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

D5.3: Memory management and threading models as translation aspects -- solutions and challenges

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: Carl Friedrich Bolz (merlinux), Armin Rigo (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 basic implementation of threading and memory management models within the PyPy project. Both of the two aspects are not directly visible in the executable specification of Python, which is constitued by our Interpreter Implementation, but are interweaved while translating the specification to a lower level backend.

Contents

1 Executive Summary **4**

2 Introduction **4**

2.1 Purpose of this Document 4

2.2 Scope of this Document 4

2.3 Related Documents 4

3 Introduction **4**

4 The Low Level Object Model **5**

4.1 Subclass Checking 6

4.2 Identity Hashes 6

4.3 Cached Functions with PBC Arguments 6

4.4 Changing the Representation of an Object 6

5 Automatic Memory Management Implementations **7**

5.1 Using the Boehm Garbage Collector 7

5.2 Using a Simple Reference Counting Garbage Collector 7

5.3 Simple Escape Analysis to Remove Memory Allocation 8

5.4 A General Garbage Collection Framework 8

6 Concurrency Model Implementations **9**

6.1 No Threading 9

6.2 Threading with a Global Interpreter Lock 9

6.3 Stackless C Code 9

7 Future Work **12**

7.1 Garbage Collection 12

7.2 Threading Model 12

7.3 Object Model 12

8 Conclusion **12**

9 Glossary of Abbreviations **12**

9.1 Technical Abbreviations: 13

9.2 Partner Acronyms: 13

1 Executive Summary

The compliant Python implementation can be seen as a rather abstract nevertheless executable specification of the language. Memory management and threading models are introduced as translation aspects; the 0.7 release of PyPy contained choices for memory management and threading which manage to not require modifications to the executable specification (D04.2).

2 Introduction

2.1 Purpose of this Document

This document describes the status of PyPy's implementation of memory management and threading models as translation aspects.

2.2 Scope of this Document

This document describes the integration status of memory and thread models within the PyPy code base. It relates to the 0.7 and 0.8 releases which include choices for memory management and threading models. This document additionally describes a memory-management simulation framework which implements several GC algorithms (This has yet to be fully integrated with the translation tool chain. The integration will happen during Phase 2 of the project).

2.3 Related Documents

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

- D04.2 Complete Python implementation on top of CPython.
- D05.2 A compiled self-contained version of PyPy.
- D05.4 Encapsulating low level language aspects

3 Introduction

One of the goals of the PyPy project is to have memory and concurrency models flexible and changeable without having to reimplement the interpreter manually. In fact, PyPy, by the time of the 0.8 release contains code for memory management and concurrency models which allows experimentation without requiring early design decisions. This document describes many of the more technical details of the current state of the implementation of the memory object model, automatic memory management and concurrency models and it describes possible future developments.

4 The Low Level Object Model

One important part of the translation process is *rtyping* (DLT), (TR). Before that step all objects in our flow graphs are annotated with types at the level of the RPython type system which is still quite high-level and target-independent. During rtyping they are transformed into objects that match the model of the specific target platform. For C or C-like targets this model consists of a set of C-like types like structures, arrays and functions in addition to primitive types (integers, characters, floating point numbers). This multi-stage approach provides flexibility in how a given object is represented at the target's level. The RPython process can decide which representation to use based on the type annotation and on the way the object is used.

In the following the structures used to represent RPython classes are described. There is one "vtable" per RPython class, with the following structure: The root class "object" has a vtable of the following type (expressed in a C-like syntax):

```
struct object_vtable {
    struct object_vtable* parenttypeptr;
    RuntimeTypeInfo * rtti;
    Signed subclassrange_min;
    Signed subclassrange_max;
    array { char } * name;
    struct object * instantiate();
}
```

The structure members `subclassrange_min` and `subclassrange_max` are used for subclass checking (see below). Every other class X, with parent Y, has the structure:

```
struct vtable_X {
    struct vtable_Y super;   // inlined
    ...                      // extra class attributes
}
```

The extra class attributes usually contain function pointers to the methods of that class, although the data class attributes (which are supported by the RPython object model) are stored there.

The type of the instances is:

```
struct object {       // for instances of the root class
    struct object_vtable* typeptr;
}

struct X {            // for instances of every other class
    struct Y super;   // inlined
    ...               // extra instance attributes
}
```

The extra instance attributes are all the attributes of an instance.

These structure layouts are quite similar to how classes are usually implemented in C++.

4.1 Subclass Checking

The way we do subclass checking is a good example of the flexibility provided by our approach: in the beginning we were using a naive linear lookup algorithm. Since subclass checking is quite a common operation (it is also used to check whether an object is an instance of a certain class), we wanted to replace it by the more efficient relative numbering algorithm (see (PVE) for an overview of techniques). This was a matter of changing just the appropriate code of the rtyping process to calculate the class-ids during rtyping and insert the necessary fields into the class structure. It would be similarly easy to switch to another implementation.

4.2 Identity Hashes

In the RPython type system, class instances can be used as dictionary keys using a default hash implementation based on identity, which in practice is implemented using the memory address. This is similar to CPython's behavior when no user-defined hash function is present. The annotator keeps track of the classes for which this hashing is ever used.

One of the peculiarities of PyPy's approach is that live objects are analyzed by our translation toolchain. This leads to the presence of instances of RPython classes that were built before the translation started. These are called "pre-built constants" (PBCs for short). During rtyping, these instances must be converted to the low level model. One of the problems with doing this is that the standard hash implementation of Python resorts to using the id of object which is just the memory address of the underlying C-structure. If the RPython program explicitely captures the hash of a PBC by storing it (for example in the implementation of a data structure) then the stored hash value will not match the value of the object's address after translation.

To prevent this the following strategy is used: for every class whose instances are hashed somewhere in the program (either when storing them in a dictionary or by calling the hash function) an extra field is introduced in the structure used for the instances of that class. For PBCs of such a class this field is used to store the memory address of the original object and new objects have this field initialized to zero. The hash function for instances of such a class stores the object's memory address in this field if it is zero. The return value of the hash function is the content of the field. This means that instances of such a class that are converted PBCs retain the hash values they had before the conversion whereas new objects of the class have their memory address as hash values. A similar strategy is required, anyway, if we want to use a copying garbage collector later on.

4.3 Cached Functions with PBC Arguments

As explained in (DLT) the annotated code can contain functions from a finite set of PBCs to something else. The set itself has to be finite but its content does not need to be provided explicitly but is discovered as the annotation of the input argument by the annotator itself. This kind of function is translated by recording the input-result relationship by calling the function concretely at annotation time, and adding a field to the PBCs in the set and emitting code reading that field instead of the function call.

4.4 Changing the Representation of an Object

One example of the flexibility the RTyper provides is how we deal with lists. Based on information gathered by the annotator the RTyper chooses between two different list implemen-

tations. If a list never changes its size after creation, a low-level array is used directly. For lists which might be resized, a representation consisting in a structure with a pointer to an array is used, together with over-allocation.

We plan to use similar techniques to use tagged pointers instead of using boxing to represent builtin types of the PyPy interpreter such as integers. This would require attaching explicit hints to the involved classes. Field access would then be translated to the corresponding masking operations.

5 Automatic Memory Management Implementations

The whole implementation of the PyPy interpreter assumes automatic memory management, e.g. automatic reclamation of memory that is no longer used. The whole analysis toolchain also assumes that memory management is being taken care of -- only the backends have to concern themselves with that issue. For backends that target environments that have their own garbage collector, like Smalltalk or Javascript, this is not an issue. C and LLVM backends have to provide support code for garbage collection themselves.

This approach has several advantages. It makes possible to target different platforms, with and without integrated garbage collection. Furthermore, the interpreter implementation is not burdened by the need to do explicit memory management everywhere. Even more important the backend can optimize the memory handling to fit a certain situation (like a machine with very restricted memory) or completely replace the memory management technique or memory model with a different one without the need to change source code. Additionally, the backend can use information that was inferred by the rest of the toolchain to improve the quality of memory management.

5.1 Using the Boehm Garbage Collector

Currently there are two different garbage collectors implemented in the C backend (which is the most complete backend right now). One of them uses the existing Boehm-Demers-Weiser garbage collector (BOEHM). For every memory allocating operation in a low level flow graph the C backend introduces a call to a function of the boehm collector which returns a suitable amount of memory. Since the C backend has a lot of information available about the data structure being allocated it can choose the memory allocation function out of the Boehm API that fits best. For example, for objects that do not contain references to other objects (e.g. strings) there is a special allocation function which signals to the collector that it does not need to consider this memory when tracing pointers.

Using the Boehm collector has disadvantages as well. The problems stem from the fact that the Boehm collector is conservative which means that it has to consider every word in memory as a potential pointer. Since PyPy's toolchain has complete knowledge of the placement of data in memory we could generate an exact garbage collector that considers only genuine pointers.

5.2 Using a Simple Reference Counting Garbage Collector

The other implemented garbage collector is a simple reference counting scheme. The C backend inserts a reference count field into every structure that has to be handled by the garbage collector and puts increment and decrement operations for this reference count

into suitable places in the resulting C code. After every reference decrement operation a check is performed whether the reference count has dropped to zero. If this is the case the memory of the object will be reclaimed after the references counts of the objects the original object refers to are decremented as well.

The current placement of reference counter updates is far from being optimal: The reference counts are updated more often than theoretically necessary (e.g. sometimes a counter is increased and then immediately decreased again). Objects passed into a function as arguments can almost always use a "trusted reference", because the call-site is responsible to create a valid reference. Furthermore some more analysis could show that some objects don't need a reference counter at all because they either have a very short foreseeable life-time or because they live exactly as long as another object.

Another drawback of the current reference counting implementation is that it cannot deal with circular references, which is a fundamental flaw of reference counting memory management schemes in general. CPython solves this problem by having special code that handles circular garbage which PyPy lacks at the moment. This problem has to be addressed in the future to make the reference counting scheme a viable garbage collector. Since reference counting is quite succesfully used by CPython it will be interesting to see how far it can be optimized for PyPy.

5.3 Simple Escape Analysis to Remove Memory Allocation

We also implemented a technique to reduce the amount of memory allocation. Sometimes it is possible to deduce from the flow graphs that an object lives exactly as long as the stack frame of the function it is allocated in. This happens if no pointer to the object is stored into another object and if no pointer to the object is returned from the function. If this is the case and if the size of the object is known in advance the object can be allocated on the stack. To achieve this, the object is "exploded", that means that for every element of the structure a new variable is generated that is handed around in the graph. Reads from elements of the structure are removed and just replaced by one of the variables, writes by assignments to same.

Since quite a lot of objects are allocated in small helper functions, this simple approach which does not track objects across function boundaries only works well in the presence of function inlining.

5.4 A General Garbage Collection Framework

In addition to the garbage collectors implemented in the C backend we have also started writing a more general toolkit for implementing exact garbage collectors in Python. The general idea is to express the garbage collection algorithms in Python as well and translate them as part of the translation process to C code (or whatever the intended platform is).

To be able to access memory in a low level manner there are special `Address` objects that behave like pointers to memory and can be manipulated accordingly: it is possible to read/write to the location they point to a variety of data types and to do pointer arithmetic. These objects are translated to real pointers and the appropriate operations. When run on top of CPython there is a *memory simulator* that makes the address objects behave like they were accessing real memory. In addition the memory simulator contains a number of consistency checks that expose common memory handling errors like dangling pointers, uninitialized memory, etc.

At the moment we have three simple garbage collectors implemented for this framework: a simple copying collector, a mark-and-sweep collector and a deferred reference counting collector. These garbage collectors are working when run on top of the memory simulator, but at the moment it is not yet possible to translate PyPy to C with them. This is because it is not easy to find the root pointers that reside on the C stack -- both because the C stack layout is heavily platform dependent, and also due to the possibility of roots that are not only on the stack but also hiding in registers (which would give a problem for *moving garbage collectors*).

There are several possible solutions for this problem: One of them is not to use C compilers to generate machine code, so that the stack frame layout gets into our control. This is one of the tasks that need to be tackled in phase 2, as directly generating assembly is needed anyway for a just-in-time compiler. The other possibility (which would be much easier to implement) is to move all the data away from the stack to the heap before collecting garbage, as described in section "Stackless C code" below. In summary, fully integrating our garbage collection algorithm remains a task for later phases in the project.

6 Concurrency Model Implementations

At the moment we have implemented two different concurrency models, and the option of not supporting concurrency at all (another proof of the modularity of our approach): threading with a global interpreter lock and a "stackless" model.

6.1 No Threading

By default, multi-threading is not supported at all, which gives some small benefits for single-threaded applications since even in the single-threaded case there is some overhead if threading capabilities are built into the interpreter.

6.2 Threading with a Global Interpreter Lock

Right now, there is one non-trivial threading model implemented. It follows the threading implementation of CPython and thus uses a global interpreter lock. This lock prevents any two threads from interpreting python code at the same time. The global interpreter lock is released around calls to blocking I/O functions. This approach has a number of advantages: it gives very little runtime penalty for single-threaded applications, makes possible many of the common uses for threading, and it is relatively easy to implement and to maintain. The disadvantage is that multiple threads cannot be distributed accross multiple proccessors.

To make this threading-model usable for I/O-bound applications, the global intepreter lock should be released around blocking external function calls (which is also what CPython does). This has been partially implemented.

6.3 Stackless C Code

"Stackless" C code is C code that only uses a bounded amount of space in the C stack, and that can generally obtain explicit control of its own stack. This is commonly known as "continuations", or "continuation-passing style" code, although in our case we will limit ourselves

to single-shot continuations, i.e. continuations that are captured and subsequently will be resumed exactly once.

The technique we have implemented is based on the recurring idea of emulating this style via exceptions: a specific program point can generate a pseudo-exception whose purpose is to unwind the whole C stack in a restartable way. More precisely, the "unwind" exception causes the C stack to be saved into the heap in a compact and explicit format, as described below. It is then possible to resume only the innermost (most recent) frame of the saved stack -- allowing unlimited recursion on OSes that limit the size of the C stack -- or to resume a different previously-saved C stack altogether, thus implementing coroutines or light-weight threads.

In our case, exception handling is always explicit in the generated code: the C backend puts a cheap check after each call site to detect if the callee exited normally or generated an exception. So when compiling functions in stackless mode, the generated exception handling code special-cases the new "unwind" exception. This exception causes the current function to respond by saving its local variables to a heap structure (a linked list of records, one per stack frame) and then propagating the exception outwards. At the end of the frame chain, the outermost function is a manually-written dispatcher that catches the "unwind" exception.

At this point, the whole C stack is stored away in the heap. This is a very interesting state in itself, because precisely there is no C stack below the dispatcher left. It is this which will allow us to write all the algorithms in a portable way, that normally require machine-specific code to inspect the stack, in particular garbage collectors.

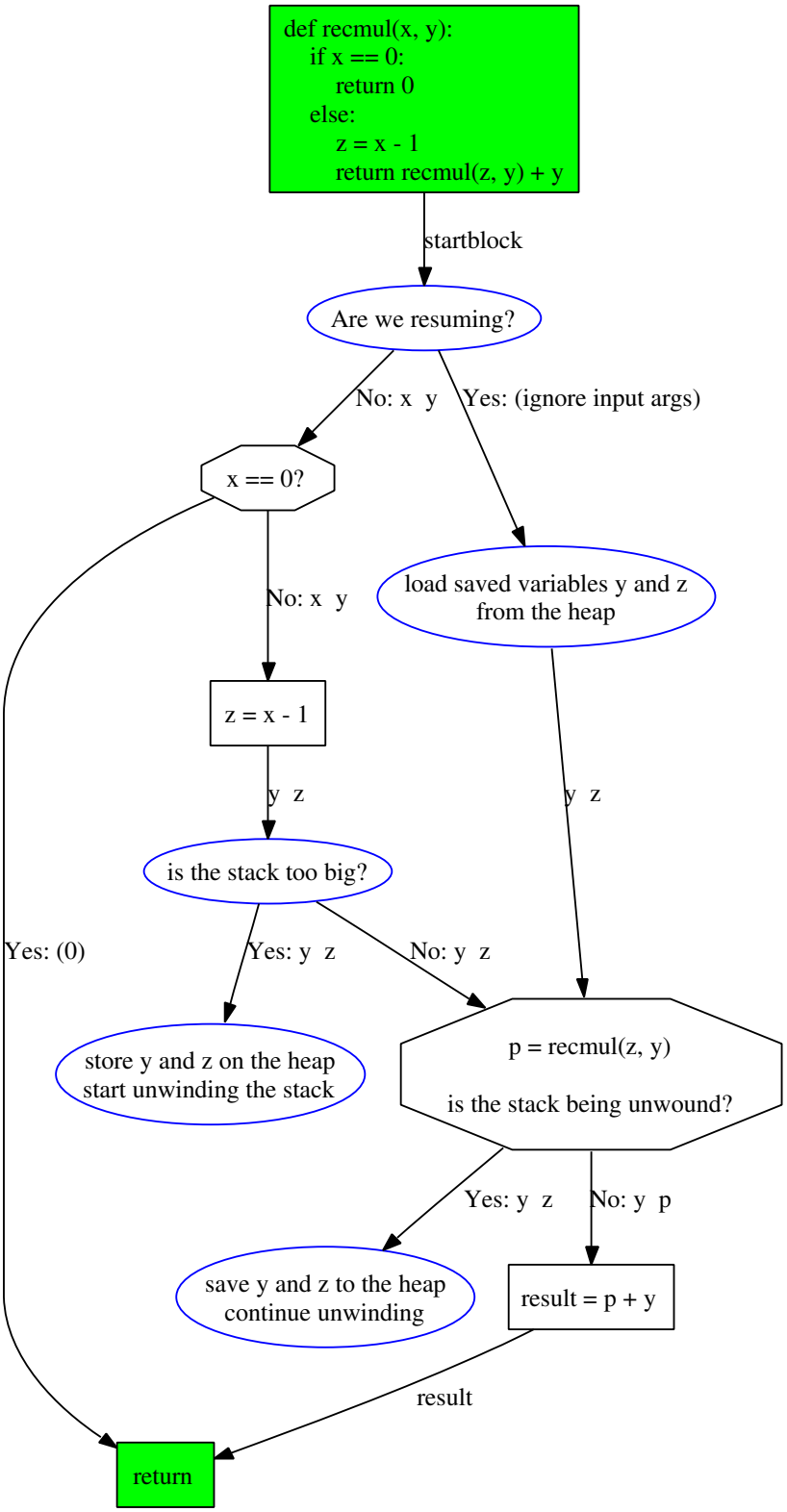
To continue execution, the dispatcher can resume either the freshly saved or a completely different stack. Moreover, it can resume directly the innermost (most recent) saved frame in the heap chain, without having to resume all intermediate frames first. This not only makes stack switches fast, but it also allows the frame to continue to run on top of a clean C stack. When that frame eventually exits normally, it returns to the dispatcher, which then invokes the previous (parent) saved frame, and so on. We insert stack checks before calls that can lead to recursion by detecting cycles in the call graph. These stack checks copy the stack to the heap (by raising the special exception) if it is about to grow deeper than a certain level. As a different point of view, the C stack can also be considered as a cache for the heap-based saved frames in this model. When we run out of C stack space, we flush the cache. When the cache is empty, we fill it with the next item from the heap.

To give the translated program any amount of control over the heap-based stack structures and over the top-level dispatcher that jumps between them, there are a few "external" functions directly implemented in C. These functions provide an elementary interface, on top of which useful abstractions can be implemented, like:

- coroutines: explicitly switching code, similar to Greenlets (GREENLET).
- "tasklets": cooperatively-scheduled microthreads, as introduced in Stackless Python (STK).
- implicitly-scheduled (preemptive) microthreads, also known as green threads.

An important property of the changes in all of the generated C functions is that they are written in a way that does only minimally degrade their performance in the non-exceptional case. Most optimizations performed by C compilers, like register allocation, continue to work...

The following picture shows a graph function together with the modifications that are necessary for the stackless style: the check whether the stack is too big and should be unwound, the check whether we are in the process of currently storing away the stack and the check whether the call to the function is not a regular call but a reentry call.


def recmul(x, y):
if x == 0:
return 0
else:
z = x - 1
return recmul(z, y) + y
startblock
Are we resuming?
No: x y
Yes: (ignore input args)
x == 0?
No: x y
load saved variables y and z
from the heap
z = x - 1
y z
y z
is the stack too big?
Yes: (0)
Yes: y z
No: y z
store y and z on the heap
start unwinding the stack
p = recmul(z, y)
is the stack being unwound?
Yes: y z
No: y p
save y and z to the heap
continue unwinding
result = p + y
result
return

7 Future Work

Open challenges for phase 2:

7.1 Garbage Collection

One of the biggest missing features of our current garbage collectors is finalization. At present finalizers are simply not invoked if an object is freed by the garbage collector. Along the same lines weak references are not supported yet. It should be possible to implement these with a reasonable amount of effort for reference counting as well as the Boehm collector (which provides the necessary hooks).

Integrating the now simulated-only GC framework into the rtyping process and the code generation will require a considerable effort. It requires being able to keep track of the GC roots which is hard to do with portable C code. One solution would be to use the "stackless" code since it can move the stack completely to the heap. We expect that we can implement GC read and write barriers as function calls and rely on inlining to make them more efficient.

We may also spend some time on improving the existing reference counting implementation by removing unnecessary incref-decref pairs and identifying trustworthy references. A bigger task would be to add support for detecing circular references.

7.2 Threading Model

One of the interesting possibilities that stackless offers is to implement *green threading*. This would involve writing a scheduler and some preemption logic.

We should also investigate other threading models based on operating system threads with various granularities of locking for access of shared objects.

7.3 Object Model

We also might want to experiment with more sophisticated structure inlining. Sometimes it is possible to find out that one structure object allocated on the heap lives exactly as long as another structure object on the heap pointing to it. If this is the case it is possible to inline the first object into the second. This saves the space of one pointer and avoids pointer-chasing.

8 Conclusion

As concretely shown with various detailed examples, our approach gives us flexibility and lets us choose various aspects at translation time instead of encoding them into the implementation itself.

9 Glossary of Abbreviations

The following abbreviations may be used within this document:

9.1 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 Direct-media 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.

9.2 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

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

Hans Boehm, 1988-2004

(GREENLET) Lightweight concurrent programming, py-lib Documentation 2003-2005

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

(TR) Translation, PyPy documentation, 2003-2005

(LE) Encapsulating low-level implementation aspects, PyPy documentation (and EU deliverable D05.4), 2005

(DLT) Compiling dynamic language implementations, PyPy documentation (and EU deliverable D05.1), 2005

(PVE) Simple and Efficient Subclass Tests, Jonathan Bachrach, Draft submission to ECOOP-02, 2001