An Executable Semantics for Faster Development of Optimizing Python Compilers

Olivier Melançon
olivier.melancon.1@umontreal.ca
Université de Montréal
Montréal, Canada

Marc Feeley
feeley@iro.umontreal.ca
Université de Montréal
Montréal, Canada

Manuel Serrano
Manuel.Serrano@inria.fr
Inria/Université Côte d'Azur
Sophia Antipolis, France

Abstract

Python is a popular programming language whose performance is known to be uncompetitive in comparison to static languages such as C. Although significant efforts have already accelerated implementations of the language, more efficient ones are still required. The development of such optimized implementations is nevertheless hampered by its complex semantics and the lack of an official formal semantics. We address this issue by presenting an approach to define an executable semantics targeting the development of optimizing compilers. This executable semantics is written in a format that highlights type checks, primitive values boxing and unboxing, and function calls, which are all known sources of overhead. We also present semPy, a partial evaluator of our executable semantics that can be used to remove redundant operations when evaluating arithmetic operators. Finally, we present Zipi, a Python optimizing compiler prototype developed with the aid of semPy. On some tasks, Zipi displays performance competitive with that of state-of-the-art Python implementations.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**.

Keywords: compiler, dynamic programming language, optimization, executable semantics, partial evaluation, python

ACM Reference Format:
Olivier Melançon, Marc Feeley, and Manuel Serrano. 2023. An Executable Semantics for Faster Development of Optimizing Python Compilers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623529

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623529

1 Introduction

Python is a dynamic language known for its extensive standard library, object-oriented approach and admittedly poor performance in comparison to static languages such as C and dynamic languages such as JavaScript [12]. It is nonetheless among the most popular languages in use today and its popularity shows no sign of decline [32].

The Python language specification is *The Python Language Reference* [31]. While the syntax is formally specified, this is not the case of the semantics, leaving room for ambiguities and making it difficult to reason about programs [21].

In our experience, these challenges are related. Python's complex semantics and absence of formal specification complicate the development of a compiler compatible with CPython, the reference implementation. The effort spent getting the semantics right leaves little time for optimization.

Furthermore, part of the semantics makes Python implementations susceptible to execute redundant type checks, extensive boxing and unboxing of primitive values, and abundant method calls, which affects performance [14, 34]. Optimization of operations on atomic types (such as `int` and `float`) has been suggested to resolve this issue [34].

This paper offers two main contributions. First, we describe an executable semantics for Python that is written in a Python syntax to allow reuse in existing compilers. Second, we present a tool that applies partial evaluation to remove redundant type checks, boxing and unboxing, and method calls from arithmetic operations on atomic types. We use this executable semantics to automate the implementation of arithmetic operations in an optimizing compiler and demonstrate that it provides run time performance competitive with those of PyPy [5], a state-of-the-art Python implementation.

This paper is organized as follows. In Section 2, we provide an overview of Python's semantics. In Section 3, we define an executable semantics that describes the behavior of various Python operations. In sections 4 and 5, we present a technique for partial evaluation of our executable semantics that focuses on removing redundant type checks, boxing and unboxing, and method lookups and invocations from Python operations. In Section 6, we show how we reused our executable semantics in the implementation of Zipi, our partial implementation of a Python optimizing compiler. Finally, in Section 7 we provide an overview of performance.

2 Overview of Python's Semantics

Python's semantics is highly dynamic. This is an obstacle to the implementation of an optimizing compiler. This section gives an overview of this problem.

As of today, Python does not have an official formal semantics. The reference manual [31] uses prose instead of a formal specification, which leaves room for ambiguities. When such ambiguities arise, we refer to the behavior of CPython [28], the reference implementation.

2.1 Data Model

Python's abstraction for data are *objects*, which are entirely defined by their *identity*, *type* and *value*. The *identity* is a unique integer value that never changes across the life of the object and is available by calling `id(obj)`. The *value* is the data represented by the object, for example an integer, a floating point number or a pointer to another data structure. Finally, the *type* determines the operations allowed on the object. An object's type is itself an object that can be obtained with `type(obj)`. Under certain conditions, an object's type can be modified. However, this is not possible for objects whose type is a built-in type such as booleans, floats, integers, strings, lists, tuples, sets and dictionaries.

All values in a Python program are objects. For example, Python boolean values are represented by the singleton objects `True` and `False`, which belong to the `bool` type, a subtype of the `int` type. This contrasts with other object-oriented languages such as JavaScript where primitive data types such as `number` and `boolean` exist [20]. In the absence of primitive values, the operations allowed on an object are defined entirely by the methods available on its type. Such methods governing operations are called *magic methods*.

Figure 1 shows the semantics of the operation `(x + y)`. It attempts to invoke the `__add__` magic method of `x`'s type. If the `__add__` method is found, it is invoked with `x` and `y` as arguments and returns the result of `(x + y)`. If `__add__` is not found, or if it is found but returns the special singleton object `NotImplemented`, addition falls back on the `__radd__` method of the type of `y` (the `r` in the name stands for *reflected*). Otherwise, it raises a `TypeError` exception with an explicative message (which is omitted for brevity).

The semantics of other arithmetic operations are similar, only the names of the required methods change. For example, the semantics of the subtraction operator is identical to that of addition, but calls the methods `__sub__` and `__rsub__`.

The only operators that cannot be overloaded are the "`is`" operator, which compares objects by identity, and the "`and`", "`or`" and "`not`" boolean operators. All other operations are governed by magic methods. For example, iteration in a `for`-loop calls the `__iter__` method, which returns an iterator. The syntactical form `obj.attr` for attribute access calls the `__getattribute__` method. The same applies for function invocation, truthiness, type casting and so on.

```
py_add(x, y): # semantics of x + y
  if type(x) has a method __add__:
    result = type(x).__add__(x, y)
    if result is the object NotImplemented:
      return py_radd(y, x)
    else:
      return result
  else:
    return py_radd(y, x)

py_radd(y, x): # reflected addition
  if type(y) has a method __radd__:
    result = type(y).__radd__(y, x)
    if result is the object NotImplemented:
      raise TypeError
    else:
      return result
  else:
    raise TypeError
```

Figure 1. Pseudocode for the semantics of the + operator

2.2 Method Resolution Order

The Python language supports multiple inheritance. Inheritance expands the features of a type by enabling it to access its parents' magic methods. When recovering a method on a type, such as the `__add__` method in Figure 1, Python executes an ordered search across the type and its parents. The expression `type(x).__add__` first looks for `__add__` on `type(x)` itself. If no such method is found, it is looked up recursively on the parents of `type(x)`.

To avoid inconsistencies in the context of multiple inheritance, searching for a magic method (or any attribute) requires an order in which to traverse the parents, called the *method resolution order* (MRO). The MRO is a property of a type computed at the creation of the `type` object by using the *C3 superclass linearization* algorithm [23]. The MRO cannot be altered afterward, but the types contained in the MRO are often mutable. Their attributes may be updated, new ones may be introduced, or existing ones may be removed. This prevents determining a result of the lookup of each magic method at the creation of a type.

An important exception is that attributes of all built-in types are read-only. That is the case both in CPython and PyPy [27], another popular implementation of the language. Immutability of built-in types is part of Python's semantics.

2.3 Dynamic Environments and Attributes

Python incorporates features such as dynamic typing, late binding, and dynamic code evaluation. It also offers a deep level of introspection that allows altering the behavior of a program in ways that a compiler can hardly predict through static analysis [17].

Python supports modular programming through `module` objects. No syntactical distinction is made between a code file intended to be run directly and one intended to be imported. When a file is executed, an object of type `module` is created. All global assignments executed in its code are stored as attributes of the module. Conversely, any modifications applied to a module's attributes is reflected on its global environment. Other Python programs can then import this module to access and also update its attributes.

Python also allows the global scopes of its modules to be reified by invoking the `globals()` built-in function. This function returns a dictionary (a hash table) that allows the global environment to be read and written. Since the returned dictionary is a Python object, any program can keep a reference to it and update it. The prospect of dynamically loaded code updating the environment at any point of the execution always remains. This makes static analysis of global variables impracticable.

2.4 Dynamic Type Checks

In Figure 1, we showed that two objects can be added if the left-hand operand's type has a `__add__` method that does not return `NotImplemented` for the given right-hand operand. This process requires two forms of type checks[1]: (1) looking up whether the left-hand operand's type has an `__add__` method and (2) checking if the type of the right-hand operand is suitable for the operation. No explicit type check is required on the left-hand operand since the method `__add__` was recovered on its type. This ensures that the magic method receives a first argument of a suitable type.

However, magic methods are not private and can be called with unexpected arguments. Figure 2 shows the result of directly calling `int.__sub__` and `float.__rsub__` with various arguments. The first call returns the expected result of the operation `(43 - 1)`. The second call shows that `int.__sub__` does not know how to handle an argument of type `float`. Subtraction between an `int` and a `float` is handled by `float.__rsub__` (third call). The last call shows that if the first argument of `int.__sub__` is not an `int` then a `TypeError` is raised. We conclude that the `int.__sub__` method contains a type check of its first argument. When computing the result of the "−" operator, this check is redundant since the left-hand operand is already known to be an instance of `int`. We observed similar behaviors for other built-in magic methods.

2.5 Sources of Overhead

The features presented in this section explain the poor performance of a naive implementation of Python. An operation as simple as subtracting an integer and a floating point number requires two method searches in the MRO

[1]We employ *type check* in a broad sense to refer to any operation that requires a test on the type of an object, including magic method dispatches.

```
>>> int.__sub__(43, 1)
42
>>> int.__sub__(43, 1.0)
NotImplemented
>>> float.__rsub__(1.0, 43)
42.0
>>> int.__sub__(43.0, 1.0)
TypeError: descriptor '__sub__' requires a
           'int' object but received a 'float'
```

Figure 2. Results of direct calls to magic methods in CPython

of `int` and `float` respectively. Both methods are called due to the first returning `NotImplemented`. In both cases, the magic methods apply a redundant type check on their first argument.

In particular, implementing number arithmetic with method calls introduces a major overhead on operations that could otherwise be computed with a single assembly instruction as C would do. In the case of CPython, function and method calls are the primary source of overhead [34].

Furthermore, in the context of arithmetic operations, magic methods are required to extract the values from `int` and `float` objects and generate a new object to store the result. This procedure, known as *boxing and unboxing*, leads to additional overhead [14].

3 Executable Semantics for Python

We now present an executable semantics aimed at developing optimizing Python compilers. Our goal is for such a formalization to (1) automate the implementation of a Python compiler, (2) be easily reusable by existing Python compilers and (3) yield performant implementations.

Writing the numerous magic methods of Python's built-in types by hand is tedious and error-prone. We ought to automate this process to accelerate development, including that of existing compilers, independently of the language and tools chosen for its implementation. We achieve this by writing the semantics in the syntax of Python. Hence, it is possible to interface with the semantics by using the parsing infrastructure of an existing compiler.

Our strategy is similar to that of RPython, which implements a subset of Python with limited dynamic features [1]. It differs in that we instead use a superset of Python to highlight parts of the semantics causing overhead such as boxing and unboxing of primitive values, type checking, and method calls. In this section, we introduce this superset of Python and use it to write an executable semantics. We will show how this semantics can be read by a compiler to implement optimized versions of various operators in sections 5 and 6.

3.1 The Compiler Intrinsics Statement

To express the semantics of operators, we extend Python with the *compiler intrinsics* statement. Its syntax is the same as that of an `import` statement, except that the module name must be `__compiler_intrinsics__` followed by a sequence of names. The imported names correspond to low-level primitives that we call *intrinsics* (we detail all intrisincs in Appendix A). Intrinsics imported with the compiler intrinsics statement are static, they cannot be shadowed by another assignment or assigned to a variable. Since the compiler intrinsics statement reuses the syntax of Python's `import`, its implementation requires no change to the parser.

The compiler intrinsics statement has been sufficient to implement all arithmetic operators, unary operators, comparison operators, truthiness, length, type casts, attribute access and assignment, subscript access and assignment, and context managers [15]. These operators are magic-method-dependent, which the compiler intrinsics statement is well suited to implement. We have yet to extend our executable semantics to describe control flow, scoping rules and other features that do not rely on magic methods.

In sections 3.2 and 3.3, we provide two examples of operators for which an executable semantics can be written with the compiler intrinsics statement: addition and truthiness. These examples effectively illustrate why seemingly simple operations incur a significant overhead.

3.2 Example: Semantics of Addition

In Figure 3, we translate the addition semantics from Figure 1. We import three intrinsics: (1) `define_semantics`, which indicates that a decorated function is not a Python function, but rather the definition of an operator's semantics, (2) `class_getattr`, which implements the MRO lookup of a magic method, and (3) `absent`, a sentinel value returned by `class_getattr` if no corresponding magic method is found.

The addition semantics in Figure 3 defines the nested function `normal` (line 6) and `reflected` (line 17). Since those are in the scope of a `define_semantics`, the compiler can avoid the allocation of `function` objects and define low-level procedures instead. It is also possible to apply *lambda-lifting* to prevent the creation of closures capturing '`x`' and '`y`'. All arithmetic operators can be defined in similar fashion.

When a function is decorated with `define_semantics`, we refer to it as *a semantics* or the *semantics of* a given operator.

3.3 Example: Semantics of Truthiness

An object's truthiness is computed when it is used as the condition of an `if` statement or `while` statement, or if converted to a boolean using `bool(x)`. Objects considered to be falsy include `False`, `None`, zeros of numeric types and empty sequences (e.g., an empty list or string).

The operation of truthiness is especially convoluted since it falls back on recovering the length of objects whose type does not have a `__bool__` magic method. Computing the length of an object has its own semantics, which must assert that the resulting length is a *small* integer.[2] In Figure 4, we implement the truthiness operation. It attempts to call the `__bool__` method, but may fall back on the `maybe_length` operation. The latter computes the length of an object, but returns `absent` if the object's type does not have a `__len__` method (in which case the object is always truthy).

The `maybe_length` semantics must assert that the computed length is a small integer. This operation is implemented by the `index` semantics in Figure 5. To abstract the notion of *small* integers, we introduce two intrinsic types: `sint` and `bint`, which respectively stand for small and big integer. Those are abstract subtypes of `int` that differentiate between small and big integers using the `isinstance` built-in function (fig. 5, lines 12 and 14) while leaving room for implementation-dependent details. The result of the `index` semantics is returned and compared to zero. The object is truthy only if its length is non-zero.

```
1  from __compiler_intrinsics__ \
2  import class_getattr, define_semantics, absent
3
4  @define_semantics
5  def add(x, y):
6    def normal():
7      magic_method = class_getattr(x, "__add__")
8      if magic_method is absent:
9        return reflected()
10     else:
11       result = magic_method(x, y)
12       if result is NotImplemented:
13         return reflected()
14       else:
15         return result
16
17   def reflected():
18     magic_method = class_getattr(y, "__radd__")
19     if magic_method is absent:
20       raise TypeError
21     else:
22       result = magic_method(y, x)
23       if result is NotImplemented:
24         raise TypeError
25       else:
26         return result
27
28   return normal()
```

Figure 3. Semantics of the + operator written with the compiler intrinsics statement

[2] *Small* is implementation-dependent, but typically means an integer that fits in a machine word.

```
1  @define_semantics
2  def truth(obj):
3    magic_method = class_getattr(obj, "__bool__")
4    if magic_method is not absent:
5      result = magic_method(obj)
6      if type(result) is bool:
7        return result
8      else:
9        raise TypeError
10   else:
11     len_result = maybe_length(obj)
12     if len_result is not absent:
13       return len_result != 0
14     else:
15       return True
16
17 @define_semantics
18 def maybe_length(obj):
19   magic_method = class_getattr(obj, "__len__")
20   if magic_method is absent:
21     return absent
22   else:
23     len_result = magic_method(obj)
24     index_result = index(len_result)
25     if index_result < 0:
26       raise ValueError
27     else:
28       return index_result
```

Figure 4. Semantics of computing the truthiness of an object

```
1  from __compiler_intrinsics__ \
2  import class_getattr, define_semantics, absent, \
3         sint, bint
4
5  @define_semantics
6  def index(obj):
7    magic_method = class_getattr(obj, "__index__")
8    if magic_method is absent:
9      raise TypeError
10   else:
11     result = magic_method(obj)
12     if isinstance(result, sint):
13       return result
14     elif isinstance(result, bint):
15       raise OverflowError
16     else:
17       raise TypeError
```

Figure 5. Semantics of casting an object to an index-sized integer

3.4 Magic Methods

We cannot fully describe Python's semantics without describing the magic methods of its built-in types. For instance, the add semantics from Figure 3 fails to predict the specific result of the expression `(41 + 1.0)`. In this section, we introduce intrinsics to describe magic methods.

Applying an operation requires boxing and unboxing objects' values. An unboxed value is not a Python object. Its exact format depends on the *host* language used by a compiler, we thus call it a *host value*. To write magic methods, we need to express how host values are manipulated. Therefore, we introduce a family of intrinsic functions that are named `X_to_host` and `X_from_host`.

The intrinsic function `X_to_host` takes a single argument of type `X` and returns the host value of that argument. For example, the expression `int_to_host(42)` returns the numerical representation of 42 in the host language. If the argument is not an instance of `X`, then the behavior of the function is undefined.

The intrinsic function `X_from_host` is the inverse of `X_to_host`. It takes a host value as argument and returns an object of type `X` that encapsulates this value. While `X` could be any built-in type, we limit ourselves to numerical types such as `int` and `float` for now.

We also introduce the `builtin` intrinsic, which is similar to the `define_semantics` decorator. It is used as a *class decorator* and indicates that a given class definition is the definition of the corresponding built-in type.

In Figure 6, we use these new intrinsics to implement the `__add__` and `__floordiv__` (floor division) magic methods of `int`. Notice the redundant type check of both methods on lines 7 and 18 (yet, they are necessary when calling a magic method directly). In the case of `__floordiv__`, we also check that there is no division by zero on line 20. These magic methods introduce arithmetic operations in the host languages. In the expressions on lines 10, 20 and 22, the left-hand and right-hand sides are all host values. However, the usage of `int_to_host` can be detected statically, allowing to generate code for host integers addition. Throughout the remainder of this paper, examples will frequently show overloading of operators to execute arithmetic in the host language.

Magic methods defining the behavior of arithmetic operators are numerous, but they can be generated from templates to automate writing down the executable semantics [15].

For non-numerical types, it is straightforward to extend our pool of intrinsics to manipulate other types of host values. For example, we introduce the `str_len_to_host` intrinsic function, which takes a Python string as argument and returns a host integer representing its length. In Figure 7, we use it to implement the `__len__` method of `str` (string type).

```
1  from __compiler_intrinsics__ \
2  import builtin, int_from_host, int_to_host
3
4  @builtin
5  class int:
6    def __add__(self, other):
7      if isinstance(self, int):
8        if isinstance(other, int):
9          return int_from_host(
10                   int_to_host(self) +
11                   int_to_host(other))
12       else:
13         return NotImplemented
14     else:
15       raise TypeError
16
17   def __floordiv__(self, other):
18     if isinstance(self, int):
19       if isinstance(other, int):
20         if int_to_host(other) != int_to_host(0):
21           return int_from_host(
22                    int_to_host(self) //
23                    int_to_host(other))
24         else:
25           raise ZeroDivisionError
26       else:
27         return NotImplemented
28     else:
29       raise TypeError
```

Figure 6. The `__add__` and `__floordiv__` methods of `int`

```
1  from __compiler_intrinsics__ \
2  import builtin, int_from_host, str_len_to_host
3
4  @builtin
5  class str:
6    def __len__(self):
7      if isinstance(self, str):
8        return int_from_host(str_len_to_host(self))
9      else:
10       raise TypeError
```

Figure 7. The `__len__` magic method of `str`

3.5 Redundant Operations in the Semantics

Now that we defined some magic methods for `int` and `str` we can analyze the extent of the semantics's overhead. Consider what happens if we recover the truthiness value of a string. The `truth` semantics looks up for the `__bool__` method (fig. 4, line 3). Since this method cannot be found on `str`, the `__len__` method is looked up (fig. 4, line 19). So is the `__index__` method later on (fig. 5, line 7). Both the `__len__` and `__index__` methods are invoked.

Once we know that the object is a string, multiple checks are superfluous. For example, the `__len__` method checks the type of its argument (fig. 7, line 7). Furthermore, the length of a string will always be a positive small integer. Thus the whole invocation of the `index` semantics is unneeded, as well as the assertion that the length is positive (fig. 4, line 25).

A naive implementation of the `truth` semantics would execute these redundant operations. Yet, once we know that the object is a string, only recovering the length of the string (fig. 7, line 8) and checking whether it is non-zero (fig. 4, line 13) is relevant. The required computation boils down to `int_from_host(str_len_to_host(obj)) != 0`.

The same exercise with the expression `(1 + 2)` reveals redundant operations despite the required computation boiling down to `int_from_host(int_to_host(1) + int_to_host(2))`. Expressing the semantics of primitive operators using our formalism enables a compiler to implement that sort of optimization.

4 Behaviors

A compiler can implement arithmetic operators from the semantics defined in Section 3. Yet, by doing so in a naive way, that is calling each magic method, the implementation would likely offer poor performance.

We pointed out that the magic methods of built-in types cannot be changed. Given an operator and built-in types for its operands, we can thus predict which magic methods will be looked up and which of these will contribute to computing a result. This makes looking up or calling some magic methods superfluous, for instance if a method is known to be `absent` or if it can be predicted that it will return `NotImplemented`. We exploit that fact to generate optimized versions of Python operators.

We define a *behavior* to be a procedure that describes how to compute the result of an operator *for a given combination of built-in types* without redundant type checks or superfluous method calls. Behaviors are written in a similar fashion to operators' semantics by using the `define_behavior` intrinsic decorator, which behaves identically to `define_semantics`, but labels functions differently.

In Figure 8, we implement the behaviors for addition of an integer and a float (`add_intX_floatY`), floor division between two integers (`floordiv_intX_intY`) and truthiness of a string (`truth_strX`).

We use *`operation_ltype`*`X_`*`rtype`*`Y` as naming convention for behaviors, where *`operation`* is the short-circuited semantics, *`ltype`* is the required type of the left-hand operand and *`rtype`* is the required type of the right-hand operand. We also include the types in the annotation of the behavior (annotations are the types written after each argument and are part of Python's syntax) as it is more

```
1  @define_behavior
2  def add_intX_floatY(x: int, y: float):
3    return float_from_host(
4              int_to_host(x) + float_to_host(y))
5
6  @define_behavior
7  def floordiv_intX_intY(x: int, y: int):
8    if int_to_host(y) != int_to_host(0):
9      return int_from_host(
10               int_to_host(x) // int_to_host(y))
11   else:
12     raise ZeroDivisionError
13
14 @define_behavior
15 def truth_strX(x: str):
16   return int_from_host(str_len_to_host(x)) != 0
```

Figure 8. Behaviors for addition of an integer and a float (`add_intX_floatY`), integer floor division (`floordiv_intX_intY`) and string truthiness (`truth_strX`)

convenient for a compiler to read them from the annotation than from the behavior's name. Unary behaviors are written by omitting the right-hand type, for example `truth_strX`.

We can use partial evaluation to generate all behaviors for arithmetic operations on numeric types by identifying which methods return a result for each operator. This is made possible by the fact that a built-in magic method returns `NotImplemented` for a value of a given type if and only if it returns `NotImplemented` for all instances of that type.

Within a given magic method, most `if` statements' conditions are type checks that can be resolved from the operands' types. The only exceptions are division and bitwise-shift, which respectively check for zero division and negative shift. These are left to be evaluated at run time (see Figure 8, line 8).

5 A Partial Evaluator to Generate Behaviors

This section presents semPy, a Python tool for generating behaviors by removing redundant type checks, boxing and unboxing, and method calls whenever possible.[3]

semPy is a Python partial evaluator supporting the compiler intrinsics statement. It takes as inputs a semantics and a *context* that consists of built-in types for each of the arguments. It outputs a specialization of the semantics given that context, which is a behavior. The behaviors presented in Figure 8 were generated by semPy. For example, the `add_intX_floatY` was generated from the `add` semantics (Figure 3) in a context where the left-hand operand is an `int` and the right-hand operand is a `float`.

The structure of operators and built-in magic methods is sufficiently homogeneous that behaviors can be generated by

[3]The semPy source code is available online [16].

```
def __pos__(self):
  if isinstance(self, int):
    return int_from_host(int_to_host(self))
  else:
    raise TypeError
```

Figure 9. The `__pos__` magic method of `int`

using only three transformations: (1) aggressive inlining of method calls, (2) branch resolution based on type information and (3) removal of redundant boxing and unboxing.

5.1 Inlining

When a semantics or magic method is invoked, semPy systematically inlines the callee's code at the call site. This removes method calls from semantics specializations. Magic methods are returned by invocations of the `class_getattr` intrinsic function. This function is always called on the arguments of a semantics, whose types are provided in the type context, so it is always possible to resolve which method is to be called, or if that method is absent.

5.2 Branch Resolution

When semPy successfully computes the truthiness of the condition of an `if` statement, we can get rid of the branch that is not executed. Since semantics are written without using Python dynamic features, we can resolve the value of expressions that would normally be hard to evaluate statically. We can resolve conditions such as `isinstance(X, Y)`, which checks whether `X` is an object of type `Y`. Comparisons of the form `(magic_method is absent)` can always be resolved since built-in magic methods are immutable. We can also resolve comparisons of the form `(result is NotImplemented)`. In this case, we usually cannot infer the exact value of `result`, but we can at least infer that it is not the object `NotImplemented`.

Most branches are removed by resolving the aforementioned conditions. Some branches may still depend on the value of an object and can only be resolved if its origin provides sufficient information (such as lengths being non-negative). If not, the branch must be evaluated at run time.

5.3 Removal of Redundant Boxing and Unboxing

A naive implementation of Python's semantics sometimes causes unnecessary boxing and unboxing. For example, the `pos` semantics, which corresponds to unary +, is equivalent to the identity operation when applied to an integer. Yet, the magic method `__pos__` of `int` applies boxing and unboxing to account for the possibility that the argument is of a strict subtype of `int` in which case the result should be cast to an `int` (see Figure 9). A simple example is that of the expression `+True`, which must return 1.

```
@define_behavior
def pos_sintX(x: sint):
  return x

@define_behavior
def pos_boolX(x: bool):
  return int_from_host(int_to_host(x))
```

Figure 10. Removal of unboxing in unary + of `int` by semPy

In the context of a behavior where the argument is known to be of type `int`, semPy removes this type conversion. When the cast must occur, for example in the case of unary + on a `bool`, semPy preserves it as shown in the generated behaviors of Figure 10. This simplification occurs after inlining and branch resolution. At this point, unnecessary boxing manifests as chains of calls to primitives that are one another inverses and can be removed from the behavior.

5.4 The `test` Behavior

We present another example of unnecessary boxing removal. Consider the semantics of the `if` statement where Python evaluates the truthiness of a value and branches accordingly. This truthiness is determined by the `truth` semantics (see Figure 4), which returns either `True` or `False`. We show the behavior for truthiness of an integer in Figure 11. This behavior requires the `bool_from_host_bool` intrinsic function, which maps booleans in the host language to the corresponding Python boolean objects.

```
@define_behavior
def truth_intX(obj: int):
  return bool_from_host_bool(
           int_to_host(0) != int_to_host(obj))
```

Figure 11. Behavior for truthiness of `int`

In the condition of an `if`, this behavior takes the host result `int_to_host(0) != int_to_host(obj)` and converts it to a Python `bool` that the `if` statement immediately needs to convert back to a host boolean. Thus, the call to the intrinsic `bool_from_host_bool` is a form of unnecessary boxing.

We solve this by introducing the `test` semantics (fig. 12, line 2) and the intrinsic `bool_to_host_bool`, which acts as the inverse of `bool_from_host_bool`. The purpose of the `test` semantics is solely to express a variant of the `truth` semantics where we prefer the output to be a host boolean rather than a Python object. This semantics can be fed to semPy to return behaviors in which the unnecessary boxing was removed, such as the `test_intX` behavior (fig. 12, line 6).

This strategy is expandable to other cases where a condition is tested but a Python `bool` is not required, for example when the condition of an `if` statement is the result

```
1 @define_semantics
2 def test(obj):
3   return bool_to_host_bool(truth(obj))
4
5 @define_behavior
6 def test_intX(obj: int):
7   return int_to_host(0) != int_to_host(obj)
```

Figure 12. The `test` semantics and behavior of `test` for `int`

of a comparison. This would generally invoke one of the comparison semantics (`eq`, `ne`, `lt`, `le`, `gt` or `ge`), then check the truthiness of the result (Python comparison operators can return a value other than `True` or `False`). Instead, semPy can generate behaviors for those specific cases. To those behaviors we assign the names `test_`*`comp`*`_ltype`**`X`**`_rtype`**`Y`** where *`comp`* is the partially evaluated comparison semantics.

Figure 13 shows the result of semPy partial evaluation of `le(x, y)` (semantics of the <= operator) and its counterpart, the `test_le(x, y)` comparison where `x` and `y` are respectively an `int` and a `float`.[4]

```
@define_behavior
def le_intX_floatY(x: int, y: float):
  return bool_from_host_bool(
           float_to_host(y) >= int_to_host(x))

@define_behavior
def test_le_intX_floatY(x: int, y: float):
  return float_to_host(y) >= int_to_host(x)
```

Figure 13. The `le` and `test_le` behaviors for `int` and `float`.

6 Zipi: a Compiler Using Behaviors

We now detail how the tools described in this paper can be used to implement an optimizing Python compiler. We present Zipi, a compiler prototype that implements arithmetic operators and magic methods using the compiler intrinsics statement.

6.1 Zipi

The Zipi compiler [15] is an ahead-of-time (AOT) compiler from Python to Scheme [6]. It implements arithmetic operations using behaviors generated by semPy and extends this strategy to other operators. Zipi compiles Python to Scheme code, which is then compiled to an executable using either the Bigloo [22] or Gambit [10] Scheme compilers.

[4] In Figure 13, the `le_intX_floatY` and `test_le_intX_floatY` behaviors use the >= operator instead of the expected <= operator. Comparison magic methods can also return `NotImplemented`, which may lead to their reflected magic method to be called. In that case, the `__le__` method of `int` returns `NotImplemented` and the comparison resorts to the `__ge__` method of `float`.

```python
# A Python program that sums a list of integers
s = 0
for x in [1, 2, 3]:
    s = s + x
```

```
1  ; The main body of the code generated
2  ; by Zipi from the above program
3  (define x #f) ; #f indicates the variable
4  (define s #f) ; is not yet bound
5  (global-register! global (& "x")
6                    (lambda () x)
7                    (lambda (v) (set! x v)))
8  (global-register! global (& "s")
9                    (lambda () s)
10                   (lambda (v) (set! s v)))
11 (set! s (py-int-from-scheme 0))
12 (py-for-each
13   #:target
14   (py-make-list (py-int-from-scheme 1)
15                 (py-int-from-scheme 2)
16                 (py-int-from-scheme 3))
17   (begin
18     (set! x #:target)
19     (set! s
20       (py-add (or s (global-get (& "s")))
21               (or x (global-get (& "x")))))))
```

Figure 14. An example of Scheme code generated by Zipi

In Figure 14, we present a snippet of code generated by Zipi from a small Python program. The compiler maps most operations directly to a procedure or macro provided by Zipi's runtime system. For example, the forms `py-for-each` (line 12), `py-make-list` (line 14) and `py-add` (line 20) are all Scheme macros whose expansions implement `for`-loops, list allocations and addition, respectively. Only relevant parts of the generated code are shown and variable names have been demangled for readability.

Zipi supports all compiler intrinsics statement. In Figure 15, we show the Scheme version of the `add` semantics from Figure 3. Note that Zipi compiles the semantics to the `py-add-fallback` macro. The full semantics is used only as a fallback when no specialized behavior exists. Behaviors are generated and compiled once, at Zipi's build time.

When generating behaviors for Zipi, we distinguish between small integers (`sint`) and big integers (`bint`). This allows semPy to generate more specialized behaviors and the Zipi runtime system to further optimize integer arithmetic by representing small integers with Scheme fixnums.

In Figure 16, we show the compiled `add` behaviors for small integers. The `fx+?` operator applies small integer addition with an overflow check. In case of overflow, the `+2` operator applies addition and returns a Scheme big integer. The `py-bint-to-scheme` procedure is the Scheme equivalent of the `int_to_host` intrinsic for big integers.

```
1  (define-macro (py-add-fallback x y)
2    `(let ((x ,x) (y ,y))
3       (py-add-fallback:normal x y)))
4
5  (define (py-add-fallback:normal x y)
6   (let ((magic_method (getattribute-from-obj-mro
7                         x (&& "__add__"))))
8     (if (py-test-is magic_method py-absent)
9         (py-add-fallback:reflected x y)
10        (let ((result (py-call magic_method x y)))
11          (if (py-test-is result py-NotImplemented)
12              (py-add-fallback:reflected x y)
13              result)))))
14
15 (define (py-add-fallback:reflected x y)
16  (let ((magic_method (getattribute-from-obj-mro
17                        y (&& "__radd__"))))
18    (if (py-test-is magic_method py-absent)
19        (py-raise-binary-TypeError-fallback
20          (&& "+") x y)
21        (let ((result (py-call magic_method y x)))
22          (if (py-test-is result py-NotImplemented)
23              (py-raise-binary-TypeError-fallback
24                (&& "+") x y)
25              result)))))
```

Figure 15. Scheme version of the `add` semantics

```python
# Python add behavior for small integers
@define_behavior
def add_sintX_sintY(x: sint, y: sint):
    return int_from_host(int_to_host(x) +
                         int_to_host(y))
```

```
; add behavior compiled by Zipi
(define-macro (py-add-sintX-sintY-inline x y)
  `(let ((x ,x) (y ,y))
     (or (fx+? x y) (py-bint-to-scheme (+2 x y)))))

(define (py-add-sintX-sintY-fallback x y)
  (or (fx+? x y) (py-bint-to-scheme (+2 x y))))
```

Figure 16. `add_sintX_sintY` behavior compiled to Scheme

6.2 Behaviors in Zipi

To dispatch an operation to a specific behavior at run time, Zipi stores the procedures of each behavior within an array, called a *behavior array*. Each operator has its own behavior array, for example the *add behavior array* contains behaviors of addition. Once the type of each operand is known, it is

possible to recover the corresponding behavior from that operand's array and invoke it.

To recover behaviors from a behavior array, we assign a unique identifier to each type. We call this identifier the *class index* of a type. Since we generated separate behaviors for small integers and big integers, those have separate class indices despite having the same Python type. For example, small integers may have the class index `1`, big integers the index `2`, `bool` the index `3` and so on. We reserve the index `0` for all types that have no specialized behavior.

When an arithmetic operator is applied, we recover the class index of the types of each operand. In the case of unary operators, this index is the position of the corresponding behavior in that operator's behavior array. In the case of binary behaviors, we apply the formula `(right + N * left)` where `right` and `left` are the class indices of both operands and `N` is the number of existing class indices (Zipi currently has 17). The procedure at that computed index can be safely invoked without further type-checking.

In some cases, the procedure stored at the class index is not a behavior, but rather the full semantics without specialization. For example, if we add two objects whose type is user-defined, the resulting index will be `0`. The `add` behavior array contains the `py-add-fallback:normal` procedure (fig. 15, line 5) at that index.

A special case of the dispatch of a behavior happens when operands are both small integers or both `float` objects, which are represented by Scheme `fixnum` and `flonum` values respectively. Since those are common arithmetic operations, we inline the corresponding behavior for those cases. We limit inlining to those frequent use cases to avoid code bloat.

Figure 16 shows that Zipi generates two versions of each behavior. The `inline` version is a macro allowing to invoke a behavior inline while the `fallback` version is a first-class procedure, which we store in the behavior array.

In Figure 17, we show this inlining process with the `py-pos` macro, which implements the unary + operation. Whenever the operand `x` is either a `fixnum` (line 4) or a `flonum` (line 5), we execute the corresponding inline behavior. Otherwise, we recover the class index of the object and invoke the corresponding behavior from the `add` behavior array (line 7).

```
1 (define-macro (py-pos x)
2   `(let ((x ,x))
3      (cond
4        ((fixnum? x) (py-pos-sintX-inline x))
5        ((flonum? x) (py-pos-floatX-inline x))
6        (else
7         ((vector-ref py-pos-behaviors-array
8                      (py-obj-class-index x))
9          x)))))
```

Figure 17. Dispatch of the behavior for unary +

The code for dispatching behaviors is similar in the case of binary operators. Behaviors are inlined when both operands are either `fixnums` or `flonums`, otherwise the behavior is recovered from the corresponding behavior array. The same happens for comparisons, and the `truth` and `test` semantics.

7 Performance

In this section, we discuss Zipi's performance in comparison to CPython and PyPy [27], the current state-of-the-art implementation performance-wise. Performance was measured through microbenchmarks as well as regular benchmarks implementing well-known algorithms.

Initialization and compilation times vary across CPython, PyPy, and Zipi. CPython compiles code ahead-of-time (AOT) into bytecode that is then interpreted by a virtual machine. PyPy uses a tracing just-in-time (JIT) compiler [5]. Lastly, Zipi is AOT and has a deep pipeline that compiles Python code to Scheme, then to C, and finally to machine code[5]. As this occurs before execution, we do not consider it in this evaluation report. We configured our benchmarks to only measure the run time performance after initialization. We also allow PyPy's JIT to warm up by executing a dry run that does not count toward execution time for each benchmark. Benchmarks measure real time using the Python `time` module, which all implementations provide.

Results were generated by Forensics, our tool for tracking performance. Both Forensics' source code [8] and the benchmarks results are available online [9]. Benchmarks were executed on a machine with an Intel Core i7-7700K, 48 GB of RAM, and under Debian 10.13 with kernel version SMP Debian 4.19.269-1. We used CPython 3.9.0 with *profile guided optimization* enabled [29]. As for PyPy, we used version 7.3.5. Each PyPy release implements more than one version of Python, we used the newest version at the time, which was Python 3.7. Zipi was compiled with Gambit 4.9.3-1380, with `single-host` enabled, and GCC 10.3.

7.1 Microbenchmarks

We use microbenchmarks to evaluate the performance of individual operations and determine whether a targeted optimization, such as behaviors for arithmetic operators, is effective. The microbenchmarks have been useful to focus our optimization efforts on operations suffering from poor performance. The operation being evaluated is wrapped in a loop to reach a measurable time on the order of one second on CPython. To minimize the loop overhead, its body contains several repetitions of the measured operation (typically 20). The microbenchmarks allow a direct comparison between Zipi and CPython on individual operations. Unfortunately,

[5]To provide an idea of compilation time, the `deltablue` program discussed in Section 7.2 contains 440 lines of code and takes about 50 seconds to compile. This compilation time could be improved by compiling Python code directly to machine code.

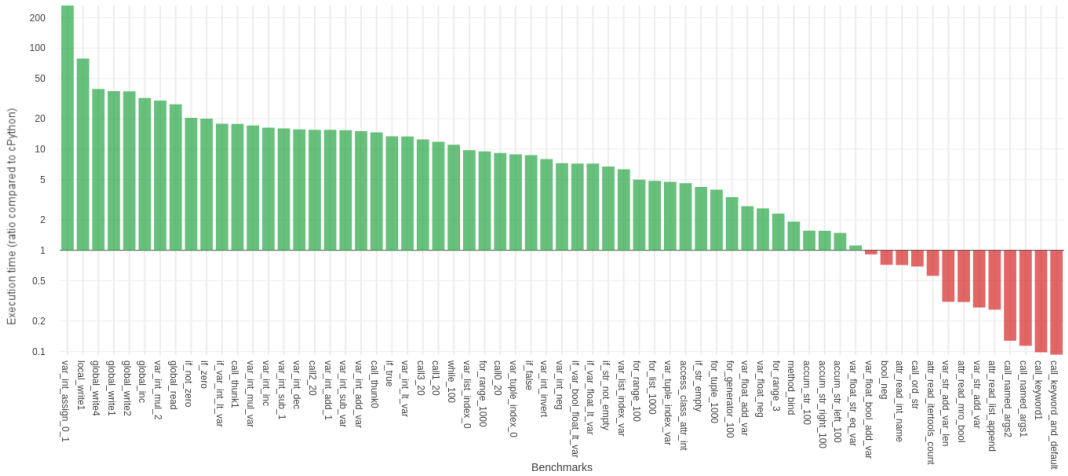

Figure 18. Microbenchmarks results of Zipi compared to CPython v3.9.0. A ratio higher than 1 (green) indicates an execution faster than CPython. A ratio lower than 1 (red) indicates a slower execution.

it does not allow a comparison with PyPy, which treats the kernel of many of our benchmarks as dead code. Neither Zipi nor CPython do this, so every operation is actually executed. Figure 18 shows the results of our microbenchmarks. All microbenchmarks are described in more details in [15].

Microbenchmarks indicate that behavior optimizations provide a significant performance boost for binary operators on small integers (between 15× and 30× faster) and `floats` (between 3.0× and 7.2×), truthiness of `bools` (between 8.7× and 14×), `ints` (20x) and `strs` (between 4.2× and 6.7×), and comparison between `ints` (18×) and `floats` (7.2×).

Performance improvements from other optimizations unrelated to behaviors also show up in the microbenchmarks. For instance, assignment to global variables, function calls and iteration on built-in types are all faster than with CPython. On the other hand, some microbenchmarks display poor performance. Those are unoptimized features that we implemented in a naive way, such as function calls with keyword arguments.

7.2 Benchmarks

We compared Zipi to CPython and PyPy using custom benchmarks and benchmarks from PyPerformance, an authoritative suite of benchmarks for Python [26]. Zipi being at an early development stage, only four benchmarks from PyPerformance are supported at the moment, hence the need for custom benchmarks.

Our custom benchmarks include `ack`, `fib`, `queens`, `bague` and `sieve`. The code for all custom benchmarks is available in [15]. Benchmarks from PyPerformance include `deltablue`, `fannkuch`, `richards` and `float` and are available online [30]. Each benchmark is executed once using parameters that result in a run time on the order of one second on CPython. Figure 19 compares the execution time of Zipi and PyPy using the CPython execution time as a baseline.

Zipi fares especially well on programs that extensively use small integer arithmetic: `ack` (38× faster than CPython), `fib` (24×) and `queens` (14×) execute faster than with PyPy. The `bague` (3.9×) and `sieve` (1.2×) benchmarks are slightly faster than CPython with Zipi. These benchmarks use small integer arithmetic, but also list and attribute access. The behavior optimization has a noticeable but limited effect in those cases. Finally, `fannkuch` (0.8×), `richards` (0.7×), `deltablue` (0.5×) and `float` (0.3×) execute slower than with CPython. These benchmarks make extensive use of user-defined types, which we did not optimize, our focus being on built-in types.

Overall, Zipi's performance on benchmarks making intensive use of small integer arithmetic rival with PyPy. Yet, this speedup does not translate to benchmarks that make a limited use of arithmetic. This is expected since behaviors specifically target arithmetic. We wish to extend the behavior optimization to other operations in the future to further analyze its impact on performance.

7.3 Threats to Validity

The validity of our results faces the common potential issues of assessing the performance of a prototype compiler.

Despite implementing Python's core features, including those identified as the main source of overhead in CPython (see Section 2), Zipi only supports a subset of the language. It lacks features such as threads, `async` functions, and most of the standard library. It remains to measure the impact of introducing these features in our prototype.

Our benchmarks show a clear performance increase when executing arithmetic-heavy programs. Nonetheless, the absence of most modules from Python's standard library limits our ability to measure the extent of this speed up on real-life programs. The PyPerformance benchmark suite also makes use of external libraries (such as `django`, a high-level

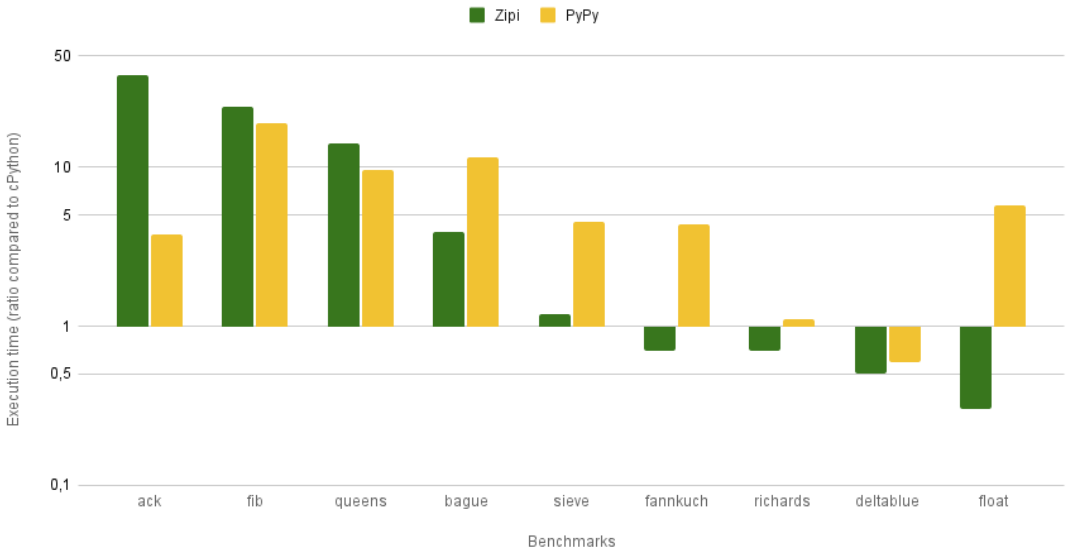

Figure 19. The execution time of Zipi (green) and PyPy3.7 v7.3.5 (yellow) is compared to that of CPython v3.9.0. A ratio higher than 1 indicates an execution faster than CPython. A ratio lower than 1 indicates a slower execution.

web framework written in Python) [25], which prevents executing some of its benchmarks with Zipi.

8 Related Work

We attribute the first instance of compiler generation from a formal semantics to Mosses [18], who developed a compiler generator based on denotational semantics. However, the generated compilers were inefficient. Mosses later outlined the pragmatic issues of denotational semantics for compiler generation. First, extension to a language's semantics often requires to completely reformulate the denotational semantics. Furthermore, denotational semantics fail to convey how a program must be executed, hindering the generation of performant compilers [19].

Executable semantics have been implemented for various languages, including C [7], Java [4], JavaScript [2], LLVM IR [33], Lua [24], PHP [11], POSIX shell [13], Python [21], and R [3]. Nowadays, these typically employ frameworks such as K [4, 7, 11], Redex [24], or a proof assistant such as Coq to extract an executable semantics [2, 3, 13, 33]. This generally results in significantly slower implementations than modern, hand-optimized compilers.

Politz et al. [21] proposed an alternative strategy for defining a Python executable semantics. It involves translating code into a lambda calculus equipped with key features such as method lookup. While not focused on performance, the technique demonstrates how the semantics can be described by desugaring code into key features.

Our approach was inspired by the RPython experiment, which allows to express high-level details about a language's semantics while remaining easy to analyze statically [1].

9 Conclusion

We presented an approach to define an executable semantics for Python operators allowing reuse in optimizing compilers. We expressed this semantics using a syntax similar to that of Python for seamless integration to an existing compiler. Our approach enhances Python with primitive functions to describe operations at a lower level. This allows us to define the notion of *behavior*, a specialization of an operator for a given combination of built-in types. In particular, we showed how behaviors remove redundant type checks, magic method calls, boxing and unboxing.

We implemented semPy, a tool for partial evaluation of the semantics, to generate behaviors automatically. The overall structure of Python's operators and magic methods allows to generate behaviors using straightforward function inlining and branch resolution.

We integrated these behaviors to Zipi, an AOT optimizing Python compiler prototype. Zipi dispatches operations to their corresponding behaviors at run time. This increases execution speed, offering performance that rivals PyPy. Although this speedup is limited to arithmetic-heavy programs, behaviors could be extended to other operations or serve alongside other optimization techniques.

We hope semPy and the behavior optimization can contribute to the ongoing optimization efforts of Python implementations. It appears to us that they would be well suited for CPython, as they specifically address the known overhead of this implementation.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. New York, NY, USA, 53–64. https://doi.org/10.1145/1297081.1297091

[2] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. 87–100. https://doi.org/10.1145/2535838.2535876

[3] Martin Bodin, Tomás Diaz, and Éric Tanter. 2020. A Trustworthy Mechanized Formalization of R. *SIGPLAN Not.* 53, 8 (apr 2020), 13–24. https://doi.org/10.1145/3393673.3276946

[4] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. *SIGPLAN Not.* 50, 1 (jan 2015), 445–456. https://doi.org/10.1145/2775051.2676982

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. 18–25. https://doi.org/10.1145/1565824.1565827

[6] Dybvig, Kent. 2009. *The Scheme Programming Language* (fourth edition ed.). The MIT Press. https://www.scheme.com/tspl4/.

[7] Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. 533–544. https://doi.org/10.1145/2103656.2103719

[8] Marc Feeley and Marc-André Bélanger. 2023. Forensics. https://github.com/udem-dlteam/forensics/.

[9] Marc Feeley and Marc-André Bélanger. 2023. Zipi Forensics. https://zipi-forensics.gambitscheme.org/.

[10] Feeley, Marc. 2023. Gambit. https://www.iro.umontreal.ca/~gambit/doc/gambit.pdf.

[11] Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). 567–592.

[12] Gouy, Isaac. 2023. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[13] Michael Greenberg and Austin J. Blatt. 2019. Executable Formal Semantics for the POSIX Shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (dec 2019), 30 pages. https://doi.org/10.1145/3371111

[14] Mohamed Ismail and G. Suh. 2018. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Raleigh, NC, USA, 36–47. https://doi.org/10.1109/IISWC.2018.8573512

[15] Olivier Melançon. 2022. *Reusable Semantics for Implementation of Python Optimizing Compilers*. M.Sc. Thesis. https://papyrus.bib.umontreal.ca/xmlui/handle/1866/26538.

[16] Olivier Melançon. 2023. semPy. https://github.com/omelancon/semPy.

[17] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.17

[18] Peter D. Mosses. 1975. *Mathematical semantics and compiler generation*. Ph.D. Thesis. University of Oxford. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.466424

[19] Peter D. Mosses. 1996. Theory and Practice of Action Semantics. *MFCS '96* 1113, 37–61. https://doi.org/10.1007/3-540-61550-4_139

[20] Mozilla. 2023. Primitive. https://developer.mozilla.org/en-US/docs/Glossary/Primitive.

[21] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty — A Tested Semantics for the Python Programming Language. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

[22] Serrano, Manuel. 2023. Bigloo. https://www-sop.inria.fr/mimosa/fp/Bigloo.

[23] Simionato, Michele. 2023. The Python 2.3 Method Resolution Order. https://www.python.org/download/releases/2.3/mro.

[24] Mallku Soldevila, Beta Ziliani, and Bruno Silvestre. 2022. From Specification to Testing: Semantics Engineering for Lua 5.2. *J. Autom. Reason.* 66, 4 (nov 2022), 905–952. https://doi.org/10.1007/s10817-022-09638-y

[25] Stinner, Victor. 2023. Benchmarks — Python Performance Benchmark Suite. https://pyperformance.readthedocs.io/benchmarks.html.

[26] Stinner, Victor. 2023. The Python Performance Benchmark Suite. https://pyperformance.readthedocs.io.

[27] The PyPy Team. 2023. PyPy. https://www.pypy.org.

[28] The Python Software Foundation. 2023. cPython 3.9. https://www.python.org.

[29] The Python Software Foundation. 2023. Performance Options. https://docs.python.org/3/using/configure.html#performance-options.

[30] The Python Software Foundation. 2023. PyPerformance. https://github.com/python/pyperformance.

[31] The Python Software Foundation. 2023. The Python Language Reference. https://docs.python.org/3/reference/index.html.

[32] TIOBE Software BV. 2023. TIOBE Index for August 2023. https://www.tiobe.com/tiobe-index/.

[33] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

[34] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. 2022. Quantifying the Interpretation Overhead of Python. *Science of Computer Programming* 215, C (March 2022). https://doi.org/10.1016/j.scico.2021.102759

A Compiler Intrinsics

This appendix describes the full specification of all intrinsics encountered in this paper. Some intrinsics' names have been shortened in the context of this paper for readability.

`define_semantics`

The `define_semantics` intrinsic is used as a function decorator. It indicates that a given definition is not that of a Python function, but rather the definition of a semantics. Thus, the compiler does not need to allocate a `function` object and is free to store the semantics in its preferred format (such as a function in the host language). The targeted semantics is defined by the name of the function. Labelling a function with `define_semantics` declares to the compiler that, within the code of the semantics, the compiler can assume that:

1. Built-in names such as `int` and `isinstance` have their standard binding;
2. The built-in functions `globals()`, `locals()`, `vars()` and `super()` are never called;
3. No global variable is used except to refer to other semantics

This precludes the use of problematic Python features, which in turn allows the compiler to apply optimizations such as inlining built-in function calls. Preventing the usage of global variables allows the compiler to skip the creation of a module altogether as it removes the need for a dynamic global environment. The behavior of `define_semantics` is undefined if not used as a function decorator.

`class_getattr`

The `class_getattr` intrinsic function takes a Python object and a string literal as arguments. It traverses the MRO of the object's type to recover the attribute specified by the string literal. If the attribute is found, it is returned. Otherwise, the value `absent` is returned to indicate that the attribute was not found. Figure 20 shows a pseudocode implementation of `class_getattr`. In most cases, the result of a call to `class_getattr` is a magic method. However, due to Python's dynamic nature, any object could be returned in which case calling the returned value may raise an exception. The behavior of `class_getattr` is undefined if it is called with anything but the aforementioned arguments.

```
class_getattr(obj, name):
  for each class in the mro of type(obj):
    if class has an attribute name:
      return class.name
  return absent  # instrinsic value 'absent'
```

Figure 20. Pseudocode for the `class_getattr` intrinsic

`absent`

The `absent` intrinsic is a primitive value similar to the JavaScript `undefined` [20]. It has an *identity* and can be compared with the `is` operator. It is not a Python object and so any other operation on it is undefined.

`sint`

An abstract subtype of `int` representing *small* integers. It is not a proper Python type, but allows to differentiate between small and big integers using the `isinstance` built-in function, while leaving room for implementation-dependent details regarding the exact threshold between small and big integers. For instance, `isinstance(x, sint)` returns `True` if `x` has type `int` and is a small integer, and returns `False` otherwise. Usage of `sint` in another context than as second argument of `isinstance` is undefined.

`bint`

Similar to `sint`, but for *big* integers.

`builtin`

The `builtin` intrinsic is used as a *class* decorator. It indicates that a class definition is the definition of the corresponding built-in type. Similarly to the `define_semantics` decorator, it declares that the class body does not use Python's most dynamic features: built-in names have their standard binding, no calls to `globals()`, `locals()`, `vars()` and `super()` occur and no global variable is used.

`define_behavior`

The `define_behavior` intrinsic is used as a function decorator. It indicates that a function is the definition of a behavior. Similarly to the `define_semantics` decorator, it declares that the function does not use Python most dynamic features.

`X``_from_host`

A family of primitive functions where *`X`* can be any built-in type, although we limit ourselves to `int` and `float` in the scope of this paper. The primitive *`X`*`_from_host` takes the host representation of an object of type *`X`* and returns the corresponding Python object of type *`X`*. This applies the operation of boxing a native value in a Python object.

`X``_to_host`

The primitive *`X`*`_to_host` takes a Python object of type *`X`* and returns the corresponding native representation of the object. This applies the operation of unboxing a native value from a Python object.

There exists one case where *`X`*`_to_host` does not behave as the inverse of *`X`*`_from_host`. The `bool` type is a subtype of `int` and the boolean values, `True` and `False`, are respectively equal to `1` and `0`. Thus, `int_from_host(int_to_host(True))` must in fact return `1`. This is why the `bool_from_host_bool` and `bool_to_host_bool` primitive functions are required.

`bool_from_host_bool`

A primitive function that maps booleans in the host language to the corresponding Python boolean. It does not allocate a new object, since Python booleans are singleton objects.

`bool_to_host_bool`

A primitive function that maps Python booleans to the host language representation of booleans. This function is the inverse of the `bool_from_host_bool` intrinsic.

`str_len_to_host`

A primitive function that returns the length of a Python string as an integer in the host language.

Received 2023-07-07; accepted 2023-09-01