Codon: A Compiler for High-Performance Pythonic Applications and DSLs

Ariya Shajii[*]
ariya@exaloop.io
Exaloop Inc.
Brookline, MA, USA

Gabriel Ramirez[*]
glram@mit.edu
MIT CSAIL
Cambridge, MA, USA

Haris Smajlović
hsmajlovic@uvic.ca
University of Victoria
Victoria, BC, Canada

Jessica Ray
jray@csail.mit.edu
MIT CSAIL
Cambridge, MA, USA

Bonnie Berger
bab@mit.edu
MIT CSAIL
Cambridge, MA, USA

Saman Amarasinghe[†]
saman@csail.mit.edu
MIT CSAIL
Cambridge, MA, USA

Ibrahim Numanagić[†]
inumanag@uvic.ca
University of Victoria
Victoria, BC, Canada

Abstract

Domain-specific languages (DSLs) are able to provide intuitive high-level abstractions that are easy to work with while attaining better performance than general-purpose languages. Yet, implementing new DSLs is a burdensome task. As a result, new DSLs are usually embedded in general-purpose languages. While low-level languages like C or C++ often provide better performance as a host than high-level languages like Python, high-level languages are becoming more prevalent in many domains due to their ease and flexibility. Here, we present Codon, a domain-extensible compiler and DSL framework for high-performance DSLs with Python's syntax and semantics. Codon builds on previous work on ahead-of-time type checking and compilation of Python programs and leverages a novel intermediate representation to easily incorporate domain-specific optimizations and analyses. We showcase and evaluate several compiler extensions and DSLs for Codon targeting various domains, including bioinformatics, secure multi-party computation, block-based data compression and parallel programming, showing that Codon DSLs can provide benefits of familiar high-level languages and achieve performance typically only seen with low-level languages, thus bridging the gap between performance and usability.

Keywords: domain-specific languages, type checking, Python, optimization, intermediate representation

[*]Both authors contributed equally to this research.
[†]Both authors share senior authorship.

CC '23, February 25–26, 2023, Montréal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0088-0/23/02.
https://doi.org/10.1145/3578360.3580275

1 Introduction

Domain-specific languages (DSLs) usually come in one of two varieties: *embedded* or *standalone*. Embedded DSLs are integrated within a general-purpose host language, whereas standalone DSLs introduce new languages with idiosyncratic syntax and semantics. Both varieties have pros and cons, but embedded DSLs enjoy wider popularity in practice due to ease of implementation and adoption. Consequently, designers of high-performance DSLs face a crucial choice at the onset: which language to embed their DSL in. In the past, the choice was obvious—high performance required embedding in a lower-level, statically analyzable language like C or C++. Since these languages were already commonly used in their respective domains, such DSLs frequently saw seamless adoption with few issues. Further, by virtue of their host languages, embedded DSLs inherited existing language and compiler infrastructure, providing overall end-to-end performance and general-purpose functionality. This approach was taken by many high-performance DSLs like CUDA [41], Halide [46] and Tiramisu [7], which derive from C++—the go-to language at the time for their target audiences.

Nowadays, the decision is no longer so simple. Dynamic languages like Python and Ruby, combined with the widespread availability of relatively high-performance domain-specific libraries [1, 24, 44, 45], have captured a large share of potential DSL users. Thus, building new DSLs on top of lower-level languages can, in fact, become a barrier to widespread adoption and even alienate a large fraction of the potential user base. Nonetheless, building optimized language features on top of dynamic languages like Python or Ruby that do not prioritize performance can be perilous: for that reason, popular high-performance libraries such as TensorFlow [1] or PyTorch [44] primarily use their host languages (e.g., Python) as an interface for interacting with optimized C/C++ libraries that do the heavy lifting. However, developing and debugging libraries in C/C++ is often hard and error-prone. Moreover, some domains make this approach infeasible, particularly when the data comprise billions of small objects directly defined and accessed by the

DSL, drastically increasing the cost of interfacing with an external library [52].

In this paper, we introduce Codon, a novel solution to bring high-performance DSLs to the Python user community by building a flexible development framework on top of an optimized Pythonic base. Codon is a full language and compiler that borrows Python 3's syntax and (with some limitations) semantics, but compiles to native machine code with low runtime overhead, allowing it to rival C/C++ in performance. To this end, Codon leverages ahead-of-time compilation, specialized bidirectional type checking, and a novel bidirectional intermediate representation (IR) to enable optional domain-specific extensions both in the language's syntax (front-end) and in compiler optimizations (back-end). These features allow not only new DSLs to be seamlessly built on top of the existing Codon framework but also enable different DSLs to be composed within a single program. Because Codon DSLs are built on top of a Pythonic base, they benefit from advantages specific to embedded DSLs; on the other hand, their ability to extend the syntax and ship custom compiler passes allows them to unlock features typically only accessible to standalone DSLs.

Codon was initially conceived as a limited Pythonic DSL for high-performance scientific computing but eventually evolved into a language that is highly compatible with Python 3—one of the most popular programming languages today [56]—in terms of syntax and semantics. As such, it enables programmers to write high-performance code in an intuitive, high-level and familiar manner. While Codon is not a drop-in replacement for Python as it intentionally omits some dynamic features (e.g., dynamic type manipulation and runtime reflection), we note that this dynamism often goes unused in many computing domains (such as scientific computing) and often hinders performance by preventing ahead-of-time compile-time optimizations and introducing unwanted behavior [6, 16]. Despite these decisions, Codon circumvents the hurdle of having to learn an entirely new language or ecosystem and still allows for substantial code reuse from existing Python programs. Unlike other performance-oriented Python implementations (such as PyPy [10] or Numba [3]), Codon is built from the ground up as a standalone system that compiles ahead-of-time to a static executable and is not tied to an existing Python runtime (e.g., CPython or RPython [4]) for execution. As a result, Codon can achieve better performance and overcome runtime-specific issues such as the global interpreter lock.

We demonstrate Codon's utility by showcasing several Codon-based, high-performance DSLs and extensions for various domains, including bioinformatics, secure multi-party computation, data compression, and parallel programming. Each of these extensions leverages Codon's compiler infrastructure—from the parser to type checker to intermediate representation—to implement a performant DSL or library that allows writing high-level Python code for the target domain which nevertheless achieves superior performance thanks to the various domain-specific optimizations and transformations included under the hood.

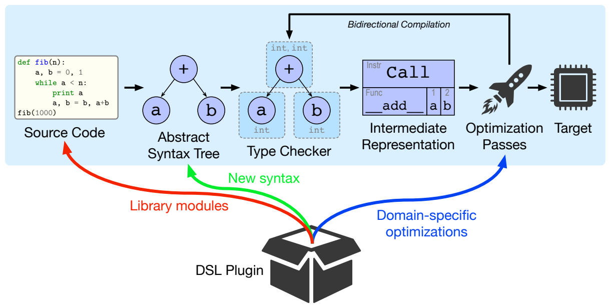
Overall, this paper makes the following contributions:

- *Bidirectional IRs.* We propose a new class of IRs called *bidirectional IRs*, with which compilation does not follow a linear path after parsing but can return to the type checking stage during IR passes to generate new specialized IR nodes. We demonstrate the utility of bidirectional IRs by using them to implement various optimizations and transformations for several domains.
- *Domain-extensible compiler.* We show how to construct a domain-extensible compiler via a plugin system, for which domain-specific extensions can be seamlessly implemented and composed in the context of a high-level, dynamic language (Python).
- *Framework for high-performance, Pythonic DSLs.* We implement and evaluate Codon, a new framework for creating Pythonic DSLs built on top of the previous two contributions. This framework enables the development of DSLs that share Python's syntax and semantics together with added domain-specific features and IR optimizations. Since Codon DSLs operate independently of the standard Python runtimes, they can achieve a performance comparable to C while being readily usable by anyone with a knowledge of Python.

2 Type Checking and Inference

Unlike Python, Codon utilizes static type checking and compiles to LLVM IR that does not use any runtime type information, similar to previous work on end-to-end type checking in the context of dynamic languages such as Python [4, 50, 52] and Ruby [40]. To that end, Codon ships with a static bidirectional type system, called LTS-DI, that utilizes Hindley-Milner (HM)-style inference to deduce the types in a program without requiring the user to manually annotate types (a practice that is, although supported, not widespread among the Python developers).

Due to the peculiarities of Python's syntax and common Pythonic idioms, LTS-DI makes adjustments to the standard HM-like inference to support notable Python constructs such as comprehensions, iterators, generators (both sending and receiving), complex function manipulation, variable arguments, static type checks (e.g., `isinstance` calls), and more. To handle these constructs and many others, LTS-DI relies on (1) monomorphization (instantiating a separate version of a function for each combination of input arguments), (2) localization (treating each function as an isolated type checking unit), and (3) delayed instantiation (function instantiations are delayed until all function parameters become known). Many Python constructs also necessitate compile-time expressions (akin to C++'s `constexpr` expressions), which Codon supports. More details about LTS-DI are available in Appendix A.



```
def fib(n):
    a, b = 0, 1
    while a < n:
        print a
        a, b = b, a+b
fib(1000)
```

Figure 1. Codon's compilation pipeline. Compilation proceeds at first in a linear fashion, where source code is parsed into an abstract syntax tree (AST), on which type checking is performed to generate an intermediate representation (IR). Unlike other compilation frameworks, however, Codon's is *bidirectional*, and IR optimizations can return to the type checking stage to generate new IR nodes and specializations not found in the original program, which is required for several key optimizations we present in this work. The framework is "domain-extensible", and a "DSL plugin" consists of library modules, syntax, and domain-specific optimizations.

While these methods are not uncommon in practice (e.g., monomorphization is used by C++'s templates, while delayed instantiaion has been used in the HMF type system [36]), we are not aware of their joint use in the context of type checking Python programs. Finally, note that Codon's type system in its current implementation is completely static and does not perform any runtime type deduction; as a result, some Python features, such as runtime polymorphism or runtime reflection, are not currently supported. In the context of scientific computing, we have found that dropping these features represents a reasonable compromise between utility and performance.

3 Intermediate Representation

Many languages compile in a relatively direct fashion: source code is parsed into an abstract syntax tree (AST), optimized, and converted into machine code, typically with the help of a compiler framework such as LLVM [34]. Although this approach is comparatively easy to implement, ASTs often contain many more node types than necessary to represent a given program. This complexity can make implementing optimizations, transformations, and analyses difficult or even impractical. An alternate approach involves converting the AST into an intermediate representation (IR) prior to performing optimization passes. IRs typically contain a substantially reduced set of nodes with well-defined semantics, making them much more conducive to transformations and optimizations.

Codon implements this approach in its IR, which is positioned between the type checking and optimization phases, as shown in Figure 1. The Codon Intermediate Representation (CIR) is radically simpler than the AST, with both a simpler structure and fewer node types (Appendix 4 and D). Despite this simplicity, CIR maintains most of the source's semantic information and facilitates "progressive lowering," enabling optimization at multiple levels of abstraction similar to other IRs [35, 59]. Optimizations that are more convenient at a given level of abstraction are able to proceed before further lowering.

3.1 High-Level Design

CIR is a value-based IR inspired in part by LLVM IR [34]. As in LLVM, we employ a structure similar to single static assignment (SSA) form, making a distinction between *values*, which are assigned at one location, and *variables*, which are conceptually similar to memory locations and can be modified repeatedly. To mirror the source's structure, values can be nested into arbitrarily-large trees. Keeping this SSA-like tree structure enables easy lowering at the Codon IR level. For example, this structure enables CIR to be lowered to a control flow graph easily. Unlike LLVM, however, CIR initially represents control flow using explicit nodes called *flows*, allowing for a close structural correspondence with the source code. Explicitly representing the control flow hierarchy is similar to the approaches taken by Suif [59] and Taichi [26]. Importantly, this makes optimizations and transformations that depend on precise notions of control flow much easier to implement. A simple example is a `for` flow that keeps explicit loops in CIR and allows Codon to easily recognize patterns such as the common `for x in range(y)` loop instead of having to decipher a maze of branches, as is done in lower-level IRs like LLVM IR. An example of source code mapping to CIR is shown in Figure 2.

3.2 Operators

CIR does not represent operators like + explicitly but instead converts them to corresponding function calls (also known as "magic methods" in the Python world). For example, the + operator resolves to an `__add__` call (Figure 2). This enables seamless operator overloading for arbitrary types via magic methods, the semantics of which are identical to Python's.

A natural question that arises from this approach is how to implement operators for primitive types like `int` and `float`. Codon solves this by allowing inline LLVM IR via the `@llvm` function annotation, which enables all primitive operators to be written in Codon source code. Information about operator properties like commutativity and associativity can be passed as annotations in the IR.

3.3 Bidirectional IRs

Traditional compilation pipelines are linear in their data flow: source code is parsed into an AST, usually converted to an IR, optimized, and finally converted to machine code. With Codon, we introduce the concept of a *bidirectional IR*, wherein IR passes are able to return to Codon's type checking stage to generate new IR nodes and specializations not present in the source program. Among the benefits of a bidirectional IR are:

- *Large portions of complex IR transformations can be implemented directly in Codon.* For example, the prefetch optimization mentioned in Section 4.3 involves a generic dynamic coroutine scheduler that is impractical to implement purely in Codon IR.
- *New instantiations of user-defined data types can be generated on demand.* For example, an optimization that requires the use of Codon/Python dictionaries can instantiate the `Dict` type for the appropriate key and value types. Instantiating types or functions is a non-trivial process that requires a full re-invocation of the type checker due to cascading realizations, specializations and so on.
- *The IR can take full advantage of Codon’s high-level type system.* By the same token, IR passes can themselves be generic, using Codon’s expressive type system to operate on a variety of types.

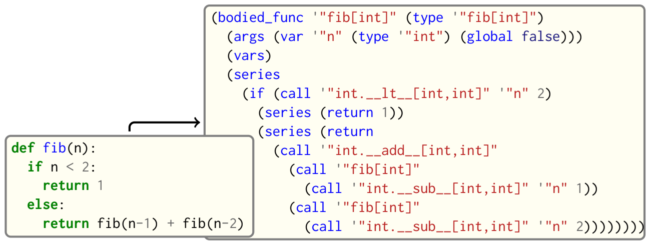
While CIR’s type system is very similar to Codon’s, CIR types are fully realized and have no associated generics (unlike Codon/AST types). However, every CIR type carries a reference to the AST types used to generate it, along with any AST generic type parameters. These associated AST types are used when re-invoking the type checker and allow CIR types to be queried for their underlying generics, even though generics are not present in the CIR type system (e.g. it is straightforward to obtain the type `T` from a given CIR type representing `List[T]`, and even use it to realize new types or functions). Note that CIR types correspond to high-level Codon types; LLVM IR types are more low-level and do not map back directly to Codon types.

The ability to instantiate new types during CIR passes is, in fact, critical to many CIR operations. For example, creating a tuple `(x, y)` from given CIR values `x` and `y` requires instantiating a new tuple type `Tuple[X,Y]` (where the uppercase identifiers indicate types), which in turn requires instantiating new tuple operators for equality and inequality checking, iteration, hashing and so on. Calling back to the type checker makes this a seamless process, however.

We demonstrate the power of bidirectional IRs by implementing several real-world domain-specific optimizations in Section 4, each of which relies on the bidirectional IR features listed above.

3.4 Passes and Transformations

CIR provides a comprehensive analysis and transformation infrastructure: users write passes using various CIR built-in utility classes and register them with a `PassManager`, which is responsible for scheduling execution and ensuring that any required analyses are present. In Figure 13, we show a simple addition constant folding optimization that utilizes the `OperatorPass` helper, a utility pass that visits each node



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

```
(bodied_func '"fib[int]" (type '"fib[int]")
  (args (var '"n" (type '"int") (global false)))
  (vars)
  (series
    (if (call '"int.__lt__[int,int]" '"n" 2)
      (series (return 1))
      (series (return
        (call '"int.__add__[int,int]"
          (call '"fib[int]"
            (call '"int.__sub__[int,int]" '"n" 1))
          (call '"fib[int]"
            (call '"int.__sub__[int,int]" '"n" 2))))))))
```

Figure 2. Example of Codon source mapping for a simple Fibonacci function into CIR. The function `fib` maps to a CIR `BodiedFunc` with a single integer argument. The body contains an `IfFlow` that either returns a constant or recursively calls the function to obtain the result. Notice that operators like + are converted to function calls (e.g., `__add__`), but that the IR otherwise mirrors the original source code in its structure, allowing easy pattern matching and transformations.

in an IR module automatically. In this case, we simply override the handler for `CallInstr`, check to see if the function matches the criteria for replacement and perform the action if so (recall that binary operators in CIR are expressed as function calls). Users can also define their own traversal schemes and modify the IR structure at will.

More complex passes can make use of CIR’s bidirectionality and re-invoke the type checker to obtain new CIR types, functions, and methods, an example of which is shown in Figure 3. In this example, calls of the function `foo` are searched for, and a call to `validate` on `foo`’s argument and its output is inserted after each. As both functions are generic, the type checker is re-invoked to generate three new, unique `validate` instantiations. Instantiating new types and functions requires handling possible specializations and realizing other nodes (e.g., the == operator method—`__eq__`—must be realized in the process of realizing `validate` in the example), as well as caching realizations for later use.

3.5 Code Generation and Execution

Codon uses LLVM to generate native code. The conversion from Codon IR to LLVM IR is generally a straightforward process, following the mappings listed in Table 4. Most Codon types also translate to LLVM IR types intuitively: `int` becomes `i64`, `float` becomes `double`, `bool` becomes `i8` and so on—these conversions also allow for C/C++ interoperability. Tuple types are converted to structure types containing the appropriate element types, which are passed by value (recall that tuples are immutable in Python); this approach for handling tuples allows LLVM to optimize them out entirely in most cases. Reference types like `List`, `Dict` etc. are implemented as dynamically-allocated objects that are passed by reference, which follows Python’s semantics for mutable types. Codon handles `None` values by promoting types to `Optional` as necessary; optional types are implemented via a tuple of LLVM’s `i1` type and the underlying type, where the former indicates whether the optional contains a value.

```
def foo(x): return x*3 + x
def validate(x, y):
    assert y == x*4
a = foo(10)
b = foo(1.5)
c = foo('a')
```

```
a = foo(10)
validate(10, a)
b = foo(1.5)
validate(1.5, b)
c = foo('a')
validate('a', c)
```

```
class ValidateFoo : public OperatorPass {
  void handle(AssignInstr *v) {
    auto *M = v->getModule();
    auto *var = v->getLhs();
    auto *call = cast<CallInstr>(v->getRhs());
    if (!call) return;
    auto *foo = util::getFunc(call->getCallee());
    if (!foo || foo->getUnmangledName() != "foo") return;
    auto *arg1 = call->front(); // argument of 'foo' call
    auto *arg2 = M->Nr<VarValue>(var); // result of 'foo' call
    auto *validate = M->getOrRealizeFunc("validate",
                       {arg1->getType(), arg2->getType()});
    auto *validateCall = util::call(validate, {arg1, arg2});
    insertAfter(validateCall); // call 'validate' after 'foo'
  }
};
```

Figure 3. Example of bidirectional compilation in Codon IR. The simple pass, shown in the bottom box, searches for calls of function `foo`, and inserts after each a call to `validate`, which takes `foo`'s argument as well as its output and verifies the result. Both functions are generic and can take as an argument any type that can be multiplied by an integer, so the type checker is re-invoked to generate three distinct `validate` instantiations for the example code in the top left box, producing code equivalent to that in the top right box.

Optionals on reference types are specialized to use a null pointer to indicate a missing value.

Generators are a prevalent language construct in Python; in fact, every `for` loop iterates over a generator (e.g. `for i in range(10)` iterates over the `range(10)` generator). Hence, it is critical that generators in Codon carry no extra overhead and compile to equivalent code as standard C `for`-loops whenever possible. To this end, Codon uses LLVM coroutines [42] to implement generators. LLVM's coroutine passes elide all coroutine overhead (such as frame allocation) and inline the coroutine iteration whenever the coroutine is created and destroyed in the same function. (We found in testing that the original LLVM coroutine passes—which rely on explicit "create" and "destroy" intrinsics—were too conservative when deciding to elide coroutines generated by Codon, so in Codon's LLVM fork this process is replaced with a capture analysis of the coroutine handle, which is able to elide coroutine overhead in nearly all real-world cases.)

Codon uses a small runtime library when executing code. In particular, the Boehm garbage collector [9]—a drop-in replacement for `malloc`—is used to manage allocated memory[1], and OpenMP for handling parallelism. Codon offers two compilation modes: *debug* and *release*. Debug mode includes full debugging information, allowing Codon programs to be debugged with tools like GDB and LLDB, and also includes full backtrace information with file names and line numbers. Release mode performs a greater number of optimizations (including `-O3` optimizations from GCC/Clang) and omits some safety and debug information. Users can therefore use the debug mode for quick programming and debugging cycle and the release mode for high-performance deployment.

3.6 Extensibility

Due to the framework's flexibility and bidirectional IR, as well as the overall expressiveness of Python's syntax, a large fraction of the DSL implementation effort can be deferred to the Codon source. Indeed, as shown in Section 4, Codon applications typically implement large fractions of their domain-specific components in the source itself. This has the benefit of making Codon DSLs intrinsically interoperable—so long as their standard libraries compile, disparate DSLs can be used together seamlessly. Along these lines, we propose a modular approach for incorporating new IR passes and syntax, which can be packaged as dynamic libraries and Codon source files. At compile time, the Codon compiler can load the plugin, registering the DSL's elements.

Some frameworks, such as MLIR [35], allow customization for all facets of the IR. Codon IR, on the other hand, restrict customization of the IR to a few types of nodes, and relies on bidirectionality for further flexibility. In particular, CIR allows users to derive from "custom" types, flows, constants, and instructions, which interact with the rest of the framework through a declarative interface. For example, custom nodes derive from the appropriate custom base class (`CustomType`, `CustomFlow`, etc.) and expose a "builder" to construct the corresponding LLVM IR (see Appendix 9 a brief example which implements a 32-bit float type). Notice that implementing custom types (and custom nodes in general) involves defining a `Builder` that specifies LLVM IR generation via virtual methods (e.g. `buildType` and `buildDebugType`); the custom type class itself defines a method `getBuilder` to obtain an instance of this builder. This standardization of nodes enables DSL constructs to work seamlessly with existing passes and analyses.

4 Applications

4.1 Microbenchmark Performance

Given Codon's roots in Python, it can substantially accelerate many standard Python programs out of the box thanks to AOT compilation. Codon IR makes it easy to optimize several patterns commonly found in Python code, such as dictionary updates (that can be optimized to use a single lookup instead of two; Figure 4), or consecutive string additions (that can be folded into a single join to reduce allocation overhead). Here we provide a few benchmarks that show the extent of Codon's performance improvements.

Figure 5 shows Codon's runtime performance, as well as that of CPython (v3.10) and PyPy (v7.3), on the pybenchmark benchmark suite[2] restricted to a set of "core" benchmarks that do not rely on external libraries such as Django

[1]Note that Codon DSLs can use different memory management solutions.

[2]https://github.com/duboviy/pybenchmark

```
d = {'a': 42}
d['a'] = d.get('a', 0) + 1
```

↓

```
d = {'a': 42}
d.__dict_do_op__('a', 1, 0, int.__add__)
```

Figure 4. Example of dictionary optimization. The pass recognizes the *get/set* pattern and replaces it with a single call to `__dict_do_op__`. As this function is generic, we instantiate a new version and pass the `int.__add__` function as a parameter. This optimization results in a 12% performance improvement on a `wordcount` benchmark.

or DocUtils.[3] Codon is always faster, sometimes by orders of magnitude when compared to CPython and PyPy. For some benchmarks, we also provided corresponding C++ implementations and observed that Codon provides similar—if not improved—performance. More information about these benchmarks is given in Appendix B; prior work that uses Codon ([52]) contains more extensive performance measurements on a broader set of benchmarks, not only against CPython and PyPy, but also against other Python implementations such as Nuitka [25] and Shed Skin [21]. As those results follow the exact same trends as reported here, we omit them for brevity.

While microbenchmarks are a decent proxy for performance, they are not without drawbacks and often do not tell the whole story. For that reason, the rest of this section focuses on the practical, real-world applications and DSLs that utilize Codon to enable users to write simple Python code for various domains, and yet deliver high performance on real applications and datasets. Note that these DSLs can be embedded directly within an existing Python codebase with the appropriate decorator provided by Codon.

4.2 OpenMP: task- and loop-parallelism

Because Codon is built from the ground up independently of the existing Python runtimes, it does not suffer from CPython's infamous *global interpreter lock* and can therefore take full advantage of multi-threading. To support parallel programming, we implemented a Codon extension that allows end-users to use OpenMP within Codon itself. An example OpenMP program in C++ and Codon is shown in Figure 6, exhibiting Codon's ability to extend the syntax of the base Python language and its ability to implement complex transformations needed to interface with OpenMP.

OpenMP predominately leverages *outlining* to parallelize code—in general, a parallel loop's body is outlined into a new function, which is then called by multiple threads by the OpenMP runtime. For example, the body of the loop from Figure 6 would be outlined to a function `f` that takes as parameters the variables `a`, `b`, `c` and the loop variable `i`.

[3]While Codon can use existing Python libraries through a CPython bridge, their performance will be equal to that of CPython.

Then, a call to `f` would be inserted into a new function `g` that invokes OpenMP's dynamic loop scheduling routines for a chunk size of 10. Finally, `g` would be called by all threads in the team via OpenMP's `fork_call` function. The result is shown in the right snippet of Figure 6 (note that for simplicity, this code omits a few details like the loop schedule code and thread or location identifiers). Our passes also take special care to handle private variables (e.g., local to the outlined function), as well as shared ones (details omitted for brevity). Reductions over variables also require additional code generation for atomic operations (or the use of locks), as well as an additional layer of OpenMP API calls.

The bidirectional compilation is a critical component of Codon's OpenMP pass. The various loop "templates" (e.g., dynamic loop scheduling routines in the example above, or static and task-based loop routines) are implemented in high-level Codon source code. Following the code analysis, the reduction pass copies and specializes these "templates" by filling in the loop body, chunk size and schedule, rewriting expressions that rely on shared variables, and more. This design tremendously simplifies the pass implementation and adds a degree of generality (e.g., it is easy to implement new templates and strategies directly in Codon for new types of loops without having to redesign the pass itself). Unlike Clang or GCC, Codon's OpenMP pass deduces which variables are shared, which are private, as well as any reductions taking place (e.g. `a += i` within the loop body would generate code for a +-reduction on `a`). Custom reductions can be implemented simply by providing an appropriate atomic magic method (e.g. `__atomic_add__`) on the reduction type. Codon also employs several lowering passes that lower certain `for`-loops that iterate over a generator (the default behaviour of Python loops) to "imperative loops"—C-style loops with a start, stop, and step values. For example, `for i in range(N)` will be lowered to an imperative loop with start index `0`, stop index `N`, and step `1`; iterations over lists will also be lowered to imperative loops. Imperative loops are, in turn, converted into OpenMP parallel loops if the `@par` tag is present. Non-imperative parallel loops are parallelized by spawning a new OpenMP task for each loop iteration and placing a synchronization point after the loop. This scheme allows all Python `for`-loops to be parallelized.

OpenMP transformations are implemented as a set of CIR passes that match the for loops marked by the `@par` attribute (a syntactic extension provided by the Codon parser) and transform such loops into the appropriate OpenMP construct within CIR. Nearly all OpenMP constructs were implemented as higher-order functions in Codon itself.

4.3 Seq: a DSL for Bioinformatics

Seq [52] is a complete DSL for bioinformatics and computational genomics and the original motivation for Codon itself. On top of standard Python, Seq adds new data types for "sequences" (DNA strings comprised of `ACGT` characters)

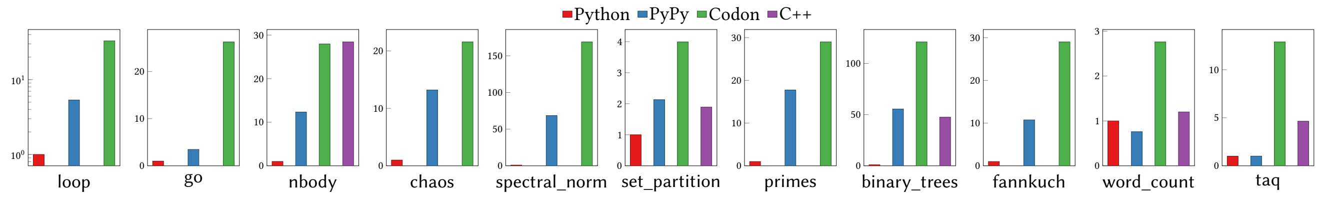


Figure 5. Comparison of Python (CPython 3), PyPy, Codon and C++ (where applicable) on several benchmarks from Python’s benchmark suite (`pyperformance`). The y-axis shows the speedup (in times) over the CPython implementation.

```
// C++
#pragma omp parallel for schedule(dynamic, 10) num_threads(8)
for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i]
```

```
# Codon
@par(schedule='dynamic', chunk_size=10, num_threads=8)
for i in range(N):
    c[i] = a[i] + b[i]
```

↓

```
import openmp as omp
def f(a, b, c, i):
    c[i] = a[i] + b[i]
def g(loop, a, b, c):
    omp._dynamic_init(loop=loop, chunk=10)
    while True:
        more, subloop = omp._dynamic_next(loop)
        if not more: break
        for i in subloop: f(a, b, c, i)
omp._push_num_threads(8)
omp._fork_call(g, range(N), a, b, c)
```

Figure 6. OpenMP program in C++ and in Codon. The result of applying Codon’s OpenMP pass and lowering the parallel loop is given in the bottom snippet.

and “k-mers” (fixed length-k sequences), as well as numerous library features packaged in a new `bio` module. Finally, Seq leverages Codon IR to perform several performance-critical domain-specific optimizations. Importantly, bioinformatics and genomics have distinct computational characteristics that make interfacing with a high-performance library in plain Python impractical—in particular, billions of distinct sequences and k-mers are often processed concurrently by an application, so any per-object overhead imposed by the Python’s runtime (either CPython or RPython [4]) is detrimental to the overall performance. Since Codon avoids the high overhead of the existing Python implementations, it is a natural choice for implementing a genomics DSL.

Many genomics applications can be conceptualized as a series of stages through which data is passed. To that end, Seq uses Codon’s “pipeline” syntax—added on top of Python’s base syntax and denoted with the `|>` operator—to concisely express many genomics operations. For example, the pipeline shown at the bottom of Figure 7 reads sequences from a FASTQ file (standard format for storing sequencing data), partitions each into length-k subsequences, and process each subsequence using the `search` function, the output of which is passed to some function `process`. Pipeline stages can be plain functions, in which case the function is simply applied to the whole input to produce the output; stages can also be generators, in which case all values produced by the generator are lazily passed to the rest of the pipeline. Additionally, the “parallel” pipe operator `||>` indicates that all subsequent stages can be executed in parallel.

Since pipelines are a central component of Seq, and the way in which the main loop of most applications is expressed, Seq implements several optimizations on them through Codon IR passes. One example pertains to expensive genomic index queries: the `search` function in Figure 7 queries a sequence in an FM-index data structure, which entails repeatedly updating an FM-index interval by randomly accessing an auxiliary array, incurring numerous cache misses (note that genomic indices are often very large exceeding tens of gigabytes or more). The `@prefetch` annotation instructs the compiler to perform pipeline optimizations to overlap the function’s cache misses with other useful work by converting `search` to a coroutine that issues a software prefetch and yields just before the expensive memory access, in conjunction with a dynamic scheduler of coroutines, written in Codon itself (Appendix Figure 11), that manages several instances and dynamically switches between them as appropriate. The prefetch optimization is implemented as a Codon IR pass in roughly 100 lines of code. The two necessary transformations for this optimization are (1) converting the annotated function to a coroutine and (2) inserting the dynamic scheduler in the pipeline. The scheduler is implemented in Codon as a generic function and appropriately instantiated during the IR pass by re-invoking the bidirectional type checker.

```
import sys, bio, bio.fmindex
@prefetch
def search(s, index):
    # initialize interval
    intv = index.interval(s[-1])
    s = s[:-1]  # trim last base
    while s and intv:
        # extend interval
        intv = index[intv, s[-1]]
        s = s[:-1]  # trim last base
    return intv
# create the index
idx = bio.fmindex.FMIndex(argv[1])
( bio.FASTQ(argv[2])
  |> bio.seqs  |> bio.split(k=20)
  |> search(idx)  |> process )
```

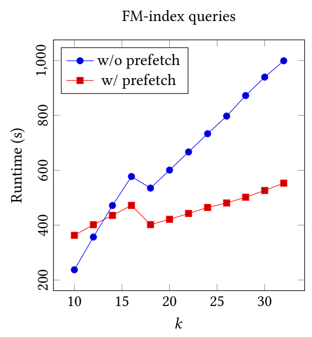


Figure 7. Example of a prefetch optimization on a FM-index, together with performance results—with and without `@prefetch` annotation—for various k. More experiments are available in Appendix B.

To illustrate the effect of this optimization, Figure 7 shows the runtime of the FM-index query code with and without the `@prefetch` annotation, for various subsequence lengths (`k` in the code). For shorter sequences, less of the FM-index is accessed, so cache misses are less likely (note that querying an FM-index is conceptually equivalent to walking down a tree—in this case, the "head" of the tree is cache-resident). For longer sequences, more cache misses are incurred, leading to a substantial improvement with the prefetch optimization, up to about 2× in this example. Note that even the version that does not use prefetch optimization already equals C implementation of FM-index used in other tools [52], showing that even "raw" Codon can be used for high-performance tasks before being augmented by large-scale complex code transformations that can enable surpassing even the state-of-the-art C/C++ implementations without necessitating major code refactoring. The details are available in Appendix B; the real-world demonstration of prefetch, as well as other bioinformatics-specific Codon IR passes, is available in [53]. For example, the Seq version of BWA MEM [39] sequence alignment tool was 2× faster than the highly optimized C version and up to 4× shorter.

4.4 Sequre: a DSL for Multi-Party Computation

Secure multi-party computation (MPC) [19] is a strategy for securely performing computations over multiple computing parties in a distributed manner on shared data without actually revealing the data itself to the parties. To satisfy the security guarantees, MPC relies on specialized algebraic operations and, as such, requires a complete re-implementation of all core operations (arithmetic, matrix manipulation, communication primitives, etc.) in a secure and distributed manner. Under such guarantees, common arithmetic operations (such as multiplication) are significantly slower (more than 100× on average [47]) compared to their standard non-secure counterparts. A naïve library implementation of MPC protocols (e.g. through operator overloading [51]) considers each operation independently and thus misses on efficient compile-time optimization opportunities such as secure loop unrolling or network-latency-based prefetching [31]. For that reason, the existing MPC applications require careful bookkeeping and manual optimizations to achieve satisfactory performance [18] and do not generalize well beyond their original use-case.

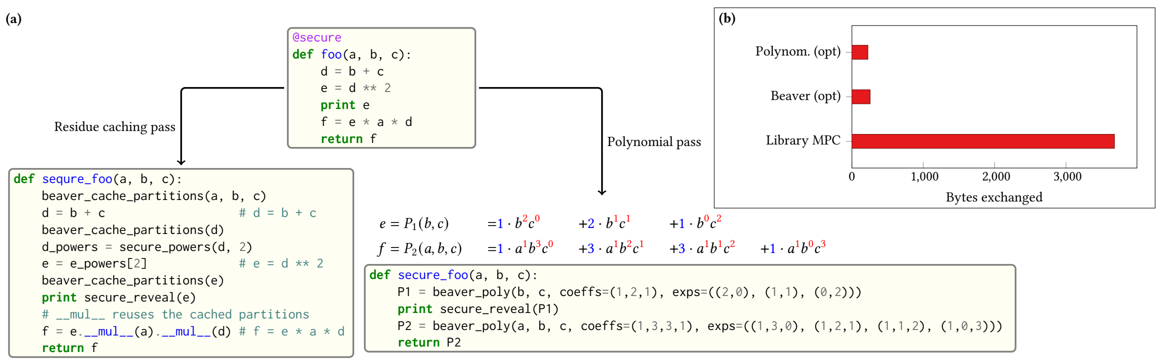
We used Codon to implement a high-level DSL—called Sequre [55]—for rapid development ofsecure MPC pipelines. The implementation of standard MPC primitives and secure core operations is done in Codon itself. Codon IR and its bidirectionality is used to implement *residue caching* and *polynomial scaffolding* optimizations [18] in an automatic fashion at the compile time as follows.

Sequre initially transforms all arithmetic operators to their secure multiparty counterparts. While doing so, it rearranges the expressions into the shapes suitable for MPC-friendly order of execution (an order that might not be optimal in a non-secure environment). For example, in the polynomial scaffolding pass, a series of arithmetic expressions are reshaped into a generalized polynomial form and then relayed to an efficient secure polynomial evaluator that minimizes the network communication between the MPC parties (Figure 8). To that end, Codon IR is used to detect such expressions and rearrange them into an MPC-optimal shape, while the secure polynomial evaluator is implemented in Codon itself. Sequre thus allows end-users to write the idiomatic Python code without worrying about MPC specifics to implement performant protocols that automatically minimize the network utilization of the essential arithmetic operations.

Residue caching is another MPC-specific optimization that Sequre implements. Each secure multiplication requires the operands to be transformed into randomized tuples—*multiplicative residues* commonly known as *Beaver partitions* [18]—before computing the final product. These residues can be cached and reused for subsequent multiplications because obtaining them requires communication between the computing parties and is thus expensive. Furthermore, some operations can reuse the existing operand residues to cheaply construct the final residue (e.g., the residue of a sum can be obtained by simply adding summand residues). Unfortunately, computing, propagating, and reusing of the residues are in almost all cases done manually, unavoidably resulting in complex implementations that are hard to understand and review and even harder to extend. We used Codon IR to automate residue caching by analyzing the binary expression tree of the expressions and labelling variables reused either directly or through propagation for residue caching.

Together with the efficient MPC primitive library implemented in Codon itself, residue caching and polynomial scaffolding significantly reduced the network communication between the computing parties. In the example shown in Figure 8, Sequre exchanges 3,680 bytes over the network without the two IR transformations enabled. Enabling the transformations decreases this amount up to 10× (256 bytes for residue optimization and 224 bytes for polynomial optimization). Finally, we note that additional optimizations (e.g., using an MPC-specific modulo operator that is inserted by the Codon IR where needed) allowed us to produce high-level Sequre code for genome-wide association studies, achieving up to 4× speed-ups with 7× reductions in code size over the existing C++ state-of-the-art [18, 55]. In concrete terms, a high-level Python implementation of GWAS protocol that can be easily reviewed for security takes 20 days to complete as compared to the optimized 80-day long C++ pipeline.

Both Seq and Sequre are Codon plugins that contain a library of core primitives written in Codon, and a set of CIR and LLVM passes that operate within the functions marked with plugin-specific attributes (e.g., `prefetch` or `secure`).



```
@secure
def foo(a, b, c):
    d = b + c
    e = d ** 2
    print e
    f = e * a * d
    return f
```

```
def sequre_foo(a, b, c):
    beaver_cache_partitions(a, b, c)
    d = b + c                         # d = b + c
    beaver_cache_partitions(d)
    d_powers = secure_powers(d, 2)
    e = e_powers[2]                   # e = d ** 2
    beaver_cache_partitions(e)
    print secure_reveal(e)
    # __mul__ reuses the cached partitions
    f = e.__mul__(a).__mul__(d) # f = e * a * d
    return f
```

$$e = P_1(b,c) = 1 \cdot b^2c^0 + 2 \cdot b^1c^1 + 1 \cdot b^0c^2$$

$$f = P_2(a,b,c) = 1 \cdot a^1b^3c^0 + 3 \cdot a^1b^2c^1 + 3 \cdot a^1b^1c^2 + 1 \cdot a^1b^0c^3$$

```
def secure_foo(a, b, c):
    P1 = beaver_poly(b, c, coeffs=(1,2,1), exps=((2,0), (1,1), (0,2)))
    print secure_reveal(P1)
    P2 = beaver_poly(a, b, c, coeffs=(1,3,3,1), exps=((1,3,0), (1,2,1), (1,1,2), (1,0,3)))
    return P2
```

Figure 8. (a) A simple example of how Sequre leverages Codon's bidirectional IR in `foo` function decorated with `@secure` attribute to optimize the enclosed arithmetic expressions via either the residue caching pass (left) or the polynomial pass (right). (b) Total network utilization for standard MPC library ("Library MPC") and Sequre's two optimized solutions ("Beaver" and "Polynomial") in terms of exchanged bytes between computing parties on a degree-30 polynomial.

4.5 CoLa: a DSL for Block-Based Compression

CoLa[4] is a Codon-based DSL targeting block-based data compression, which forms the core of many common image and video compression algorithms in use today. These types of compression rely heavily on partitioning regions of pixels into a series of smaller and smaller blocks, forming a multi-dimensional data hierarchy where each block needs to know its location relative to the other blocks. For example, one stage of H.264 video compression partitions an input frame of pixels into a series of 16x16 blocks of pixels, partitions each of those into 8x8 blocks of pixels, and then partitions those into 4x4 blocks of pixels. Tracking the locations between these individual blocks of pixels requires significant book-keeping information, which quickly obscures the underlying algorithms in existing implementations.

CoLa introduces the Hierarchical Multi-Dimensional Array (HMDA) abstraction, which simplifies the representation and use of hierarchical data. HMDAs represent multi-dimensional arrays with a notion of location, which tracks the origin of any given HMDA relative to some global co-ordinate system. HMDAs also track their dimensions and strides. With these three pieces of data, any HMDA can determine its location relative to any other HMDA at any point in the program. CoLa implements the HMDA abstraction in Codon as a library centered around two new data types: the `block` and `view`. `block`s create and own an underlying multi-dimensional array, while `view`s point to a particular region of a `block`. CoLa exposes two main hierarchy-construction operations, location copy and partition, which create `block`s and `view`s, respectively. CoLa supports standard indexing with integer and slice indices but also introduces two unique indexing schemes, which model how compression standards describe data access. "Out-of-bounds" indexing allows users to access data that surrounds a `view`, and "colocation" indexing lets users index an HMDA with another HMDA.

While the combination of Codon's Pythonic features and CoLa's abstractions provides users the benefits of a high-level language and compression-specific abstractions, the HMDA abstraction incurs significant run-time overhead due to the extra indexing operations required. For compression, many HMDA accesses occur at the innermost levels of computation, so any extra computations on top of accessing the raw array prove detrimental to run-time. CoLa utilizes Codon's IR and pass framework to implement hierarchy collapsing and location propagation passes. Hierarchy collapsing reduces the number of intermediate `view`s created, and location propagation attempts to infer the location of any given HMDA. Together, these passes decrease the overall size of the hierarchy and simplify the actual indexing computations. Without these optimizations, CoLa runs 48.8×, 6.7×, and 20.5× slower on average compared to reference C codes for JPEG [27], JPEG lossless [49], and H.264 [28], respectively. With the optimizations, performance reaches parity, with average run-times of 1.06×, 0.67×, and 0.91× relative to the same reference codes.

CoLa is implemented as a Codon plugin, and as such, ships with a library of compression primitives written in Codon itself and a set of CIR and LLVM passes that operate on hierarchical data structures defined by CoLa and optimize their creation and access routines. CoLa also uses Codon to provide custom data structure access syntax and operators that simplify common indexing and reduction operations.

5 Related Work

Many frameworks for implementing DSLs rely on embedding into another language and extending its features through metaprogramming techniques. Delite framework [12] utilizes Scala's rich operator overloading support to implement custom syntax, represents applications as valid Scala code, and constructs an intermediate representation at runtime; DSL optimizations can manipulate its IR through "rewrite rules" and "traversals" before final compilation. BuildIt [11]

[4]Currently under submission [5]

uses C++ for code generation. AnyDSL [38] takes a similar approach of embedding in a new language called Impala. Other approaches include Racket-based Sham [58] that embed in dynamic languages and emit IR, and extensible compilers such as ableC [29] that allows extending C with new syntactic and semantic constructs, and JastAdd [22] that does the same with Java. Some languages, such as Julia [13], also use static compilation and elaborate type systems and allow end-users to extend the core language through macros in limited fashion. While these frameworks allow considerable DSL customization, they are constrained by their parent languages in terms of syntax familiarity (e.g., unfamiliar or uncommon syntax) and typing flexibility (e.g., being limited by the host language typing system). By contrast, Codon attempts to build performant DSLs with the ahead-of-time compilation on top of an existing, widely-used dynamic language, which comes with its own unique challenges that are not directly addressed by prior work.

Codon is modelled upon the Seq language [52], a DSL for bioinformatics itself inspired by various other DSLs [2, 8, 15, 17, 32, 33, 46, 61]. Seq was originally designed as a Pythonic DSL with many advantages such as ease of adoption, excellent performance, and expressiveness. However, it did not support many common Python language constructs due to rigid typing rules, nor did it offer a mechanism to easily implement new compiler optimizations. By applying a bidirectional IR and improved type checker, Codon builds substantially on Seq's foundation and offers a general solution to these issues. In particular, Codon covers a *much* larger portion of Python's features, and provides a framework for implementing domain-specific optimizations. Furthermore, Codon offers a flexible type system that better handles various Pythonic idioms. Similar work on type systems for Pythonic codebases includes RPython [4] and the related PyPy [10], linters such as Pyright/Pylance, Mypy [37] and Pytype, and static type systems such as Starkiller [50] and others [14]. Many of these ideas have also been applied in the context of other dynamic languages, such as InferDL [30], Hummingbird [48] and PRuby [23] (Ruby) and TypeScript. Finally, we note that the back-and-forth approach used by Codon's bidirectional IR shares similarities with the previous work on pluggable type systems [20] (an example of such approach is Checker framework [43] for Java).

While the Codon Intermediate Representation is not the first customizable IR, it differs from frameworks like MLIR [35] in its approach. Rather than supporting customization of *everything*, Codon's IR opts for a simple, clearly-defined customizations that can be combined with bidirectionality for more complex features. In terms of structure, CIR takes inspiration from LLVM [34] and Rust's IRs [57]. These IRs benefit from a vastly simplified node set and structure, which in turn simplifies the implementation of IR passes. Structurally, however, these representations are too low-level to effectively express Pythonic optimizations. In particular, they radically restructure the source code, eliminating semantic information that must be reconstructed to perform transformations. To address this shortcoming, many IRs like Taichi [26] and Suif [59] adopt hierarchical structures that maintain control flow and semantic information, albeit at the cost of increased complexity. Unlike Codon's, however, these IRs are largely disconnected from their languages' front-ends, making maintaining type correctness and generating new code impractical or even impossible. Therefore, CIR takes the best of these approaches by utilizing a *simplified* hierarchical structure, maintaining both the source's control flow nodes and a radically decreased subset of internal nodes. Importantly, it augments this structure with bidirectionality, making new IR easy to generate and manipulate.

6 Conclusion

We have introduced Codon, a domain-configurable framework for designing and rapidly implementing Pythonic DSLs. By applying a specialized type checking algorithm and novel bidirectional IR, Codon enables easy optimization of dynamic code in a variety of domains. Codon DSLs achieve considerable performance benefits over Python and can match C/C++ performance without compromising high-level simplicity. We note that Codon is already being used commercially in quantitative finance and bioinformatics.

Currently, there are several Python features that Codon does not support. They mainly consist of runtime polymorphism, runtime reflection and type manipulation (e.g., dynamic method table modification, dynamic addition of class members, metaclasses, and class decorators). There are also gaps in the standard Python library coverage. While Codon ships with Python interoperability as a workaround to some of these limitations, future work is planned to expand the amount of Pythonic code immediately compatible with the framework by adding features such as runtime polymorphism and by implementing better interoperability with the existing Python libraries. Finally, we plan to increase the standard library coverage, as well as extend syntax configurability for custom DSLs.

Acknowledgments

This work was partially supported by NSERC Discovery Grant RGPIN-04973 (to I.N.), Canada Research Chair program (to I.N.), DARPA HR0011-18-3-0007 and HR0011-20-9-0017 (to S.A.), and NIH R01 HG010959 (to B.B.). Codon is currently maintained by Exaloop, Inc. We thank all contributors whose contributions were invaluable to the development of Codon.

Competing interests. A.S., B.B., S.A., and I.N. are shareholders of Exaloop, Inc., which oversees the development of Codon.

Availability. Codon source code, binary distributions, as well as the benchmarking suite is freely available at https://github.com/exaloop/codon. Full documentation is available at https://docs.exaloop.io/codon/.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[3] Anaconda. 2018. Numba. https://numba.pydata.org/

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

[5] Anonymous. 2023. Hierarchical Multi-Dimensional Arrays and its Implementation in the CoLa Domain Specific Language for Block-Based Data Compression. In *Under submission to CGO'23*.

[6] John Aycock. 2000. Aggressive Type Inference. In *Proceedings of the 8th International Python Conference*, Vol. 1050. 18.

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 193–205. http://dl.acm.org/citation.cfm?id=3314872.3314896

[9] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

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

[11] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 39–51. https://doi.org/10.1109/CGO51591.2021.9370333

[12] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 89–100. https://doi.org/10.1109/PACT.2011.15

[13] Tyler A. Cabutto, Sean P. Heeney, Shaun V. Ault, Guifen Mao, and Jin Wang. 2018. An Overview of the Julia Programming Language. In *Proceedings of the 2018 International Conference on Computing and Big Data* (Charleston, SC, USA) *(ICCBD '18)*. Association for Computing Machinery, New York, NY, USA, 87–91. https://doi.org/10.1145/3277104.3277119

[14] Brett Cannon. 2005. Localized Type Inference of Atomic Types in Python.

[15] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-specific Approach to Heterogeneous Parallelism. *SIGPLAN Not.* 46, 8 (Feb. 2011), 35–46. https://doi.org/10.1145/2038037.1941561

[16] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An empirical study on dynamic typing related practices in python systems. In *Proceedings of the 28th International Conference on Program Comprehension*. 83–93.

[17] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: a parallel DSL for image analysis and visualization. In *Acm sigplan notices*, Vol. 47. ACM, 111–120.

[18] Hyunghoon Cho, David J. Wu, and Bonnie Berger. 2018. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology* 36, 6 (01 Jul 2018), 547–551. https://doi.org/10.1038/nbt.4108

[19] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. 2015. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press. https://doi.org/10.1017/CBO9781107337756

[20] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. 2011. Building and Using Pluggable Type-Checkers. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 681–690. https://doi.org/10.1145/1985793.1985889

[21] Mark Dufour. 2006. *Shed skin: An optimizing python-to-c++ compiler*. Master's thesis. Delft University of Technology.

[22] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 1–18.

[23] Michael Furr, Jong-hoon An, and Jeffrey S Foster. 2009. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 283–300.

[24] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[25] K Hayen. 2012. Nuitka. http://nuitka.net

[26] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (Nov. 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

[27] Independent JPEG Group. 2022. JPEG software. https://ijg.org/

[28] Joint Video Team. 2009. JM software (v19.0). http://iphome.hhi.de/suehring/

[29] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C: the

ableC extensible language framework. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[30] Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. 2020. Sound, Heuristic Type Annotation Inference for Ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 112–125. https://doi.org/10.1145/3426422.3426985

[31] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1575–1590.

[32] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *Proc. IEEE/ACM Automated Software Engineering*. IEEE, 943–948.

[33] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 20.

[34] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Palo Alto, California, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL]

[36] Didier Le Botlan and Didier Rémy. 2014. MLF: raising ML to the power of System F. *ACM SIGPLAN Notices* 49, 4S (2014), 52–63.

[37] Jukka Antero Lehtosalo. 2015. *Adapting dynamic object-oriented languages to mixed dynamic and static typing.* Ph.D. Dissertation. University of Cambridge.

[38] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276489

[39] Heng Li. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997 [q-bio.GN]

[40] Manas. 2023. Crystal. https://crystal-lang.org/

[41] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[42] Gor Nishanov. 2017. ISO/IEC TS 22277:2017. https://www.iso.org/standard/73008.html

[43] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/1390630.1390656

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[47] Jaak Randmets. 2017. *Programming languages for secure multi-party computation application development.* Ph.D. Dissertation. PhD Thesis, University of Tartu.

[48] Brianna M Ren and Jeffrey S Foster. 2016. Just-in-time static type checking for dynamic languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 462–476.

[49] Richter, Thomas. 2022. libjpeg. https://github.com/thorfdbg/libjpeg

[50] Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[51] Berry Schoenmakers. 2018. MPyC—Python package for secure multiparty computation. In *Workshop on the Theory and Practice of MPC. https://github. com/lschoe/mpyc.*

[52] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. 2019. Seq: A High-Performance Language for Bioinformatics. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 125 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360551

[53] Ariya Shajii, Ibrahim Numanagić, Alexander T Leighton, Haley Greenyer, Saman Amarasinghe, and Bonnie Berger. 2021. A Python-based programming language for high-performance computational genomics. *Nature Biotechnology* 39, 9 (2021), 1062–1064. https://doi.org/10.1038/s41587-021-00985-6

[54] Jared T. Simpson and Richard Durbin. 2012. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res* 22, 3 (Mar 2012), 549–556. https://doi.org/10.1101/gr.126953.111 22156294[pmid].

[55] Haris Smajlović, Ariya Shajii, Bonnie Berger, Hyunghoon Cho, and Ibrahim Numanagić. 2023. Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. *Genome Biology* 24, 1 (2023), 1–18.

[56] Stack Overflow. 2022. Stack Overflow Developer Survey 2022. https://survey.stackoverflow.co/2022/

[57] Rust Team. 2013. The MIR. https://rust-lang.org

[58] Rajan Walia, Chung chieh Shan, and Sam Tobin-Hochstadt. 2020. Sham: A DSL for Fast DSLs. arXiv:2005.09028 [cs.PL]

[59] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. 1994. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.* 29, 12 (Dec. 1994), 31–37. https://doi.org/10.1145/193209.193217

[60] Matei Zaharia, William J. Bolosky, Kristal Curtis, Armando Fox, David A. Patterson, Scott Shenker, Ion Stoica, Richard M. Karp, and Taylor Sittler. 2011. Faster and More Accurate Sequence Alignment with SNAP. *CoRR* abs/1111.5572 (2011). arXiv:1111.5572 http://arxiv.org/abs/1111.5572

[61] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491

Appendix

A A short overview of LTS-DI

The LTS-DI (Localized Type System with Delayed Instantiation) type checking algorithm operates on a localized block (or list) of statements that, in practice, represents either a function body or the top-level code. The crux of LTS-DI's typing algorithm consists of a loop that continually runs the type checking procedure on expressions whose types are still not completely known until either all types become known or no changes can be made (the latter case implies a type checking error, often due to a lack of type annotations). Multiple iterations are necessary because types of later expressions are often dependent on the types of earlier expressions within a block, due to dynamic instantiation (e.g. `x = []; z = type(x); x.append(1); z()`).

Type checking of literals is straightforward, as the type of a literal is given by the literal itself (e.g. `42` is an `int`, `3.14` is a `float`, etc.). Almost all other expressions—binary operations, member functions, constructors, index operations, and so on—are transformed into a call expression that invokes a suitable magic method (e.g. `a + b` becomes `a.__add__(b)`). Each call expression is type checked *only* if all of its arguments are known in advance and fully realized. Once they are realized, the algorithm recursively type checks the body of the function with the given input type argument types and caches the result for later uses.

Call expression type checking will also attempt to apply expression transformations if an argument type does not match the method signature, an example of which is unwrapping `Optional` arguments. Finally, references to other functions are passed not as realized functions themselves (as we often cannot know the exact realization at the time of calling), but instead as temporary named function types (or partial function types, if needed) that point to the passed function. This temporary type is considered to be "realized" in order to satisfy LTS-DI's requirements.

Below, we provide a semi-formal characterization of the algorithm and highlight its differences from the standard Hindley-Milner type checking algorithm.

Notations and definitions

Before proceeding, we will introduce the following notations and definitions:

- Each function $\mathcal{F}$ is defined to be a list of statements coupled with argument types $\mathcal{F}_1^{\text{arg}}, \ldots, \mathcal{F}_n^{\text{arg}}$ and a return type $\mathcal{F}^{\text{ret}}$.
- Each statement is defined to be a set of expressions $e_1, \ldots, e_m$. Each expression e has a type e_{type}—the goal of type checking is to ascertain these types.
- All types are either *realized* (meaning they are known definitively) or *unrealized* (meaning they are partially or completely unknown). For example, `int` is a realized type, `List[T]` is only partially realized as `T` is a generic type, and `T` itself is completely unrealized. Let Realized(t) denote whether type t is fully realized.
- Some expressions are *returned* from a function and thus used to infer the function's return type. Let Returned(e) denote whether expression e is returned.
- Let UnrealizedType() return a new, unrealized type instance.
- Unification is the process by which two types are forced to be equivalent. If both types are realized, both must refer to the same concrete type, or a type checking error will occur. Partially realized types are recursively unified; for example, unifying `List[T]` and `List[float]` results in the generic type `T` being realized as `float`. Let Unify(t_1, t_2) denote this operation for types t_1 and t_2.
- Define an *expression transformation* to be a function $\xi : \mathbb{E} \mapsto \mathbb{E}$ that converts one expression into another, where $\mathbb{E}$ is the set of expressions. LTS-DI employs a set of expression transformations $\mathcal{X}$ to handle various aspects of Python's syntax and semantics, such as what is described in 21.

The algorithm

The LTS-DI algorithm is primarily based on two subroutines that recursively call one another. Firstly, LTSDI$(\mathcal{F})$ (Algorithm 1) takes a function $\mathcal{F}$ and assigns realized types to each expression contained within or reports an error if unable to do so. This procedure continually iterates over the contained expressions, attempting to type check each that has an unrealized type. If no expression types are modified during a given iteration, an error is reported. Otherwise, if all expression types are realized, the procedure terminates.

Secondly, TypeCheck(e) performs type checking for the individual expression e. Since this process predominantly entails type checking call expressions, Algorithm 2 outlines the algorithm specifically for such expressions. Each argument is first recursively type checked, after which the types of the argument expressions are unified with the function's argument types. If unification fails, expression transformations are applied in an effort to reach a successful unification; if none is encountered, an error is reported. In the end, if all argument expression types are realized, the function body is recursively type checked by again invoking LTSDI.

Special cases

Optional values. In Python, all objects are treated as references, and there is no formal distinction between optional and non-optional references. As LTS-DI supports by-value passing and thus makes a distinction between optional and non-optional types, it automatically promotes certain types in a program to `Optional` as needed in order to accommodate Python's `None` construct. It also automatically coerces non-optionals and optionals when needed to maximize the compatibility with Python.

Algorithm 1: Type checking of a function $\mathcal{F}$.

```
Result: LTSDI($\mathcal{F}$)
$\mathcal{F}^{\text{ret}} \leftarrow$ UnrealizedType();
foreach $s \in \mathcal{F}$ do // iterate over statements
    foreach $e \in s$ do // iterate over expressions
        $e_{\text{type}} \leftarrow$ UnrealizedType();
    end
end
$\mathcal{T} \leftarrow \{(e, e_{\text{type}}) \mid e \in s, \forall s \in \mathcal{F}\}$;
loop
    $\mathcal{T}_0 \leftarrow \mathcal{T}$;
    foreach $s \in \mathcal{F}$ do // iterate over statements
        foreach $e \in s$ do // iterate over expressions
            if Realized($e_{\text{type}}$) then
                if Returned($e$) then
                    Unify($e_{\text{type}}, \mathcal{F}^{\text{ret}}$);
                end
            else
                $e_{\text{type}} \leftarrow$ TypeCheck($e$);
        end
    end
    $\mathcal{T} \leftarrow \{(e, e_{\text{type}}) \mid e \in s, \forall s \in \mathcal{F}\}$;
    if $\bigwedge_{(e,t)\in\mathcal{T}}$ Realized($t$) then
        return
    else if $\mathcal{T} = \mathcal{T}_0$ then // check change in types
        error // type checking error
end
```

Algorithm 2: Type checking of a *call*-expression $e = (\mathcal{F}, a_1, \ldots, a_n)$ for called function $\mathcal{F}$ and argument expressions $a_1, \ldots, a_n$.

```
Result: TypeCheck$_{\text{call}}$($e$)
foreach $a_i \in \{a_1, \ldots, a_n\}$ do
    $t \leftarrow$ TypeCheck($a_i$);
    if $\neg$Unify($t, \mathcal{F}_i^{\text{arg}}$) then
        unified $\leftarrow 0$;
        foreach $\xi \in \mathcal{X}$ do
            $a_i' \leftarrow \xi(a_i)$;
            $t' \leftarrow$ TypeCheck($a_i'$);
            if Unify($t', \mathcal{F}_i^{\text{arg}}$) then
                unified $\leftarrow 1$;
                break
            end
        end
        if unified $= 0$ then
            error
        end
    end
end
if $\bigwedge_{a\in\{a_1,\ldots,a_n\}}$ Realized($a_{\text{type}}$) then
    LTSDI($\mathcal{F}$);
    return $\mathcal{F}^{\text{ret}}$
end
```

Functions. Codon supports partial function application and manipulation through Python's `functools.partial` construct or via a new internal ellipsis construct (e.g. `f(42, ...)` is a partial function application with the first argument specified). Each partial function is typed as a named tuple of known arguments, where the names correspond to the original function's names. Unlike in ML-like languages, LTS-DI allows functions *and* partial functions to be generic and thus instantiated multiple times differently. LTS-DI also automatically "partializes" functions that are passed as an argument or returned as a function value and thus allows passing and returning generic functions that can be instantiated multiple times (e.g. lambdas). By doing so, the system is able to support decorators that rely on generic function passing and returning. This approach also results in a somewhat higher number of types and instantiations than a Standard ML-like approach; however, duplicate instantiations are merged later in the compilation pipeline by an LLVM pass and thereby have no effect on code size.

Finally, we note that Codon allows users to overload methods either through using static expression and `isinstance` checks (as commonly done in Python) or through *overloaded methods* with `@overload` decorator (syntactic sugar for manual `isinstance` conditionals).

Miscellaneous considerations. In order to match the behaviour of Python, Codon processes import statements at *runtime*. This is done by wrapping each import in a function that is called only once by the first statement that reaches it. Codon's LTS-DI also unwraps iterables when needed and casts `int` to `float` when needed.

Codon also supports Python interoperability and can handle objects managed by the Python runtime via its `pyobj` interface. Such objects are automatically wrapped and unwrapped by LTS-DI, depending on the circumstances, in a similar fashion to `Optional[T]`. As a result, all existing Python modules and libraries (NumPy, SciPy, etc.) can be used within Codon.

B Additional Benchmarks

Microbenchmarks

`pyperformance` benchmarks were done on an arm64 (Apple M1 Max) machine with 64 GB of RAM running macOS 12.5.1, and on x86-64 (64-core Intel Xeon Gold 5218) machine with 754 GB of RAM running CentOS 7. We ran CPython 3.10.8, 3.8.2, PyPy 7.3.9, Clang 13.0.1 and Codon 0.15.

The following table contains the runtimes `pyperformance` microbenchmarks on arm64/macOS. The same trends were observed in Linux/x86-64 environment as well.

macOS	Python	PyPy	C++	Codon
sum	2.54	0.11	0.00	0.00
float	10.76	2.02	-	0.33
go	16.43	4.73	-	0.62
nbody	4.29	0.35	0.15	0.15
chaos	13.78	1.04	-	0.64
spectral_norm	39.49	0.58	-	0.23
set_partition	48.50	22.75	25.73	12.17
primes	12.86	0.72	-	0.50
binary_trees	437.90	7.89	9.22	3.62
fannkuch	349.34	32.50	-	12.00
word_count	4.36	5.74	3.64	1.58
taq	56.34	56.04	12.14	4.37

Table 1. `pyperformance` benchmark runtime (in seconds) on four different implementations in arm64/macOS environment. `sum` benchmark was completely inlined by Codon and C++; therefore the runtime was instantaneous.

Note that we were able to trivially add OpenMP parallelism to Codon implementations of `primes` and `fannkuch` benchmarks, resulting in 12× and 4× runtime improvements, respectively.

Seq benchmarks

macOS	Time (m:s)	Mem. (GB)	Seq imprv.
BWA (C)	1:19	**0.4***	2.3×
Rust-Bio (Rust)	4:17	61.1	7.6×
SeqAn (C++)	4:53	2.6	8.6×
Seq	50	7.3	1.5×
Seq (prefetch)	**34**	7.3	—

Table 2. Performance in SMEM finding from using the Seq language and compiler on a macOS system. The Seq version achieves nearly a 2× speed improvement over BWA MEM, and roughly a 6–9× improvement over high-performance genomics libraries Rust-Bio and SeqAn. The reported timings represent the time needed for each tool to find SMEMs in a set of FASTQ reads. Seq, Rust and SeqAn FM-index implementations are not compressed, and as such use more RAM than BWA’s compressed implementation (marked with *). See [53] for more details.

	Seq	C++	C++	Seq	**Speedup**
	w/o prefetch	**Clang**	**GCC**	**with prefetch**	
SNAP	328.1	450.5	327.5	**211.9**	1.5–2.1×
SGA	453.0	569.3	610.1	**409.6**	1.4–1.5×

Table 3. Seq runtime compared to C++ as compiled with Clang and GCC (seconds) when querying large genomic indices generated by SNAP [60] and SGA [54]. The difference between the without-prefetch and with-prefetch Seq programs is just a single `prefetch` statement. See [52] for more details.

C Additional Figures

```
class Builder : public TypeBuilder {
  llvm::Type *buildType(LLVMVisitor *v) {
    return v->getBuilder()->getFloatTy();
  }

  llvm::DIType *buildDebugType(LLVMVisitor *v) {
    auto *module = v->getModule();
    auto &layout = module->getDataLayout();
    auto &db = v->getDebugInfo();
    auto *t = buildType(v);
    return db.builder->createBasicType(
           "float_32",
           layout.getTypeAllocSizeInBits(t),
           llvm::dwarf::DW_ATE_float);
  }
};

class Float32 : public CustomType {
  unique_ptr<TypeBuilder> getBuilder() const {
    return make_unique<Builder>();
  }
};
```

Figure 9. 32-bit float `CustomType`

	LLVM equivalent	Examples
`Node`	N/A	See below
`Module`	`Module`	N/A
`Type`	`Type`	`IntType`, `FuncType`, `RecordType`
`Var`	`AllocaInst`	`Var`, `Func`
`Func`	`Function`	`BodiedFunc`, `LLVMFunc`, `ExternalFunc`
`Value`	`Value`	See below
`Const`	`Constant`	`IntConst`, `FloatConst`, `StringConst`
`Instr`	`Instruction`	`CallInstr`, `TernaryInstr`, `ThrowInstr`
`Flow`	Various	`IfFlow`, `WhileFlow`, `ForFlow`

Table 4. CIR structure

```
for y in range(pred.dims[0]):
    pred[y,:] = ref[pred](y,-1)
```

```
int pos_x = pix_a.pos_x;
int pos_y = pix_a.pos_y;
for (i=0; i<cr_MB_y; i++)
    vline[i] = ref[pos_y++][pos_x];
for (j=0; j<cr_MB_y; j++) {
    int predictor = vline[j];
    for (i = 0; i < cr_MB_x; i++)
        pred[j][i] = (imgpel) predictor;
}
```

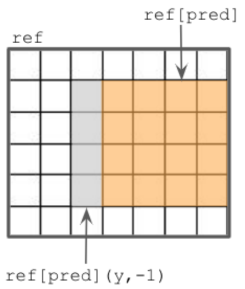


Figure 10. (Left) CoLa code (top) and C code (bottom) for horizontal prediction. (Right) A visualization of the data accessed in the reference region for the computation.

```
class PrefetchFunctionTransformer : public Operator {
  // return x --> yield x
  void handle(ReturnInstr *x) override {
    auto *M = x->getModule();
    x->replaceAll(
      M->Nr<YieldInstr>(x->getValue(), /*final=*/true));
  }

  // idx[key] --> idx.__prefetch__(key); yield; idx[key]
  void handle(CallInstr *x) override {
    auto *func =
      cast<BodiedFunc>(util::getFunc(x->getCallee()));
    if (!func ||
        func->getUnmangledName() != "__getitem__" ||
        x->numArgs() != 2) return;

    auto *M = x->getModule();
    Value *self = x->front(), *key = x->back();
    types::Type *selfType = self->getType();
    types::Type *keyType = key->getType();
    Func *prefetchFunc = M->getOrRealizeMethod(
      selfType, "__prefetch__", {selfType, keyType});
    if (!prefetchFunc) return;

    Value *prefetch = util::call(prefetchFunc, {self, key});
    auto *yield = M->Nr<YieldInstr>();
    auto *replacement = util::series(prefetch, yield);

    util::CloneVisitor cv(M);
    auto *clone = cv.clone(x);
    see(clone); // don't visit clone
    x->replaceAll(M->Nr<FlowInstr>(replacement, clone));
  }
};
```

```
@inline
def _dynamic_coroutine_scheduler[A,B,T,C](
    value: A, coro: B, states: Array[Generator[T]],
    I: Ptr[int], N: Ptr[int], M: int, args: C):
    n = N[0]
    if n < M:
        states[n] = coro(value, *args)
        N[0] = n + 1
    else:
        i = I[0]
        while True:
            g = states[i]
            if g.done():
                if not isinstance(T, void):
                    yield g.next()
                g.destroy()
                states[i] = coro(value, *args)
                break
            i = (i + 1) & (M - 1)
        I[0] = i
```

Figure 11. (Top) Function-to-coroutine transformer for Codon IR. This transformation is utilized by several of Seq’s pipeline optimizations for bioinformatics and genomics applications. (Bottom) Coroutine scheduler for Seq’s prefetch optimization. A pipeline stage marked with `@prefetch` is converted to a coroutine by the pass in Figure 7, and this scheduler is used to manage multiple instances of this coroutine, overlapping cache miss latency from one with useful work from another. The scheduler state is passed as pointers (`Ptr[int]`) and modified by the scheduler, which itself gets inserted in the pipeline. Codon IR’s bidirectionality is used to instantiate the scheduler with concrete argument types when applying the prefetch optimization.

D Codon Extension API

New Python-like DSLs can be authored by creating a Codon DSL plugin and putting it into the Codon search path. A DSL plugin consists of:

- plugin-specific Codon code (which is treated akin to a Python library package), and
- A set of syntactic extensions and compiler passes in C++.

Codon C++ API can be used by including `codon/codon.h`. All API primitives are located within `codon` namespace.

Every Codon plugin derives from a `codon::DSL` class that provides several virtual methods for providing general plugin info, registering Codon IR and LLVM IR passes, and adding new keywords.

Defining new keywords

A Codon DSL can "extend" a Python syntax by adding a new "soft" keyword that can:

1. redefine a simple expression, or
2. redefine a code block.

In Python grammar notation,[5] a keyword `kwd` defines the following grammar blocks that are processed if and only if all other parses fail:

```
custom_simple_stmt:
| 'kwd' expression
custom_block_stmt:
| 'kwd' expression? ':' block
```

Such keywords can be registered through overriding `getExprKeywords` or `getBlockKeywords` methods of `codon::DSL` class that return a C++ vector of plug-in specific keywords together with the associated transformation functions that transform the keyword into a valid Codon statement that can be typechecked[6].

Adding transformation passes

New Codon IR passes can be made by implementing `codon::ir::transform::Pass` interface that provides stubs to be called upon encountering the specific CIR nodes that all inherit from common CIR classes (see Figure 12) to provide a recipe for their manipulation and transformation into a different set of CIR nodes. Such passes need to be registered with the compiler by overriding `addIRPasses` method of `codon::DSL` class. Note that a pass can load and process new Codon code from a given module, and can instantiate new types and functions through `getOrRealizeFunc`, `getOrRealizeMethod` and `getOrRealizeType` methods in `codon::ir::Module` class. Each of these methods will invoke the typechecker when needed to ensure the soundness of new instantiations.

[5] https://docs.python.org/3/reference/grammar.html

[6] While there is no API for extending the typechecker at the moment of writing this manual, we plan to address this issue in the future.

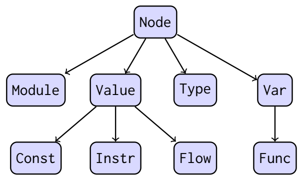


Figure 12. CIR hierarchy

Similarly, an LLVM pass can be added by defining it through the LLVM API infrastructure and registered by overriding `addLLVMPasses` method of `codon::DSL` class.

```
class AddFolder : public OperatorPass {
  void handle(CallInstr *v) {
    auto *f = util::getFunc(v->getCallee());
    if (!f || f->getUnmangledName() != "__add__") return;
    auto *lhs = cast<IntConst>(v->front());
    auto *rhs = cast<IntConst>(v->back());
    if (lhs && rhs) {
      auto sum = lhs->getVal() + rhs->getVal();
      v->replaceAll(v->getModule()->getInt(sum));
    }
  }
};
```

Figure 13. Simple integer addition constant folder pass in Codon IR. This pass recognizes expressions of the form `<int> + <int>` (where `<int>` is a constant integer) and replaces them with the correct sum.

More details on how to write a CIR pass and how to utilize CIR API efficiently, together with working examples, is available at https://docs.exaloop.io/codon/advanced/ir.

Received 2022-11-10; accepted 2022-12-19