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

D08.2 JIT Compiler Architecture

Due date of deliverable: March 2007

Actual Submission date: 2nd May, 2007

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

Lead Contractor of this WP: Strakt

Authors: Armin Rigo, Samuele Pedroni

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
2006-11-30	A.Rigo, S.Pedroni	Outline
2006-12-12	S.Pedroni	Drafting introduction
2006-12-13	A.Rigo, S.Pedroni	First sections about binding-time annotation
2006-12-15	A.Rigo	Started explaining “timeshifting”
2006-12-27	A.Rigo, S.Pedroni	Intermediate version up to timeshifting of calls
2006-12-30	S.Pedroni	Drafting Executive Summary
2007-01-28	A.Rigo	Executive Summary reviewed
2007-01-28	S.Pedroni	Publishing of intermediate version on the web
2007-04-26	A.Rigo, S.Pedroni	Describe Promotion and Virtual structures
2007-04-30	A.Rigo, S.Pedroni	Report completed
2007-05-01	PyPy team	Internal review
2007-05-01	C.F.Bolz	Publishing of final version on the web

Abstract

PyPy’s translation tool-chain – from the interpreter written in RPython to generated VMs for low-level platforms – is now able to extend those VMs with an automatically generated dynamic compiler, derived from the interpreter. This is achieved by a pragmatic application of partial evaluation techniques guided by a few hints added to the source of the interpreter. Crucial for the effectiveness of dynamic compilation is the use of run-time information to improve compilation results: in our approach, a novel powerful primitive called “promotion” that “promotes” run-time values to compile-time is used to that effect. In this report, we describe it along with other novel techniques that allow the approach to scale to something as large as PyPy’s Python interpreter.

Purpose, Scope and Related Documents

This document describes the new techniques and architecture of how our translation tool-chain can derive a dynamic compiler from an interpreter.

The framework implementing these techniques, as a translation aspect, was released with PyPy 1.0, and within that release can be applied to PyPy’s Python interpreter. For practical details see [D08.1].

Some familiarity with dynamic compilation and partial evaluation can be useful for reading this document.

Related documents:

- [D08.1] Release a JIT Compiler for PyPy Including Processor Backends for Intel and PowerPC
- [VMCDLS] PyPy’s approach to virtual machine construction
- [D07.1] Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects
- [D05.4] Overview paper about the success of encapsulating low level language aspects as well defined parts of the translation phase
- [D05.1] Compiling Dynamic Language Implementations

Contents

1 Executive Summary

Performance is one of the important factors to enable the benefits of dynamic languages to reach wider applicability. Dynamic compilation techniques in the form of just-in-time compilers can help achieve a good level of performance, in a range comparable to the level of statically compilable languages.

However, writing a high-performance production-quality just-in-time compiler may require a lot of effort and fine-tuned, hand written dynamic compilers may well be fragile with respect to changes to language and its semantics. Open-source dynamic languages (like Python) tend to evolve quickly and their design community does not usually consider ease of compilation as a design constraint. Typically, the main implementation of these languages is a straight-forward bytecode interpreter with no dynamic compilation.

New solutions are needed to solve these contradictory demands. Ideally we would like to be able to simply *generate automatically* the dynamic compiler from the straight-forward bytecode interpreter.

This was the main research goal and what we set out to do since the inception of the PyPy project.

The PyPy project planned since the start to enhance the produced Python bytecode virtual machines with a just-in-time compiler. This dynamic compiler – like low-level aspects such as memory management – is not part of the source interpreter but is *generated during the translation process,* by transformation of the translated interpreter. The transformation is guided by a few hints interspersed within the interpreter source and should to a large extent be robust against language changes and evolution. It should also be directly usable with interpreters for very different programming languages.

The transformation itself is a pragmatic application of partial evaluation techniques inspired by Psyco, a hand-written run-time specializer that can be used with CPython. Our approach uses a novel powerful primitive called "promotion" that "promotes" run-time values to compile-time. Promotion is the key to exploiting run-time information to produce optimized code.

Together, promotion and other new techniques we introduced make dynamic compiler generation possible at all and effective.

2 Introduction

Dynamic compilers are resource costly to write and hard to maintain, but highly desirable for competitive performance. Straight-forward bytecode interpreters are easier to write. Hybrid approaches have been experimented with [REJIT], but this is clearly an area in need of research and innovative approaches.

One of the central goals of the PyPy project is to automatically produce dynamic compilers from an interpreter, with as little modifications of the interpreter code base itself as possible.

The forest of flow graphs that the translation process [VMCDLS] generates and transforms constitutes a reasonable base for the necessary analyses. That's a further reason why having a high-level runnable and analyzable interpreter implementation was always a central tenet of the project: in our approach, the dynamic compiler is just another aspect transparently introduced by and during the translation process.

Partial evaluation techniques should, at least theoretically, allow such a derivation of a compiler from an interpreter [PE], but it is not reasonable to expect the code produced for an input program by a compiler derived using partial evaluation to be very good, especially in the case of a dynamic language. Essentially, the input program doesn't contain enough information to generate good code; for example the input program contains mostly no kind of type information in that case.

What is really desired is not to produce a compiler doing static ahead-of-time compilation, as classical partial evaluation would do, but one capable of dynamic compilation, exploiting run-time information in its result. Compilation should be able to suspend, let the produced code run to collect run-time information (for example

language-level types), and then resume with this extra information. This will allows the compiler to generate code optimized for the effective run-time behaviour of the program.

Inspired by Psyco [PSYCO], which is a hand-written dynamic compiler based on partial evaluation for Python, we developed a technique - *promotion* - for our dynamic compiler generator. Simply put, promotion on a value stops compilation and waits until the run-time reaches this point. When it does, the actual run-time value is promoted into a compile-time constant, and compilation resumes with this extra information.

Promotion is an essential technique to be able to generate really dynamic compilers that can exploit run-time information. Besides promotion, the novel techniques introduced by PyPy that allow the approach to scale are virtualizable structures and need-oriented binding time analysis (see the corresponding sections).

2.1 Overview of partial evaluation

Partial evaluation is the process of evaluating a function, say `f(x, y)`, with only partial information about the values of its arguments, say the value of the `x` argument only. This produces a *residual* function `g(y)`, which takes less arguments than the original – only the information not specified during the partial evaluation process needs to be provided to the residual function, in this example the `y` argument.

Partial evaluation (PE) comes in two flavors:

- *On-line* PE: a compiler-like algorithm takes the source code of the function `f(x, y)` (or its intermediate representation, i.e. its control flow graph in PyPy's terminology), and some partial information, e.g. `x=5`. From this, it produces the residual function `g(y)` directly, by following in which operations the knowledge `x=5` can be used, which loops can be unrolled, etc.
- *Off-line* PE: in many cases, the goal of partial evaluation is to improve performance in a specific application. Assume that we have a single known function `f(x, y)` in which we think that the value of `x` will change slowly during the execution of our program – much more slowly than the value of `y`. An obvious example is a loop that calls `f(x, y)` many times with always the same value `x`. We could then use an on-line partial evaluator to produce a `g(y)` for each new value of `x`. In practice, the overhead of the partial evaluator might be too large for it to be executed at run-time. However, if we know the function `f` in advance, and if we know *which* arguments are the ones that we will want to partially evaluate `f` with, then we do not need a full compiler-like analysis of `f` every time the value of `x` changes. We can precompute once and for all a specialized function `f1(x)`, which when called produces the residual function `g(y)` corresponding to `x`. This is *off-line partial evaluation;* the specialized function `f1(x)` is called a *generating extension*.

The PyPy JIT generation framework is based on off-line partial evaluation. The function called `f(x, y)` above is typically the main loop of some interpreter written in RPython. The size of the interpreter can range from a three-liner used for testing purposes to the whole of PyPy's Python interpreter. In all cases, `x` stands for the input program (the bytecode to interpret) and `y` stands for the input data (like a frame object with the binding of the input arguments and local variables). Our framework is capable of automatically producing the corresponding generating extension `f1(x)`, which takes an input program only and produces a residual function `g(y)`. This `f1(x)` is a compiler[1] for the very same language for which `f(x, y)` is an interpreter.

Off-line partial evaluation is based on *binding-time analysis,* which is the process of determining among the variables used in a function (or a set of functions) which ones are going to be known in advance and which ones are not. In the example of `f(x, y)`, such an analysis would be able to infer that the constantness of the argument `x` implies the constantness of many intermediate values used in the function. The *binding time* of a variable determines how early the value of the variable will be known.

Once binding times have been determined, one possible approach to producing the generating extension itself is by self-applying on-line partial evaluators. This is known as the second Futamura projection [FU]. So far it is

[1] What we get in PyPy is more precisely a *just-in-time compiler:* if promotion is used, compiling ahead of time is not possible.

unclear if this approach can lead to optimal results, or even if it scales well. In PyPy we selected a more direct approach: the generating extension is produced by transformation of the control flow graphs of the interpreter, guided by the binding times. We call this process *timeshifting*.

3 Architecture and Principles

PyPy contains a framework for generating just-in-time compilers using off-line partial evaluation. As such, there are three distinct phases:

- *Translation time:* during the normal translation of an RPython program, say PyPy's Python interpreter, we perform binding-time analysis and off-line specialization ("timeshifting") of the interpreter. This produces a generating extension, which is linked with the rest of the program.
- *Compile time:* during the execution of the program, when a new bytecode is about to be interpreted, the generating extension is invoked instead. As the generating extension is a compiler, all the computations it performs are called compile-time computations. Its sole effect is to produce residual code.
- *Run time:* the normal execution of the program (which includes the time spent running the residual code created by the generating extension).

Translation time is a purely off-line phase; compile time and run time are actually highly interleaved during the execution of the program.

3.1 Binding Time Analysis

At translation time, PyPy performs binding-time analysis of the source RPython program after it has been turned to low-level graphs, i.e. at the level at which operations manipulate pointer-and-structure-like objects.

The binding-time terminology that we are using in PyPy is based on the colors that we use when displaying the control flow graphs:

- *Green* variables contain values that are known at compile-time;
- *Red* variables contain values that are not known until run-time.

The binding-time analyzer of our translation tool-chain is based on the same type inference engine that is used on the source RPython program, the annotator. In this mode, it is called the *hint-annotator*; it operates over input graphs that are already low-level instead of RPython-level, and propagates annotations that do not track types but value dependencies and manually-provided binding time hints.

The normal process of the hint-annotator is to propagate the binding time (i.e. color) of the variables using the following kind of rules:

- For a foldable operation (i.e. one without side effect and which depends only on its argument values), if all arguments are green, then the result can be green too.
- Non-foldable operations always produce a red result.
- At join points, where multiple possible values (depending on control flow) are meeting into a fresh variable, if any incoming value comes from a red variable, the result is red. Otherwise, the color of the result might be green. We do not make it eagerly green, because of the control flow dependency: the residual function is basically a constant-folded copy of the source function, so it might retain some of the same control flow. The value that needs to be stored in the fresh join variable thus depends on which branches are taken in the residual graph.

3.1.1 Hints

Our goal in designing our approach to binding-time analysis was to minimize the number of explicit hints that the user must provide in the source of the RPython program. This minimalism was not pushed to extremes, though, to keep the hint-annotator reasonably simple.

The driving idea was that hints should be need-oriented. Indeed, in a program like an interpreter, there are a small number of places where it would be clearly beneficial for a given value to be known at compile-time, i.e. green: this is where we require the hints to be added.

The hint-annotator assumes that all variables are red by default, and then propagates annotations that record dependency information. When encountering the user-provided hints, the dependency information is used to make some variables green. All hints are in the form of an operation `hint(v1, someflag=True)` which semantically just returns its first argument unmodified.

The crucial need-oriented hint is `v2 = hint(v1, concrete=True)` which should be used in places where the programmer considers the knowledge of the value to be essential. This hint is interpreted by the hint-annotator as a request for both `v1` and `v2` to be green. It has a *global* effect on the binding times: it means that not only `v1` but all the values that `v1` depends on – recursively – are forced to be green. The hint-annotator complains if the dependencies of `v1` include a value that cannot be green, like a value read out of a field of a non-immutable structure.

Such a need-oriented backward propagation has advantages over the commonly used forward propagation, in which a variable is compile-time if and only if all the variables it depends on are also compile-time. A known issue with forward propagation is that it may mark as compile-time either more variables than expected (which leads to over-specialization of the residual code), or less variables than expected (preventing specialization to occur where it would be the most useful). Our need-oriented approach reduces the problem of over-specialization, and it prevents under-specialization: an unsatisfiable `hint(v1, concrete=True)` is reported as an error.

In our context, though, such an error can be corrected. This is done by promoting a well-chosen variable among the ones that `v1` depends on.

Promotion is invoked with the use of a hint as well: `v2 = hint(v1, promote=True)`. This hint is a *local* request for `v2` to be green, without requiring `v1` to be green. Note that this amounts to copying a red value into a green one, which is not possible in classical approaches to partial evaluation. See the Promotion section for a complete discussion of promotion.

For examples and further discussion on how the hints are applied in practice see *Make your own JIT compiler* included as an annex of [D08.1].

3.2 Timeshifting

Once binding times (colors) have been assigned to all variables in a family of control flow graphs, the next step is to mutate the graphs[2] accordingly in order to produce a generating extension. We call this process *timeshifting* because it changes the time at which the graphs are meant to be run, from run-time to compile-time.

Despite the execution time and side-effects shift to produce only residual code, the timeshifted graphs have a shape (flow of control) that is closely related to that of the original graphs. This is because at compile-time the timeshifted graphs go over all the operations that the original graphs would have performed at run-time, following the same control flow; some of these operations and control flow constructs are constant-folded at compile-time, and the rest is turned into equivalent residual code. Another point of view is that as the timeshifted graphs form

[2] One should keep in mind that the program described as the "source RPython program" in this document is typically an interpreter – the canonical example is that it is the whole PyPy Standard Interpreter. This program is meant to execute at run-time, and directly compute the intended result and side-effects. The translation process transforms it into a forest of flow graphs. These are the flow graphs that timeshifting processes (and not the application-level program, which typically cannot be expressed as low-level flow graphs).

a generating extension, they perform the equivalent of an abstract interpretation of the original graphs over a domain containing compile-time values and run-time value locations.

The rest of this section describes this timeshifting process in more detail.

3.2.1 Red and Green Operations

The basic idea of timeshifting is to transform operations in a way that depends on the color of their operands and result. Variables themselves need to be represented based on their color:

- The red (run-time) variables have abstract values at compile-time; no actual value is available for them during compile-time. For them we use a boxed representation that can carry either a run-time storage location (a stack frame position or a register name) or an immediate constant (for when the value is, after all, known at compile-time).
- On the other hand, the green variables are the ones that can carry their value already at compile-time, so they are left untouched during timeshifting.

The operations of the original graphs are then transformed as follows:

- If an operation has no side effect nor any other run-time dependency, and if it only involves green operands, then it can stay unmodified in the graph. In this case, the operation that was run-time in the original graph becomes a compile-time operation, and it will never be generated in the residual code. (This is the case that makes the whole approach worthwhile: some operations become purely compile-time.)
- In all other cases, the operation might have to be generated in the residual code. In the timeshifted graph it is replaced by a call to a helper which will generate a residual operation manipulating the input run-time values and return a new boxed representation for the run-time result location.

These helpers will constant-fold the operation if the inputs are immediate constants and if the operation has no side-effects. Immediate constants can occur even though the corresponding variable in the graph was red: a variable can be dynamically found to contain a compile-time constant at a particular point in (compile)-time, independently of the hint-annotator proving that it is always the case. In Partial Evaluation terminology, the timeshifted graphs are performing some *on-line* partial evaluation in addition to the off-line job enabled by the hint-annotator.

3.2.2 Merges and Splits

The timeshifted code carries around an object that stores the compilation-time state – mostly the current bindings of the variables. This state is used to shape the control flow of the generated residual code, as follows.

After a *split,* i.e. after a conditional branch that could not be folded at compile-time, the compilation state is duplicated and both branches are compiled independently. Conversely, after a *merge point,* i.e. when two control flow paths meet each other, we try to join the two paths in the residual code. This part is more difficult because the two paths may need to be compiled with different variable bindings – e.g. different variables may be known to take different compile-time constant values in the two branches. The two paths can either be kept separate or merged; in the latter case, the merged compilation-time state needs to be a generalization (*widening*) of the two already-seen states. Deciding when to do each is a classical problem of partial evaluation, as merging too eagerly may loose important precision and not merging eagerly enough may create too many redundant residual code paths (to the point of preventing termination of the compiler).

So far, we did not investigate this problem in detail. We settled for a simple widening heuristic: two different compile-time constants merge as a run-time value, but we try to preserve the richer models of run-time information that are enabled by the techniques described in the sequel (promotion, virtual structures...). This heuristic seems to work for PyPy to some extent.

3.2.3 Calls and inlining

For calls timeshifting can either produce code to generate a residual call operation or recursively invoke the timeshifted version of the callee. The residual operations generated by the timeshifted callee will grow the compile-time produced residual function; this effectively amounts to the compile-time inlining of the original callee into its caller. This is the default behaviour for calls within the user-controlled subset of original graphs of the interpreter that are timeshifted. Inlining only stops at re-entrant calls to the interpreter main loop; the net result is that at the level of the interpreted language, each function (or method) gets compiled into a single piece of residual code.

3.3 Promotion

In the sequel, we describe in more details one of the main new techniques introduced in our approach, which we call *promotion*. In short, it allows an arbitrary run-time value to be turned into a compile-time value at any point in time. Each promotion point is explicitly defined with a hint that must be put in the source code of the interpreter.

From a partial evaluation point of view, promotion is the converse of the operation generally known as "lift". Lifting a value means copying a variable whose binding time is compile-time into a variable whose binding time is run-time – it corresponds to the compiler "forgetting" a particular value that it knew about. By contrast, promotion is a way for the compiler to gain *more* information about the run-time execution of a program. Clearly, this requires fine-grained feedback from run-time to compile-time, thus a dynamic setting.

Promotion requires interleaving compile-time and run-time phases, otherwise the compiler can only use information that is known ahead of time. It is impossible in the "classical" approaches to partial evaluation, in which the compiler always runs fully ahead of execution This is a problem in many large use cases. For example, in an interpreter for a dynamic language, there is mostly no information that can be clearly and statically used by the compiler before any code has run.

A more theoretical way to see the issue is to consider that the possible binding time for each variable in the interpreter is constrained by the binding time of the other variables it depends on. For some kind of interpreters this set of constraints may have no interesting global solution – if most variables can ultimately depend on a value, even in just one corner case, which cannot be compile-time, then in any solution most variables will be run-time. In the presence of promotion, though, these constraints can be occasionally violated: corner cases do not necessarily have to influence the common case, and local solutions can be patched together.

A very different point of view on promotion is as a generalization of techniques that already exist in dynamic compilers as found in modern object-oriented language virtual machines. In this context feedback techniques are crucial for good results. The main goal is to optimize and reduce the overhead of dynamic dispatching and indirect invocation. This is achieved with variations on the technique of polymorphic inline caches [PIC]: the dynamic lookups are cached and the corresponding generated machine code contains chains of compare-and-jump instructions which are modified at run-time. These techniques also allow the gathering of information to direct inlining for even better optimization results.

In the presence of promotion, dispatch optimization can usually be reframed as a partial evaluation task. Indeed, if the type of the object being dispatched to is known at compile-time, the lookup can be folded, and only a (possibly inlined) direct call remains in the generated code. In the case where the type of the object is not known at compile-time, it can first be read at run-time out of the object and promoted to compile-time. As we will see in the sequel, this produces very similar machine code.[3]

The essential advantage is that it is no longer tied to the details of the dispatch semantics of the language being interpreted, but applies in more general situations. Promotion is thus the central enabling primitive to make timeshifting a practical approach to language independent dynamic compiler generation.

[3] This can also be seen as a generalization of a partial evaluation transformation called "The Trick" (see e.g. [PE]), which again produces similar code but which is only applicable for finite sets of values.

3.3.1 Promotion in practice

The implementation of promotion requires a tight coupling between compile-time and run-time: a *callback*, put in the generated code, which can invoke the compiler again. When the callback is actually reached at run-time, and only then, the compiler resumes and uses the knowledge of the actual run-time value to generate more code.

The new generated code is potentially different for each run-time value seen. This implies that the generated code needs to contain some sort of updatable switch, which can pick the right code path based on the run-time value.

While this describes the general idea, the details are open to slight variations. Let us show more precisely the way the JIT compilers produced by PyPy 1.0 work. Our first example is purely artificial:

```
...
b = a / 10
c = hint(b, promote=True)
d = c + 5
print d
...
```

In this example, `a` and `b` are run-time variables and `c` and `d` are compile-time variables; `b` is copied into `c` via a promotion. The division is a run-time operation while the addition is a compile-time operation.

The compiler derived from an interpreter containing the above code generates the following machine code (in pseudo-assembler notation), assuming that `a` comes from register `r1`:

```
...
   r2 = div r1, 10
Label1:
   jump Label2
   <some reserved space here>

Label2:
   call continue_compilation(r2, <state data pointer>)
   jump Label1
```

The first time this machine code runs, the `continue_compilation()` function resumes the compiler. The two arguments to the function are the actual run-time value from the register `r2`, which the compiler will now consider as a compile-time constant, and an immediate pointer to data that was generated along with the above code snippet and which contains enough information for the compiler to know where and with which state it should resume.

Assuming that the first run-time value taken by `r1` is, say, 42, then the compiler will see `r2 == 4` and update the above machine code as follows:

```
...
   r2 = div r1, 10
Label1:
   compare r2, 4              # patched
   jump-if-equal Label3       # patched
   jump Label2                # patched
   <less reserved space left>

Label2:
```

```
    call continue_compilation(r2, <state data pointer>)
    jump Label1

Label3:                     # new code
    call print(9)           # new code
    ...
```

Notice how the addition is constant-folded by the compiler. (Of course, in real examples, different promoted values typically make the compiler constant-fold complex code path choices in different ways, and not just simple operations.) Note also how the code following `Label1` is an updatable switch which plays the role of a polymorphic inline cache. The "polymorphic" terminology does not apply in our context, though, as the switch does not necessarily have to be on the type of an object.

After the update, the original call to `continue_compilation()` returns and execution loops back to the now-patched switch at `Label1`. This run and all following runs in which `r1` is between 40 and 49 will thus directly go to `Label3`. Obviously, if other values show up, `continue_compilation()` will be invoked again, so new code will be generated and the code at `Label1` further patched to check for more cases.

If, over the course of the execution of a program, too many cases are seen, the reserved space after `Label1` will eventually run out. Currently, we simply reserve more space elsewhere and patch the final jump accordingly. There could be better strategies which which we did not implement so far, such as discarding old code and reusing their slots in the switch, or sometimes giving up entirely and compiling a general version of the code in which the value remains run-time.

3.3.2 Implementation notes

The *state data pointer* in the example above contains a snapshot of the state of the compiler when it reached the promotion point. Its memory impact is potentially large – a complete continuation for each generated switch, which can never be reclaimed because new run-time values may always show up later during the execution of the program.

To reduce the problem we compress the state into a so-called *path*. The full state is only stored at a few specific points[4]. The compiler records a trace of the multiple paths it followed from the last full snapshot in a lightweight tree structure. The *state data pointer* is then only a pointer to a node in the tree; the branch from that node to the root describes a path that let the compiler quickly *replay* its actions (without generating code again) from the latest full snapshot to rebuild its internal state and get back to the original promotion point.

For example, if the interpreter source code contains promotions inside a run-time condition:

```
if condition:
    ...
    hint(x, promote=True)
    ...
else:
    ...
    hint(y, promote=True)
    ...
```

then the tree will contain three nodes: a root node storing the snapshot, a child with a "True case" marker, and another child with a "False case" marker. Each promotion point generates a switch and a call to `continue_compilation()` pointing to the appropriate child node. The compiler can re-reach the correct promotion point by following the markers on the branch from the root to the child.

[4] More precisely, at merge points that the user needs to mark as "global". The control flow join point corresponding to the looping of the interpreter main loop is a typical place to put such a global merge point.

3.4 Virtual structures

Interpreters for dynamic languages typically allocate a lot of small objects, for example due to boxing. For this reason, we implemented a way for the compiler to generate residual memory allocations as lazily as possible. The idea is to try to keep new run-time structures "exploded": instead of a single run-time pointer to a heap-allocated data structure, the structure is "virtualized" as a set of fresh variables, one per field. In the compiler, the variable that would normally contain the pointer to the structure gets instead a content that is neither a run-time value nor a compile-time constant, but a special *virtual structure* – a compile-time data structure that recursively contains new variables, each of which can again store a run-time, a compile-time, or a virtual structure value.

This approach is based on the fact that the "run-time values" carried around by the compiler really represent run-time locations – the name of a CPU register or a position in the machine stack frame. This is the case for both regular variables and the fields of virtual structures. It means that the compilation of a `getfield` or `setfield` operation performed on a virtual structure simply loads or stores such a location reference into the virtual structure; the actual value is not copied around at run-time.

It is not always possible to keep structures virtual. The main situation in which it needs to be "forced" (i.e. actually allocated at run-time) is when the pointer escapes to some non-virtual location like a field of a real heap structure.

Virtual structures still avoid the run-time allocation of most short-lived objects, even in non-trivial situations. The following example shows a typical case. Consider the Python expression `a+b+c`. Assume that `a` contains an integer. The PyPy Python interpreter implements application-level integers as boxes – instances of a `W_IntObject` class with a single `intval` field. Here is the addition of two integers:

```
def add(w1, w2):            # w1, w2 are W_IntObject instances
    value1 = w1.intval
    value2 = w2.intval
    result = value1 + value2
    return W_IntObject(result)
```

When interpreting the bytecode for `a+b+c`, two calls to `add()` are issued; the intermediate `W_IntObject` instance is built by the first call and thrown away after the second call. By contrast, when the interpreter is turned into a compiler, the construction of the `W_IntObject` object leads to a virtual structure whose `intval` field directly references the register in which the run-time addition put its result. This location is read out of the virtual structure at the beginning of the second `add()`, and the second run-time addition directly operates on the same register.

An interesting effect of virtual structures is that they play nicely with promotion. Indeed, before the interpreter can call the proper `add()` function for integers, it must first determine that the two arguments are indeed integer objects. In the corresponding dispatch logic, we have added two hints to promote the type of each of the two arguments. This produces a compiler that has the following behavior: in the general case, the expression `a+b` will generate two consecutive run-time switches followed by the residual code of the proper version of `add()`. However, in `a+b+c`, the virtual structure representing the intermediate value will contain a compile-time constant as type. Promoting a compile-time constant is trivial – no run-time code is generated. The whole expression `a+b+c` thus only requires three switches instead of four. It is easy to see that even more switches can be skipped in larger examples; typically, in a tight loop manipulating only integers, all objects are virtual structures for the compiler and the residual code is theoretically optimal – all type propagation and boxing/unboxing occurs at compile-time.

3.5 Virtualizable structures

In the PyPy interpreter there are cases where structures cannot be virtual – because they escape, or are allocated outside the JIT-generated code – but where we would still like to keep the "exploding" effect and carry the fields of the structure as local variables in the generated code.

It is likely that the same problem occurs more generally in many interpreters: the typical example is that of frame objects, which stores among other things the value of the local variables of each function invocation. Ideally, the effect we would like to achieve is to keep the frame object as a purely virtual structure, and the same for the array or dictionary implementing the bindings of the locals. Then each local variable of the interpreted language can be represented as a separate run-time value in the generated code, or be itself further virtualized (e.g. as a virtual `W_IntObject` structure as seen above).

The issue is that the frame object is sometimes built in advance by non-JIT-generated code; even when it is not, it immediately escapes into the global list of frames that is used to support the frame stack introspection primitives that Python exposes. In other words, the frame object cannot be purely virtual because a pointer to it must be stored into a global data structure (even though in practice most of frame objects are deallocated without ever having been introspected).

To solve this problem, we introduced *virtualizable structures,* a mix between regular run-time structures and virtual structures. A virtualizable structure is a structure that exists at run-time in the heap, but that is simultaneously treated as virtual by the compiler. Accesses to the structure from the code generated by the JIT are virtualized away, i.e. don't involve run-time copying. The trade-off is that in order to keep both views synchronized, accesses to the run-time structure from regular code not produced by the JIT needs to perform an extra check.

Because of this trade-off, a hint needs to be inserted manually to mark the classes whose instances should be implemented in this way – the class of frame objects, in the case of PyPy. The hint is used by the translation toolchain to add a hidden field to all frame objects, and to translate all accesses to the object fields into low-level code that first checks the hidden field. This is the only case so far in which the presence of the JIT compiler imposes a global change to the rest of the program during translation[5].

The hidden field is set when the frame structure enters JIT-generated code, and cleared when it leaves. When a recursive call to non-JIT-generated code finds a structure with the field set, it invokes a JIT-generated callback to perform the reading or updating of the field from the point of view of its virtual structure representation. The actual fields in the heap structure are not used during this time.

The effect that can be obtained in this way is that although frame objects are still allocated in the heap, most of them will always remain essentially empty. A pointer to these empty frames is pushed into and popped off the global frame list, allowing the introspection mechanisms to still work perfectly.

3.6 Other implementation details

We quickly mention below a few other features and implementation details of the implementation of the JIT generation framework. More information can be found in the on-line documentation.

- There are more user-specified hints available, like *deep-freezing,* which marks an object as immutable in order to allow accesses to its content to be constant-folded at compile-time.
- The compiler representation of a run-time value for a non-virtual structure may additionally remember that some fields are actually compile-time constants. This occurs for example when a field is read from the structure at run-time and then promoted to compile-time.
- In addition to virtual structures, lists and dictionaries can also be virtual.
- Exception handling is achieved by inserting explicit operations into the graphs before they are timeshifted. Most of these run-time exception manipulations are then virtualized away, by treating the exception state as virtual.

[5] This is not a problem per se, as it is anyway just a small extension to the translation framework, but it imposes a performance overhead to all code manipulating frame objects. To mitigate this, we added a way to declare during RPython type inference that the indirection check is not needed in some parts of the code where we know that the frame object cannot have a virtual counterpart.

- Timeshifting is performed in two phases: a first step transforms the graphs by updating their control flow and inserting pseudo-operations to drive the compiler; a second step (based on the RTyper [D05.1]) replaces all necessary operations by calls to support code.
- The support code implements the generic behaviour of the compiler, e.g. the merge logic. It is about 3500 lines of RPython code. The rest of the hint-annotator and timeshifter is about 3800 lines of Python code.
- The machine code backends (two so far, Intel IA32 and PowerPC) are about 3500 further lines of RPython code each. There is a well-defined interface between the JIT compiler support code and the backends, making writing new backends relatively easy. The unusual part of the interface is the support for the run-time updatable switches.

3.7 Open issues

Here are what we think are the most important points that will need attention in order to make the approach more robust:

- The timeshifted graphs currently compile many branches eagerly. This can easily result in residual code explosion. Depending on the source interpreter this can also result in non-termination issues, where compilation never completes. The opposite extreme would be to always compile branches lazily, when they are about to be executed, as Psyco does. While this neatly sidesteps termination issues, the best solution is probably something in between these extremes.
- As described in the Promotion section, we need fall-back solutions for when the number of promoted run-time values seen at a particular point becomes too large.
- We need more flexible control about what to inline or not to inline in the residual code.
- The widening heuristics for merging needs to be refined.
- The JIT generation framework needs to be made aware of some other translation-time aspects [D05.4] [D07.1] in order to produce the correct residual code (e.g. code calling the correct Garbage Collection routines or supporting Stackless-style stack unwinding).
- We did not work yet on profile-directed identification of program hot spots. Currently, the interpreter must decide when to invoke the JIT or not (which can itself be based on explicit requests from the interpreted program).
- The machine code backends can be improved.

The latter point opens an interesting future research direction: can we layer our kind of JIT compiler on top of a virtual machine that already contains a lower-level JIT compiler? In other words, can we delegate the difficult questions of machine code generation to a lower independent layer, e.g. inlining, re-optimization of frequently executed code, etc.? What changes would be required to an existing virtual machine, e.g. a Java Virtual Machine, to support this?

4 Results

The following test function is an example of purely arithmetic code written in Python, which the PyPy JIT can run extremely fast:

```
def f1(n):
    "Arbitrary test function."
    i = 0
    x = 1
    while i<n:
        j = 0
        while j<=i:
            j = j + 1
            x = x + (i&j)
        i = i + 1
    return x
```

We measured the time required to compute `f1(2117)` on the following interpreters:

- Python 2.4.4, the standard CPython implementation.
- A version of pypy-c including a generated JIT compiled.
- gcc 4.1.1 compiling the above function rewritten in C (which, unlike the other two, does not do any overflow checking on the arithmetic operations).

The relative results have been found to vary by 25% depending on the machine. On our reference benchmark machine, a 4-cores Intel(R) Xeon(TM) CPU 3.20GHz with 5GB of RAM, we obtained the following results (the numbers in parenthesis are the slow-down ratio relative to the unoptimized gcc compilation):

Interpreter	Seconds per call
Python 2.4.4	0.82 (132x)
Python 2.4.4 with Psyco 1.5.2	0.0062 (1.00x)
pypy-c with the JIT turned off	1.77 (285x)
pypy-c with the JIT turned on	0.0091 (1.47x)
gcc	0.0062 (1x)
gcc -O2	0.0022 (0.35x)

This table shows that the PyPy JIT is able to generate residual code that runs within the same order of magnitude as an unoptimizing gcc. It shows that all the abstraction overhead has been correctly removed from the residual code; the remaining slow-downs are only due to a suboptimal low-level machine code generation backend. We have thus reached our goal of automatically generating a JIT whose performance is similar to the hand-written Psyco without having its limitations[6].

In particular, the ratio of 1.47x between the unoptimizing gcc and the PyPy JIT matches the target of 1.5x that we set ourselves as our goal within the duration of the EU project. We should also mention that on an Intel-based Mac OS/X machine we have measured this ratio to be as low as 1.15x.

5 Conclusion

Producing the results described in the previous section requires the generated compiler to completely cut the overhead and fold at compile-time some rather involved lookup algorithms like Python's binary operation dis-

[6] As mentioned above, Psyco gives up compiling Python functions if they use constructs it does not support, and is not 100% compatible with introspection of frames. By construction the PyPy JIT does not have these limitations. The PyPy JIT is also easier to retarget, and already supports more architectures than Psyco does, namely the Intel Mac OS/X and the PowerPC Mac OS/X.

patch. Promotion proved itself to be sufficiently powerful to achieve this. Other features we introduced allowed to preserve information about intermediate value types, to avoid their boxing and to propagate them in the CPU stack and registers.

Some slight reorganisation of the interpreter main loop without semantics influence, marking the frames as virtualizable, and adding hints at a few crucial points was all that was necessary for our Python interpreter.

We think that our results make viable an approach to implement dynamic languages that needs only a straightforward bytecode interpreter to be written. Dynamic compilers would be generated automatically guided by the placement of hints.

These implementations should stay flexible and evolvable. Dynamic compiler would be robust against language changes up to the need to maintain and possibly change the hints.

We consider this as a major breakthrough in term of the possibilities it opens for language design and implementation; it was one of the main goals of the research program within the PyPy project.

6 Glossary of Abbreviations

The following abbreviations may be used within this document:

6.1 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.

6.2 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

[D05.1] *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

[D05.4] *Encapsulating Low-Level Aspects*, PyPy EU-Report, 2005

[D07.1] *Support for Massive Parallelism, Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2006

[D08.1] *Release a JIT Compiler for PyPy Including Processor Backends for Intel and PowerPC*, PyPy EU-Report, 2007

[FU] *Partial evaluation of compuation process – an approach to a compiler-compiler*, Yoshihito Futamura, Higher-Order and Symbolic Computation, 12(4):363-397, 1999. Reprinted from Systems Computers Controls 2(5), 1971

[PE] *Partial evaluation and automatic program generation*, Neil D. Jones, Carsten K. Gomard, Peter Sestoft, Prentice-Hall, Inc., Upper Saddle River, NJ, 1993

[REJIT] *Retargeting JIT Compilers by using C-Compiler Generated Executable Code*, M. Anton Ertl, David Gregg, Proc. of the 13th Intl. Conf. on Parallel Architectures and Compilation Techniques, 2004.

[PIC] *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*, U. Hölzle, C. Chambers, D. Ungar, ECOOP'91 Conference Proceedings, Geneva, 1991.

[PSYCO] *Representation-based just-in-time specialization and the psyco prototype for python*, Armin Rigo, in PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 15-26, ACM Press, 2004

[VMCDLS] *PyPy's approach to virtual machine construction*, Armin Rigo, Samuele Pedroni, in OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications, pp. 944-953, ACM Press, 2006