and faces different problems (e.g. the graphs have unstructured control flow, so they are difficult to render in a language with no `goto` equivalent). The C back-end works itself again in two steps:

- it first collects recursively all functions (i.e. their low-level flow graphs) and all pre-built data structures, remembering all "struct" C types that will need to be declared;
- it then generates one or multiple C source files containing:
 1. a forward declaration of all the "struct" C types;
 2. the full declarations of the latter;
 3. a forward declaration of all the functions and pre-built data structures;
 4. the implementation of the latter (i.e. the body of functions and the static initialisers of pre-built data structures).

Each function's body is implemented as basic blocks (following the basic blocks of the control flow graphs) with jumps between them. The low-level operations that appear in the low-level flow graphs are each turned into a simple C operation. A few functions have no flow graph attached to them: the "primitive" functions. No body is written for them; GenC assumes that a manually-written implementation will be provided in another C file.



8 Conclusion

We have presented a flexible static analysis and compilation toolchain that is suitable for a restricted subset of Python called RPython. (We have also argued against the existence or usefulness of such a tool for full Python or any sufficiently dynamic language; instead, PyPy contains a complete interpreter for the full Python language, itself written in RPython.)

Our approach seems to be general enough to insert a variety of low-level aspects during successive phases of the translation and target a number of quite different languages and platforms. It is thus a tool that can be used to compile portable RPython programs to all of these platforms. As described in more details in (LLA), the still high level of abstraction of RPython is an important factor in hiding the platform-specific details as well as the particular needs of a program in terms of execution model.

We have presented a detailed model of the [Annotator](#), which is our central analysis component. This model is quite regular, with an abstract interpretation basis. This is why it can be easily extended or even – in our opinion – quickly adapted to perform type inference on any other language with related properties.

We have given a short overview of the [RTyper](#), which is our central cross-level translation component. This overview should have given some hints about how we use variations of the RTyper to target very different platforms. In addition, the basic principles of the RTyper are again regular enough to allow it to be easily extended to support a larger RPython language or even adapted to different but related languages, like the Annotator.

8.1 Limits of Static analysis

Static analysis is and remains slightly fragile in the sense that the input program must be globally consistent (inconsistent types, even locally, could yield to the propagation through the whole program of the *Top* annotation). This is also a reason why we believe that dynamic analysis is ultimately more powerful.

In PyPy, our short-term future work is to focus on using the translation toolchain presented here to generate a modified low-level version of the same full Python interpreter. This modified version will drive a just-in-time specialization process, in the sense of providing a description of full Python that will not be directly executed, but specialized for the particular user Python program.

As of October 2005, we are only starting the work in this direction. The details are not fleshed out nor documented yet, but the ([Psyco](#)) project has already given a proof of concept.

8.2 Test-driven development

As a conclusion, we should reiterate the importance of test-driven development. The complete Annotator and RTyper have been built in this way, by writing small test cases covering each aspect even before implementing that aspect. This has proven essential, especially because of the absence of medium-sized RPython programs: we have jumped directly from small tests and examples to the full PyPy interpreter, which is about 50'000 lines of code. Any problem or limitation of the Annotator discovered in this way was added back as a small test.

To help locate typing errors in the source RPython program, the Annotator can complain on the first appearance of the degenerated *Top* annotation. This was not possible until recently, because the *Top* annotation was an essential fall-back while the toolchain itself was being



developed. But now, under the condition that the analysed RPython program is itself extensively tested -- a common theme of our approach -- our toolchain should be robust enough and give useful information about error locations.



9 Glossary of Abbreviations

Glossary and links mentioned in the text:

- Abstract interpretation: http://en.wikipedia.org/wiki/Abstract_interpretation
- CPython: <http://www.python.org>
- Flow Object Space: see [Object Space](#).
- JavaScript: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- LLVM (Low-Level Virtual Machine): <http://llvm.cs.uiuc.edu/>
- Object Space: <http://codespeak.net/pypy/dist/pypy/doc/objspace.html>
- Perl 6 compiler mailing list post: <http://www.nntp.perl.org/group/perl.perl6.compiler/1107>
- Squeak: <http://www.squeak.org/>
- Standard Object Space: see [Object Space](#).
- Thunk Object Space: see [Object Space](#).

References

- (ARCH) Architecture Overview, PyPy documentation. <http://codespeak.net/pypy/dist/pypy/doc/architecture.html>
- (TR) Translation, PyPy documentation. <http://codespeak.net/pypy/dist/pypy/doc/translation.html>
- (LLA) Encapsulating low-level implementation aspects, PyPy documentation. <http://codespeak.net/pypy/dist/pypy/doc/low-level-encapsulation.html>
- (Psyco) Home page: <http://psyco.sourceforge.net>. Paper: Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python, ACM SIGPLAN PEPM'04, August 24-26, 2004, Verona, Italy. <http://psyco.sourceforge.net/psyco-pepm-a.ps.gz>
- (PyPy) <http://codespeak.net/pypy/>
- (SSI) C. S. Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Sep 1999.
- (SSA) R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, Oct 1991.
- (DaMi) Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, pages 207-212. ACM Press, 1982.
- (Milner) Robin Milner: A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17(3): 348-375, 1978.