License d'Informatique Fondamentale
École Normale Supérieure de Lyon

Internship Report
ROMAIN LIAUTAUD

Building a Python front-end for *BOLDR*

A journey in the internals of CPython

JUNE-JULY 2017
UNDER THE SUPERVISION OF KIM NGUYEN

ABSTRACT

During my six-week internship in the VALS team at the LRI, I was tasked with building a Python library which would interface with their existing *BOLDR* project, and would allow Python developers to write database queries using idiomatic Python constructs such as user-defined functions and comprehensions.

This report is a summary of this work, which involved topics ranging from linear algebra or assembly decompilation to lambda calculus.

TABLE OF CONTENTS

Introduction

While database management systems are usually built around a specific data model (e.g. relational or graph-based), and are interfaced with a specific query language (e.g. SQL or MapReduce), most real-world applications which depend on databases are written in general-purpose programming languages. As most of these languages lack native support for querying databases, this leads to a problem of **impedance mismatch**: developers have to write their application logic and their queries in different languages, and can't reuse their existing application logic in their queries.

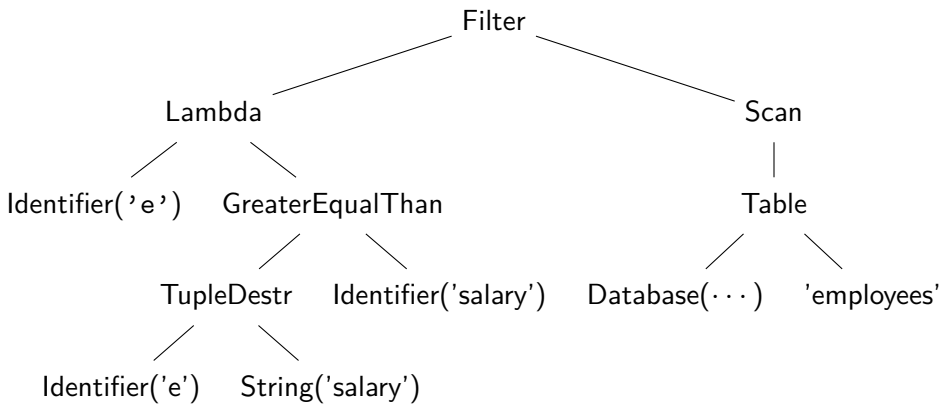
The most popular way around this problem is through the use of ORMs —for Object Relational Mappers— like [4] SQLAlchemy or [2] Doctrine, which wrap database operations in object-oriented interfaces. This allows developers to write database queries using only constructs from their host language, abstracting away the target query language. However, most ORM solutions fail to totally solve the problem, in part because of the fundamental differences between the object-oriented and the relational paradigms, and because they usually yield badly optimized queries.

Another approach is that of **language-integrated querying**, which works by extending the syntax and semantics of the host language in such a way that database queries become first-class citizens of the language. This is done either by writing queries using regular language constructs, and using some form of static analysis to isolate the parts of programs which are actually queries; or by adding new syntactic constructs for queries —like what Microsoft did with [3] LINQ.

In this vein, a team at the LRI in Orsay is working on *BOLDR* —for Breaking boundaries Of Language and Data representations—, a modular framework for language-integrated querying which enabled the evaluation of queries containing application logic (and user-defined functions in particular, as they are poorly supported by LINQ). *BOLDR* allows developers to write data-querying programs in the host language of their choice, treating queries as first-class citizens, and allowing them to be mixed with application logic.

Specifically, as shown on Appendix B, *BOLDR* detects and isolates the parts of the program which represent queries (cf. ①), translates them into an intermediate representation (cf. ②) called the *QIR* —for Query Intermediate Representation— and optimizes them to avoid query avalanches and make the most out of database optimizations. This representation is then translated into the right database query language (cf. ③), at which point *BOLDR* queries the database and finally sends the results back to the host language runtime (cf. ④). This is explained in greater detail in [5] and [6].

Figure 1: *QIR* term for a simple query on the `employees` table



This intermediate representation, whose precise definition is given in Appendix C, is at the core of the *BOLDR* framework. It is essentially a lambda calculus extended with native support for scalar data types —integers, floats or booleans—, lists and tuples, which also features operators from linear algebra.

This makes it easy to represent any data query —regardless of the data model— in the *QIR*, and then to convert this representation into a variety of database query languages. Take the term from Fig. 1 for instance: it represents a query for all the employees in a table called `employees` whose salary is greater than a given `'salary'` variable. If we choose a value of 4000 for `'salary'`, for instance, we would be able to translate this term into the SQL query `SELECT * FROM employees AS e WHERE e.salary >= 4000`.

Thanks to the modular nature of the framework, support for new host programming languages can be added to *BOLDR* through the implementation of so-called front-ends (the "Host language" part of the diagram). Up until now, *BOLDR* has had front-ends for a toy programming language called SL and for the statistical programming language R, so the goal of this internship was to extend the framework by building a front-end for a general-purpose language: Python. However, the SL and R front-ends are implemented by altering the compiler and the interpreter, so they require their users to drop the reference implementaion of their language in favor of the custom, "*BOLDR*-enabled" one.

As this severely curbs developer adoption of the framework, we tried a different approach for this internship: our Python front-end **should work on a standard CPython interpreter**, so we can't alter the compiler or interpreter in any way. This makes things much more complicated, as it prevents us from adding syntax elements to the language, or from performing the kind of static analysis that was used in the other front-end implementations.

This report is the summary of my work trying to build that front-end. *To try and give you a good understanding of that work while remaining within the imposed page limit, I deliberately chose to focus the next sections on examples and intuitive explanations, and added the formalized definitions and algorithms as appendices to the report.*

1 Choosing the query syntax

We first have to settle on the syntax that will be used to write database queries in Python, as it will influence some of our choices later on. This syntax has to meet several requirements:

- It should be **expressive**, meaning that it should provide a way to write all the relational operators that are supported by the *QIR* —namely Scan, Filter, Project, Sort, Limit, Group and Join.
- It should be **idiomatic**, which is an informal way of saying that it should feel as familiar as possible to Python developers.
- It should be **implementable without changing** the Python grammar, as the main goal of this internship was to build a Python front-end for the *QIR* that would work on a standard CPython interpreter.

1.1 Basic syntax

The simplest syntax that I could come up with is to directly construct *QIR* terms in Python. For instance, to query the names of all employees whose salary is greater than `salary`, we would write the following Python code.

```
def at_least(salary):
    return Project(
        lambda e: {'name': e.name},
        Filter(
            lambda e: e.salary >= salary,
            Scan(employees)
        )
    )
```

where `employees` is a Python object which holds a reference to the database table which contains the employees.

Although this syntax is expressive by construction, the nested terms make it a bit cumbersome to indent and parenthesize, and having to read the query "from the inside out" to figure out what it does feels too counter-intuitive, so I decided to try and find a clearer alternative.

1.2 Method chaining syntax

One such alternative is the "method chaining" syntax, which many object-relational mapping libraries —such as [4] SQLAlchemy or [2] Doctrine— already use. The idea is that, when we want to apply relational operators op_1 and op_2 to x consecutively, instead of writing $op_2(op_1(x, y_1, \cdots, y_n), z_1, \cdots, z_m)$, we write $x \cdot op_1(y_1, \cdots, y_n) \cdot op_2(z_1, \cdots, z_m)$. This allows us to rewrite our example query in a much clearer way:

```
def at_least(salary):
    return employees.scan()
                    .filter(lambda e: e.salary >= salary)
                    .project(lambda e: {'name': e.name})
```

1.3 Comprehension syntax

While the "method chaining" syntax is already a good improvement upon the basic one, it is still not quite idiomatic. As *The Zen of Python* puts it, "There should be one —and preferably only one— obvious way to do it", and it turns out Python already provides a well-known syntax to filter and project over iterable objects: **comprehension expressions**.

For instance, to build a list `pairs` of $(x, x+1)$ for every integer $x \in [\![0, n[\![$ that is divisible by 3, a Python developer would typically write:

```
pairs = [(x, x + 1) for x in range(0, n) if x % 3 == 0]
```

In this case, every element of `pairs` is evaluated immediately. If one wanted to delay the evaluation of those elements until they are accessed for the first time, they would replace the brackets with parenthesis to form a generator comprehension:

```
pairs = ((x, x + 1) for x in range(0, n) if x % 3 == 0)
```

In an effort to provide an idiomatic syntax, I decided to replicate this idiom for queries which can be expressed using only Scan, Filter and Project operators. This way, we can rewrite the previous query in a way that feels much more natural to Python developers:

```
def at_least(salary):
    return ({'name': e.name}
                for e in employees
                if e.salary >= salary)
```

This syntax is not perfect, however, as it can't express all the operators (e.g. Group). Ultimately, I decided that the best compromise would be to implement both the method chaining and the comprehension syntaxes, and let developers choose which syntax they use depending on how complex their query is.

2 Introspecting Python objects

Now that we have settled on a syntax, we must find a way to implement it, while keeping in mind that our implementation must **work on a standard CPython interpreter**. To illustrate how this complicates things a bit, we'll use the example of

```
employees.scan().filter(lambda e: e.salary >= salary)
```

Because we must turn that query into the *QIR* term on Fig. 1, we have to translate the Python `lambda` into a syntactic *QIR* Lambda.

Ideally, we would hook into the compiler to perform the translation at compile-time, as this would give us access to the function's abstract syntax tree which we could then turn into a *QIR* term (see the SimpleLanguage front-end implementation from [6]). However, because we cannot alter the compiler, we have to work with whatever information is available at runtime: the opaque closure that gets passed to the `filter` function. Essentially, we must try to "reverse engineer" the closure object that we get at runtime in the hope of retrieving the code that was used to define it.

2.1 Using the `inspect` module

Thankfully, Python provides a few mechanisms for **introspection**, which is the ability for a program to examine itself at runtime.

One of those mechanisms is the `inspect` module: given a Python function, it can retrieve the Python code that was used to define it and return that code as a string, as shown in Fig. 2. We could then parse this string into an abstract syntax tree, which we would finally turn into a *QIR* term.

However, `inspect.getsource` isn't perfect: it only supports functions which were declared in a file —not from the command-line— using the `def` keyword, so it won't work with `lambda e: e.salary >= salary`.

2.2 Using the `dis` module

Another introspection mechanism, which is only available on CPython, is the `dis` module. Unlike `inspect`, it works with all function objects (even those which were not declared using `def`), but the trade-off is that we don't get the code or AST of our function.

Figure 2: Example of using the `inspect` module on a function

```
def f(x):
    return x + 1

import inspect
inspect.getsource(f)
>>> 'def f(x):\n\treturn x + 1\n'
```

Instead, we get its **bytecode**, which is the internal representation of a Python program in the CPython interpreter. A program written in this intermediate language is a sequence of instructions for a stack machine which operates on Python objects.

Figure 3: Example of using the `dis` module on a function

```
def f(x):
    return x + 1

import dis
dis.dis(f)
>>>   0 LOAD_FAST                0 (x)
>>>   3 LOAD_CONST               1 (1)
>>>   6 BINARY_ADD
>>>   7 RETURN_VALUE
```

As we can see on Fig. 3, each instruction in the bytecode is a combination of a line number (e.g. `3`), an operation (e.g. `LOAD_CONST`) and sometimes an argument (e.g. the integer `1`). In this simple case, to execute `return x + 1`, the CPython stack machine starts by fetching the value of the variable named `x` in the environment and pushes it onto the stack; then it pushes the integer `1` onto the stack; then it pops the top two elements from the stack, pushes their sum onto the stack; and finally returns the top value from the stack.

Formally, we start by defining the set $\mathcal{O}$ of all the **operations** supported by the bytecode. For the sake of clarity, I have omitted a few non-essential operations; the complete list of operations can be found at [1].

$$
\begin{aligned}
\mathcal{O}_{stack} &= \texttt{NOP, POP\_TOP, ROT\_TWO, ROT\_THREE, DUP\_TOP, CALL\_FUNCTION} \\
&\quad \texttt{LOAD\_ATTR, STORE\_ATTR, DELETE\_ATTR,} \\
&\quad \texttt{BINARY\_SUBSCR, STORE\_SUBSCR, DELETE\_SUBSCR} \\
\mathcal{O}_{env} &= \texttt{LOAD\_CONST, LOAD\_NAME,LOAD\_FAST, STORE\_FAST, DELETE\_FAST,} \\
&\quad \texttt{LOAD\_GLOBAL, STORE\_GLOBAL, DELETE\_GLOBAL,} \\
\mathcal{O}_{bool} &= \texttt{COMPARE\_OP} \\
\mathcal{O}_{unary} &= \texttt{UNARY\_POSITIVE, UNARY\_NEGATIVE, UNARY\_NOT, ...}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{O}_{binary} &= \texttt{BINARY\_POWER, BINARY\_MULTIPLY, BINARY\_ADD, ...} \\
\mathcal{O}_{inplace} &= \texttt{INPLACE\_POWER, INPLACE\_MULTIPLY, INPLACE\_ADD, ...} \\
\mathcal{O}_{branch} &= \texttt{POP\_JUMP\_IF\_TRUE, POP\_JUMP\_IF\_FALSE,} \\
&\quad\ \texttt{JUMP\_IF\_TRUE\_OR\_POP, JUMP\_IF\_FALSE\_OR\_POP} \\
\mathcal{O}_{jump} &= \texttt{JUMP\_FORWARD, JUMP\_ABSOLUTE}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{O} = \mathcal{O}_{stack} \cup \mathcal{O}_{env} \cup \mathcal{O}_{bool} \cup \mathcal{O}_{unary} \\
\cup \mathcal{O}_{binary} \cup \mathcal{O}_{inplace} \cup \mathcal{O}_{branch} \cup \mathcal{O}_{jump}
\end{aligned}
$$

Then, we define a **bytecode program** as a couple $\mathcal{P} = (\mathcal{L},\ (\mathcal{C}_n)_{n\in\mathcal{L}})$, where $\mathcal{L} \subset \mathbb{N}$ is a finite set of line numbers and $(\mathcal{C}_n)_{n\in\mathcal{L}}$ is a sequence of $(o_n, a_n) \in \mathcal{O} \times \Omega$, Ω being the set of all Python objects, o_n being the operation of line n and a_n its argument. We chose to define $\mathcal{L} \subset \mathbb{N}$ instead of simply using $[\![0, n[\![$, with n the number of lines in the program, because the line numbers are not contiguous (as shown in Fig. 3).

To model the execution of the CPython stack machine, we define a **state** of the machine as a $\langle n,\ S,\ E\rangle$ tuple, where n is the line number of the current instruction, $S \subset \Omega$ the current stack and $E : \texttt{string} \to \Omega$ the current environment. We will write $E[x]$ the value of x in the environment E, and $E[x \mapsto y]$ the copy of E in which x has the value y. On the next page, we finally link those states together by defining the **semantic** σ of the machine, which deterministically gives the next state of the machine depending on its current one.

To make things easier, we define $s(n)$ as the smallest integer $m \in \mathcal{L}$ such that $m > n$. Essentially, $s(n)$ represents the line number that comes after the one at line n. Also, we map every operation $o \in \mathcal{O}_{unary} \cup \mathcal{O}_{binary} \cup \mathcal{O}_{inplace}$ to the mathematical function $\underline{o}$ that it represents. For instance, BINARY_ADD is the $+$ function, and BINARY_AND is the $\wedge$ function.

▷ *Appendix D contains the complete definition of σ.*

Eventually, I settled for this type of introspection —using the `dis` module and working with bytecode— instead of the first one, for several reasons. First of all, as mentioned above, it is more flexible than `inspect` which only works under certain conditions.

Using the bytecode representation is also useful when dealing with the scope of variables —local, nonlocal or global— as each scope has its own set of instructions —suffixed with `_FAST`, `_NAME` and `_GLOBAL` respectively; as well as when dealing with comprehensions.

Also, on a more personal note, it sounded much more interesting to try and convert an assembly-like representation to a lambda calculus than it would have been to convert an abstract syntax tree to a lambda term.

3 Translating bytecode into *QIR* terms

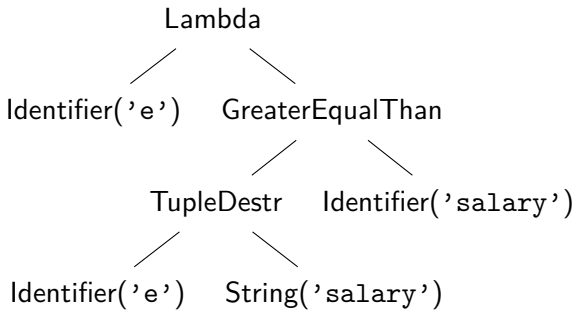
Now that we have access to the bytecode of any function at runtime, we must find a way to "decompile" it into a *QIR* term. For instance, for the simple `lambda e: e.salary >= salary` function, we would like to translate the bytecode on Fig. 4 to the term on Fig. 5.

My first approach was to map each instruction of the bytecode to a *QIR* Lambda taking as first argument a *QIR* list representing the stack —recall that *QIR* lists are Lisp-like lists

Figure 4: Bytecode for the `lambda e: e.salary >= salary` function

```
 0 LOAD_FAST                0 (e)
 3 LOAD_ATTR                0 (salary)
 6 LOAD_GLOBAL              0 (salary)
 9 COMPARE_OP               5 (>=)
12 RETURN_VALUE
```

Figure 5: *QIR* term for the `lambda e: e.salary >= salary` function



encoded using the ListNil and ListCons constructors— and returning the state of the stack after the instruction is executed. I would then assemble those Lambdas; effectively writing an emulator of the CPython stack machine using the *QIR* lambda calculus.

However, I rejected this approach since it often yielded *QIR* terms that would later be hard to optimize or rewrite into SQL queries (the terms were "polluted" by nested ListCons and ListDestr which represented successive push-pop operations).

Instead, I decided to simulate the execution of the bytecode directly at translation time, on a **"symbolic stack machine"** which manipulates *QIR* terms instead of Python objects. After a few refinements to handle the translation of conditional statements, loops and comprehensions, I eventually came up with an algorithm to translate the bytecode of most functions into an equivalent *QIR* term, which I will explain below.

3.1 Building the control flow graph

The bytecode of `lambda e: e.salary >= salary` was simple enough that it didn't contain jump instructions, but that isn't true in most cases, so the first step of this algorithm is to split the bytecode into blocks of "linear code", i.e. instructions which follow each other in every execution path of the bytecode. We then add edges between those chunks to indicate where jumps can occur, and get the **control flow graph** of the bytecode.

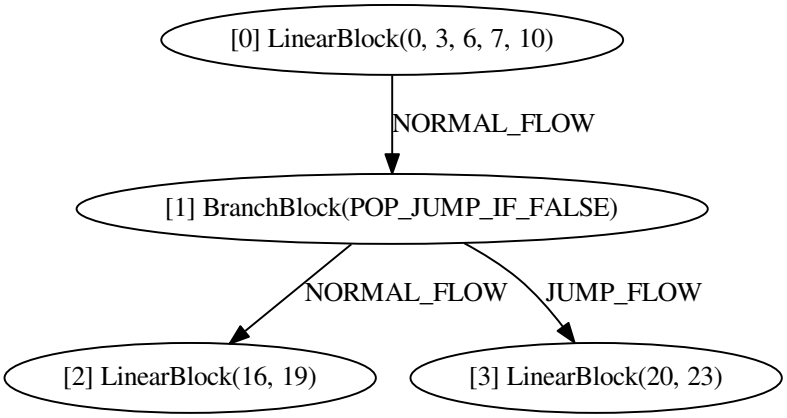
Take the `is_even` function from Fig. 6, for instance. Because of the `if` statement, its bytecode contains a `POP_JUMP_IF_FALSE` instruction, so we can split it into four blocks: one from line `0` to line `10`, which evaluates `x % 2 == 0`, one for the instruction on line `13`, one for the "true branch" and one for the "false branch". The result is the graph on Fig. 7.

Figure 6: Python code and bytecode program for the `is_even` function

```
def is_even(x):
    if x % 2 == 0:
        return True

    return False
```

```
 0 LOAD_FAST                0 (x)
 3 LOAD_CONST               1 (2)
 6 BINARY_MODULO
 7 LOAD_CONST               2 (0)
10 COMPARE_OP               2 (==)

13 POP_JUMP_IF_FALSE       20

16 LOAD_CONST               3 (True)
19 RETURN_VALUE

20 LOAD_CONST               4 (False)
23 RETURN_VALUE
```

Figure 7: Control flow graph for the `is_even` function



Formally, given the bytecode $\mathcal{P} = (\mathcal{L}, (\mathcal{C}_n)_{n\in\mathcal{L}})$, we define its control flow graph $\mathcal{G}_\mathcal{P}$ as a directed graph $(V_\mathcal{P}, E_\mathcal{P})$ where $V_\mathcal{P} \in P(\mathcal{L})$ is a partition of the line numbers such that for every $A \in V_\mathcal{P}$, all the lines in A follow each other in every execution path of $\mathcal{P}$, and for every $(u, v) \in E_\mathcal{P}$, it is possible to jump from the last instruction of u to the first instruction of v. As you can see on Fig. 7, we also label the blocks and the edges:

$$\begin{aligned} \mathit{block\_type} &: V_\mathcal{P} \to \{\texttt{LINEAR}, \texttt{BRANCH}, \texttt{JUMP}\} \\ \mathit{edge\_type} &: E_\mathcal{P} \to \{\texttt{NORMAL\_FLOW}, \texttt{JUMP\_FLOW}\} \end{aligned}$$

The `BRANCH` and `JUMP` labels are used for blocks which contain a single instruction from $\mathcal{O}_{branch}$ and $\mathcal{O}_{jump}$ respectively, while the `LINEAR` label is used everywhere else. For edges, the `JUMP_FLOW` label is used between a `BRANCH` block and the target of the jump, while `NORMAL_FLOW` is used everywhere else.

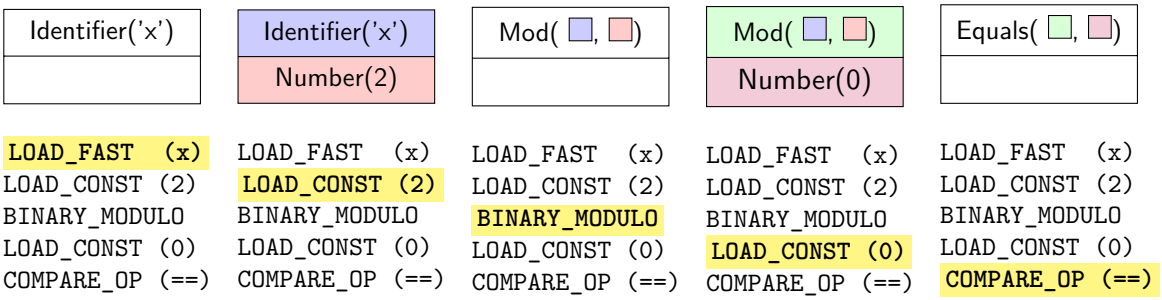
▷ *Appendix E contains the algorithm which builds the graph of a given $\mathcal{P}$ in linear time.*

3.2 Executing the symbolic stack machine

Now that we have split the bytecode into blocks of "linear code", we will simulate the execution of the CPython stack machine on each of the `LINEAR`-labeled blocks using a **symbolic stack machine**. This machine has almost the same semantic as the original machine, but it manipulates *QIR* terms instead of Python objects, and uses a list of bindings instead of an environment.

Let's start with an example, which will hopefully give us an intuition of how it works: Fig. 8 shows the execution of the machine on the first block of Fig. 7 (which corresponds to instructions `0` to `10` of Fig. 6), starting from instruction `0` with an empty stack and an empty bindings list.

Figure 8: Step by step execution of the symbolic machine *(current instruction is highlighted)*



Notice that, while the `LOAD_FAST` instruction normally fetches the value of `x` from the environment and pushes it onto the stack, we don't know that value yet, so we push the term Identifier(`'x'`) instead (which is why this machine is "symbolic").

After the last instruction, we can see that the top of the stack contains the QIR term Equals(Mod(Identifier('x'), Number(2)), Number(0)), which is exactly the QIR translation of the `x % 2 == 0` condition from the `if` statement.

As mentioned above, the symbolic machine uses a list of variable bindings instead of an environment. The idea is that, every time it encounters a `STORE_FAST` or `STORE_GLOBAL` instruction, the machine adds a $(name,\ value)$ couple to the list of bindings. We will use those bindings later in the algorithm, as we will wrap the translated QIR term inside a Application(Lambda(Identifier($name$), ...), $value$) for each $(name,\ value)$ couple in the list.

Formally, we define a **state** of the symbolic machine as a $\langle n,\ S,\ B\rangle$ tuple, with S a stack of *QIR* terms and B a list of $(name, value) \in \texttt{string} \times QIR$. Then, we define the **semantic** σ' of the symbolic machine. It is essentially a slightly modified version of σ.

We use the $\texttt{encode} : \Omega \to QIR$ function, which converts basic Python objects —such as strings, integers, floats or lists— into their *QIR* equivalent. Also, we map every operation $o \in \mathcal{O}_{unary} \cup \mathcal{O}_{binary} \cup \mathcal{O}_{inplace}$ to the *QIR* constructor $\overline{o}$ that it represents. For instance, $\overline{\texttt{BINARY\_ADD}}$ is the Add constructor, and $\overline{\texttt{BINARY\_AND}}$ is the And constructor.

Stack-related operations.

$$
\begin{array}{rcll}
\sigma'\langle n,\, S,\, B\rangle & = & \langle s(n),\, S,\, B\rangle & \text{if } o_n = \texttt{NOP} \\
\sigma'\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, S,\, B\rangle & \text{if } o_n = \texttt{POP\_TOP} \\
\sigma'\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, s\cdot s\cdot S,\, B\rangle & \text{if } o_n = \texttt{DUP\_TOP} \\
\sigma'\langle n,\, s_1\cdot s_2\cdot S,\, B\rangle & = & \langle s(n),\, s_2\cdot s_1\cdot S,\, B\rangle & \text{if } o_n = \texttt{ROT\_TWO} \\
\sigma'\langle n,\, s_1\cdot s_2\cdot s_3\cdot S,\, B\rangle & = & \langle s(n),\, s_2\cdot s_3\cdot s_1\cdot S,\, B\rangle & \text{if } o_n = \texttt{ROT\_THREE} \\
\sigma'\langle n,\, v_{a_n}\cdots v_1\cdot f\cdot S,\, B\rangle & = & \langle s(n),\, \texttt{currify}(f, v_1, \cdots, v_{a_n})\cdot S,\, B\rangle & \text{if } o_n = \texttt{CALL\_FUNCTION} \\
 & & & \\
\sigma'\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, \mathsf{TupleDestr}(s,\, \mathsf{String}(id))\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_ATTR},\, id) \\
\sigma'\langle n,\, s_1\cdot s_2\cdot S,\, B\rangle & = & \langle s(n),\, \mathsf{TupleCons}(\mathsf{String}(id),\, s_2,\, s_1)\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_ATTR},\, id) \\
\sigma'\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, \mathsf{TupleCons}(\mathsf{String}(id),\, \texttt{null},\, s)\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{DELETE\_ATTR},\, id) \\
 & & & \\
\sigma'\langle n,\, s_1\cdot s_2\cdot S,\, B\rangle & = & \langle s(n),\, \mathsf{TupleDestr}(s_2,\, s_1)\cdot S,\, B\rangle & \text{if } o_n = \texttt{BINARY\_SUBSCR} \\
\sigma'\langle n,\, s_1\cdot s_2\cdot s_3\cdot S,\, B\rangle & = & \langle s(n),\, \mathsf{TupleCons}(s_1,\, s_3,\, s_2)\cdot S,\, B\rangle & \text{if } o_n = \texttt{STORE\_SUBSCR} \\
\sigma'\langle n,\, s_1\cdot s_2\cdot S,\, B\rangle & = & \langle s(n),\, \mathsf{TupleCons}(s_1,\, \texttt{null},\, s_2)\cdot S,\, B\rangle & \text{if } o_n = \texttt{DELETE\_SUBSCR}
\end{array}
$$

With $\texttt{currify}(f, v_1, \cdots, v_m) = \mathsf{Application}(\mathsf{Application}(\mathsf{Application}(v_1,\, f),\, \cdots),\, v_m)$.

Environment-related operations.

$$
\begin{array}{rcll}
\sigma'\langle n,\, S,\, B\rangle & = & \langle s(n),\, \texttt{encode}(c)\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_CONST},\, c) \\
 & & \langle s(n),\, \mathsf{Identifier}(id)\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_NAME},\, id) \\
 & & \langle s(n),\, \mathsf{Identifier}(id)\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_FAST},\, id) \\
 & & \langle s(n),\, \mathsf{Identifier}(id)\cdot S,\, B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_GLOBAL},\, id) \\
 & & & \\
\sigma\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, S,\, (id, s)\cdot B\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_FAST},\, id) \\
 & & \langle s(n),\, S,\, (id, s)\cdot B\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_GLOBAL},\, id) \\
 & & & \\
\sigma\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, S,\, (id, \texttt{null})\cdot B\rangle & \text{if } \mathcal{C}_n = (\texttt{DELETE\_FAST},\, id) \\
 & & \langle s(n),\, S,\, (id, \texttt{null})\cdot B\rangle & \text{if } \mathcal{C}_n = (\texttt{DELETE\_GLOBAL},\, id)
\end{array}
$$

Arithmetic operations.

$$
\begin{array}{rcll}
\sigma'\langle n,\, s\cdot S,\, B\rangle & = & \langle s(n),\, \overline{i_n}(s)\cdot S,\, B\rangle & \text{if } o_n \in \mathcal{O}_{unary} \\
\sigma'\langle n,\, s_1\cdot s_2\cdot S,\, B\rangle & = & \langle s(n),\, \overline{o_n}(s_2, s_1)\cdot S,\, B\rangle & \text{if } o_n \in \mathcal{O}_{binary} \\
 & & \langle s(n),\, \overline{o_n}(s_2, s_1)\cdot S,\, B\rangle & \text{if } o_n \in \mathcal{O}_{inplace} \\
 & & \langle s(n),\, \overline{a_n}(s_2, s_1)\cdot S,\, B\rangle & \text{if } o_n = \texttt{COMPARE\_OP}
\end{array}
$$

Using that semantic, we can build a simple algorithm which takes a `LINEAR`-labeled block v and a starting stack S_0 as input, and stores the stack and bindings list after the last instruction of v is executed. Because `LINEAR`-labeled blocks don't contain jumping or branching instructions by construction, we know that the computation of $\sigma'\langle n_0,\, S_0,\, B_0\rangle$ will run in finite time, ending on the last instruction of v.

▷ *Appendix F contains the pseudo-code for this algorithm.*

3.3 Gluing it all together

Now that we are able to simulate the execution of the CPython stack machine on each of the `LINEAR`-labeled blocks in a given control flow graph, we need a way to translate the whole graph into a *QIR* term.

For the time being, we make the assumption that **there are no cycles** in the control flow graph of our bytecode, making it a directed acyclic graph. That just means that we won't consider functions with loops; we will come back to them later.

Because of that, we can compute a **topological ordering** of the graph, i.e. a function $order : V_{\mathcal{P}} \to [\![1, |V_{\mathcal{P}}|]\!]$ such that for every $(u, v) \in E_{\mathcal{P}}$, we have $order(u) < order(v)$. We also add the constraint that $order(start) = 1$, $start$ being the block which contains the first instruction of $\mathcal{P}$. Let COMPUTEORDERING be the function which computes this ordering; it is easy to implement using a depth-first search of the graph starting from $start$.

The algorithm that I came up with to do this translation in linear time works in two passes. The first pass **executes** the symbolic machine on each `LINEAR`-labeled blocks in the graph, in ascending order. This way, when we execute a block v, we have already executed all its predecessors, so we can use their final stack S_f as the starting stack S_0 of this block.

Note that we assume that all the predecessors have the same final stack, which is why this algorithm works in linear time. Otherwise, we would have to execute each path of the graph separately, which would make the algorithm exponential. While I haven't been able to formally prove this invariant, mostly because I couldn't reverse engineer the entire CPython compiler, it appears to hold in every test case I could come up with.

The second pass **expresses** the blocks, meaning that it assigns a *QIR* term to each of them. It starts with the leaves: if they end with the `RETURN_VALUE` instruction, it assigns them the head of S_f, otherwise it assigns them Null. If the block has a non-empty list of bindings B_f, it wraps that term into a Application(Lambda(Identifier($name$), ...), $value$) for each ($name$, $value$) couple in the list. Then, for each block in descending order:

- if the block has the `LINEAR` or `JUMP` label, it assigns it the term that was assigned to its only successor;
- if the block has the `BRANCH` label, it assigns it the Conditional($cond$, $true$, $false$) term, with $cond$ the head of the block's S_0 and $true$ or $false$ the term that was assigned to its successor from the "true" or "false" branch respectively.

▷ *Appendix K contains a formalized version of this algorithm.*

3.4 Dealing with `while` loops

Now that we know how to translate the bytecode of simple functions into *QIR* terms, we would like to extend this translation to functions containing `while` loops —as on Fig 9.

As we can already tell from looking at the bytecode, the control flow graph that will be produced for the `closest_multiple` function will contain cycles, as the instruction on line 29 jumps back to line 3. That, in turn, will violate the assumption that we made at the beginning of the previous section, so we won't be able to translate this function as is.

Figure 9: Python code and bytecode program for the `closest_multiple` function

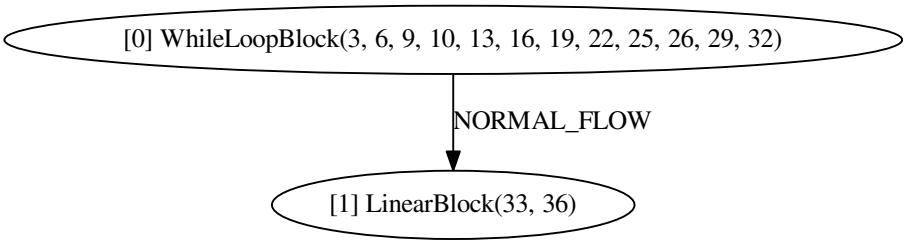
```
def closest_multiple(x, n):
    while x % n != 0:
            x = x + 1
    return x
```

```
 0 SETUP_LOOP              30 (to 33)
 3 LOAD_FAST                0 (x)
 6 LOAD_FAST                1 (n)
 9 BINARY_MODULO
10 LOAD_CONST               1 (0)
13 COMPARE_OP               3 (!=)
16 POP_JUMP_IF_FALSE       32

19 LOAD_FAST                0 (x)
22 LOAD_CONST               2 (1)
25 BINARY_ADD
26 STORE_FAST               0 (x)
29 JUMP_ABSOLUTE            3
32 POP_BLOCK

33 LOAD_FAST                0 (x)
36 RETURN_VALUE
```

Thankfully, the CPython compiler prefixes the bytecode of every `while` loop with a `SETUP_LOOP` instruction, which takes as an argument the line offset of the first instruction after the loop. In our case, the instruction on line `0` tells us that instructions `3` to `32` are part of a `while` loop. That makes it possible to "isolate" those instructions —which would create cycles— and encapsulate them into a special `LOOP`-labeled block.

Fig. 10 shows the control flow graph for our function, with the `LOOP` block encapsulating instructions `3` to `32`.

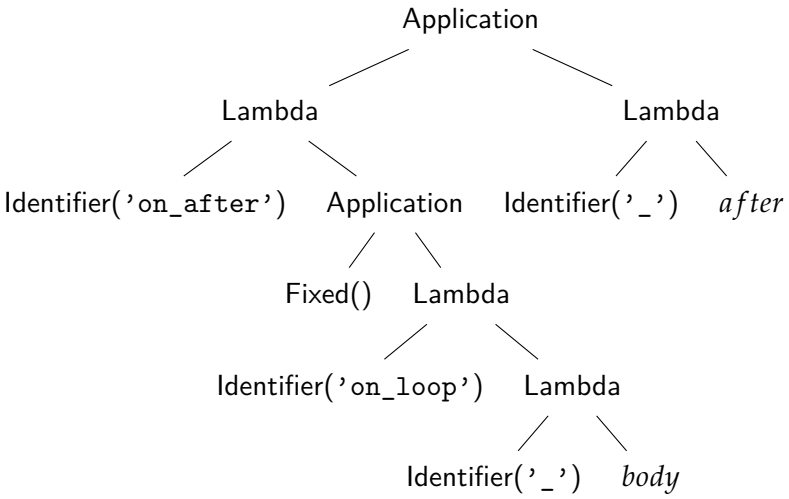
Figure 10: Control flow graph for the `closest_multiple` function



We must then find a way to translate those encapsulated instructions into a *QIR* term. Because the *QIR* is essentially a lambda calculus, we can use the fixed-point combinator $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ to express the loop as a recursive function. Precisely, let *body* and $after$ be *QIR* translations of the body of the loop and the code to execute after the loop, then we would translate the entire `LOOP`-labeled block into the *QIR* term on Fig. 11.

But how do we find *body* in practice? Let's start by introducing a special type of block, labeled `PLACEHOLDER`, which doesn't hold any instructions, but rather a *QIR* term —let's

Figure 11: *QIR* translation of a `while` loop

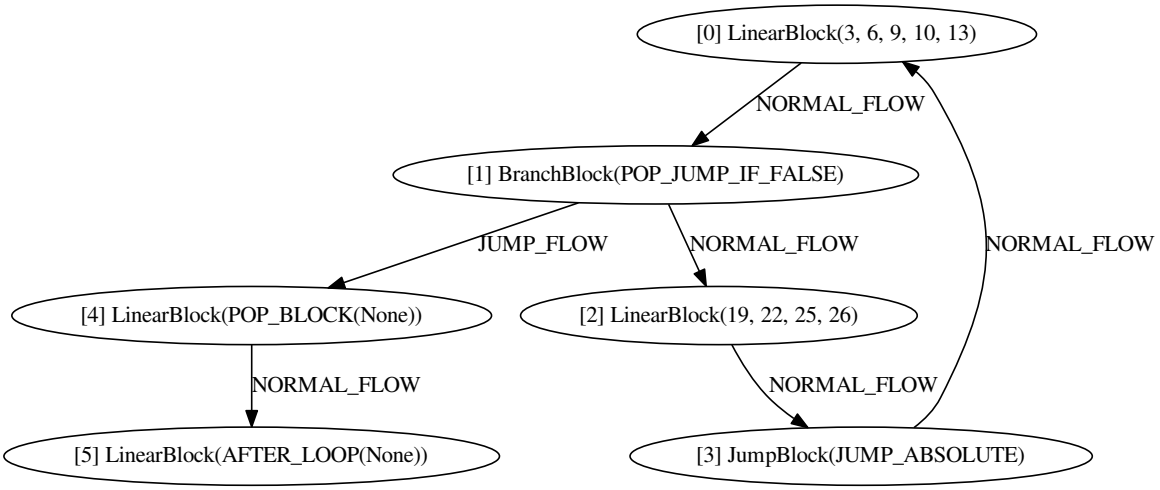


Fixed() is a *QIR* shorthand for the fixed-point combinator.

name it *e*. Nothing happens for this block on execution pass, but on expression pass it is always assigned the term *e*. Essentially, this type of block provides a way to force a portion of the control flow graph to be expressed into a term of our choice.

The idea, then, is to reuse our BUILDGRAPH algorithm, but this time on the instructions inside the LOOP-labeled block, which gives us the graph on Fig. 12. That makes it fairly easy to identify the blocks which exit the loop (here block `[5]`) and those which start another iteration (here block `[3]`).

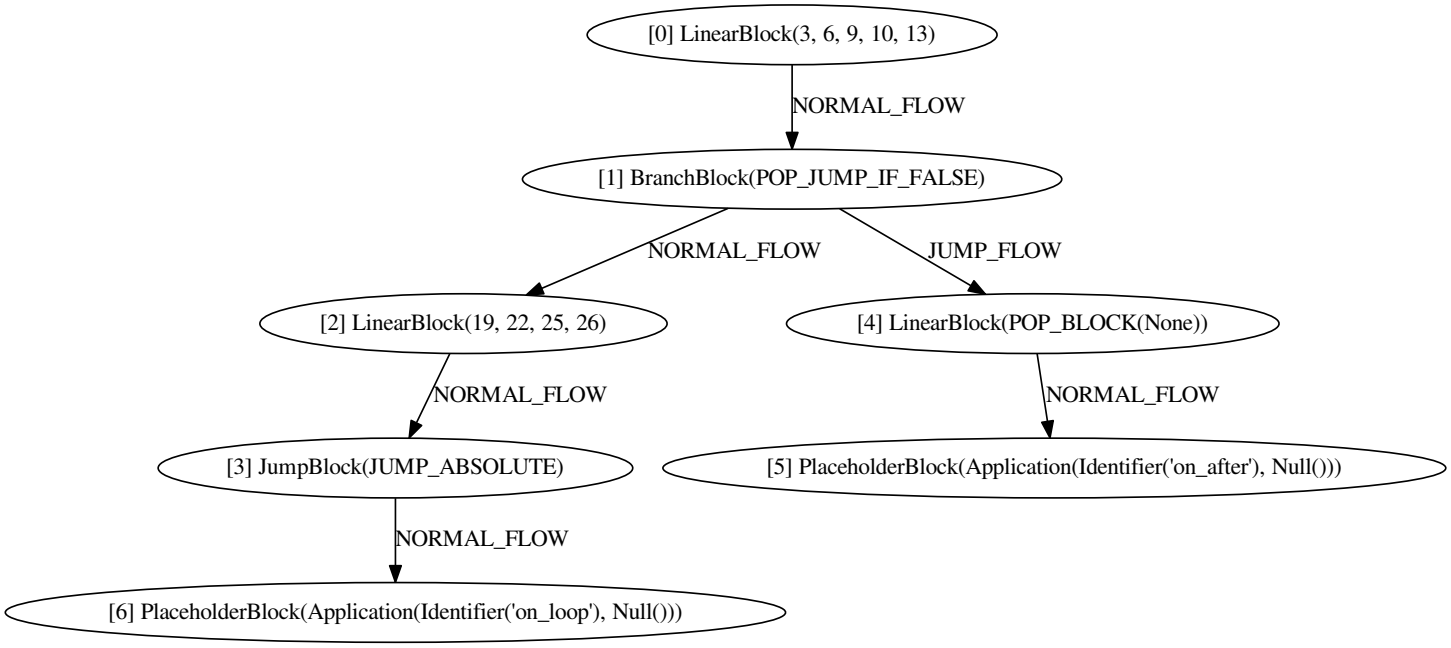
Figure 12: Control flow graph of the instructions inside the LOOP-labeled block



We then replace the blocks which exit the loop with a `PLACEHOLDER` block containing `Application(Identifier('on_loop'), Null())`; and the blocks which start another iteration with a `PLACEHOLDER` block containing `Application(Identifier('on_after'), Null())`, while making sure that we remove edges that go back to the beginning of the loop (so that the modified graph is cycle-free).

That gives us the modified graph on Fig. 13. All that's left to do is to run the TRANSLATEGRAPH algorithm on that modified graph, and the *QIR* term that it returns is our *body*.

Figure 13: Control flow graph of the instructions inside the `LOOP`-labeled block *(modified)*



Then, at the expression phase, we will already have a *QIR* term for the only successor of the `LOOP`-labeled block —thanks to the descending topological ordering— so we can use it as our $after$. Now that we have our *body* and $after$ terms, we can assign the term from Fig. 11 to the entire `LOOP`-labeled block, and we're done.

To implement this idea, we will have to change the BUILDGRAPH algorithm to recognize the `SETUP_LOOP` instruction, and change the TRANSLATEGRAPH algorithm so that the execution and expression of `LOOP`-labeled nodes gets handled by the EXECUTELOOP and EXPRESSLOOP algorithms.

▷ *On Appendix M, the `build_graph` function on line 61, the `execute` functions on line 605 and 700 and the `express` function on line 704 contain those detailed algorithms.*

3.5 Dealing with `for` loops

The next step is to extend the translation to functions containing `for` loops. Even though they look similar to `while` loops, the major difference is that, instead of checking a boolean condition over and over again, they **iterate** over an iterable object — according to the Python documentation, "An object capable of returning its members one at a time". For the sake of simplicity, we'll only consider iterating over lists.

Figure 14: Python code and bytecode program for the `closest_multiple` function

```
                              0 SETUP_LOOP              15 (to 18)
                              3 LOAD_FAST                0 (l)
                              6 GET_ITER
def contains(l, x):           7 FOR_ITER                 7 (to 17)
    for y in l:              10 STORE_FAST               2 (y)
        if x == y:
            return True      13 LOAD_CONST               1 (True)
                             16 RETURN_VALUE
    return False             17 POP_BLOCK

                             18 LOAD_CONST               2 (False)
                             21 RETURN_VALUE
```

As we can see on Fig. 14, line 7 is where the magic happens. According to the documentation of the CPython stack machine, the `FOR_ITER` instruction fetches the list on top of the stack, and tries to read its next element. If it works, it pushes that element onto the stack; otherwise —when the entire list is exhausted— it jumps to the offset passed as an argument.

At first, I thought I could translate `for` loops into simple *QIR* Projects, because this operator essentially applies a function to every element in a *QIR* list. However, a common Python idiom is to change the value of a variable inside the body of a `for` loop, so that the new value can be used in the next iterations. That would have been impossible to reproduce using Project, as it doesn't allow state to be passed from one iteration to the next.

Instead, we have to use a convoluted combination of —once again— the fixed-point combinator and the ListDestr constructor to translate the `for` loop into a *QIR* function which recursively reads each element of the list and executes the loop body with it.

Precisely, let *iter* be the *QIR* term representing the list being iterated on, $after$ the *QIR* translation of the code to execute after the loop, and *name* the name of the variable holding the current element (`'y'` here), then the loop gets translated to the term on Appendix. L.

From then on, the process is quite similar to the one of `while` loops. We build a control flow graph using the instructions from the `LOOP`-labeled block, we identify the blocks which either jump to the first instrution after the loop or jump back to the beginning of the loop, we replace them by well-chosen `PLACEHOLDER` blocks, and —after a few minor adjustements— we get the value of *body* by translating this modified graph into a *QIR* term. The value of *iter* can be found on top of the S_0 of the `LOOP`-labeled block; and we can use whatever string we want for *name*, as long as it is unique —to avoid name clashes in nested loops.

▷ *On Appendix M, the `execute` functions on line 605 and 727 and the `express` function on line 736 contain those detailed algorithms.*

3.6 Dealing with comprehensions

Lastly, we would like to extend the translation to list and generator comprehensions (which we already mentioned in the first section). Interestingly enough, the bytecode produced by the CPython compiler for `for` loops and list comprehensions is almost the same; which is understandable given that a list comprehension is, in essence, a simple `for` loop which pushes an element into a list at each iteration.

It would thus be tempting to reuse the previous algorithm for comprehensions, but it is a bad idea: the term on Fig. L is so convoluted that it is —almost— impossible to optimize, and while that term is a good translation of a general `for` loop, it can be greatly simplified in the more restricted case of comprehensions.

According to the Python grammar, the most general form of a comprehension is:

$$[f(x_1, \cdots, x_n)\ \texttt{for}\ x_1\ \texttt{in}\ l_1()\ \texttt{if}\ c_1(x_1)\ \cdots\ \texttt{for}\ x_n\ \texttt{in}\ l_n(x_1, \cdots, x_{n-1})\ \texttt{if}\ c_n(x_1, \cdots, x_n)]$$

In effect, this builds the list of all the $f(x_1, \cdots, x_n)$ with $x_1 \in \{y \in l_1() \mid c_1(x_1) = \texttt{true}\}$, and then for a given x_1, with $x_2 \in \{y \in l_2(x_1) \mid c_2(x_1, x_2) = \texttt{true}\}$, and so on.

For the sake of simplicity, this section will focus on the basic case of:

$$[f(x)\ \texttt{for}\ x\ \texttt{in}\ l\ \texttt{if}\ c_1(x)\ \cdots\ \texttt{if}\ c_m(x)]$$

This builds the list of all the $f(x)$ with $x \in \{y \in l \mid c_1(y) = \texttt{true} \wedge \cdots \wedge c_m(y) = \texttt{true}\}$. Note that we will try to give ourselves a good intuition of the algorithm rather than overly formalizing it. For those interested in the details of the implementation, see Appendix M.

Figure 15: Python code and bytecode program for the `comp` function

```
def comp(n):
    return [x + 5 for x \
                  in range(n) \
                  if x % 3 == 0]
```

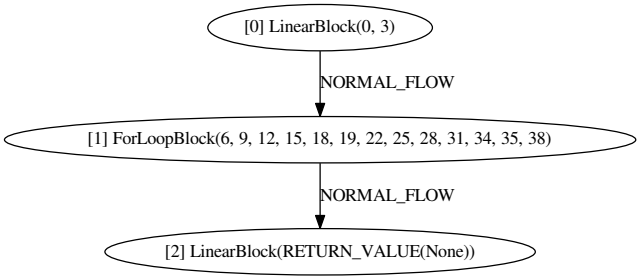
```
 0 LOAD_CONST       1 (<code object>)
 3 LOAD_CONST       2
 6 MAKE_FUNCTION    0
 9 LOAD_GLOBAL      0 (range)
12 LOAD_FAST        0 (n)
15 CALL_FUNCTION    1
18 GET_ITER
19 CALL_FUNCTION    1
22 RETURN_VALUE
```

Fig. 15 shows the bytecode for a simple function which returns a list comprehension. Notice how the bytecode doesn't contain information about the comprehension; instead, that information is hidden in a constant (the `<code object>`), and is accessible by calling `dis.dis(comprehension.__code__.co_consts[1])`. That gives us the bytecode on Fig. 16, which is —indeed— very similar to that of a `for` loop.

In order to translate it to a *QIR* term, we could essentially use two approaches. The first one is to "reverse engineer" the portion of the CPython compiler which turns comprehensions into bytecode. Then, by knowing which sequence of instructions gets generated for which combination of `for`s and `if`s in the comprehension, we could try to find the reverse function (which would give us the combination of `for`s and `if`s that were used to define the comprehension just by looking at its instructions).

Figure 16: Bytecode and graph for the list comprehension inside the `comp` function

```
 0 BUILD_LIST           0
 3 LOAD_FAST            0 (.0)
 6 FOR_ITER            32 (to 41)
 9 STORE_FAST           1 (x)
12 LOAD_FAST            1 (x)
15 LOAD_CONST           0 (3)
18 BINARY_MODULO
19 LOAD_CONST           1 (0)
22 COMPARE_OP           2 (==)
25 POP_JUMP_IF_FALSE    6
28 LOAD_FAST            1 (x)
31 LOAD_CONST           2 (5)
34 BINARY_ADD
35 LIST_APPEND          2
38 JUMP_ABSOLUTE        6
41 RETURN_VALUE
```



The second one, which I chose because it is more robust, is to generate a control flow graph from the instructions inside the `LOOP`-labeled block —see Fig 18— to try and understand "what they do". Precisely, we want to isolate three things:

- The **iterable** l, i.e. the list which is being iterated upon;
- The **projection** f, i.e. the function being applied to all the elements of the iterable;
- The **filters** $c_1, \cdots, c_m$, i.e. the list of conditions which have to be met for an element to be appended to the output list.

Getting the iterable is fairly easy: it is on top of the stack right before the `FOR_ITER` instruction (on line `6`) gets executed. Then, to get the projection and the filters, we have to really understand how the bytecode of comprehensions works: it all starts on line `0` with the `BUILD_LIST` instruction, which creates an empty list and pushes it onto the stack.

This list —which is effectively the output list— will receive the $f(x)$ for all the x which pass through the filters; and will be returned at the very end (on line `41`). However, only certain paths in the graph from Fig 18 reach the `LIST_APPEND` instruction (on line `35`), which is where the $f(x)$ are added to the output list; and that's how filters work.

This leads to the following algorithm. First, we identify the block which contain the `LIST_APPEND` instruction (in our case block `[3]`), and execute it with the symbolic stack machine. However, we hijack the semantic for the `LIST_APPEND` instruction so that, instead of appending the term on top of the stack to the output list, that term gets stored to use it as our projection later —let's call it *proj*.

Then, we backtrack the paths which go from that block to the first block of the graph, and everytime we encounter a `BRANCH`-labeled block, we store the *QIR* term that must evaluate to `true` in order for our path to be taken. This gives us a conjunction of *QIR* terms that must be true in order for an element to be appended to the output list —in other words we get our list of filters.

In our case, the only `BRANCH`-labeled block between block `[3]` and block `[0]` is block `[2]`; and in order to take the `NORMAL_FLOW`-labeled edge from block `[2]`, the *QIR* term Eq(Mod(Identifier('x'), Number(3)), Number(0)) must evaluate to `true`. We now have the *QIR* term for our projection, and the terms for $c_1, \cdots, c_m$, so all that's left to do is to replace the output list with the term on Fig. 17.

▷ *On Appendix M, lines 780 to 877 contain the precise implementation of this algorithm.*

Figure 17: *QIR* translation of a simple list comprehension

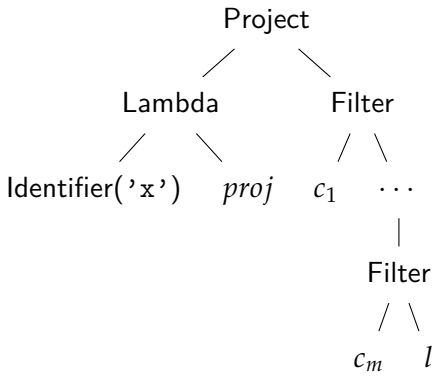
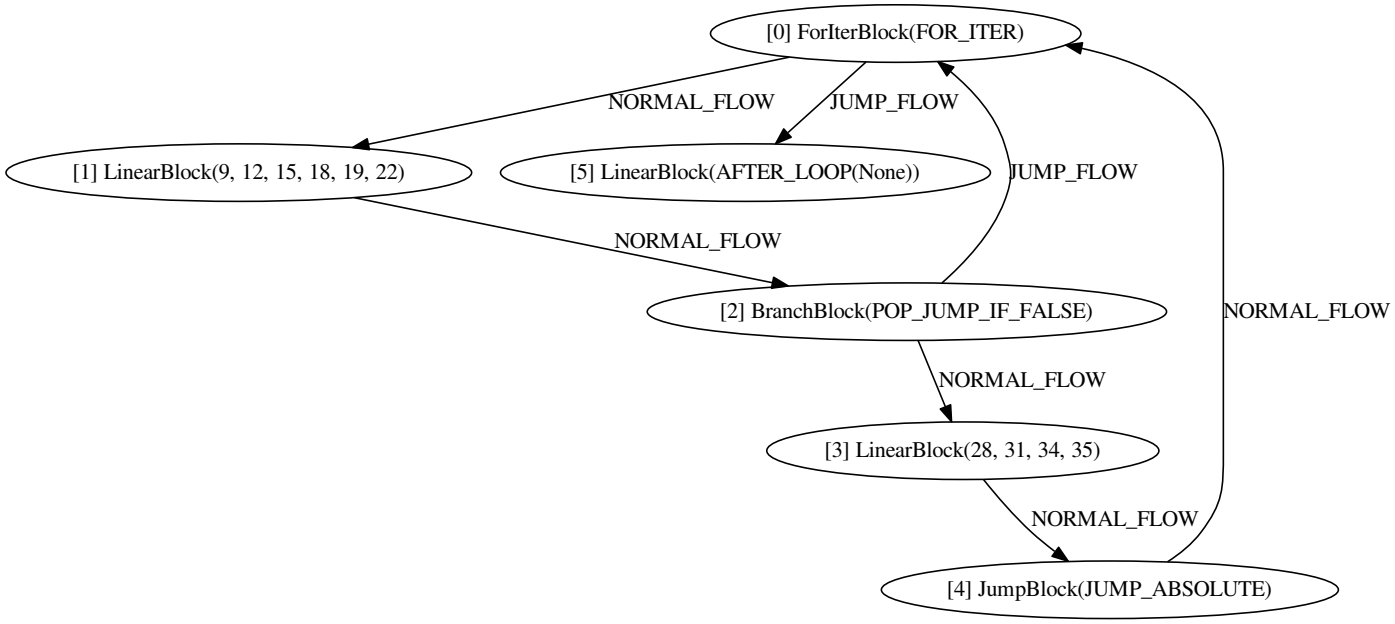


Figure 18: Graph of the instructions inside the `LOOP`-labeled block



4 Implementation details

Now that we have settled over a syntax that developers will use to write database queries in Python, and that we have come up with an algorithm which translates that syntax into *QIR* terms, the front-end is almost complete.

Let's take a closer look at what happens when we execute the following piece of code.

```
1  from boldr import local, batch
2
3  db = boldr.db(
4      driver='postgresql',
5      name='app',
6      host='localhost',
7      port=3306)
8
9  query1 = db.table('users')
10             .filter(lambda u: u.verified)
11             .project(lambda u: {'score': u.age * u.points})
12             .filter(lambda u: u.score >= 100)
13
14 query2 = (max(u.score, 0) for u in query1 if u.score < 50)
15
16 print(local % query1)
17 print(batch % query2)
```

4.1 Converting Python values to QIR terms

The first thing that happens, on line 9, is the evaluation of the right-hand side of the variable assignment. The call to `db.table('users')` creates a `Query` object which encapsulates the *QIR* term Scan(Table(Database($\cdots$))).

Then, calling the `filter` method on this object with argument f_1 returns a new instance of `Query`, which now encapsulates the term Filter(`encode(`f_1`)`, Scan(Table(Database($\cdots$)))). This happens again with the call to the `project` method (with f_2), then to `filter` (with f_3), and in the end `query1` receives a `Query` object which encapsulates the term below.

Filter(`encode(`f_3`)`, Project(`encode(`f_2`)`, Filter(`encode(`f_1`)`, Scan(Table(Database($\cdots$))))))

The `encode` function is used to convert Python values to *QIR* term. It works with "simple" values like integers or strings (for instance, `encode(2)` returns Number(2)), with "complex" values like lists or objects (`encode([1, 2])` returns ListCons(Number(2), ListCons(Number(1), ListNil()))) and even with closure objects like f_1, f_2 or f_3, which get converted to *QIR* terms using the algorithm from the previous section.

In the case of closure objects which refer to a built-in Python function, however, `encode` returns a special Builtin node containing the name of the function. This way, once the *QIR* term gets translated into the language of the target database (e.g. SQL), that node can be replaced by a call to an equivalent database-native function (for instance, `random.randint` would be replaced by the native `RAND()` function from SQL).

Note that, if the algorithm is unable to convert a closure object into a *QIR* term (e.g. if the bytecode for that closure contains unsupported instructions), the `encode` function wraps

the bytecode inside a special Bytecode node, which will then be executed on the database side using an embedded Python interpreter.

On line 14, the evaluation of the right-hand side returns an instance of `Generator`. Due to the lazy semantics of generators (see the first section), the terms of the generator are not computed at this stage.

4.2 Evaluating QIR terms

Line 16 and 17 are where the queries represented by the `Query` and `Generator` object actually get executed. Precisely, `local` and `batch` are two operators which force the conversion of their right operand into a *QIR* term (if not already done, which is the case for the `Generator` object), then evaluate this term, and finally convert the resulting term back to a Python object using the `decode` function. The two operators have different semantics.

- The `local` operator evaluates the term directly on the Python client. In our case, it fetches the entire contents of the `users` table in memory, and applies the Filter, Project and Filter operators in memory as well. Unless you know what you are doing, using this operator is usually a bad idea.
- The `batch` operator, on the other hand, evaluates the term remotely. In practice, the term is first serialized (see below), then sent over TCP to a *QIR* server. Along with the term, the server also receives the current value of all the variables which are bound in it, be they global or local. The server then normalizes (understand "optimizes") the term to make it easier for databases to handle, translates it into the language of the target database (SQL for instance), and queries that database. Once it gets the results, it finally sends them back to the Python client.

In order for the remote evaluation strategy to be efficient, we must choose the serialization format carefully: it must be fast enough to encode and decode, and be as lightweight as possible. For this reason, I chose to use Protocol Buffers, an open-source library from Google, which helps build binary serialization formats and provides encoders and decoders for all major programming languages, including Python and Java.

Conclusion

During the six weeks of this internship, I successfully implemented a Python front-end for the *BOLDR* project in the form of a 2000-line Python module. To do so, after choosing a query syntax, I had to explore the internals of CPython for a way to find information about closures at runtime, and come up with an algorithm to transform that information into terms of the *QIR* —the intermediate representation that the *BOLDR* team built to represent data queries. While this algorithm is far from perfect, it already covers most use cases, and provides a fallback (the Bytecode node) for functions outside of those use cases.

The *BOLDR* project and its integration to "mainstream" languages such as Python are already an improvement on existing language-integrated querying frameworks, but there is still room for improvement. For instance, the team is looking at ways of consistently dealing with side-effects on the database side; or at integrating a type system into the *QIR* to catch most errors in queries before they get sent to the database.

For further details about the current state of the *BOLDR* project, including a link to the implementation of the Python front-end presented in this report, please visit `https://www.lri.fr/~kn/boldr_en.html`.

References

[1] CPython bytecode documentation. `https://docs.python.org/3/library/dis.html`.

[2] Doctrine documentation. `http://docs.doctrine-project.org/en/latest/`.

[3] LINQ documentation. `https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries`.

[4] SQLAlchemy documentation. `http://docs.sqlalchemy.org/en/latest/`.

[5] V. Benzaken, G. Castagna, L. Daynès, and K. Nguyen. Deep Integration of Programming Languages in Databases with Truffle. 2017.

[6] V. Benzaken, G. Castagna, L. Daynès, and K. Nguyen. Language-integrated queries with user-defined functions: a BOLDR approach. 2017.

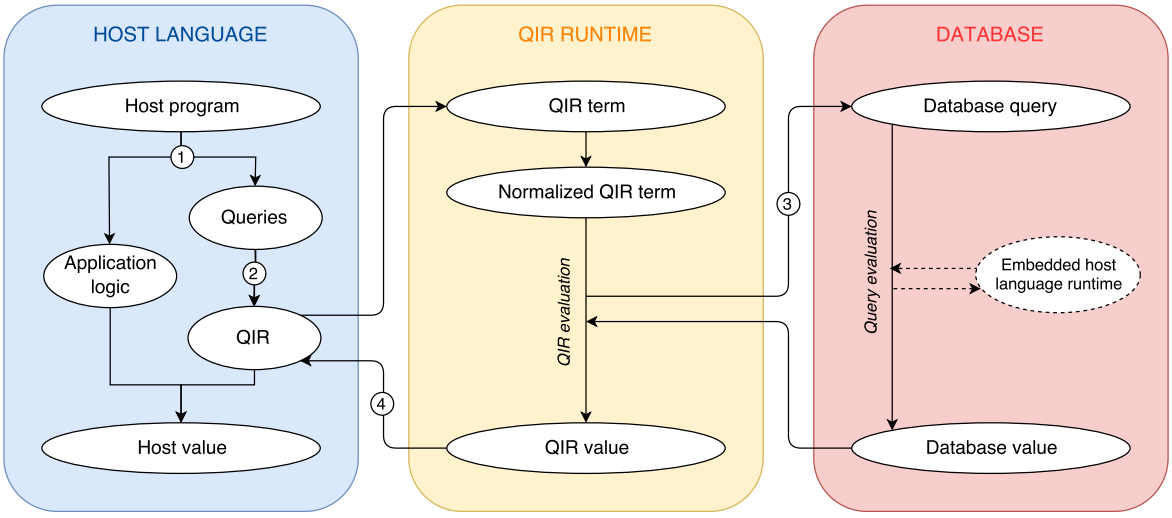
Appendices

Appendix A: About this internship

This internship took place within the VALS team of the LRI, in Orsay. I was mostly under the supervision of Kim Nguyen, but I also exchanged with other members of the *BOLDR* project —Giuseppe Castagna and Julien Lopez in particular.

I really enjoyed the experience, for several reasons. First of all, I already had an interest in language-integrated querying before this internship, so I was happy being able to contribute to a project which solves an issue that I believe many developers (me included) are faced with. I was also pleased with the overall working atmosphere, as I believe it had the right balance between having the freedom to explore my ideas and making sure that things went in the right direction.

Appendix B: Evaluation of a program in *BOLDR*



Appendix C: BNF definition of the Query Intermediate Language (or *QIR*)

$$
\begin{array}{rcl}
expr & ::= & const \mid id \\
 & \mid & func \mid list \\
 & \mid & tuple \mid op \\
 & \mid & alg \mid special \\
\\
const & ::= & \mathsf{Null}() \\
 & \mid & \mathsf{Number}(\texttt{int}) \\
 & \mid & \mathsf{Double}(\texttt{float}) \\
 & \mid & \mathsf{String}(\texttt{str}) \\
 & \mid & \mathsf{Boolean}(\texttt{bool}) \\
\\
id & ::= & \mathsf{Identifier}(\texttt{str}) \\
\\
func & ::= & \mathsf{Lambda}(id, expr) \\
 & \mid & \mathsf{Application}(expr, expr) \\
 & \mid & \mathsf{Conditional}(expr, expr, expr) \\
\\
list & ::= & \mathsf{ListNil}() \\
 & \mid & \mathsf{ListCons}(expr, expr) \\
 & \mid & \mathsf{ListDestr}(expr, expr, expr) \\
\\
tuple & ::= & \mathsf{TupleNil}() \\
 & \mid & \mathsf{TupleCons}(expr, expr, expr) \\
 & \mid & \mathsf{TupleDestr}(expr, expr)
\end{array}
\qquad
\begin{array}{rcl}
op & ::= & \mathsf{Scan}(expr) \\
 & \mid & \mathsf{Filter}(expr, expr) \\
 & \mid & \mathsf{Sort}(expr, expr, expr) \\
 & \mid & \mathsf{Limit}(expr, expr) \\
 & \mid & \mathsf{Group}(expr, expr) \\
 & \mid & \mathsf{Join}(expr, expr, expr) \\
\\
alg & ::= & \mathsf{Not}(expr) \\
 & \mid & \mathsf{Div}(expr, expr) \\
 & \mid & \mathsf{Minus}(expr, expr) \\
 & \mid & \mathsf{Mod}(expr, expr) \\
 & \mid & \mathsf{Plus}(expr, expr) \\
 & \mid & \mathsf{Star}(expr, expr) \\
 & \mid & \mathsf{Power}(expr, expr) \\
 & \mid & \mathsf{And}(expr, expr) \\
 & \mid & \mathsf{Or}(expr, expr) \\
 & \mid & \mathsf{Equal}(expr, expr) \\
 & \mid & \mathsf{LowerOrEqual}(expr, expr) \\
 & \mid & \mathsf{LowerThan}(expr, expr) \\
 & \mid & \mathsf{GreaterOrEqual}(expr, expr) \\
 & \mid & \mathsf{GreaterThan}(expr, expr) \\
\\
special & ::= & \mathsf{Builtin}(\texttt{str}) \\
 & \mid & \mathsf{Bytecode}(\texttt{bytes}) \\
 & \mid & \mathsf{Database}(\cdots) \\
 & \mid & \mathsf{Table}(\cdots)
\end{array}
$$

Appendix D: Semantic σ of the CPython stack machine

Stack-related operations.

$$
\begin{array}{rcll}
\sigma\langle n,\, S,\, E\rangle & = & \langle s(n),\, S,\, E\rangle & \text{if } o_n = \texttt{NOP} \\
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E\rangle & \text{if } o_n = \texttt{POP\_TOP} \\
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, s \cdot s \cdot S,\, E\rangle & \text{if } o_n = \texttt{DUP\_TOP} \\
\sigma\langle n,\, s_1 \cdot s_2 \cdot S,\, E\rangle & = & \langle s(n),\, s_2 \cdot s_1 \cdot S,\, E\rangle & \text{if } o_n = \texttt{ROT\_TWO} \\
\sigma\langle n,\, s_1 \cdot s_2 \cdot s_3 \cdot S,\, E\rangle & = & \langle s(n),\, s_2 \cdot s_3 \cdot s_1 \cdot S,\, E\rangle & \text{if } o_n = \texttt{ROT\_THREE} \\
\sigma\langle n,\, v_{a_n} \cdots v_1 \cdot f \cdot S,\, E\rangle & = & \langle s(n),\, f(v_1, \cdots, v_{a_n}) \cdot S,\, E\rangle & \text{if } o_n = \texttt{CALL\_FUNCTION} \\
\\
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, s[id] \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_ATTR},\, id) \\
\sigma\langle n,\, s_1 \cdot s_2 \cdot S,\, E\rangle & = & \langle s(n),\, s_1[id \mapsto s_2] \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_ATTR},\, id) \\
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, s_1[id \mapsto \texttt{null}] \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{DELETE\_ATTR},\, id) \\
\\
\sigma\langle n,\, s_1 \cdot s_2 \cdot S,\, E\rangle & = & \langle s(n),\, s_2[s_1] \cdot S,\, E\rangle & \text{if } o_n = \texttt{BINARY\_SUBSCR} \\
\sigma\langle n,\, s_1 \cdot s_2 \cdot s_3 \cdot S,\, E\rangle & = & \langle s(n),\, s_2[s_1 \mapsto s_3] \cdot S,\, E\rangle & \text{if } o_n = \texttt{STORE\_SUBSCR} \\
\sigma\langle n,\, s_1 \cdot s_2 \cdot S,\, E\rangle & = & \langle s(n),\, s_2[s_1 \mapsto \texttt{null}] \cdot S,\, E\rangle & \text{if } o_n = \texttt{DELETE\_SUBSCR}
\end{array}
$$

Environment-related operations.

$$
\begin{array}{rcll}
\sigma\langle n,\, S,\, E\rangle & = & \langle s(n),\, c \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_CONST},\, c) \\
 & & \langle s(n),\, E[id] \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_NAME},\, id) \\
 & & \langle s(n),\, E[id] \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_FAST},\, id) \\
 & & \langle s(n),\, E[id] \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_GLOBAL},\, id) \\
 & & & \\
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E[id \mapsto s]\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_FAST},\, id) \\
 & & \langle s(n),\, S,\, E[id \mapsto s]\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_GLOBAL},\, id) \\
 & & & \\
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E[id \mapsto \texttt{null}]\rangle & \text{if } \mathcal{C}_n = (\texttt{DELETE\_FAST},\, id) \\
 & & \langle s(n),\, S,\, E[id \mapsto \texttt{null}]\rangle & \text{if } \mathcal{C}_n = (\texttt{DELETE\_GLOBAL},\, id)
\end{array}
$$

Arithmetic operations.

$$
\begin{array}{rcll}
\sigma\langle n,\, s \cdot S,\, E\rangle & = & \langle s(n),\, \underline{i_n}(s) \cdot S,\, E\rangle & \text{if } o_n \in \mathcal{O}_{unary} \\
\sigma\langle n,\, s_1 \cdot s_2 \cdot S,\, E\rangle & = & \langle s(n),\, \underline{o_n}(s_2, s_1) \cdot S,\, E\rangle & \text{if } o_n \in \mathcal{O}_{binary} \\
 & & \langle s(n),\, \underline{o_n}(s_2, s_1) \cdot S,\, E\rangle & \text{if } o_n \in \mathcal{O}_{inplace} \\
 & & \langle s(n),\, \underline{a_n}(s_2, s_1) \cdot S,\, E\rangle & \text{if } o_n = \texttt{COMPARE\_OP}
\end{array}
$$

Control flow operations.

$$
\begin{array}{rcll}
\sigma\langle n,\, S,\, E\rangle & = & \langle n + delta,\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{JUMP\_FORWARD},\, delta) \\
\sigma\langle n,\, S,\, E\rangle & = & \langle target,\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{JUMP\_ABSOLUTE},\, target) \\
 & & & \\
\sigma\langle n,\, \texttt{false} \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{POP\_JUMP\_IF\_TRUE},\, target) \\
\sigma\langle n,\, \texttt{true} \cdot S,\, E\rangle & = & \langle target,\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{POP\_JUMP\_IF\_TRUE},\, target) \\
\sigma\langle n,\, \texttt{true} \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{POP\_JUMP\_IF\_FALSE},\, target) \\
\sigma\langle n,\, \texttt{false} \cdot S,\, E\rangle & = & \langle target,\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{POP\_JUMP\_IF\_FALSE},\, target) \\
 & & & \\
\sigma\langle n,\, \texttt{false} \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{JUMP\_IF\_TRUE\_OR\_POP},\, target) \\
\sigma\langle n,\, \texttt{true} \cdot S,\, E\rangle & = & \langle target,\, \texttt{true} \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{JUMP\_IF\_TRUE\_OR\_POP},\, target) \\
\sigma\langle n,\, \texttt{true} \cdot S,\, E\rangle & = & \langle s(n),\, S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{JUMP\_IF\_FALSE\_OR\_POP},\, target) \\
\sigma\langle n,\, \texttt{false} \cdot S,\, E\rangle & = & \langle target,\, \texttt{false} \cdot S,\, E\rangle & \text{if } \mathcal{C}_n = (\texttt{JUMP\_IF\_FALSE\_OR\_POP},\, target)
\end{array}
$$

Appendix E: Algorithm to build the control flow graph of bytecode $\mathcal{P}$

```
1 function BUILDGRAPH(P)
2     (V_P, E_P) ← (∅, ∅)
3     current_block ← ∅
4
5     ▷ We keep a mapping of line numbers to blocks.
6     mapping ← ∅
7
8     function CREATEBLOCK(type)
9         ▷ Add the previous block to the graph.
```

```
10        old_block ← current_block
11        V_P ← V_P ∪ old_block
12
13        ▷ Create a new block.
14        current_block ← ∅
15        block_type(current_block) ← type
16
17        ▷ Add an edge between the previous block and the new one.
18        link_edge ← (old_block, current_block)
19        edge_type(link_edge) ← NORMAL_FLOW
20        E_P ← E_P ∪ link_edge
21
22    for n ∈ L do
23        if i_n ∈ I_jump then
24            ▷ Add a block containing only the jump instruction.
25            OpenBlock(JUMP)
26            current_block ← current_block ∪ {n}
27            mapping[n] ← current_block
28
29            ▷ Open a block for the next instructions.
30            OpenBlock(LINEAR)
31
32        else if i_n ∈ I_branch then
33            ▷ Add a block containing only the branch instruction.
34            OpenBlock(BRANCH)
35            current_block ← current_block ∪ {n}
36            mapping[n] ← current_block
37
38            ▷ Open a block for the next instructions.
39            OpenBlock(LINEAR)
40
41        else if is_jump_target(n) then
42            ▷ We open a new block.
43            OpenBlock(LINEAR)
44            current_block ← current_block ∪ {n}
45            mapping[n] ← current_block
46
47        else
48            ▷ Simply append the instruction to the current block.
49            current_block ← current_block ∪ {n}
50            mapping[n] ← current_block
51
52
53    ▷ We add the remaining edges.
54    for block ∈ V_P do
55        if block_type(block) = JUMP then
56            ▷ Add an edge between the jump instruction and its target.
57            n ← head(block)
58            jump_edge ← (jump_block, mapping[a_n])
59            edge_type(jump_edge) ← NORMAL_FLOW
```

```
60              E_P ← E_P ∪ jump_edge
61
62          else if block_type(block) = BRANCH then
63              ▷ Add an edge between the branch instruction and the next instruction.
64              n ← head(block)
65              nojump_edge ← (nojump_block, mapping[s(n)])
66              edge_type(nojump_edge) ← NORMAL_FLOW
67              E_P ← E_P ∪ nojump_edge
68
69              ▷ Add an edge between the branch instruction and its target.
70              jump_edge ← (jump_block, mapping[a_n])
71              edge_type(jump_edge) ← JUMP_FLOW
72              E_P ← E_P ∪ jump_edge
73
74      return (V_P, E_P)
```

Appendix F: Algorithm to execute the symbolic stack machine on a `LINEAR`-labeled block

```
1  function EXECUTELINEAR(V_P, E_P, v, S_0)
2      n_0 ← min(v)
3      B_0 ← []
4      ⟨n_f, S_f, B_f⟩ ← σ'⟨n_0, S_0, B_0⟩
5
6      v.stack ← S_f
7      v.bindings ← B_f
```

Appendix G: Algorithm to execute `JUMP`- and `BRANCH`-labeled blocks

```
1  function EXECUTEJUMP(V_P, E_P, v, S_0)
2      ▷ We keep the stack untouched.
3      v.stack ← S_0
4      v.bindings ← []
5
6  function EXECUTEBRANCH(V_P, E_P, v, S_0)
7      ▷ We keep the stack untouched.
8      v.stack ← S_0
9      v.bindings ← []
```

Appendix H: Algorithm to express a `LINEAR`-labeled block

```
1  function ExpressLinear(V_P, E_P, v)
2      if last_instr = RETURN_VALUE then
3          v.expression ← head(v.stack)
4      else if {w ∈ V_P | (v, w) ∈ E_P} = ∅ then
5          v.expression ← Null()
6      else
7          ▷ We assign the term that was assigned to the block's only successor.
8          succ ← head({w ∈ V_P | (v, w) ∈ E_P})
9          v.expression ← succ.expression
10
11     for (name, value) ∈ rev(v.bindings) do
12         v.expression ← Application(Lambda(Identifier(name), v.expression), value)
```

Appendix I: Algorithm to express a `JUMP`-labeled block

```
1  function ExpressJump(V_P, E_P, v)
2      ▷ We assign the term that was assigned to the block's only successor.
3      succ ← head({w ∈ V_P | (v, w) ∈ E_P})
4      v.expression ← succ.expression
```

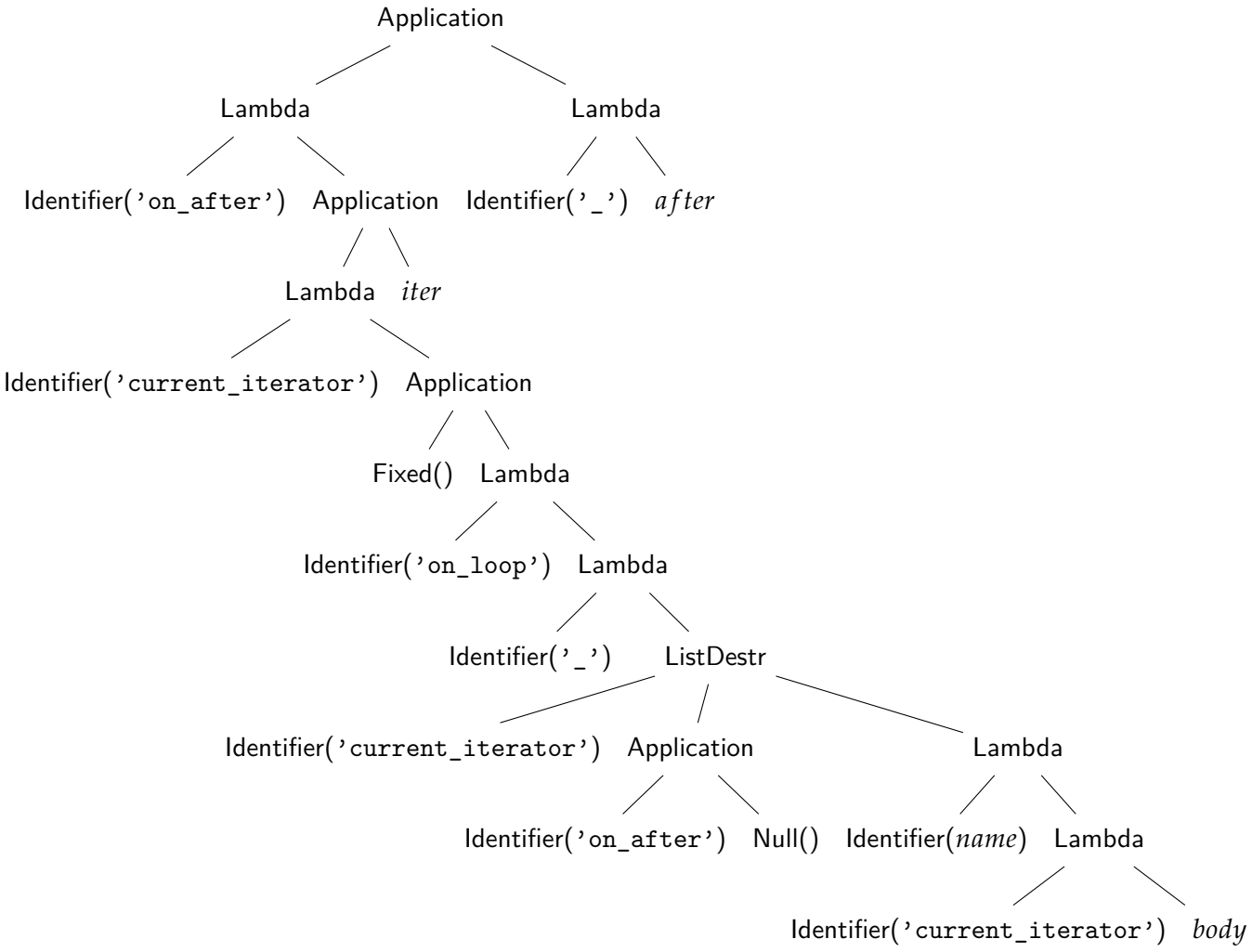
Appendix J: Algorithm to express a `BRANCH`-labeled block

```
1  function ExpressBranch(V_P, E_P, v)
2      n ← head(v)
3      normal_succ ← head({w ∈ V_P | e = (v, w) ∈ E_P ∧ edge_type(e) = NORMAL_FLOW})
4      jump_succ ← head({w ∈ V_P | e = (v, w) ∈ E_P ∧ edge_type(e) = JUMP_FLOW})
5
6      ▷ We get the condition from the top of the stack.
7      cond ← head(v.stack)
8
9      if o_n ∈ {POP_JUMP_IF_FALSE, JUMP_IF_FALSE_OR_POP} then
10         true ← normal_succ.expression
11         false ← jump_succ.expression
12
13     else if o_n ∈ {POP_JUMP_IF_TRUE, JUMP_IF_TRUE_OR_POP} then
14         true ← jump_succ.expression
15         false ← normal_succ.expression
16
17     v.expression ← Conditional(cond, true, false)
```

Appendix K: Algorithm to translate graph $(V_{\mathcal{P}}, E_{\mathcal{P}})$ into a *QIR* term

```
1  function TranslateGraph((V_P, E_P))
2      order ← ComputeOrdering(V_P, E_P)
3
4      ▷ Retrieves the stack shared by all predecessors.
5      function PredecessorsStack(v)
6          preds ← {u ∈ V_P | (u, v) ∈ E_P}
7          if ∃u, u' ∈ preds, u.stack ≠ u'.stack then
8              Raise(PredecessorStacksError)
9          else
10             return head(preds).stack
11
12     ▷ Execution pass in ascending order.
13     S_0 ← []
14     for i ∈ [[1, |V_P|]] do
15         v ← order^{-1}(i)
16         S_0 ← PredecessorsStack(v)
17
18         if block_type(v) = LINEAR then
19             ExecuteLinear(V_P, E_P, v, S_0)
20         else if block_type(v) = JUMP then
21             ExecuteJump(V_P, E_P, v, S_0)
22         else if block_type(v) = BRANCH then
23             ExecuteBranch(V_P, E_P, v, S_0)
24
25     ▷ Expression pass in descending order.
26     for i ∈ rev([[1, |V_P|]]) do
27         v ← order^{-1}(i)
28         if block_type(v) = LINEAR then
29             ExpressLinear(V_P, E_P, v)
30         else if block_type(v) = JUMP then
31             ExpressJump(V_P, E_P, v)
32         else if block_type(v) = BRANCH then
33             ExpressBranch(V_P, E_P, v)
34
35     return order^{-1}(1).expression
```

Appendix L: *QIR* translation of a `for` loop



Appendix M: The Python implementation of the bytecode to *QIR* translation

```
from . import *

import dis
import types
import random
import graphviz

from functools import reduce

NORMAL_FLOW = 0
JUMP_FLOW = 1

JUMP_OPNAMES = [
    'JUMP_FORWARD',
    'JUMP_ABSOLUTE',
    'CONTINUE_LOOP']

BRANCH_POP_OPNAMES = [
    'POP_JUMP_IF_TRUE',
    'POP_JUMP_IF_FALSE']

BRANCH_MAY_POP_OPNAMES = [
    'JUMP_IF_TRUE_OR_POP',
    'JUMP_IF_FALSE_OR_POP']

BRANCH_OPNAMES = \
    BRANCH_POP_OPNAMES + \
    BRANCH_MAY_POP_OPNAMES


class PredecessorStacksError(Exception):
    pass


class Decompiler():
    def __init__(self):
        self.blocks = []

        # In order to be able to compute the graph in linear time, we keep a
        # mapping of the block in which any instruction is contained.
        self.block_mapping = {}

        # Whether to use comprehension mode, which is designed to handle the
        # decompilation of list, set and map comprehensions. In this mode, a
        # FOR_ITER instruction will be turned into a ComprehensionLoopBlock
        # instead of a ForLoopBlock.
        self.comprehension_mode = False

    @property
    def first_block(self):
        return self.blocks[0]

    @property
    def previous_block(self):
        return self.blocks[-2]

    @property
    def current_block(self):
        return self.blocks[-1]

    def build_graph(self, instructions, ignore_loop=False):
        """
        Separate the instructions into blocks and build a control flow graph.

        instructions: The list of instructions to process.
```

```
ignore_loop: Whether to ignore a SETUP_LOOP or a FOR_ITER instruction
if it is the first instruction of the list. This is useful to avoid
infinite recursion when using a separate Decompiler instance to
process the inside of a LoopBlock.
"""

# The offset before which all instructions should be added to the
# current block as is, no matter their type.
ignore_until = False

# Whether to start a new block on the next instruction, even if it
# is not a jump target.
force_new = True

for i, instruction in enumerate(instructions):
    if instruction.offset < ignore_until:
        self.current_block.add(instruction)

    elif (instruction.opname == 'FOR_ITER' and
          (not ignore_loop or i > 0)):
        if self.comprehension_mode:
            block_type = ComprehensionLoopBlock
        else:
            block_type = ForLoopBlock

        self.blocks.append(block_type(self, instruction))
        ignore_until = instruction.argval
        force_new = True

    # We must determine whether the SETUP_LOOP instruction introduces
    # a for or a while loop in O(1), so what we can do is check
    # whether the next instruction is a jump target, in which case
    # either that instruction is FOR_ITER, and the loop is a for loop,
    # or the loop is a while loop.
    elif (instruction.opname == 'SETUP_LOOP' and
          instructions[i + 1].opname != 'FOR_ITER' and
          instructions[i + 1].is_jump_target and
          (not ignore_loop or i > 0)):
        self.blocks.append(WhileLoopBlock(self, instruction))
        ignore_until = instruction.argval
        force_new = True

    elif instruction.opname in JUMP_OPNAMES:
        self.blocks.append(JumpBlock(self, instruction))
        force_new = True

    elif instruction.opname in BRANCH_OPNAMES:
        self.blocks.append(BranchBlock(self, instruction))
        force_new = True

    elif instruction.opname == 'FOR_ITER':
        self.blocks.append(ForIterBlock(self, instruction))
        force_new = True

    else:
        if instruction.is_jump_target or force_new:
            force_new = False
            self.blocks.append(LinearBlock(self))

        self.current_block.add(instruction)

# Once all the blocks have been created - which also means that all
# the instructions were assigned to one and only one block - we can
# "close" the blocks. For instance, this allows JumpBlocks and
# BranchBlocks to translate the offset of the instruction they
# should jump to into the block which contains that instruction.
for block in self.blocks:
    block.close()

# Finally, we compute a reverse map of the direct predecessors of
```

```
        # every block so that execute_blocks can run in linear time.
        for block in self.blocks:
            for (successor, edge_type) in block.successors:
                successor.predecessors.append((block, edge_type))

    def sort_blocks(self):
        """
        Compute a topological ordering of the control flow graph.

        This is possible as we have removed cycles from the graph by hiding
        them into LoopBlocks, and so the graph is a DAG.
        """
        marked = [False] * len(self.blocks)
        ordering = []

        def visit(block):
            marked[block.index] = True

            for (next, _) in block.successors:
                if not marked[next.index]:
                    visit(next)

            ordering.append(block.index)

        visit(self.first_block)
        self.ordering = list(reversed(ordering))

    def detach_unreachable(self):
        """
        Detaches all the blocks which are not reachable from the first block.
        As the topological ordering that was computed earlier in sort_blocks
        only explores the first block's connected component, we can easily
        deduce which blocks should be removed.
        """
        for i in set(range(len(self.blocks))) - set(self.ordering):
            self.blocks[i].detach()

    def execute_blocks(self, starting_stack=[], starting_env={}):
        """
        Partially execute each block in topological ordering.
        """
        for index in self.ordering:
            self.blocks[index].execute(starting_stack, starting_env)

    def express_blocks(self):
        """
        Turn each block into a QIR expression in reversed topological ordering.
        """
        for index in reversed(self.ordering):
            self.blocks[index].express()


class Block():
    def __init__(self, context):
        self.context = context

        # The index of this block in the graph that is being built.
        self.index = len(context.blocks)

        # The instructions which belong to this block.
        self.instructions = []

        # The block to which we should jump once we reach the end of this one.
        self.next = None

        # The blocks from which we could come from.
        # This reverse map will be filled automatically by build_graph.
        self.predecessors = []

        # Whether the block contains a RETURN_VALUE instruction.
```

```
        self.contains_return = False

        # Whether the block contains a LIST_APPEND instruction.
        self.contains_append = False

        # The expression that the block returns, if any.
        self.returns = None

        # We add an edge between the previous block and this one.
        if self.index > 0:
            self.context.current_block.next = self

    @property
    def successors(self):
        if self.next is not None:
            return [(self.next, NORMAL_FLOW)]
        else:
            return []

    def add(self, instruction):
        if not self.contains_return:
            self.instructions.append(instruction)

        # Even though we only add the new instruction to the block if we have
        # not yet reached a RETURN_VALUE instruction, we still have to declare
        # that the block contains the new instruction, otherwise we might break
        # the translation from jump offsets to blocks that happens in
        # JumpBlock@close, BranchBlock@close and ForIterBlock@close.
        self.context.block_mapping[instruction.offset] = self

        if instruction.opname == 'RETURN_VALUE':
            self.contains_return = True

        if instruction.opname in ['LIST_APPEND', 'SET_ADD', 'MAP_ADD']:
            self.contains_append = True

    def close(self):
        # If the block contains a RETURN_VALUE instruction, we don't want to
        # jump to any other block at the end of this one.
        if self.contains_return:
            self.next = None

    def detach(self):
        """
        Detach the block from its neighbours and remove it from the graph.
        """
        for (successor, edge_type) in self.successors:
            successor.predecessors.remove((self, edge_type))

        for (predecessor, edge_type) in self.predecessors:
            if edge_type == NORMAL_FLOW:
                predecessor.next = None
            else:
                predecessor.next_jumped = None

        self.predecessors = []
        self.next = None
        self.next_jumped = None

    def execute(self, starting_stack=[], starting_env={}):
        """
        Ensure that all the block's direct predecessors share the same final
        stack state, and if so make it the initial state of the block's stack.

        The starting_stack is the stack which will be used if the block has no
        predecessors. This is used when dealing with loops, for instance, as
        LoopBlock execute the loop instructions inside a separate builder.
        """
        stacks = []
```

```
        for (predecessor, edge_type) in self.predecessors:
            # Getting the predecessor's final stack state is slightly tricky
            # when that predecessor is a BranchBlock, as the stack might be
            # popped right before the jump depending on the precise branching
            # instruction and the type of edge that links this block to the
            # predecessor.
            if isinstance(predecessor, (BranchBlock, ForIterBlock)):
                name = predecessor.instruction.opname
                if ((name == 'FOR_ITER' and
                         edge_type == NORMAL_FLOW) or
                    (name in BRANCH_MAY_POP_OPNAMES and
                         edge_type == JUMP_FLOW)):
                    stacks.append(predecessor.stack[:])
                else:
                    stacks.append(predecessor.stack[:-1])
            else:
                stacks.append(predecessor.stack[:])

        if len(stacks) < 1:
            self.stack = starting_stack[:]
        elif any(stacks[0] != stack for stack in stacks[1:]):
            # Just a little bit of debugging.
            for index in self.context.ordering:
                if index == self.index:
                    break

                block = self.context.blocks[index]
                print('[%d] %s' % (index, str(block.stack)))

            raise PredecessorStacksError((self.index, stacks))
        else:
            self.stack = stacks[0]


BINARY_OPERATIONS = {
    'BINARY_POWER': Power,
    'BINARY_MULTIPLY': Star,
    'BINARY_TRUE_DIVIDE': Div,
    'BINARY_MODULO': Mod,
    'BINARY_ADD': Plus,
    'BINARY_SUBTRACT': Minus}

INPLACE_OPERATIONS = {
    'INPLACE_POWER': Power,
    'INPLACE_MULTIPLY': Star,
    'INPLACE_TRUE_DIVIDE': Div,
    'INPLACE_MODULO': Mod,
    'INPLACE_ADD': Plus,
    'INPLACE_SUBTRACT': Minus}

COMPARE_OPERATIONS = {
    '==': Equal,
    '<=': LowerOrEqual,
    '<': LowerThan}

class LinearBlock(Block):
    def execute(self, starting_stack=[], starting_env={}):
        super().execute(starting_stack, starting_env)

        stack = self.stack
        bindings = []

        for instruction in self.instructions:
            name = instruction.opname

            # General instructions
            if name == 'NOP':
                pass

            elif name == 'POP_TOP':
```

```
            stack.pop()

        elif name == 'ROT_TWO':
            stack[-1], stack[-2] = stack[-2], stack[-1]

        elif name == 'ROT_THREE':
            stack[-1], stack[-2], stack[-3] =\
                stack[-2], stack[-3], stack[-1]

        elif name == 'DUP_TOP':
            stack.append(stack[-1])

        elif name == 'DUP_TOP_TWO':
            stack.append(stack[-2])
            stack.append(stack[-2])

        # Binary and in-place operations
        elif name in BINARY_OPERATIONS:
            right = stack.pop()
            left = stack.pop()
            stack.append(BINARY_OPERATIONS[name](left, right))

        elif name in INPLACE_OPERATIONS:
            right = stack.pop()
            left = stack.pop()
            stack.append(INPLACE_OPERATIONS[name](left, right))

        elif (name == 'COMPARE_OP' and
              instruction.argval in COMPARE_OPERATIONS):
            right = stack.pop()
            left = stack.pop()
            stack.append(
                COMPARE_OPERATIONS[instruction.argval](left, right))

        elif name == 'BINARY_SUBSCR':
            key = stack.pop()
            container = stack.pop()
            stack.append(TupleDestr(container, key))

        elif name == 'STORE_SUBSCR':
            key = stack.pop()
            container = stack.pop()
            value = stack.pop()
            stack.append(TupleCons(key, value, container))

        elif name == 'DELETE_SUBSCR':
            container = stack.pop()
            value = stack.pop()
            stack.append(TupleCons(key, Null(), container))

        # Miscellaneous opcodes
        elif name in ['RETURN_VALUE', 'YIELD_VALUE']:
            self.returns = stack.pop()

        elif name in ['LIST_APPEND', 'SET_ADD']:
            value = stack.pop()
            tail = stack[-1 * instruction.argval]
            stack[-1 * instruction.argval] = ListCons(value, tail)

        elif name == 'MAP_ADD':
            key = stack.pop()
            value = stack.pop()
            tail = stack[-1 * instruction.argval]
            stack[-1 * instruction.argval] = TupleCons(key, value, tail)

        elif name == 'POP_BLOCK':
            pass

        elif name == 'LOAD_CONST':
            stack.append(encode(instruction.argval))
```

```
elif (name == 'LOAD_NAME' or
      name == 'LOAD_GLOBAL' or
      name == 'LOAD_FAST' or
      name == 'LOAD_DEREF'):
    stack.append(Identifier(instruction.argval))

elif name == 'LOAD_CLOSURE':
    pass

elif name == 'LOAD_ATTR':
    container = stack.pop()
    stack.append(TupleDestr(container, String(instruction.argval)))

elif (name == 'STORE_NAME' or
      name == 'STORE_FAST'):
    value = stack.pop()
    bindings.append((instruction.argval, value))

elif name == 'STORE_GLOBAL':
    raise NotImplementedError

elif (name == 'DELETE_NAME' or
      name == 'DELETE_FAST'):
    bindings.append((instruction.argval, Null()))

elif name == 'DELETE_GLOBAL':
    raise NotImplementedError

elif name == 'CALL_FUNCTION':
    count = instruction.argval
    inner = stack[-(count + 1)]

    # Because the QIR functions are currified, we have to make as
    # many applications as there are arguments. The good news is,
    # as the right-most argument is on top of the stack, this is
    # all pretty straightforward.
    for i in range(count):
        inner = Application(inner, stack.pop())

    stack.pop()
    stack.append(inner)

elif (name == 'BUILD_TUPLE' or
      name == 'BUILD_LIST' or
      name == 'BUILD_SET'):
    pos = (-1) * instruction.argval
    values = stack[pos:]
    stack = stack[:pos]

    container = ListNil()
    for value in values:
        container = ListCons(value, container)

    stack.append(container)

elif name == 'BUILD_MAP':
    pos = (-1) * instruction.argval
    values = stack[pos:]
    stack = stack[:pos]

    container = TupleNil()
    for key, value in zip(values[0::2], values[1::2]):
        container = TupleCons(key, value, container)

    stack.append(container)

elif name == 'BUILD_STRING':
    pos = (-1) * instruction.argval
    values = stack[pos:]
```

```
                stack = stack[:pos]

                string = ''.join(map(lambda x: x.value, values))
                stack.append(String(string))

            elif name == 'MAKE_FUNCTION':
                if instruction.argval > 0:
                    raise errors.NotYetImplementedError

                stack.pop()

            elif name == 'MAKE_CLOSURE':
                if instruction.argval > 0:
                    raise errors.NotYetImplementedError

                stack.pop()
                stack.pop(-2)

            elif name == 'SETUP_LOOP':
                pass

            elif name == 'GET_ITER':
                pass

            # Other opcodes
            else:
                raise NotImplementedError(name)

        self.stack = stack
        self.bindings = bindings

    def express(self):
        if self.returns is not None:
            inner = self.returns
        elif self.next is None:
            inner = Null()
        else:
            inner = self.next.expression

        # Wrap the expression inside all the variable bindings.
        for (name, value) in reversed(self.bindings):
            inner = Application(Lambda(Identifier(name), inner), value)

        self.expression = inner


class JumpBlock(Block):
    def __init__(self, context, instruction):
        super().__init__(context)
        self.instruction = instruction
        self.add(instruction)

    def close(self):
        super().close()

        # Because we will always take the jump, we can replace the edge which
        # was created between this block and the one that follows it with a
        # new edge between this block and the one it should jump to.
        self.next = self.context.block_mapping[self.instruction.argval]

    def express(self):
        self.expression = self.next.expression


class BaseBranchBlock(Block):
    def __init__(self, context, instruction):
        super().__init__(context)
        self.instruction = instruction
        self.add(instruction)
```

```
    @property
    def successors(self):
            return [(self.next, NORMAL_FLOW), (self.next_jumped, JUMP_FLOW)]

    def close(self):
        super().close()

        # We keep self.next untouched - it will point to the block to go to if
        # we don't take the jump, and we add self.next_jumped which will point
        # to the block to go to if we take the jump.
        self.next_jumped = self.context.block_mapping[self.instruction.argval]


class BranchBlock(BaseBranchBlock):
    def express(self):
        condition = self.stack[-1]

        if self.instruction.opname in \
                ['POP_JUMP_IF_FALSE', 'JUMP_IF_FALSE_OR_POP']:
            on_true = self.next.expression
            on_false = self.next_jumped.expression

        elif self.instruction.opname in \
                ['POP_JUMP_IF_TRUE', 'JUMP_IF_TRUE_OR_POP']:
            on_true = self.next_jumped.expression
            on_false = self.next.expression

        self.expression = Conditional(condition, on_true, on_false)


class ForIterBlock(BaseBranchBlock):
    def execute(self, starting_stack=[], starting_env={}):
        super().execute(starting_stack, starting_env)

        # We must push a reference to the current value of the iterator on the
        # stack, so that blocks inside the loop's body can use it. We use the
        # instruction's offset as a way to avoid name clashes in nested loops.
        self.stack.append(Identifier('cv_' + str(self.instruction.offset)))

    def express(self):
        self.expression = self.next.expression


class LoopBlock(Block):
    def __init__(self, context, instruction):
        super().__init__(context)
        self.instruction = instruction
        self.add(instruction)

    def execute(self, starting_stack=[], starting_env={}):
        super().execute(starting_stack, starting_env)

        # We use a separate instance of the decompiler to process the code
        # inside the loop. We have to add a placeholder for the instruction
        # following the end of the loop.
        instructions = self.instructions + \
            [dis.Instruction('AFTER_LOOP', -1, None, None,
             None, self.instruction.argval, None, True)]

        # For some reason, breaks are translated into BREAK_LOOP instructions
        # instead of the standard JUMP_ABSOLUTE, so we must fix that manually.
        for i in range(len(instructions)):
            instr = instructions[i]

            if instr.opname == 'BREAK_LOOP':
                instructions[i] = dis.Instruction(
                    'JUMP_ABSOLUTE', 113, self.instruction.argval,
                    self.instruction.argval, None, instr.offset,
                    instr.starts_line, instr.is_jump_target)
```

```
        decompiler = Decompiler()
        decompiler.comprehension_mode = self.context.comprehension_mode
        decompiler.build_graph(instructions, True)

        start_block = decompiler.first_block
        last_block = decompiler.current_block

        decompiler.sort_blocks()
        decompiler.detach_unreachable()

        display_graph(decompiler)

        # We then identify the edges which jump back to the start_block, and
        # make them point to a placeholder block instead. This block, once
        # expressed, will turn into a call to on_loop.
        loop_placeholder = PlaceholderBlock(
            decompiler, Application(Identifier('on_loop'), Null()))

        decompiler.blocks.append(loop_placeholder)

        previous_predecessors = start_block.predecessors
        start_block.predecessors = []

        for (predecessor, edge_type) in previous_predecessors:
            if predecessor.index < start_block.index:
                start_block.predecessors.append((predecessor, edge_type))
            elif edge_type == JUMP_FLOW:
                predecessor.next_jumped = loop_placeholder
                loop_placeholder.predecessors.append(
                    (predecessor, JUMP_FLOW))
            else:
                predecessor.next = loop_placeholder
                loop_placeholder.predecessors.append(
                    (predecessor, NORMAL_FLOW))

        # We also replace all the references to the last block, which only
        # contains the AFTER_LOOP instruction that we added earlier, with a
        # placeholder block which will turn into a call to on_after.
        after_placeholder = PlaceholderBlock(
            decompiler, Application(Identifier('on_after'), Null()))

        after_placeholder.index = last_block.index
        decompiler.blocks[last_block.index] = after_placeholder

        for (predecessor, edge_type) in last_block.predecessors:
            if edge_type == JUMP_FLOW:
                predecessor.next_jumped = after_placeholder
                after_placeholder.predecessors.append(
                    (predecessor, JUMP_FLOW))
            else:
                predecessor.next = after_placeholder
                after_placeholder.predecessors.append(
                    (predecessor, NORMAL_FLOW))

        # This is not pretty, but we must remove the edge that is created
        # between a block and the one which follows it.
        loop_placeholder.next = None
        after_placeholder.next = None

        display_graph(decompiler)

        self.loop_placeholder, self.after_placeholder, self.decompiler =\
            loop_placeholder, after_placeholder, decompiler


class WhileLoopBlock(LoopBlock):
    def __init__(self, context, instruction):
        super().__init__(context, instruction)

        # We don't want SETUP_LOOP to appear in the instructions, as the first
```

```
        # block of the loop's body should be the jump target.
        if instruction.opname == 'SETUP_LOOP':
            del self.instructions[0]

    def execute(self, starting_stack=[], starting_env={}):
        super().execute(starting_stack, starting_env)
        self.decompiler.execute_blocks(self.stack[:])

    def express(self):
        self.decompiler.express_blocks()

        on_loop = self.decompiler.first_block.expression
        on_after = self.next.expression

        # Using the Y fixed-point combinator, we return a recursive function
        # which can either call itself again (to run the loop once more) or
        # call the rest of the code.
        self.expression = Application(
            Lambda(
                Identifier('on_after'),
                Application(
                    Fixed(),
                    Lambda(
                        Identifier('on_loop'),
                        Lambda(Identifier('_'), on_loop)))),
            Lambda(
                Identifier('_'),
                on_after))


class ForLoopBlock(LoopBlock):
    def execute(self, starting_stack=[], starting_env={}):
        super().execute(starting_stack, starting_env)
        self.decompiler.execute_blocks(self.stack[:])

        # We have to remove the iterator from the stack to reproduce what
        # FOR_ITER does once it jumps out of the loop, but we must also store
        # it somewhere in order to get it back in express().
        self.iterator = self.stack.pop()

    def express(self):
        self.decompiler.express_blocks()

        identifier = 'cv_' + str(self.instruction.offset)
        on_loop = self.decompiler.first_block.expression
        on_after = self.next.expression

        # We have to use the Y fixed-point combinator together with calls to
        # ListDestr. This isn't very pretty, but it's the only way that makes
        # it possible to change variable bindings between iterations - which
        # is not possible with Project().
        self.expression = Application(
            Lambda(
                Identifier('on_after'),
                Application(
                    Lambda(
                        Identifier('current_iterator'),
                        Application(
                            Fixed(),
                            Lambda(
                                Identifier('on_loop'),
                                Lambda(
                                    Identifier('_'),
                                    ListDestr(
                                        Identifier('current_iterator'),
                                        Application(
                                            Identifier('on_after'),
                                            Null()
                                        ),
                                        Lambda(
```

```
                                        Identifier(identifier),
                                        Lambda(
                                            Identifier('current_iterator'),
                                            on_loop
                                        )
                                    )
                                )
                            ))),
                    self.iterator)),
            Lambda(
                Identifier('_'),
                on_after))


class ComprehensionLoopBlock(LoopBlock):
    def execute(self, starting_stack=[], starting_env={}):
        super().execute(starting_stack, starting_env)

        identifier = 'cv_' + str(self.instruction.offset)

        # Even though the graph might contain multiple paths leading to the
        # loop_placeholder block, only one of them contains the LIST_APPEND
        # (or SET_ADD or MAP_ADD) instruction.
        def find_append_path(start, already_found):
            """
            Find the path in the graph which contains the LIST_APPEND, SET_ADD
            or MAP_ADD instruction.

            start: The block from which to start the path.
            already_found: Whether the instruction was already found earlier.
            """
            if start.index == 0:
                return [start] if already_found else None

            if start.contains_append:
                already_found = True

            for (predecessor, edge_type) in start.predecessors:
                path = find_append_path(predecessor, already_found)

                if path is not None:
                    return path + [start]

            return None

        # We keep an environment with the current bindings of all variables so
        # that we can try to substitute identifiers with their values in the
        # terms we encounter along the path.
        environment = starting_env.copy()

        # We also build a conjunction of QIR expressions which must evaluate
        # to True in order for the path to be taken.
        append_when = []

        path = find_append_path(self.loop_placeholder, False)

        for i in range(len(path)):
            block = path[i]
            block.execute(self.stack[:], environment)

            if isinstance(block, LinearBlock):
                for (key, value) in block.bindings:
                    environment[key] = value

            elif isinstance(block, BranchBlock):
                condition = substitute(block.stack[-1], environment)
                is_normal_flow = (path[i + 1].index == block.next.index)

                if ((is_normal_flow and block.instruction.opname in
                     ['POP_JUMP_IF_TRUE', 'JUMP_IF_TRUE_OR_POP']) or
```

```
                    (not is_normal_flow and block.instruction.opname in
                     ['POP_JUMP_IF_FALSE', 'JUMP_IF_FALSE_OR_POP'])):
                        condition = Not(condition)

                append_when.append(condition)

        # We have to remove the iterator from the stack to reproduce what
        # FOR_ITER does once it jumps out of the loop, but we must also store
        # it somewhere in order to use it later.
        iterator = self.stack.pop()

        # If the conjunction is not empty, we must filter the iterator to keep
        # only the elements for which the path will be taken.
        if len(append_when) > 0:
            condition = reduce(lambda a, b: And(a, b), append_when)
            iterator = Filter(
                Lambda(Identifier(identifier), condition), iterator)

        # The stack of the last block in the path (excluding loop_placeholder)
        # should now contain a ListCons(head, tail), where head is what gets
        # appened to the list in the comprehension's body; or, in the case of
        # nested loops, a Project() expression.
        source_list = next(item for item in path[-2].stack
                           if isinstance(item, (ListCons, Project)))

        if isinstance(source_list, ListCons):
            source_expression = substitute(source_list.head, environment)
        else:
            source_expression = source_list

        # The trick, now, is just to replace that list with a Project().
        self.stack[-1] = Project(
            Lambda(Identifier(identifier), source_expression), iterator)

    def express(self):
        # We don't want to turn our loop comprehensions into expressions, but
        # rather replace the list on top of the stack, which was supposed to
        # be filled during the iterations of the loop, with a Project(...).
        if self.next is None:
            self.expression = Null()
        else:
            self.expression = self.next.expression


class PlaceholderBlock(Block):
    def __init__(self, context, expression):
        super().__init__(context)
        self.expression = expression

    def execute(self, starting_stack=[], starting_env={}):
        # We don't want to call super().execute() because that might raise a
        # PredecessorStacksError, as we use the same PlaceholderBlock in
        # different branches of While and For loops for instance.
        pass

    def express(self):
        pass


def decompile(code):
    decompiler = Decompiler()
    decompiler.comprehension_mode =\
        code.co_name in ['<listcomp>', '<setcomp>', '<dictcomp>', '<genexpr>']

    decompiler.build_graph(list(dis.get_instructions(code)))
    decompiler.sort_blocks()
    decompiler.detach_unreachable()
    decompiler.execute_blocks()
    decompiler.express_blocks()
```

```
inner = decompiler.first_block.expression

# Wrap the inner expression around a Lambda function with the right
# argument names. This works because, apparently, the names of the
# arguments always come before the names of the local variables in
# code.co_varnames.
for name in reversed(code.co_varnames[:code.co_argcount]):
    inner = Lambda(Identifier(name), inner)

return inner
```