# Unipycation: A Case Study in Cross-Language Tracing

Edd Barrett Carl Friedrich Bolz Laurence Tratt

Software Development Team, Informatics, King's College London

http://eddbarrett.co.uk/ http://cfbolz.de/ http://tratt.net/laurie/

## Abstract

Language composition approaches have traditionally suffered from poor performance. In this paper we hypothesise that meta-tracing provides a means to compose independent language interpreters while retaining the performance levels of each. To study this approach, we compose Python and Prolog interpreters to form Unipycation. We present a case study of its use and a suite of micro-benchmarks which give us some understanding of its cross-language performance.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—code generation, incremental compilers, interpreters, run-time environments

## 1. Introduction

Traditionally, most software projects have used a single programming language to implement all their aspects. While this often works well, it can also be frustrating, forcing some aspects to be written in less than ideal languages. We believe that this compromise is inevitable when software developers can only easily write systems in a single language.

Language composition allows users to mix languages (programming languages and / or domain specific languages) together in a fine-grained manner, so that each part of a problem can be expressed using the most appropriate language. The need for this has been articulated, in different ways, since at least the late 60s (e.g. [12, 22, 27]). While several approaches have tackled parts of the vision (e.g. [9, 10, 24, 29]), no approach can be said to have caught on. There are many reasons for this, but in this paper we focus on one: the difficulty of creating an efficient composition of language run-times.

At the moment, a language composition implementer has only one realistic choice: to translate each language in the composition down to a single base language (e.g. C, JVM bytecode). If all the languages in the composition are broadly similar, this can work well. In most cases, however, the languages have sufficiently different semantics, meaning that optimising them all simultaneously is impossible. Such a situation is a *semantic mismatch* in language implementation terms [7]. Examples include dynamically typed languages hosted upon HotSpot. Despite HotSpot's wonderful performance on statically typed languages, and despite the introduction of `invokedynamic`, translating such languages to JVM bytecode often leads to disappointing performance. Jython – Python on the JVM – is generally slower than CPython – the 'classic' interpreter-only version of Python.

We hypothesise that this long-standing problem can be overcome by composing together meta-tracing interpreters. Meta-tracing allows one to write an interpreter from which a tracing JIT compiler is automatically generated. The performance of the resulting VMs significantly outperforms interpreter-only VMs [7]—PyPy, a meta-tracing Python VM is significantly faster than CPython (and, by extension, Jython). Composing together meta-tracing compatible interpreters such that their individual performance remains unaffected is relatively easy. The challenge is to have the two interpreters interact – to exchange data, call functions, and so on – and to do so in an efficient manner.

In this paper we present Unipycation, a simple composition of PyPy – a fast Python VM – with Pyrolog [6] – a fairly fast Prolog VM – as a way of beginning to test our thesis. We chose Python and Prolog not only because we have existing interpreters for them, but because they have substantially different semantics. If our hypothesis can be validated, it will only be by picking tricky cases such as this. Unipycation is a concrete example of interpreter composition that is intended to highlight the issues involved. Many challenges remain untackled: Unipycation, for example, currently defers issues of syntactic composition to other projects. We do not pretend to have all the solutions yet—the only thing we are sure of is that we have not yet identified all of the problems.

The contributions of this paper are as follows:

- We present a prototype of Unipycation, the first composition of meta-tracing interpreters.
- We evaluate Unipycation's suitability for writing composed programs via a small case study.

[Copyright notice will appear here once 'preprint' option is removed.]

- We evaluate the performance of Unipycation with a suite of micro-benchmarks.

This paper is structured as follows. After discussing preliminaries we present a case study of a Connect Four implementation with a Python GUI and a Prolog game-playing engine as a way of demonstrating that Unipycation's composition is indeed usable (Section 4). We then show the design of Unipycation, including the APIs it exposes to allow Python and Prolog code to call each other (Section 5). We then present 7 synthetic language composition benchmarks, each of which has 4 further variations, to understand the performance of cross-language interactions in Unipycation (Section 6). Finally, we discuss our subjective experiences of Unipycation and possible next routes (Section 8).

For full repeatability, Unipycation, the case study, and the benchmarks can be downloaded from:

`http://soft-dev.org/pubs/files/unipycation/`

## 2. Meta-tracing

Meta-tracing takes an interpreter and creates from it a tracing JIT compiler [2, 5, 28, 32]. The resulting VM contains both the interpreter and the tracing JIT compiler. At run-time, user programs running in the VM begin their execution in the interpreter. When a 'hot loop' in the user program is encountered, the actions of the interpreter are traced (i.e. recorded), optimised, and then converted to machine code. Subsequent executions of the loop then use the fast machine code version rather than the slow interpreter. Guards are left behind in the machine code so that if execution needs to diverge from the path recorded by the trace, execution can safely fall back to the interpreter.

Due to the particular nature of interpreters, meta-tracing is able to create tracing JIT compilers automatically. Whether it operates on bytecode or ASTs, an interpreter is fundamentally a large loop: 'load the next instruction, perform the associated actions, go back to the beginning of the loop'. To generate a tracing JIT, the language implementer annotates the interpreter to inform the meta-tracing system when a loop[1] at position *pc* (program counter) has been encountered; the meta-tracing system then decides if the loop has been encountered often enough to start tracing. The annotation also tells the meta-tracing system that execution of the program at position *pc* is about to begin and that if a machine code version is available, it should be used; if it is not, then the standard interpreter will be used. Although tracing is not a new idea (see [1, 18]), traditional implementations required manually creating both the interpreter and trace compiler, whereas meta-tracing fully automates the latter.

The main extant meta-tracing language is RPython, a statically-typed subset of Python which translates to C. RPython's type system is similar to Java's, extended with further analysis e.g. to assure that list indices are not negative. Users can influence the analysis with `assert` statements, but otherwise it is fully automatic. Unlike seemingly similar languages (e.g. Slang [21] or PreScheme [25]), RPython is more than just a thin layer over C: it is, for example, fully garbage collected and has several high-level datatypes (e.g. lists and dictionaries).

[1] Loops are often, though not exclusively, program counter jumps with a negative index.

## 3. Prolog background

We assume that most readers have a working knowledge of a mainstream 'imperative' language such that they can understand the simple uses of Python used in this paper. However, we can not reasonably make the same assumption about Prolog. This section serves as a brief introduction to Prolog for unfamiliar readers (see e.g. [8] for more details). Those familiar with Prolog will notice a distinct imperative flavour to our explanations. This is intentional, given the paper's likely audience, but nothing we write should be considered as precluding the logic-based view of Prolog.

Prolog is a rule-based logic programming language whose programs consist of a database of predicates which is then queried. A predicate is related to, but definitely not the same as, the traditional programming language concept of a function. Predicates can be loosely thought of as overloaded pattern-matching functions that can generate a stream of solutions (including no solutions at all). Given a database, a user can then query it to ascertain the truth of an expression.

Prolog supports the following data types:

**Numeric constants** Integers and floats.

**Atoms** Identifiers starting with a lowercase letter e.g. `chair`.

**Terms** Composite structures beginning with a lowercase letter e.g. `vector(1.4, 9.0)`. The name (e.g. `vector`) is the term's *functor*, the items within parantheses its *arguments*. The arguments of a term should not be confused with the arguments of a function; terms can be thought of as named tuples, where the arguments are merely data.

**Variables** Identifiers beginning with either an uppercase letter (e.g. `Person`) or an underscore. The variable `_` is the *anonymous variable*. Variables are not just named storage places, as in imperative languages, but denote unknown values that may become known later. They can occur at any point within a data structure. For example the term `vector(31, Y)` has a second argument whose value is not yet known. A concrete value can be *bound* to such a variable later.

**Lists** Lists are made out of cons cells, which are simply terms of the functor `'.'`. Since lists are common, and the `'.'` syntax rather verbose, lists can be expressed using a comma separated sequence of elements enclosed inside square brackets. For example, the list `[1,2,3]` is equivalent to `'.'(1, '.'(2, '.'(3, [])))`. Further, a list can be denoted in terms of its *head* and *tail*, for example `[1 | [2, 3]]` is equivalent to `[1,2,3]`. A list

with a variable as its tail, such as `[1 | X]`, is called a *partial list*.

To demonstrate some of these concepts, consider the Prolog rule database shown in Listing 1. The `edge` predicate describes a directed graph which may, for example, represent a transit system such as the London Underground. The `path` predicate accepts four arguments and describes valid paths of length `MaxLen` or under, from the node `From` to the node `To`, as the list `Nodes`. In this example, `a`, `b`, ..., `g` are atoms. The expression `edge(a, c)` defines a predicate called `edge` which is true when the arguments `a` and `c` are passed. The expression `path(From, To, MaxLen, Nodes, 1)` is a call to the `path` predicate passing as arguments, four variables and a integer constant. Finally, `[From | Ahead]` is a partial list.

```
1  edge(a, c). edge(c, b). edge(c, d). edge(d, e).
2  edge(b, e). edge(c, f). edge(f, g). edge(e, g).
3  edge(g, b).
4
5  path(From, To, MaxLen, Nodes) :-
6      path(From, To, MaxLen, Nodes, 1).
7
8  path(Node, Node, _, [Node], _).
9  path(From, To, MaxLen, [From | Ahead ], Len) :-
10     Len < MaxLen, edge(From, Next),
11     Len1 is Len + 1,
12     path(Next, To, MaxLen, Ahead, Len1).
```

**Listing 1.** A Prolog rule database.

Queries can either succeed or fail. For example, running the query `edge(c, b)` ("is it possible to transition from node `c` to node `b`"?) against the above database succeeds, but `edge(e, f)` ("is it possible to transition from node `e` to node `f`?") fails. When a query contains a variable, Prolog searches for solutions, binding values to variables. For example, the query `edge(f, Node)` ("which node can I transition to from `f`?") binds `Node` to the atom `g`. Queries can produce multiple solutions. For example, `path(a, g, 7, Nodes)` ("Give me paths from `a` to `g` of maximum length 7") finds several bindings for `Nodes`: `[a, c, b, e, g]`, `[a, c, d, e, g]`, `[a, c, f, g]`, and `[a, c, f, g, b, e, g]`.

Solutions are enumerated by recording *choice points* where more than one rule is applicable. If a user requests another solution, or if an evaluation path fails, Prolog backtracks to the most recent choice point and explores alternative search paths. In the above example, `edge(From, Next)` (line 10) can introduce a choice point, as there can be several ways of transitioning from one node to the next.

## 4. Case study

To demonstrate a wider-ranging use case for our composition of Python and Prolog, this section describes a small case study implemented in Unipycation. Connect Four is a well known strategy game, first released in 1974. The game involves two players (red and yellow), and a vertically standing grid-like board, divided into 6 rows and 7 columns. Players take turns dropping one of their coloured counters into a column. The first player to place 4 counters in a contiguous horizontal, vertical, or diagonal line wins.

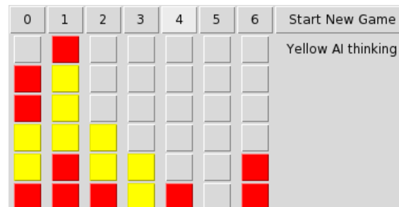


**Figure 1.** The Connect 4 GUI using `Tkinter`.

Connect Four makes an interesting case study for language composition, as a reasonable implementation has two distinct aspects: a user interface and an AI player. Without wishing to start a war between fans of either language, we humbly suggest that Python and Prolog are well suited to one of the aspects, but not the other. An imperative language like Python is well suited to dealing with the presentational side of the game (i.e. a graphical user interface), whereas Prolog is well suited to expressing the behaviour of an AI player. The Python component consists of a simple Tk GUI via the `tkinter` module. As with most GUI frameworks, Tk is event driven, and is a natural fit within Python. The GUI is under 190 LoC. The AI player, on the other hand, is well suited to implementation in Prolog. The AI player uses the minimax method (accelerated with alpha-beta pruning) to compute the best move [8]; it is around 170 lines of Prolog code.

The basic operation of the case study is that the Python part of the program performs all interactions with the users, stores the state of the board, and invokes the Prolog AI player. After every move, the Python part divides the state of the board into two lists `reds` and `yellows`, encoding counter positions as Prolog terms (e.g. a counter at row one, column two is encoded as `c(1, 2)`). Unipycation's Python → Prolog interface is then used to query the Prolog `has_won` predicate with these two lists and a final unbound variable. If the Prolog interpreter finds a binding to variable, then the game has finished, and the last player to move has won.

If no player has won, and the next player to move is the AI opponent, the Python part hands over to the Prolog AI player. To decide a good move, the computer opponent uses a bounded-depth minimax solver [30, 31] implemented efficiently using alpha-beta pruning [16]; we used Bratko's alpha-beta framework as a reference implementation [8, p. 586]. This approach considers the game as a tree of potential moves, where each move is characterised by the positions of the counters (again as two lists) and whose move it is next. Each move has an associated cost which is used as a basis for deciding a good move. We model Connect 4 as a zero sum game, so one player will aim to minimise the cost, whilst the other will aim to maximize the cost. The

alpha-beta framework requires us to define three predicates: one to inform the framework which player is minimising and which player is maximising, one to calculate all possible next moves, and one to calculate the cost of each move. Given these predicates, the alpha-beta framework can make an informed decision about the best move for the computer opponent. Once the best move has been chosen, it is passed back to Python and the game state is updated to reflect the AI player's move.

We discuss our experiences of using Unipycation for the case study in Section 8.

# 5. Unipycation

## 5.1 Constituent interpreters

To compose Python and Prolog, we first need meta-tracing interpreters for each. Fortunately – and not entirely coincidentally – RPython interpreters exist for both languages. PyPy [26] is a fast RPython interpreter for Python. It is fully compatible with CPython (v2.7.3) and is currently the fastest Python VM [7]. Pyrolog [6] is a fairly fast interpreter for Prolog. The two systems have different goals: PyPy is an industrial strength VM, whereas Pyrolog is an extended effort to understand meta-tracing's applicability to a logic programming language. With the normal caution to readers about reading too much into Lines of Code (LoC) counts, PyPy is approximately 35KLoC and Pyrolog 6KLoC. Capturing the performance of VMs in a single number is arguably even more dangerous than LoC codes. Using different benchmarks can substantially change one's perception of a VM's speed, and it is easy to mislead. However, we feel that unfamiliar readers need to have an idea of roughly where each VM sits in the overall performance landscape to understand why we have chosen PyPy and Pyrolog. With that warning in mind, PyPy 2.1 is approximately 6 times faster than CPython on a wide range of benchmarks. Benchmarking in Prolog is less developed than in languages such as Python, and we must be even more cautious than normal when considering Pyrolog's performance. Relative to SWI Prolog, current benchmarking suggests Pyrolog's performance is on an approximate par with it. As this suggests, Pyrolog is not as mature a VM as PyPy, though its performance is still reasonably competitive.

## 5.2 Design

Unipycation can be thought of both as the glue which binds together PyPy and Pyrolog, and the resulting composed interpreter itself. Around 600LoC were added to PyPy and Pyrolog to make Unipycation a reality. Currently, Unipycation hosts Pyrolog within PyPy, meaning that all Unipycation programs start with the execution of Python code; "raw" Prolog code must be either embedded in Python strings or loaded from a separate file.[2]

[2] In a related strand of work, we are creating an editor which will allow a more harmonious syntactic language composition [15].

There are multiple ways in which one could compose Python and Prolog. Our working assumption is that compositions which do not interfere with existing expectations about how each language works are most likely to be acceptable. Therefore, neither the syntax or semantics of Python or Prolog were modified in the composition. We also suspect that compositions which minimise the gap between languages are more likely to be acceptable. We have thus tried, whenever possible, to reuse familiar idioms in each language to represent the other. This requires careful thought, particularly for Prolog features (e.g. unbound variables) which have no direct Python equivalent.

We do not claim that Unipycation is a perfect composition, but it is fairly consistent and simple. In the rest of this section, we discuss the decisions and challenges faced in its design and implementation.

## 5.3 Data type conversions

To allow PyPy and Pyrolog to communicate, language specific data types must be mapped between each interpreter and converted where appropriate. Each interpreter has a hierarchy of RPython classes to represent the data types that it can operate on. Unipycation adds functions to each interpreter to convert data crossing the language boundary; one function is added for each type in each direction.

Some conversions are simple: Prolog integers, big integers, and floats are converted to Python `ints`, `longs`, and `floats` respectively and vice versa. Similarly, Prolog atoms are translated to Python strings and vice versa (note that Pyrolog does not implement Prolog strings). The underlying value of such data types can thus be easily wrapped in the appropriate data type class within the other interpreter.

Prolog terms passed to Python are more interesting. They are wrapped in a special Python class `Term` from which the functor and arguments can be accessed. The `Term` class lazily forwards accesses to the underlying Prolog term (recursively calling the conversion function). This means that passing a Prolog term to Python is efficient, and that users only pay data conversion costs for the data they access. When a Python `Term` is passed back to Prolog, the underlying term is trivially unwrapped. Analogously, Prolog variables are translated to Python objects of type `Var` (see Section 5.4).

Although Unipycation defines conversions for all Prolog data types passed to Python, Python users can define their own classes and Unipycation can not possibly define conversions for all of them. We therefore need a fallback mechanism to allow Python objects of unknown type to be passed to, and through, Prolog code. Our simple solution is to wrap all such objects in a 'black box': Pyrolog can pass such black boxes around, and call methods on them, but can not directly manipulate their contents. Passing a black box back to Python causes it to be unwrapped.

### 5.4 Calling Prolog code from Python

Unipycation programs begin with execution of normal Python code; the Prolog interpreter can be called via the `uni` module. Continuing the path finding example, assume that the rule database of Listing 1 is stored in the file `path.pl` and that we wish to execute, from Python, the query `path(b, To, 4, Nodes)` ("Starting at node `b`, where can we get `To` by visiting at most 4 `Nodes`?"). Listing 2 shows the resulting code. First the `uni` module is imported (line 1). A Prolog engine is then instantiated (line 3). This exposes each predicate in the rule database (which in this case is loaded from file) via the engine's `db` attribute. Queries can then be made by calling predicates either: directly e.g. `engine.db.path(...)` in which case they are expected to provide a single solution and an exception is raised if one is not found; or indirectly via their `iter` method, which can be called to return 0 or more solutions as an iterator (lines 4 and 6). Unbound variables in a query are represented with the `None` object (Python's equivalent of `null`), meaning that `paths("b", None, 4, None)` on line 6 is equivalent to the Prolog query `paths(b, To, 4, Nodes)`. The bindings are then returned, in order, as a Python tuple. In this case, the two bindings are returned (line 6) and printed out (line 7). The output of the program is shown in Listing 3.

```
1  from uni import Engine
2
3  engine = Engine.from_file("path.pl")
4  paths = engine.db.path.iter
5
6  for (to, nodes) in paths("b", None, 4, None):
7      print("To %s via %s" % (to, nodes))
```

**Listing 2.** The Python to Prolog interface.

```
1  To b via ['b']
2  To e via ['b', 'e']
3  To g via ['b', 'e', 'g']
4  To b via ['b', 'e', 'g', 'b']
```

**Listing 3.** The output of the program shown in Listing 2.

Interfacing the execution models of the two interpreters is not trivial. PyPy is a naive recursive bytecode interpreter, whereas Pyrolog uses continuation-passing style. In fact, Pyrolog uses two continuations: one for success and one for failure [6] using a trampoline (which is a common way to implement backtracking). Thus when calling the `iter`-variant of a Prolog predicate, Unipycation returns an instance of an iterator class that captures the state of the Pyrolog continuations. When the Python code asks the iterator for the next element, it induces backtracking on the stored Prolog continuations and runs Prolog from there. If that yields a new Prolog solution, it is returned to Python, otherwise the iterator is depleted.

#### 5.4.1 Using variables and terms explicitly

While the query interface presented so far works well in most cases, it is not fully general. Unbound variables must appear at the top level of a query, making `paths.iter("b", "e", 5, ["b", "e", None])` invalid. Forbidding nested `None`s means the basic query interface does not have to fully traverse every query, making this aspect $O(1)$ no matter the size of the query. Similarly, the eager conversion of lists makes representing Prolog partial lists (nested cons cells with an undefined tail) such as [1, 2 | X] impossible: a run-time exception is raised when a partial list is returned to Python.

To cope with such cases, Unipycation users can model variables and terms explicitly. Consider the Prolog query `path(a, To, 5, [a, c | OtherNodes])` which uses a partial list and a nested variable. Listing 4 demonstrates how terms and variables can be used explicitly to represent this query. First we import the relevant types (line 1) and manually create Prolog variables (line 5). We then build the partial list as a chain of cons terms (line 6). We then build a `Term` to represent the query (line 7) before explicitly calling the Prolog engine's `query_iter` function to produce all solutions (line 9). Unbound variables must be explicitly passed to `query_iter` to avoid having to search composite expressions. The result of `query_iter` is a dictionary of bindings. Prolog lists returned via this interface are left as chains of cons cells (using Term instances).

```
1  from uni import Engine, Term, Var
2
3  # ... Instantiate engine as before...
4
5  (to, tail) = (Var(), Var())
6  part_list = Term('.', ["a", Term('.', ["c", tail])])
7  query = Term('path', ["a", tail, 5, part_list])
8
9  for sol in engine.query_iter(query, [to, tail]):
10     print("to=%s, tail=%s" % (sol[to], sol[tail]))
11
12 print("done")
```

**Listing 4.** Querying with explicit terms and variables.

Although the explicit query interface is sometimes the only possible route, careful thought can reduce the need to use its somewhat verbose idioms. "Wrapper predicates" added to the Prolog rule database can subsume many use cases. For example to avoid having to write Listing 4, one could add the predicate shown in Listing 5 to the Prolog database and call it using the simple query interface as `e.db.path_via(a, None, [a, c], None)`.

```
1  path_via(From, To, MaxLen, Prefix, OtherNodes) :-
2          append(Prefix, OtherNodes, FullList),
3          path(From, To, MaxLen, FullList).
```

**Listing 5.** Avoiding explicit terms and variables.

### 5.4.2 Dynamically generating rules from Python

Not only do the individual interpreters allow meta-programming within themselves, but Unipycation allows meta-programming across the language boundary. Suppose, in our running example, that our graph represents tube stations on London's underground. Stations open and close (e.g. for maintenance work) and we would like to alter the graph dynamically. Python code can use the `assertz` Prolog builtin to insert new rules into the database. Listing 6 shows Python code which adds an edge from `a` to `g` in the Prolog database.

```
1 edge = engine.terms.edge("a", "g")
2 engine.db.assertz(edge)
```

**Listing 6.** Adding rules to the database using `assertz`.

Using this technique, it is not difficult to imagine a path finder which uses Python to dynamically acquire map updates through a web JSON API and updates the Prolog database.

## 5.5 Calling Python code from Prolog

Just as Python code can call Prolog code, Unipycation also enables the reverse. Deciding which names to expose to Prolog is more complex than deciding which names to expose to Python. Similar to the `db` attribute, the `python` interface (which masquerades as a Prolog module) uses run-time meta-programming to lookup names and make their values available to Prolog. Users can explicitly make names available by passing a dictionary to the Prolog engine when it is initialised. If no match for a name is found, the Python builtins are then searched, making it easy for Prolog code to call builtins such as `dict`, `len` and the like. Since Prolog has no syntax for common Python operations such as looking up an item in a list, the contents of Python's `operator` module are also exposed. Prolog code which wants to execute `Z = l[i]`, for example, can call `python:getitem(l, i, Z)`.

To make method calls syntactically palatable, we overload Prolog's module lookup operator `:` so that a list `L`, for example, can be sorted in Prolog via `L:sort(_)`. As this suggests, calls to Python are realised by passing one extra parameter than normal. The result then unifies the extra parameter with the result of the Python call. If a function does not return a result, or the user wants to ignore it, the anonymous variable `_` can be passed as the final argument in normal Prolog style, as is the case with `sort`. Multiple `:` indirections can be chained together provided that the chain ultimately results in a function or method call e.g. the Python call `os.path.exists("hosts")` can be expressed in Prolog as `python:os:path:exists(hosts, Exists)`.

Python functions and methods which return iterators become Prolog choice points. These are encapsulated in a new kind of Prolog failure continuation, and then pumped for further solutions when backtracking occurs. The successive elements of the iterator are the potential solutions of the call to Python, the result of which is unified with the last argument of the call one by one.

Continuing the running example, the ability to call from Prolog back to Python means that the representation of the edges of the graph can be stored in a Python dictionary for fast lookup. To achieve this we firstly move the edge descriptions into the Python part of the program as shown in Listing 7. Additionally a helper function `get_edges` is defined which, given a source node, returns an iterator over the destination nodes, thus describing the possible edges. Finally, we replace the `edge` predicates many rules with a single rule:

```
edge(From, To) :- python:get_edges(From, To)
```

Since `get_edges` returns an iterator, Unipycation converts it to a Prolog choice point, making it a basis for backtracking when finding solutions.

```
1 edges = {
2     "a" : ["c"], "c" : ["b", "d", "f"], "d" : ["e"],
3     "b" : ["e"], "f" : ["g"], "e" : ["g"], "g" : ["b"]
4 }
5
6 def get_edges(src_node):
7     return iter(edges[src_node])
```

**Listing 7.** Storing the edges in a dictionary.

## 5.6 Interaction with meta-tracing

Unipycation's constituent interpreters have their own meta-tracing JITs, such that code running in a single interpreter is as fast as running it in a stand-alone PyPy or Pyrolog VM. In this section we explore what happens on the level of the tracing JITs when the interpreters call each other. Our explanation is necessarily qualitative: we have performed spot-checks of the generated traces but have not yet systematically evaluated the effects quantitatively. In order to do so, we will have to invent a new evaluation methodology, which is no small undertaking. Nevertheless we believe that this section is a vital part of understanding the later preliminary benchmarks.

The most important thing to note about Unipycation is that most of its optimisations are inherent to meta-tracing, which is the reason we believe that it is a promising approach to language composition implementation. Both PyPy and Pyrolog are optimised for meta-tracing, in the sense that their implementation has, where necessary, been structured according to meta-tracings demands. Such structuring is relatively minor (see [7] for more details): most commonly, tracing annotations [4] are added to the interpreter to provide hints (e.g. "this RPython function's loops can safely be unrolled") and guarantees (e.g. "this RPython function always returns the same results given the same inputs") to the tracer; less commonly, code is e.g. moved into its own function to allow an annotation to be added. To improve the

performance of Unipycation, we added ten such annotations, but PyPy and Pyrolog themselves were left unchanged.

A tracing JIT's natural tendency to aggressively type-specialise code (see [4, 17]) is important in reducing the overhead of object conversions between interpreters. Tracing a call from one language to the other naturally traces the object conversion code (Section 5.3), type specialising it. In essence, Unipycation assumes that the types of objects converted at a given point in the code will stay relatively constant; similar assumptions are the basis of all dynamically typed language JITs. Type specialisation leaves behind a type guard, so that if the assumption of type constancy is later invalidated, execution returns to the interpreter. If, as is likely, the object conversion is part of a bigger trace with a previous type guard, RPython's tracing optimiser will remove the type guard in the object conversion entirely. Our experience, therefore, is that this aggressive type-specialisation helps reduce the overhead of converting objects between the languages.

A related optimisation occurs on the wrapped objects that are created by cross-interpreter object conversion. The frequent passing of objects between the two interpreters would seem to be highly inefficient, as most will have to have a new wrapper object created each time they are passed to another interpreter. Our experience is that most such objects are short-lived. Fortunately, RPython's trace optimiser performs escape analysis on traces which is able to remove allocation costs for objects which live and die within a trace [3]. Our experience, again, is that this is a very effective optimisation for Unipycation.

The final optimisation we would hope to inherit from meta-tracing is inlining, since tracing naturally inlines functions unless they are very large or contain loops. Unfortunately, while it would seem relatively easy to inline cross-language calls, this is a work in progress: Python functions are inlined into Prolog code if they don't return iterators; Prolog functions can not yet be inlined into Python. Though we hope to remove this limitation soon, it is interesting to understand why it happens. In short, determining where a loop starts in Prolog is tricky, because loops are realised as tail-recursive predicates. Pyrolog therefore marks every predicate call as a potential loop, which means that the tracer avoids inlining them. We hope to fix this by adding heuristics to Pyrolog to identify potential loops.

## 6. Performance evaluation

Apart from our tests and a few case studies, no other software has been written with Unipycation. Not only do we lack programs which make good benchmarks, but it is not even clear what good benchmarks might look like, nor which systems to compare against. We have therefore created 7 micro-benchmarks to give us some idea of Unipycation's cross-language performance. While it is important to realise that these results can not be generalised to large programs, they are a useful first step in designing good benchmarks for composed VMs.

Because we are most interested in evaluating the performance of cross-language calling, each benchmark has two functions, one calling the other. The micro-benchmarks are as follows:

**SmallFunc** The outer function calls a tiny inlinable inner function in a loop.

**Loop1Arg0Result** The outer function and inner function are loops, the inner function receiving a single integer argument.

**Loop1Arg1Result** The outer function and inner function are loops, the inner function receiving a single integer argument and returning a single integer result.

**NondetLoop1Arg1Result** The outer function calls an inner function in a loop. The inner function produces more than one integer result (by returning an iterator in Python, and leaving a choice point in Prolog). The outer function asks for all the results (with a `for` loop in Python, and with a failure-driven loop in Prolog).

**Lists** The inner function produces a list, and the outer function consumes it. The lists are converted between Prolog linked lists and Python array-based lists when passing the language barrier.

**PythonInstances** The inner function produces a linked list using Python instances, the outer function walks the linked list.

**TermConstruction** The inner function produces a linked list using Prolog terms, the outer function walks the linked list.

The last two benchmarks have an important difference over the first 5: in *PythonInstances*, even the pure Prolog version handles Python instances; and in *TermConstruction*, even the pure Python variant handles Prolog terms.

For each benchmark we have created four variants:

**Python** both functions are written in Python.

**Prolog** both functions are written in Prolog.

**Python → Prolog** the caller is written in Python, the callee in Prolog.

**Prolog → Python** the caller is written in Prolog, the callee in Python.

The first two variants give us a baseline, while the latter two variants measure calling across the two languages. If the inter-language variants are approximately the same speed as the intra-language variants, we consider the VM composition to be efficient.

Benchmarks are run on an otherwise idle Intel Core i5-3230M CPU with 2.60GHz and 3072 KiB cache and 16 GiB RAM running Ubuntu Linux 13.04 in 64 bit mode. We run

the benchmarks 50 times within the same process, discarding the first 5 runs (which a manual inspection showed is sufficient to allow the JIT to warm up; indeed, most benchmarks are warmed up after one iteration). We then report the average time together with a confidence interval using a 95% confidence level. The benchmarks are fully repeatable and available for download (see page 2).

### 6.1 Analysis

The benchmark's absolute run times in seconds can be seen in Figure 1; Figure 2 shows the run times normalised to the pure Python version, which we use as a baseline.

The two simplest benchmark results are *Loop1Arg0-Result* and *Loop1Arg1Result* which support the common perception that cross-language overheads can be neglected if the work done on either side is large and the crossings few.

In Section 5.6 we explained why Unipycation currently inlines Python functions called from Prolog, but not the reverse. *SmallFunc* (unintentionally) highlights how important cross-language inlining can be. The Prolog → Python variant, where cross-language inlining occurs, is as fast as the pure Python version; the Python → Prolog variant, where cross-language inlining does not occur, is almost 100 times slower than the pure Python variant. A similar effect can be seen in *NondetLoop1Arg1Result*. There, the Python function called from Prolog returns an iterator and can not be inlined in Prolog. Thus neither the Python → Prolog nor Prolog → Python version inlines cross-language calls, leading to poor performance in both directions.

The overheads of the cross-language versions of *Lists*, *PythonInstances*, and *TermConstruction* are relatively small, at least compared to the very large overheads of *Small-Func* and *NondetLoop1Arg1Result*. However, we do not yet fully understand why these benchmarks perform as they do. For example, why is pure Prolog so much faster operating on Python instances in *PythonInstances* than on its native terms? The reasons could be due to one or more of PyPy, Pyrolog, RPython, or Unipycation. It is likely that we will need to devise new benchmarks, and possibly new types of analyses, to uncover the explanations of such results.

By our simple definition of efficiency (see Section 6), our benchmarks suggest Unipycation is often an efficient cross-language composition but sometimes decidedly not. While we caution readers not to assume that these micro-benchmarks tell one anything about 'real' Unipycation programs, we believe that they show that the approach of composing meta-tracing interpreters is a promising one. Clearly, we have much work to do to improve performance further, but Unipycation already appears to be a reasonable start.

## 7. Related Work

The motivation for language composition dates back to the late 60s [12], though most of the early work was on extensible languages (e.g. [22]); to the best of our knowledge such work largely disappeared from view for many years, though there has been occasional successor work (e.g. [10, 24]). Unfortunately, the passing of time has made it hard to relate much of this early work to the current day—in some cases it has been decades since the systems involved could be run.

Though it lacks a single defining authority, the modern Domain Specific Language (DSL) movement aims for a limited form of language composition. One part of the movement (best represented by [20]) sees DSLs as a specific way of using a language's existing syntax and semantics, and is of little relevance to this paper. The other part of the movement aims to actively extend a language's syntax and semantics. It can be subdivided into heterogeneous and homogeneous language embeddings [29]. Homogeneous embedding uses a single language's system to express and host a language embedding; most commonly via macros (e.g. Lisp) or compile-time meta-programming (e.g. Converge [29]). Heterogeneous embedding uses an external system (e.g. Stratego [9]) to express the embedding in a separate host system. The two types of DSL embedding have important trade-offs: homogeneous embedding is safe but inexpressive; heterogeneous embedding is expressive but unsafe. In either case, it is hard to make the eventual running programs efficient because of inevitable semantic mismatches (see page 1) between the DSL and host language. For example, composing Converge and a rule-based DSL was extremely inefficient due to the encoding of backtracking [29]. Not only was the added machinery large, but it also defeated many of the VM's optimisations.

Semantic mismatches make it difficult to create performant language compositions atop a single VM. While Java programs on HotSpot have excellent performance, other languages (e.g. Python) on HotSpot often run slower than simple C-based interpreters [7]. While better VM extensions (e.g. `invokedynamic`) or compiler alterations (e.g. [23]) can improve performance, the results still lag some way behind their meta-tracing equivalents [11]. We believe that language composition by meta-tracing has the potential to sidestep the semantic mismatch problem entirely by allowing each language's VM to be optimised specifically for that language. As Unipycation is very simple, we can not yet say whether this potential is realisable or not, though we have not yet seen any evidence to suggest that it is not.

Similar approaches to Unipycation at the language-level are the composition of Smalltalk and SOUL (a Prolog-like logic programming language) [14, 19] and Java and tuProlog [13]. The SOUL / Smalltalk composition has a similar cross-language API to Unipycation. For example, SOUL predicates are mapped to message sends, and SOUL's multiple solutions are mapped to collections (albeit not lazily). However, SOUL is implemented in Smalltalk, and tuProlog in Java. This is very different to Unipycation which does not implement one of the composed languages in terms of another, but composes both as separate interpreters. Thus while

| *Benchmark* | Python | Py → Prolog | Prolog → Py | Prolog |
|---|---|---|---|---|
| SmallFunc | 0.0944 ± 0.0013 | 9.3157 ± 0.0054 | 0.0943 ± 0.0000 | 0.7004 ± 0.0008 |
| Loop1Arg0Result | 0.1015 ± 0.0001 | 0.1155 ± 0.0003 | 0.1016 ± 0.0001 | 0.1151 ± 0.0013 |
| Loop1Arg1Result | 0.1340 ± 0.0013 | 0.1569 ± 0.0005 | 0.1336 ± 0.0001 | 0.1578 ± 0.0011 |
| NondetLoop1Arg1Result | 0.5498 ± 0.0002 | 8.1446 ± 0.0063 | 82.3137 ± 0.0844 | 0.6144 ± 0.0001 |
| Lists | 0.9955 ± 0.0004 | 3.2495 ± 0.0029 | 7.9950 ± 0.0079 | 7.3660 ± 0.0079 |
| PythonInstances | 1.9009 ± 0.0010 | 1.9732 ± 0.0010 | 2.2898 ± 0.0042 | 2.3849 ± 0.0026 |
| TermConstruction | 2.6400 ± 0.0019 | 4.3984 ± 0.0061 | 8.8248 ± 0.0078 | 3.8277 ± 0.0041 |

**Table 1.** Performance (in seconds) for the synthetic benchmarks.

| *Benchmark* | Python | Py → Prolog | Prolog → Py | Prolog |
|---|---|---|---|---|
| SmallFunc | 1.0 | 98.6895 ± 1.3580 | 0.9986 ± 0.0137 | 7.4203 ± 0.1023 |
| Loop1Arg0Result | 1.0 | 1.1371 ± 0.0030 | 1.0007 ± 0.0008 | 1.1335 ± 0.0129 |
| Loop1Arg1Result | 1.0 | 1.1713 ± 0.0121 | 0.9968 ± 0.0098 | 1.1776 ± 0.0143 |
| NondetLoop1Arg1Result | 1.0 | 14.8134 ± 0.0126 | 149.7116 ± 0.1625 | 1.1175 ± 0.0004 |
| Lists | 1.0 | 3.2643 ± 0.0032 | 8.0313 ± 0.0087 | 7.3995 ± 0.0086 |
| PythonInstances | 1.0 | 1.0380 ± 0.0007 | 1.2046 ± 0.0023 | 1.2546 ± 0.0015 |
| TermConstruction | 1.0 | 1.6661 ± 0.0026 | 3.3427 ± 0.0038 | 1.4499 ± 0.0019 |

**Table 2.** Performance normalized to the Python results for the synthetic benchmarks.

Unipycation composes two independently optimised and JIT compiled interpreters, the other two compositions embed a (slow) interpreter inside a normal program.

## 8. Discussion

Although neither PyPy or Pyrolog was designed with the possibility of interpreter composition in mind, Unipycation was relatively easy to implement, taking under four man months. We believe this level of effort compares favourably to traditional language composition approaches (e.g. translating to C or the JVM). The early signs are that Unipycation's performance inherits meta-tracing's general benefits, in which case our experience from [7] suggests that it will outperform the equivalent composition on the JVM. Clearly, there is still considerable scope for improving performance, for example by inlining Prolog code called from Python.

Although we are not the first to try composing imperative and logic-based systems (see e.g. the Smalltalk/SOUL composition [14]), we were pleased with the relative ease of designing the language interactions in both directions. We believe this bodes well for designing interactions between languages with more obviously compatible semantics (e.g. Python and Ruby).

We found writing Unipycation case studies surprisingly natural. The boundary between the languages is clear and passing data between them simple. The most frequently encountered difficulty involved converting user-defined types, which are not covered by Unipycation's built-in rules (see Section 5.3). For example, the Connect 4 program needs to use two different representations of a counter. In Python a counter is a binary tuple, while in Prolog it is a binary term of functor `c`. Listing 8 shows a function which converts a list of counters between these two representations. Although Python's list comprehension syntax makes the conversion somewhat easier, manual type conversions can feel clumsy. It may be plausible to 'register' such conversions with Unipycation and have them performed automatically.

```
1  def _counters_to_terms(self):
2      """ convert the board to Prolog terms """
3      reds = [self.pl_engine.terms.c(x, y) for \
4        (x, y) in self._collect_token_coords("red")]
5      yellows = [self.pl_engine.terms.c(x, y) for \
6        (x, y) in self._collect_token_coords("yellow")]
7      return (reds, yellows)
```

**Listing 8.** Converting lists of counters in Connect 4.

Our experience highlighted some usability problems which will take more effort to address. Debugging is an obvious problem. Python code can be debugged with the PyPy debugger `pdb`, but Pyrolog has no debugger. Even if it did, it is not clear how cross-language debugging might best be presented to the user. Such problems will have to be solved before the style of language composition we have described in this paper will become acceptable.

## 9. Conclusions

In this paper, we presented the first composition of meta-tracing interpreters, an approach which should eventually allow different languages to be composed while maintaining their individual high levels of performance. Unipycation is relatively simple, but it is not trivial: Python and Pro-

log are languages with substantially different outlooks. We performed a simple evaluation of Unipycation with micro-benchmarks which suggest its performance is, at the very least, usable. Equally importantly, we showed via a case study that the composition can be used for tasks that a real developer may find useful.

**Acknowledgements:** We thank Tim Felgentreff for comments on a draft of this paper. This research was funded by the EPSRC *Cooler* grant EP/K01790X/1.

## References


[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN*, 35(5):1–12, 2000.

[2] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *Proc. OOPSLA*, pages 708–725, 2010.

[3] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. *Proc. PEPM*, pages 43–52, 2011.

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proc. ICOOOLPS*, page 9:1–9:8, 2011.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proc. ICOOOLPS*, pages 18–25, 2009.

[6] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a Jitting VM for prolog execution. In *Proc. PPDP*, pages 99–108, 2010.

[7] C. F. Bolz and L. Tratt. The impact of meta-tracing on VM design and implementation. *To appear SCICO*, 2013.

[8] I. Bratko. *Prolog programming for artificial intelligence*. Addison Wesley, 2001.

[9] M. Bravenboer and E. Visser. Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In *Proc. OOPSLA*, pages 365–383, 2004.

[10] L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In *Workshop on Database Programming Languages*, pages 11–31, 1993.

[11] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. pages 195–212, Oct. 2012.

[12] T. E. Cheatham. Motivation for extensible languages. *SIGPLAN*, 4(8):45–49, Aug. 1969.

[13] E. Denti, A. Omicini, and A. Ricci. tuProlog: a light-weight prolog for internet applications and infrastructures. In *Proc. PADL*, volume 1990, pages 184–198. 2001.

[14] M. D'Hondt, K. Gybels, and V. Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. Proc. SAC, pages 1328–1335, 2004.

[15] L. Diekmann and L. Tratt. Parsing composed grammars with language boxes. In *Workshop on Scalable Language Specifications*, 2013.

[16] D. J. Edwards and T. P. Hart. The alpha-beta heuristic. Technical Report AIM-30, MIT, 1961.

[17] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-In-Time type specialization for dynamic languages. In *Proc. PLDI*, pages 465–478, 2009.

[18] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proc. VEE*, pages 144–153, 2006.

[19] K. Gybels. SOUL and Smalltalk - just married: evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proc. DP-COOL*, 2003.

[20] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.

[21] 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 *Proc. OOPSLA*, pages 318–326, 1997.

[22] E. T. Irons. Experience with an extensible language. *Communications of the ACM*, 13(1):31–40, 1970.

[23] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: Performance evaluation, analysis, and tradeoffs. In *VEE*, pages 169–180. ACM, 2012.

[24] G. F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proc. POPL*, pages 141–151, 1985.

[25] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp Symb. Comput.*, 7(4):315–335, 1994.

[26] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proc. DLS*, pages 944–953, 2006.

[27] G. L. Steele, Jr. Growing a language. *HOSC*, 12(3):221–236, 1999.

[28] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Workshop on Interpreters, Virtual Machines and Emulators*, pages 50–57, 2003.

[29] L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.

[30] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[31] J. von Neumann. *Theory of games and economic behavior*. Princeton University Press, 1944.

[32] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proc. DLS*, pages 79–88, 2009.