IST FP6-004779

PYPY

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

STREP

IST Priority 2

D10.1 Aspect-Oriented, Design-by-Contract Programming and Advanced Static Checking Capabilities in PyPy

Due date of deliverable: March 2007

Actual Submission date: March 23rd, 2007

Start date of Project: 1st December 2004 **Duration: 28 months**

Lead Contractor of this WP: Logilab

Authors: Alexandre Fayolle, Adrien Di Mascio, Sylvain Thénault

Revision: Final

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

Dissemination Level: PU (Public)

Revision History

Date	Name	Reason of Change
Feb 16 2007	Alexandre Fayolle	Interim version: exec. summary, AOP intro, RPylint
Mar 09 2007	Alexandre Fayolle	Added AST mod part, and AOP
Mar 14 2007	Alexandre Fayolle	More information on AOP
Mar 20 2007	Sylvain Thénault	Take Jacob's remark on the RPylint part into account
Mar 21 2007	Alexandre Fayolle	Take Jacob's remark on the AOP part into account
Mar 21 2007	Carl Friedrich Bolz	Publish Final version on the Web Page

Abstract

This document presents several dynamic aspects of PyPy, especially the ability to modify the Abstract Syntax Tree of modules at import time, and how this facility is used to build an Aspect Oriented Programming module.

A pure python static checker for RPython is also presented.

Purpose, Scope and Related Documents

This report introduces the `aop` module of PyPy, which is used to write Python code using Aspect Oriented Programming. It also presents the RPylint checker of Logilab's Pylint tool.

This documents is related to the Abstract Syntax Tree analysis and manipulation of python and RPython programs.

Logilab provided an Aspect Oriented Programming module, which works by changing the AST of a module when it is parsed by the python interpreter to weave advices in the code of the module.

To answer needs which arose during the work on WP11, Logilab also developed an extension to its Pylint static code checker, which specialises on analysing RPython code. This checker, nicknamed RPylint can be used to help python developers convert their Python code to RPython by finding snippets which are not supported by RPython faster than the PyPy annotator.

The AST modification task of WP10 uses the facilities provided by WP04. This reports assumes that the reader has a good knowledge of the deliverables of that work package and especially:

- D04.3: Report about the parser and byte-code compiler implementation

Contents

1 Executive Summary

The PyPy projects has led to a more flexible version of the Python interpreter. Part of this flexibility means that it is possible to change the grammar of the language at run time, in order to add new features to the language. It is also possible with PyPy to dynamically change the Abstract Syntax Tree of a module when it is parsed in memory by the interpreter. These two facilities can be used to add features such as Aspect Oriented Programming (AOP) to Python: we can change the grammar of Python to add point cut and weaving syntax. We show how the AOP module could be used to provide a Design by Contract facility.

The RPython translator is not able to cope with fully general Python code, and is not yet always user friendly enough when dealing with RPython. This report also presents a special RPython checker for the Pylint static python code analyser.

2 Modification of the Abstract Syntax Tree

2.1 API Specification

The `parser` module of PyPy exposes a way of accessing the AST of a module just after its creation by the parser, before this AST is fed to the byte-compiler. This facility provides the programmer with a way of examining or fixing the AST on the fly.

The main entry point is `parser.install_compiler_hook(callback)` which takes a callback function as an argument. This function will be called each time a new AST has been built:

- when a python file is parsed
- when a python string is parsed by `eval` or `exec`

The function should accept three arguments: the first one is a reference on the newly created `parser.ASTModule` instance, the second one is a string containing the name of the encoding of the file or of the string that has been parsed, and the third one is the name of the parsed file (in case of the evaluation or execution of a string, the value of the filename argument is `"<string>"`).

The function can then examine the AST and make some in-place changes to it. After the callback function returns normal operations resume and the modified AST is fed to the compiler for byte-code generation.

The AST examination can be done manually. However, the AST nodes are meant to be visited with an `ASTVisitor` or `ASTMutator` instance. They expose two methods to achieve this:

- `node.accept(ASTVisitor)`: this is the standard Visitor design pattern, in which each node calls the appropriate `visitXXX` method on the visitor instance for itself and all its child nodes. This method is meant for read only access on the AST.
- `node.mutate(ASTMutator)`: when used, this method will run through the AST in a slightly different way, and the `visitXXX` methods are expected to return an `ASTNode` instance, which will replace the visited node in the AST.

AST mutation is meant to be performed with great care: it is easy to generate an invalid AST which will result in crashes of the interpreter.

2.2 Example

The following example is a very simple demonstration which changes each literal 3 in an AST to 2:

```
import parser
class ConstMutator(parser.ASTMutator):
    def visitConst(self, node):
        if node.value == 3:
            node.value = 2
        return node

def threebecomestwo(ast, enc, filename):
    ast.mutate(ConstMutator())
    return ast

 # install the hook
 parser.install_compiler_hook(threebecomestwo)
 print eval('3*2')
```

When run, this code samples prints `4`, and not `6`.

We use `eval` here to have the AST generation happen after the installation of the hook. Importing an external module, provided this is done after the installation of the hook will achieve the same effect.

3 AOP library

3.1 Goals of AOP

Aspect-Oriented Programming was originally described by Gregor Kiczales and al. in [AOP97]. An aspect is a representation of a software design concern which affects a large part of a program. Such concerns are said to be "cross-cutting", and include synchronisation, persistence, logging, security, value sharing, rewrite of performance-critical code snippets, etc.

The goal of AOP is to make it possible to deal with cross-cutting aspects of a system's behaviour as separately as possible. In Aspect-Oriented software design, programmers express each of the system's aspect of concern in a separate and natural form, and then automatically combine those descriptions into an executable. This process is known as weaving.

An aspect can alter the behavior of the base code by adding behavior at various locations in the program, known as join points. These join points are described using point cuts which describe a precise location in the code. Another family of aspects deals with making structural changes to classes, for instance adding members or parents.

3.2 AOP in CPython

AOP implementations exist for a wide variety of programming languages, and many languages have several implementations available. A list can be found on the aosd.net tools for developers wiki page. While this list does not mention python tools for AOP, such tools exist, for instance:

- lightweight aop for Python
- springpython.aop

- logilab.aspects

These AOP implementations use the dynamicity of the Python language to do the weaving at run time using introspection and dynamic code replacement (commonly known as “monkey patching”).

Some features of standard CPython can be seen as providing functionalities matching the goals of AOP. The decorators facility [PEP318] is an especially useful way of modifying the behaviour of a set of functions. The main use of decorators in CPython is to execute a some code before and/or after some functions by wrapping the function call in another function. This is useful, though not as feature rich as a full blown AOP framework. The main limitation of this approach is its lack of transparency: writing a generic decorator which preserves the signature of the decorated functions is not straightforward, nor is handling exceptions occurring inside the decorated function without the decorator showing up. Moreover, this mainly restricts the scope of the aspects to execution point cuts, which, while useful, do not offer a model as rich as standard AOP.

3.3 AOP in PyPy

The Aspect Oriented Programming module provided in PyPy is a pure python module, which uses the advanced facilities provided by PyPy’s `parser` module. Importing the `aop` module has several side effects:

- an instance of `aop.Weaver` is created and stored in `__builtin__.__aop__`, which makes it globally available in modules which will be imported later
- this instance registers one of its methods with `parser.install_compiler_hook` (see above)

From then on, any module imported will be examined by the `Weaver` instance.

The code can define new aspects by using the `Aspect` metaclass in the `aop` module. This metaclass can register methods decorated with an advice from the `aop` module with the `Weaver` when the `Aspect` is instantiated. When a module is imported, the `Weaver` looks for symbols (classes, functions and methods) which are targets of registered aspects and weaves the matching aspects into the function. The weaving is done by mutating the AST nodes of the module. The mutation generally involves calling a method on the `__aop__` instance with a unique identifier. The Weaver will use this identifier at run time to find which advice it should execute.

This approach has a number of important consequences when compared to the traditional (dynamic) approaches used in CPython AOP modules:

- The code is only woven when the `module.py` source file is read from disc. If a module has already byte-compiled in a `module.pyc` file, it will not be seen by the weaver, because the import code will not involve building an AST. On the other hand, the woven code will be saved in the byte-compiled module.
- However, once the aspects have been woven in the module, they are present in the byte-code, without a new weaving taking place.

This static weaving also has a number of drawbacks. The first and most significant one is that pre-compiling modules at installation is a very common practise. Furthermore, once an aspect is woven in a globally installed byte-compiled module, all the code using this module will see and execute the woven advices, which is generally not what is intended. This makes weaving code on library modules awkward, but not altogether impossible: an application designer wanting to weave aspects on third party modules should symlink or copy the source modules in a dedicated directory of his own source tree, and configure the search path adequately. PyPy also provides an option which tells the interpreter to ignore byte-compiled files when importing a module. In this way, the modules are re-parsed, and aspects get woven, each time the module is imported.

3.4 Introduction to the aop module

The `aop` module offers an API which is strongly inspired by AspectC++_. It uses standard python facilities to declare aspects statically.

3.4.1 Aspects

Aspects are declared in classes using the `aop.Aspect` metaclass. Such classes can have normal methods, and additionally methods decorated with an advice decorator, and aspect instances will be registered with the Weaver by the metaclass. These decorators take a `PointCut` as an argument to describe the join points.

3.4.2 Advices

The available advices are:

- `before`: the advice code will be woven before a dynamic join point
- `after`: the advice code will be woven after a dynamic join point
- `around`: the advice code will be woven in replacement of the dynamic join point
- `introduce`: the advice code will be added to a static join point.

3.4.3 Point cuts

A static point cut is created by instantiating the `aop.PointCut` class, passing one, two or three arguments. Each argument is a regular expression which can match a package/module, a class or a function/method.

Point cuts can be combined using the python binary operators & and |, which will produce a new PointCut representing respectively the intersection and the union of the join points of the original point cuts.

A dynamic point cut can be obtained from a static point cut by calling one of the following methods:

method	dynamic pointcut matching
pc.call()	all the places where the method or function matched by the static point-cut is called
pc.execution()	the execution of the function or method matched by the static point cut
pc.initialization()	the initialization of the class matched by the static point cut
pc.destruction()	the destruction of the class matched by the static point cut

3.5 Example

In the startup code of the application, before all other imports:

```
from aop import Aspect, before, PointCut
class LoggingAspect:
    __metaclass__=Aspect

    # We weave in all methods and functions, except if the name of
    # the method starts with an underscore
```

```
        @before(PointCut(module='mytoplevelpackage',
                         klass='.*', func='[^_].*').execution())
        def logStartMethod(self, joinpoint):
            print 'in', joinpoint.name()
    log = LoggingAspect()
```

Caveat: the declaration in the file which imports the aop module for the first time are not considered for weaving, since the module has been parsed before the hook in installed.

3.6 Design by contract

Design by contract is a way of writing programs described by Bertrand Meyer [DBC86] in which the programmer specifies pre and post conditions for the methods, as well as class invariants, and the programming language guarantees that these conditions and invariants are met at run time, by raising an exception if they are not.

There are several third party libraries for Python which provide design by contract facilities:

- pyDBC uses a metaclass which will check that assertions written in method foo__pre and foo__post are valid before and after a call to method foo
- pycontract uses method docstring with a special format, which are compiled by the module on demand to produce the checks
- logilab.aspects provides an implementation using a generic aspect which parses a docstring format similar to the one used by pycontract, and weaves in the pre and post condition methods.

These three implementations are pure-python, and work with PyPy.

It is not possible to directly port the logilab.aspects implementation of design by contract to PyPy's aop module, mainly because of the static nature of AST-level weaving of this implementation. However, it is possible to achieve the same effects, as demonstrated by the dbc module of PyPy. This module provides an around advice which gets woven on method execution: the code of method is surrounded by a call on _pre_method and a call on _post_method if they exist. These methods are expected to return True, otherwise, an exception is raised.

3.7 Conclusion

The module demonstrates the feasibility of weaving code at import time in PyPy, by mutating the AST tree of the module. The difficult part of the implementation was that the code we need to weave in the imported module is not available as an AST, but as compiled Python objects. Two options are available to let both world communicate. The first one is converting the compiled python objects to AST, for instance by reparsing the module declaring the aspects. The second one, chosen in this implementation, is to add an indirection layer, namely the __aop__ built-in instance, and to generate new AST nodes using that instance. This choice was made because it lead to a slightly simpler implementation, especially because it removed the necessity to take care of symbol collision and scope issues, and allowed more dynamicity in the writing of the Advices. However, it is not clear whether working at the AST level to weave advices gives significant advantages over the dynamic approach used by all other implementations of AOP in Python. On the other hand, the hook provided in the PyPy for the edition of the AST provides a very powerful tool at application level, and the idea could easily be generalized by providing some more hooks in the parsing process, for instance giving a chance of modifying the source code before it is parsed to an AST. This could be the first step towards adding macros to the language.

The dbc module shows that it is possible to use the aspect oriented facilities provided by the aop module to add design by contract facilities to PyPy. However, we feel that this approach to implementing DBC is not the best one in a language such as Python, and did not push the effort too far. It seems to us that using a facility such as PyDBC or pycontract is a more pythonic way of making DBC available to Python programmers, with the added benefit of being available on both PyPy and CPython.

4 Checking RPython programs

4.1 Need for a checking tool

For a significant number of users, one key benefit of the PyPy project is the availability of RPython, which makes it possible to translate python code to various back-ends, including C, LLVM and Javascript. This means that they can write RPython code and compile it to a native program, directly executable on the processor, and therefore gain significant speed compared to using the Python interpreter, and still gain benefits from the Python language in terms of readability, memory management, exception handling, etc. over C or C++ code. Such people are quite often experienced Python programmers and moving from standard Python to RPython can be tricky: RPython is very close to Python, but there are a lot of very commonly used features in Python which are not available. For instance, some builtin functions and some methods on builtin types are not available.

Of course the translation process and the annotator will generally fail if the code is not RPython. However this failure is currently not as nice as it could be (and will surely be in future releases of PyPy). The main shortcomings of the translation process are:

- the translation can take a long time and is not incremental,
- the errors are reported one by one: this can be irritating, especially if combined with the lengthy translation above, especially for experienced Python programmers with little or no RPython experience
- some of the error messages are not as explicit as expected, mostly when a high level problem is detected at a lower level during the translation
- the translation script is built with a successful run in mind, therefore errors are treated as exceptions, which means that the error is reported as a stack trace.

The RPython checker is meant to overcome some of these shortcomings, by reporting all the problems it can find in one pass, using nice and explicit messages, and if possible being faster than the translator.

4.2 Introduction to Pylint

Pylint is a modular static type and coding style checker for Python developed by Logilab. The main features of Pylint are:

- modularity: checkers are available as modules which can be enabled or disabled,
- extendability: users can add their own checkers,
- configurability: each checker provides its set of options and checks, which can be enabled or disabled from several places, including the command line, configuration files, and the source code being examined (to silence known false positives).

Pylint works at the source code level, without importing the modules. It builds an abstract syntax tree for the module, does some type inference and runs the various checkers available according to the configuration.

Pylint is also integrated in some commonly used Python development environments, including Eclipse/Pydev, Eric3, Emacs, and vim. While the integration level varies greatly from one environment to another, it provides at least a quick way of launching Pylint and examining suspicious lines of code.

Since Pylint is meant to work on Python code, its type inference capabilities are far less advanced than those of the PyPy annotator, even when working on RPython code. However, in order to quickly get a list of points to check on a source file in order to convert it to RPython, Pylint can help.

4.3 RPylint features

We have added a new checker for Pylint dedicated to RPython. This checker is not enabled by default, but a wrapper script called RPylint enables it.

This checker provides the following verifications:

- use of unavailable keywords or constructions such as generator expressions
- use of unavailable Python builtins
- use of unavailable python protocols (e.g. *__new__*)
- rules for multiple inheritance
- type consistency of identifiers and object attributes
- homogeneity of list content's types
- module variables are supposed to be constants after the module has been initialised, and cannot therefore be modified in functions or methods
- use of negative indices in slices

These checks are planned but not yet implemented:

- use of unimplemented methods on builtin types (such as list.sort, string.rsplit, etc.)
- conformity of the translation entry point
- number of arguments given to functions like *os.path.join* (which can not be called with more than two arguments today)
- changing Pylint analysis mode to be closer to the translator

The type inference in pylint is coarser than the one done during translation, and consistency checks are much less strict than the equivalent checks performed by the RPython annotator. They can therefore lead to false positives or false negatives.

4.4 RPylint contribution to RPython

One thing that has been developed for the RPython checker that will hopefully be useful to RPython development is its test framework. This framework is based on isolating simple samples of code demonstrating a Python feature that is expected to be translatable (or not). These samples are used to:

- check whether the translation actually fails or succeeds

- check that RPylint issues the correct message for the issue

One problem with RPython is that the language has no specification and is evolving as PyPy is being developed. This framework could help in:

- collecting a set of code snippets demonstrating what is available in RPython, and what is not
- help to disseminate information when something that was not available is being implemented by an newer version of PyPy.

4.5 Conclusion

RPylint has proven its ability to correctly detect the issues it checks for without producing too many false positives, in one run, and in a acceptable time. While the currently available checks are not enough to be very useful to the experimented RPython programmer, they are very helpful for beginners or even to get a quick idea of how long it would take to port a Python program to RPython.

There is still a lot of room for enhancements in this tool, which could at some point even become useful for experimented RPython programmers when more verifications are added. However, the huge differences in the inference techniques involved in RPylint and the PyPy tool chain make it impossible to guarantee the success of the translation of code that passed RPylint's checks successfully.

5 Glossary of Abbreviations

The following abbreviations may be used within this document:

5.1 Technical Abbreviations:

AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.

IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple DirectMedia Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

5.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH
Impara	Impara GmbH

References

[AOP97] *Aspect-Oriented Programming*, Gregor Kiczales, John Irving, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, 1997

[DBC86] *Design by Contract*, Bertrand Meyer, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986

[PEP318] *Decorators for Functions and Methods*, Kevin D. Smith, Jim Jewett, Skip Montanaro, Anthony Baxter, 2003