Université de Montréal

Reusable Semantics for Implementation of Python Optimizing Compilers

par

Olivier Melançon

Département d'Informatique et de Recherche Opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Août 2021

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Reusable Semantics for Implementation of Python Optimizing Compilers

présenté par

Olivier Melançon

a été évalué par un jury composé des personnes suivantes :

Michalis Famelis
(président-rapporteur)

Marc Feeley
(directeur de recherche)

Stefan Monnier
(membre du jury)

Résumé

Le langage de programmation Python est aujourd'hui parmi les plus populaires au monde grâce à son accessibilité ainsi que l'existence d'un grand nombre de librairies standards. Paradoxalement, Python est également reconnu pour ses performances médiocres lors de l'exécution de nombreuses tâches. Ainsi, l'écriture d'implémentations efficaces du langage est nécessaire. Elle est toutefois freinée par la sémantique complexe de Python, ainsi que par l'absence de sémantique formelle officielle.

Pour régler ce problème, nous présentons une sémantique formelle pour Python axée sur l'implémentation de compilateurs optimisants. Cette sémantique est écrite de manière à pouvoir être intégrée et analysée aisément par des compilateurs déjà existants.

Nous introduisons également `semPy`, un évaluateur partiel de notre sémantique formelle. Celui-ci permet d'identifier et de retirer automatiquement certaines opérations redondantes dans la sémantique de Python. Ce faisant, `semPy` génère une sémantique naturellement plus performante lorsqu'exécutée.

Nous terminons en présentant Zipi, un compilateur optimisant pour le langage Python développé avec l'assistance de `semPy`. Sur certaines tâches, Zipi offre des performances compétitionnant avec celle de PyPy, un compilateur Python reconnu pour ses bonnes performances. Ces résultats ouvrent la porte à des optimisations basées sur une évaluation partielle générant une implémentation spécialisée pour les cas d'usage fréquent du langage.

Mots clés: Python, langage de programmation dynamique, sémantique formelle, évaluation partielle, compilateur, optimisation

Abstract

Python is among the most popular programming language in the world due to its accessibility and extensive standard library. Paradoxically, Python is also known for its poor performance on many tasks. Hence, more efficient implementations of the language are required. The development of such optimized implementations is nevertheless hampered by the complex semantics of Python and the lack of an official formal semantics.

We address this issue by presenting a formal semantics for Python focussed on the development of optimizing compilers. This semantics is written as to be easily reusable by existing compilers.

We also introduce `semPy`, a partial evaluator of our formal semantics. This tool allows to automatically target and remove redundant operations from the semantics of Python. As such, `semPy` generates a semantics which naturally executes more efficiently.

Finally, we present Zipi, a Python optimizing compiler developped with the aid of `semPy`. On some tasks, Zipi displays performance competing with those of PyPy, a Python compiler known for its good performance. These results open the door to optimizations based on a partial evaluation technique which generates specialized implementations for frequent use cases.

Keywords: Python, dynamic programming language, formal semantics, partial evaluation, compiler, optimization

Contents

List of figures

List of acronyms and abbreviations

AOT	Ahead-of-time
AST	Abstract Syntax Tree
JIT	Just-in-time
MRO	Method Resolution Order
RTE	Runtime Environment

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu possible cette maîtrise. Mener à bien un tel projet en contexte de pandémie mondiale est une tâche essouflante qui n'aurait été possible sans le soutien de chacune des personne mentionnées ci-bas.

Je remercie d'abord mon directeur, Marc Feeley, pour son soutien, sa disponibilité et surtout son incroyable générosité. Je m'estime chanceux d'avoir été si bien accompagné.

Je remercie toute l'équipe du laboratoire de recherche sur les langages dynamiques de l'Université de Montréal. Travailler à leur côté, même à distance, a été un réel plaisir. Je remercie tout particulièrement Léonard Oest O'Leary et Marc-André Bélanger pour leur travail sur les nombreux outils présentés dans ce mémoire.

Je remercie Manuel Serrano de l'Institut national de recherche en sciences et technologies du numérique de France avec qui j'ai eu le privilège de collaborer tout au long de ma maîtrise.

Je remercie mes amis, Maude Gauthier et Olivier Leduc, qui m'ont permis d'éviter l'isolement qu'aurait autrement occasionné une maîtrise en temps de pandémie.

Je remercie mes parents sans le support et les encouragements desquels je n'aurais pas le privilège de rédiger aujourd'hui ce mémoire. Je remercie également mon frère Nicolas et ma soeur Pascale d'être des inspirations dans leurs parcours académiques respectifs.

Je ne pourrais terminer autrement qu'en remerciant ma conjointe, Andréanne Dufour. Lors de l'été 2019, nous avons décidé de nous inscrire tous deux à la maîtrise. Je n'aurais pu choisir une meilleure complice avec qui traverser ces dernières années pleines de rebondissements. Merci de me pousser à chaque jour à offrir le meilleur de moi-même.

Introduction

Programming languages are oftentimes categorized as either static or dynamic. This commonly refers to the type system of the language. Statically typed languages apply type checking during the compilation of a program, either by requesting that the programmer declares variables' types or through type inference. On the other hand, dynamically typed languages assign types to values at run time and apply type checking during a program execution.

However, the *dynamicity* of a language refers to more than its type system. More broadly, dynamic languages are those which defer operations otherwise done at compile time to the program execution. Beyond type checking, such operations can include late binding, dynamic code loading and garbage collection [**1**].

Unsurprisingly, defering operations to run time makes dynamic languages generally slower than static languages [**2, 1**]. Nonetheless, dynamic languages have seen a steady rise in popularity in the last decades. Programmers seem to favor shallower learning curves [**1**], shorter development time [**2**] and extensive built-in librairies [**3**] over performance alone. Following this trend, highly dynamic languages such as JavaScript and Python have rised among the most popular languages in the world [**4, 5**].

This inevitably raises the issue of optimizing such highly dynamic languages. In particular, we turn our focus onto Python whose dynamic features extend well beyond the aformentionned run time operations. Along with dynamic type checking, late binding, dynamic code loading and garbage collection, Python also offers a deep level of introspection which renders static analysis of a Python program especially hard.

In some sense, Python is *more* dynamic than other languages. Hence, we expect optimization techniques applicable to the Python language to be portable to other dynamic languages.

When the Python language was released, its author, Guido Van Rossum, intended a language focussed on readability and offering powerful data types out of the box [**6**]. Today, Python is indeed praised as being both powerful and (superficially) simple [**3**]. Yet, while it is true that Python's extensive standard libraries and built-in functions offer high expressiveness, the Python language also has a standard implementation which yields poor performance

[**7**]. Moreover, a complex semantics without any formal specification [**8**] makes it difficult to reason about Python [**9**].

In our experience, both of these challenges which Python faces are related. Its complex semantics and absence of formal specification complicate the development of a Python compiler compatible with the standard implementation. This in turn leaves less time for optimization.

When initially developping the Python compiler presented in Chapter 4, a large part of our time was simply devoted to *getting the semantics right*. Furthermore, realizing that a specific optimization contradicts part of the semantics midway through development has not been uncommon either. With this came the desire to automate part of the implementation of the semantics in a way which still yields good performance.

Similar concerns have been reported with the JavaScript language [**10, 11**] whose specifications counts no less than 875 pages [**12**]. We suggest the development of executable semantics aimed at automating the development of optimizing compilers as a solution to this recurring issue.

Contributions

This thesis makes three main contributions.

- The development of a formal semantics for Python which can be easily integrated to an existing Python compiler (Chapter 2).
- A tool to automatically generate fast paths for executing Python arithmetic operations using the formal semantics (Chapter 3).
- A technique to automate parts of the implementation of a Python optimizing compiler using this tool (Chapter 4).

Chapter 1

Background

In this chapter, we discuss useful background to understand the contributions of this thesis. We assume basic knowledge about Python's syntax and provide an overview of relevant parts of the language's semantics. We then discuss Python's dynamic features and their impact on the implementation of optimizing compilers.

1.1. The Python language

Python is a high-level, general-purpose, multi-paradigm programming language [**13**]. It is most well known for its extended standard library [**14**], object-oriented approach and admittedly poor performance [**7**].

There exists two distinct versions of the Python language, `Python2` and `Python3`. Both are dissimilar enough that they can effectively be treated as different languages [**15**]. In this thesis, all mentions to *Python* refer to the `Python3` language.

The specifications of the Python language are described in *The Python Language Reference* [**16**]. We will refer to this document when describing and implementing Python's semantics. Yet, the language reference does not provide a formal specification, except for syntax and lexical analysis, which leaves room for ambiguities. When such ambiguities arise, we refer to the behavior of cPython [**13**], the reference implementation.

```
def adder(x):
    def add_x(y):
        return y + x    # y: local scope, x: enclosing non-global scope
    return add_x

add_2 = adder(2)        # add_2: global scope

print(add_2(3))         # print: builtin scope
```

Fig. 1.1. A sample Python program

In Figure 1.1, we display a small Python program. Note how Python uses indentation levels to delimit nested code blocks and variables do not have a static type.

1.1.1. Scoping rules

When encountering a reference to a variable, Python resolves it as either part of the local, enclosing, global or *builtins* scope.

The local scope is delimited by a function's body. When an assignment to a name exists within the body, any reference to this name within the body is labelled as local. In the event a local variable is referenced before assignment, Python throws a run time exception.

The enclosing scope exists when defining a nested function. In that case, free variables for which an assignment exists only in the body of an enclosing function are part of the enclosing scope. Similarly to the local scope, a reference to an unbound variable labelled as part of the enclosing scope results in a run time exception being thrown.

Importantly, Python rules for local and enclosing scopes depend solely on the closest lexical assignment to a variable and are independent of whether the variable will be defined at run time. Thus, both the local and enclosing scopes can be resolved statically.

When a variable is neither local, nor part of an enclosing scope, Python looks up its name in the global scope. The global scope contains all definitions executed at the top-level of a script. If the lookup of a global name fails, Python then searches the *builtins* scope. The builtins scope contains functions and objects provided by the `builtins` module such as `print`, `range` and `len`. Finally, if the variable is not found within the builtin scope, Python raises a run time exception. Figure 1.1 shows a program making use of all four scopes.

As we shall explain in Section 1.2, determining whether a variable is global or builtin can only be achieved at run time due to the ability to delete global variables at run time.

1.1.2. Data model

Python's abstraction for data are *objects* [**17**], which are entirely defined by their *identity*, *type* and *value*. The *identity* is a unique integer which 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 it can be an integer, a floating point number or a pointer to another data structure. An object is sometimes *mutable* such as with lists and dictionaries whose content can change. In other cases, it is *immutable* such as with integers and floats whose value cannot be updated. Finally, the *type* determines the operations allowed on that object, it is obtainable with `type(obj)`. Under certain conditions, the type of an object can be modified. However, this is not possible for objects whose type is a built-in type [**17**].

All values in a Python program are objects. This is by opposition to other object-oriented languages such as Java where primitive data types such as `byte`, `char` or `boolean` exist [**18**].

For example, Python boolean values are represented by the singleton objects `True` and `False` which belong to the `bool` type. Likewise, Python offers a `None` value which fulfills a similar role to that of the JavaScript `null` [**19**]. However, `None` is not a primitive value but rather a singleton object which belongs to the class `NoneType`.

In the absence of primitive values, the operations allowed on an object are defined entirely by its *type*. The Python language features basic types such as `bool` (boolean), `int` (unlimited integers), `float`, `complex` and `str` (strings). It also offers container types such as `tuple` (fixed size array), `list` (dynamic array), `dict` (hash-table) and `set`. Custom types can also be defined using the `class` keyword.

```
py_add(x, y):
  """Semantics of the Python + operator"""
  if type(x) has a method __add__:
    result = type(x).__add__(x, y)

    if result is NotImplemented:
      return py_radd(y, x)
    else:
      return result
  else:
    return py_radd(y, x)

py_radd(y, x):
  """
  Reflected fallback for the Python addition operator
  """
  if type(y) has a method __radd__:
    fallback_result = type(y).__radd__(y, x)

    if fallback_result is NotImplemented:
      raise TypeError
    else:
      return fallback_result
  else:
    raise TypeError
```

Fig. 1.2. Pseudocode of the semantics of the `+` operator

The type of an object is itself an object which defines a set of methods. Whether an object supports a given operation depends on the methods available on its type. For example, the addition operation (`x + y`) is defined if the type of `x` possesses a method named `__add__`. If this method exists, it is invoked with `x` and `y` as arguments which yields a result for the operation `x + y`. In the event where the `__add__` method is not found on the type of `x`, Python will fall back on the `__radd__` method of the type of `y` (note that the `r` in the name stands for *reflected*). There is also the possibility that the `__add__` method exists, but does not know how to handle the object `y`. In that case, the method will return the singleton object

`NotImplemented`. This indicates to Python that despite the method's existence, its result should not be used in that case and the `__radd__` method should be called as a fallback [**20**]. Figure 1.2 expresses semantics of addition in a pseudocode style similar to the Python syntax, the reason for that choice will become apparent in Chapter 2. The result of `py_add(x, y)` is the same as `x + y`.

Python implements semantics of other arithmetic operations in a similar fashion, where only the name of the required methods change. For example, the subtraction semantics is identical to the addition semantics, but with the required methods being named `__sub__` and `__rsub__`. The methods required by the semantics are called *special methods* or *magic methods*.

```
class Cons:
    """
    Implementation of Lisp-like pair
    A sequence of Cons must end with None which acts as the empty list
    """
    def __init__(self, car, cdr):
        self.car = car
        self.cdr = cdr

    def __iter__(self):
        yield self.car
        if self.cdr is not None:
            yield from self.cdr

    def __add__(self, other):
        if self.cdr is None:
            return Cons(self.car, other)
        else:
            return Cons(self.car, self.cdr + other)

    def __str__(self):
        return "(" + " ".join(map(str, self)) + ")"

l1 = Cons(1, Cons(2, None))
l2 = Cons(3, Cons(4, Cons(5, None)))

print(l1 + l2)  # (1 2 3 4 5)
```

Fig. 1.3. Python implementation of Lisp-like pairs

Magic methods play a role beyond arithmetic. For almost all operations in Python, there exists a set of special methods which dictates the semantics of the operation. For example, iteration in a `for`-loop relies on the `__iter__` method to return an iterator. The type of this iterator must implement the `__next__` method which yields the element on which to iterate. The syntactical form `obj.attr` for attribute access relies on the `__getattribute__` method and falls back on the `__getattr__` method if no attribute is found. The same applies for

function invocation, truthiness of objects used as conditions, type casting and so on. Figure 1.3 shows an example of a user-defined data structure `Cons` which mimics Lisp's `cons` to implement linked lists. The `__init__` magic method defines the behavior of an object when initialized. Implementing `__iter__` defines what it means to iterate over a `Cons`. Namely, it allows to use `map` to map a function onto a `Cons`. Defining `__add__` adds supports for addition, which implements list concatenation. Finally, `__str__` defines how instances should be printed.

1.1.3. Inheritance

Inheritance is a feature through which a type inherits properties from a parent type. In the context of Python, the inherited properties are methods, including magic methods. In particular, inheritance allows to override the behavior of a type by redefining magic methods. In Figure 1.4, we provide an example where we use inheritance to extend the `Cons` class shown in Figure 1.3 to support concatenation with lists. We achieve this by defining a new `__add__` method on the subclass `MyNewCons`.

```
class MyNewCons(Cons):
    """A subclass of Cons supporting concatenation with 'list' objects"""
    def __add__(self, other):
        if self.cdr is None:
            return Cons(self.car, other)
        elif isinstance(other, list):
            return list(self) + other
        else:
            return Cons(self.car, self.cdr + other)

l1 = MyNewCons(1, MyNewCons(2, None))
l2 = [3, 4, 5]

print(l1 + l2)  # [1, 2, 3, 4, 5]
```

Fig. 1.4. Implementation of Lisp-like pairs with inheritance

When recovering a method from a type, Python executes an ordered search across the type and its parents. For example, when iterating over an object of type `MyNewCons` (Figure 1.4), the method `__iter__` is looked up on the `MyNewCons` where it is not found. Python then looks up the method on the parent type `Cons` where it finds the method, thus allowing for iteration over a `MyNewCons` object.

The Python language also supports *multiple inheritance*. This allows a type to inherit from more than one parent. In such cases, more than one way to traverse the tree of parents of a type exists. To avoid inconsistencies, searching for a magic method (or any class attribute) thus requires an order in which the parents are traversed. This order is called the *method resolution order* (MRO). The MRO is a property of a type which is computed during the

creation of the `type` object and cannot be altered afterward. The MRO is computed using the *C3 superclass linearization* algorithm. This algorithm has the advantage of preserving monotonicity: it prevents inherited attributes from skipping over all direct parent classes [**21, 22**].

1.1.4. Magic method driven semantics

It should now be apparent that a significant part of Python's semantics is driven by its so-called magic methods. However, this does not apply to the whole semantics, which leads us to differentiate between parts of the semantics which are magic method driven and which are not. We say a part of the semantics is *magic method driven* if it can be overloaded by defining the relevant magic method of a class as in Figure 1.4.

As previously mentioned in Section 1.1.2, the semantics of binary arithmetic operators (`+`, `-`, `*`, `/`, `//`, `%`, `**` and `@`), bit-wise operators (`<<`, `>>`, `|`, `&` and `^`), unary operators (`+`, `-` and `~`) and comparison operators (`<`, `<=`, `>` and `>=`, `==`, `!=` and `in`) are all magic method driven.

The same applies for the following syntactic forms.

- `if` `while` statements and which use `__bool__` or `__len__`
- `for`-loops (`__iter__` and `__next__`)
- function invocation (`__call__`)
- instance creation (`__init__` and `__new__`)
- `with`-statements (`__enter__` and `__exit__`)
- attribute access (`__getattribute__`, `__getattr__` and `__setattr__`)
- subscription (`__getitem__` and `__setitem__`)
- the `del`-statement (`__del__`)
- class declaration (`__prepare__` and `__new__`)

The semantics of the following built-in functions is also magic method driven.

- `abs` (`__abs__`)
- `dir` (`__dir__`)
- `divmod` (`__divmod__`)
- `format` (`__format__`)
- `hash` (`__hash__`)
- `isinstance` (`__instancecheck__`)
- `len` (`__len__`)
- `repr` (`__repr__`)
- `reversed` (`__reversed__`)
- `round` (`__round__`) `__new__`)
- `subclass` (`__subclasscheck__`)

Similarly, the semantics of type casts (explicit or implicit) is driven by magic methods. For example, when a cast to an `int` is required, the `__int__` magic method is invoked. The same takes place for casts to a `float` (`__float__`), a `str` (`__str__`), a `bool` (`__bool__`) and so on.

There also exists a few features whose semantics is not magic method driven, namely control-flow keywords (`break`, `continue`, `raise`, `return` and `yield`), scoping rules, function declaration and the `try-except` statement.

Through this thesis, we will focus on the magic method driven parts of the semantics. This choice is motivated by the fact those constitute a substantial part of the semantics. In particular, this is a part of the semantics of Python that is highly dynamic (Section 1.2.3) and thus has a significant impact on performance (Section 1.3). It is also the most atypical part of the semantics, since the lexical scoping and control flow of Python are similar to those of other languages and are straightforward to understand and implement. In general, we will thus use the term *semantics* to refer to magic method driven semantics unless stated otherwise. In particular, we will often use the term *semantics of* a specific feature, by which we mean the way relevant magic methods are composed to implement the feature (we already provided an example of the *semantics of addition* in Figure 1.2).

1.1.5. Decorators

Python's decorators are a syntactic sugar used throughout this thesis. A decorator is an expression prefixed with the `@` character which precedes a function definition. The result of the expression must be a callable which is invoked at run time with the function or class it decorates. The value returned by the invocation is then bound to the function or class name instead of the initial object itself [**23**].

Decorators are typically used to either create a wrapper function (Figure 1.5) or register functions with some properties (Figure 1.6).

In Chapter 2, decorators will be used as a way to label certain functions for the compiler. This usage is similar to that of Figure 1.6 which registers specific functions. However, it will take place at compile time.

1.2. Dynamic features

Similarly to other dynamic programming languages, Python incorporates features such as dynamic typing, dynamic binding and dynamic code evaluation. Python also offers a deep level of introspection which allows altering the behavior of a program in ways which a compiler can hardly predict. Introspection is the ability to manipulate reified objects such as lexical environments, stack frames or the garbage collector. While *The Python Language Reference* presents some introspection capabilities as implementation details of cPython, it

```
def trace(f):
  def wrapper(x):
    print(f.__name__, "called with argument", x)
    return f(x)
  return wrapper

@trace
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)

# The function 'fib' prints a trace before each call
fib(5)
```

Fig. 1.5. A Python decorator use to wrap a function

```
function_register = []

def register(f):
  function_register.append(f)
  return f

# The 'fact' function is added to the 'function_register' list
@register
def fact(n):
  result = 1
  for x in range(1, n + 1):
    result *= x
  return result
```

Fig. 1.6. A Python decorator used to register a function

also defines others as features in their own right. In this section, we cover such features with the goal of demonstrating some of the obstacles encountered when implementing a Python optimizing compiler. All code examples within this section are reproducible using the cPython interpreter.

1.2.1. Modules

Python implements modular programming through `module` objects. When Python executes a script, it creates an object of type `module`. All global assignments executed in the script are stored as attributes of the module. Conversely, all modifications applied to a module attributes are reflected on its global environment. Other Python programs can then import this module to access its functions and other attributes.

As such, there exist no notion of private or read-only module property. All modules can access and update all attributes from other modules. In particular, since Python loads modules dynamically one can never have the certainty that a global variable is defined within a module as a process external to the module may have updated it. This process of updating other modules global variables is called *monkey patching*.

```
# === my_module.py ======================================================

message = "I was defined in my_module.py"

def print_message():
    print(message)


# === monkey_patcher.py =================================================

import my_module

my_module.message = "I was monkey patched!"


# === main.py ===========================================================

import my_module
import monkey_patcher

my_module.print_message() # prints "I was monkey patched!"
```

Fig. 1.7. Monkey patching

In Figure 1.7, we show an example of monkey patching. Three modules are defined: `main.py` is the entry point of our program while `my_module.py` and `monkey_patcher.py` are external modules imported by our main script. The main script first imports `my_module.py` which Python executes to return a `module` object. When the module `monkey_patcher` is imported, its script makes an update to the `message` variable within the global scope of `my_module.py`. This change later shows through the call to the `print_message` function of `my_module` which now prints the monkey patched value from `monkey_patcher.py`.

Aside from assignments and modules attributes, there exist yet another way to update the global environment of a module. Python offers the possibility to reify the global scopes of its modules by invoking the `globals()` built-in function. This function returns a Python dictionary which allows to read and write the global environment. Since the returned object is a dictionary, any program can keep a reference to it and update it at any point of the execution.

In Figure 1.8, we show a Python module which uses the `globals()` function to assign values within its global environment. Furthermore, the global dictionary is bound to the

`environment` variable in that example. Doing so exports its global environment and allows other modules to manipulate it.

```
"""A Python module which reifies its global environment"""

environment = globals()

environment["var_name"] = "foo"
environment[var_name] = 42

print(foo) # prints 42
```

Fig. 1.8. A program using the `globals()` function

This peculiar behavior of Python environments makes static analysis of global variables impracticable. While a compiler can hypothesize about a variable's existence and value at a given point of the execution, the prospect of dynamically loaded code updating the environment remains.

1.2.2. Builtin functions

As mentionned in Section 1.1.1, references to a name which is neither found in the local scope, nor in the global scope are looked up in the builtin scope. The builtin scope is itself a module part of the Python standard library. One can import this module with the `import builtins` statement and update its attributes.

As explained in Section 1.2.1, updating a module's attributes is equivalent to update its global environment. The same holds for the `builtins` modules. Thus, any update to this module affects the builtins available throughout a Python program.

In Figure 1.9 we show an example where we use monkey patching to update the built-in `print` function. The script first imports the `builtins` module and updates its `print` attribute. When we later call the `print` function, we observe that Python calls our updated version of `print`.

In this last example, the same module updates the `builtins` module and then calls the `print` function. However, the result would be the same if an external, dynamically loaded, module updated the `builtins` module.

Thus, any appearance of a global reference to a built-in name could point to a different object than the original built-in function. Not only can dynamically executed code overshadow the name within the module's global scope, but the builtin scope itself can be updated.

1.2.3. Magic methods

Now that we studied how dynamically loaded code can affect modules' scopes, we shift our focus toward how it has similar impacts on various Python operators.

```
import builtins

# Keep a reference to the original 'print' to allow printing
old_print = builtins.print

def patched_print(message):
    old_print("'print' was monkey patched!")

builtins.print = patched_print

print(42) # prints "'print' was monkey patched!"
```

Fig. 1.9. Monkey patching of a built-in function

As a reminder from Section 1.1.2, the behavior of most Python operations depends on which magic methods the arguments' types implement. Yet, since magic methods are attributes on Python types, we can update them, as we would with any attribute, at any point during a program execution. This implies the behavior of an operator on a given set of types may change during the program execution. In Figure 1.10, we show a rather extreme example where we update the `__next__` method of an iterator during iteration. Thus, the `MyCounter` object which we initially defined to count from zero upward starts counting downward during its iteration. Moreover, this affects all objects of type `MyCounter` across our program.

```
class MyCounter:
    def __init__(self):
        self.next = 0

    def __iter__(self):
        return self

    def __next__(self):
        next = self.next
        self.next = next + 1
        return next

def __next__2(self):
    next = self.next
    self.next = next - 1
    return next

for x in MyCounter():
    print(x)
    if x == 3:
        # Updating the __next__ method of MyCounter changes its
        # behavior on subsequent iterations
        MyCounter.__next__ = __next__2
```

Fig. 1.10. Updating the `__next__` method during a loop

In general, we cannot assume that the same operation applied to the same object will return the same result. A change to any type within the object's MRO may result in a different magic method being called and thus a different result. Similarly, a change to the global scope in which any of the types in the MRO was defined may also change the behavior of a magic method and thus change the result of the operation.

An important exception is that of Python built-in types: attributes of all built-in types are read-only. For example, if one attempts to redefine the `__add__` magic method of the `int` type, the cPyton implementation raises a `TypeError`. The behavior is similar for other built-in types such as `bool`, `float`, `str` and so on. PyPy, another popular implementation of the language, exhibits a similar behavior [**24**]. Immutability of built-in types is part of Python's semantics.

This exception to the rule will play an important role in the upcoming chapters as it allows to bypass the search and invocation of magic methods when the operands' types are built-in ones.

1.3. Performance

Fundamentally, dynamic languages such as Python trade run time performance for developer productivity [**7, 25, 2**]. This choice appears to be paying off with Python standing as the second most used general-purpose language after JavaScript [**4, 5**]. Nonetheless, implementations such as PyPy (Section 1.3.2) demonstrated that one can obtain good performance [**24**]. In this section we cover the performance of both cPython and PyPy. We chose those implementations as reference points for the prominent place they occupy within the Python ecosystem.

1.3.1. cPython

The cPython interpreter is Python's standard implementation [**13**]. It performs poorly on many tasks even compared to other dynamic languages [**7, 26**]. Nonetheless, it is still widely used by developers.

The cPython interpreter is a stack-based bytecode interpreter. It compiles the user's source code to bytecodes which the cPython virtual machine executes. In a sense, cPython constitutes a straightforward implementation of the Python language with minimal optimizations. This permits profiling its interpreter to acquire insights about the overhead of various dynamic features [**7, 27**].

Among such features, boxing and unboxing of numerical values (`int` and `float` objects) accounts for a significant share by occupying up to a third of a program's execution time on some benchmarks. To a lesser extent, the same holds for boxing of containers (`list` and `tuple` objects). Dynamic type checking can account for up to 15% of a program's execution

time and late binding of method calls for up to 30% [**7, 27**]. Name resolution also has a significant impact on execution time [**27**].

When implementing an optimizing compiler, cPython thus acts as a guideline regarding potentially high-reward optimizations. Furthermore, as the reference implementation, it serves as a suitable baseline for performance comparisons.

1.3.2. PyPy

The PyPy project is a Just-In-Time (JIT) bytecode compiler often viewed as the state of the art among Python implementations [**24**]. It imposes itself as the main alternative to the standard implementation.

PyPy uses a tracing JIT compiler. Tracing JIT compilers rely on profiling to identify hot code paths within a program. After identifying such a path, the tracing JIT generates highly optimized machine code for its next executions. PyPy differs from other tracing JIT by applying tracing to the interpreter's code instead of the user's program [**28**]. This allows to unroll the bytecode dispatch loop itself to generate optimized machine code for the user's code hot paths.

On average a Python program executes 4.5 times faster when using PyPy over cPython. Although in some specific cases, performance improvements reach one or two orders of magnitude [**29**]. Furthermore, PyPy is among the most popular optimizing implementations of the Python language. This makes it an excellent reference point when implementing a Python optimizing compiler.

1.4. Formal and executable semantics

While the syntax of a language describes the textual form that a valid program can take, its semantics describes the expected behavior of such a program. The description of the behavior of the syntactic elements of a language can take various degrees of formality. This degree of formality refers to how precise and unambiguous the semantics of the language is. Various frameworks exist to provide formal semantics. Although these frameworks typically involve a mathematical description of a language's semantics, a formal semantics can take other forms, such as diagrammatic notation [**30**].

Mathematical frameworks such as *operational semantics* and *denotational semantics* [**31**] have been used to implement formal semantics of modern programming languages such as JavaScript [**32**], PHP [**33**], Python [**9**], and LUA [**34**] to name only a few.

Although the aforementioned techniques for describing semantics provide formal specifications, they do not provide an implementation of the language in general. They still require a programmer to implement the semantic rules within a compiler or an interpreter [**35**]. A

formal semantics from which we can automatically derive an implementation is called an *executable semantics* [**36**].

This thesis presents an executable semantics of the Python language which has some vague similarities with denotational semantics: it maps syntactic elements of the Python language to composable functions which describe their output and side effects. However, our formal semantics focusses on describing how those features should be computed, which denotational semantics typically do not [**31**].

1.5. Python executable semantics

As of today, Python does not have an official formal semantics. The reference manual uses prose instead of a formal specification which explicitly leaves room for ambiguities [**16**]. The absence of a formal semantics complicates the process of reasoning about Python. As a result, many third-party tools, such as static analyzers or interactive development environments are unsound [**9**].

Over the last decades, multiple attempts at defining a formal semantics for Python took place with increasing success [**37, 38, 9**]. Such formal semantics allow to use automatic theorem provers to demonstrate properties of the language [**37, 9**]. Executable formal semantics also permit feeding the semantics to specialized tools to yield an implementation of the language [**9**].

Despite recent progress, all existing formal semantics we encountered describe only a subset of the Python languages. The approach which allowed to define the widest subset relies on semantic *desugaring* [**9, 39**]. Desugaring consists in defining a core language with a small number of primitive constructs. In the context of Python, primitives represent low-level operations such as manipulation of an object's value or method lookup. A desugaring function then takes Python programs and expresses them into that core language [**9**]. Desugaring allowed to successfully define executable semantics of both (a subset of) Python [**9**] and JavaScript [**39**]. However, executable semantics yield implementations which are significantly slower than cPython [**9**].

To the best of our knowledge, there exist no executable semantics which yields an efficient implementation of the Python language. There remains to define such a formal semantics usable by optimizing compilers.

Chapter 2

A Python reusable semantics

In this chapter, our main goal will be to present a formal semantics aimed at developing optimizing Python compilers. We want such a formalization to:

- Automate part of the implementation of a Python compiler
- Be easily usable by multiple Python compilers
- Yield performant implementations

As seen in Section 1.1.2, Python's semantics highly depends on magic methods. Writing the numerous methods of Python's built-in types by hand is tedious and error-prone. We ought to automate this process to accelerate development. This automation process should also be reusable by other compilers. Integration to an existing compiler must be possible independently of the language and tools chosen for its implementation. We will achieve this by writing this semantics in a syntax similar to that of Python. Hence, interfacing with the semantics will only require updating the existing compiler's parsing infrastructure.

The semantics should yield a performant implementation. This requirement is trickier as it depends on the code the compiler generates from the semantics. Our approach will be to clearly identify part of the semantics which cause likely overhead in the implementation such as boxing and unboxing of primitive values and type checking [**7**]. In chapters 3 and 4, we will cover how an optimizing compiler can then reduce such overheads using the semantics.

2.1. The *compiler intrinsics* directive

To express the semantics, we extend Python with a statement which we call the *compiler intrinsics* statement. The syntax of this statement is the same as that of an `import` statement, except that the module name must be `__compiler_intrinsics__` which must be followed by a sequence of intrinsic functions or values. This is similar to Python's *future statement* which uses an import of the module `__future__` to enable features introduced in future releases [**40**]. Note that the parser does not need to be extended to include the *compiler*

intrinsics statement. The intrinsics imported with the compiler intrinsics statement resemble the primitive constructs used for semantics desugaring [**9**] presented in Section 1.5: they implement low-level building blocks used for describing Python semantics.

Figure 2.1 shows the compiler intrinsics statement used to define the addition semantics [**20**] which has been covered in Section 1.1.2. The statement prompts the compiler to make use of three intrinsics: `define_semantics`, `class_getattr` and `absent`.

```
from __compiler_intrinsics__ import define_semantics,
                                    class_getattr, absent

@define_semantics
def add(x, y):
    def normal():
        magic_method = class_getattr(x, "__add__")
        if magic_method is absent:
            return reflected()
        else:
            result = magic_method(x, y)
            if result is NotImplemented:
                return reflected()
            else:
                return result

    def reflected():
        magic_method = class_getattr(y, "__radd__")
        if magic_method is absent:
            return raise_binary_TypeError("+", x, y)
        else:
            result = magic_method(y, x)
            if result is NotImplemented:
                return raise_binary_TypeError("+", x, y)
            else:
                return result

    return normal()
```

Fig. 2.1. Reusable semantics of the `+` operator

The `define_semantics` intrinsic is intended to be used as a *function decorator*. It indicates to the compiler that a given function is not a Python function, but rather the definition of a semantics. Thus, the compiler does not need to allocate a Python `function` object and is free to store the semantics in its preferred format (such as a function in the target language). The targeted semantics is defined by the name of the function, in Figure 2.1 it is the `add` semantics. Labelling a function with `define_semantics` declares to the compiler that, with the code of the semantics, the compiler must assume that:

(1) Builtin names such as `int` and `isinstance` have their standard binding
(2) The builtin 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 some optimization such as replacing built-in name lookups by static references and 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 used as something else than a function decorator.

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 exists, it is returned. Otherwise, the value `absent` is returned to indicate that the attribute does not exist. Figure 2.2 shows a pseudocode implementation of `class_getattr`. In most cases, the result of a call to `class_getattr` is a magic method (see Section 2.4). 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(object, "name"):
  """Resolution of an attribute in an object's class mro"""

  class = type(object)  # The object's class

  for each parent in class mro: # as computed by the C3 linearization
    if parent has an attribute "name":
      return parent.name

  return absent  # the instrinsic value 'absent'
```

Fig. 2.2. Pseudocode of the `class_getattr` intrinsic

The `absent` intrinsic is a primitive value similar to the JavaScript `undefined`. 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.

Note that the `raise_binary_TypeError` function is not an intrinsic, but rather a semantics whose role is to raise a `TypeError` (see Figure A.1).

2.2. Semantics' nested functions

The semantics shown in Figure 2.1 is composed of the nested function definitions `normal` and `reflected`. 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. In particular, it is possible to apply *lambda-lifting* to prevent the creation of closures capturing the `x`

and y variables [**41**]. This process can be made to yield the functions add_normal and add_reflected which each corresponds to a specific point of execution in the semantics.

This allows an optimizing compiler to select which part of the semantics must be executed. If x + y is evaluated in a context where it is known that type(x) does not have a __add__ method or that this method will return NotImplemented, the compiler can short-circuit part of the semantics by directly invoking add_reflected(x, y).

2.3. Examples

We now demonstrate how the intrinsics presented in Section 2.1 are able and sufficient to define most Python semantics through some examples.

2.3.1. Arithmetic

We first provide examples for the semantics of arithmetic operations. These include the common operators: +, -, *, /, // (floor division), % (remainder) and ** (exponentiation). Python has the following bitwise operators: << (shift left), >> (shift right), & (bitwise and), | (bitwise or) and ^ (bitwise xor). Python also has the following comparison operators: <, >, <=, >=, == and !=. Finally, Python offers three unary operators: +, - and ~ (bitwise inverse). Binary operators support mixed types. For example 1 + 2 naturally returns 3, but 1 + 2.0 is also valid and returns 3.0. When an operator does not support a specific combination of types, it raises a Python exception.

The semantics of all binary operators are similar to that of the add semantics described in the previous section. For example, while the addition uses the __add__ and __radd__ magic methods, the semantics of multiplication requires __mul__ and __rmul__. It is thus possible to generate semantics of all binary operators from a unique template where only the names of the semantics, the magic method and the reflected method vary (see Figure A.2).

Unary operations such as the unary + and - operators or the bitwise inverse operator ~ can be defined in a similar way. These cases are in fact simpler as they do not possess a reflected magic method, having no second argument, and instead rely on the __pos__, __neg__ and __invert__ magic methods respectively. Figure 2.3 shows the semantics of pos.

Python implements the comparison operators in an identical fashion as binary operators. For example, the < operator uses the __lt__ magic methods and __gt__ as its reflected magic methods. Equality operators == and != somewhat stand out as they never raise an exception, but rather fall back to comparing objects by identity with the is operator if no suitable magic method is found (Figure 2.4).

Each Python operator also has an *inplace* variant. The += operator implements in-place addition, while *= implements in-place multiplication and so on. Inplace operators have their

```
@define_semantics
def pos(x):
    def normal():
        magic_method = class_getattr(x, "__pos__")
        if magic_method is absent:
            return raise_unary_TypeError("+", x)
        else:
            return magic_method(x)

    return normal()
```

Fig. 2.3. Reusable semantics of the unary `+` operator

```
@define_semantics
def eq(x, y):
    def normal():
        magic_method = class_getattr(x, "__eq__")
        if magic_method is absent:
            return reflected()
        else:
            result = magic_method(x, y)
            if result is NotImplemented:
                return reflected()
            else:
                return result

    def reflected():
        magic_method = class_getattr(y, "__eq__")
        if magic_method is absent:
            return default()
        else:
            result = magic_method(y, x)
            if result is NotImplemented:
                return default()
            else:
                return result

    def default():
        return x is y

    return normal()
```

Fig. 2.4. Reusable semantics of the `==` operator

own semantics, that is to say the statement `x = x + y` is not always semantically equivalent to `x += y`. The canonical example is that of lists for which `+` implements concatenation while `+=` extends a list, mutating the left operand instead of creating a new instance. We provide an example of this behavior in Figure 2.5.

Inplace operators rely on in-place magic methods such as `__iadd__`, `__isub__` and so on. For example, the semantics of `x += y` first determines the result of the inplace addition

by calling the `__iadd__` method of `type(x)`. If such a method does not exist or returns `NotImplemented`, Python calls the `__add__` method of `type(x)`. If that still fails, the `__radd__` method of `type(y)` is called as a last resort (see Figure 2.6). The result is then assigned to the left-hand side of the operator (see Section 2.3.3).

```
x = [1, 2]; y = x
y = y + [3]
# Result: x ← [1, 2]; y ← [1, 2, 3]

x = [1, 2]; y = x
y += [3]
# Result: x ← [1, 2, 3]; y ← [1, 2, 3]
```

Fig. 2.5. Example of usage of the `+=` operator on lists

2.3.2. Attribute access

The `getattribute` semantics (Figure 2.8) corresponds to the behavior of the syntactic form *obj*.*attr*. It allows accessing the attribute with name *attr* from the object *obj*. The attribute *attr* is treated as a string literal. Similarly to other semantics, `getattribute` relies on magic methods: `__getattribute__` and `__getattr__`. It is not possible for an object not to implement attribute access, as the `__getattribute__` is defined on the `object` type which all Python objects inherit from. However, an object can, of course, lack an attribute, in which case the `__getattribute__` method will raise an `AttributeError`. Python's semantics then falls back on the `__getattr__` method if it exists, otherwise the `AttributeError` bubbles up. In Figure 2.7, we define a custom class to demonstrate the usage of the `__getattr__` method.

Since the semantics requires handling a potential `AttributeError`, we express `getattribute` using a `try-except` block. If an `AttributeError` was raised and no `__getattr__` method is found, we use a `raise` statement to express that the expression should be reraised.

It is worth mentioning that the `class_getattr` intrinsic is independent from the `getattribute` semantics. The `__getattribute__` and `__getattr__` are bypassed when looking for a magic method as the result of implicit invocation via language syntax. Magic method must thus be set on the class to be invoked by the semantics, it cannot be returned by either the `__getattribute__` or `__getattr__` method [**42**]. This design choice exists to provide scope for speed optimizations [**43**]. It also solves the inherent circular dependency between `class_getattr` and `getattribute` which would arise otherwise.

```
@define_semantics
def iadd(x, y):
    def inplace():
        magic_method = class_getattr(x, "__iadd__")
        if magic_method is absent:
            return normal()
        else:
            result = magic_method(x, y)
            if result is NotImplemented:
                return normal()
            else:
                return result

    def normal():
        magic_method = class_getattr(x, "__add__")
        if magic_method is absent:
            return reflected()
        else:
            result = magic_method(x, y)
            if result is NotImplemented:
                return reflected()
            else:
                return result

    def reflected():
        magic_method = class_getattr(y, "__radd__")
        if magic_method is absent:
            return raise_binary_TypeError("+=", x, y)
        else:
            result = magic_method(y, x)
            if result is NotImplemented:
                return raise_binary_TypeError("+=", x, y)
            else:
                return result

    return inplace()
```

Fig. 2.6. Reusable semantics of the `+=` inplace operator

2.3.3. Assignment

A Python assignment can have four kinds of targets [**44**]:

(1) Identifiers: `x =` *expr*
(2) Attribute references: `x.`*attr* `=` *expr*
(3) Subscriptions: `x[`*item*`] =` *expr*
(4) Slices: `x[`*s*`] =` *expr*, where *s* is a `slice` object (*start:stop:step*)

Identifier assignment (`x =` *expr*) is the only kind of assignment whose semantics is not related to a magic method. It assigns a value to the identifier's name in the environment.

```
class MyObject:
    def __getattr__(self, attr):
        # If an attribute does not exist, we return None
        return None

obj = MyObject()
obj.x = 42

print(obj.x) # prints '42'
print(obj.y) # prints 'None'
```

Fig. 2.7. Example of usage of the `__getattr__` method

```
@define_semantics
def getattribute(o, attr):
    __getattribute__ = class_getattr(o, "__getattribute__") # cannot fail
    try:
        return __getattribute__(o, attr)
    except AttributeError as e:
        __getattr__ = class_getattr(o, "__getattr__")
        if __getattr__ is absent:
            raise e
        else:
            return __getattr__(o, attr)
```

Fig. 2.8. Reusable semantics of the attribute access *obj*.*attr*

Note that Python does not have variable declarations, when looked up a variable may be undefined, which raises an exception at run time.

Attribute reference assignment (`x.`*attr* `=` *expr*) is implemented by the `setattribute` semantics (Figure 2.9) which calls the `__setattr__` magic method with the target object, the attribute name as a Python string, and the assigned value. Similarly to the `__getattribute__` method, it is sound not to check whether `__setattr__` is absent as it is defined for the `object` type and thus all objects.

```
@define_semantics
def setattribute(o, attr, val):
    __setattr__ = class_getattr(o, "__setattr__") # cannot fail
    return __setattr__(o, attr, val)
```

Fig. 2.9. Reusable semantics of attribute assignment

Subscription assignment (`x[`*item*`] =` *expr*) is used to assign to a specific list or tuple index (in which cases the *item* is an integer index), and dictionaries (in which case the *item* is a hashable object, often a string). The semantics of subscription assignment (Figure 2.10) is similar to that of `setattribute`, except for the fact the magic method `__setitem__` may be absent.

```
@define_semantics
def setitem(o, index, value):
    __setitem__ = class_getattr(o, "__setitem__")
    if __setitem__ is absent:
        raise TypeError("object does not support item assignment")
    else:
        return __setitem__(o, index, value)
```

Fig. 2.10. Reusable semantics of subscription assignment

Slice assignment (`x[`*start:stop:step*`] =` *expr*) is in fact syntactic sugar. It relies on the same semantics as subscription assignments. The form `x[start:stop:step] =` *expr* is equivalent to invoking the `setitem` semantics with a `slice` object as second argument.

2.3.4. Truthiness and length

Finally, we provide an example where multiple semantics are at play: truthiness. An object truthiness has to be determined whenever it is used as the condition of an `if`-statement or `while`-statement, or if directly converted to a boolean using the `bool` builtin. Objects considered to be falsy include `False`, `None`, zeros of numeric types (e.g., `0` and `0.0`). A somewhat peculiar characteristic of Python is that empty sequences are also considered falsy (e.g., an empty list, tuple or dictionary).

This truthiness property of an object is determined by the `__bool__` magic method. Should this method not exist, Python will fall back to checking whether the object is an empty container. An empty container would be such that its length is `0`.

The length of an object is generally tested with the `len` builtin which relies on the `__len__` magic method. Since it depends on a magic method, we treat recovering the length of an object as a semantics in its own right. The Python language reference mentions that the `__len__` method should return a positive integer. However, the `cPython` implementation allows any numerical value which can be treated as an *index*. The notion of index refers to small positive integers. While *small* is implementation dependent, it should be understood as a 32-bit integer (on a 32-bit machine) or 64-bit integer (on a 64-bit machine).[1]

To express this segment of the semantics, we introduce two new intrisics which abstract the notion of *small* integers: `sint` and `bint`, which respectively stand for small and big integer. In the context of reusable semantics those are to be treated as subtypes of the `int` type which differentiate between small and big integers using the `isinstance` built-in function, while leaving room for implementation dependent details.

[1]The exact threshold for small integers usually depends on the target language used by a Python compiler. For example, cPython indeed uses 32-bit or 64-bit integers depending on the architecture. However, the compiler we will present in Chapter 4 does not compile directly to C, but rather to Scheme. The Scheme compilers we use actually implement small integers as 30-bit or 62-bit respectively.

This notion of conversion to a *small* integer is recurring in Python's semantics. Since there exists a magic method, `__index__`, which determines whether an object can be treated as an index, we also treat this as a semantics in its own right (see Figure 2.11).

```
@define_semantics
def index(obj):
    __index__ = class_getattr(obj, "__index__")
    if __index__ is absent:
        raise TypeError("'" + class_getattr(obj, "__name__")
                        + "' object cannot be interpreted as an integer")
    else:
        result = __index__(obj)

        if isinstance(result, sint):
            return result
        elif isinstance(result, bint):
            raise OverflowError("cannot fit into an index-sized integer")
        else:
            raise TypeError("__index__ returned non-int type: '"
                            + class_getattr(obj, "__name__") + "'")
```

Fig. 2.11. Reusable semantics of conversion to index

Our next concern is to avoid redundancy when expressing semantics. The `length` semantics raises a `TypeError` if the object has no `__len__` method in the context of the `len` builtin. Yet, in the context of the truthiness semantics, Python raises no exception if the method does not exist, but returns `True` which is the default if truthiness methods are not specified. We solve this issue by defining an intermediate semantics called `maybe_length` which never raises an exception, but allows expressing both the `length` and `truth` semantics (Figure 2.12).

2.4. Magic methods

The *reusable semantics* presented in the previous sections only partially describe Python's semantics. Since an object's type determines the operations supported for that object, we cannot fully describe Python's semantics without describing its built-in types, namely the magic methods they possess. For instance, while the `add` semantics from Figure 2.1 describes the general behavior of Python's `+` operator, it fails to predict the result of the expression `42 + 42.0`. In this section we introduce additional intrinsics methods which allow us to describe magic methods as well.

2.4.1. Operations on host values

All Python objects are defined by their *identity*, *type* and *value* (see Section 1.1.2). Applying an operation to an object requires extracting its value, computing a new value

```
@define_semantics
def maybe_length(obj):
    __len__ = class_getattr(obj, "__len__")
    if __len__ is absent:
        return absent
    else:
        len_result = __len__(obj)
        index_result = index(len_result)
        if index_result < 0:
            raise ValueError("__len__() should return >= 0")
        else:
            return index_result

@define_semantics
def length(obj):
    result = maybe_length(obj)
    if result is absent:
        raise TypeError("object of type"
                        + class_getattr(obj, "__name__") + "has no len()")
    else:
        return result

@define_semantics
def truth(obj):
    __bool__ = class_getattr(obj, "__bool__")
    if __bool__ is not absent:
        result = __bool__(obj)
        if type(result) is bool:
            return result
        else:
            raise TypeError("__bool__ should return bool, returned '"
                            + class_getattr(obj, "__name__") + "'")
    else:
        len_result = maybe_length(obj)
        if len_result is not absent:
            return len_result != 0
        else:
            return True
```

Fig. 2.12. Reusable semantics of length

from the former and encapsulating the result within a new object which is returned. This process of extracting and encapsulating a value is called boxing and unboxing among dynamic languages. An unboxed value is not a Python object. Its exact format depends on the target language used by a compiler, which we call the *host* language. We thus differentiate between Python objects and host values. The former are the objects within a Python program while the latter are the values encapsulated by objects.

To write down magic methods, we need to express this process of boxing and unboxing values. Therefore, we introduce a family of intrinsic functions which are named `py_`X`_from_host`

and `py_X_to_host`. While X could be any built-in Python type which encapsulates a value, we will limit ourselves to numerical types such as `int` and `float` for now.

The intrinsic function `py_X_from_host` takes a single argument which must be of type X and returns the host value of that argument. For example, the expression `py_int_from_host(42)` should return a value which represents the numerical value 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 `py_X_to_host` acts as the inverse of `py_X_from_host`. It takes a single argument which is a host value and encapsulates it in an object of type X. Thus, the expression `py_float_from_host(py_float_to_host(42.0))` should give back a Python object of type `float` with value 42.0.

In Figure 2.13 we show how these newly introduced intrinsic functions allow expressing the behavior of the `__add__` magic method of the `int` type. It only remains to define what we mean by `py_int_to_host(self) + py_int_to_host(other)` in which none of the operands of `+` are Python objects. The `+` operator does not implement the usual addition semantics in that context, but rather the host addition with respect to integral numbers. It would be possible to use another symbol than `+` to express that difference, although we will always write such magic methods in a context where Python dynamic features are known to be left aside. Thus, a compiler (or reader) can easily infer that the host version of an operator is to be used here.

```
def __add__(self, other):
    if isinstance(self, int):
        if isinstance(other, int):
            return py_int_from_host(py_int_to_host(self)
                                    + py_int_to_host(other))
        else:
            return NotImplemented
    else:
        raise TypeError("'__add__' requires a 'int' but received '"
                        + class_getattr(self, "__name__") + "'")
```

Fig. 2.13. Reusable semantics of the `int.__add__` method

An important aspect of the `py_int_to_host` intrinsic is that it will extract the value of an object even if its type is a subtype of `int` (such as `bool`). On the other hand, `py_int_from_host` will always return an object of type `int`. This conforms with the cPython behavior in which a built-in magic method must be overloaded for its return type to change. For example, given a user-defined type `MyInteger` which inherits from `int`, the addition of objects of type `MyInteger` would still return an object of type `int` unless the `__add__` method was overloaded to do otherwise.

Another concern one may have is what would happen if an object was both a subtype of `int` and `float`. It may then become ambiguous how a compiler would implement the `py_X_from_host` form. However, if we attempt to create such a subtype under the cPython implementation, a `TypeError` is raised with the message "multiple bases have instance layout conflict". It is thus reasonable to assume that such subtypes are forbidden by Python's semantics.

The `int.__add__` method from Figure 2.13 is sufficient to determine the result of the expression `42 + 42`. However, the method returns `NotImplemented` whenever the right operand is not an `int`, indicating it is the responsibility of the `float.__radd__` method to determine the result of the `42 + 42.0` (Figure 2.14). This method indeed has a branch for floating point addition and one for mixed-type addition. The former contains the expression `py_int_to_host(other) + py_float_to_host(self)`, it should be understood as a floating point addition where the left-hand side is converted to a floating point number, again this can be inferred statically by a compiler (or the reader).

```
def __radd__(self, other):
    if isinstance(self, float):
        if isinstance(other, float):
            return py_float_from_host(py_float_to_host(other)
                                      + py_float_to_host(self))
        elif isinstance(other, int):
            return py_float_from_host(py_int_to_host(other)
                                      + py_float_to_host(self))
        else:
            return NotImplemented
    else:
        raise TypeError("'__radd__' requires a 'float' but received a '"
                        + class_getattr(self, "__name__") + "'")
```

Fig. 2.14. Reusable semantics of the `float.__radd__` method

Now that we defined relevant magic methods for addition, we can determine the result of the expression `42 + 42.0`. The left operand being an integer, the `int.__add__` method is called with `42` and `42.0` and returns `NotImplemented`. The `float.__radd__` method is then called with `42.0` and `42` and returns the result of the expression `py_float_from_host(py_int_to_host(42) + py_float_to_host(42.0))` which is a `float` object with value 84.0.

2.4.2. Builtin classes

For the magic method defined in the previous section to be reusable by a compiler, we introduce the `builtin` intrinsic. This intrinsic decorator behaves similarly to `define_semantics`, but is used to indicate a built-in class to the compiler. Labelling a class with `builtin` assures

the compiler that the body of the class and the methods that it contains do not make use of Python's dynamic features. In Figure 2.15, we demonstrate how this can be used to define the built-in `int` type. Although we restrict ourselves to the addition-related magic methods of `int` in this example, all its magic methods can be represented in a similar fashion.

```
@builtin
class int:
    def __add__(self, other):
        if isinstance(self, int):
            if isinstance(other, int):
                return py_int_from_host(py_int_to_host(self)
                                        + py_int_to_host(other))
            else:
                return NotImplemented
        else:
            raise TypeError("'__add__' requires 'int' but received a '"
                            + class_getattr(self, "__name__") + "'")

    def __radd__(self, other):
        if isinstance(self, int):
            if isinstance(other, int):
                return py_int_from_host(py_int_to_host(other)
                                        + py_int_to_host(self))
            else:
                return NotImplemented
        else:
            raise TypeError("'__radd__' requires 'int' but received a '"
                            + class_getattr(self, "__name__") + "'")

    def __iadd__(self, other):
        if isinstance(self, int):
            if isinstance(other, int):
                return py_int_from_host(py_int_to_host(other)
                                        + py_int_to_host(self))
            else:
                return NotImplemented
        else:
            raise TypeError("'__iadd__' requires 'int' but received a '" \
                            + class_getattr(self, "__name__") + "'")
```

Fig. 2.15. Definition of the addition-related magic methods of the `int` type

Similarly to semantics, magic methods defining the behavior of arithmetic operators can be generated from templates. While semantics of arithmetic operators were generated from two templates (binary and unary operators), arithmetic magic methods can be broken down into five families, each requiring a different template. The basic template (Figure A.3) generates methods such as `__add__` and `__mul__`. The *division* template (Figure A.4) generates division and modulo methods such as `__floordiv__` and `__mod__`. The *shift* template (Figure A.5) generates bitwise-shift methods `__lshift__` and `__rshift__`. The

`comparison` template (Figure A.6) generates comparison magic methods such as `__eq__` and `__lt__`. The *unary* template (Figure A.7) generates the unary operator methods `__pos__`, `__neg__` and `__invert__`.

The requirement for multiple templates comes from the fact that most of the logic for Python operators appears in magic methods. For example, the semantics of the `/` division operator defers the check for division by zero to magic methods. Thus, magic methods generated by the *division* template will check for divisions by zero and raise a `ZeroDivisionError` accordingly.

Magic methods are also responsible for the return type of an operation. The *comparison* template provides an example of that; while other magic methods return integer or floating-point numbers, methods such as `__eq__` return a boolean object. To express this, we introduce yet another intrinsic function: `py_bool_from_host_bool`. This intrinsic converts a host boolean to a Python boolean (`True` or `False`). Thus an expression such as `py_bool_from_host_bool(py_int_to_host(x) == py_int_to_host(y))` would compare the host value of `x` and `y` and convert the result to a Python boolean.

Templates for the magic methods of the type `float` are similar to those of `int`, but admit a branch to treat mixed-type operations.

2.5. Summary

In this chapter, we provided a way to express part of Python's semantics in a syntax compatible with that of Python using the *compiler intrinsics* directive. We focussed on the semantics of arithmetic operators and described templates which allow us to generate the semantics of arithmetic operators as well as all magic methods required by those semantics of operations on `int` and `float` objects. All templates for semantics generation can be found in Appendix A. The technique described in this section to generate semantics and magic method can be extended in a fairly obvious way to other numeric types such as `bool` and `complex`. The ability to generate semantics which can be evaluated by the compiler alleviates the burden of implementing semantics and magic methods manually, which is both time consuming (altogether numeric types possess more than a hundred methods) and error-prone (magic methods all possess a similar structure). In the next chapters, we explore how reusable semantics can assist in the implementation of a Python optimizing compiler.

Chapter 3

Partial evaluation of semantics

A compiler can be made to evaluate the semantics as defined in the previous chapter to implement arithmetic operators. Yet, by doing so in a naive way, that is calling each magic method in order, the implementation would offer poor performance. For example, the evaluation of the expression `42 + 42.0` would require first calling `int.__add__(42, 42.0)` which returns `NotImplemented`, then calling `float.__add_(42.0, 42)` which finally returns the result `2.0`. Fundamentally, the result is just the output of the expression `py_float_from_host(py_int_to_host(42) + py_float_to_host(42.0))`. Furthermore, throughout the calls to the magic methods `__add__` and `__radd__` redundant type-checking takes place. An optimizing compiler should check the types of the left and right operands once and by doing so avoid redundant type-checking and skip the calls to the magic methods altogether by predicting the outcome of the type checks and method resolution by `class_getattr`.

In this chapter we present a technique for partial evaluation of our reusable semantics which generates procedures to quickly evaluate the result of an arithmetic operation for a given combination of built-in types. This technique focusses on removing redundant type checks, boxing and unboxing, and method calls whenever possible. Our goal is to then reuse those specialized procedures in an optimizing compiler. This will reduce the overhead caused by dynamic type-checking, boxing, unboxing and method calls which are known to slow down the cPython interpreter [**7, 27**].

3.1. Behaviors

In Section 1.2, we explained that a program can update most magic methods dynamically at run time. However, this is not possible for built-in types which are immutable. Thus, during the execution of arithmetic semantics (or any semantics relying on magic methods), calling methods of built-in types is superfluous as their behavior is known at compile time. We will exploit that fact to write down optimized versions of Python semantics.

We define *behaviors* as procedures which describe how to compute the result of Python semantics or operators *for a given combination of built-in types*. In particular, a behavior does not refer to magic methods. In Figure 3.1, we show the behavior for addition when the left operand is an `int` and the right operand is a `float` (how exactly behaviors can be generated will be discussed in Section 3.2). Behaviors are written in a similar fashion to semantics, using the `define_behavior` intrinsic decorator. That intrinsic behaves identically to `define_semantics` by preventing the use of Python's dynamic features. Thus `define_semantics` and `define_behavior` are semantically equivalent, but they label procedures differently. For example, in Figure 3.1, the `define_behavior` decorator indicates that `py_add_intX_floatY` is a specific case of the `add` semantics and not a general semantics for addition.

The nomenclature for behaviors goes as follows: `py_`*operation*`_`*ltype*`X_`*rtype*`Y` where *operation* is the short-circuited semantics, *ltype* is the required type of the left operand and *rtype* is the required type of the right operand. Note that we also include the left and right required types in the annotation of the behavior (the annotation involves the types written after each argument). This kind of annotation is part of Python's syntax [**45**]. We chose this redundant annotation as it is more convenient for a Python compiler to read required types from the annotation than from parsing the function name. Unary behaviors can be written by omitting the right operand, for example `py_neg_floatX`.

```
@define_behavior
def py_add_intX_floatY(x: int, y: float):
    return py_float_from_host(py_int_to_host(x) + py_float_to_host(y))
```

Fig. 3.1. Behavior of the `+` operator for an `int` and a `float`

We can use partial evaluation to generate all behaviors for arithmetic operations on numeric types. Given a semantics, we can identify which magic method returns a result. This is made possible by the fact a built-in magic method returns `NotImplemented` for a value of a given type if and only if it returns `NotImplemented` for all instances sharing the same type. In the event both magic methods return `NotImplemented`, the correct behavior is to raise a `TypeError` (Figure 3.2).

```
@define_behavior
def py_lshift_intX_floatY(x: int, y: float):
    raise TypeError("unsupported operand types for <<: 'int' and 'float'")
```

Fig. 3.2. Behavior of the `<<` operator for an `int` and a `float`

Within a given magic method, most `if`-statements' conditions are calls to the `isinstance` built-in function and can be resolved from the behavior's type context. The only exceptional

cases are those of division and bitwise-shift which respectively have to check for zero division and negative shift count. These conditions cannot be resolved at compile time, the behaviors are thus left to evaluate them at run time (Figure 3.3).

```
@define_behavior
def py_floordiv_floatX_floatY(x: float, y: float):
    if py_float_to_host(y) != py_float_to_host(0.0):
        return py_float_from_host(py_float_to_host(x)
                                  // py_float_to_host(y))
    else:
        raise ZeroDivisionError('float division by zero')

@define_behavior
def py_lshift_intX_intY(x: int, y: int):
    if py_int_to_host(y) >= py_int_to_host(0):
        return py_int_from_host(py_int_to_host(x) << py_int_to_host(y))
    else:
        raise ValueError('negative shift count')
```

Fig. 3.3. Behaviors requiring a run time conditional

This process alone suffices to generate all behaviors of binary and unary arithmetic operations for numeric types.

3.2. A partial evaluator to generate behaviors

We now present `semPy`, a Python implementation of the technique defined above for generating behaviors in a reusable syntax.

`semPy` is a Python partial evaluator written in Python and specialized for the syntax presented in Chapter 2. It takes as inputs a function written in that syntax and a *context* which consists of types for each of the arguments. It outputs a specialization of the function given that context (a behavior). To generate a specialization, `semPy` interprets the function's body by traversing its abstract syntax tree (AST). As the body is interpreted, `semPy` keeps track of the control flow in the given context to generate the new, optimized AST which is converted back to source code.

`semPy` manipulates the AST with the `ast` Python module. This module is part of Python's standard library and offers functions to convert source code to an AST, traverse and manipulate this AST, and convert it back to source code.

The functions provided to `semPy` are generated from the templates from Appendix A. For example, the behavior presented in Figure 3.1 was generated by `semPy` using the `add` semantics from Figure 2.1 and the magic method `__add__` from Figure 2.15.

In this section we present the implementation details of `semPy`. We first describe the `semPy` run time environment, then the main strategies used by `semPy` to generate specializations:

function inlining, branch resolution and removal of unnecessary boxing and unboxing. We conclude by presenting the `semPy` command line usage.

3.2.1. Runtime environment

The main challenge with the implementation of the `semPy` run time environment (RTE) was to allow the environment to contain partial data. While it is generally possible to infer the type of an object, its exact value if often unknown or only partially known. To address this, every object in the RTE is defined by three properties: its *type*, its *origin*, the *context* in which it was computed.

The type of an object is another `semPy` object which describes its class. This class models a Python class; it defines magic methods and possesses a MRO for resolving attributes. Thus, even though a value may be unknown, information about its type allows to recover the magic methods or to determine whether it is an instance of a given type with the `isinstance` built-in function.

The origin of an object is the expression which returned it. It takes the form of a node in the AST. The origin is accompanied by a context which is a snapshot of the RTE at the moment the value was computed. At any moment during the partial evaluation, `semPy` can use the origin to infer something about the value. For example, if the value originates from the length of a string, it is known to be positive. Keeping the origin of an object instead of the set of values it could take allows to gather information about a value lazily. Most of the time, `semPy` only requires information about an object's type (most expressions in the semantics are calls to `class_getattr` or `isinstance`) and pre-computing a set of possible values would be superfluous.

Additionally, an object's value can be conditional to the result of some expression, for example if it was computed within an if-statement whose condition could not be resolved to either `True` or `False`. In that case, we attach the condition to the object (a node in the AST) as well as the origin of the object given the condition is true and its origin given the condition is false.

Aside from partial values, the `semPy` RTE behaves similarly to that of Python. The environment mimics Python's scoping rules according to which variables are looked up sequentially in the local scope, enclosing scope (if a closure was created), global scope and built-in scope. The RTE supports the creation of closures, which are common in our reusable semantics (for example in the case of the `add` semantics from Figure 2.1). Many features are missing such as most built-in functions. Unsupported features will generally make `semPy` exit gracefully by raising a `NotImplementedError` exception. This is acceptable because the goal of `semPy` is to partially evaluate Python code whose structure we known and control.

3.2.2. Function inlining

When a function, semantics or magic method is invoked, `semPy` copies the callee's body at the call site. `semPy` then specializes the callee's body as if it were part of the caller's body. Variables are properly renamed in the callee's body to avoid name collisions.

Function inlining allows us to remove the magic method calls from semantics specializations. Magic methods appear in semantics as output of invocations of the `class_getattr` intrinsic function. Since this function is always called on the arguments of a semantics, whose type is provided to the specialisation, it is always possible to resolve which magic method is to be called (or if that method is absent). This is a consequence of the fact that builtin types have read-only attributes.

Most of the time, this process results in a single expression being inlined. Although in some cases, the magic method body cannot be specialised into a single expression, this is the case of division semantics where the partial evaluation process lacks information about the value of the dividend to remove tests for zero division. Due to Python's syntax this sometimes leads `semPy` to generate an AST which cannot be converted back to source code. For example if a call is resolved to a sequence of statement, inlining these statements within an assignment would yield an invalid AST. `semPy` solves this issue by inlining such statements nonetheless. A post-processing phase later resolves such issues by transforming the AST to an equivalent, but valid format.

3.2.3. Branch resolution

Branch resolution occurs when `semPy` succeeds in computing the condition of an `if`-statement, or at least its truthiness. In this case, we can get rid of the branch which is not executed.

`semPy` uses the fact the evaluated function is a semantics written without using Python dynamic features. This allows resolving the value or type of expressions which would normally be hard to evaluate statically. In particular, it allows resolving conditions of the form `isinstance(X, Y)` which determines whether `X` is an object of type `Y` when the type of `X` is known and `Y` is the name of a built-in type. In most cases, conditions can thus be resolved to `True` or `False`. Alternatively, the form `type(X)` which returns the type of `X` can also be resolved. This is useful when one needs to test whether an object has an exact type with the form `type(X) is Y`.

Comparisons can sometimes be resolved. The most frequent case is that of the `is` operator which tests whether two values have the same identity. Within the semantics, the `is` operator generally compares a value to the singleton `NotImplemented` or to the intrinsic value `absent`. These comparisons can always be resolved as they depend on the type of the arguments alone, never on their values.

Resolving the aforementioned conditions alone is sufficient to remove most `if`-statements. The exception being conditions depending on the value of the semantics' arguments such as those in the division and bitwise-shift magic methods (Figure 3.3). There exists some specific cases where these can be resolved, for example when the origin of a value provides us information about its sign.

3.2.4. Unnecessary boxing and unboxing

There exists cases where the naive expression of Python's semantics causes unnecessary boxing and unboxing of values. 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__` as we wrote it applies boxing and unboxing to account for the possibility the argument is of a strict subtype of `int` (Figure 3.4) in which case the result should be cast to an `int`. The simplest example is that of the expression `+True` which should return `1`.

```
def __pos__(self):
    if isinstance(self, int):
        return py_int_from_host(py_int_to_host(self))
    else:
        raise TypeError("'__pos__' requires a 'int' but received a '" \
                        + class_getattr(self, "__name__") + "'")
```

Fig. 3.4. The `int.__pos__` magic method

In the context of behaviors where we know the argument to be of type `int`, this type conversion is unnecessary and `semPy` removes it. When type conversion must occur, for example in the case of unary `+` on a `bool`, then `semPy` knows to preserve it (Figure 3.5).

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

@define_behavior
def py_pos_boolX(x: bool):
    return py_int_from_host(py_int_to_host(x))
```

Fig. 3.5. Removal of unboxing in the behavior of unary `+` of `int` by `semPy`

This simplification occurs after inlining and branch resolution where unnecessary boxing becomes apparent when chains of procedures which are one another inverses appear in behaviors. We provided an example with `py_int_from_host` and `py_int_to_host`, the same happens with `py_float_from_host` and `py_float_to_host`.

3.2.5. The `test` semantics optimisation

In this section we present another example where `semPy` allows removing unnecessary boxing. Consider the semantics of the Python `if`-statement where Python evaluates the truthiness of a value and branches accordingly. This truthiness is determined by the `truth` semantics (Figure 2.12) which returns either `True` or `False`. This determines which branch should be executed. `semPy` can correctly generate behaviors for the `truth` semantics. We show the behavior for truthiness of an integer in Figure 3.6.

```
@define_behavior
def py_truth_intX(obj: int):
    return py_bool_from_host_bool(py_int_to_host(0) != py_int_to_host(obj))
```

Fig. 3.6. Behavior for truthiness of `int`

In the context of an `if`-statement, this behavior will take the boolean host result `py_int_to_host(0) != py_int_to_host(obj)` and convert it to a Python `bool` which the `if`-statement will immediately need to compare against `True` to obtain a host boolean. Thus, the call to the intrinsic `py_bool_from_host_bool` is a form of unnecessary boxing. In Figure 3.7, we show two examples where conditions are evaluated but conversions to a `bool` are unnecessary. We also show a condition which must be converted to a `bool` as it can be referenced later in the program.

```
# The expression X does not need to be converted to a Python bool
if X:
    print("X is truthy")
else:
    print("X is falsy")


# The same if true when evaluating the condition of a while-loop
while Y:
    print("looping because Y is truthy")


# Here the result of "x == y" needs to be converted to a bool because
# it is bound to the global name "condition" and can be referenced
condition = x == y
if condition:
    print("x equals y")
else:
    print("x does not equal y")
```

Fig. 3.7. Examples of unnecessary boxing of the result of a condition

We solve this issue by introducing a new semantics which we call `test` (Figure 3.8) and a new intrinsic `py_bool_to_host_bool` which converts Python `bool` to a boolean in

the host language. In particular this intrinsic is known to `semPy` as being the inverse of `py_bool_from_host_bool`. The purpose of that 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. In general, we can achieve this by invoking the `truth` semantics and converting the result to a host boolean.

```
@define_semantics
def test(obj):
    return py_bool_to_host_bool(truth(obj))
```

Fig. 3.8. The `test` semantics

This semantics can be fed to `semPy` for generating behaviors. It will return behaviors for `test` which is exempt of the unnecessary boxing as shown in 3.9.

```
@define_behavior
def py_test_intX(obj: int):
    return py_sint_to_host(0) != py_int_to_host(obj)
```

Fig. 3.9. Behavior `test` for `int`

This strategy is expendable 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 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`). The naive approach without behaviors offers a convoluted sequence of invocations where a comparison semantics is invoked followed by the `truth` semantics, causing numerous boxing and unboxing. `semPy` can instead generate behaviors for those specific cases by applying partial evaluation to a chain of semantics invocations. To those behaviors we assign the names `py_test_`*comp*`_`*ltype*`X_`*rtype*`Y` where *comp* is the comparison semantics being partially evaluated.

In Figure 3.10, we show the result of `semPy` partial evaluation for the invocation chain `test(le(x, y))` where `x` and `y` are respectively an `int` and a `float`.

```
@define_behavior
def py_test_le_intX_floatY(x: int, y: float):
    return py_float_to_host(y) >= py_int_to_host(x)
```

Fig. 3.10. The `test_le` behavior for `int` and `float`

3.2.6. Command line usage

`semPy` requires Python 3.9 or higher and has no external dependencies. It can be downloaded from `https://github.com/omelancon/semPy` and used with no further configuration.

`semPy` is a Python standalone module which can be invoked with the `python3 -m sempy` command. As command line arguments, it minimally requires the paths to the `semantics.py` and `builtins.py` files written in the syntax from Chapter 2. Additional modules containing functions to be partially evaluated can be provided with the `-modules` option. Here is the command to execute `semPy` on the `semantics.py` and `builtins.py` modules as well as a custom `math.py` module.

```
python3 -m semPy path/semantics.py path/builtins.py -modules path/math.py
```

This command alone is not sufficient to generate behaviors, as it contains no information regarding which function or semantics should be specialized. For each module, a file with the extension `.sempy` must be present in the same directory to specify which specializations are required. A `.sempy` file is a Python script which has access to the function `add_specializations`. Such a file is executed by `semPy` with the `exec` built-in function.

The `add_specializations` function takes a function name and a list of types as arguments. For example, suppose one was to provide the following `semantics.sempy` file within the same directory as the `semantics.py` module.

```
add_specializations("add", types=[("int", "bool"), "int"])
add_specializations("iadd", types=["float", "float"])
add_specializations("pos", types=["int"])
```

`semPy` would then output the following behaviors: `py_add_intX_intY`, `py_add_boolX_intY`, `py_iadd_floatX_floatY` and `py_pos_intX`.

The `.sempy` used to generate all behaviors presented in this thesis can be found at `https://github.com/omelancon/semPy/blob/0e39fb6419/out/semantics.sempy`.

3.3. Summary

In the chapter, we defined *behaviors* which are functions specialized for computing semantics for a given type or combination of types. We can compute behaviors for built-in types statically as the magic methods of built-in types are immutable.

We described a technique to generate behaviors by partial evaluation of the semantics presented in Chapter 2. This technique succesfully removes redundant type checks, boxing and unboxing from the semantics. It also entirely removes calls to magic methods. Thus,

behaviors will reduce the overhead of such redundant computations [**7, 27**] when implementing an optimizing compiler.

We finally presented `semPy`, an implementation of such a partial evaluator. Writing behaviors by hand would be tedious and error-prone due to the large number of Python semantics and built-in types. Thus, `semPy` automates the generation of behaviors from our reusable semantics.

Chapter 4

The Zipi compiler

In this chapter we present the Zipi compiler, an ahead-of-time (AOT) compiler from Python to Scheme written in Scheme and Python which implements arithmetic operations using behaviors generated with `semPy` and extends this strategy to some other semantics. Our goal is to show that the tools described in previous chapters are well suited for the implementation of an optimizing compiler for Python.

In the following sections we give an overview of the Scheme language as well as the architecture of the Zipi compiler. We describe how `semPy`-generated behaviors are used to implement the semantics of the Python language. We conclude by discussing the performance of the Zipi compiler by presenting some benchmarks.

4.1. The Scheme language

The Scheme language [**46**] is a dialect of Lisp whose standards are defined by *Revised n^{th} report on the Algorithmic Language* or RnRS. We focus on the Bigloo [**47**] and Gambit [**48**] optimizing Scheme compilers the Zipi compiler relies on. Both of these implementations mostly follow the R5RS standard [**49**].

Scheme, like Python, is a dynamically typed language. However, both Bigloo and Gambit have strategies to mitigate the performance cost of run time type-checking. Bigloo can use type information provided at compile time by the user to partially apply type checking at compile time. Directives can be given to the Gambit compiler to skip type-checking altogether on certain operators. While this prevents gracefully exiting in case of a type error, it significantly improves the performance of a sound program.

All implementations of Scheme are properly tail-recursive. A procedure call which takes place at the tail of a procedure's body will not require to allocate a new stack frame due to its return site being the same as that of the active procedure. Thus, no space is required for tail calls, allowing for an unbounded number of tail calls. As a consequence, Scheme has no `while`-loop or `for`-loop statements. Loops are instead implemented by recursive tail calls. In

Figure 4.1, we show an implementation of the `factorial` function relying on such recursion. In that case, not only is proper tail-call ensured by the language standard, but compilers such as Bigloo and Gambit will implement the tail call to `loop` by using a `goto` instead of allocating a new stack frame. As a side effect of Scheme's proper tail recursion, the Zipi compiler inherits this property from its host language, making for a properly tail-recursive Python implementation.

```
(define (factorial n accumulator)
  (if (= n 0)
    accumulator
    (factorial (- n 1) (* accumulator n))))

(println (factorial 42 1))
```

Fig. 4.1. A Scheme program computing the factorial of 42

As a dialect of Lisp, Scheme employs the parenthesized prefix notation (e.g., Figure 4.1). Likewise, a Scheme program represents data using the parenthesized notation. For example, while `(+ 1 2)` expresses the addition of two integers, the quoted expression `'(+ 1 2)` represents a list composed of the symbol `+` and two integers. This uniformity between programs and data representation allows for Scheme programs to seamlessly manipulate Scheme expressions as if they were data. This makes for straightforward metaprogramming through the use of macros which can manipulate Scheme expressions to implement new syntactical forms. In figure 4.2 we used the `define-macro` syntax to define a new syntactical form which iterates over a list using a syntax similar to that of Python's `for` statement. As we will see, the Zipi compiler uses macros extensively to inline Python operations.

```
(define-macro (for x lst . body)
  `(let loop ((lst ,lst))
     (if (pair? lst)
         (let ((,x (car lst)))
           ,@body
           (loop (cdr lst))))))

(for x '(1 2 3)
  (println x))
```

Fig. 4.2. A macro defining a `for`-loop syntax similar to that of Python

4.2. Zipi architecture

The Zipi compiler includes two main modules: the `pyc` compiler which compiles Python code to Scheme and the Zipi runtime system which implements Python's data model and

operators. The generated code is then compiled to an executable using either the Bigloo or Gambit compiler.

4.2.1. The pyc compiler

pyc compiles Python source code to a Scheme module. The compilation to Scheme is mostly straightforward as the main optimizations are applied at the level of the runtime system.

In Figure 4.3, we present a snippet of the code generated by pyc on a small Python program. The compiler maps most operations directly to a procedure or macro provided by the runtime system. For example, the forms py-for-each, py-make-list and py-iadd are all macros whose expansions implement the semantics of the Python for-loop, list allocation and in-place addition, respectively. In all examples we present in this chapter, only relevant parts of the generated code are shown and variable names have been demangled for readability.

```
# A Python program which sums a list of integers
s = 0

for x in [1, 2, 3, 4]:
    s += x

# ============================================================================

;; The main body of the code generated by pyc from the above program
(define x #f)
(define s #f)

(global-register! global (& "x") (lambda () x) (lambda (v) (set! x v)))
(global-register! global (& "s") (lambda () sum) (lambda (v)
                                                   (set! s v)))
(set! s (py-int-from-scheme 0))
(py-for-each
  target
  (py-make-list
    (py-int-from-scheme 1)
    (py-int-from-scheme 2)
    (py-int-from-scheme 3)
    (py-int-from-scheme 4))
  (begin
    (set! x target)
    (set! s
      (py-iadd (or s
                   (global-get global (& "s") (vector-ref caches 2)))
               (or x
                   (global-get global (& "x") (vector-ref caches 3)))))))
```

Fig. 4.3. An example of Scheme code generated by pyc

`pyc` uses Scheme variables to implement Python's variables. Thus, a Python assignment to a variable is mapped directly to a Scheme `set!` expression. This turns out to be especially efficient in the case of global variables for which the Bigloo and Gambit compilers allocate static memory. However, the Python global environment is both accessible through the `globals()` function as a dictionary and as a `module` whenever imported by another module. Thus, implementing global variables with Scheme global variables only works when those features are not used and requires a fallback otherwise. For this reason, the `global-register!` macro used at the beginning of the compiled code ensures global variables can be accessed by name through a closure in those cases.

The `pyc` compiler also supports the `__compiler_intrinsics__` directive from Chapter 2. This allows reuse of the semantics generated from templates in Appendix A as well as the behaviors generated by `semPy` to generate efficient Scheme code. In Figure 4.4 we show the Scheme version of the `add` semantics from Figure 2.1. Note that `pyc` compiles the semantics to the `py-add-fallback` macro. This name reveals that the full semantics is used only as a last resort, that is when no specialized behavior exists for the combination of operands' types (more on this in Section 4.3).

```
(define-macro (py-add-fallback x y)
  `(let ((x ,x) (y ,y)) (py-add-fallback:normal x y)))

(define (py-add-fallback:normal x y)
 (let ((magic_method (getattribute-from-obj-mro x (&& "__add__"))))
   (if (py-test (py-is magic_method py-absent))
       (py-add-fallback:reflected x y)
       (let ((result (py-call magic_method x y)))
         (if (py-test (py-is result py-NotImplemented))
             (py-add-fallback:reflected x y)
             result)))))

(define (py-add-fallback:reflected x y)
 (let ((magic_method (getattribute-from-obj-mro y (&& "__radd__"))))
   (if (py-test (py-is magic_method py-absent))
       (py-raise-binary-TypeError-fallback (&& "+") x y)
       (let ((result (py-call magic_method y x)))
         (if (py-test (py-is result py-NotImplemented))
             (py-raise-binary-TypeError-fallback (&& "+") x y)
             result)))))
```

Fig. 4.4. Scheme version of the `add` semantics as generated by `pyc`

Similarly, behaviors such as those introduced in Chapter 3 can be compiled by `pyc`. When using `semPy` for generating behaviors for the Zipi compiler, we distinguish between small integers (`sint`) and big integers (`bint`). This both allows `semPy` to generate slightly more specialized behaviors and the Zipi runtime system to further optimize small integer arithmetic (see Section 4.2.2). In Figure 4.3, we show the `add` behaviors for small integers as compiled by

pyc. The py-int-from-scheme and py-sint-to-scheme are the Scheme equivalent of the py_int_from_host and py_int_to_host intrinsics, respectively. The py+ is a special form recognized by the py-int-to-scheme macro and translated to the correct Scheme operator, in that case the fx+? small integer addition with overflow check operator which both Bigloo and Gambit compile to a single machine addition instruction followed by a check of the overflow flag. In case of overflow, fx+? returns #f (Scheme's false value) and Zipi falls back on the pyadd-UintX-UintY-fallback procedure which executes big integer addition out of line.

```
# Python add behavior for small integers
@define_behavior
def py_add_sintX_sintY(x: sint, y: sint):
    return py_int_from_host(py_int_to_host(x) + py_int_to_host(y))

# ================================================================================

;; add behavior compiled by pyc
(define-macro (py-add-sintX-sintY-inline x y)
  `(let ((x ,x) (y ,y))
     (py-int-from-scheme
      (py+ (py-sint-to-scheme x) (py-sint-to-scheme y)))))

(define (py-add-sintX-sintY-fallback x y)
  (py-int-from-scheme (py+ (py-sint-to-scheme x) (py-sint-to-scheme y))))

# ================================================================================

;; full expansion of the expression
;; (py-int-from-scheme (py+ (py-sint-to-scheme 1) (py-sint-to-scheme 2)))
(let ((X 1))
  (let ((Y 2))
    (or (fx+? X Y) (pyadd-UintX-UintY-fallback X Y))))
```

Fig. 4.5. The py_add_sintX_sintY behavior as compiled to Scheme by pyc

4.2.2. The Zipi runtime system

Executing Scheme code generated by pyc requires importing the Zipi runtime system. The runtime system defines all procedures and macros implementing Python semantics, operators and data model. It both includes code generated by pyc from the semantics, behaviors and the builtins Python modules (itself generated from the magic methods templates from Appendix A), and hand-written code implementing the rest of Python's data model.

The builtin types int, float, str, range, classmethod and staticmethod as well as the BaseException class and all its subtypes are written in Python using the __compiler_intrinsics__ directive and compiled to Scheme by pyc. The same is true for

the `sorted` built-in function as well as the modules `itertools`, `math`, `random`, `sys` and `time` which are partially written in Python. Building the runtime system thus requires compiling the aforementioned modules, semantics and behaviors. Those are merged with the rest of the runtime system which is handwritten in Scheme and defines the rest of the data model.

Objects are represented by records which contain their *type* (a `type` Python object), *value*, *attributes* (a vector) and a *hidden class*. A hidden class is an object which contains information about the object's layout [**50**]. Objects which have the same type and are created in the same way share the same hidden class. Among other things, the hidden class contains the name of each attribute in an object's attribute vector. Thus while the first access to an attribute requires a linear search through the hidden class, we can use inline caches to read or write the attribute on a subsequent access [**51**]. On the first access, the attribute is recovered by a linear search of the hidden class. If an attribute is found, the hidden class is stored in a cache along with the index of the attribute. On subsequent accesses, we can compare the object's hidden class to the cached hidden class: whenever they are the same, we can skip the hidden class linear search and use the stored index. The intent behind the usage of hidden classes is to speed up attribute access. Although the language allows polymorphism, Python programs are predominantly monomorphic [**52**]. Hence, we expect a specific attribute access within the code to mostly apply to objects of the same type and thus have the same layout. This in turn allows the application of inline caches for attribute access.

Some objects also contain extra fields. For example, `function` objects have fields for their *arity*, *signature* and *code*. However, we can conceptually treat those extra fields as part of the value of the object despite requiring multiple slots in the record.

An exception to that object representation is the implementation of small integers (objects of type `int` with a small integer value) and floating point numbers (objects of type `float`). These are represented as unboxed Scheme `fixnum` and `flonum` objects. As an example, the `py-sint-to-scheme` macro from Figure 4.4 is simply the identity macro as no unboxing is necessary. The behavior for small integer addition corresponds to a Scheme small integer addition which result is boxed only if the result overflows the small integer range (see Figure 4.5). This results in significantly faster arithmetic operations on small integers and floating point numbers. However, whenever the runtime system attempts to recover the type or hidden class of an object, it must first check that the object is not a Scheme `fixnum` or `flonum`. As we will see in Section 4.5, this tradeoff almost always offers a net gain in performance as it entirely removes the need for boxing and unboxing values when executing arithmetic operations.

4.3. Behaviors in Zipi

We have described how `semPy` generates behaviors and how `pyc` compiles those behaviors and includes them into the runtime system. We now explain how Zipi dispatches an operation to a specific behavior at runtime. The general idea is to store each behavior procedure within a vector at a known index. Once the type of each operand is known, it is then possible to recover the corresponding behavior from that vector and invoke it.

4.3.1. Behavior dispatch

To properly recover behaviors from the *behavior vector*, we assign a unique identifier to each type. We call this identifier the *class index* of a type. These class indices take values starting from `1` up to the number of types which possess a behavior. Since we generated separate behaviors for small integers and big integers, we treat those as having 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 to be the class index of all types which have no specialized behavior, for example user-defined types.

When a Python arithmetic operator is applied, we recover the class index of the types of each operand. In the case of unary operators, the recovered index is the position of the corresponding behavior in the *behavior vector* for that operator. In the case of binary behaviors we recover the index by applying 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 recovered at that index can be safely called by Zipi with both operands without further type-checking.

In some cases, the procedure stored at the computed index will not be a compiled behavior. This happens whenever either of the operands' class indices is `0`. The recovered index then contains a procedure calling the full semantics with no specialization. For example, if we add two objects whose type is user-defined, the resulting index will be `0`. The `add` behavior vector contains the `py-add-fallback:normal` procedure (see Figure 4.4) at that index.

An exception to the dispatch of a behavior is the case where both operands are either small integers or `float` objects. As mentioned in section 4.2.2, these objects are represented by Scheme `fixnum` and `flonum` values respectively. Whenever recovering a class index, we must check that operands are proper Python objects first. We use this to our advantage by inlining the corresponding semantics in the case where operands are `fixnum` or `flonum` objects.

Figure 4.5 shows that `pyc` 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 a behavior table.

In Figure 4.6 we demonstrate the process described above by showing an uncluttered version of the `py-pos` macro, which implements the unary `+` operation. The original version of `py-pos` found in the Zipi runtime system contains code related to debugging, we removed such code for the purpose of readability. Whenever the operand `x` is either a `fixnum` or `flonum`, we execute the inline behaviors `py-pos-sintX-inline` and `py-pos-floatX-inline` respectively. Otherwise, we use `py-obj-class-index-general` which is a macro specialized for recovering the class index of an object which is known not to be a `fixnum` or a `flonum`. Finally, we use the recovered index to extract and invoke the corresponding behavior from the behavior vector `py-pos-behaviors-table`.

```
(define-macro (py-pos x)
  `(let ((x ,x))
     (cond
       ((fixnum? x) (py-pos-sintX-inline x))
       ((flonum? x) (py-pox-floatX-inline x))
       (else
         (let ((class-index (py-obj-class-index-general x)))
           ((vector-ref py-pos-behaviors-table class-index) x))))))
```

Fig. 4.6. Dispatch of the behavior for unary `+`

The process of dispatching behaviors presented in Figure 4.6 is similar in the case of binary arithmetic. Behaviors are inlined when both operands are either `fixnums` or `flonums`, otherwise the behavior is recovered from the corresponding behavior vector. The same happens for comparisons operators and the `truth` and `test` semantics.

4.3.2. Behaviors code size

The behaviors presented until now were all generated by `semPy` and compiled to Scheme using `pyc`. With 38 binary semantics (including in-place and comparison semantics), this generates over six hundred behaviors for the numerical types alone (`sint`, `bint`, `bool` and `float`). This number grows when including behaviors for `str` (`+`, `*` and `%` are valid operators on `str` objects) and unary semantics. `semPy` thus prevents the time-expensive and error-prone process of specializing semantics for each behavior by hand and ensures a single source of truth between behaviors and magic methods. Nonetheless, this raises a concern regarding the size of the generated code since the number of behaviors grows with the square of the number of specialized types (in the case of binary semantics).

To alleviate this issue, we can use different set of optimized types for different semantics. For example, for the family of arithmetic semantics Zipi uses behaviors for all combinations of two types from `sint`, `bint`, `bool`, `float` and `str`. If one wishes to generate behaviors for the subscript operator (`obj[item]`) denoting the `getitem` semantics, they should choose different combinations of types. This choice depends on usage across Python programs. Since

numbers are not subscriptable, we should not expect one to be the left argument of the `getitem` semantics. Thus, a sound choice of types for `getitem` would be to generate all behaviors whose left operand's type is a container (`tuple`, `list`, `str` and `dict` for example) and right operand is a valid subscript (`sint`, `bool` and `str`). This would lead to generate behaviors such as `py-getitem-listX-sintY`, but not `py-getitem-sintX-listY`. Whenever a specialization is not generated, the corresponding slot in the behavior vector can be filled with the procedure executing the full semantics.

The proposed solution still generates behavior vectors growing quadratically with the number of specialized types for a given operator. While this has not yet caused problems in Zipi, we will likely have to explore alternatives in the future. The usage of sparse tables is considered, but would require an analysis of the impact it would have on performance.

4.4. Other optimizations

Aside from behaviors generated by `semPy`, Zipi implements some other optimization techniques which we mention here. These techniques generally involve a clever Scheme macro which attempts to guess the type of its arguments at compile time and inlines efficient code for the most likely case. The list of optimizations presented is not exhaustive, it aims at proving that behaviors can cohabit with other optimization techniques.

4.4.1. `for`-loops

Python generally requires calling the `__iter__` method of the iterable before the loop to obtain an iterator, then the `__next__` method on every iteration. As one may guess by now, these method calls have a high cost and are superfluous when the iterable is a built-in type. However, in the case of `for`-loops, we cannot check the type of the iterable and branch to an optimized loop as this would duplicate the loop's body. The solution we found involves representing iterators with Scheme values. Indeed, the iterator of a loop is unattainable by the programmer, it is thus sound for it not to be a Python object. For example, when iterating over a `range`, the internal iterator is not a `range_iterator` object but rather a Scheme `pair` with an integer as the first element. At every iteration, Zipi checks the type of the iterator and computes the next element in the loop accordingly. This proved significantly more efficient than calling the `__next__` method at every iteration.

4.4.2. Function calls

Zipi's `function` objects are records containing a Scheme procedure (the function's code) and a signature. In general, every call requires comparing the provided argument to the function's signature. This process has a high cost, but a necessary one for when functions have default arguments. However, functions often have a fixed arity. We thus store the arity

of each function in its record if it is indeed fixed. Whenever a call contains only positional arguments, this allows Zipi to directly compare the number of arguments to the function's arity. Whenever both match, the function's procedure is extracted and called inline without having to parse the function's signature.

4.4.3. Scheme compiler optimizations

Aside from high-level optimizations applied by `pyc` and Zipi's runtime system, we build and compile Zipi with Gambit or Bigloo which are Scheme optimizing compilers. Many optimizations thus take place at a lower-level. Hence, performance is highly dependent on the choice of the underlying Scheme compiler.

4.5. Performance

In this section, we present Zipi's performance in comparison to cPython, the standard implementation of the language, and PyPy, the current state of the art implementation performance-wise.

We measured performance by the usual approach of executing benchmarks. Benchmarks are separated into two categories: microbenchmarks aimed at measuring the performance of a specific operation and regular benchmarks measuring the performance on implementations of well-known algorithms.

Due to being an AOT compiler, Zipi has a significant overhead caused by compilation of the Python scripts. Zipi also has a slower initialization speed when loading its runtime system. Our benchmarks are written in such a way that the timer starts after compilation and initialization, as to measure only the runtime performance of the program. Similarly, it is recommended to allow PyPy's JIT to warm up when executing benchmarks [**53**]. We do so by executing a dry run of benchmarks which does not count toward execution time. Execution time is measured using the Python `time` module which all implementations provide. Benchmarks output their own execution time after execution.

All results presented here were generated with the tool Forensics (available at `https://zipi-forensics.gambitscheme.org`). Forensics is a tool used to follow performance change of different implementations of a language across versions. It was initially developed for measuring performance of the Gambit Scheme compiler and adapted for Python during the development of Zipi.

For benchmarking, we used cPython version 3.9.0. As for PyPy, we used the version 7.3.5. Each PyPy release implements more than one version of Python, we used the newest version available which is Python 3.7. Finally, the results we present were obtained with the Gambit version of the Zipi compiler.

4.5.1. Microbenchmarks

We now discuss the results from our microbenchmarks suite. A microbenchmark is a benchmark aimed at testing the performance of a single, isolated operation. For example, it may test the performance of the `+` operator when both operands are `int` objects. Microbenchmarks are useful to determine whether a targeted optimization (such as behaviors for arithmetic operators) is effective. However, we cannot use microbenchmarks to assert that Zipi is faster than other Python implementations overall. First, they cannot be used as an indicator of how well Zipi would perform on real-life programs. Second, they are biased toward features which have been optimized in Zipi. Furthermore, our microbenchmarks proved to be ineffective at comparing Zipi to PyPy. In most cases, a microbenchmark executes calculations and immediately discard the result. This allows PyPy to treat most benchmarks as dead code and skip their execution altogether. We compare Zipi to PyPy with benchmarks in Section 4.5.2. Zipi and cPython do not apply dead code elimination in most cases. Hence, the measured operation has generally been genuinely executed.

In this section we discuss the results obtained from our microbenchmarks. A description of the complete microbenchmark suite and how it is executed is available in Appendix B.

Microbenchmarks indicate that *behaviors* optimizations provide a significant performance boost. We observe a speed up which ranges from 15x to 30x for binary operations on small integers. We can say the same of binary operations on `float`s with a speed up between 3.0x and 3.5x.

Evaluating the truthiness of the condition of an `if`-statement is also significantly faster when the condition is a `bool` (between 8.7x and 14x), an `int` (20x), a `str` (between 4.2x and 6.7x) or a comparison between `int`s (18x) or `float`s (7.2x)

We also observe some corner cases where behaviors could not improve performance over cPython. For example, unary operators on booleans are slightly slower (0.7x in the case of unary `-`). This is a considerable setback in comparison to the same operation on small integers (7.2x). We explain this difference by the fact Zipi inlines unary operations only when the operand is either a small `int` or a `float`. We chose not to inline unary operations on `bool` objects to avoid code bloat which may affect performance [**54**] and compilation time. This is a reasonnable tradeoff since unary operations arithmetic operators are not as frequently used on `bool` when compared to `int` and `float` objects. A solution for the future would be to implement type inference to identify cases where the operand is likely a `bool` object and apply speculative inlining in those cases [**55, 56**].

Performance improvements from other optimizations also show up in the microbenchmarks. Writing a global variable is approximately 38x faster than cPython, while reading is 28x faster. Function calls are between 9.2x and 16x faster when the function has a fixed arity (no

keyword arguments or default arguments). Iteration is also faster on `range` objects (2.1x to 9.5x depending on the size of the `range`), tuples (4.0x), lists (4.9x) and generators (3.4x).

Finally, our microbenchmarks show some features which Zipi does not optimize yet and are significantly slower than cPython. For example, calling a function with keyword arguments is about ten times slower than with cPython.

4.5.2. Benchmarks

We compared Zipi to cPython and PyPy using both custom benchmarks and benchmarks extracted from the PyPerformance benchmark suite [**57**]. Due to Zipi still being at an early development stage, only four benchmarks from the PyPerformance test suite are supported at the moment, hence the need for custom benchmarks.

Our custom benchmarks suite includes the following programs: `ack`, `fib`, `queens`, `bague`, `sieve`. The code for all custom benchmarks is available in Appendix C. Benchmarks from the PyPerformance suite include `deltablue`, `fannkuck`, `richards` and `float` and are available online at `https://github.com/python/pyperformance`. Figure 4.7 compares the execution time of Zipi and PyPy when compared to cPython on all benchmarks.

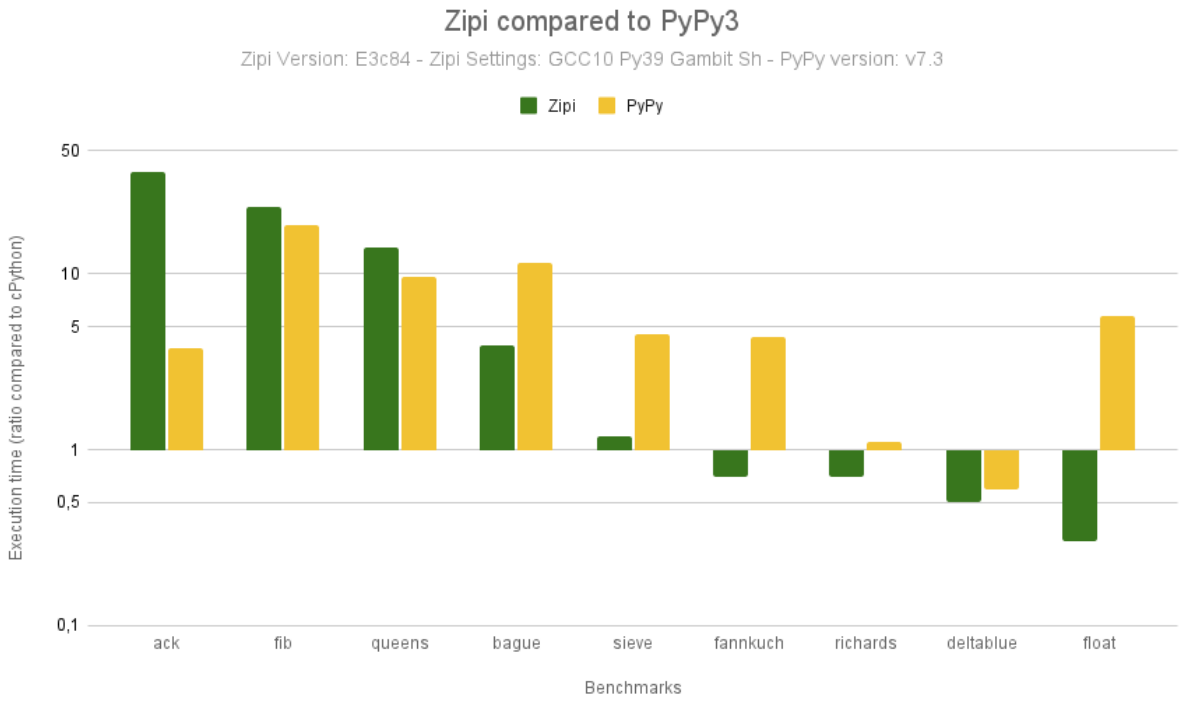

Fig. 4.7. Benchmarks results
The execution time of Zipi (green) and PyPy (yellow) on each benchmark is compared to that of cPython v3.9. A ratio higher than 1 indicates an execution faster than cPython. A ratio lower than 1 indicates a slower execution.

In comparison to cPython alone, Zipi fares especially well in the `ack` (38x faster), `fib` (24x) and `queens` (14x) benchmarks. We explain these results by the high reliance of these benchmarks on small integer arithmetic which have been optimized with *behaviors*. The `bague` (3.9x) and `sieve` (1.2x) are also slightly faster with Zipi. Finally, `fannkuck` (0.8x), `richards` (0.7x), `deltablue` (0.5x) and `float` (0.3x) execute slower than with cPython. We explain these last results by the fact PyPerformance benchmarks make extensive use of user-defined types. We invested little time in optimizing operations on user-defined types.

When compared to PyPy, the `ack` benchmarks (10x) shines by executing significantly faster with Zipi. In particular, the condition `(m > 0 and n == 0)` (see benchmark C.1) seems to execute significantly faster with Zipi. When used as a condition, the Zipi compiler recognizes that no `bool` object needs to be returned by either of the comparisons and thus uses `test` behaviors (see Section 3.2.5).

The benchmarks `queens` (1.5x) and `fib` (1.3x) execute slightly faster with Zipi, arguably due to behaviors. On the other hand, `deltablue` (1.0x), `richards` (0.7x), `sieve` (0.3x), `bague` (0.4x), `fannkuch` (0.2x) and `float` (0.06x) all execute slower with Zipi than PyPy. Interestingly, the `deltablue` benchmark executes faster with cPython than PyPy.

4.6. Summary

In this chapter we presented some implementation details of the Zipi compiler. We focussed on Zipi's capability to compile semantics and behaviors written with the `__compiler_intrinsics__` directive and reuse them in its runtime system. We then presented how this optimization fares in comparison to cPython and PyPy. Microbenchmarks seem to indicate that arithmetic and comparison operations on small integers are significantly faster with this optimization. The same operations also show a decent speed up for floating point and mixed-type operators. Benchmarks also confirm this observation as programs which make intensive usage of small integer arithmetic are faster with Zipi in comparison to both cPython and PyPy. However, Zipi is still in development and thus shows poor performance in other aspects such as user-defined types and functions with keyword arguments.

With these results we hope to convince the reader that the `__compiler_intrinsics__` directive allows to ease the development of an optimizing Python compiler. It does so by automating the writing of magic methods and semantics and by allowing optimizations which yield performance significantly superior to those of cPython and in some cases even to those of PyPy.

Chapter 5

Conclusion

In this thesis, we presented a semantics of both arithmetic and some non-arithmetic Python operations with a focus on reusability by optimizing compilers. We expressed this semantics using a syntax similar to that of Python, enhanced with primitive constructs which we named *compiler intrinsics*. These intrinsics allow low-level operations such as manipulating an object's value and resolving attributes within a type's MRO. They also serve as an annotation for compilers to apply optimizations which would be unsound in a generic Python program.

We explained how the semantics of most operations have a similar control flow, but call different magic methods. This allows to generate the semantics of all arithmetic operators and some non-arithmetic operators from a few templates where only magic method names change. Moreover, we explained how an operator's semantics is incomplete without also describing magic methods of built-in types. Fortunately, most magic methods have a similar control flow as well which permits generating them from a few templates using compiler intrinsics.

We proceeded by noting that magic methods of Python built-in types are conveniently immutable. Thus, the same operator applied to the same built-in types always calls the same magic method. This allows to define the notion of *behavior*, a specialization of a semantics for a given combination of built-in types. In particular, we showed how behaviors allow writing a specialization of the semantics of an operator without redundant type-checking and unboxing of an object's primitive value.

The existence of hundreds of behaviors for arithmetic operations on Python numeric types alone makes writing such specialized behaviors by hand impractical. We thus introduced `semPy`, a tool capable of partial evaluation of our semantics and magic methods. Using function inlining and branch resolution, it generates behaviors with all type checks and unnecessary unboxing of objects' values removed. The `semPy` source code is available at `https://github.com/omelancon/semPy`.

Finally, we integrated our semantics and behaviors in Zipi, an AOT optimizing Python compiler. Zipi recognizes our compiler intrinsics which allows to compile behaviors to specialized functions. When executing an optimized operation, Zipi's runtime system checks operands' types to dispatch the operation to the corresponding behavior.

We applied this method to arithmetic operators and operations which need to compute the truthiness of an object. In almost all microbenchmarks measuring those operations, this led to an increase in execution speed, oftentimes by a factor of more than 10 in comparison to cPython. We observed similar results on benchmarks making intensive use of arithmetic operators. In those cases, Zipi offered performance which rivals that of PyPy (faster by an order of magnitude in one case).

We conclude that our formal semantics is suited for the implementation of optimizing compilers. Being written in a syntax similar to that of Python, it can be integrated seamlessly to an existing compiler as it requires no change to the parser. Furthermore, the ability to generate the semantics from templates makes it straightforward to extend to operations and types which we did not yet cover. Finally, with the aid of `semPy`, this formal semantics yields an implementation of arithmetic operations which is faster than that of PyPy.

5.1. Future Work

We conclude this thesis with insights on what we think should be the next steps in developing our reusable semantics.

While most semantics can be expressed by calling the proper magic methods, some others display more complex behaviors. For example, Python has a rich semantics for calling functions with positional arguments, keyword arguments, default values and starred arguments. We should explore how to expand the semantics to implement all Python operations.

It would be interesting to explore how our reusable semantics can be extended to describe control flow and Python scoping rules. For example, one could provide control flow semantics of statements such as `for`, `with`, assignments and so on. Additionally, built-in libraries form a significant part of the Python language. We should explore how efficient it would be to use compiler intrinsics to write built-in modules, as well as additional built-in types.

The `semPy` tool we developed focuses on generating specializations of semantics for a given combination of types. Although this proved sufficient for arithmetic operators, we expect that we would gain from being able to generate specializations for specific values when writing additional semantics.

Finally, the Zipi compiler shows promising results performance-wise. However, some features were implemented in a naive way and display poor performance. We wish to continue developing Zipi to make it a fully compatible Python compiler with performance competitive with those of PyPy on all aspects.

References

[1] Laurence Tratt. Dynamically Typed Languages. *Advances in Computers*, 77:149–184, July 2009.

[2] Andreas Stuchlik and Stefan Hanenberg. Static vs. Dynamic Type Systems: An Empirical Study About the Relationship between Type Casts and Development Time. *ACM SIGPLAN Notices*, 47(2):97–106, October 2011.

[3] Leo A. Meyerovich and Ariel S. Rabkin. Empirical Analysis of Programming Language Adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 1–18, New York, NY, USA, October 2013.

[4] Stack Exchange Inc. Stack Overflow Developer Survey 2019. `insights.stackoverflow.com/survey/2019`, 2019.

[5] Stack Exchange Inc. Stack Overflow Developer Survey 2020. `insights.stackoverflow.com/survey/2020`, 2020.

[6] Guido van Rossum. The Making of Python. `https://www.artima.com/articles/the-making-of-python`, January 2003.

[7] Gergö Barany. Python Interpreter Performance Deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 1–9, New York, NY, USA, June 2014.

[8] The Python Software Foundation. Introduction. `https://docs.python.org/3.9/reference/introduction.html`, August 2021.

[9] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 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, pages 217–232, New York, NY, USA, October 2013.

[10] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 346–356, New York, NY, USA, June 2015.

[11] Manuel Serrano. Of JavaScript AOT compilation performance. *Proceedings of the ACM on Programming Languages*, 5(ICFP):70:1–70:30, August 2021.

[12] Ecma International. ECMAScript 2021 language specification (12th Edition). `https://www.ecma-international.org/publications-and-standards/standards/ecma-262`, June 2021.

[13] The Python Software Foundation. cPython. `www.python.org`.

[14] The Python Software Foundation. The Python Standard Library. `https://docs.python.org/3.9/library`, August 2021.

[15] Lennart Regebro. Language differences and workarounds. In *Porting to Python 3: An in-Depth Guide*. CreateSpace Independent Publishing Platform. `http://python3porting.com/differences.html`.

[16] The Python Software Foundation. The Python Language Reference. https://docs.python.org/3.9/reference, August 2021.

[17] The Python Software Foundation. Data model. https://docs.python.org/3.9/reference/datamodel.html, August 2021.

[18] Oracle. Primitive Data Types. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html, August 2021.

[19] Mozilla. Null. https://developer.mozilla.org/docs/Glossary/Null, August 2021.

[20] The Python Software Foundation. Emulating numeric types. https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types, August 2021.

[21] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 69–82, New York, NY, USA, October 1996.

[22] Michele Simionato. The Python 2.3 Method Resolution Order. https://www.python.org/download/releases/2.3/mro, August 2021.

[23] The Python Software Foundation. Function Definitions. https://docs.python.org/3/reference/compound_stmts.html#function-definitions, October 2021.

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

[25] S. Nanz and C. A. Furia. A Comparative Study of Programming Languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788, May 2015.

[26] Gregory Alan Hildstrom. Programming Language Performance Comparison. http://www.hildstrom.com/projects/langcomp/index.html, March 2021.

[27] Mohamed Ismail and G. Edward Suh. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 36–47, September 2018.

[28] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 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*, ICOOOLPS '09, pages 18–25, New York, NY, USA, July 2009.

[29] The PyPy Team. PyPy Speed. https://speed.pypy.org/.

[30] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, October 2004.

[31] Peter D. Mosses. The Varieties of Programming Language Semantics. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, Lecture Notes in Computer Science, pages 624–628, Berlin, Heidelberg, 2000.

[32] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, New York, NY, USA, January 2014.

[33] Daniele Filaretti and Sergio Maffeis. An Executable Formal Semantics of PHP. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 567–592, Berlin, Heidelberg, 2014.

[34] Mallku Soldevila, Beta Ziliani, Bruno Silvestre, Daniel Fridlender, and Fabio Mascarenhas. Decoding Lua: Formal semantics for the developer and the semanticist. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pages 75–86, New York, NY, USA, October 2017.

[35] David A. Schmidt. Programming language semantics. *ACM Computing Surveys*, 28(1):265–267, March 1996.

[36] David A. Watt. Executable semantic descriptions. *Software: Practice and Experience*, 16(1):13–43, January 1986.

[37] James F. Ranson, Howard J. Hamilton, and Philip W. L. Fong. A Semantics of Python in Isabelle/HOL. December 2008. `https://www.semanticscholar.org/paper/A-Semantics-of-Python-in-Isabelle%2FHOL-Ranson-Hamilton/60eb68ffbcd6974d26e22518a3c0e1126ff74296`.

[38] Gideon Smeding. *An Executable Operational Semantics for Python*. M.S. Thesis, University of Utrecht, January 2009.

[39] Junsong Li, Justin Pombrio, Joe Gibbs Politz, and Shriram Krishnamurthi. Slimming Languages by Reducing Sugar: A Case for Semantics-Altering Transformations. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 90–106, New York, NY, USA, October 2015.

[40] The Python Software Foundation. Future statements. `https://docs.python.org/3.9/reference/simple_stmts.html#future-statements`, August 2021.

[41] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 190–203, Berlin, Heidelberg, 1985.

[42] The Python Software Foundation. Object.__getattribute__. `https://docs.python.org/3/reference/datamodel.html#object.__getattribute__`, October 2021.

[43] The Python Software Foundation. Special Method Lookup. `https://docs.python.org/3/reference/datamodel.html#special-method-lookup`, October 2021.

[44] The Python Software Foundation. Assignment statements. `https://docs.python.org/3.9/reference/simple_stmts.html#assignment-statements`, August 2021.

[45] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Type Hints. PEP 484, September 2014. `www.python.org/dev/peps/pep-0484`.

[46] Robert Bruce Findler and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. Technical report, September 2007. `http://www.r6rs.org`.

[47] Manuel Serrano. Bigloo. `https://www-sop.inria.fr/mimosa/fp/Bigloo`.

[48] Marc Feeley. Gambit. `https://www.iro.umontreal.ca/~gambit/doc/gambit.pdf`.

[49] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[50] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, September 1989.

[51] Jiho Choi, Thomas Shull, and Josep Torrellas. Reusable Inline Caching for JavaScript Performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 889–901, New York, NY, USA, June 2019.

[52] Beatrice Åkerblom and Tobias Wrigstad. Measuring polymorphism in python programs. *ACM SIGPLAN Notices*, 51(2):114–128, October 2015.

[53] The PyPy Team. PyPy Performance. `https://www.pypy.org/performance.html`, December 2019.

[54] Peng Zhao and José Nelson Amaral. To Inline or Not to Inline? enhanced Inlining Decisions. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 405–419, Berlin, Heidelberg, 2004.

[55] Marc Feeley. Speculative Inlining of Predefined Procedures in an R5RS Scheme to C Compiler. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 237–253, Berlin, Heidelberg, 2008.

[56] Andreas Sewe, Jannik Jochem, and Mira Mezini. Next in line, please! exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 317–328, New York, NY, USA, October 2011.

[57] Victor Stinner. The Python Performance Benchmark Suite. `https://pyperformance.readthedocs.io`, August 2021.

Appendix A

Templates for semantics generation

In this appendix, we list all the templates for generating semantics and magic methods. Parameters of a template are highlighted as such within the template: {parameter}. Below each template, we also list parameters used to generate our semantics.

```
@define_semantics
def raise_binary_TypeError(op, x, y):
    raise TypeError("'"
                    + op
                    + "' not supported between instances of '"
                    + class_getattr(x, "__name__")
                    + "' and '"
                    + class_getattr(y, "__name__")
                    + "'")

@define_semantics
def raise_unary_TypeError(op, x):
    raise TypeError("bad operand type for unary"
                    + op
                    + ":' "
                    + class_getattr(x, "__name__")
                    + "'")
```

Fig. A.1. Formatting of error messages expressed as semantics

```
@define_semantics
def {operation}(x, y):
    def normal():
        magic_method = class_getattr(x, "{method}")
        if magic_method is absent:
            return reflected()
        else:
            result = magic_method(x, y)
            if result is NotImplemented:
                return reflected()
            else:
                return result

    def reflected():
        magic_method = class_getattr(y, "{rmethod}")
        if magic_method is absent:
            return err()
        else:
            result = magic_method(y, x)
            if result is NotImplemented:
                return err()
            else:
                return result

    def err():
        raise TypeError("unsupported operand type(s) for {operator}: '"
                        + class_getattr(x, "__name__") + "' and '"
                        + class_getattr(y, "__name__") + "'")

    return normal()
```

Template parameters

{operation}	{method}	{rmethod}	{operator}
add	__add__	__radd__	+
bitand	__and__	__rand__	&
bitor	__or__	__ror__	|
floordiv	__floordiv__	__rfloordiv__	//
lshift	__lshift__	__rlshift__	<<
matmul	__matmul__	__rmatmul__	@
mul	__mul__	__rmul__	*
pow	__pow__	__rpow__	**
rshift	__rshift__	__rrshift__	>>
sub	__sub__	__rsub__	-
truediv	__truediv__	__rtruediv__	/
xor	__xor__	__rxor__	^

Fig. A.2. Template for semantics of binary operators

```
def {magic_method}(self, other):
    if isinstance(self, int):
        if isinstance(other, int):
            return py_int_from_host(py_int_to_host(self)
                                    {operator} py_int_to_host(other))
        else:
            return NotImplemented
    else:
        raise TypeError("requires a 'int' but received a '"
                        + class_getattr(self, "__name__") + "'")
```

Template parameters

{magic_method}	{operator}
__add__	+
__and__	&
__or__	|
__mul__	*
__pow__	**
__sub__	-
__xor__	^

Fig. A.3. Template of non-division arithmetic magic methods of `int`

```
def {magic_method}(self, other):
    if isinstance(self, int):
        if isinstance(other, int):
            if py_int_to_host(other) != py_int_to_host(0):
                return py_int_from_host(py_int_to_host(self)
                                        {operator} py_int_to_host(other))
            else:
                raise ZeroDivisionError("integer division by zero")
        else:
            return NotImplemented
    else:
        raise TypeError("requires a 'int' but received a '"
                        + class_getattr(self, "__name__") + "'")
```

Template parameters

{magic_method}	{operator}
__floordiv__	//
__mod__	%
__truediv__	/

Fig. A.4. Template of division magic methods of `int`

```
def {magic_method}(self, other):
    if isinstance(self, int):
        if isinstance(other, int):
            if py_int_to_host(other) >= py_int_to_host(0):
                return py_int_from_host(py_int_to_host(self)
                                        {operator} py_int_to_host(other))
            else:
                raise ValueError("negative shift count")
        else:
            return NotImplemented
    else:
        raise TypeError("requires a 'int' but received a '"
                        + class_getattr(self, "__name__") + "'")
```

Template parameters

{magic_method}	{operator}
__lshift__	<<
__rshift__	>>

Fig. A.5. Template of bitwise-shift magic methods of `int`

```
def {magic_method}(self, other):
    if isinstance(self, int):
        if isinstance(other, int):
            return py_bool_from_host_bool(py_int_to_host(self)
                                          {operator} py_int_to_host(other))
        else:
            return NotImplemented
    else:
        raise TypeError("requires a 'int' but received a '"
                        + class_getattr(self, "__name__") + "'")
```

Template parameters

{magic_method}	{operator}
__eq__	==
__ne__	!=
__ge__	>=
__gt__	>
__le__	<=
__lt__	<

Fig. A.6. Template of comparison magic methods of `int`

```
def {magic_method}(self):
    if isinstance(self, int):
        return py_int_from_host({unary_operator} py_int_to_host(self))
    else:
        raise TypeError("requires a 'int' but received a '"
                        + class_getattr(self, "__name__") + "'")
```

Template parameters

{magic_method}	{operator}
__pos__	*no operator*
__neg__	-
__invert__	~

Fig. A.7. Template of unary magic methods of `int`

Appendix B

Microbenchmarks

We generate microbenchmarks for Zipi from a unique template shown in Figure B.1. Five parameters define each microbenchmark. We highlight those parameters in the template. The *number of iterations* defines how many iterations are executed. The *number of repetitions by iteration* defines how many times the benchmark's operation is duplicated inside the `while`-loop body. For each benchmark, we choose a high enough number of repetitions to minimize the overhead from the execution of the `while`-loop condition. We choose a number of iterations and number of repetitions which together give an execution time of around one second with the cPython interpreter. The *initialization* is an optional statement or sequence of statements required before the benchmark. For example, it may define a variable used in the microbenchmark's operation. The *loop initialization* is similar to the *initialization* parameter, but it is executed at the beginning of each iteration. For example, it may reset an accumulator. Finally, the *operation* is the expression or statement we wish to test with the microbenchmark.

In Table B.1, we provide the *operation*, *initialization* and *loop initialization* parameters used to generate each benchmark. Additionally, we provide the acceleration factor as a ratio between the execution time with cPython and Zipi. A ratio over `1` signifies the microbenchmarks executes faster with Zipi.

In general, Zipi and cPython do not apply dead code elimination in which case the *operation* is genuinely executed. Microbenchmarks which are known to undergo dead code elimination were excluded from Table B.1. Nonetheless, microbenchmarks greatly differ from real-life code. Thus, one implementation can have an advantage over the other on a microbenchmark without that advantage transpiring in real-life code or vice versa.

```
import time

bench_iterations = {number of iterations}
bench_duplications = {number of repetitions by iteration}

def bench_exec():
    {initialization}
    bench_counter = bench_iterations
    while bench_counter > 0:
        bench_counter -= 1

        {loop initialization}

        {operation}   # repeated 'number of iterations' times
        {operation}
        {operation}
        ...


bench_time = time.time()
bench_exec()
bench_time = time.time() - bench_time
print(str(bench_time) + ' seconds for ' \
      + str(bench_iterations * bench_duplications) + ' executions')
```

Fig. B.1. Template for microbenchmarks

Parameters of microbenchmarks

operation	*initialization*	*loop initialization*	Ratio
g1=x g2=x g3=x g4=x	global g1,g2,g3,g4 x=0	g1=0 g2=0 g3=0 g4=0	39x
g=x	global g x=0	g=0	38x
g1=x g2=x	global g1,g2 x=0	g1=0 g2=0	38x
g+=1	global g	g=0	32x
x=x*2	-	x=1	30x
x=g	global g x=0	g=0	28x
if zero: x=0	zero=0	-	22x
if non_zero: x=0	non_zero=1	-	21x
if x<y: x=0	x=1 y=2	-	18x

Parameters of microbenchmarks

operation	*initialization*	*loop initialization*	Ratio
`f(1)`	`def f(x):` ` pass`	-	18x
`x=x*y`	`y=2`	`x=1`	17x
`x+=1`	`x=1`	-	16x
`x=x-1`	`x=1`	-	16x
`x-=1`	`x=1`	-	16x
`x=x+1`	`x=1`	-	16x
`x=x-y`	`x=1` `y=1`	-	16x
`x=x+y`	`x=1` `y=1`	-	15x
`f()`	`def f():` ` pass`	-	15x
`if true:` ` x=0`	`true=True`	-	14x
`z=(x<y)`	`x=1` `y=2`	-	13x
`while x>0:` ` x-=1`	-	`x=250`	11x
`x=lst[0]`	`lst=[0]`	-	9.7x
`for x in range(1000):` ` pass`	-	-	9.5x
`x=tup[0]`	`tup=(0,)`	-	8.9x
`if false:` ` x=0`	`false=False`	-	8.7x
`x=~x`	`x=1`	-	8.0x
`x=-x`	`x=1`	-	7.2x
`if x<y:` ` x=0.0`	`x=1.0` `y=2.0`	-	7.2x
`if x<y:` ` x=0.0`	`x=True` `y=2.0`	-	7.2x
`if not_empty:` ` x=0`	`not_empty="foo"`	-	6.7x
`x=lst[y]`	`lst=[0]` `y=0`	-	6.3x
`for x in range(100):` ` pass`	-	-	4.9x
`for x in lst:` ` pass`	`lst=list(range(1000))`	-	4.9x

Parameters of microbenchmarks

operation	*initialization*	*loop initialization*	Ratio
`x=tup[y]`	`tup=(0,)` `y=0`	-	4.7x

`a.x`	`class A:`
	`  x=1`
	`a=A()`
		-	4.5x

`if empty:` `  x=0`	`empty=""`	-	4.2x
`for x in tpl:` `  pass`	`tpl=tuple(range(1000))`	-	4.0x
`x=x+y`	`x=1.5` `y=2.6`	-	3.5x

`for x in gen(100):`
`  pass`
	`def gen(i):`
	`  while i > 0:`
	`  yield i`
	`  i -= 1`
		-	3.4x

`x=x-y`	`x=1.5` `y=2.6`	-	3.3x
`x=x*y`	`x=1.5` `y=2.6`	-	3.0x
`x=-x`	`x=1.0`	-	2.6x
`for x in range(3):` `  pass`	-	-	2.1x
`x=s.__add__`	`s="a"`	-	1.9x

`while x>0:`
`  x-=1`
`  s2=s2+s1`
	`s1="a"`	`s2="b"`
		`x=100`
			1.6x

`while x>0:`
`  x-=1`
`  s2+=s1`
	`s1="a"`	`s2="b"`
		`x=100`
			1.5x

`while x>0:`
`  x-=1`
`  s2=s1+s2`
	`s1="a"`	`s2="b"`
		`x=100`
			1.5x

`z=(x==y)`	`x=1.5` `y="foo"`	-	1.0x
`x=x+y`	`x=1.5` `y=True`	-	0.9x
`x=m.__name__`	`m=int`	-	0.8x
`y=-x`	`x=True`	-	0.7x
`ord(s)`	`s="x"`	-	0.7x
`x=itertools.count`	`import itertools`	-	0.5x

Parameters of microbenchmarks

operation	*initialization*	*loop initialization*	Ratio
`x=m.__getattribute__`	`m=bool`	-	0.3x
`x=m.append`	`m=list`	-	0.28x
`x=s1+s2`	`s1="a"` `s2="b"`	-	0.27x
`f(x=1, y=2)`	`def f(x, y):` `  pass`	-	0.13x
`f(x=1)`	`def f(x):` `  pass`	-	0.11x
`f(x=3)`	`def f(x=1):` `  pass`	-	0.1x
`f(x=4, y=5)`	`def f(x=1, y=2):` `  pass`	-	0.09x

Table B.1. Parameters and acceleration factor of microbenchmarks

Appendix C

Benchmarks

```
import time

# Required by cPython to avoid a RecursionError
__import__('sys').setrecursionlimit(5000)

m = 3
n = 9

def ack(m, n):
    if m == 0:
        return n+1
    elif m > 0 and n == 0:
        return ack(m-1, 1)
    else:
        return ack(m-1, ack(m, n-1))

result = ack(3, 7) - 1021 # dry run

bench_time = time.time()
result = result + ack(m, n)
bench_time = time.time() - bench_time
print(str(bench_time) + ' seconds: ack(' + str(m) + ', ' + str(n) \
      + ') = ' + str(result))
```

Fig. C.1. ack benchmark
Computes the Ackermann function using recursion

```
import time

n = 12

def q(i, diag1, diag2, cols):
    if i == 0:
        return 1
    else:
        free = diag1 & diag2 & cols
        col = 1
        nsols = 0
        while col <= free:
            if col & free:
                nsols += q(i-1,
                           ((diag1-col)<<1)+1,
                           (diag2-col)>>1,
                           cols-col)
            col <<= 1
        return nsols

def queens(n):
    return q(n, -1, -1, (1<<n)-1)

result = queens(8) - 92 # dry run

bench_time = time.time()
result = result + queens(n)
bench_time = time.time() - bench_time
print(str(bench_time) + ' seconds: queens(' + str(n) + ') = ' + str(result))
```

Fig. C.2. `queens` benchmark
Solves the n-queens problem using bit sets and recursion

```
import time

n = 33

def fib(n):
    if n<2:
        return n
    else:
        return fib(n-1) + fib(n-2)

result = fib(30) - 832040 # dry run

bench_time = time.time()
result = result + fib(n)
bench_time = time.time() - bench_time
print(str(bench_time) + ' seconds: fib(' + str(n) + ') = ' + str(result))
```

Fig. C.3. `fib` benchmark
Computes the n^{th} fibonacci number using double recursion

```
import time

nombre_de_coups = 0
jeu = []

une_pierre, une_case_vide = 1, 0

def init_jeu( nombre_de_pierres ):
    global nombre_de_coups
    global jeu
    nombre_de_coups = 0
    jeu = [ 0 ] * nombre_de_pierres

    for i in range( nombre_de_pierres ):
        jeu[ i ] = une_pierre

def la_case( n ):
    return n - 1;

def enleve_la_pierre( n ):
    if jeu[ la_case( n ) ] == une_pierre:
        jeu[ la_case( n ) ] = une_case_vide

def pose_la_pierre( n ):
    if jeu[ la_case( n ) ] == une_case_vide:
        jeu[ la_case( n ) ] = une_pierre

def autorise_mouvement( n ):
    if n == 1:
        return True
    elif n == 2:
        return jeu[ la_case( 1 ) ] == une_pierre
    else:
        if jeu[ la_case( n - 1 ) ] != une_pierre:
            return False
        b = True
        for i in range( la_case( n - 2 ) + 1 ):
            b = b and (jeu[ i ] == une_case_vide)
        return b

def enleve_pierre( n ):
    global nombre_de_coups
    nombre_de_coups = nombre_de_coups + 1

    if autorise_mouvement( n ):
        enleve_la_pierre( n )

def pose_pierre( n ):
    global nombre_de_coups
    nombre_de_coups = nombre_de_coups + 1

    if autorise_mouvement( n ):
        pose_la_pierre( n )
```

```
def run( nombre_de_pierres ):
    def bague( n ):
        if n == 1:
            enleve_pierre( 1 )
            return
        elif n == 2:
            enleve_pierre( 2 )
            enleve_pierre( 1 )
            return
        else:
            bague( n - 2 )
            enleve_pierre( n )
            repose( n - 2 )
            bague( n - 1 )

    def repose( n ):
        if n == 1:
            pose_pierre( 1 )
            return
        elif n == 2:
            pose_pierre( 1 )
            pose_pierre( 2 )
            return
        else:
            repose( n - 1 )
            bague( n - 2 )
            pose_pierre( n )
            repose( n - 2 )

    init_jeu( nombre_de_pierres )
    bague( nombre_de_pierres )
    res = 0;

    if nombre_de_pierres == 1:
        res = 1
    elif nombre_de_pierres == 2:
        res = 2
    elif nombre_de_pierres == 10:
        res = 682
    elif nombre_de_pierres == 14:
        res = 10922
    elif nombre_de_pierres == 20:
        res = 699050
    elif nombre_de_pierres == 24:
        res = 11184810
    elif nombre_de_pierres == 25:
        res = 22369621
    elif nombre_de_pierres == 26:
        res = 44739242
    elif nombre_de_pierres == 27:
        res = 89478485
    elif nombre_de_pierres == 28:
        res = 178956970

    return "res=" + str( res ) + " nb-coups=" + str( nombre_de_coups )
```

```
def main( bench, n, arg ):
    res = 0
    k = (n+9) // 10
    i = 1

    print( bench, "(", n, ")..." )

    for n in range( n ):
        if (n % k) == 0:
            print( i )
            i = i + 1
        run( arg )

def bench_exec( name, N, arg ):
    bench_time = time.time()
    main( name, N, arg )
    bench_time = time.time() - bench_time
    print(str(bench_time) + ' seconds for ' + str(N) \
          + ' executions of: ' + name)

bench_exec( "bague", 2, 20 )
```

Fig. C.2. bague benchmark
Solves the Baguenaudier puzzle recursively

```
import time
import sys

class Cons(object):
    def __init__(self, a, d):
        self.car = a;
        self.cdr = d;

def interval( min, max ):
    if min > max:
        return []
    else:
        return Cons( min, interval( min + 1, max ) )

def sfilter( p, l ):
    if l == []:
        return l
    else:
        a = l.car
        r = l.cdr

        if p( a ):
            return Cons( a, sfilter( p, r ) )
        else:
            return sfilter( p, r )

def remove_multiples_of( n, l ):
    return sfilter( lambda m: m % n != 0, l )

def sieve( max ):
    def filter_again( l ):
        if l == []:
            return l
        else:
            n = l.car
            r = l.cdr

            if n * n > max:
                return l
            else:
                return Cons(n,
                            filter_again( remove_multiples_of( n, r ) ) )
    return filter_again( interval( 2, max ) )

def length( lst ):
    res = 0

    while lst != []:
        res = res + 1
        lst = lst.cdr

    return res
```

```
def list2array( lst ):
    len = length( lst )
    res = []

    for i in range( len ):
        res[ i ] = lst.car
        lst = lst.cdr

    return res

expected_result = [
   2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
   79, 83, 89, 97 ];

def main( bench, n, arg ):
    s100 = sieve( 100 )
    res = 0
    k = (n+9) // 10
    i = 1

    print( bench, "(", n, ")..." )

    for n in range( n ):
        if (n % k) == 0:
            print( i )
            i = i + 1
        res = sieve( arg )

    return res

def bench_exec( name, N, arg ):
    bench_time = time.time()
    main( name, N, arg )
    bench_time = time.time() - bench_time
    print(str(bench_time) + ' seconds for ' + str(N) \
          + ' executions of: ' + name)

sys.setrecursionlimit( 3100 )
bench_exec( "sieve", 100, 3000 )
```

Fig. C.2. `sieve` benchmark
Finds prime numbers using a sieve