
# 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**

# Publish a Report About the New JIT Architecture, its Performance and how its Techniques Compare to the ones Found in Other Languages, and how our JIT architecture Applies to Other Languages Than Python

**Due date of deliverable: March 2007**

**Actual Submission date: XXX insert submission date here l**

**Start date of Project: 1st December 2005** **Duration: 2 years**

**Lead Contractor of this WP: Strakt**

**Authors: Arimin Rigo, Samuele Pedroni**

**Revision: Draft**

**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 |
| 2006-01-28 | S. Pedroni | Publishing of intermediate version on the web |

## Abstract

PyPy translation tool-chain - from the interpreter written in Python to low-level platform generated VMs - is now able to extended 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 general primitive called "promotion" that "promotes" run-time values to compile-time is used to that effect.

# 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 the higher the dynamism of the language, the more effort is required to effectively gather run-time information and exploit it for compilation. Additionally, 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.

Given this context and the wish for performance, 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.

The transformation itself is a pragmatic application of partial evaluation techniques inspired by Psyco, a hand-written runtime specializer that can be used with CPython.

Using such techniques effectively to automatically generate a dynamic compiler - in particular for a language as large as Python - represents a novel result.

# 2 Introduction

## 2.1 Purpose of this Document

This document describes the architecture and some implementation details of how our translation tool-chain can derive a dynamic compiler from an interpreter. (XXX comparison with classic dynamic compiler construction).

## 2.2 Scope of this Document

The dynamic compiler generation described here, as translation aspect, is what is currently implemented (XXX and will be part of release 1.0 of PyPy). The overall details of the translation process with the notion of translation aspect are not presented here, and neither are the other implemented translation aspects.

## 2.3 Related Documents

- (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

## 3 Generating Dynamic Compilers

One the of the central goals of the PyPy project is to automatically produce a Just in Time Compiler from the interpreter, with as little as possible intervention on the interpreter codebase itself. The Just in Time Compiler should be another aspect as much as possible 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. (XXX references)

The forest of flow graphs that the translation process generates and transforms constitutes a reasonable base for the necessary analyses. That's a further reason why having an high-level runnable and analysable description of the language was always a central tenet of the project.

Transforming an interpreter into a compiler involves constructing a so called generating extension, which takes input programs to the interpreter, and produces what would be the output of partial evaluating the interpreter with the input data left variable and the input program being fixed. The generating extension is essentially capable of compiling the input programs.

Generating extensions can be produced by self-applying partial evaluators, but this approach may lead to not optimal results or be not scalable (XXX expand this argument)

For PyPy, our approach aims at producing the generating extension more directly from the analysed interpreter in the form of a forest of flow graphs. We call such process *timeshifting*.

To be able to achieve this, gathering binding time information is crucial, this means for an input program distinguishing values in the data-flow that are compile-time bound and immutable at run-time, versus respectively runtime values.

Currently we base the binding time computation on propagating the information based on a few hint inserted in the interpreter. Propagation is implemented by reusing our annotation/type inference framework.

The code produced by a generating extension for an input program may not be good, especially for a dynamic language, because essentially the input program doesn't contain enough information to generate good code. What is really desired is not a generating extension doing static compilation, but one capable of dynamic compilation, exploiting run-time information in its result. Compilation should be able to suspend and resume letting the produced code run to collect run-time information (for example language-level types) to be used to generate code optimised for the effective run-time behaviour of the program.

Inspired by Psyco, which in some sense is such a specialising generating extension for Python, but hand-written, we added support for so-called *promotion* to our framework for producing generating extensions.

Simply put, promotion on a value stops compilation when encountered, when the same point is reached at run-time compilation is restarted and the current run-time value is used and

propagated as a compile-time value. Concretely the promotion expands into a switch to choose based on the run-time value one of the possible specialised code paths, plus a default case to compile further specialised paths. This can be also thought as a generalisation of polymorphic inline caches (XXX reference).

## 3.1 Terminology

Partial evaluation is the process of evaluating a function, say `f(x, y)`, with only partial information about the value 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 need 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 off-line a specialized function `f1(x)`, which when called produces a residual function `g(y)`.

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 above example, 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.

The PyPy JIT is generated using off-line partial evaluation. As such, there are three distinct phases:

- *Translation time*: during the normal translation of an RPython program like PyPy, we perform binding-time analysis and off-line specialization. This produces a new set of functions (`f1(x)` in our running example) which are linked with the rest of the program, as described in Timeshifting.
- *Compile time*: during the execution of the program, when a new value for `x` is found, `f1(x)` is invoked. All the computations performed by `f1(x)` are called compile-time computations. This is justified by the fact that `f1(x)` is in some sense a compiler, whose sole effect is to produce residual code.

- *Run time*: the normal execution of the program.

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 - e.g. `x`.
- *Red* variables contain values that are not known until run-time - e.g. `y`.

# 4 Binding-time analysis

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-structures-like objects.

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.

## 4.1 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. In a program like an interpreter, there are a few clear places where it would be 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 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.

The hint-annotator assumes that all variables are red by default (with the exception of constants, which are always green). It then propagates annotations that record dependency information. When encountering the user-provided hints, the dependency information is used to make some variables green. (Technically, the color itself is not part of the annotations propagated by the annotator.) All hints are in the form of an operation `hint(v1, someflag=True)` which semantically just returns its first argument unmodified. The three kinds of hints that are useful in practice are:

**`v2 = hint(v1, concrete=True)`** This is interpreted by the hint-annotator as a request for both `v1` and `v2` to be green. It is used in places where the programmer considers the knowledge of the value to be essential. This hint 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. Typically, it can only be applied on values that directly depend on some input arguments, making these input arguments 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 out of a non-immutable structure.

The color of the result `v2` is green as well. Unlike `v1`, all further operations involving `v2` are checked to not meet any red variable (i.e. `v2` color is eagerly and recursively propagated, while `v1` is only green and may be involved in further operations that will produce red results).

**`v2 = hint(v1, promote=True)`** This hint is a *local* request for `v2` to be green. Unlike the previous hint, this one has no effect on the color of `v1` (which is typically red - the hint has no effect otherwise).

Note that in classical approaches to partial evaluation, it is not possible to copy a red value into a green one. The implementation of such an operation ("promotion") is only possible in a "just-in-time" approach that only Psyco implemented so far (to the best of our knowledge). Our hint is a direct generalization of the latter.

**`v2 = hint(v1, variable=True)`** Force `v2` to be red, even if `v1` is green.

A program using promotion also needs to contain a `global_merge_point` hint; this has no effect on the hint-annotator and is described in the section about promotion.

## 4.2 Example

Let's consider a very small interpreter-like example:

```
def ll_plus_minus(s, x, y):
    acc = x
    pc = 0
    while pc < len(s):
        op = s[pc]
        op = hint(op, concrete=True)
        if op == '+':
            acc += y
        elif op == '-':
            acc -= y
        pc += 1
    return acc
```

`s` here is an input program, simply a string of `'+'` or `'-'`, `x` and `y` are integer input arguments.

The annotation of `op = hint(op, concrete=True)` will follow the dependencies of the argument `op`, which is the result of `op = s[pc]`, so both `s` and `pc` will be marked green. `x`, `y` and `acc` will stay red.

The result `op` compared to the possible "instructions" will also be green. Because `s` is green also `len(s)` will be.

As we will see later, the timeshifter can use this information to generate code that unfolds the loop (because the `while` condition is green) and folds instruction dispatching (because the `if` conditions are green) at compile-time. In other words, the only operations left involving red variables are `acc += y` and `acc -= y`, so only these operations will eventually appear in the residual graph.

## 4.3 Calls

The `concrete` hint requires precise tracking of dependencies across calls. More specifically, unlike the regular type-inferencing annotator, the hint-annotator does not simply propagate this information through called functions, but keeps all dependency information local to a function. The problem is best shown by an example:

```
x = ...
y = ...
z = f(x, y)
```

In this example, assuming that `f(x, y)` has no side effects, then we want the dependency set of `z` to be the union of the dependencies of `x` and `y` (if both contribute to the result). But if another call site calls `f(x2, y2)` then precision would be lost - the annotations that would propagate within `f` would be marked as depending on all of `x`, `x2`, `y` and `y2`, and then so would `z`.

To fix this problem, we only propagate dependencies that are local to a function. Around a call like `z = f(x, y)`, we check in the annotations of the function `f` on which input arguments the result is marked as depending on, and we reproduce the same dependency relationship locally between `x` and `y` and `z`.

There are other aspects of the information carried by annotations that are used in the "usual" annotator way, i.e. directly propagated between caller and callee and back. If the result `z` of a call is forced to be green by a `concrete` hint, then this forcing effect is propagated inside the callee (more precisely, in a fresh copy of the callee, so that forced and non-forced call sites can respectively call a mostly-green and a mostly-red version of the function; this is done with the technique of specialization of functions that is already available in the normal annotator).

Also, if all call sites provide a green value for an argument, then the callee's corresponding input argument can safely be green. (This can only be determined at the end of the process, though, after all call sites are known and stable enough; we use another fixpoint loop for this.)

## 4.4 Deep freezing

Among the low-level operations, the one that reads data from a structure in memory (`getfield`) requires special care. In a first approximation it can only return a red value, because the reading from the structure cannot be performed at compile-time - the structure may be mutated between compile-time and the point at run-time where the read was supposed to occur in the original code.

This is a problem in most real-life examples. Unlike our example above, input arguments to such functions are typically not just strings and integers but complex data structures. To produce sensible results, it is necessary to assume that some of these data structures will not be

mutated after the compile-time process used their content. For example, PyPy's own interpreter works with an instance of a PyCode class containing not only a string representing the bytecode, but various simple Python objects (tuple of names, ...). None of this data can be modified after its construction, though.

To express this, we use a hint `v2 = hint(v1, deepfreeze=True)`, where `v1` is a pointer (in the low-level graph - it typically comes from a regular RPython object reference in the RPython source). The hint-annotation will then propagate a "deepfrozen" flag on the annotation attached to `v2`. If `v2` is green, a `getfield(v2, "name")` operation then also returns a green. The flag is also relevant if `v2` is red, as we will see in Red and Green Operations below.

As the name suggests, the result of a `getfield` is itself "deepfrozen" if the structure was. We decided to implement recursive freezing and not one-level-only freezing, as the latter seems more fragile with respect to changes both in the RPython source and in the way the source maps to low-level graphs; but variants could easily be implemented if needed.

### 4.5 Blue containers

Undocumented yet. Not implemented in the timeshifter, so not used so far.

## 5 Timeshifting: transforming interpreter into compilers

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 accordingly. This process is called *timeshifting*, because it changes the time at which the graphs are meant to be run.

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 cannot be expressed as low-level flow graphs anyway).

After being timeshifted, the graphs of our interpreter become logically very different: they are no longer intended to be executed at run-time, but just ahead of it (what we call "compile-time"). Their only side effects is to produce residual code. The residual code itself runs "at run-time".

Despite the conceptual difference, though, the form (flow of control) of timeshifted graphs is related and close to the original graphs. The rest of this section describes this timeshifting process in more detail.

### 5.1 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 green variables are the compile-time variables. Their values are meant to be available during compile-time already. A variable that used to contain e.g. an integer at run-time in the original graph, and which the hint-annotator found to be green, is turned

into a variable that contains an integer again, but now at compile-time. In other words, timeshifting leaves green variables untouched - they are identical in the original and in the timeshifted graph.

- The red (run-time) variables on the other hand cannot stay unmodified in the timeshifted graph, because no actual value is available for them during compile-time. They are replaced by the notion of "red box": a red box is a small object that describes where, in the residual code, the value will be stored at run-time. Multiple red variables are turned into multiple red boxes, which are used to distinguish the residual storage location for each of them.

The basic feature of each red box is to provide a field `genvar`, which is a backend-specific object that represents a machine code operand - either a value location (e.g. the register where it is stored) or a constant (an immediate). Constants are used for red boxes whose value is, after all, known at compile-time. This can occur even though the corresponding variable in the graph was red; it is the case when the hint-annotator cannot tell statically that a given variable will always contain a compile-time constant, even though it might dynamically be found to contain one at a particular point in (compile-)time. 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.

In addition to the variables, all operations of the original graphs need to be checked and accordingly transformed:

- If an operation has no side effect or 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. There is one such helper per operation, which takes red boxes as its arguments, and asks the backend to generate the corresponding residual operation on the `genvar` of the red boxes. The backend answers by giving a new `genvar`, which is the location of the result, and the helper puts it into a new red box, which is returned to the timeshifted graph as the result of the call.

Additionally, the helper for some operations checks if all arguments are red boxes containing constants, and if so just returns a red box containing the result without producing any residual code. A particular case to note is that of the `getfield` operation: when applied on a constant red box, the field can sometimes be directly read out of the structure at compile-time - this decision is based on the static Deep freezing analysis performed by the hint-annotator.

## 5.2 The Example again

If we timeshift the ll_plus_minus example, given the described binding time assignments, the only red operations generating code in the output residual code are `acc += x`, `acc -= x`. Timeshifting substitute them with helpers that do emit such operations.

All the rest involves green variables and is left unmutated by timeshifting. The corresponding computations, the loop over the "instructions" and dispaching on them would happen at compile-time.

The timeshifted version of ll_plus_minus for the input "+-+" would produce something like (in pseudo notation):

```
residual_plus_minus(x,y):
    v0 = int_add(x , y)
    v1 = int_sub(v0, y)
    v2 = int_add(v1, y)
    return v2
```

## 5.3 Support code

To implement red boxes, we have written a family of support classes in pypy/jit/timeshifter/rvalue.py: IntRedBox, DoubleRedBox and PtrRedBox. They are used, respectively, for integral values of any size, for floating-point values, and for pointers and addresses. As we will see below, these classes support a number of additional features, particularly PtrRedBox.

These classes are regular RPython classes that are translated to low-level and linked with the rest of the program. The same holds for the helper functions that inspect red boxes and generate residual operations. We make extensive use of this technique (described in more detail in VMC): all the complex support code that is needed for the timeshifted graphs is implemented as regular RPython code and linked with the rest of the program, by inserting calls to these helpers from the timeshifted graphs, and also (as in the red box case) by putting variables in the graphs whose types are (the low-level representation of) instances of helper RPython classes.

The amount of support code is quite extensive, as it includes the whole machine code back-end as well: when helpers need to produce residual code, they do so through a well-defined API on classes that are provided by one of our JIT backends. (This will be described in more details in the Backends section.) All this support code can be arbitrarily complex RPython code, which allows us to experiment quite freely.

## 5.4 Two-Phases Transformation

Concretely, the process of timeshifting replaces some variables with fresh variables of new types, and replaces some operations with calls to helper functions. This is very similar to what the RPython Typer does during a normal translation. The latter replaces RPython-level variables and operations with low-level variables and operations and calls to helpers; it is guided by the annotations produced by the regular type inference. The former does the same, except that the input is already low-level operations, and the annotations are colors, produced by the hint-annotator.

In light of this close analogy, we have implemented the timeshifter based on the RTyper. This gives us direct benefits, like automatic conversions for operation arguments or along links. For example, if an operation takes two arguments, but one of them is red and the other green, then the whole operation must classify as "red" and be replaced by a call to the corresponding helper. This helper expects two red boxes as arguments, though. The red argument is a red box in the timeshifted graph, but not the green one. Reusing the RTyper logic, the necessary conversion (creating a red box and putting the green value into it as an immediate) is inserted automatically in cases like this one.

Unlike the regular RTyper, though, the timeshifter is faced with an additional difficulty that will only become apparent in the sequel: in addition to the local replacement of variables and operations, it also needs to modify the control flow of the graph.

The approach that we have taken is in two phases:

- The graphs are first transformed "manually". This phase modifies the control flow and adds pseudo-operations in key places. In some sense, all the interesting large-scale issues, experiments and solutions with timeshifting a graph are present in this transformation. The pseudo-operations are placeholders for bookkeeping operations like saving local variables into JIT state structures, scheduling various parts of the compilation, and so on. This phase preserves the colors; its output is graphs that look like pseudo-code, convenient to inspect in the graph viewer for debugging and explanation purposes.
- The modified RTyper (called "HRTyper") is applied in a second phase. It replaces variables and operations locally as described above. The pseudo-operations are replaced along with all the other ones, and become calls to helpers.

## 5.5 Merges and Splits

The description of the timeshifting transformation given so far misses a critical aspect: how to handle control flow (e.g. loops) that involves red variables. It is easy to see why the timeshifted graphs cannot have exactly the same control flow than the original graphs: we may not know at compile-time how many iterations a loop will do at run-time, nor what branches of a condition will be taken.

The issue of loops is handled by *merge points*. Merge points are special operations inserted just after join points, i.e. just after two or more incoming control flow paths meet. A merge point is a check to see if the compiler, i.e. the timeshifted graph, has come back to a state that was previously seen at the same merge point. The goal is that when the timeshifted graph loops back, the residual code that its red operations produces should loop as well, as soon as possible. Without the merge point logic, the timeshifted graph would simply go on looping forever, producing an infinitely long sequence of residual operations. The merge point prevents that: when a state is encountered that was already seen before, the merge logic emits in the residual code a jump that goes back to the residual code corresponding to the older state, thus creating a residual loop. After this, the timeshifted graph stops executing.

The mirror issue is that of *split points*. Splits are conditional branches. There are two kinds of splits: the ones whose condition is itself on a green variable, and the ones whose condition is red. A green split (i.e. the former) is left untouched by the timeshifting transformation: as the condition will be known a compile-time, we know at compile-time which of the branches has to be followed.

A red split is more delicate. It must generate a conditional branch in the residual code. From a theoretical point of view, this is done by emitting a conditional jump instruction, and then *forking* the compile-time process in two identical copies: each copy follows one of the branches in the timeshifted graph, and goes on generating code in the corresponding branch in the residual code.

In practice, this is implemented by making a copy of the current JIT state, which is added to a scheduler's queue. At the end of the timeshifted graph, an extra call to the scheduler will fetch the next pending JIT state and jump back to it. (This implies that the control flow of the timeshifted graph is mangled by the timeshifting transformation, to allow the last block of the graph to jump back to any of the red splits.)

## 5.6 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 compile-time inlining the original callee into its caller.

Again timeshifting determines how to transform call operations based on the color of arguments and result:

- If all arguments and the result are green and we can detect the called function not to have side-effects, the call is a so-called *green call*, and will be left untouched which means there will be a call executed at compile-time.
- Calls to function that have no return value (i.e. their return type is the low-level type Void), are assumed to have side-effects, they are *gray calls*.
- Calls with a green result but with side-effects are so-called *yellow calls*.
- Calls with a red result are *red calls*

The treatment of *red*, *gray* and *yellow* calls is similar, timeshifting will transform them into calls to the timeshifted version of the callee, resulting into an inlining-effect at compile time.

At compile-time the JIT state contains a chain of virtual frames (similar to virtual structures), its top frame is updated with all the red boxes for the call-point red variables before such calls and the callee will attach a fresh new frame. This chain guarantees that values (represented by the boxes) are propagated correctly, accordingly split points duplicate and merge points treat the full chain properly.

Each timeshifted function has its own dispatch logic for splits, it returns to the caller when all the split scheduled states have been dealt with. A list of all the states that reached the return of the function (there can be more than one) is returned to the caller which then schedules these states on its dispatch queue. This is crucial for yellow calls to keep the various green return values disjunct.

What we have described applies to direct calls where the callee is a constant function. For indirect calls, where the callee is a variable it is checked whether the callee is constant at compile-time in which case a mapping between the pointers to the original functions and their timeshifted versions is used to call the appropriate timeshifted callee as for a direct call. If the exact callee is unknown a residual call operation is generated.

## 5.7 Virtual structures

The support code introduced by the timeshifter contains an optimization for all memory allocations that would normally have to occur at run-time: it makes these allocations lazy. A *virtual structure* is such a lazily-allocated run-time data structure.

The hint-annotator gives the color red to the result of a malloc operation. However, the timeshifter turns the malloc into a pure compile-time operation that returns a special PtrRedBox object. The PtrRedBox contains neither a compile-time value (because its value as a pointer is not a compile-time constant) nor a run-time value (because no residual code was generated to perform the malloc yet, so no run-time machine code register contains the pointer's value). Instead, the PtrRedBox references a *virtual structure* object that describes what shape and what values the structure would contain at run-time; the values are themselves further RedBoxes.

For example, in the following code:

```
obj = MyClass()
obj.x = somevalue
...
print obj.x
```

all the variables are red. At compile-time, the malloc corresponding to the instantiation of `MyClass` returns a PtrRedBox with a virtual structure. In the next line, the attribute assignment stores the RedBox corresponding to `somevalue` into the virtual structure. Later on, reading `obj.x` reads out the same RedBox out of the virtual structure. This is all done without generating any residual code; the machine code register that holds `somevalue` at run-time simply stays around and is used directly in the `print` statement.

The above example is of course too simple to reflect the complex control flow patterns of real-life programs. In some cases, merge points prevent us from following virtual structures. Indeed, consider:

```
if condition:
    obj = MyFirstClass()
else:
    obj = MySecondClass()
```

Following the `if` statement, the control flow graph contains such a merge point. Assume first that the `condition` is a compile-time constant. In this case, only one path is considered anyway, so `obj` is propagated into the rest of the code as a PtrRedBox with a specific virtual structure. But assume now that `condition` is not a compile-time constant. In this case, in each incoming path the variable `obj` contains a PtrRedBox with a different type of virtual structure. To avoid an explosion of the number of cases to consider, the compile-time logic *forces* the virtual structures at this point, i.e. it generates code that actually allocates them at run-time. For the rest of the code, the PtrRedBox changes its status and becomes a regular run-time PtrRedBox.

Another case in which a virtual structure must be forced is when the pointer escapes to some uncontrolled place, e.g. when it is stored into a (non-virtual) run-time structure.

Note that the virtual structure object itself stores enough information to know exactly the run-time type it was allocated with. This is necessary because the PtrRedBox itself might be cast up to a less precise pointer type, without necessarily forcing the virtual structure. This is the case in the previous example: the inferred type for `obj` in the rest of the code is the less precise common base class of `MyFirstClass` and `MySecondClass`, whereas if `condition` is a compile-time constant then the virtual structure can carry the precise class of the instance even in the rest of the code.

## 5.8 Virtual lists and dicts

The examples of virtual structures seen so far are about virtualizing objects like instances of RPython classes, which have a `GcStruct` low-level type, i.e. a kind of C `struct`. The same idea can be generalized to objects with other kinds of low-level representations; we call the result *virtual containers*. Beside structures, there are two further kinds of virtual containers: *virtual lists* and *virtual dicts*. The idea is the optimize the following kind of code:

```
lst = []
lst.append(somevalue)
...
print lst.pop()
```

As in the examples of virtual structures, the list manipulations here produce no residual code whatsoever. At compile-time the `lst` variable is a PtrRedBox referencing a *virtual list* object. A virtual list is implemented as a list of red boxes: operations like `append(x)` just append the red box of `x` to the virtual list. The virtual list must be forced in situations where the shape of the list can no longer be known at compile-time, e.g. when inserting an element at an index which is not a compile-time constant.

Virtual dicts are similar, but only support compile-time constants as keys (otherwise, we would not know at compile-time whether two run-time keys are equal or not). They are useful in situations where we build small dicts with fixed keys; in the PyPy interpreter for example we foresee that they will be useful to implement the dictionary of application-level objects, which typically have a few well-known string keys (the instance's attributes).

Note that some care is required when timeshifting low-level code that handles lists and dicts: operations like `lst.append(x)` have already been turned into calls to low-level helpers that manipulate low-level structures and arrays. If no care is taken, the timeshifter will consider this as code that handles structures and arrays, and try to make them virtual structures and (yet unsupported) virtual arrays. However, this is a bit too low-level: e.g. lists that should be similar may have accidentally different structures, because of the details of the over-allocation logic in the low-level helpers. The situation is even worse for dictionaries, where the details of the key placement in the hash table is irrelevant at the RPython level but is likely to confuse the merge logic when trying to compare two virtual dicts. The timeshifter avoids these issues by special-casing the helpers for lists and dicts and mapping them to compile-time operations on virtual lists and virtual dicts. (This is the purpose of the `oopspec` annotation found in the source code of these helpers; as e.g. in rpython/rlist.py.)

## 5.9 Exceptions

The graphs that the timeshifter inputs are RPython code. As such, they typically use exceptions. They must be handled in a specific way - the timeshifted graph itself cannot simply raise and catch the same exceptions at the same places, because most of these exceptions will only occur at run-time under some conditions.

The solution we picked is to *exception-transform* the input graphs before they are timeshifted. The exception transformer is a module that is normally used to prepare a graph for being turned into C code, by removing the exception handling implicit at the graph level and turning it into a form suitable for C. This transformation introduces a *global exception state* structure that stores the current exception object, if any; it turns exception raising into writing into the global exception state, and it adds checks in each graph after each operation that could potentially raise.

Now, as the timeshifter takes exception-transformed graphs as input, it would do the right thing without special support. Indeed, loads from and stores to the global exception state are regular red operations, so they are simply generated into the residual code. In this model, the generated residual code would simply check for exception as often as the original, non-timeshifted graphs did.

The result is not efficient, however, because many original operations were removed - for example, green operations are no longer present, and calls are inlined. This means that the residual graph contains unnecessary repeated checks. To solve this, we made the global exception state structure virtual in some sense. The compile-time state contains two red boxes that correspond to the current exception type and instance; the intent is that they describe run-time values which together are considered as the current exception by the residual code. In other words, when the residual code is itself running, the global exception state structure is

not really containing the current exception; instead, it is in regular local variables (if the red boxes are run-time) or simply known as constants at this point (if the red boxes are compile-time).

To implement this, the timeshifter needs to special-case the following places:

- Operations in the input graphs that directly read from or store to the global exception state structure become compile-time copies from or to the compile-time exception red boxes. For example, when the input, exception-transformed graph wants to clear the current exception, it stores zeroes in the global exception state; the timeshifter transforms these into putting red boxes with compile-time zeroes as the compile-time exception red boxes. Afterwards, any compile-time code that checks if an exception is set would find the compile-time zeroes and know that there isn't any.
- Residual operations that can indirectly raise, like residual calls, would store an exception into the global exception state; so at compile-time, just after we generate such a residual operation, we also generate a pair of `getfield` operations that load the run-time exception back into local run-time variables. This produces non-constant red boxes which are used as the compile-time exception boxes.
- At the end of a residual function, just before returning to the run-time caller, we generate code that copies the exception boxes back into the global exception state.

With the above rules, we can match direct raises with their corresponding exception handlers purely at compile-time. Indeed, consider a graph that explicit raises some exception. The exception transformer first turns it into setting a constant type pointer into the global exception state. Then the timeshifter turns that code into storing a constant red box into the compile-time exception state. The exception remains a compile-time constant until it is either used (allowing the matching code inserted by the exception transformer to give compile-time constant answers) or it escapes out of the residual code (and only in the latter case will residual code store it into the global exception state for the benefit of the run-time caller).

## 5.10 Promotion and global merges

...global merge point...

## 5.11 Partial data

...

## 5.12 Portals

...

## 5.13 Scaling to PyPy

...

### 5.14 Virtualizables

...

## 6 Backends

The compilers produced by the timeshifter are linked with one of our backends, which are written by hand in RPython. We currently have a backend for producing IA32/i386 machine code in memory, PowerPC machine code in memory, or (for testing) further low-level control flow graphs.

### 6.1 The Backend interface

The interface (which is not yet completely stable) is documented in pypy/jit/codegen/model.py.

## 7 Results

...

## 8 Related work

...

## 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. |
| CCLP | Concurrent Constraint Logic Programming. |
| CPS | Continuation-Passing Style. |
| CSP | Constraint Satisfaction Problem. |
| 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. |
| 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. |
| 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. |

### 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 |
| 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

(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