Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python

Armin Rigo
School of Electronics and Computer Science
University of Southampton
SO17 1BJ
United Kingdom
arigo@tunes.org

ABSTRACT

A powerful application of specialization is to remove interpretative overhead: a language can be implemented with an interpreter, whose performance is then improved by specializing it for a given program source. This approach is only moderately successful with very high level languages, where the operation of each single step can be highly dependent on run-time data and context. In the present paper, the Psyco prototype for the Python language is presented. It introduces two novel techniques. The first is *just-in-time specialization,* or *specialization by need,* which introduces the "unlifting" ability for a value to be promoted from run-time to compile-time during specialization – the inverse of the lift operator of partial evaluation. Its presence gives an unusual and powerful perspective on the specialization process. The second technique is *representations,* a theory of data-oriented specialization generalizing the traditional specialization domains (i.e. the compile-time/run-time dichotomy).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Python*

General Terms

Languages, Performance

Keywords

Unlift, Just-In-Time Specialization, Representation, Python

1. INTRODUCTION

Most programming languages can be implemented by interpretation, which is generally a relatively simple and clear approach. The drawback is efficiency. Some languages are designed to lead themselves naturally to more efficient execution techniques – typically static compilation – while others are not. The example on which we will focus is the Python [21] language. Python is trivially strongly typed, i.e. types are attached to the values and any variable can contain values of any type. Moreover, all operations that appear in the source code are fully polymorphic, with their semantics usually defined by the run-time types of the involved values. The operations are heavily overloaded by built-in types and can be further overloaded by user-defined types.

Python is thus essentially an interpreted language, although a number of attempts have been made to compile it, notably Python2C and its successor Pyrex[1]. The problem is that merely removing the interpretative overhead (associated with decoding and dispatching individual bytecode operations) falls short of the expected major performance increase, because each operation still has to determine the types of the involved values at run-time. In particular, values cannot be unboxed.

More recent efforts to compile Python statically [19] have focused on whole-program type inference to solve this problem. While this attempt goes further than the previous ones, it has to limit to some extend the extreme dynamism and reflectivity available in the language. For example, in a Python module, any function f calling another function g does so by looking up and calling the object associated with the global name g in the module; however, each module's mapping of names to objects is itself a Python dictionary that can be obtained and modified from any other point of the program, meaning that the call graph itself is theoretically entirely unknown. This example together with some more common examples of dynamism – object-oriented dispatch with dynamically built or modified classes; an *eval* function to compile and execute arbitrary code; integer arithmetic that silently generates "long integers" when they overflow; etc. – impose a limit on what can be achieved by any static technique.

1.1 Psyco

The Psyco[2] prototype is a practical experiment on how far a completely dynamic optimization system can go. The goal of Psyco was to recover good performance transparently for the programmer, and by techniques that do not limit

[1] http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
[2] http://psyco.sourceforge.net

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'04, August 24–26, 2004, Verona, Italy.
Copyright 2004 ACM 1-58113-835-0/04/0008 ...$5.00.

nor penalize the usage of dynamic features. The present paper discusses the general techniques used in Psyco with a rather theoretical focus. A pragmatic introduction[3] to partial evaluation and the way Psyco does it was presented recently at the ACCU[4] and EuroPython[5] 2004 conferences.

Our work is at the intersection of *on-line partial evaluation* and *just-in-time compilation:*

Just-in-time compilation (as defined by [1]) broadly refers to any kind of compilation (translation between languages, e.g. from Java bytecode to native machine code) that occurs in parallel with the actual execution of the program.

Specialization refers to translation (typically from a language into itself) of a general program into a more limited version of it, in the hope that the specialized version can be more efficient than the general one. **Partial evaluation** [11] is the specialization technique that we will generally consider in the present paper: partial information about the variables and arguments of a program is propagated by abstractedly "evaluating", or interpreting, the program.

1.2 Staged Analysis

The classical presentation of partial evaluation is the following: consider a function $f(x, y)$ of two arguments. If, during the execution of a program, the value of the first argument x is generally less variable than the value of y, then it can be interesting to generate a family of functions $f_1, f_2, f_3 \ldots$ for a family of commonly occurring values $x_1, x_2, x_3 \ldots$ such that $f_n(y) = f(x_n, y)$. Each function f_n can then be optimized independently.

The notation $f(x, y)$ points to an aspect of specialization that is a major difficulty: the choice of how exactly to divide the arguments into the compile-time (x) and the run-time (y) ones. This compile-time/run-time categorization is typically provided either by the programmer as annotations in the source, or by automatic analysis. An efficient result is a delicate balance between under-specialization (missing optimization opportunities) and over-specialization (creating numerous versions of the same code which are only slightly or even not at all better than the more general version). In recent approaches (e.g. [3]) the categorization includes more than two stages, with gradually more information (and less computational time) available for optimization. The categorization, however, is typically done statically, i.e. the *amount* of information that will be available at each stage is fixed in advance.

One could argue that the difficulties involved in finding such a categorization stem from two problems:

- What we are looking for is a global solution of a problem defined by local (but non-modular) constrains induced by the program source. Combining two locally optimal solutions is generally impossible.
- It might not be clear for a single source level operation what amount of information the specializer will need; every single fully polymorphic operation can involve an arbitrary amount of both built-in and user code, and in some languages all operations are fully polymorphic.

A related problem in the most dynamic languages – already described above – is that some of their features could potentially make any static analysis not just incomplete but invalid (dynamic source execution, custom meta-object protocols, etc.).

1.3 Plan

In section 2 we present just-in-time specialization and investigate the extra operational power enabled by this specialization process done entirely at run time instead of compile time, most importantly the ability to constantly query for individual run-time values ("poll") as needed and make them available to the specializer, on a level more fine-grained than profile-based recompilation (section 2.1). It naturally leads to specialization-by-need, where we specialize for all relevant values and only them (section 2.3).

The extra flexibility prompts for a more general theory of data-oriented specialization, sketched in section 3.

Experimental results and discussion is presented in section 4. Finally, section 5 compares Psyco with related work and concludes.

2. JUST-IN-TIME SPECIALIZATION

The present section introduces the basic idea behind just-in-time specialization from a practical point of view. The following section 3 will sketch the formal theory supporting it.

2.1 The Unlift Operator

Specialization makes use of a subset of all information that will be available at execution, called the *compile-time* data. The rest, the *run-time* information, is only available later, and the specialization cannot depend on it. While it is obvious in off-line specialization, which runs before execution, it is also true in on-line specialization: the specialized version of a function, by definition, should only depend on the compile-time values, because the goal is to apply the same specialized function several times, to multiple run-time values. It is possible for the specializer to copy the value of a compile-time variable into a run-time variable, thus "forgetting" it. The inverse is normally not possible.

Thus, a specializer is typically invoked on some function (or method) with specific information about the function's argument (or the method's receiver), e.g. their types. It then propagates this information through the body of the function to create a specialized version of it. However, it may happen that after such propagation not enough information is known about the values involved in some operation to compile that operation efficiently. The problem of avoiding (or minimizing the impact of) these cases while still controlling the number of different specialized versions to generate is a difficult one, typically involving a global analysis of the program.

A more local point of view is that in these cases it would be interesting to gather more compile-time (i.e. early) information about some value and make it available to the specializer. This operation is essential; in some respect, it is what on-line specializers implicitly do when they start their job: they take an input (run-time) value and start generating a version of the function specialized for this (now considered compile-time) value.

Let us make this operation explicit. We call it **unlift**, as it is effectively the inverse of the **lift** operator which in partial evaluation denotes that a compile-time value should be "forgotten" (i.e. considered as run-time) in the interest

[3] http://psyco.sourceforge.net/accu2004-psyco.tgz
[4] http://www.accu.org/conference/python.html
[5] http://www.europython.org/

of a greater generality of the residual code. Unlift allows the specializer to directly ask for arbitrarily more information about any value at any time (e.g. "what is the type of x?"). It is an oracle that non-deterministically returns each of the answers that the question will have during execution (e.g. `integer`, `float`, `long-precision integer`, ...).

It is actually possible to implement such an operator: lazily. The specializer is suspended when it asks such a question, and the actual execution of the residual code generated so far is started. When execution reaches the point where unlift was invoked, the run-time answer is now available; so it can be sent back to the specializer, which is wakened. From the specializer's point of view, the unlift now returns an answer which can be used for further specialization. This operator is non-deterministic because a future execution of the same residual code may return a different run-time answer, in which case the specializer will be resumed again from the same position.

In practice, the unlift operation saves the state of the specializer in a snapshot that can be used as a continuation. Each new answer that shows up at run-time causes the continuation to be resumed. This differs from profile-based specialization in that specialization is not only guided by some data collected during a previous run of the program; it both controls what information is relevant to collect, and it is controlled by this information in a very immediate and fine-grained manner.

Introducing unlift does not raise serious practical problems. By comparison, the common problems found in most forms of on-line specialization (see section 2.5) are much more difficult.

2.2 Example

Consider the following function:[6]

```
def f(n):
    return 2*(n+1)
```

For reasons explained in section 2.3 below, we start the specializer with the most general case: nothing is known about the input argument n. Two notes first:

- If we assumed that we have no unlift operator, we would produce in such a situation a version of f that is actually not specialized at all, but only "compiled" in the sense that the interpretative overhead is removed. Decoding the type of the arguments would still be done at run-time for each operation.

- Some programming languages like ML infer locally the type of the function's variables and input arguments by the way they are used; e.g. the `+` and `*` in the body of the function could hint that n is a number. This approach does not work in Python because a wide range of built-in types (and an unknown family of user-defined ones) can implement these operators.

Table 1 shows how specialization and execution are interleaved in time when unlift is used. Note that we only starts specialization when the first actual (run-time) call to f takes place.

The steps taken by the specializer should be pretty intuitive: it follows the source code of f and – starting from a situation where nothing is known about n – it asks the most general questions that are deemed useful to implement the next operation efficiently. So the addition in the source code forces the specializer to ask for (i.e. unlift) the type of n. When unlift returns `int` it knows that the addition is between two integers, and produces the efficient residual code that computes the (unboxed) result into a machine register. The multiplication is then known without further unlifting to be between two unboxed integers, so the specializer directly produces the integer multiplication in the residual code.

Table 1 also shows the specializer's concern about overflows: checks are implemented by unlifting the overflow bit from the processor's flags registers.

Subsequent invocations of f with another integer argument n will reuse the already-compiled code, i.e. the left column of the table. Reading the left column only, you will see that it is nothing less than the optimal run-time code for doing the job of the function f, i.e. it is how the function would have been manually written, at least for the signature "accepts an arbitrary value and returns an arbitrary value".

As long as none of the three answers changes, no further specialization work is needed. This is implemented by compiling each excursion through the right column into a single conditional jump in the left column. For example, the answer to an "overflow?" question is implemented as a "jump-if-not-overflow" instruction whose target is the next line in the left column. As long as the question receives the same answer, it is a single machine jump that completely bypasses the specializer.

If, however, a different answer is later encountered (e.g. when executing $f(2147483647)$ which overflows on 32-bit machines), then it is passed back to the specializer again, which resumes its job at that point (unlift returns a new answer). This results in a different code path, which does not replace the previously-generated code but completes it. The specializer actually patches the conditional jump instruction to include the new case as well. In the above example, the "jump-if-overflow" instruction will be patched: the non-overflowing case goes (as before) to the first version of the code, but the overflowing case now points to the new code, where arbitrary-precision integers are used.

As another example, say that f is later called with a floating-point value. Then the first question "what is the type of n?" receives a different answer. This point in the left column was compiled as a two-way jump: if the type is `int` jump directly to the next line in the left column; otherwise jump back to the specializer. When the latter occurs, new code is compiled and the conditional jump is patched to become a three-way jump[7]: when the answer is `int` it jumps to the first version; when it is `float` to the second version; and otherwise it calls back again the specializer.

The point here is that all choices are made at run-time following purely local decisions by the specializer. In a sense, we are specializing for all possible cases with guards that select the correct case among the potential infinity,[8] lazily.

[6] We use the syntax of Python, which should be immediately obvious in these small examples.

[7] which probably requires more than one processor instruction, and which grows when new cases are encountered. This kind of machine code patching is quite interesting in practice.

[8] New types can be created at any time in Python even without using the obviously dynamic constructions, but simply

execution		specialization
Program call $f(12)$	$-\ start\ \rightarrow$	Start compiling $f(n)$ with nothing known about n
	|	For $n+1$ it would be better to know the type of n.
Start executing $f(n)$ as compiled so far with $n = 12$.	$\leftarrow\ run\ -$	What is the type of n?
	|	
Read the type of n: `int`. The value asked for:	$-\ int\ \rightarrow$	Proceed with the addition of two integer values:
	|	Write code that reads the value into a register and adds 1.
Execute the read and add instructions, result 13.	$\leftarrow\ run\ -$	Did it overflow?
	|	
The answer asked for:	$-\ no\ \rightarrow$	We know that $(n+1)$ and 2 are integers
	|	so we write code that multiply them.
Execute the multiplication instruction, result 26.	$\leftarrow\ run\ -$	Did it overflow?
	|	
The answer asked for:	$-\ no\ \rightarrow$	Result is an integer,
	|	return it.
Return 26.	$\leftarrow\ run\ -$	

Table 1: Mixed specialization and execution with three applications of the unlift operator (the three questions asked during specialization). The control flow follows the arrows.

2.3 The Top-Down Approach

Unlifting makes specialization and execution much more interleaved in time than even on-line specialization. We call this particular technique **just-in-time specialization.** This approach sidesteps a number of common issues and lowers the need for complex heuristics or detailed source code annotations.

For example, *widening* is not essential in the top-down approach. Instead of starting with highly specialized versions of the code and generalizing when new values are found that do not fit in the previous constrains (occasionally over-generalizing for fear of never terminating), we can start with the most general inputs and gradually specialize by applying the unlift operator. Perhaps even more important: we can unlift only when there is a need, i.e. an immediately obvious benefit in doing so. In other words, we can do **need-based specialization.**

A "need to specialize" is generally easy to define: try to avoid the presence in the residual code of some constructs like indirect function calls or large switches, because they prevent further optimizations by introducing run-time choice points. Specializing away this kind of language construct is a natural target. This can be done simply by unlifting the value on which the dispatch takes place.

By generalization, unlifting is also used to specialize simple two-way choices like a check for overflow. This is done by unlifting the single bit among the processor's flags[9] that represents the overflowing condition of the previous operation. This unlifted value – 0 or 1 – is only used by the specializer to know which branch it must now inspect. The general unlifting machinery will ensure that it is compiled as a simple conditional jump.

2.4 Entry Points and Loops

The residual code (left column) of table 1 was generated using this top-down approach: it has got a single entry point accepting any argument, and it forks inside the body if and where it needs to. In Psyco, functions can have additional entry points when they are called by other functions: the information known about the arguments in the caller (or a part thereof) is propagated to the called function. At this place, widening and associated heuristics are still needed in Psyco: although the specialization of a function proceeds in a top-down way that helps control the explosion of unnecessary entry points, their number should be bounded in practice. If an entry point deemed too similar to another one is generated, some information is lifted ("forgotten") in order to unify it with any sufficiently similar future entry point.

If Python had no looping construct[10] then looping would be achieved by recursive calls, which the specializer would compile into calls (or tail-calls) to one of the entry points of the same function, using nothing but the techniques described above. The body of the function would thus be compiled into a tree of alternatives, forking at unlift points, whose leaves are either returns or tail-calls (i.e. jumps).

To handle looping constructs, Psyco essentially preprocesses each function to turn it into a family of mutually recursive non-looping functions. This whole-function analysis is the most global kind of analysis performed by Psyco. It is done by splitting the function into blocks along the head of loops and some other control path merge points, carefully selected to minimize duplication of specialization efforts while still avoiding very small blocks that would lead to the proliferation of the total number of entry points (each of which requires associated management data).

An important consequence is that specialization terminates if the program does, because specialization is never far ahead of the execution of the program: at most a single "edge" in the "tree of alternatives" that is the body of a non-looping function.

2.5 Issues with Just-In-Time Specialization

Just-in-time specialization, like on-line specialization, requires caching techniques to manage the set of specialized versions of the code, typically mapping function entry point patterns to generated machine code. This cache potentially requires sophisticated heuristics to keep memory usage under control, and to avoid over-specialization. As mentioned

by instantiating the `type` class.

[9] At least on x86 architectures.

[10] Most functional languages do not.

above, this cache is not only used on entry point for functions that explicitly appear in the source code, but also at the head of loops and some other control path merge points in the function bodies.

Perhaps the most important problems introduced by the top-down approach are:

1. memory usage, not for the generated code, but because a large number of continuations are kept around for a long time — forever, a priori. In the example of section 2.2, we can never be sure that f will not be called later with an argument of yet another type.

2. low-level performance: the generated code blocks are extremely fine-grained. As in the above example, only a few machine instructions can typically be generated before the specializer must give the control back to execution. The instructions just produced are immediately executed. This defies common compiler optimization techniques like register allocation. Moreover, it may be unsuited to modern processors where the data and instruction caches need to be explicitly (and slowly) synchronized. Care must also be taken to keep some code locality: processors are not good at running code spread over numerous small blocks linked together with far-reaching jumps.

A possible solution to these low-level problems would be to consider the code generated by the specializer as an intermediate version on the efficiency scale. It may even be a low-level pseudo-code instead of real machine code, which makes memory management easier and avoids issues with the processor caches. It would then be completed with a better compiler that is able to re-read this pseudo-code later and optimize it more seriously based on real usage statistics. Such a two-phase compilation has been successfully used in a number of projects (see [1]).

3. REPRESENTATIONS

This section introduces a formalism to support and represent partial information about a value.

When the values manipulated by a source program are potentially complex objects, they cannot be simply either run-time (unknown) or compile-time (entirely known). Intermediate levels of information are necessary, like "known-to-be-of-type-`int`". To this end, it is common in partial evaluation to follow a pre-existing representation of the values and introduce static (compile-time) and dynamic (run-time) divisions on top of it. For example, in a logic language in which values are recursive terms like `fraction(22,7)`, we can represent a value which is known to be a fraction as `fraction(_,_)` where each "`_`" is a placeholder meaning "this part of the term is run-time".

While Psyco originally started in a similar way by decomposing the C-level structures implementing Python objects in the standard Python interpreter, it was found not to be expressive enough. Here are two motivating examples:

1. In Python, `range(5,15)` returns the list[11] of integers from 5 included to 15 excluded. The `range` function is commonly used in loops, because the basic looping construct of Python is the iteration over a sequence. Psyco tries never to build the whole list of integers in memory; this is possible if it remembers that some variable z contains "the range of integers between x and y".

2. More subtly, at the core of unboxing there is a difference between, say, a structure in memory that is known to represent an integer object, and a 32-bit integer in some processor register that stands for an integer object which Psyco is trying not to copy into a fully-formed heap structure.

To generalize these examples, we introduce a theory supporting arbitrary representation of values and operations between them, motivated by Psyco (see section 5.1 for related ideas and formalisms).

3.1 Representations

We call *type* a set of values; the type of a variable is the set of its allowed values.

DEFINITION 1. *Let X be a type. A* **(type) representation** *of X is a function $r : X' \to X$. The set $X' = dom(r)$ is called the* **domain** *of the representation.*

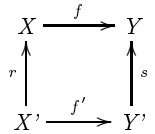
The name *representation* comes from the fact that r allows some values from X to be "represented" by an element of X'. An $x' \in X'$ represents the value $r(x') \in X$. For example, the domain X' could be a subtype of X, r being just the inclusion; or if X is the set of all Python objects, and X' is the set of machine-sized words, then r could map a machine word to the corresponding Python integer object, a representation which is not trivial because the interpreter associates meta-data to all integer (at least its type in the Python sense and a reference counter).

The two extreme examples of representation are

1. the universal representation $id_X : X \to X$ that represents any object as itself;

2. for any $x \in X$, the constant representation $c_x : \{\cdot\} \to X$, whose domain is a set with just one (arbitrary) element "$\cdot$", whose image $c_x(\cdot)$ is precisely x.

DEFINITION 2. *Let $f : X \to Y$ be a function. A* **(function) representation**[12] *of f is a function $f' : X' \to Y'$ together with two type representations $r : X' \to X$ and $s : Y' \to Y$ such that $s(f'(x')) = f(r(x'))$ for any $x' \in X'$:*



r is called the **argument representation** *and s the* **output representation**.

A **partial representation** *is a partial function f' with r and s as above, where the commutativity relation holds only where f' is defined.*

If r is the inclusion of a subtype X' into X, and if $s = id_Y$, then f' is a **specialization** of f: indeed, it is a function that

[11] "Lists" in Python are actually arrays, not linked lists.

[12] We use the word "representation" for both types and functions: a function representation is exactly a type representation in the arrow category.

gives exactly the same results as f, but which is restricted to the subtype X'. Computationally, f' can be more efficient than f — it it the whole purpose of specialization. More generally, a representation f' of f can be more efficient than f not only because it is specialized to some input arguments, but also because both its input and its output can be represented more efficiently.

For example, if $f : \mathbb{N} \to \mathbb{N}$ is a mathematical function, it could be partially represented by a partial function $f' : M \to M$ implemented in assembly language, where M is the set of machine-sized words and $r, s : M \to \mathbb{N}$ both represent small integers using unsigned machine words. This example also shows how representations can naturally express relationships between levels of abstractions: r is not an inclusion of a subtype into a type; the type M is much lower-level than a type like $\mathbb{N}$ which can be expected in high-level programming languages.

3.2 Specializers

DEFINITION 3. *Let $f : X \to Y$ be a function and R a family of representations of X. We call R-***specializer** *a map S_f that can extend any $r \in R$ into a representation of f with argument r:*

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle r}\uparrow & & \uparrow{\scriptstyle s} \\ X' & \xrightarrow{S_f(r)} & Y' \end{array}$$

Note that if R contains the universal representation id_X, then S_f can also produce the (unspecialized) function f itself: $s(S_f(id_X)(x)) = f(x)$ i.e. $f = s \circ S_f(id_X)$, where s is the appropriate output representation of Y.

The function $x' \mapsto S_f(r)(x')$ generalizes the compile-time/run-time division of the list of arguments of a function. Intuitively, r encodes in itself information about the "compile-time" part of the arguments of f, whereas x' provides the "run-time" portion. In theory, we can compute $r(x')$ by expanding the run-time part x' with the information contained in r; this produces the complete value $x \in X$. Then the result $f(x)$ is represented as $s(S_f(r)(x'))$.

For example, consider the particular case of a function $g(w, x')$ of two arguments. For convenience, rewrite it as a function $g((w, x'))$ of a single argument which is itself a pair (w, x'). Call X the type of all such pairs. To make a specific value of w compile-time but keep x' at run-time, pick the following representation of X:

$$\begin{array}{rcl} r_w : X' & \longrightarrow & X = W \times X' \\ x' & \longmapsto & (w, x') \end{array}$$

and indeed:

$$\begin{array}{ccc} \mathrm{W} \times \mathrm{X'} & \xrightarrow{g} & \mathrm{Y} \\ {\scriptstyle r_w}\uparrow & & \uparrow \\ \mathrm{X'} & \xrightarrow{S_g(r_w)} & \mathrm{Y} \end{array}$$

$S_g(r_w)(x') = g(r_w(x')) = g((w, x'))$, so that $S_g(r_w)$ is the specialized function $g((w, -))$.[13] With the usual notation $f_1 \times f_2$ for the function $(a_1, a_2) \mapsto (f_1(a_1), f_2(a_2))$, a compact way to define r_w is $r_w = c_w \times id_{X'}$.[14]

3.3 Example

Consider a compiler able to do constant propagation for a statically typed language like C. For simplicity we will only consider variables of type `int`, taking values in the set Int.

```
void f(int x) {
  int y = 2;
  int z = y + 5;
  return x + z;
}
```

The job of the compiler is to *choose a representation* for each variable. In the above example, say that the input argument will be passed in the machine register A; then the argument x is given the representation

$$\begin{array}{rcl} r_A : \mathit{Machine\ States} & \longrightarrow & \mathit{Int} \\ \text{state} & \longmapsto & \text{register } A \text{ in state} \end{array}$$

The variable y, on the other hand, is given the constant representation c_2. The compiler could work then by "interpreting" symbolically the C code with representations. The first addition above adds the representations c_2 and c_5, whose result is the representation c_7. The second addition is between c_7 and r_A; to do this, the compiler emits machine code that will compute the sum of A and 7 and store it in (say) the register B; this results in the representation r_B.

Note how neither the representation alone, nor the machine state alone, is enough to know the value of a variable in the source program. This is because this source-level value is given by $r(x')$, where r is the (compile-time) representation and x' is the (run-time) value in $dom(r)$ (in the case of r_A and r_B, x' is a machine state; in the case of c_2 and c_5 it is nothing, i.e. "$\cdot$" – all the information is stored in the representation in these extreme cases).

This is an example of off-line specialization of the body of a function f. If we repeated the process with, say, c_{10} as the input argument's representation, then it would produce a specialized (no-op) function and return the c_{17} representation. At run-time, this function does nothing and returns nothing, but it is a nothing (i.e. a "$\cdot$") that represents the value 17, as specified by c_{17}.

An alternative point of view on the symbolic interpretation described above is that we are specializing a C interpreter $interp(source, input)$ with an argument representation $c_f \times r_A$. This representation means "*source* is known to be exactly f, but *input* is only known to be in the run-time register A".

3.4 Representations in Psyco

For specializers, the practical trade-off lies in the choice of the family R of representations. It must be large enough to include interesting cases for the program considered, but small enough to allow $S_f(r)$ to be computed and optimized with ease. But there is no reason to limit it to the examples seen above; let us introduce some more flexibility.

The Python language is trivially typed; let Obj be the set of all Python objects. The standard Python interpreter

[13] If R contains at least all the r_w representations, for all w, then we can also reconstruct the three Futamura projections, though we will not use them in the sequel.

[14] The latter is actually a function $\{\cdot\} \times X' \to W \times X'$ but we will systematically identify $\{\cdot\} \times X'$ with X'.

represents any object as a pointer to a C structure called `PyObject` with some common meta-data followed by type-dependent data. If we call $PyObj$ the set of pointers to such structures,[15] then the interpreter uses a single representation for all objects:

$$cpy : PyObj \to Obj$$

Psyco itself is best described as handling representations of the C types manipulated by the standard interpreter. Note that Psyco is not an independent program executing Python code, but an extension module for the standard interpreter, for reasons described in the next section.

The domains of the representations are at the level of the generated assembler code (i.e. Psyco represents C-level information using processor-level data). Integers and pointers are represented as in section 3.3 above; additionally, pointers can recursively specify representations for the individual fields of the structure they point to. For example, a heap structure which is known to be a `PyObject` of type `int` could be represented in Psyco by $r_A[c_{int}]$, i.e. the address in the register A known to be pointing to a structure whose first field (the Python type) can be represented by c_{int}.

As described in the introduction of section 3, these representations are however not enough in general. So Psyco also handles a family of custom representations for the C type "pointer to `PyObject`". Here are the main examples of such representations:

- unboxing of integers: for any representation r, $vint[r]$ represents a pointer to a (virtual) `PyObject` structure of type `int` whose value is represented by r. The actual structure does not exist in memory. This allows Python-level integer objects to be implemented directly in processor registers.

- unboxing of other simple structures: tuples of known size can be represented by $vtuple[r_1, \ldots, r_n]$ where r_1, $\ldots$, r_n individually represent the elements of the tuple. Here again, the goal is to avoid building the tuple `PyObject` structure in memory.

- ranges: $vrange[r_1, r_2]$ stands for the list of all integers between the bounds represented by r_1 and r_2.

- string slices (i.e. substrings): $vslice[r_1, r_2, r_3]$ stands for the substring of r_1 starting at r_2 and ending at r_3. The goal is to avoid duplicating the string storage memory.

- methods: $vmethod[r_1, r_2]$ represents the closure of the function r_1 with first argument r_2. Methods in Python are regular function objects with an explicit receiver conventionally called `self`. Python is lookup-based: when a name m is looked up on an object x, if m resolves to a function object stored in the type of x (or in a superclass of it[16]) then the result is a closure, a so-called "bound method" object: when (and if) this object is called, the underlying function is invoked with its first argument bound to x. Psyco avoids building the temporary bound method object in memory by unboxing closures.

- strings as over-allocated buffers: a common idiom to build a long string (e.g. a dynamic HTML page) is to repeatedly concatenate small bits. However, this has a bad (quadratic) performance because for each concatenation Python must copy the data of both strings into a new one. Thus Psyco has a representation $vbuf[r_1, r_2]$ where the first r_2 bytes of a buffer r_1 stand for a string object. The buffer is over-allocated, so that it can accommodate concatenation a number of small strings before it needs to be reallocated. The operation is now amortized linear.

The latter example cannot be done by the Python interpreter directly, because all objects have a single representation cpy and for compactness reasons string objects do not have enough indirections. The advantage Psyco has is that it can choose and change the representation of objects as needed, as described in section 3.6.

3.5 Integration with an Interpreter

Psyco is an extension module for the standard interpreter, and its representations are of the C-level data manipulated by the interpreter, not directly of Python objects.

Consider a family R of representations of the set Obj of Python objects. One way to ensure that all values can be represented (without adding ever more cases in the definition of R) is to include the universal representation id_X among the family, or a similarly "broad" representation. In other words, for Psyco we need a uniform way to represent arbitrary Python objects by assembler-level data. The cpy representation of section 3.4 does precisely this (though with C-level data instead; the "broad" representations of Psyco are actually the ones of the form $cpy \circ r_A$, i.e. a pointer in register A to some `PyObject` structure). This means that cpy is a very interesting representation to have in Psyco. For this reason, Psyco is tightly integrated with the standard Python interpreter instead of being a stand-alone tool.

Actually, the presence of a broad representation b among R suddenly makes any specializer tightly integrated with a regular interpreter.[17] A function "specialized" with all its variables represented as b is inefficient: although the interpretative overhead is removed, it still contains the overhead of decoding at run-time the operand types for all the operations and dispatching to the correct implementation. In other words it is very close to an interpreted version of f. Let us now assume that an interpreter is available for use by the specializer. Then the introduction of b provides a safe "fall-back" behavior: the specializer cannot fail; at worst it falls back to interpreter-style type-based dispatching. This is a desirable property if we consider interpreters which are large and even dynamically extensible: no predefined representation set R can cover all possible cases unless it contains such a b.

Note that without the unlift operator, the representation b would be very pervasive: typically, operations involving it are encoded as a call to some API function of the interpreter (e.g. `PyNumber_Add()`) which produces a result which is also entirely unknown, i.e. that is also represented by b.

[15] A $p \in PyObj$ actually encodes both a memory address and the content of the memory reachable through this address.

[16] Types are the same thing as classes.

[17] Interestingly, it also means that the specializer is relatively independent of the language being interpreted, and only depends on the general principles of the interpreter.

3.6 Changes of Representation

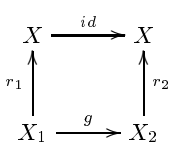
Any kind of behavior using a fall-back representation raises the problem of its pervasiveness in the subsequent computations. This was the major motivation behind section 2: just-in-time specialization enables "unlifting". The top-down approach (section 2.3) is precisely about starting with a broad representation and gathering more specific information as needed.

Recall that to *lift* is to move a value from compile-time to run-time; in term of representations, it means changing from a specific representation (e.g. c_{42}) to a more general one (e.g. r_A, the change being done by residual assembler code that loads 42 into the register A). Then *unlifting* is a technique to solve the pervasiveness problem by doing the inverse, i.e. switching from a general representation like r_A to a more specific one like $r_A[c_{int}]$.

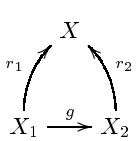
The job of Psyco can be described as choosing and changing the representations of values until they match a reasonably efficient implementation of the desired operation. For example, the processor provides an integer addition operation that can be described as $S_+(vint[r_A] \times vint[r_B])$: in other words, it corresponds to the specialization of the addition between Python objects, provided that these objects are represented (and in particular representable) as unboxed integers in two registers A and B. The example of table 1 can be reformulated as follows:

- n is initially represented by, say, r_A if the input argument is passed in the machine register A.
- After the first unlift, it is a pointer to a `PyObject` known to be of type `int`, i.e. $r_A[c_{int}]$.
- The integer value is loaded from the structure into register B; now n is represented as $vint[r_B]$.
- The addition is the processor instruction corresponding to $S_+(vint[r_B] \times c_1)$. (The existence of such an instruction is what prompted the previous step: adapting the representation of n to match the available efficient instruction.)
- After the addition, the result – say in register C – is represented as an unboxed integer object, i.e. $vint[r_C]$.
- Similarly, the multiplication is $S_*(c_2 \times vint[r_C])$ and its result is $vint[r_D]$.
- The function must return a real pointer, so the result's representation is changed into $r_E[c_{int}]$ by building the structure in the heap; E is the result of the corresponding `malloc()`.

Both lifting and unlifting are instances of the more general *change of representation* kind of operation. In the terminology of section 3.1, a change of representation is a representation of an identity, i.e. some low-level code that has no high-level effect:



or equivalently



A lift is a function g that is an inclusion $X_1 \subset X_2$, i.e. the domain of the representation r_1 is widened to make the domain of r_2. Conversely, an unlift is a function g that is a restriction: using run-time feedback about the actual $x_1 \in X_1$ the specializer restricts the domain X_1 to a smaller domain X_2 containing $g(x_1)$. Unlifts are partial representations. Run-time values may later show up that a given unlift (or any other partial representation) cannot handle, requiring re-specialization.

Another example of partial representation is that in practice, a number of functions have an efficient representation for "common cases" but require a significantly more complex representation to cover all cases. For example, integer addition is often representable by the processor's addition of machine words, but this representation is partial in case of overflow.

In the spirit of section 2.2 we solve this problem by forking the code into a common case and an exceptional one; e.g. by default we select the (partial) representation "addition of machine words" $S_+(vint[r_A] \times vint[r_B])$; if an overflow is detected, we build the exceptional branch by first lifting the arguments to some more general representation – typically b – and then using $S_+(b \times b)$, which in the case of Psyco is the generic Python API addition function `PyNumber_Add`. (This is similar to recent successful attempts to use a regular interpreter as a fall-back for exceptional cases, e.g. [22].)

4. EXPERIMENTAL RESULTS

4.1 Overview

Psyco generates machine code by writing the corresponding bytes directly into executable memory (it cannot save machine code to disk; there is no linker to read it back). Its architecture is given in figure 1.

Psyco consists of three main parts (second row), only the latter two of which (in solid frames) are hard-coded in C. The former part, the *machine code,* is dynamically generated.

- The Python C API is provided by the unmodified standard Python interpreter. It performs normal interpretation for the functions that Psyco does not want to specialize. It is also continuously used as a data manipulation library. Psyco is not concerned about loading the user Python source and compiling it into bytecode (Python's pseudo-code); this is all done by the standard Python interpreter. (Psyco itself works on this in-memory bytecode, not directly on the source code.)
- The *specializer* is the symbolic Python interpreter: it works by symbolically interpreting Python bytecodes with representations instead of real values. This interpreter is not complete: it only knows about a subset of the built-in types and a subset of their operations. For any missing piece, it falls back to general representations (section 3.5).
- The *machine code* implements the execution of the Python bytecode. After some time, when the specializer is no longer invoked because all needed code has been generated, then the machine code is an almost-complete, efficient low-level translation of the Python source. (It is the left column of table 1.)

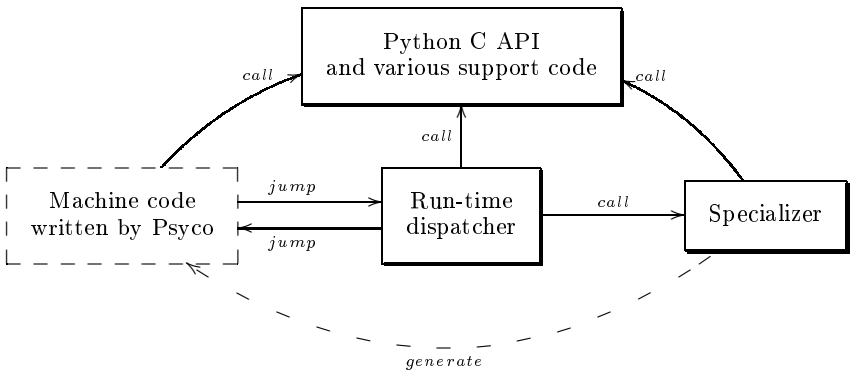


Figure 1: The architecture of Psyco (bottom row) within the standard Python interpreter (top box)

- The *run-time dispatcher* is a piece of supporting code that interfaces the machine code and the specializer. Its job is to manage the caches containing machine code and the continuations that can resume the specializer when needed.

Finally, a piece of code not shown on the above diagram provides a set of hooks for the Python profiler and tracer. These hooks allow Psyco to instrument the interpreter and trigger the specialization of the most computationally intensive functions.

4.2 Implementation Notes

Particular attention has been paid to the **continuations** underlying the whole specialization process. Obviously, being implemented in C, we do not have general-purpose continuations in the language. However, general tools like Lisp or Scheme continuations would probably be totally impractical. The reason is the memory impact (section 2.5). It would not be possible to save the state of the specializer at all the points where it could potentially be resumed from, i.e. all the points where it uses unlift. Psyco emulates continuations by saving the state only at entry points (section 2.4). The intermediate C state at which the unlift operation was invoked can be reconstructed by carefully replaying all the C code from the most recent entry point, repeating the steps taken by the specializer but discarding the assembler code re-generated. Also note that the entry points snapshots, which have to describe the representation of each active variable, are stored in memory in a more compact form than that used during specialization. Memory usage has been a real problem in the early versions of Psyco.

Code generation is also based on custom algorithms, not only for performance reason, but because general compilation techniques cannot be applied to code that is being executed piece by piece almost as soon as it is created. For this reason, existing run-time code generation frameworks like VCODE [7] have been considered for Psyco but rejected. To date, relatively little efforts have been spent in Psyco on optimizing the generated code (which is known to be difficult in general).

Besides the Intel i386-compatible machine code, Psyco has recently been "ported" to a custom **low-level virtual machine architecture.** This architecture will be described in a separate paper.

The **profiler hooks** in Psyco select the functions to specialize based on an "exponential decay" weighting algorithm, also used e.g. in Self [9]. An interesting feature is that, because the specializer is very close in structure to the original interpreter (being a symbolic interpreter for the same language), it was easy to allow the profiler hooks to initiate the specialization of a function *while it is running,* in the middle of its execution – e.g. after some number of iterations in a long-running loop, to accelerate the remaining iterations. This is done essentially by building the *cpy* representation of the current (interrupted) interpreter position (i.e. the representation in which nothing is known about the objects apart from their existence as `PyObject` structures), and starting the specializer from there.

4.3 Performance Results

As expected, Psyco gives massive performance improvements in specific situations. Larger applications where time is not spent in any obvious place benefit much less from the current, rather low-level incarnation of this prototype. In general, on small benchmarks, Python programs run with Psyco exhibit a performance that is near the middle of the (large) gap between interpreters and static compilers. This result is already remarkable, given that few efforts have been spent on optimizing the generated machine code (more in section 5.2).

Here are the programs we have timed:

- **int arithmetic:** An arbitrary integer function, using addition and subtraction in nested loops. This serves as a test of the quality of the machine code.
- **float arithmetic:** Mandelbrot set computation, without using Python's built-in complex numbers. This also shows the gain of removing the object allocation and deconstruction overhead, without accelerating the computation itself: Psyco does not know how to generate machine code handling floating points so has to generate function calls.
- **complex arithmetic:** Mandelbrot set computation. This shows the raw gain of removing the interpretative

Benchmark	Python	Psyco		Pyrex		Pyrex w/ types		C		C with ovf.	
	time	time	speed-up	time	speed-up	time	speed-up	time	speed-up	time	speed-up
int arith.	28.5	0.262	109×	15.6	1.83×	0.099	288×	0.102	281×	0.393	73×
float arith.	26.9	2.47	10.9×	12.7	2.12×	0.993	27×	0.181	149×		
complex arith.	18.9	5.18	3.65×	14.1	1.34×	14.1	1.34×	0.480	39×		
files and lists	20.9	1.52	13.8×	19.0	1.10×	20.1	1.04×	0.095	220×		
Pystone	19.3	3.94	4.9×								
ZPT	12.3	6.1	2×								
PyPy 1	5.27	3.54	1.49×								
PyPy 2	60.7	59.9	1.01×								

Table 2: Timing the performance improvement of Psyco

overhead only: Psyco does not know about complex numbers.

- **files and lists:** Counts the frequency of each character in a set of files.
- **Pystone:** A classical benchmark for Python,[18] though not representative at all of the Python programming style.
- **ZPT:** Zope Page Template, an HTML templating language interpreted in Python. Zope[19] is a major Python-based web publishing system. The benchmark builds a string containing an HTML page by processing custom mark-ups in the string containing the source page.
- **PyPy 1:** The test suite of PyPy, a Python interpreter written in Python, first part (interpreter and module tests).
- **PyPy 2:** Second part (object library implementation).

Each simple benchmark was written for and executed by the following compilers or interpreters:

- Python 2.3.3, the standard interpreter.
- Psyco (on unmodified source).
- Pyrex[20], a compiler for a language close to Python (removing the interpretative overhead only).
- Pyrex on a source with explicit type annotations (which typically suffice to make simple benchmarks compilable to efficient C).
- Hand-written C code, compiled by gcc 2.95.2.
- Hand-written C code checking for integer overflow.[21]

[18] `Lib/test/pystone.py` in the Python distribution.
[19] http://www.zope.org
[20] http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
[21] Although no operation in these tests overflows the 32-bit words, both Python and Psyco systematically check for it. It is thus interesting to compare it with a C version that does it too. The checks are encoded in the C source; Psyco can be faster because it can use the native processor overflow checks. Pyrex does not perform any overflow checking on typed source.

The larger programs used as benchmark have not been translated to Pyrex or C.

The results (table 2) have been obtained on a Linux Pentium III laptop at 700MHz with 64MB RAM. Times are seconds per run. The "speed-ups" are the acceleration factor with respect to Python times. All tests are run in maximum compilation mode (`psyco.full()`), i.e. without using the profiler but blindly compiling as much code as possible.

These results are *not representative* in general because we have, obviously, selected examples where good results were expected. They show the behavior of Psyco on specific, algorithmic tasks. Psyco does not handle large, unalgorithmic applications so well. It is also difficult to get meaningful comparisons for this kind of application, because the same application is generally not available both in Python and in a statically compiled language like C. Still, there is much anecdotical evidence that Psyco is useful in practice, primarily as a way to avoid manual translation of time-consuming algorithms (number-crunching, image filters, text manipulation, etc.).

The present prototype moreover requires some tweaking to give good results on non-trivial examples, as described in [18], section 2.2.

It would be interesting to compare the results with the Starkiller type inferencer and compiler [19], but although it has been announced it is not yet publicly available.

5. RELATED AND FUTURE WORK

5.1 Related Work

We mention here the work to which our techniques are related. Section 1 lists the other techniques in the current state of the art for Python.

The classical reference for efficient execution of dynamic programming languages is the implementation of Self [4], which transparently specializes functions for specific argument types using statistical feed-back. A number of projects have followed with a similar approach, e.g. [6]. Our approach differs in that specialization is not based on profiling and gathered statistics.

Trying to apply the techniques on increasingly reflective languages in which the user can tamper with increasingly essential features (e.g. via a meta-object protocol, or MOP [12]) eventually led to entirely run-time specialization; Sullivan introduces in [20] the theory of *dynamic partial evaluation*, which is specialization performed as a side effect of regular evaluation. To our knowledge this is the closest work

to ours because the specializer not only knows what set of values a given variable can take, but also which specific value it takes right now. (Sullivan does not seem to address run-time choice points in [20], i.e. how the multiple paths of a residual conditional expression are handled.)

Intermediate approaches for removing the interpretative overhead in specific reflective object-oriented languages can be found in [15] and [2]; however, both assume a limited MOP model.

Java has recently given just-in-time compilation much public exposure; Aycock [1] gives a history and references. Some projects (e.g. J3 [16] for Squeak [10]) aim at replacing an interpreter with a compiler within an environment that provides the otherwise unmodified supporting library. Throughout history, a number of projects (see [1]) offered the ability to use both the interpreter and the compiler at the same time, though considerable care was required to keep the interpreted and compiled evaluations synchronized (as was attempted by J2, the precursor of J3; [16] describes the related hassle).

Whaley [22] discusses compilation with a finer granularity than whole functions, which was previously thought to be impractical. His motivations are different: minimizing the compilation time and creating new optimization opportunities by focusing on hot regions within functions.

Low-level run-time code generation techniques that we have reused include lazy compilation of uncommon branches ([4], p. 123) and optimistic optimization using likely invariants, with guards in the generated code ([17]). More generally, run-time code generation has been investigated e.g. in [14] and [7]. Some of the results presented in these papers is likely to be of interest to future low-level optimizations by Psyco, although all frameworks considered so far (VCODE [7], GNU Lightning[22], LLVM [13]) focus on whole-function code generation and thus cannot be reused directly.

Specialization with respect to arbitrary properties was already investigated by Haraldsson [8] and formalized by Consel and Khoo [5] with a different presentation, organizing such properties into an algebra. In section 3 the emphasis is on representations as a way to link abstraction levels, and Psyco as a representation chooser.

Note that specialization in object-oriented languages often focuses on the selection of the correct method that a call must be dispatched to, depending on the type of the receiver of the call. While minimal efforts have been spent in Psyco on the object-oriented aspects, the basic effect of specializing for the receiver type is achieved by unlifting the type of each object out of which an attribute is looked up.[23] (See section 3.4 for more details about methods.)

5.2 Future Directions

More work is needed to improve object-oriented programs. An important missing feature is the ability to use long-lived custom representation for class instances. Python stores the set of attributes of each instance as a dictionary mapping attribute names to `PyObject` structures. To improve this, our goal is to use an instance storage format that varies dynamically, and group instances not only by class but by storage format. This format would be built by combining the representations of the individual values that are stored as attributes on the instance.

Another long-term goal is porting Psyco to other architectures and focusing on the efficiency of the generated code, probably starting with the ideas sketched in section 2.5.

It is probable that none of these goals will be implemented within the current Psyco prototype, because the research interest has shifted towards a platform called PyPy[24], a Python interpreter written in Python. One of the goals of the PyPy project is to provide a generic interpreter which can do both concrete and abstract (i.e. symbolic) interpretation. As the specializer in Psyco is itself a symbolic interpreter, we plan to regenerate it from PyPy in a more flexible way than the current incarnation in C.

The same techniques should be applied to different languages for comparison. Python was chosen because – apart from its practical interest – it defies the usual optimization techniques, notably by heavily overloading all operators and providing hard-to-control built-in dynamism.

The representation theory gives a natural way to map data between abstraction levels. Further developed, it might provide an integrated point of view on language-transformation processes like compilation and specialization and debugging and un-optimization.

5.3 Conclusion

In conclusion, we presented a novel "just-in-time specialization" technique motivated by a complete prototype. It differs from on-line specialization as follows:

- The top-down approach (2.3) introduces specialization-by-need based on the *unlift* operator.
- It introduces some low-level efficiency issues (2.5, 4.2) not present in on-line specialization.
- It prompts for a more involved "representation-based" theory of data-oriented specialization (3.1).
- Our approach makes specialization more tightly coupled with regular interpreters (3.5).

6. ACKNOWLEDGEMENTS

All my gratitude goes to the Python community as a whole for a great language that never sacrifices design to performance, forcing new and interesting optimization techniques to be developed.

The author also wishes to thank the anonymous referees for quite constructive comments.

7. REFERENCES

[1] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[2] M. Braux and J. Noyé. Towards partially evaluating reflection in java. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 2–11. ACM Press, 1999.

[3] C. Chambers. Staged compilation. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 1–8. ACM Press, 2002.

[22] http://www.gnu.org/software/lightning/lightning.html

[23] Methods in Python are just class attributes that happen to be callable.

[24] http://codespeak.net/pypy/

[4] C. D. Chambers. *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages*. PhD thesis, 1992.

[5] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.*, 15(3):463–493, 1993.

[6] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102. ACM Press, 1995.

[7] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 160–170. ACM Press, 1996.

[8] A. Haraldsson. *A program manipulation system based on partial evaluation*. PhD thesis, Linköping Univ., Sweden, Linköping Studies in Science and Technology Dissertations 14, 1977.

[9] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326, 1997.

[11] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[12] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[13] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[14] P. Lee and M. Leone. Optimizing ml with run-time code generation. *SIGPLAN Not.*, 39(4):540–553, 2004.

[15] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 300–315. ACM Press, 1995.

[16] I. Piumarta. J3 for squeak. http://www-sor.inria.fr/~piumarta/squeak/unix/zip/j3-2.6.0/doc/j3/.

[17] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *USENIX Association, editor, Computing Systems*, 1(Winter):11–32, 1988.

[18] A. Rigo. The ultimate psyco guide. http://psyco.sourceforge.net/psycoguide.ps.gz.

[19] M. Salib. Starkiller: A static type inferencer and compiler for python. In *Proceedings of the EuroPython conference*, 2004. http://web.mit.edu/msalib/www/writings/talks/.

[20] G. T. Sullivan. Dynamic partial evaluation. In *Lecture Notes In Computer Science, Proceedings of the Second Symposium on Programs as Data Objects*, pages 238–256. Springer-Verlag, 2001.

[21] G. van Rossum. Python reference manual. http://docs.python.org/ref/ref.html.

[22] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179. ACM Press, 2001.