Representation-based Just-in-time Specialization
and
the Psyco prototype for Python

Armin Rigo

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 dynamic languages, where the outcome of each single step can be highly dependent on run-time data. We introduce in the present paper two novel specialization techniques and discuss in particular their potential to close the performance gap between dynamic and static languages:

Just-in-time specialization, or *specialization by need,* introduces the "unlifting" ability for a value to be promoted from run-time to compile-time during specialization – the converse of the lift operator of partial evaluation. Its presence gives an unusual and powerful perspective on the specialization process.

Representations are a generalization of the traditional specialization domains, i.e. the compile-time/run-time dichotomy (also called static/dynamic, or "variables known at specialization time"/"variables only known at run time"). They provide a theory of data specialization.

These two techniques together shift some traditional problems and limitations of specialization. We present the prototype Psyco for the Python language.

1 Introduction

Most programming languages can be implemented by interpretation, which is a generally 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). Others require more involved techniques. We present in the following a technique at the intersection of *on-line partial evaluation* and *just-in-time compilation.*

Just-in-time compilation 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** is the specialization technique we will generally consider in the sequel: partial information about the variables and arguments of a program is propagated by abstractedly "evaluating", or interpreting, the program.

In the present paper we investigate the extra operational power offered by applying specialization at run time instead of compile time, a process which could be called **just-in-time specialization.** It sidesteps a number of common issues. For example, when specialization proceeds in parallel with the actual execution, it is guaranteed to terminate, and even not to incur more than a constant worse-case overhead. But the major benefit is that the specializer can "poll" the execution at any time to ask for actual values, or for some amount of information about actual values, which in effect **narrows** run-time values down to compile-time constants. We will argue throughout the present paper that this has deep implications: most notably, it makes specialization much less dependent on complex heuristics or detailled source code annotations to guide it.

1.1 Plan

- Section 1: introduction.
- Section 2: just-in-time specialization. By entierely mixing specialization and execution, we obtain a technique that leads to the use of run-time values at compile-time in an on-line specializer.
- Section 3: representation theory. It is a flexible formalization generalizing the classical compile-time/run-time dichotomy, to match the needs of section 2.
- Section 4: putting the pieces together.
- Appendix A: the Psyco prototype for Python.

Sections 2 and 3 can be read independently.

1.2 Background

The classical presentation of specialization 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 archetypical application is if $interp(source, input)$ is an interpreter, where *source* is the source code of the program to interpret and *input* the input variables for the interpreted program. In this case, the function $interp_1(input)$ can be considered as the compiled version of the corresponding source code $source_1$. The interpretative overhead can indeed be statically compiled away if $source_1$ is fixed.

Depending on context, this technique is commonly subdivided into *on-line* and *off-line* specialization. If the set of values $x_1, x_2, x_3 \ldots$ is statically known, the functions $f_1, f_2, f_3 \ldots$ can be created in advance by a source-to-source transformation tool. This is **off-line** specialization. For example, in a program using constant regular expressions to perform text searches, each static regular expression $regexp_n$ can be translated into an efficient matcher $match_n(string)$ by specializing the general matcher $match(regexp, string)$.

If, on the other hand, the regular expressions are not known in advance, e.g. because they are given to the program as a command-line argument, then we can still use **on-line** specialization to translate and optimize the pattern at the beginning of the execution of the program. (Common regular expression engines that pre-compile patterns at run-time can be considered as a hand-written version of the specialization of a generic regular expression interpreter.)

In on-line specialization, the time spent specializing is important because the process occurs at run-time. In this respect on-line specialization is a form of just-in-time compilation, particularly when it is hand-crafted to directly produce lower-level code instead of code in the same high-level language as the source.

1.3 Compile-time and run-time values

The notation $f(x, y)$ hides a major difficulty of both off-line and on-line specialization: the choice of how exactly to divide the arguments into the compile-time (x) and the run-time (y) ones. The same problem occurs for the local variables and the function calls found in the definition of f.

In some approaches the programmer is required to annotate the source code of f. This is a typical approach if f is a not-excessively-large, well-known function like an interpreter *interp* for a specific language. The annotations are used by the specializer to constant-propagate the interpretation-related computations at compile-time (i.e. during the translation of *interp* into a specialized $interp_1$), and leave only the "real" computations of the interpreted program for the run-time (i.e. during the execution of $interp_1$).

In other approaches, many efforts are spent trying to automatically derive this categorization compile-time/run-time from an analysis of the source code of *interp*.

However, consider a function call that might be identifiable as such in the source, but where the function that is being called could be an arbitrary object whose constantness[1] cannot be guaranteed. The call can thus not be specialized into a direct call. Some overhead remains at run-time, and the indirect call prevents further cross-function optimizations. Even more importantly, if the basic operations are fully polymorphic, even a simple addition cannot be specialized into a processor integer addition: the actual operation depends on the dynamic run-time classes of the variables. Actually, even the classes themselves might have been previously tampered with.

For the above examples, one could derive by hand a more-or-less reasonable categorization, e.g. by deciding that the class of all the objects must be compile-time, whereas the rest of the objects' value is run-time. But one can easily construct counter-examples in which this (or any other) categorization is suboptimal. Indeed, in specialization, an efficient result is a delicate balance

[1] In object-oriented languages, even its class could be unknown.

between under-specialization (e.g. failure to specialize a call into a direct call if we only know at compile-time that the called object is of class "function") and over-specialization (e.g. creating numerous versions of a function which are only slightly or even not better at all than the more general version).

1.4 Contribution of the present paper

In our approach, specialization is entierely performed at run-time; in particular the categorization compile-time/run-time itself is only done during the execution. Starting from this postulate, our contributions are:

- The specialization process is not done at the function level, but at a much finer-grained level,[2] which allows it to be deeply intermixed with actual execution.
- Specialization can query for actual run-time values, a process which is effectively the converse of the lift operator (section 2.1).
- Specialization is not only based on types, i.e. subdomains of the value domains, but on which representations are choosen to map the domains. For example, we can specialize some code for particular input values, or only for particular input types; in the latter case, the way run-time information represents a value within the allowed domain can itself vary (section 3).

The most important point is that using the just-in-time nature of the approach, i.e. the intermixed specialization and execution processes, we can perform specialization that uses feed-back from run-time values in a stronger way than usual: values can be promoted from run-time to compile-time. In other words, we can just use actual run-time values directly while performing specialization. This kind of feed-back is much more fine-grained than e.g. statistics collected at run-time used for recompilation.

1.5 Related work

The classical reference for efficient execution of dynamic programming languages is the implementation of Self [C92], which transparently specializes functions for specific argument types using statistical feed-back. A number of projects have followed with a similar approach, e.g. [D95] and [V97].

Trying to apply the techniques on increasingly reflective languages in which the user can tamper with ingreasingly essential features (e.g. via a meta-object protocol, or MOP [K91]) eventually led to entierely run-time specialization; Sullivan introduces in [S01] 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 does not only know what set of values a given variable can take, but also which specific value it takes

[2] It is not the level of basic blocks; the boundaries are determined dynamically according to the needs of the specializer.

right now. (Sullivan does not seem to address run-time choice points in [S01], i.e. how the multiple paths of a residual conditional expressions are handled.)

Intermediate approaches for removing the interpretative overhead in specific reflective object-oriented languages can be found in [M98] and [B00]; however, both assume a limited MOP model.

Java has recently given just-in-time compilation much public exposure; Aycock [A03] gives a history and references. Some projects (e.g. J3 [Piu] for Squeak [I97]) 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 [A03]) offered the ability to complementarily use both the interpreter and the compiler, thought considerable care was required to keep the interpreted and compiled evaluations synchronized (as was attempted by J2, the precursor of J3; [Piu] describes the related hassle).

Whaley [W01] discusses compilation with a finer granularity than whole functions.

Low-level code generation techniques include lazy compilation of uncommon branches ([C92], p. 123) and optimistic optimization using likely invariants, with guards in the generated code ([P88]).

2 Just-in-time specialization

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

2.1 The Unlift operator

Assume that the variables in a program have been classified into compile-time and run-time variables. During specialization, it is only possible to make use of the compile-time[3] part of the values. Their run-time part is only available later, during execution. This is traditional in specialization: the *amount* of information available for the specializer is fixed in advance, even if what this information might actually be is not, in the case of on-line specialization. As an extreme example, [C02] describes a multi-stage compilation scheme in which gradually more information (and less computational time) is available for optimization while the system progresses towards the later stages.

The restriction on *what* information is expected to be present at all at a given stage places a strong global condition on the compile-time/run-time classification of a program. There are cases where it would be interesting to gather compile-time (i.e. early) information about a run-time value. This operation is essential; in some respect, it is what on-line specializers implicitely do when

[3]"Compile-time" could be more specifically called "specialization-time" when doing specialization, but the border between compiling and specializing is fuzzy.

they start their job: they take an input (run-time) value, and start generating a version of the source specialized for this (now considered compile-time) value.

Let us make this operation explicit. We call it **unlift**, as it is effectively the converse 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 of a greater generality of the residual code. Althought the possibility of unlift is not often considered, it does not raise numerous problems. By comparison, the common problems found in most forms of on-line specialization (see section 2.4) are much more difficult.

The technique to read a run-time value from the specializer is best explained with explicit continuations: when a run-time value is asked for, the specializer is suspended (we capture its state in a continuation); and residual code is emitted that will resume the specializer (by invoking the continuation) with the run-time value. In other words, specialization is not simply guided by run-time feed-back; it is literally controlled by the run-time, and does not take place at all (the continuation remains suspended) before these run-time values actually show up.

2.2 The top-down approach

Unlifting makes specialization and execution much more intermixed in time than even on-line specialization, as we will see on an example in section 2.3. We call this particular technique **just-in-time specialization.** Interestingly, unlifting seems to lessen the need for termination analysis or widening heuristics.

The reason behind the latter claim is that 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 (as we would have to do 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.

2.3 Example

Consider the following function:

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

As discussed in section 2.2 we will enter the specializer with the most general case: nothing is known about the input argument n. Figure 1 shows how specialization and execution are intermixed in time in this top-down approach. Note that specialization only starts when the first actual (run-time) call to f takes place.

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: read the value into a register, write code that adds 1.
execute the addition machine instruction, 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* $-$	

Figure 1: Mixed specialization and execution

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".

In fact, each excursion through the right column is compiled into a single conditional jump in the left column. For example, an "overflow?" question corresponds to 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 no longer goes through 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. This results in a different code path, which does not replace the previously-generated code but completes it. When invoked, the specializer 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 is (as before) the first version of the code, but the overflowing case now points to the new code.

As another example, say that f is later called with a floating-point value. Then new code will be compiled, that will fork away from the existing code at the first question, "what is the type of n?". After this additional compilation, the patched processor instruction at that point is a three-way jump[4]: 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 to the specializer.

2.4 Issues with just-in-time specialization

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

This cache is not only used on function entry points, but also at the head of loops in the function bodies, so that we can detect when specialization is looping back to an already-generated case. The bottom-up approach of traditional on-line specialization requires widening (when too many different compile-time values have been found at the same source point, they are tentatively generalized) to avoid generating infinitely many versions of a loop or a function. The top-down specialization-by-need approach of just-in-time specialization might remove the need for widenening, although more experimentation is needed to settle the question (the Psyco prototype does some widening which we have not tried to remove so far).

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 — even forever, a priori. In the above example, 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 seen above, only a few machine instructions can typically be generated before the specializer must give the control back to execution, and often this immediately executes the instructions just produced. This defies common compiler optimization techniques like register allocation. Care must also be taken to keep some code locality: processors are not good

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

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. It would then be completed with a better compiler that is able to re-read it 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 (described in [A03]).

The Psyco prototype currently implements a subset of these possible techniques, as described in section A.3.

3 Representation-based specialization

This section introduces a formalism to support the process intuitively described above; more specifically, how we can represent partial information about a value, e.g. as in the case of the input argument n of the function $f(n)$ in 2.3, which is promoted from run-time to "known-to-be-of-type-`int`".

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 the values in X, or at least some of them (the ones that are in the image of r), to be "represented" by an element of X'. An $x' \in X'$ represents the value $r(x') \in X$. As an example, the domain X' could be a subtype of X, r being just the inclusion. Here is a different example: say X is the set of all first-class objects of a programming language, and X' is the set of machine-sized words. Then r could map a machine word to the corresponding integer object in the programming language, a representation which is often not trivial (because the interpreter or the compiler might associate meta-data to integer objects).

The two extreme examples of representations 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**[5] *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 $x' \in X'$:*

$$\begin{tikzcd} X \arrow[r, "f"] & Y \\ X' \arrow[u, "r"] \arrow[r, "f'"] & Y' \arrow[u, "s"'] \end{tikzcd}$$

r is called the **argument representation** *and s the* **result 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 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 (say, unsigned) machine words. This example also shows how representation 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 all $r \in R$ into representations of f with argument r:*

$$\begin{tikzcd} X \arrow[r, "f"] & Y \\ X' \arrow[u, "r"] \arrow[r, "S_f(r)"] & Y' \arrow[u, "s"'] \end{tikzcd}$$

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 result representation of Y.

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

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 in 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 couple (w, x'). Call X the type of all such couples. 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:

$$\begin{tikzcd} \mathrm{W} \times \mathrm{X'} \arrow[r, "g"] & \mathrm{Y} \\ \mathrm{X'} \arrow[u, "r_w"] \arrow[r, "S_g(r_w)"] & \mathrm{Y} \arrow[u] \end{tikzcd}$$

$S_g(r_w)(x') = g(r_w(x')) = g((w, x'))$, so that $S_g(r_w)$ is the specialized function $g((w, -))$.[6] 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 $r_w = c_w \times id_{X'}$.[7]

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

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

[6]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.

[7]We will systematically identify $\{\cdot\} \times X$ with X.

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, that function does nothing and returns nothing, but it is a nothing 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 Application

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 instead of introducing some more flexibility.

Consider a small language with constructors for *integers, floats, tuples,* and *strings.* The variables are untyped and can hold a value of any of these four (disjoint) types. The "type" of these variables is thus the set X of all values of all four types.[8]

```
def f(x, y):
    u = x + y
    return (x, 3 * u)
```

Addition and multiplication are polymorphic (tuple and string addition is concatenation, and $3 * u = u + u + u$).

We will try to compile this example to low-level C code. The set R of representations will closely follow the data types. It is built recursively and contains:

- the constant representations c_x for any value x;

[8] The syntax we use is that of the Python language, but it should be immediately obvious.

- the integer representations $r_{i_1}, r_{i_2}, \ldots$ where $\texttt{i1}, \texttt{i2}, \ldots$ are C variables of type `int` (where r_{i_n} means that the value is an integer found in the C variable called $\texttt{i}n$);
- the float representations $r_{f_1}, r_{f_2}, \ldots$ where $\texttt{f1}, \texttt{f2}, \ldots$ are C variables of type `float`;
- the string representations $r_{s_1}, r_{s_2}, \ldots$ where $\texttt{s1}, \texttt{s2}, \ldots$ are C variables of type `char*`;
- the tuple representations $r_1 \times \ldots \times r_n$ for any (previously built) representations $r_1, \ldots, r_n$.

The tuple representations allow information about the items to be preserved across tupling/untupling; it represents each element of the tuple independently.

Assuming a sane definition of addition and multiplication between representations, we can proceed as in section 3.3. For example, if the above f is called with the representation $r_{s_1} \times r_{s_2}$ it will generate C code to concatenate and repeat the strings as specified, and return the result in two C variables, say `s1` and `s4`. This C code is a representation of the function f; its resulting representation is $r_{s_1} \times r_{s_4}$. If f had been called with $r_{i_1} \times r_{f_1}$ instead it would have generated a very different C code, resulting in a representation like $r_{i_1} \times r_{f_3}$.

The process we roughly described defines an R-specializer S_f: if we ignore type errors for the time being, then for any representation $r \in R$ we can produce an efficient representation $S_f(r)$ of f. Also, consider a built-in operation like $+$. We have to choose for each argument representation a result representation and residual C code. This choice is itself naturally described as a built-in R-specializer S_+: when the addition is called with an argument in a specific representation (e.g. $r_{i_1} \times r_{i_2}$), then the operation can be represented as specified by S_+ (e.g. $S_+(r_{i_1} \times r_{i_2})$ would be the low-level code `i3 = i1+i2;`) and the result is in a new, specific representation (e.g. r_{i_3}).

In other words, the compiler can be described as a symbolic interpreter over the abstract domain R, with rules given by the specializers. It starts with predefined specializers like S_+ and then, recursively, generates the user-defined ones like S_f.

3.5 Integration with an interpreter

The representations introduced in section 3.4 are not sufficient to be able to compile arbitrary source code (even ignoring type errors). For example, a multiplication `n*t` between an unknown integer (e.g. r_{i_1}) and a tuple returns a tuple of unknown length, which cannot be represented within the given R.

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 R. This slight change suddenly makes the compiler tightly integrated with a regular interpreter. Indeed, this most general representation stands for an arbitrary value whose type is not known at compile-time. This representation is very pervasive: typically, operations involving it produce a result that is also represented by id_X.

A function "compiled" with all its variables represented as id_X is inefficient: it still contains the overhead of decoding 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 assume that a regular interpreter is already available for the language. Then the introduction of id_X provides a safe "fall-back" behavior: the compiler cannot fail; at worst it falls back to interpreter-style dispatching. This is an essential property if we consider a much larger programming language than described above: some interpreters are even dynamically extensible, so that no predefined representation set R can cover all possible cases unless it contains id_X.

A different but related problem is that in practice, a number of functions (both built-in and user-defined) 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 2.3 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" for $S_+(r_{i_1} \times r_{i_2})$; if an overflow is detected we fork the exceptional branch using a more general representation $S_+(r) = + : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Generalization cannot fail: in the worst case we can use the fall-back representation $S_+(id_X \times id_X)$. (This is similar to recent successful attempts at using a regular interpreter as a fall-back for exceptional cases, e.g. [W01].)

4 Putting the pieces together

Sections 2 and 3 are really the two sides of the same coin: any kind of behavior using id_X as a fall-back (as in 3.5) raises the problem of the pervasiveness of id_X in the subsequent computations. This was the major motivation behind section 2: just-in-time specialization enables "unlifting".

Recall that to *lift* is to move a value from compile-time to run-time; in term of representation, it means that we change from a specific representation (e.g. c_{42}) to a more general one (e.g. r_{i_1}, the change being done by the C code `i1 = 42;`). Then *unlifting* is a technique to solve the pervasiveness problem by doing the converse, i.e. switching from a general representation like id_X to a more specific one like r_{i_1}. We leave as an exercice to the reader the reformulation of the example of section 2.3 in terms of representations.

4.1 Changes of representation

Both lifting and unlifting are instances of the more general *change of representation* kind of operation. In the terminology of section 3, a change of representation is a representation of an identity, i.e. some low-level code that has no high-level effect:

$$\begin{tikzcd} X \arrow[r, "id"] & X \\ X_1 \arrow[u, "r_1"] \arrow[r, "g"] & X_2 \arrow[u, "r_2"'] \end{tikzcd} \quad \text{or equivalently} \quad \begin{tikzcd} & X & \\ X_1 \arrow[ur, bend left, "r_1"] \arrow[rr, "g"] & & X_2 \arrow[ul, bend right, "r_2"'] \end{tikzcd}$$

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. Unlifts are partial representations of the identity. As in 3.5, run-time values may later show up that a given partial representation cannot handle, requiring re-specialization.

4.2 Conclusion

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

- The top-down approach (2.2) introduces specialization-by-need as a promizing alternative to the widening heuristics based on the *unlift* operator.
- It introduces some low-level efficiency issues (2.4, A.3) not present in on-line specialization.
- It prompts for a more involved "representation-based" theory of value management (3.1), which is in turn more powerful (3.4) and gives a natural way to map data between abstraction levels.
- Our approach makes specialization more tightly coupled with regular interpreters (3.5).

The prototype is described in appendix A.

4.3 Acknowledgements

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

A Psyco

In the terminology introduced above, Psyco[9] is a *just-in-time representation-based specializer* operating on the Python[10] language.

A.1 Overview

The goal of Psyco is to transparently accelerate the execution of user Python code. It is not an independent tool; it is an extension module, written in C, for the standard Python interpreter.

Its basic operating technique was described in section 2.3. It 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 2.

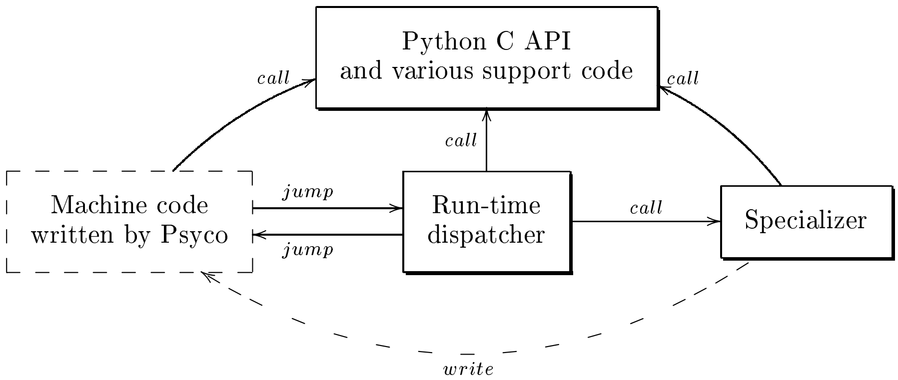

Figure 2: The architecture of Psyco

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 doesn't want to specialize. It is also continously 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.

- The *specializer* is a symbolic Python interpreter: it works by interpreting Python bytecodes with representations instead of real values (see section

[9]http://psyco.sourceforge.net
[10]http://www.python.org

3.4). This interpreter is not complete: it only knows about a subset of the built-in types, for example. But it does not matter: for any missing piece, it falls back to universal 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 in the example of 2.3.)
- 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.

A.2 Representations

The **representations** in Psyco are implemented using a recursive data structure called `vinfo_t`. These representations closely follow the C implementation of the standard Python interpreter. Theoretically, they are representations of the C types manipulated by the interpreter (as in section 3.3). However, we use them mostly to represent the data structure `PyObject` that implements Python language-level objects.

There are three kinds of representation:

1. *compile-time,* representing a constant value or pointer;
2. *run-time,* representing a value or pointer stored in a specific processor register;
3. *virtual-time,* a generic name[11] for a family of custom representations of `PyObject`.

Representations of pointers can optionally specify the sub-representations of the elements of the structure they point to. This is used mostly for `PyObject`. A run-time pointer A to a `PyObject` can specify additional information about the `PyObject` it points to, e.g. that the Python type of the `PyObject` is `PyInt_Type`, and maybe that the integer value stored in the `PyObject` has been loaded in another processor register B. In this example, the representation of the pointer to the `PyObject` is

$$r_A[c_{int}, r_B]$$

where:

[11] The name comes from the fact that the represented pointer points to a "virtual" `PyObject` structure.

- c_{int} is the representation of the constant value "pointer to `PyInt_Type`";
- r_A and r_B are the run-time representations for a value stored, respectively, in the registers A and B;
- the square brackets denote the sub-representations.

Sub-representations are also used for the custom (virtual-time) representations. For example, the result of the Python addition of two integer objects is a new integer object. We must represent the result as a new `PyIntObject` structure (an extension of the `PyObject` structure), but as long as we do not need the exact value of the pointer to the structure in memory, there is no need to actually allocate the structure. We use a custom representation v_{int} for integer objects: for example, the (Python) integer object whose numerical value is in the register B can be represented as $v_{int}[r_B]$. This is a custom representation for "a pointer to some `PyIntObject` structure storing an integer object with value r_B".

A more involved example of custom representation is for string objects subject to concatenation. The following Python code:

```
s = ''              # empty string
for x in somelist:
    s = s + f(x)    # string concatenation
```

has a bad behavior, quadratic in the size of the string `s`, because each concatenation copies all the characters of `s` into a new, slightly longer string. For this case, Psyco uses a custom representation, which could be[12] $v_{concat}[str1, str2]$, where $str1$ and $str2$ are the representations of the two concatenated strings.

Python fans will also appreciate the representation $v_{range}[start, stop]$ which represents the list of all number from *start* to *stop*, as so often created with the `range()` function:

```
for i in range(100, 200):
    ...
```

Whereas the standard interpreter must actually create a list object containing all the integers, Psyco does not, as long as the v_{range} representation is used as input to constructs that know about it (like, obviously, the `for` loop).

A.3 Implementation notes

Particular attention has been paid to the **continuations** underlying the whole specializing process. Obviously, being implemented in C, we do not have general-purpose continuations in the language. However, in Psyco it would very probably prove totally impractical to use the powerful general tools like Lisp or

[12]For reasons not discussed here, the representation used in practice is different: it is a pointer to a buffer that is temporarily over-allocated, to make room for some of the next strings that may be appended. A suitable over-allocation strategy makes the algorithm amortized linear.

Scheme continuations. The reason is the memory impact, as seen in section 2.4. It would not be possible to save the state of the specializer at all the points where it could potentially be resumed from.

Psyco emulates continuations by saving the state only at some specific positions, which are always between the specialization of two opcodes (pseudo-code instructions) – and not between any two opcodes, but only between carefully selected ones. The state thus saved is moreover packed in memory in a very compact form. When the specializer must be resumed from another point (i.e. from some precise point in the C source, with some precise local variables, data structures and call stack) then the most recent saved state before that point is unpacked, and execution is replayed until the point is reached again. This recreates almost exactly the same C-level state as the last time we reached the point.

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. Actually, the prototype allocates registers in a round-robin fashion and tries to minimize memory loads and stores, but performs few other optimizations. It also tries to keep the code blocks close in memory, to improve the processor cache hits.

Besides the Intel i386-compatible machine code, Psyco has recently be "ported" to a custom **low-level virtual machine architecture.** This architecture will be described in a separate paper. It could be used as an intermediate code for two-stage code generation, in which a separate second stage compiler would be invoked later to generate and agressively optimize native code for the most heavily used code blocks.

The **profiler hooks** in Psyco select the functions to specialize based on an "exponential decay" weighting algorithm, also used e.g. in Self [H96]. 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 universal representation of the current (interrupted) interpreter position (i.e. the representation in which nothing specific is known about the objects), and starting the specializer from there.

In its current incarnation, Psyco uses a mixture of **widening, lifting and unlifting** that may be overcomplicated. To avoid infinite loops in the form of a representation being unlifted and then widened again, the compile-time representations are marked as *fixed* when they are unlifted. The diagram of figure 3 lists all the state transitions that may occur in a `vinfo_t`.

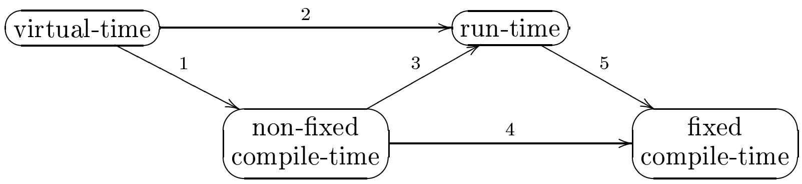

Figure 3: State transitions in Psyco: widening (3), unlifting (5) and other representation changes (1, 2, 4)

A.4 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, extremely 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.

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 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,[13] though not representative at all of the Python programming style.
- **ZPT:** Zope Page Template, an HTML templating language interpreted in Python. Zope 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).

[13] Available in `Lib/test/pystone.py` in the Python distribution.

- **PyPy 2:** Second part (object library implementation).

The results (figure 4) have been obtained on a Pentium III laptop at 700MHz with 64MB RAM. Times are seconds per run. Numbers in parenthesis 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, which tends to give better results on small examples.

Benchmark	Python (2.3.3)	Psyco	C (gcc 2.95.2)
int arithmetic	28.5	0.262 (109×)	0.102 (281×)
			ovf:[14] 0.393 (73×)
float arithmetic	28.2	2.85 (9.9×)	0.181 (156×)
complex arithmetic	19.1	7.24 (2.64×)	0.186 (102×)
			sqrt:[15] 0.480 (40×)
files and lists	20.1	1.45 (13.9×)	0.095 (211×)
Pystone	19.3	3.94 (4.9×)	
ZPT	123	61 (2×)	
PyPy 1	5.27	3.54 (1.49×)	
PyPy 2	60.7	59.9 (1.01×)	

Figure 4: Timing the performance improvement of Psyco

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 very 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.

The present prototype moreover requires some tweaking to give good results on non-trivial examples, as described in section 2.2 of [R03].

More benchmarks comparing the Psyco-accelerated Python with other languages have been collected and published on the web
(e.g. http://osnews.com/story.php?news_id=5602).

[14] Although no operation in this test overflows the 32-bit words, both Python and Psyco systematically check for it. The second version of the equivalent C program also does these checks (encoded in the C source). Psyco is faster because it can use the native processor overflow checks.

[15] This second version extracts the square root to check if the norm of a complex number is greater than 2, which is what Python and Psyco do, but we also included the C version with the obvious optimization because most of the time is spent there.

References

[A03] John Aycock. *A Brief History of Just-In-Time.* ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 97-113.

[B00] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in Java. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM-00),* pages 2–11, N.Y., January 22-23 2000. ACM Press.

[C92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Computer Science Departement, Stanford University, March 1992.

[C02] Craig 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.

[D95] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI),* pages 93–102, La Jolla, California, 18-21 June 1995. *SIGPLAN Notices* 30(6), June 1995.

[H96] Urs Holzle and David Ungar. *Reconciling responsiveness with performance in pure object-oriented languages.* ACM Transactions on Programming Languages and Systems, 18(4):355–400, July 1996.

[I97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97,* pages 318–326, November 1997.

[K91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[M98] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications,* pages 300–315. ACM Press, 1995.

[Piu] Ian Piumarta. *J3 for Squeak.* http://www-sor.inria.fr/~piumarta/squeak/unix/zip/j3-2.6.0/doc/j3/

[P88] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. In USENIX Association, editor, *Computing Systems, Winter, 1988.,* volume 1, pages 11–32, Berkeley, CA, USA, Winter 1988. USENIX.

[R03] Armin Rigo. *The Ultimate Psyco Guide.* http://psyco.sourceforge.net/psycoguide.ps.gz

[S01] Gregory T. Sullivan. Dynamic Partial Evaluation. In *Lecture Notes In Computer Science, Proceedings of the Second Symposium on Programs as Data Objects,* pp. 238-256, Springer-Verlag, London, UK, 2001.

[V97] Engen N. Volanschi, Charles Consel, and Crispin Cowan. *Declarative specialization of object-oriented programs.* In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97), volume 32, 10 of ACM SIGPLAN Notices, pages 286–300, New York, October 5-9 1997. ACM Press.

[W01] John Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems, Languages, and Applications,* October 2001, pages 166–179, Tampa Bay, FL, USA. ACM Press.