HiPy: Extracting High-Level Semantics from Python Code for Data Processing

MICHAEL JUNGMAIR, Technical University of Munich, Germany
ALEXIS ENGELKE, Technical University of Munich, Germany
JANA GICEVA, Technical University of Munich, Germany

Data science workloads frequently include Python code, but Python's dynamic nature makes efficient execution hard. Traditional approaches either treat Python as a black box, missing out on optimization potential, or are limited to a narrow domain. However, a deep and efficient integration of user-defined Python code into data processing systems requires extracting the semantics of the entire Python code.

In this paper, we propose a novel approach for extracting the high-level semantics by transforming general Python functions into program generators that generate a statically-typed IR when executed. The extracted IR then allows for high-level, domain-specific optimizations and the generation of efficient C++ code. With our prototype implementation, HiPy, we achieve single-threaded speedups of 2–20x for many workloads. Furthermore, HiPy is also capable of accelerating Python code in other domains like numerical data, where it can sometimes even outperform specialized compilers.

CCS Concepts: • **Software and its engineering** → *Translator writing systems and compiler generators*; **Runtime environments**; • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: Python, High-Level Optimizations, Data Processing, Program Generation

ACM Reference Format:
Michael Jungmair, Alexis Engelke, and Jana Giceva. 2024. HiPy: Extracting High-Level Semantics from Python Code for Data Processing. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 297 (October 2024), 27 pages. https://doi.org/10.1145/3689737

1 Introduction

Data-processing pipelines frequently contain user-defined code written in Python, the lingua franca of data analysts and data engineers. However, there is a high impedance mismatch between high-performance data processing systems and the dynamic nature of Python. Data processing systems such as relational databases or distributed dataflow systems like Spark [39] implement a set of *native operators* with clearly defined semantics that allow for high-level logical optimizations and efficient execution. However, as expressing complex computations in such a representation (e.g., SQL) is often non-trivial, data scientists turned to Python, whose dynamically-typed nature makes it easy to express complex algorithms and stack together functionality from a huge ecosystem of high-quality packages. Figure 1a shows an example of such a user-defined function written in Python using the popular pandas dataframe library [20] and Figure 1b depicts an example SQL query using that UDF. However, the same properties that make Python attractive to users also make it very difficult to analyze, optimize, and execute efficiently.

Authors' Contact Information: Michael Jungmair, Technical University of Munich, Munich, Germany, jungmair@in.tum.de; Alexis Engelke, Technical University of Munich, Munich, Germany, engelke@in.tum.de; Jana Giceva, Technical University of Munich, Munich, Germany, jana.giceva@in.tum.de.

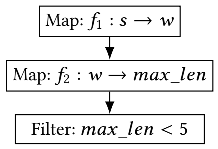
ACM 2475-1421/2024/10-ART297
https://doi.org/10.1145/3689737

```python
def udf(table)
  df=table.to_pandas()
  df["w"]=df['s'].str.split(" ")
  df["max_len"]= df["w"].apply(
  lambda words:
    max([len(p) for p in words])
  )
  df=df[df["max_len"]<5]
  return to_db(df)
```

(a) Tabular UDF using pandas

```sql
select s from udf(select * from ...)
```

(b) SQL Query



Map: $f_1 : s \rightarrow w$
Map: $f_2 : w \rightarrow max\_len$
Filter: $max\_len < 5$

```
def f1(s : str):
    return "string.split"(s)
def f2(w : list[str]):
    max_len : i64 = "const" 0
    "list.iter" p : s in w:
     l : i64 = "string.len"(w)
     max_len = "int.max"(max_len,l)
```

(c) Semantics in pseudo-IR

```cpp
bool predicate(string s) {
 size_t max_len = 0, pos = 0;
 while (pos != string::npos) {
  auto next = s.find(" ", pos);
  if (next == string::npos)
   break;
  auto w = s.substr(pos, next-pos);
  max_len = max(w.size(), max_len);
  pos = nextPos + 1;
 }
 return max_len < 5;
}
```

(d) Generated C++ predicate derived from the UDF

Fig. 1. Extracting the high-level semantics from an UDF allows for effective optimization and compilation.

Traditional, naïve integrations of user-defined Python code into data processing systems come with major disadvantages. Having to treat Python code as a black box prevents both global logical optimizations (e.g., query optimization, parallelization) and local optimizations of the UDF in the context of the entire pipeline (e.g., constant folding). Furthermore, embedding the standard Python implementation is substantially slower than optimized implementations due to calling and interpretation overhead. Most existing research focuses on the latter problem, either improving Python's speed [4, 18, 26, 29, 32] or the integration [10, 12, 30], but these approaches do not help with the former problem due to the persisting lack of insights into the user-defined Python programs. The missing piece to address the open problem is an approach that extracts the entire high-level semantics of Python programs in a way that it is easily digestible by data processing systems. The extracted program representation should not be at a lower abstraction level than the Python code itself (e.g., not LLVM IR), should reduce unnecessary abstractions (e.g., specific libraries), and should be statically typed.

The closest prior works propose and apply different techniques for supporting embedded DSLs in Python, typically using tracing [9, 13, 31, 37]. While these approaches also aim to extract *some semantics* from Python programs, they only aim to capture DSL-related operations and, by design, replace the semantics of some language constructs with their own. In contrast, we aim to extract the entire high-level semantic from general Python programs.

In this paper, we propose a novel approach for effectively extracting the entire high-level semantics from Python code in the form of a high-level, statically-typed IR. We first generate a program generator from the input code and then run this generator using Python to produce the IR. This way, we avoid the limitations of existing approaches and capture the whole program while enjoying the benefits of using the Python interpreter itself to deal with Python's complex and dynamic semantics. The captured IR then allows for performing global and logical optimizations.

For the initial example, Figure 1c sketches the extracted high-level semantic. Operators such as map and filter are extracted from pandas operations on data frames and two simple, statically typed functions f_1 and f_2 are extracted from the callback functions. Global knowledge that columns `w` and `max_len` are not used later allows fusing these operators into a single filter, enabling us to apply logical optimizations (e.g., pushing the filter down the pipeline). Finally, we generate efficient C++ code for the filter predicate as shown in Figure 1d.

We fully implement the proposed approach in HiPy, an end-to-end prototype system primarily targeting UDFs and medium-sized data-engineering scripts. HiPy generates an SSA-based IR, performs data-processing-related optimizations, and emits efficient C++ code. On data science

Table 1. Overview of related work, classified by their primary goal.

Class	Approach	Primary Domain	Examples	Comment
General Compiler		General Purpose	PyPy [32], Pyston [29], GraalPy [26]	
		General Purpose	Cython [4], Nuitka[25]	Conceptually: unrolling interpreter loop
Specialized Compiler	AST	Restricted	Codon [35]	Slightly different semantics and syntax
	Bytecode	Numerical	Numba [18]	Compiles numpy/numerical ops with LLVM
	AST	Numerical	Hope [3]	Compiles numerical operations to C++
	AST	Data Science	Tuplex [38]	Focus on UDFs, converts AST to LLVM
Support Embedded DSLs	AST	ML	TorchScript (JIT) [1]	Narrow focus on tensor operations
	Tracing	ML	torch.fx [31]	
	Tracing	Data Science	AFrame [36], Grizzly [13], PolyFrame [37], weld [27]	
	Virt.+Tracing	ML	Autograph [22]	Only captures computations on tensors
Extract High-Level Semantics	Generate Program Generator	Data Science	HiPy	Our approach

workloads, HiPy achieves single-threaded speedups between 1.8x and 18x over CPython and pandas. While primarily designed for data science workloads, HiPy also supports complex Python code in other domains due to a novel, fine-granular mechanism to transparently fall back to an embedded Python interpreter. Therefore, HiPy also shows good performance for other workloads, sometimes even outperforming specialized compilers like Numba or Codon.

This paper makes the following main contributions:

- A novel methodology for extracting the high-level semantics in the form of a high-level statically-typed IR tailored for Python by generating a program generator from input code.
- A novel approach for an automatic, fine-grained fallback to the Python interpreter, enabling to support arbitrary Python functions without a substantial performance penalty.
- An end-to-end prototype system, HiPy, that has built-in support for relevant parts of Python's standard library, numpy, pandas, and scikit-learn, performs general and domain-specific optimizations, and generates efficient C++ code.

In the remainder of this paper, we first revisit related work in section 2 and discuss why prior work does not already solve the highlighted problem. Next, we explain the core methodology of our approach in section 3 and describe the Python implementation in section 4. In section 5, we explain how HiPy can be extended to support other libraries and discuss our implementation of Python's standard library, numpy, pandas, and scikit-learn. In section 6, we describe our end-to-end system and evaluate it in section 7. Finally, we discuss different aspects like limitations, developer experience, and security in section 8 before summarizing our findings.

2 Related Work

The gap between a large user-base of Python, especially in the area of data analytics, data engineering, and machine learning, and the underwhelming performance compared to highly tuned frameworks is well known. Likewise, extracting high-level semantics from imperative programs to enable performance optimization and increase portability is a long-standing problem.

One way to approach the problem is program synthesis, which has not only been done for numerical workloads [2, 8, 17], but was more recently also applied to Python. For example, SOAR [24] translates between the APIs of different data frame and deep learning libraries. Nonetheless, the slow performance of the synthesis process and, in the more general case, the difficulty to determine whether the found solution is correct makes program synthesis impractical for many use cases.

Because Python's flexibility makes analysis and optimizations difficult, different languages like Julia [5] or, more recently, Mojo [21] have been proposed. However, such approaches have two substantial downsides: first, they come with a new language having different syntax and semantics, which increases the barrier of adoption. And second, they provide at most some degree of interfacing to call existing Python code, but do not help with extracting high-level semantics of existing Python packages, which are widely used in practice.

Additionally, a large variety of research has been conducted around improving Python's performance, both in the compiler community and in neighboring communities. We can classify existing approaches by their main goal, as summarized in Table 1: the first class of systems are general-purpose Python accelerators; the second class target direct compilation by restricting themselves to a subset of Python; and the third class are systems that support embedded DSLS through different techniques. In the remainder of this section, we cover these three classes in more detail.

2.1 Alternative Python Implementations

The relatively low performance of Python is a general problem, not only relevant to certain domains of data-intensive computing, but also for general workloads, e.g., web servers. Thus, different alternative implementations for Python try to reach a high degree of compatibility with CPython, while improving the observable performance. Typical implementations that apply JIT compilation such as PyPy [32], Pyston [29], or GraalPy [26] can speed up Python code by orders of magnitude for selected workloads, while making smaller compromises with compatibility. Other implementations like Cython [4] or nuitka [25] transform Python code to C code calling into the Python interpreter ahead of time. Already this unrolling of the interpreter loop can bring noticeable improvements with full compatibility. However, while such implementations can speed up Python code in data processing pipelines [10, 12], they do not help with extracting the high-level semantics to enable high-level, logical optimizations.

2.2 Specialized Python Compilers

In addition to the general-purpose Python implementations, various implementations specialize for certain use-cases and workloads and do not aim for full compatibility. Most focus is on compiling numerical functions to efficient machine code and parallelized execution of loops. Numba [18] takes the generated Python bytecode, lifts it into the statically typed Numba IR, and then generates efficient LLVM IR from it. In contrast, Hope [3] takes the abstract syntax tree, performs small optimizations and translates it to C++ code. Similarly, Tuplex [38] directly generates the LLVM IR from the AST of user-defined functions used in a Spark-like API. It supports a small set of selected built-in functions and methods that are hard-coded in the code generation module.

Codon [35] aims to enable the efficient execution of domain-specific DSLs (e.g., bioinformatics) expressed in Python syntax. It also restricts the set of supported features and built-in functionality and does not aim for maintaining the same semantics. It uses a so-called bi-directional IR to dynamically refine typed computations during later passes.

While these specialized Python compilers significantly improve the performance for their use cases, they lack generality and are not designed to extract high-level semantics, as they typically operate on an abstraction layer just above LLVM.

2.3 Supporting Embedded DSLs

Many prior works apply different techniques to support DSLs embedded into Python, with the goal of capturing DSL-related operations, often representing custom semantics. Typically, the captured operations are then executed by a domain-specific, optimized execution engine. The most common cases are libraries that generate a declarative query (e.g., in SQL) from usages of a data frame library [13, 37] and machine learning frameworks generating computation graphs from Python code [1, 22, 31]. Despite the different domains, prior work is designed to only capture DSL-related operations while either failing or eagerly executing non-supported operations and functions. All implementations in this category typically use one of three approaches: AST compilation, tracing, or virtualization.

2.3.1 *AST Compilation.*

Some implementations directly generate an intermediate from the abstract syntax tree of a Python function. This is easy to implement (e.g., using Python's `ast` module) and can also cover control flow constructs. However, the dynamic nature of Python makes it difficult to statically know the types of variables and thus the semantics, which depend on the types. Hence, this approach is only useful for limited subsets of Python or selected domains. For example, PyTorch [1] implements this approach in `torch.script.jit`. However, it does not support full Python, but TorchScript, a subset of Python with a slightly different semantics. Due to the discussed limitations, without explicit type annotations, every type is assumed to be a tensor. When calling other Python functions from a function compiled by PyTorch, PyTorch attempts to recursively compile those functions, too. This will often fail for functions not designed to be compiled by PyTorch, due to limitations in the supported subset of Python. For user-defined functions, this can be avoided by manually adding an `@torch.jit.ignore` annotation; PyTorch will not compile such functions and instead call them at execution time using the Python interpreter. However, functions from other libraries are therefore impossible to use, because users cannot annotate them.

2.3.2 *Tracing.*

The idea of tracing is to supply a function with instances of *tracing classes*, which do not eagerly execute operations, but instead track operations executed on them and construct some form of IR, e.g., a computation graph or relational algebra tree. Tracing has two main advantages: First, it is fairly simple to implement without requiring static analysis or type inference, thereby avoiding the limitations of direct AST compilation and does not require deeper knowledge in compilers. And second, if implemented carefully, tracing-based approaches can be made fully compatible to the corresponding library (e.g., pandas), by materializing and converting intermediate results once an unsupported operation is executed. Consequently, tracing is frequently used [13, 27, 31, 36, 37], especially to turn eager operations of an existing API into an intermediate representation that is later executed with a different, typically faster execution back-end.

However, in practice there are also many limitations. (1) Implementing a replacement library with *strictly identical* semantics is typically a substantial effort and thus, most research work only implements *similar* APIs, which leads to low adoption in practice. (2) While tracing manages to extract high-level semantics, it only does so for a local subset of the entire program, i.e. only capturing the parts that were implemented by the traced libraries. This rules out optimizations such as hoisting independent operations out of loops and also makes it hard to efficiently execute DAG-structured computations, because of redundant materializations. (3) Libraries like Pandas often include methods that take arbitrary call-back functions written in Python (e.g., `pandas.DataFrame.apply`), which cannot be covered by tracing. (4) Executing parts of the code at an earlier stage assumes that Python objects constructed during the tracing phase are still accessible when the generated IR is executed (cross-stage persistence). However, in data processing systems, this assumption usually does not hold when integrating user-defined Python code, for example due to distributed execution. (5) By design, tracing is unable to capture control flow constructs like if-statements or loops.

In summary, tracing-based approaches only allow for extracting narrow, DSL-related parts, but cannot extract the high-level semantics of an entire Python program, which is needed for a for a deep and efficient integration into existing systems.

2.3.3 Virtualization. Virtualization [7] extends tracing-based approaches to also support native control flow syntax by rewriting such syntax constructs in a prior step. This allows embedded DSLs to overload language constructs with custom semantics deviating from Python's semantics. For example, TensorFlow's Autograph [22] first rewrites Python code and replaces control flow constructs like (e.g., `if`, `while`) with function calls (e.g., `tf.cond`), which, in the second step, are captured during tracing. However, other limitations of tracing still remain, as only computations on tracing objects are captured.

3 Core Methodology

In this paper, we propose a generic methodology for extracting the entire high-level semantics from general Python functions without deviating from Python's syntax and semantics. The goal of this extraction is to generate a suitable intermediate representation (IR) that allows for high-level, domain-specific optimizations and, afterward, compiling this IR into efficient machine code. This is a major difference compared to the goals of prior work, which either captured a lower-level representation for compilation or focused only on capturing DSL-related operations.

To be suitable for a large number of use cases, the proposed approach should be able to support full Python, not only a narrow subset, capturing the semantics as good as possible. In order to reach these goals, we make the following key design decisions for our approach:

- **Explicit annotations.** The semantics are only captured for functions that are explicitly annotated with a Python decorator. Writes to global variables and modules are out of scope, as they are uncommon for data-intensive workloads. We assume that the entry function (e.g., the one registered as UDF to a data-processing framework) as well as its argument types are known at IR generation time.
- **Simple, high-level, statically-typed IR.** The generated IR should balance simplicity to be easily adoptable by domain-specific back-ends, and remain high-level enough to enable high-level, domain-specific optimizations. Thus, the extraction process should perform the heavy lifting of translating complex Python code into simple IR operations, while still using high-level types like multi-dimensional arrays and tables.
- **Modular implementation.** Following Python's language design, the approach should be implemented in a modular way, keeping the implementation of the core logic small. This also eases extending support to other modules and libraries by writing idiomatic Python code.
- **Fallback to Python.** Python is a highly dynamic language with many complex features and libraries, making it non-trivial to derive a simple, statically-typed intermediate representation. Additionally, Python and its ecosystem are so large that supporting the extraction of semantics is infeasible for all of it. The proposed approach should automatically detect problems and unsupported modules during the extraction process, and generate special IR operations that fall back to the Python interpreter at run-time. This is much more fine-grained and transparent in contrast to, e.g., PyTorch's approach of falling back to Python at function granularity and only for explicitly user-annotated functions.
- **Carefully handling mutable objects.** Python is not only highly dynamic, but most objects are mutable, increasing the difficulty of generating an intermediate representation with the same semantics. Our approach introduces extra measures for correctly handling mutable objects and especially avoids inconsistencies that quickly occur with a naïve implementation.

Our approach is based on *generating program generators*: we first generate a program generator from the Python code and then execute this program generator to generate the high-level intermediate representation. In the first step, a program generator is generated by automatically rewriting the abstract syntax tree, similar to how Python code is rewritten for virtualization. Afterward, the

```
def pow(b: int, e: int):                          def pow_spec(b: int):
    if e == 0:                                        return b * b
        return 1
    else:
        return b * pow(b, e-1)
```

Fig. 2. Partial evaluation of pow with e=2.

generated program generator is executed using Python, which allows us to leverage the Python interpreter to generate correct IR for complex Python semantics. Thereby, we avoid the limitations of tracing and virtualization, which can only capture operations on traced objects, while enjoying the advantages of it by using Python for executing the program generator.

In the following section, we first derive our conceptual approach from prior work around staging and partial evaluation in subsection 3.1. Afterward, in subsection 3.2, we discuss the design of the emitted IR that allows for capturing both opaque Python types and operations, but also high-level types and operations, before we discuss the implementation in section 4.

3.1 Adapting the First Futamura Projection

Staging and Partial Evaluation. For a function f and inputs i_S, i_D, where i_S is known at an earlier *stage* before i_D, we can apply staging [15]: By splitting f into f_1 and f_2 such that $f(i_S, i_D) = f_2(f_1(i_S), i_D)$, $f_1(i_S)$ can be executed at an earlier stage than f_2, e.g. during compilation.

Partial evaluation is a special case of staged execution: in the earlier stage, the function *mix*[1] specializes a general program P on the statically known input values i_S to the new function P_{spec}, which computes the result with only the dynamic input values i_D. A common example in literature is specializing the power function as shown in Figure 2.

$$P_{spec} := mix(P, i_S) \qquad \Rightarrow \qquad P(i_S, i_D) = P_{spec}(i_D)$$

The first Futamura Projection. Futamura discovered that applying the *mix* function to an interpreter *int* for a programming language and one program P produces a compiled Program P_C that computes the same without interpretation overhead [11].

$$P_C := mix(int, P) \qquad \Rightarrow \qquad int(P, args) = P_C(args)$$

So in principle, generating a semantically equivalent IR from a Python program is equivalent to computing the first Futamura projection.

Python is different. The literature around staging, partial evaluation, and the Futamura projections typically discuss the application for statically-typed, functional programming languages. In contrast, Python is dynamically typed and programs have a large amount of state, both explicit and implicit. This leads to problems when we try to apply the first Futamura projection, as the semantics of a Python function f are not fully specified by f itself: First, the arguments have a significant impact, because most of Python's semantics are defined in the methods of the argument objects. For example, in Python, the semantics of `a+b` is defined by special `__add__` methods. Because Python is also dynamically typed, we do not know which method defines the semantics for a + without the actual input arguments, which are not available at this early stage. And second, the environment defines available modules and functions, which can also override built-in functions, e.g. a local function `print` overriding the built-in function. On the top of that, writing the *mix* function for Python is highly non-trivial because of the complexity and also side-effects of operations.

[1] *mix* is the historically used name.

```
class int^S:
    def __add__(self, other):
        # built-in method
    ...

def addint(x: int, y: int):
    return x + y
```

(a) Example input, we specialize on the parameters being of type `int`.

```
def addint(%0, %1):
    %2 = int.add %0, %1
    return %2
```

(b) Generated IR from $\texttt{addint}_{pgen}$, wrapped with signature and return.

```
class int^S_pgen: # virtual class for int^S
    def __init__(self):
        self._ir = generate_new_ir_name()
    def __add___pgen(self, other):
        if not isinstance(other, int^S_pgen):
            raise NotImplementedError()
        res = int^S_pgen() # create new virtual object
        print(f"{res._ir} = int.add {self._ir}, {other._ir}")
        return res
    ...

def addint_pgen(x: int^S_pgen, y: int^S_pgen):
    # Generate IR for addition; returns the IR value of result
    return x.__add___pgen(y)
```

(c) Result of applying *cogen* to the input. The *virtual class* $\texttt{int}^S$ does not operate on integer values, but instead generates IR.

Fig. 3. Conceptual approach: applying *cogen* to the input yields an IR generator. The class `int` is conceptually split into int^S and int^D, where the latter only contains the concrete integer value.

Adapting the Futamura projection for Python. We propose to adapt the concept of the first Futamura projection to Python's properties. In particular, we propose to:

(1) Conceptually split every Python object O into a static part O^S, containing methods and static attributes, and a dynamic part O^D with the attributes only known at run-time. Consequently, functions would now consume two sets of arguments, the static and the dynamic parts. This allows for applying partial evaluation on the static parts of the arguments, yielding a specialized function:

$$f_{spec} := mix(Python, f, env, args^S) \qquad \Rightarrow \qquad Python(env, f, args) = f_{spec}(args^D)$$

(2) Follow the idea of *generating program generators* [14] and avoid the monolithic, hard to implement partial evaluator *mix* by utilizing a *cogen* function that transforms a program P into a program generator *pgen* that emits the specialized function in an intermediate representation (IR). Since the static part of input arguments $args^S$ also contains Python methods, we also apply *cogen* to them so that method calls will also generate IR.

$$f_{pgen} := cogen(Python, f) \qquad \Rightarrow \qquad mix(Python, f, env, args^S) = f_{pgen}(env, args^S_{pgen})$$

When applying *cogen* to a class, we will refer to the result as *virtual classes* and instances of these as *virtual objects*. Figure 3 sketches a conceptual implementation in Python.

3.2 SSA-Based Target IR

Although our abstract approach is independent of the intermediate representation used, an implementation needs to decide on one specific IR. In this section, we describe the design of our IR which we use for our later implementation.

Structure. Structurally, a module contains a set of global imports, akin to Python's `import`, Python source code for functions required for the fallback, and, as the main part, a set of functions. A function has a set of arguments and produces a single result. The body is a list of operations in SSA form, where the last operation must be the single return operation.

Operations. To make the IR flexible and easily extendable, only few operations exist and most functionality is modeled using `builtin` calls, which specify the operation through a string operand. This way, potential back-ends can implement different sets of built-in operations without the need to change the IR. Built-in operations that are not supported by the back-end can be handled by falling back to Python using the normal fallback mechanism. The few actual IR operations are for modeling constants and Python interoperability, i.e., importing modules, getting and setting attributes, and calling python objects.

Control Flow. For control flow, there is a single structured control flow operation, `if`, which has two nested operation lists and yields results depending on the branch. Other control flow like loops is represented by a built-in (e.g., `range.iter`) that takes a callback function called for every value. As a consequence, our IR has no ϕ-nodes. This approach not only simplifies the IR, analysis, and optimizations, but maintains the possibility to generate efficient machine code.

Table 2. Types in our IR and examples for corresponding Python source types.

Type	Description	Example source
`pyobj`	Opaque Python object	
`void`	Type of `None`	Python `NoneType`
`bool`	Boolean type	Python `bool`
`int`	Variable-width integer	Python `int`
`i`*N*	Fixed-width integer	`numpy.int64`
`f32/f64`	Floating-point types	Python `float`, `numpy.single`
`str`	Byte sequence	Python `str`/`bytes`
`record[`*name*`:`*type*`,...]`	Immutable struct	Python `tuple`/`slice`
`list[`*t*`]`	List with homogeneous types	Python `list`
`dict[`*k*`,`*t*`]`	Dict with homogeneous types	Python `dict`
`array[`*shape* `x` *type*`]`	*n*-dimensional array type	`numpy.ndarray`
`column[`*type*`]`	Data column	`pandas.Series`
`table[`*name*`:`*type*`,...]`	Table	`pandas.DataFrame`
`function_ref(...)→int`	Function reference	Used for `lambda`

Data Types. The set of supported data types is targeting the needs of data science workloads; Table 2 gives an overview. The primitive types closely correspond to standard types in Python or `numpy`. We keep variable-width integers separately for correctness, but a back-end could still decide to implement slightly different semantics and model them as 64-bit integers. Python's `str` and `bytes` types are only distinguished by the operations performed on them and are unified in the IR, as they both store an immutable sequence of bytes. The `record` type represents an immutable struct mapping field names to stored values and is used to implement `tuples` and other immutable classes (e.g., `slice`, `range`). Typed list and dictionary types correspond to the corresponding Python classes, but can only store the annotated types. We include special, high-level types designed for data processing libraries representing n-dimensional arrays, tables, and table columns.

To implement fine-granular fallback to CPython, the IR can represent and operate on the `pyobj` type, which represent an opaque, reference-counted Python object. Through this type the IR is also able to represent lists and dictionaries that contain objects of different types (e.g. `list[pyobj]`).

4 Implementation

In this section, we describe the practical implementation of the *cogen* function, which transforms Python functions into IR generators, also referred to as *transformed code.*

Table 3. Transformation rules for *cogen*.

⟦**def** f$_1$(a$_1$,...,a$_n$, p=d, *args, **kwargs):
    body
⟧$_{pgen}$
	=	**def** f$_1$(a$_1$,...,a$_n$, p=**None**, *args, **kwargs, ctx=**None**):
		    p = ⟦d⟧$_m$ **if** p **is** **None** **else** p
		    args = ctx.create_tuple(args)
		    kwargs = ctx.create_dict(kwargs)
		    ⟦$body$⟧

⟦var⟧$_{pgen}$	=	ctx.get_var(var)
⟦var=expr⟧$_{pgen}$	=	var=⟦expr⟧$_{pgen}$
⟦f(v$_1$,...,v$_n$, k$_1$=x$_1$,...,k$_n$=x$_n$)⟧$_{pgen}$	=	ctx.call(⟦f⟧$_{pgen}$, [⟦v$_1$⟧$_{pgen}$,...], {"k1":⟦x$_1$⟧$_{pgen}$,...})
⟦l op r⟧$_{pgen}$	=	ctx.binary_op(⟦l⟧$_{pgen}$,⟦r⟧$_{pgen}$, "__op__", "__rop__")

⟦**if** cond:
    ibody
else:
    ebody⟧$_{pgen}$
with read as set of read variables and changed as set of changed variables
Ternary expressions are handled similarly using the same _if method.
	=	**def** f$_{if}$(read...):
		    ⟦ibody⟧$_{pgen,nested}$
		    return (changed...,)
		def f$_{else}$(read):
		    ⟦ebody⟧$_{pgen,nested}$
		    return (changed...,)
		try:
		    changed... = ctx._if(⟦c⟧$_{pgen}$, [changed...,], f$_{if}$, f$_{else}$)
		except EarlyReturn e:
		    return e.val

⟦**return** e⟧$_{pgen}$	=	**return** ⟦e⟧$_{pgen}$
⟦**return** e⟧$_{pgen,nested}$	=	**raise** EarlyReturn(⟦e⟧$_{pgen}$)

⟦**lambda** a$_1$,...,a$_n$: body⟧$_{pgen}$
with C being the set of all bound variables and ⟦expr⟧$_{closure}$ being a transformation of each occurrence of $c \in C$ to __closure__["c"]. See Sec. 4.5.
	=	**def** f$_{pgen}$(a$_1$,...,a$_n$):
		    return ⟦body⟧$_{pgen}$
		def f$_{closure}$(a$_1$,...,a$_n$, __closure__):
		    return ⟦⟦body⟧$_{closure}$⟧$_{pgen}$
		def f$_{py}$(a$_1$,...,a$_n$, __closure__):
		    return ⟦body⟧$_{closure}$
		ctx._lambda(f$_{pgen}$,
		        lambda b: b(f$_{closure}$, {"c": c **for** $c \in C$}),
		        lambda b: b(f$_{py}$, {"c": c **for** $c \in C$}))

⟦**try**:
    tbody
except:
    ebody⟧$_{pgen}$
with read as set of read variables and changed as set of changed variables
	=	**def** f$_{try}$(read...):
		    ⟦tbody⟧$_{pgen,nested}$
		    return (changed...,)
		def f$_{except}$(read):
		    ⟦ebody⟧$_{pgen,nested}$
		    return (changed...,)
		try:
		    changed... = ctx._try(⟦c⟧$_{pgen}$, [changed...,], f$_{try}$, f$_{except}$)
		except EarlyReturn e:
		    return e.val

⟦c⟧$_{pgen}$ # c is literal	=	ctx.constant(c)
⟦o.attr⟧$_{pgen}$	=	ctx.get_attr(⟦o⟧$_{pgen}$, "attr")
⟦o.attr= e⟧$_{pgen}$	=	ctx.set_attr(⟦o⟧$_{pgen}$, "attr", ⟦e⟧$_{pgen}$)
⟦[v$_1$,...,v$_n$]⟧$_{pgen}$	=	ctx.create_list([⟦v$_1$⟧$_{pgen}$])

⟦**for** e **in** l:
    body⟧$_{pgen}$
	=	**def** f$_{loop}$(e, read_only, iter_vals):
		    ⟦body⟧$_{pgen,nested}$
		    return (iter_vals...,)
		changed... = ctx._for(⟦l⟧$_{pgen}$,[read_only...],[iter_vals...],f$_{loop}$)

4.1 Transforming Python Functions to Program Generators

cogen is implemented as a Python function that acts as a decorator and transforms the decorated function by rewriting the abstract syntax tree. To generate the IR in a more useful representation and avoid producing large, redundant ASTs for specific parts, the transformed function gains an extra `ctx` parameter of type `GeneratorContext`. This class handles both the IR construction as well as the non-trivial logic for handling control flow.

```
@cogen
def abs(x):
    return -x if x<0 else x
```

```
def abs(x, ctx):
    def if_fn(x, ctx):
        _tmp = ctx.neg_(ctx.get_by_name(x))
        return (_tmp,)
    def else_fn(x, ctx):
        _tmp = ctx.get_by_name(x)
        return (_tmp,)
    _islt=ctx.perform_binop(left=ctx.get_by_name(x), right=ctx.constant(0),
                            left_method='__lt__', right_method='__gt__')
    return ctx._if(cond=_islt, bodyfn=if_fn, elsefn=else_fn, inputs=[x])[0]
```

Fig. 4. *cogen* in practice: Applied as Python decorator, it generates a Python function that generates IR by calling methods of the `GeneratorContext`

The transformation consists of four steps: first, the Python code for the decorated function is retrieved using the `inspect` module, and is parsed using the standard Python `ast` module. Second, some constructs like list comprehensions are rewritten into simpler, equivalent forms, e.g., an explicit loop. Third, the AST is rewritten according to the rules in Table 3. Finally, the rewritten AST is compiled using Python's built-in `compile` to yield the modified Python function. Figure 4 shows the effect of this transformation for an example function. Every Python expression and statement is transformed into calls to the GeneratorContext object, which will generate the IR.

Due to the nature of the Python language, this transformation process shows many similarities to the virtualization approach proposed by Decker et al. [9]. However, originating from our different goals, there are also differences: First, while virtualization only focuses on rewriting constructs that are of interest for the embedded DSL and cannot be already overloaded in Python (e.g., control flow), we need to capture all constructs include list literals, operators, and object accessors. This also allows us to easily introduce an extra layer of indirection later in subsection 4.7. Second, `*args` and `**kwargs` require special handling for the same reason. Third, for objects like lambdas that might need to be converted to a Python object during fallback, we need to generate multiple versions of the body (cf. subsection 4.5).

Some of the complex cases resulting from Python's language semantics are handled similarly to the approaches already discussed by Decker et al. [9] in the context of virtualization. Python's scoping rules need special care: for example, when wrapping the body of `if` statements, loops, and `try` statements as functions, the code must explicitly update locally changed variables in the function scope afterwards. We can avoid the complexities of directly supporting scoping keywords (e.g., `nonlocal`) through the automatic fallback. Variables of closures are explicitly captured and the function body is transformed accordingly (cf. Table 3). Nested return statements, e.g., inside `if` statements, are rewritten to raise an exception; at call sites we wrap the call with `try`/`except` statements to propagate the result.

4.2 Virtual Objects and Types

The core idea of our approach is to use the Python interpreter at generation time to run the transformed code. Generally, every virtual object corresponds to exactly one IR value. Every virtual object has a high-level, virtual type, which holds type information not expressed in the IR. For example, both `range` and `tuples` are expressed as `record` in the IR, but are different types in Python. A virtual type has sufficient information to construct a new virtual object, for example, to represent the result of a call to a built-in function.

Figure 5 depicts a basic implementation of the virtual object and type classes necessary for implementing integers and addition on them. Methods annotated with `@pgen` are not transformed and implement the IR generation explicitly. For example, `__add__` directly calls the `gen_builtin` method on the supplied `GeneratorContext`, supplying the virtual return type (`IntType`) and the

```python
class VirtualType:
    @abstractmethod
    def ir_type(self) # IR type for this type
    @abstractmethod
    def construct(self, val: ir.Value) # new virt object

class VirtualObject:
    def __init__(self, irval: ir.Value):
        self._irval = irval
    def __irval__(self):
        return self._irval
    @abstractmethod
    def __virttype__(self) # get virtual type

class IntType(VirtualType):
    def ir_type(self):
        return ir.int
    def construct(self, val: ir.Value):
        return _int(val)

@cogen_class
class _int(VirtualObject):
    def __virttype__():
        return IntType()

    @pgen # => not transformed, emits IR explicitly
    def __add__(self, other, ctx: GeneratorContext):
        if not isinstance(other, _int):
            raise NotImplementedError()
        return ctx.gen_builtin("int.add",
            IntType(), [self, other])
    @cogen # => transformed to program generator
    def __iadd__(self, other):
        return self + other
```

Fig. 5. Basic implementation required for the original example.

```python
class PyObjType(VirtualType):
    def ir_type():
        return ir.pyobj
    def construct(val: ir.Value):
        return pyobject(val)

@cogen_class
class pyobject(VirtualObject): # opaque Python object
    def __init__(irval: ir.Value):
        self.__irval__ = irval
    @pgen
    def __add__(self, other, ctx: GeneratorContext):
        return ctx.gen_builtin("python.operator.add",
          PyObjType(), [self, other.__topython__()])
    @cogen
    def __topython__(self):
        return self # already a pyobject

@cogen_class
class _int(VirtualObject):
    ...
    @pgen
    def __topython__(self, ctx: GeneratorContext):
        return ctx.gen_builtin("int.to_python",
           PyObjectType(), [self])

class GeneratorContext:
    def binary_op(left, right, name, rname):
        try:
            getattr(left, name)(right, self)
        except NotImplementedError:
            try:
                getattr(right, rname)(left, self)
            except NotImplementedError:
                # Fallback, convert left to pyobject
                left_py = left.__topython__()
                getattr(left_py, name)(right, self)
```

Fig. 6. Adding automatic fallback to Python.

```python
def pyfn(x):
    ...

@compiled
def foo():
    x = ...
    y = pyfn(x)
```

(a) Calling Python functions

```python
#numpy/__init__.py
@compiled
def zeros(..., order='C', ...):
    if order != 'C':
        raise NotImplementedError()
#main.py
array=numpy.zeros(10, order='F')
```

(b) Unsupported functionality

```python
if cond:
    x = 0.0
else:
    x = 0
# x cannot be statically typed
```

(c) Diverging types in control flow

Fig. 7. Three situations in which the proposed framework falls back to the Python interpreter.

arguments. The gen_builtin method will then collect the IR values from the virtual objects provided as arguments, create and emit a new builtin IR operation, and construct a virtual _int object using the provided type. In contrast, __iadd__ (in-place addition like x+=1) uses the @cogen annotation because the addition operator is already defined and no explicit IR generation is required.

4.3 Fine-Grained Fallback to Python

Already for the addition operator, a fallback to Python might be required, either because the virtual object has no __add__ method or because the implementation does not support the supplied type combination. Figure 6 outlines the required extensions to support this. First, we need virtual type (PyObjType) and virtual object (pyobject) classes to represent opaque Python objects. This virtual object class implements __add__ (and others) by converting the argument to a virtual Python object (pyobject) and generating a python.operator.add built-in operation. We do not directly call __add__ on the Python object, as this would have different semantics and not try __radd__ in case of exceptions. Second, every virtual object needs to implement a __topython__ method that converts the current state into an equivalent pyobject. Finally, the binary_op method of the GeneratorContext will detect exceptions and perform the fallback to Python by converting the left argument to a pyobject and calling the corresponding method on it.

For other Python features, the fallback is implemented similarly. When an unannotated Python function is encountered (Figure 7a), e.g., from a not-yet supported module, the call will be executed in Python. A failure to generate *native IR* for a specific feature in a pgen-function is signaled by raising a NotImplementedError (Figure 7b). This exception is caught by the GeneratorContext, which converts the corresponding arguments to Python and generates a Python call operation. Another case that causes fallbacks are diverging types for a variable (Figure 7c): when multiple control flow paths lead to a variable having different types, it will be converted to a Python object.

In general, the fallback is performed on the finest-granular operation, i.e., the fallback is invoked for the current operation as soon as the generation fails. However, when, e.g., one operation inside the list.append method fails, it is more appropriate to apply the fallback to the entire append method. Thus, we defer handling exceptions if we are inside a function or method having a built-in Python equivalent.

4.4 Eager Execution and Lazy IR Generation

Previously, we asserted that every virtual object corresponds to exactly one IR value. However, in some cases, it makes sense to carefully lift this restriction and defer generating IR operations and values. This allows for virtual objects that do not hold an IR value, but instead have an *unmaterialized* state (stored in Python objects) that is not yet captured in the IR. If the IR value is required at a later point, the method __irval__ on such objects will cause a materialization of the state into IR and a transition to a materialized virtual object.

The possibility of maintaining unmaterialized state is crucial for containers like lists or dictionaries that can contain *inconsistently-typed* objects. Without unmaterialized state, one would have to immediately fallback to Python objects to, e.g., create a list with different types ([1,"a"]). Often, however, such inconsistently-typed containers are deconstructed again shortly afterwards (e.g., pandas.DataFrame.from_dict takes inconsistently-typed dictionaries as argument). We implement *concrete* subclasses for lists and dicts that store the contained virtual objects as Python objects and support a subset of the operations generating IR, for example, accessing an entry using a _const_int.

Another benefit of unmaterialized state is constant folding: if both operands of, e.g., the addition $1 + 1$ are unmaterialized virtual objects, we eagerly evaluate such expressions during the generation, again resulting in an unmaterialized virtual object, in the example holding a Python integer of value 2. By subclassing these unmaterialized virtual objects (e.g., _const_int) from the corresponding materialized virtual object class (e.g., _int), we can selectively implement constant-folding for supported operations, while every other operation is handled by the base class.

Compared to approaches like lightweight modular staging [33], we do not mark selected values as being executed in a later stage and execute everything else immediately. Instead, to avoid the need for cross-stage persistence, our approach allows for implementing eager execution as optimization for selected methods, where the result can be materialized in the IR.

4.5 Implementing Lambda Functions

Lambda functions (and nested functions) are frequently used to pass specific behavior to other functions. However, implementing them requires handling captured variables. There are three different cases to handle and for each of these we generate one variant as shown in Table 3.

(1) The lambda is called explicitly. For this case, we apply *cogen* to the lambda and rely on Python's built-in capturing semantics ($\mathtt{f}_{pgen}$).
(2) The lambda is passed to a built-in IR operation (e.g., `range.iter` for iterating over a range). For this case, we apply *cogen* to the lambda, but rewrite accesses to captured variables to go through an extra `__closure__` record passed as argument ($\mathtt{f}_{closure}$). $\mathtt{f}_{closure}$ is then used to generate an IR function for a given set of argument types.
(3) The lambda is passed to Python. For this case, we do not apply *cogen*, but just rewrite accesses to captured variables to go through a lookup dictionary passed as argument ($\mathtt{f}_{py}$). The captured virtual objects are converted to Python and $\mathtt{f}_{py}$ is then partially evaluated regarding the closure argument, yielding a bound Python lambda function.

4.6 State Modifications During IR Generation and Type Inference

By design, during IR generation also state modifications occur, including emitting IR operations, transitions as discussed in subsection 4.7, and changes to unmaterialized state (e.g., appending to a `_concrete_list`). However, without additional care, these side-effects can lead to wrong results: for example, if one branch of an if-statement modifies an unmaterialized object, it must be restored to the previous value for the else-branch.

We solve this problem by implementing "transactional" behavior, similar to transactions in a database system. In the affected methods of the `GeneratorContext`, we wrap the problematic sections in a transaction that captures all modifications to the state (`with self.transaction() as T:`). Later, the captured changes can be analyzed and committed (`T.commit()`) or rolled-back (`T.abort()`).

For example, to implement an if-statement, both branches are evaluated and wrapped in two separate transactions that can be rolled back. Thus, both branches can only observe the side-effects that were present before the if-statement. Afterwards, the collected IR operations are emitted as part of the generated if operation. Changes to unmaterialized state are merged and consolidated, similar to type inference as discussed below.

Type Inference. With our approach, the need for explicit type inference only arises when there are two incoming edges in a control flow graph. This is the case for if-statements where a common type must be found for yielded results from the two branches, but also for loops where the types must stay the same independent of the number of iterations.

The required consolidation is implemented in a modular way. Every virtual object can implement a `__merge__` method to consolidate the types of itself and another virtual object. It decides on a consolidated result type and returns it together with two functions that generate IR in order to convert both virtual objects to this type. If such a consolidation is not possible, e.g., because of conflicting types, the consolidated type will be `pyobject`.

For an if-statement, the yielded virtual objects are compared pair-wise and, if necessary, the conversion functions produced by `__merge__` will be used to convert the result of both branches to the common type before yielding them. For loops, type inference is more challenging as the type of the values defined before the loop influences the loop's behavior and the type of iteration values. Thus, we perform a fix-point iteration by executing the loop's body, calling the merge function for all changed values, converting only the objects defined before the loop, and finally resetting all side-effects. Only when the types of all changed variables stay the same after one loop iteration, we actually generate the IR for the loop.

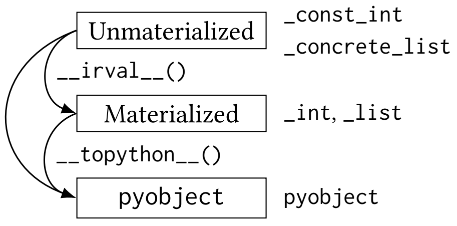


Fig. 8. Different virtual object classes with examples and possible transitions.

```
# l is an unmaterialized _concrete_list
l = [1, 2, 3]
# Create a new pyobject for l
converted = l.__topython__()
# Append a new value to the pyobject
converted.append(13)
# Without further measures, this would modify
# the _concrete_list, not the pyobject!
l.append(42)
```

Fig. 9. Transitions could lead to multiple virtual objects for the same logical value.

4.7 Transitions of Mutable Objects

Both the Python fallback and the eager evaluation optimization require transitions between different virtual objects representing the same logical object. The Python fallback requires converting arbitrary virtual objects to opaque `pyobject`'s, and unmaterialized virtual objects are eventually materialized. Figure 8 depicts those transitions systematically. While these transitions are unproblematic for immutable virtual objects (e.g., integers), a naïve implementation leads to problems for mutable objects as exemplified in Figure 9.

In our implementation, we avoid this problem by adding an extra layer of indirection by wrapping all virtual objects into a Python object of type `VirtualObjectHolder`. When transitioning between virtual object classes, we also update the reference of the holder object, so that all later usages will use the updated value. Introducing this indirection is possible, since we have control over the code through the transformation step discussed in subsection 4.1 and do not use bare Python objects to perform type-based staging.

4.8 Post-Mortem Escape Analysis

One of the core contributions of our approach is the fine-grained fallback to Python object if some functionality is not supported or Python functions are called.

However, converting *mutable* native data-structures to the Python equivalent can lead to inconsistencies if the Python object is modified and the native data-structure is still used later. Such usages of *outdated* values can occur when mutable objects are nested inside other objects and one of the objects is later converted to Python. One example where this can happen is with nested lists (e.g., `list[list[int]]`): if the outer object is converted to Python, later uses of the inner object would use a potentially outdated value. If the inner object is converted to Python independently from the outer object, usages of the outer object may still access the outdated inner value. The underlying problem is that one mutable logical object has two unsynchronized physical representations.

An example for this situation is shown in Figure 10a. This problem of inconsistent state across different representations of one logical problem is hard to detect during the translation process without over-aggressive countermeasures. Especially as outdated values only occur rarely in practice, proactively converting values to Python when, e.g., mutable values are inserted into a list, is not desirable. However, when the problematic conversion occurs, it is impossible to determine which other objects must be converted.

We propose a fix-point algorithm for this problem. First, we generate the IR optimistically and track value uses, nesting of mutable objects, and conversions to Python. Afterwards, we analyze the results (thus *post-mortem*) to identify problematic uses that could lead to inconsistent state. For this, we use a Union-Find data structure. For every observed *nest* relation between two objects, their sets are merged; for every *conversion* to Python, the object's set is merged with the special

```
def bar(l): # will be called using CPython
  l[0][0] = 42
@cogen
def foo():
  inner = [0]
  outer = [inner] # assumed to be materialized
  bar(outer)   # outer converted to Python
  print(outer) # will print [[42]]
  print(inner) # would print [0], not [42]!
```

(a) Problematic Example: inner also escaped to Python and thus needs to stored as Python object from the beginning.

```
#
#
#
#
#
Nest(outer, inner)
Convert(outer)
# already a pyobj
Use(inner)
```

(b) Trace of uses, conversions, and nestings.

$\{invalid\}$
$\{invalid\}, \{inner\}$
$\{invalid\}, \{inner, outer\}$
$\{invalid, inner, outer\}$

$Find(inner) = Find(invalid) \Rightarrow$ Problem!

(c) Analysis using union–find. inner and invalid end up in the same set, exposing the problem.

Fig. 10. Example for post-mortem escape analysis

node *invalid*. For every value *use*, we check whether the used object and *invalid* are in the same set — if they are, the object should have been eagerly converted to Python. This information is stored and the generation process is rerun with those decisions in mind to avoid the detected problems. Since the resulting changes might lead to new inconsistencies, the process is repeated until a fixed point is reached and no more observable inconsistencies are identified.

To make this approach work, virtual objects must be uniquely identifiable even across multiple generation runs. When a virtual object is created, it computes a tuple of location identifiers that correspond to a refined call-stack on expression granularity. This tuple is then checked against the list of identifiers to be eagerly converted, and if required, the virtual object is immediately converted to Python.

5 Supporting Popular Python Libraries

Our approach is highly modular and allows for partially supporting popular Python libraries with function/method granularity. Unsupported modules, functions, and methods are transparently handled by the fallback mechanisms without any involvement of users. This allows us to focus on the most frequently used methods first and then expand coverage based on needs. In this section, we discuss how we support selected libraries like Python's built-in modules, numpy, pandas, and scikit-learn. Table 4 gives an overview of the supported modules, classes and functions.

Table 4. Supported Python modules

builtins	**numpy**	**pandas**	**scikit-learn**	
bool, int, float, str,	ndarray	Series	LinearRegression	**urllib**: urlparse,
list, dict, tuple,	int64	DataFrame	LogisticRegression	**statistics**: mean,
range, str, print, min	float64	RangeIndex	Pipeline	stdev
max, sum, sorted	ones, zeros, array	Index	KMeans	**scipy**: special.erf
		MultiIndex	MinMaxScaler	**pickle**: loads

5.1 Bootstrapping with the `intrinsics` Module

Previously, we showed the implementation of non-transformed methods as raw function (@pgen), and while this approach is powerful and allows for full access to the internals, it is not trivial to write such functions correctly, particularly for more complex logic. To simplify the bootstrapping process, we developed an `intrinsic` module that implements commonly required functionality as raw functions that can be called from `@cogen` functions.

```
@cogen_class
class str:
 @cogen
 def __getitem__(self, item):
  if intrinsics.isa(item, int):
   return intrinsics.call_builtin("string.at", str, [self, item])
  elif intrinsics.isa(item, slice):
   length = len(self)
   start = item.start if item.start is not None else 0
   stop = item.stop if item.stop is not None else length
   start = start if start >= 0 else length + start
   stop = stop if stop >= 0 else length + stop
   if item.step is None:
    return intrinsics.call_builtin("string.substr", str, [self, start, stop])
   else:
    return "".join([self[i] for i in range(start, stop, item.step)])
  else:
   intrinsics.not_implemented()
```

Fig. 11. Implementation of the `__getitem__` method of `str`.

```
1 @cogen_class
2 class list:
3  @cogen
4  def sort(self):
5   # list._element_type is a type object representing the current type of the list elements
6   # for inconsistently typed lists, this type will be pyobject
7   compare_fn = intrinsics.bind(lambda l, r: l < r, [self._element_type, self._element_type])
8   intrinsics.call_builtin("list.sort", None, [self, compare_fn])
```

Fig. 12. Implementation of the `sort` method of `list`.

5.2 Python's `builtins` Module

In Python, the built-in `builtins` module contains core classes like `int` and core functions like `print`. Using the intrinsics module, we were able to implement a large subset of the `builtins` module without raw functions.

As an example, consider the method `str.__getitem__`, which implements the subscript operator, whose implementation is shown in Figure 11. We implemented this using idiomatic Python code combined with calls to functions of the `intrinsics` module. For strings there are two cases: an integer argument is used to access a single character (e.g, `"abc"[1]`), or a slice is used to obtain a substring (e.g, `"abc"[1:-1]`). For the first case, we generate a call to a builtin function to access the respective character. For the second case, we first normalize the start and stop positions of the slice, to correctly handle missing and negative values. If the slice contains no step value, then the slicing operation just corresponds to a substring operation, for which we generate a builtin call. In the case where a step value is given, we write idiomatic Python code to compose the final string. If the argument is another type (e.g, `pyobject`), we raise a `NotImplementedError` at generation time.

As another example, that also shows more how high-level semantics is extracted, consider the `list.sort` method, whose implementation is shown in Figure 12. We do not want to implement the sorting procedure in Python, but instead emit a `"list.sort"` builtin call, that also receives a callback function that can compare two list elements. In Line 5, this callback function is generated from the simple lambda using the `intrinsics.bind` method, that materializes the lambda function into an IR function and returns a function reference.

By injecting this `builtins` module into the global namespace of every transformed function during the rewrite process, these functions will automatically use the virtual objects and functions.

```
# input: lambda row: row["b"] + str(row["a"])
func lambda_fn(r : record[a: int, b: str, __index__: int]) -> str {
    a = record_get r[a]
    b = record_get r[b]
    str_a = int.to_string(a)
    res = string.concatenate(b,str_a)
    return res
}
```

Fig. 13. Generated lambda function, without the need to create a temporary `Series` object

5.3 numpy

For a basic support of the `numpy` library, we implemented virtual object classes for `int64`, `float64`, and `ndarray`, as well array creation functions and selected numerical functions (e.g., `np.sqrt`). For `int64` and `float64`, common operators and conversions to and from Python's native int and float types are supported. The `ndarray` virtual object supports common vectorized operations (e.g., element-wise additions) as well as accessing elements by indices (e.g., `arr[1,2]`), slicing to create partial views on the array (e.g., `arr[1:-1, 2]`), and reshaping. The virtual objects already perform a lot of heavy lifting such that only few different IR operations are generated. For element-wise operations only calls to two builtin functions are generated: `array.apply` and `array.binary_op`.

5.4 Pandas

Pandas is a rather complex library, but we make sure that the parts supported by HiPy are semantically equivalent, unlike several other existing implementations. In addition to the two main classes, `DataFrame` and `Series`, we also implemented different kinds of indices required for full compatibility. The entire supported functionality is mapped to simple IR operations operating on `table` and `column` types.

While Pandas is typically faster than writing raw Python code, it is still not tailored towards high performance, often due to hiding the effective semantics behind many (costly) abstractions. As an example, consider the `DataFrame.apply` method that allows for applying a function row-wise: `df.apply(lambda row: row["b"]+str(row["a"]), axis=1)`. For each row, Pandas creates a new `Series` object with the column labels forming the index and the values the series' data. However, in many cases, the supplied function will only access the values for the different columns using the subscript operator (e.g., `row["a"]`). In our virtual implementation, we can avoid creating `Series` objects at run-time, which is especially beneficial, because this object is typically inconsistently typed. Creating an unmaterialized `Series` objects with unmaterialized `Index` objects allows the lookup to happen at generation time. Still, if any other method is used, we can still materialize the series, allowing for full compatibility. Figure 13 shows the IR function generated as callback for a `table.apply_row_wise` builtin function, demonstrating how our approach can tear down (costly) abstractions to reveal the real semantics.

5.5 ML Inference with scikit-learn and pickle

It has become quite common in data processing pipelines to perform model inference on tabular data (e.g., for fraud detection), often using simple models like linear or logistic regression. Typically, these models are pre-trained and stored as a binary using Python's `pickle` module, and then unpickled for inference. We support this use case by implementing `pickle.loads`, which performs the unpickling at generation time if the binary string is already known. It then tries to lookup the corresponding virtual class and calls a special static method (e.g. `LinearRegression.__loadfrom__`) that creates a virtual object from a live Python object. For linear and logistic regression, k-Means, MinMaxScaler, and scikit-learn pipelines, we implemented this static method by creating (unmaterialized) virtual

```
# fn = extractBd
# 'b' = 'bedrooms'
# 'f' = 'facts and features'
df['b']= df['f'].apply(fn)
df=df[df['b'] < 10]
```

(a) Simplified first two lines of *Zillow* benchmark.

```
%colf = table.get_column %table, "f"
%colb = column.apply %colb, %fn
%table2 = table.set_column %table, %colb, "b"
%colb2 = table.get_column %table2, "b"
%cond = column.apply %colb2, %fn_lt_10
%r = table.filter_by_col %table2, %cond
```

(b) Generated column-centric IR operations

```
%table2 = table.map %table, %fn
↪ {"in": "f" => "out": "b"}
%res = table.filter %table2,
↪  %fn_lt_10 {"in": "b"}
```

(c) Table-centric IR operations after optimization

Fig. 14. Rewriting column-centric operations to table-centric operations

objects for each attribute, and create a virtual model object using these values. Additionally, we implemented support for both predict methods that compute the predicted values using numpy and the relevant attributes (e.g., `coeffs_` and `intercept_`). Thus, users can use existing, pre-trained models without any change, while our approach allows to extract the high-level semantics (i.e, float additions and multiplications). All other methods are of course still supported through the Python fallback.

6 End-to-End Optimization and Compilation

To evaluate the effectiveness of our approach, we also built an end-to-end prototype that performs different high-level optimizations and generates efficient C++ code.

6.1 Optimization Passes

The generated, high-level IR allows for many optimizations that can both simplify the extraction process of logical operators, but also improve performance. In this section, we discuss the implemented optimization passes.

General Optimizations. We implemented a set of standard optimization passes on the IR like dead code elimination, which mostly serve the purpose of supporting other optimizations and cleaning up the IR. Many complex, domain-specific optimizations fuse loop-like operations, which leads to unnecessary function calls in fused call-back functions. Applying *function inlining* and canonicalization patterns to, e.g., eliminate pack/unpack operations for records simplifies the IR and also improves performance.

Fusing Array Operations. Currently, we implement one central optimization for numerical workloads: fusing of declarative array operations. This is done in two steps: First, high-level array operations such as `array.binary_op` and `array.unary_op` are transformed pattern-based into `array.compute` operations that take a variadic number of arrays and a scalar function that computes the new array from the scalar values. Chained `array.compute` operations can then be easily fused into one `array.compute` operation, by combining the scalar callback functions.

Optimizations for Tabular Data Processing. We also perform different optimizations for tabular data processing, for example to efficiently execute Pandas workloads. In the first step, different rewrite patterns are applied to rewrite the IR for common patterns. This is crucial for converting column-centric operations commonly performed by pandas into table-centric operations that correspond to relational operators in existing systems.

Figure 14 shows this rewrite step for the first two lines of the *Zillow* benchmark that we will use in the evaluation. In the generated IR, columns are fetched from the table first, followed by computations, which generate new columns. These are then used to either create a modified table or filter a table. With pattern-based rewrites, we can turn these IR operations into a simple `table.map` and `table.filter` operations

As a second step, the supported operations can be extracted to form a tree of relational operators. This can be further optimized using traditional query optimization techniques, for example, by pushing down filters or joins. While one would typically delegate these optimizations to the surrounding data processing system, we also implemented them in our standalone prototype. To further demonstrate the impact of extracting the high-level semantics, we fuse pipelines of table-centric operators to generate efficient, data-centric code as proposed by Neumann [23].

6.2 C++ Back-End and Runtime Library

For the prototype, we generate standalone C++ programs from the IR using simple string patterns. In addition to generating C++ code for the IR operations, we also generate C++ code for setting up the embedded Python interpreter (i.e., importing Python modules, defining local Python functions). However, the approach is not limited to such a back-end, as one could also generate a more complex and powerful IR, like MLIR [19], which would allow further and more advanced optimizations.

Runtime Library. To avoid complex generation logic, the standalone library is linked with a small runtime library. For primitive types like strings, integers, floats, boolean values, lists, and dictionaries, we just use the corresponding types of the C++ standard library. In addition, we implement functions that are not already implemented in the standard library such as special string functions. For embedding the Python interpreter and implementing operations on Python objects, we use the `pybind11` library. To represent numpy's n-dimensional arrays and zero-cost views, we implement a custom C++ class that contains a pointer to the original data and implements logic to deal with dimensions and strides. For tabular data, we use Apache Arrow's columnar in-memory format, which allows for a zero-copy, efficient integration with data processing systems. Additionally, Apache Arrow already implements logic for the conversion pandas.

7 Evaluation

In this section, we evaluate the proposed design as well as the implemented prototype by performing end-to-end experiments. We show that HiPy can handle complex Python code, allows for effective logical optimizations, and offers high performance. Furthermore, we also show that the fallback mechanism allows us to run complex programs even if some parts are not fully supported, without a notable performance penalty. These benefits can be observed not only for the original target of data-science and engineering workloads using pandas, but also for typical scalar UDFs, numerical workloads and general Python benchmarks.

All experiments were performed on a machine running Ubuntu 24.04 (Linux 6.8.0-36) with an Intel Xeon Gold 6430 CPU (3.4 GHz) and 256 GiB memory. We focus on single-threaded execution, as auto-parallelization and auto-distribution are orthogonal, domain-specific problems. All shown performance numbers only include the execution times for the evaluated approaches where this is easily separable; for PyPy, results include tracing and compilation overhead. HiPy-generated C++ code is compiled with GCC 11. Furthermore, every experiment was run once for warmup and then repeated three times for measurement. We report the median for these three runs and the standard deviation was generally below 5%.

7.1 Data-Science and -Engineering Workloads

The main focus of our work lies on supporting data-science and data-engineering workloads written in Python using a framework like Pandas. To show the impact of our approach, we selected 4 different benchmarks. The first 2 are taken from Tuplex's [38] benchmarking suite: *Logs* parses webserver logs and joins the result with a list of malicious IP addresses; *Zillow* extracts and filters data from a string-based real estate data set using Python functions. The other benchmarks are 2 of

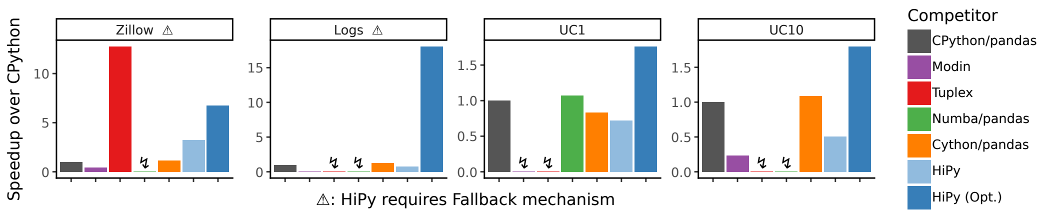


Fig. 15. Performance for data-science and data-engineering workloads.

10 use cases from the TPCx-AI benchmark [6] that are bottlenecked by preprocessing tabular data and not by machine-learning inference or sampling audio files: *UC1* joins and aggregates data and performs inference with a pre-trained k-means model; *UC10* first joins two tables, parses strings into timestamps and normalizes features, before applying a pre-trained logistic regression model for inference.

We compare the performance of our approach with and without optimizations to CPython 3.12.2 using pandas 2.2.0, Tuplex 0.3.6, and Modin [28] 0.28.0. For pandas, we also include experiments where pandas is accelerated with Numba [18] or Cython (version 3.0.9). However, Numba was only able to accelerate a single experiment, UC1. Unfortunately, Tuplex could not run the Logs experiment due to a segmentation fault and also does not implement the required pandas API for UC1 and UC10. We also looked into SDQL's pandas front-end [34], Weld's pandas front-end [27], and Grizzly [13], but they all lacked the functionality to execute any of the benchmarks: for Logs and Zillow, this is due to tracing-based approaches being unable to handle arbitrary Python code; for UC1 and UC10, they do not support the complex operations and ML inference.

Figure 15 shows the measured speedups over CPython where we can observe that *HiPy (Opt)* is significantly faster than the Pandas implementation, even when combined with Numba and Cython, which do not significantly speed up these workloads. This speedup is realized despite materializing the result first into Apache Arrow's columnar format, which is expensive to build compared to building a pandas DataFrame, and despite HiPy using the fallback mechanism for Zillow and Logs to support Python's regex module and string formatting, as this is not yet implemented. This is also why Tuplex is two times faster than HiPy: they fully implemented exactly the functionality required for this benchmark. Furthermore, we can observe that *Zillow* already profits a lot from generating efficient C++ code, as shown by the 3.2x performance increase of HiPy without optimizations, because it heavily uses Python code for string processing. We also observe an additional 2x improvement in performance from the various optimizations discussed in section 6. *Logs* substantially benefits from rewriting the IR to perform table-centric operations, which in turn allows for pushing a selective join down, yielding a speedup of 18x for *HiPy (Opt)*. For *UC1*, and *UC10*, we can observe that, without optimizations, HiPy is even slower than pandas, due to the high materialization cost of using Apache Arrow. However, the pattern-based optimizations and the fusing of operations still lead to a speedup of 1.8x for *HiPy (Opt)*.

Modin does not optimize the pandas operations itself and primarily focuses on the orthogonal problem of parallelization and distribution, which leads to a poor single-threaded performance. It also does not inline the model inference for UC10, and the execution on UC1 fails with an exception.

7.2 Scalar UDFs

User-defined Python code is often used in data-processing systems as scalar user-defined functions, that are invoked for every scalar value. Extracting the high-level semantics may allow for logical optimizations and estimating the UDF's cost, but can also avoid significant calling overhead by

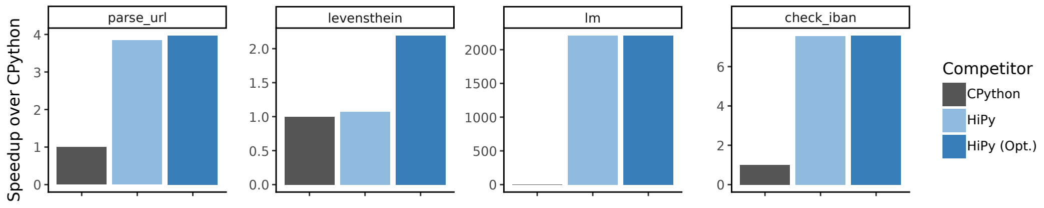


Fig. 16. End-to-end performance for a table scan written in C++, calling scalar UDFs.

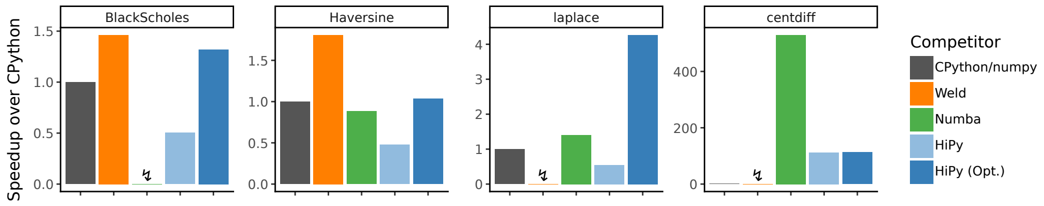


Fig. 17. Performance for numerical workloads (excluding compilation times).

generating native code. We evaluate HiPy's performance by measuring the end-to-end performance for a table scan written in C++, calling different scalar UDFs, either by using HiPy to generate native code or by embedding CPython. Due to this methodology, it is hard to compare with other implementations. We selected four different UDFs: IBAN number validation, Levenshtein string distance, URL parsing, and inference on a scikit-learn linear regression model (*lm*).

Figure 16 shows the measured speedups over CPython. We can observe that, even without optimizations, HiPy is always faster than the Python baseline, due to generating efficient code and by avoiding materialization and conversion overheads. Levenshtein further benefits from inlining, but as our implemented optimizations primarily target array and tabular data processing, they have low impact on the other UDFs. The *lm* benchmark shows the strength of our general approach as it manages to reduce inference for a linear regression model down to a few floating-point operations that can be inlined in C++, resulting in a speedup of 2200x.

7.3 Numerical Workloads

While not the primary focus of our approach, we also evaluate HiPy's performance on numerical workloads. They are supported due to the generality of our approach. For the evaluation, we performed four numerical benchmarks with different emphasizes. The first two benchmarks (*Blackscholes* and *Haversine*) were taken from Weld's [27] benchmarking suite and are mostly using numpy's vectorized array operations. The other two benchmarks were taken from Numba's benchmarking suite: *laplace* implements an iterative algorithm, performing vectorized operations on smaller array slices; *centdiff* only uses numpy arrays for efficient storage and performs the computation in Python. We compare the performance of HiPy against the baseline of using CPython 3.12.2 with numpy 1.26.4, Numba [18], and Weld 0.0.6. However, due to relying on the tracing approach, Weld cannot run the laplace and centdiff benchmarks. Numba could not run Blackscholes due lack of support for a function from `scipy.special`.

Figure 17 shows the measured, single-threaded speedups over CPython for the different implementations. For vectorized operations in the first two benchmarks, Numpy is already heavily optimized and difficult to outperform. HiPy with optimizations is just barely faster, Numba is

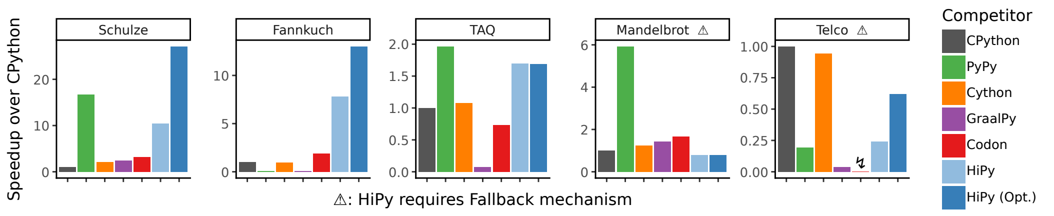


Fig. 18. Performance for general Python benchmarks. Note that *mandelbrot* and *telco* are supported by fined-grained fallback to Python.

slower than the baseline, and Weld is faster by 1.8x only for the Haversine benchmark, probably due to vectorized execution. For the other two benchmarks, both Numba and HiPy yield substantial performance improvements, as a significant part of it consists of raw Python statements operating on the numpy array. For laplace, the high-level fusing optimization makes HiPy more than 4x faster than Python, whereas Numba is just slightly faster. For centdiff, HiPy does not benefit from logical optimizations, but it is still ~110x faster than Python, due to generating efficient C++ code. Nonetheless Numba achieves a much higher speedup of 530x, as it performs more optimizations for numerical workloads.

7.4 General Python Programs

To show the generality of our approach, we also evaluate HiPy on five benchmarks used to evaluate general Python implementations. *Mandelbrot*, *Fannkuch*, and *TAQ* are taken from Codon's [35] benchmarking suite, *schulze* is taken from PyPy's benchmarking suite[2] and slightly adapted to work for Python3, and *Telco* is from the PyPerformance suite[3]. These benchmarks were selected since they are complex enough to test real behavior (e.g., no fibonacci computation or word count), but also do not stress-test Python's object-oriented features, as this would lead to HiPy to completely fallback to Python with no additional insights. HiPy relies on the automatic fallback mechanisms for *Mandelbrot* and *Telco* due to a lack of an implementation for complex and decimal numbers, which are heavily used in these benchmarks. We compare HiPy against CPython 3.12.2, Cython 3.0.9, PyPy 3.10, GraalPy 24.0, and Codon 0.16.3.

Figure 18 shows the measured speedups over CPython for all benchmarks and implementations. For the three fully supported benchmarks, we can observe that HiPy achieves substantial improvements of 1.7x (*taq*), 13x (*fannkuch*), and 27x (*schulze*) compared to CPython, even outperforming Codon, Cython, and PyPy. The key reasons for HiPy's performance are optimizations like inlining and simplification as well as generating C++ code. For the two other benchmarks, we can observe neither significant slowdowns nor speedups compared to CPython, while the performance of Codon, Cython, and PyPy varies. Performance of GraalPy varies strongly, between an 2x speedup and a 26x slowdown.

Performance of Frequent Fallbacks. While fallbacks allow us to support full Python semantics, they also come at some cost. In particular, frequent conversions of objects can cause performance regressions, even though every object is converted at most once. To evaluate this situation, Figure 18 includes two benchmarks where the main logic requires heavy fallback due to not-yet supported types: *Mandelbrot* uses complex number support and *Telco* heavily uses of decimal numbers, both

[2]https://foss.heptapod.net/pypy/benchmarks/-/blob/branch/default/own/schulze.py?ref_type=heads
[3]https://github.com/python/pyperformance/tree/main/pyperformance/data-files/benchmarks/bm_telco

are currently not implemented. Despite the main logic heavily using the fallback mechanism, the performance does not substantially regress.

8 Discussion

Explored Alternatives. Preceding the proposed approach, we initially experimented with AST-based translation and translating Python bytecode. For the AST-based approach, however, we needed to re-implement Python's semantics at many points, which we now avoid by generating Python code. Furthermore, the resulting translator was monolithic, constraining maintainability and debuggability. Extracting semantics from Python bytecode proved to be difficult for more complex programs and it would have required complex lifting logic to reconstruct the program semantics at a higher level.

Limitations. In principle, our approach could fail due to three reasons. First, users could use (currently) unsupported Python features like coroutines, class definitions, type alias, with, pattern matching, generators, scoping keywords, string templates, and decorators. Second, our implementation is vulnerable to specific code patterns that would lead to an exponential generation time. Finally, if the main assumption of our approach that the used methods/classes are not modified at runtime (i.e., monkey patching) is invalid, the generated IR would not be semantically equivalent. The first two cases can be handled generically by falling back to Python when unsupported features are encountered or bad behavior is detected (e.g., by introducing a counter for the number of iterations). The last case cannot be handled by our approach, but Python code written for data processing usually does not involve monkey patching.

User Experience. Because we follow Python's semantics, end-users can use the existing Python interpreter and standard Python debugging tools for prototyping and debugging. Dealing with the generated C++ code is only necessary if bugs in HiPy are encountered. In these cases, HiPy offers a comparable experience to established frameworks like Numba, which generates LLVM IR.

Developer Experience. By implementing everything in plain Python, our approach is easy to distribute and extend. Library replacements are also written in idiomatic Python, so even library authors without experience in compiler technology can contribute. One can also debug the IR generation by just using the standard Python debugger, since we maintain the code locations during the transformation process. In contrast to approaches like Codon, which is written in C++, we deliberately prioritize developer experience over slightly longer generation and compilation times.

Performance Profile. For workloads not evaluated in this paper, performance can vary but should not be significantly slower than CPython. When no fallbacks are required, speedups can be expected, especially if logical optimizations are applicable, but no substantial regressions. For whole-function fallback, performance does not change, as the function will be executed by CPython. Thus, only the case of frequent fallbacks could cause substantial regressions. Our evaluation in subsection 7.4 included two such cases, showing that performance with fallbacks is typically comparable to CPython.

Security. Depending on the use-case, naïvely embedding python code into data processing systems can have security implications, especially if those systems are used organization-wide. Sandboxing Python is hard, because arbitrary C code can be loaded as extensions and interpreter internals (e.g., gc, sys) are exposed. Thus, Python is usually either trusted, or sandboxed using containers or virtual machines. With our approach, more trade-offs between compatibility and security are possible. For fully secure but still performant execution, one could disable the automatic fallback and ensure that only permissible operations are generated. Although significantly reducing compatibility, many workloads can be supported without fallback, as illustrated previously. For more compatibility, a combination of only allowing trusted python libraries (e.g. numpy, pandas) and using Python's builtin `audit` mechanism is possible.

Further Optimizations. While our current set of optimizations is targeted at data science workloads, our approach is not limited to this domain and it is possible to add further transformations for improving, e.g., numerical workloads. Further, replacing our prototypical C++ back-end with a compiler infrastructure like MLIR [19] gives direct access to a wide range of existing optimizations from various domains.

9 Conclusion

This paper proposes a novel approach for extracting the high-level semantics from Python code by generating a program generator from it and executing it to generate a high-level, statically-typed IR. We successfully demonstrated with our end-to-end prototype HiPy that this IR can be leveraged to perform domain-specific, logical optimizations and generate efficient C++ code. For data-science workloads, we achieve significant single-threaded speedups, while HiPy is also capable of accelerating Python code in other domains. We hope that our focus on reaching full compatibility through the proposed fallback mechanism makes HiPy usable not only for further research like lifting declarative (sub-)operators from imperative IR operations, but also in practice.

Data-Availability Statement

Our implementation, HiPy, as well as the benchmarks and raw data from our evaluation (section 7) are available on Zenodo [16].

References

[1] [n. d.]. TorchScript. https://pytorch.org/docs/stable/jit.html. Accessed: 2024-03-27.

[2] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to Halide. *ACM Trans. Graph.* 38, 6 (2019), 204:1–204:13. https://doi.org/10.1145/3355089.3356549

[3] Joël Akeret, Lukas Gamper, Adam Amara, and Alexandre Réfrégier. 2015. HOPE: A Python just-in-time compiler for astrophysical computations. *Astron. Comput.* 10 (2015), 1–8. https://doi.org/10.1016/J.ASCOM.2014.12.001

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* 13, 2 (2011), 31–39. https://doi.org/10.1109/MCSE.2010.118

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[6] Christoph Brücke, Philipp Härtling, Rodrigo Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proc. VLDB Endow.* 16, 12 (2023), 3649–3661. https://doi.org/10.14778/3611540.3611554

[7] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. 2010. Language virtualization for heterogeneous parallel computing. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010).* 835–847. https://doi.org/10.1145/1869459.1869527

[8] João P. L. de Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José E. Moreira, and Guido Araujo. 2021. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *ACM Trans. Archit. Code Optim.* 18, 3 (2021), 38:1–38:22. https://doi.org/10.1145/3459010

[9] James M. Decker, Dan Moldovan, Andrew A. Johnson, Guannan Wei, Vritant Bhardwaj, Gregory Essertel, Fei Wang, Alexander B. Wiltschko, and Tiark Rompf. 2019. Snek: Overloading Python Semantics via Virtualization.

[10] Yannis E. Foufoulas, Alkis Simitsis, Eleftherios Stamatogiannakis, and Yannis E. Ioannidis. 2022. YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proc. VLDB Endow.* 15, 10 (2022), 2270–2283. https://doi.org/10.14778/3547305.3547328

[11] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compilercompiler. *Systems, computers, controls* 2, 5 (1971), 45–50.

[12] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow.* 15, 2 (2021), 196–210. https://doi.org/10.14778/3489496.3489501

[13] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper07.pdf

[14] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

[15] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986.* ACM Press, 86–96. https://doi.org/10.1145/512644.512652

[16] Michael Jungmair, Alexis Engelke, and Jana Giceva. 2024. *Artifact for "HiPy: Extracting High-Level Semantics From Python Code For Data Processing".* https://doi.org/10.5281/zenodo.13323059

[17] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* 711–726. https://doi.org/10.1145/2908080.2908117

[18] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC.* ACM, 7:1–7:6. https://doi.org/10.1145/2833157.2833162

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2021).* 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[20] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference (SciPy 2010).* scipy.org, 56–61. https://doi.org/10.25080/MAJORA-92BF1922-00A

[21] Modular Inc. 2024. Mojo. https://www.modular.com/mojo. Accessed: 2024-08-29.

[22] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019.* mlsys.org. https://proceedings.mlsys.org/book/272.pdf

[23] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[24] Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 112–124. https://doi.org/10.1109/ICSE43902.2021.00023

[25] Nuitka Team. 2023. Nuitka: A Python compiler written in Python. https://github.com/Nuitka/Nuitka.

[26] Oracle Labs. 2023. GraalPython: A Python 3 Implementation Built on GraalVM. https://www.graalvm.org/python/. Accessed: 2024-03-27.

[27] Shoumik Palkar, James Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analysis. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf

[28] Devin Petersohn, Dixin Tang, Rehan Sohail Durrani, Areg Melik-Adamyan, Joseph Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *Proc. VLDB Endow.* 15, 3 (2021), 739–751. https://doi.org/10.14778/3494124.3494152

[29] Pyston Team. 2023. Pyston: A faster and highly-compatible implementation of the Python programming language. https://github.com/pyston/pyston.

[30] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM '16).* Article 16. https://doi.org/10.1145/2949689.2949703

[31] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of Machine Learning and Systems.* mlsys.org, 638–651. https://proceedings.mlsys.org/paper/2022/hash/ca46c1b9512a7a8315fa3c5a946e8265-Abstract.html

[32] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06).* 944–953. https://doi.org/10.1145/1176617.1176753

[33] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130. https://doi.org/10.1145/2184319.2184345

[34] Hesam Shahrokhi, Callum Groeger, Yizhuo Yang, and Amir Shaikhha. 2023. Efficient Query Processing in Python Using Compilation. In *Companion of the 2023 International Conference on Management of Data*. ACM, 199–202. https://doi.org/10.1145/3555041.3589735

[35] Ariya Shajii, Gabriel Ramirez, Haris Smajlovic, Jessica Ray, Bonnie Berger, Saman P. Amarasinghe, and Ibrahim Numanagic. 2023. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC 2023)*. ACM, 191–202. https://doi.org/10.1145/3578360.3580275

[36] Phanwadee Sinthong and Michael J. Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *2019 IEEE International Conference on Big Data (IEEE BigData)*. 359–371. https://doi.org/10.1109/BIGDATA47090.2019.9006303

[37] Phanwadee Sinthong and Michael J. Carey. 2021. PolyFrame: A Retargetable Query-based Approach to Scaling Dataframes. *Proc. VLDB Endow.* 14, 11 (2021), 2296–2304. https://doi.org/10.14778/3476249.3476281

[38] Leonhard F. Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD '21: International Conference on Management of Data*. ACM, 1718–1731. https://doi.org/10.1145/3448016.3457244

[39] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

Received 2024-04-01; accepted 2024-08-18