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

D05.4: Publish an overview paper about the success of encapsulating low level language aspects as well defined parts of the translation phase

Due date of deliverable: September 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, Michael Hudson (HHU)

Revision: final

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

Dissemination Level: PU (Public)

Abstract

This document describes the sucess we have had in encapsulating low level language aspects as well defined parts of the translation phase. We describe briefly how our architecture allows us to do this and consider a number of examples of ascpects we have so encapsulated.

Contents

1 Executive Summary

It always has been a major goal of PyPy to not force implementation decisions. This means that even after the implementation of the standard interpreter[1] has been written we are still able to experiment with different approaches to memory management or concurrency and to target wildly different platforms such as the Java Virtual Machine or a very memory-limited embedded environment.

We do this by allowing the encapsulation of these low level aspects as well defined parts of the translation process.

In the following document we give examples of aspects that have been successfully encapsulated in more detail and contrast the potential of our approach with CPython.

2 Introduction

2.1 Purpose of this Document

This document intends to give a high-level overview of how our approach allows our translation tool chain to affect fundamental aspects of the resulting binary.

2.2 Scope of this Document

This document describes at a high-level various features and capabilities of our translation tool-chain. It does not go into detail about how these capabilities are implemented.

2.3 Related Documents

This document has been prepared in conjunction with other tasks in work packages 3, 4 and 5. Before reading this document, a close look at the following deliverable is recommended.

- D4.2 Complete Python implementation running on top of CPython

In addition D5.3 "Memory management and threading models as translation aspects -- solutions and challenges" describes some of the translation aspects covered in this document in more detail.

3 Background

One of the better known significant modifications to CPython are Christian Tismer's "stackless" patches (STK), which allow far more flexible control flow than the typical function call/return supported by CPython. Originally implemented as a series of invasive patches, Christian found that maintaining these patches as CPython itself was further developed was time consuming

[1] standard interpreter is our term for the code which implements the Python language, i.e. the interpreter and the standard object space.

to the point of no longer being able to work on the new functionality that was the point of the exercise.

One solution to the dilemma would have been for the patches to become part of core CPython but this was not done, partly because the code that fully enabled stackless required widespread modifications which made the code harder to understand (as the "stackless" model contains control flow that is not easily expressable in C, the implementation became much less "natural" in some sense).

With PyPy, however, it is possible to obtain this flexible control flow whilst retaining transparent implementation code as the necessary modifications can be implemented as a localized translation aspect, and indeed this was done at the Paris sprint in a couple of days (compared to about six months needed for the original stackless patches).

Of course this is not the only aspect that can be decided this way a posteriori, during translation.

4 Translation Aspects

Our standard interpreter is implemented at a very high level of abstraction. This has a number of lucky consequences, among which is enabling the encapsulation of language aspects as described in this document. For example the implementation code simply makes no reference to memory management, which clearly gives complete freedom to the translator to decide about this aspect. This constrasts with CPython where the decision to use reference counting is reflected tens or even hundreds of times in each C source file in the codebase.

As described in (ARCH), producing a Python implementation from the source of our standard interpreter involves various stages: the initialization code is run, the resulting code is annotated, typed and finally translated. By the nature of the task, the encapsulation of *low-level aspects* mainly affects the typer and the translation process. At the coarsest level the selection of target platform involves writing a new backend -- still a significant task, but much easier than writing a complete implementation of Python!

Other aspects affect different levels, as their needs require. The remainder of this section describes a few aspects that we have successfully enscapsulated.

An advantage of our approach is that any combination of aspects can be selected freely, avoiding the problem of combinatorial explosion of variants which can be seen in manually written interpreters.

4.1 Stacklessness

The stackless modifications are mostly implemented in the C backend, with a single extra flow graph operation to influence some details of the generated C code. The total changes only required about 300 lines of source, vindicating our abstract approach.

In stackless mode, the C backend generates functions that are systematically extended with a small amount of bookkeeping code. This allows the C code to save its own stack to the heap on demand, where it can then be inspected, manipulated and eventually resumed. This is described in more detail in (TA). In this way unlimited (or more precisely heap-limited) recursion is possible, even on operating systems that limit the size of the C stack. Alternatively a different saved stack can be resumed, thus implementing soft context switches - coroutines, or green threads with an appropriate scheduler. We reobtain in this way all the major benefits of the original "stackless" patches.

This effect requires a number of changes in each and every C function that would be extremely tedious to write by hand: checking for the signal triggering the saving of the stack, actually saving precisely the currently active local variables, and when re-entering the function check which variables are being restored and which call site is resumed. In addition, a couple of global tables must be maintained to drive the process. The point is that we can fine-tune all these interactions freely without having to rewrite the whole code all the time but only modifying the C backend (in addition, of course, being able at any time to change the high-level code that is the input of the translation process). So far this has allowed us to find a style that does not hinder the optimizations performed by the C compiler and as such only has a minor impact on performance in the normal case.

Also note that the fact that the C stack can be saved fully into the heap can tremendously simplify the portable implementation of garbage collection: after the stack has been transferred to the heap completely, there are no roots left on the stack.

4.2 Multiple Interpreters

Another implementation detail that causes tension between functionality and both code clarity and memory consumption in CPython is the issue of multiple independent interpreters in the same process. In CPython there is a partial implementation of this idea in the "interpreter state" API, but the interpreters produced by this are not truly independent -- for instance the dictionary that contains interned strings is implemented as file-level static object, and is thus shared between the interpreters. A full implementation of this idea would entirely eschew the use of file level statics and place all interpreter-global data in some large structure, which would hamper readability and maintainability. In addition, in many situations it is necessary to determine which interpreter a given object is "from" -- and this is not possible in CPython largely because of the memory overhead that adding a 'interp' pointer to all Python objects would create.

In PyPy, all of our implementation code manipulates an explicit object space instance as described in (CODG). The situation of multiple interpreters is thus handled automatically: if there is only one space instance, it is regarded as a pre-constructed constant and the space object pointer (though not its non-constant contents) disappears from the produced source, i.e. from function arguments, local variables and instance fields. If there are two or more such instances, a 'space' attribute will be automatically added to all application objects (or more precisely, it will not be removed by the translation process), the best of both worlds.

4.3 Memory Management

As mentioned above, CPython's decision to use a garbage collector based on reference counting is reflected throughout its source. In the implementation code of PyPy, it is not, and in fact the standard interpreter can currently be compiled to use a reference counted scheme or the Boehm GC (BOEHM). Again, more details are in (TA). We also have an experimental framework for developing custom exact GCs (GC), but it is not yet integrated with the low-level translation backends.

Another advantage of the aspect oriented approach shows itself most clearly with this memory management aspect: correctness. Although reference counting is a fairly simple scheme, writing code for CPython requires that the programmer makes a large number of not-quite-trivial decisions about the refcounting code. Experience suggests that mistakes will always creep in, leading to crashes or leaks. While tools exist to help find these mistakes, it is surely better to not write the reference count manipulations at all and this is what PyPy's approach allows. Writing the code that emits the correct reference count manipulations is surely harder than writing any single piece of explicit refcounting code, but once it is done and tested, it just works without further effort.

4.4 Concurrency

The aspect of CPython's implementation that has probably caused more discussion than any other mentioned here is the threading model. Python has supported threads since version 1.5 with what is commonly referred to as the "Global Interpreter Lock" or GIL; the execution of bytecodes is serialized such that only one thread can execute Python code at any one time. On the one hand this is relatively unintrusive and not too complex, but on the other multi-threaded, computation-bound Python code does not gain performance on multi-processor machines.

PyPy will offer the opportunity to experiment with different models, although currently we only offer a version with no thread support and another with a GIL-like model (TA). (We also plan to support soon "green" software-only threads in the Stackless model described above, but obviously this would not solve the multi-processor scalability issue.)

The future work in this direction is to collect the numerous possible approaches that have been thought out along the years and e.g. presented on the CPython development mailing list. Most of them have never been tried out in CPython, because of a lack of necessary resources. A number of them are clearly easy to try out in PyPy, at least in an experimental version that would allow its costs to be assessed -- for example, various forms of object-level locking.

4.5 Evaluation Strategy

Possibly the most radical aspect to tinker with is the evaluation strategy. The thunk object space (OBJS) wraps the standard object space to allow the production of "lazily computed objects", i.e. objects whose values are only calculated when needed. It also allows global and total replacement of one object with another.

The thunk object space is mostly meant as an example of what our approach can acheive -- the combination of side-effects and lazy evaluation is not easy to understand. This demonstration is important because this level of flexibility will be required to implement future features along the lines of Prolog-style logic variables, transparent persistency, object distribution across several machines, or object-level security.

5 Experimental results

All aspects described in the previous chapter have been successfully implemented and are available since the release 0.7 or 0.8 of PyPy.

We have conducted preliminary experimental measures of the performance impact of enabling each of these features in the compiled PyPy interpreter. We present below the current results as in October 2005. Most figures appear to vary from machine to machine. Given that the generated code is large (it produces a binary of 5.6MB on a Linux Pentium), there might be locality and code ordering issues that cause important cache effects.

We have not particularly optimized any of these aspects yet. Our goal is primarily to prove that the whole approach is worthwhile, and rely on future work and push for external contributions to implement state-of-the-art techniques in each of these domains.

Stacklessness

> Producing Stackless-style C code means that all the functions of the PyPy interpreter that can be involved in recursions contain stack bookkeeping code (leaf functions, functions calling only leaves, etc. do not need to use this style). The current performance impact is to make PyPy slower by about 8%. A couple of minor pending optimizations could reduce this figure a bit. We expect the rest of the performance impact to be mainly caused by the increase of size of the generated executable (+28%).

Multiple Interpreters

> A binary that allowed selection between two copies of the standard object space with a command line switch was about 10% slower and about 40% larger than the default. Most of the extra size is likely accounted for by the duplication of the large amount of prebuilt data involved in an instance of the standard object space.

Memory Management

> The (Boehm) GC is well-optimized and produces excellent results. In comparison using reference counting instead makes the interpreter twice as slow. This is almost certainly due to the naive approach to reference counting used so far, which updates the counter far more often than strictly necessary; we also still have a lot of objects that would theoretically not need a reference counter, either because they are short-lived or because we can prove that they are "owned" by another object and can share its lifetime. In the long run it will be interesting to see how far this figure can be reduced, given past experiences with CPython which seem to show that reference counting is a viable idea for Python interpreters.

Concurrency

> No experimental data available so far. Just enabling threads currently creates an overhead that hides the real costs of locking.

Evaluation Strategy

> When translated to C code, the Thunk object space has a global performance impact of 6%. The executable is 13% bigger (probably due to the arguably excessive inlining we perform).

We have described five aspects in this document, each currently with two implementation choices, leading to 32 possible translations. We did not measure the performance of each variant, but the few we have tried suggest that the performance impacts are what one would expect, e.g. a translated stackless binary using the thunk object space would be expected to be about 1.06 x 1.08 ~= 1.14 times slower than the default and was found to be 1.15 times slower.

6 Conclusion

Although the work is still in progress we believe that the success we have had in enscapsulating implementation aspects justifies the approach we have taken. In particular, the relative ease of implementing the translation aspects described in this paper -- as mentioned above, the stackless modifications took only a few days -- means we are confident that it will be easily possible to encapsulate implementation aspects we have not yet considered.

7 Glossary of Abbreviations

The following abbreviations may be used within this document:

7.1 Technical Abbreviations:

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.
docutils	The Python documentation utilities.
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.
Graphviz	Graph visualisation software from AT&T.
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 Library - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
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.

7.2 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

References

(ARCH) Architecture Overview, PyPy documentation, 2003-2005

(BOEHM) Boehm-Demers-Weiser garbage collector, a garbage collector for C and C++, Hans Boehm, 1988-2004

(CODG) Coding Guide, PyPy documentation, 2003-2005

(GC) Garbage Collection, PyPy documentation, 2005

(OBJS) Object Spaces, PyPy documentation, 2003-2005

(STK) Stackless Python, a Python implementation that does not use the C stack, Christian Tismer, 1999-2004

(TA) Memory management and threading models as translation aspects, PyPy documentation (and EU Deliverable D05.3), 2005