Stella: A Python-based Domain-Specific Language for Simulations

David Mohr
Department of Computer Science
University of New Mexico
dmohr@cs.unm.edu

Darko Stefanovic
Department of Computer Science
University of New Mexico
darko@cs.unm.edu

ABSTRACT

We wish to make it easier and quicker to write well-performing scientific simulations that (1) have single-thread performance competitive with low-level languages, (2) use object-oriented programming to properly structure the code, and (3) are very easy to develop. Instead of prototyping in a high-level language and then rewriting in a lower-level language, we created a DSL embedded in Python that is transparently usable, retains some OOP features, compiles to machine code, and executes at speed similar to C.

CCS Concepts

•**Software and its engineering** → **Domain specific languages;** *Object oriented languages;* •**Applied computing** → *Chemistry; Physics;*

Keywords

domain-specific languages; scientific simulations; Python

1. INTRODUCTION

One of the most widely used numerical methods is the Kinetic Monte Carlo (KMC) algorithm [4] of statistical physics, also known to chemists as the Gillespie algorithm [9], as the 7700 citations in the literature to just the two original articles demonstrate. In this approach, the scientist-programmers represent their natural phenomenon of interest as a continuous-time Markov process model, and each execution of the algorithm computes a single trajectory of that model by repeatedly drawing from a pseudo-random source to select next events and times. As the available choices for next event depend on the state of the model, and the state intricately depends on the complete history of past events, each execution is inherently *sequential*. However, to numerically explore the model typically requires many thousands of trajectories for statistically valid results, which means that the approach overall is inherently *embarrassingly parallel*. Much programming language research today focuses on automatic parallel execution of concurrent programs, which is neither needed nor applicable to this domain. Instead, simulations using the KMC algorithm require stellar single-thread performance, and how to deliver that to scientists wishing to write in a convenient high-level language is the focus of this paper.

Generally, scientists writing custom domain-specific simulations face a difficult choice selecting a programming language. High *execution speed* is of great importance since stochastic simulation requires many executions. This suggests using a low-level language such as C. Another issue is *code reuse*: while working on a topic a scientist often creates many different simulations that are similar but explore different properties of the model. Since it is hard to know which properties will be relevant in the end, it is important to keep the code well structured and easy to reuse, otherwise improvements to the framework in one simulation will have to be ported by hand to the remaining simulations. In C this is time-consuming and error-prone, to a large degree because it is not easily possible to use an object-oriented programming (OOP) style. Another dimension is that of code optimization, algorithmic and otherwise. We should not optimize prematurely [11], but optimizing in later stages makes it a challenge to keep the code compatible with previously written simulations. Again, C falls short.

One common approach is to use a high-level scripting language (HLSL) as a prototyping language and, once the relevant properties are identified, to rewrite the "final" simulation in C. This takes advantage of the high productivity HLSLs allow but does not solve the code reuse issues raised above. And since today HLSLs are convenient and fast enough for many general tasks, programmers would like to more fully utilize their features, and somehow avoid the headache of rewriting.

Python, a mature, general-purpose HLSL, is a popular language in the scientific community [13, 20, 22], not least because it encompasses rich libraries for numerical computation (NumPy, SciPy) and data plotting (matplotlib). For Python it is the recommended practice to refactor the performance-critical code into a separate module and then re-implement that module in a lower-level language [12]. This may be a feasible approach for software that is widely distributed and reused by many since then the additional effort is amortized. However, this is nothing but using Python as a prototyping language: there is a rewriting cost and maintenance issues as noted above. The smaller the section with the performance-critical code is, the more viable is the re-implementation advice. However, there are classes of programs where the critical section is broad, such as KMC simulations (see Section 2.2). There, fac-

SAC'16, April 4-8, 2016, Pisa, Italy

ACM. 978-1-4503-3739-7/16/04. . . $15.00

DOI: http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

toring only part of the simulation into a foreign language module results in many costly transitions into and out of the scripting language run time, negating much of the speed benefit. In that case re-implementing the performance-critical section can be tantamount to rewriting the whole program.

Would it be possible to avoid rewriting the performance-critical sections in C simply by having Python run fast enough? We explored the performance with a KMC simulation from [19]. Unfortunately the performance penalty of the default implementation is prohibitive, running up to three orders of magnitude. Different Python implementations (Jython, PyPy) only marginally improved the run-time of the simulation. For comparison we also implemented the simulation in C++, which was significantly faster than Python, but remained significantly slower than the C reference. Not surprisingly, then, several existing approaches to speeding up Python code have been developed (discussed in Section 5). While they provide a benefit, they all come rather short of C-like performance, especially for our scientific simulations, because they either aim to support the complete language (always difficult to do efficiently for a feature-rich dynamic language), or to integrate so well with Python that it is easy inadvertently to call into the slow run-time (something we would rather avoid), or focus mainly on parallel execution (not relevant to our problem domain). Therefore, we have been developing a domain-specific language (DSL), called STELLA, with Python language features selected for custom-written scientific simulations. With the reduced feature set we can make execution speed a priority, allowing us to compile the DSL into native code rather than interpreting it.

We avoid all expensive language features in STELLA, in order that it can be obvious to the programmer that any single statement is quick to execute. Among other ramifications, this design goal favors static over dynamic typing, since the run-time type checks are a hidden cost in dynamically typed languages. This potential loss of functionality will be compensated in part by staging the execution: STELLA code is constructed within *CPython*, the standard implementation of Python. Thus the full language is available in a pre-processing, or generative, stage. Hence there will be no restrictions during the many tasks that the programmer must solve outside of the main simulation loop, such as parsing command-line parameters, configuration, and state files, accessing databases, and coordinating the simulation execution [15]. Additionally, the dynamic features of Python can be used to assemble classes and objects at run-time. The results of the DSL execution are made available to the Python run-time for post-processing. Thus it is easy to use the excellent data analysis and visualization tools that Python libraries offer, because now the complete language is available again.

Properly structuring simulations and frameworks should be done with OOP, therefore we deem this support essential. Full OOP support is expensive: it takes great development effort and the complexities are likely to require some run-time support. Thus STELLA supports a limited range of OOP patterns which give the programmer useful features without adding run-time overhead.

Although it shares Python's syntax, STELLA is a different language, and some adjustments are necessary for existing Python simulations to make them valid STELLA programs. But in practice we found that only small changes were required.

We have three main goals: (1) *Execution speed at the level of C.* This requires both compiling to machine code, and a strict separation from the Python run-time. (2) *Access to object-oriented programming.* OOP is an established method for organizing source code, which we want to exploit to share code between simulations, and to encapsulate optimizations. It is important that the OOP implementation should work without run-time overhead, to preserve the first goal (3) *Ease of use.* From a programmer's perspective, the dynamic capabilities of Python make it possible to execute STELLA seamlessly—we want only minimal changes, if any, to be needed from an original Python program. Again, trade-offs are required to respect the first goal.

In this paper we describe the design and implementation of STELLA. The evaluation of the prototype suggests that our approach allows all three goals to be simultaneously achieved for the application domain of interest to us, scientific simulations.

2. FEATURES AND RATIONALE

STELLA is a statically typed, just-in-time compiled language embedded in Python. We aim to use Python and stay faithful to its semantics, but only make those features available that can be efficiently implemented without run-time support. This lets STELLA deliver predictable performance whilst remaining compatible with Python.

A program using the STELLA DSL initially runs inside the standard Python interpreter. All code is first parsed by the Python interpreter, and execution starts as for any Python program. Therefore the program can initially use all features of the complete Python language, e.g., to compute constants, dynamically assemble classes and objects, and load initial data. All libraries are usable at this point since the DSL is not yet active. STELLA then executes the simulation core. After the DSL finishes, the program returns to the full-Python stage. Thus, Python natively handles all pre- and post-processing, which avoids any compatibility problems. STELLA is activated from within Python only for the simulation core.

Marking the starting point for STELLA code, the *entry function*, is the only modification of the source code that is always required; this function must be wrapped by the STELLA library. This approach is very flexible because (1) it processes exactly the same code as the Python interpreter would, the bytecode; (2) the original function stays intact and remains usable within Python; and (3) it reduces the barrier of entry for the programmer.

When the wrapped *entry function* is called, the bytecodes that the Python interpreter created are examined at run-time. As a result, modules can be used as appropriate, objects can be dynamically assembled, and libraries can be distributed in native Python manner. Any native Python function or object is compatible with STELLA, as long as it adheres to supported Python features.

DSLs typically allow making calls into the host language. This is also the case for most existing methods to speeding up Python programs (see Section 5). STELLA takes the opposite approach: all function in the call graph of the *entry function* are automatically translated to STELLA as well. We do so for performance reasons: the Python run-time is slow, so allowing transparent calling of Python functions (or in fact, any Python feature), would introduce the risk that the programmer does so unintentionally. Such performance regressions are difficult to identify, and must be avoided by any high-speed implementation anyway. Therefore by not even implementing this convenience feature we let the programmer save time in the end because then best practices are being followed from the onset.

Since the original Python function remains usable, and has the same semantics, any prototyping and debugging can be performed directly in Python. Then once the program is working as desired, STELLA can be introduced by changing one source line—or even added to the program at run-time. Being dynamically invoked means STELLA can infer enough information about the program to behave like Python, despite being a statically typed and compiled language. This is the reason that only the *entry function* needs to be marked: types are automatically inferable, and do not require annotation, because the function input parameter types are known at run-time.

The call graph is computed while analyzing functions. Once a call is encountered, the callee is marked to be analyzed later. The call graph is easy to compute because STELLA does not support dynamic language features such as "`eval()`". While dynamic language features are a useful tool in general, they are inherently slow. Even if they were natively implemented in STELLA, the performance could not meet the programmer's expectations, therefore we disallow them.

The particular supported language features of Python are selected according to a simple criterion: does the feature have a constant run-time cost? For example, accessing a dictionary element requires calculating the hash value of the key before accessing the memory location. This is a *hidden cost* since Python's syntax does not distinguish between array indices and hash indices. Another hidden cost in scripting languages is the frequent memory allocation and deallocation. Since STELLA focuses on high-performance code, we believe it is reasonable to preallocate the required memory, and to discourage strongly from allocating memory within the DSL.

After the analysis is completed, the code is compiled and executed using LLVM [14], a proven compiler toolkit. Therefore one way to look at STELLA is as a compiled subset of Python. This view, however, immediately evokes the question of a roadmap to support *all* of Python, and that is not our goal. We believe that describing it as a domain-specific language is more accurate, since it is a more restrictive, but very efficient approach for some domains. We mention the "Python subset" viewpoint because it makes it easier to understand what STELLA looks and feels like. STELLA is a complete programming language, but the initial design is more suited for scientific simulations than other domains.

2.1 Example Program

Figure 1 lists the source for a STELLA simulation of a simple random walk in one dimension. The `RandWalk` class and the "`s`" object are created in plain Python. This includes running the initialization function, which creates a NumPy array on line 4. Line 19 invokes STELLA on "`s.run()`".

The analysis determines that the called function is a bound method of `RandWalk.run()` to the object "`s`". First a shadow structure is created for this object, and then it is initialized with the same data that "`s`" currently contains. Afterwards the method starting on line 9 is analyzed. We discover that `RandWalk.keep_going()` is called, and include it into the translation pipeline. The access of the attribute `surface` on line 12 will directly read and write the memory backing the NumPy array. The function `rng.mt_drand()` called on line 14 is part of an external C library, and therefore not included in the analysis and translation but directly called by the resulting STELLA code. The two Python methods are then compiled

```
import stella, numpy, mtpy as rng

class RandWalk(object):
    def __init__(self, size, max_t):
        self.max_t = max_t
        self.surface = numpy.zeros(size, dtype=int)
        self.t = 0
    def keep_going(self, pos):
        return pos < len(self.surface) \
                and self.t < self.max_t
    def run(self):
        pos = 0
        while self.keep_going(pos):
            self.surface[pos] += 1
            self.t += 1
            if rng.mt_drand() < 0.5 and pos > 0:
                pos -= 1
            else:
                pos += 1
s = RandWalk(3000, 30000000)
stella.wrap(s.run)()  # DSL executing the core
print(s.t, s.surface)

# pylama:ignore=E265,E401,E302,E702,E301,E126
```

Figure 1: Example simulation: a semi-infinite random walk.

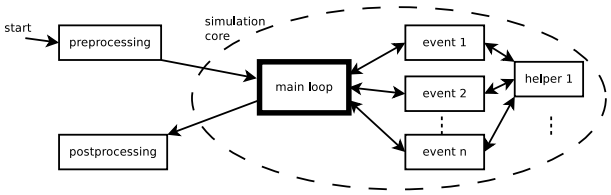

Figure 2: Simplified structure of a typical simulation.

and the entry method "`s.run()`" is executed. Once it returns, line 20 will again be executed by the regular Python interpreter, and print the results.

2.2 Size of Performance-Critical Sections

We believe that STELLA is a useful approach for a multitude of programs, but, as mentioned in Section 1, it is particularly difficult to rewrite the performance-critical section in a lower-level language for Kinetic Monte Carlo simulations. In many cases the "80/20" rule applies: 80% of the execution time is spent in 20% of the program—but there are also classes of programs where it does not apply, and this is one.

Consider the structure of a typical simulation (Figure 2): a main loop keeps track of the progress of the simulation, and often also decides what event is currently being simulated. Then, depending on the event, it calls some function to evaluate the effects. These functions may call helper functions. It is easy to see that the main loop will be called very frequently. But if only the main loop is rewritten, the program will constantly transition from the high-level to the lower-level language and back.

An attempt at a workaround would be to implement the most frequently called subset of functions. But the programmer does not

necessarily know with what frequency these events will occur, i.e., the frequency of calls to the event-handling functions. Finding the distribution of the events may very well be what the simulation is meant to compute in the first place!

Transitioning from one run-time system to another can be costly. When the operations in the optimizing language, e.g. NumPy, manipulate only small amounts of data instead of larger data sets, then the transition cost from Python is not amortized by the large speedup that the optimizing language is able to provide. This is a similar situation to when only part of a simulation is rewritten in a lower-level language.

2.3 Language Description

Since STELLA operates on the Python bytecodes, the supported language features are defined by just those. Yet for the programmer this is merely an implementation detail, and therefore we instead describe the language in high-level terms with references to Python features.

The supported expressions are arithmetic operators on integers and floating point numbers, comparisons, conditional expressions, and the creation of tuples. Lambdas are not supported, since they are implemented as a dynamically created closure in Python. Operator precedence is identical to Python.

Simple statements such as assignment, `pass`, `return`, `break`, `continue`, are supported. The `raise` statement can be used to abort the current simulation, i.e., to return to Python, but no exception handling is done within STELLA, for efficiency reasons. The `import` statement is not supported, but can easily be used in the pre-processing phase. The `global` statement is supported, but `nonlocal` does not apply, since nested scopes are not supported.

Compound statements such as `if` and `while` are fully supported. The `for` loop is currently restricted to the most common forms, but this restriction is only in place because of the state of the implementation, and not inherent to the language design. As mentioned above, `try` is not supported, because exceptions cannot be caught within STELLA. The `with` statement is not part of the language because it is mainly useful for I/O operations which are not the focus of STELLA.

Functions and classes can only be defined in Python, and subsequently used in STELLA. Nested function definitions are similar to lambda statements, and not supported. Methods (functions defined as part of a class) are supported. Dictionaries and sets are not currently supported (see Section 6 for future work).

Classes are only accessible in Python, i.e., one cannot create new objects in STELLA because this would negatively impact run-time performance. Instead, the programmer should pre-create the objects used during the life time of the simulation. This is a performance optimization in other languages, which is simply required from the outset for STELLA. Object instances that are reachable from the simulation are automatically available in STELLA programs.

The current typing rules for objects are a cross of Python's duck typing, and the static typing that STELLA imposes. An object attribute access is successful if the type that the static analysis inferred contains such an attribute. That is the synopsis of duck typing. Additionally the type inference must conclude only one concrete type whenever an object is accessed, i.e., polymorphism is not allowed. This is not a restriction for the receiving object, i.e., `self`, since these are cloned for each subtype.

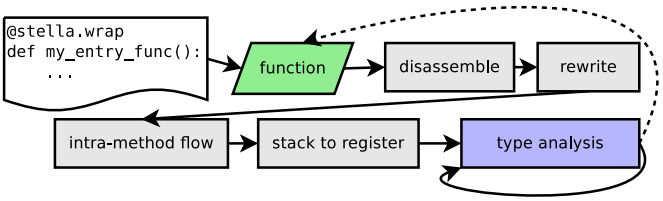

Figure 3: A high-level overview of the analysis phases for a function.

As a result the prototype implementation does not consider subtyping relationships when determining type compatibility. The reason for this limitation is that multiple inheritance is available in Python, which in general does not have a low-level memory layout that can be accessed without dynamic type checks. Therefore the only limitation that is inherent to the design is the unavailability of full multiple inheritance. More limited styles, such as mix-ins, could be supported in the future with an improved static analysis.

2.4 Semantic Differences

We were forced to deviate from the Python semantics in a few areas to preserve the goal of C-like efficiency in the generated code. We do not believe these differences will have a major impact on the programmer. These exceptions are:

Machine types: In Python the basic types are boxed, while in STELLA they are the machine types, as in C. This causes differences in, e.g., overflow or underflow of a variable because Python automatically switches to arbitrary precision math.

Module sign: Modulo always has the sign of the divisor in Python, unlike STELLA and C, where it is the sign of the dividend.

`pow` **return type:** The power function for integers in Python returns an integer result when the exponent is positive, but a float result when the exponent is negative. STELLA implements power identically to C, where the result only depends on the type (integer or float) of the exponent, not the sign.

3. IMPLEMENTATION

The analysis uses a queue to store which functions remain to be analyzed. The queue is initialized with the entry method. Functions are added to the queue once they are discovered, or if the analysis must be interrupted due to missing information.

3.1 Analysis

The analysis is organized into different phases (Figure 3). First Python's introspection facilities are used to *disassemble* the bytecode of the function. Each bytecode is decoded and stored in an intermediate representation (IR-S) object. Initially this object only stores the arguments of the bytecode, but the remaining analysis steps progressively fill in further information. If a bytecode is unsupported, an exception is raised.

Now we have an explicit representation, a list of bytecodes which can be easily manipulated programmatically. The next step is to apply rules to *rewrite* those Python bytecodes which represent higher-level functionality (Section 3.2)

With the structure of the bytecodes set, the *intra-method control flow* converts the index-based information that the Python bytecodes supply into explicit references to the IR-S objects. Not only is this representation more efficient to process, it also makes it easier to write the subsequent analysis and transformation code.

Python's bytecodes operate on a stack of values. This makes it difficult to manipulate the stream of bytecodes [17]. Therefore we transform values on the *stack into a register* to facilitate the generation of LLVM IR, which is also based on registers. Note that here the bytecode arguments are not yet references to the values being processed, but to the bytecodes that create these values. This makes the representation explicit, but temporarily adds one level of indirection (Section 3.4).

The last phase is the *type analysis*. Since the flow of information is explicit, each IR-S object examines its arguments to determine the type of its result. This removes the indirection introduced in the previous phase. Function calls interrupt this analysis if the return type is not yet known, since further type deductions depend on it.

3.2 Loop Rewriting Phase

Python handles for-loops by creating an iterator and using it to traverse the given object. While this approach works well to support flexible iteration over different object types, it is not necessarily the most efficient. Therefore STELLA recognizes the common for-loop patterns and rewrites them into a traditional C-style for-loop, in particular without creating an iterator object or using function calls.

3.3 Control Flow Phase

When the type analysis determines a function call, the called method is added to the analysis queue. Note that all steps until the *type analysis* are only performed once, but several iterations of type analysis may be necessary until types can be completely deduced. This simple control flow analysis suffices since there are no dynamic ways to call a function, such as "`eval()`" in regular Python.

3.4 Stack to Register Phase

The LLVM IR operates on an unlimited number of registers, and requires instructions to be in the single static assignment format (SSA). This in principle makes it easy to transform the stack values to registers by assigning a new register to every stack location. In reality it is slightly more complicated because Python places some things on the stack that STELLA needs to handle at compile time and therefore is not a register. The transformation of stack values is therefore split into two parts: the stack operations are replayed but instead of immediately creating registers, we keep track of which IR-S object put the value into the stack location. The creation of registers is deferred until the next phase.

3.5 Typing Phase

Each IR-S object contains its typing rules and evaluates them based on the types of the operands. The typing rules match those of Python, with the exceptions listed in Section 2.4. Typing must also make decisions about whether an IR-S object is created, i.e., the involved operation is static, or if LLVM IR will be used to perform the operation at run-time. For example, consider the expression "`foo.bar`" : if `foo` is an object, then this is an attribute access for which code must be generated. On the other hand, if `foo` is a module, then the attribute `bar` is examined at compile time, and evaluated accordingly (e.g., it could be a global variable or a function).

3.6 Types

STELLA's type system represents machine types, as used in LLVM. The resulting types are compatible with C, which also uses machine types, so the behavior should be familiar to most programmers.

Scalar Types: Booleans are represented as 8-bit integers. Integers and floating point numbers are presently defined to be 64 bits wide to support accurate calculations for science applications.

Tuples: An anonymous structure is created, which is passed by value.

Arrays: For scalar values NumPy arrays are required. These are a popular choice in existing scientific programs. NumPy arrays do not support complex types such as objects. Instead, STELLA uses regular Python lists for elements with complex types, which are implemented as arrays of pointers. No resize operations are implemented within the DSL.

Objects: Python does not have a structure type akin to the C "struct". Instead programmers use dictionaries or objects for the same purpose. In particular the ability to modify objects on the fly makes them a popular choice when a structured collection of data is required. For this reason alone it is important to support objects in STELLA. When the DSL is invoked, we use introspection to discover the current structure, and create a static C-style structure for the class. This is necessary because a Python object has a memory layout that is not accessible without overhead (e.g., attributes are stored in dictionaries). Therefore the attribute contents need to be transfered from Python to the STELLA representation. Note that object introspection leverages Python's abilities to construct objects, including multiple inheritance, mix-ins, and method resolution. All of these features behave exactly as in Python because CPython handles them. Within STELLA the behavior is much simpler, but we believe that this is not a limiting factor, in part because of the powerful construction mechanisms. Everything is statically resolved, e.g., there is no virtual method table for overloading methods in subtypes. This not only makes the language implementation simpler, but also ensures that there is no hidden overhead caused by run-time type checks.

3.7 Data Transfer Phase

The data representation outlined in Section 3.6 is more low-level than Python's, and therefore the program data needs to be transfered between the two.

NumPy arrays conveniently offer access to their internal representation, which is compatible with C-style array layouts, and hence can be directly used in STELLA as a pointer to the same memory which is being used in Python. No data transfer is necessary.

A shadow structure is created for every object and its attributes are copied over before the DSL execution starts, and copied back after it finishes. While this incurs some overhead, it will be easily amortized by the many attribute accesses which otherwise would have to be routed through the Python data layout.

NumPy arrays can only contain scalar values and therefore lists of objects must be represented in a different manner: each object in the list is handled as an individual object, i.e., STELLA creates a

shadow structure and transfers the content back and forth, and then a list of pointers to these shadow objects is used.

3.8 Compilation and Execution Phase

LLVM is a general compiler framework with an initial focus on C and C++. So while the optimization passes that LLVM currently implements may not be optimal for dynamic languages such as Python itself [10], they are an exact fit for STELLA, a static language without dynamic features: We generate LLVM IR that is very similar to the LLVM IR generated by *clang*. Therefore it comes as no surprise that existing optimizations apply well.

Given the list of IR-S objects we translate each into one or more LLVM IR instructions. This process is straightforward: the explicit control and information flow, together with the type annotations, make it easy to write LLVM IR that performs according to Python's rules. Once LLVM's IR is completely assembled, the *MCJIT* component is used to just-in-time compile and execute the simulation.

3.9 Interfacing C

C libraries are often used to provide high-speed implementations, and since they also use machine types, it is a good fit to provide easy access to them.

Recall that all STELLA programs are also valid Python programs. This continues to be the case even in the presence of C libraries, with just a little extra effort. A Cython [3] wrapper of the C library with manual type annotations is used to provide Python access to the library. This is very easy to create: Cython will automatically generate the wrapper, and the type information is present in the C sources anyway—the types only need to be translated into Python's *ctypes* representation.

When a STELLA program is run, all module references are checked to see if they are backed by a C library. If so, then instead of compiling the functions contained in the module itself, STELLA will call the C function directly.

3.10 Testing

A goal of our project is that each DSL code fragment should have the same semantics as its literal equivalent in Python (with the exceptions defined in Section 2.4). Ideally, this would be ensured via formal verification, e.g., as with the Filet-o-Fish (FoF) framework [7]. Unfortunately, Python does not have a specification but rather is defined by its reference implementation, meaning that formal verification would be a major project on its own, well outside the scope of this paper.

However, STELLA was developed using the test-driven development methodology: for any feature that was implemented, we first wrote a functional test. Here we make use of the fact that all STELLA programs are also valid Python programs: each test runs exactly the same code first in the Python interpreter, then using the DSL compiler, and finally checks if the result is the same. This gives us confidence that STELLA does not have bugs in the features that it does implement.

4. PERFORMANCE EVALUATION

We take C as the standard measure for fast-performing single-threaded programs. Therefore all benchmarks were once implemented in Python, if necessary modified for STELLA, and then translated by hand to C. The modifications to the Python programs that STELLA required were minimal. The C programs were compiled both with gcc 4.9.2 and clang 3.4 at *-O3*. STELLA was linked against LLVM 3.5 and also used *-O3*. All benchmarks were run on an AMD FX(tm)-4100 CPU. In each case, the average of 11 runs is reported.

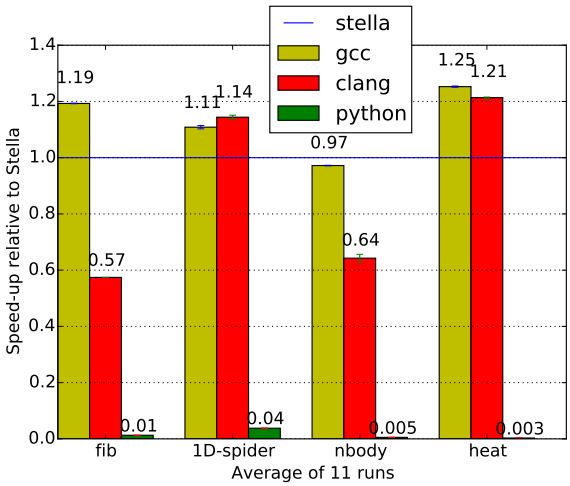

Figure 4: Benchmark: Relative performance of the C implementations compared with STELLA. Values above one correspond to better performance of the given language or compiler, values less than one correspond to better STELLA performance.

Fibonacci is a micro-benchmark that recursively calculates a large Fibonacci number. Therefore the cost of function calls dominates the cost of arithmetic operations.
While the gcc binary is faster than STELLA, the clang binary is slower. On average C is 14% faster than STELLA for this micro-benchmark. We have not investigated the difference between the two C compiler performances.

nbody is part of Debian's programming language benchmark game. It is a deterministic simulation of n celestial bodies and iteratively calculates the forces they exert on each other, and the resulting change in position and velocity.
Both C compilers produce similarly fast binaries, which are on average 13% fasten than STELLA.

1D-spider is a simple simulation in which a spider performs a random walk on a semi-infinite 1D surface. For this stochastic benchmark the random number seed was identical for all languages.
The performance of the gcc binary is virtually identical to STELLA, while the clang binary is slower. On average C is 19% slower.

heat is a simulation that iteratively computes the finite difference to approximate the differential equations for heat transfer. For STELLA we had to separate the GUI from the simulation logic—a modification that is considered good practice even for the original Python program.
Both C binaries are faster than STELLA, on average 23%.

This benchmark runs faster in its C version, but the STELLA slowdown of about 20% slower is acceptable and we have not yet fine-tuned

In summary, STELLA is on average 7% faster than C. Without the micro-benchmark "fib", STELLA is 0.1% slower than C. If the C compiler producing the fastest executable is used for every benchmark, then STELLA is 12% slower. On the other hand, if the C compiler that creates the slowest executable is used for every benchmark, then STELLA is 26% faster. While these differences are significant, the performance of STELLA is still in the same category as C's performance. This can especially be seen by comparing against the native Python performance. Note that the STELLA run-time includes some overhead which is not present in the C versions, namely LLVM's JIT compilation cost. Since this overhead is required for running a STELLA program, it is correct to include it in the comparison. The time for the analysis and creation of LLVM IR is not included in this comparison.

5. RELATED WORK

It is no secret that Python's performance can at times be inadequate. Therefore many projects exist with varying scope and use cases [12].

Speeding up all of Python: PyPy [18] is a Python virtual machine (VM) written in Python, or rather RPython (see below). It follows a very modular approach to VM construction which uses many different layers. It is incompatible with CPython's modules written in foreign languages, just like STELLA, but does not offer a staging phase where such modules can be used freely. While for many benchmarks PyPy runs much faster than CPython, the speed-up was not significant for scientific programs of interest to us.

Falcon [17] is an optimizing bytecode interpreter for Python. It is implemented as a Python library in C++ and applies aggressive optimizations, but since it implements all of Python semantics faithfully, its potential speed-up is limited.

Unladen-Swallow [21] was started as a branch of CPython to generally speed up the default Python run-time. The project stalled shortly after its inception for a multitude of reasons [10], which can all be linked to supporting the complete language with all its complexities.

Shedskin [2] is an experimental Python to C++ compiler. It supports all statically typed Python programs as long as they only use standard library functions, which have been re-implemented. It is similar to STELLA in that it separates from the Python run-time, but again aims to support the complete language. The implementation is not mature yet, and errors are difficult to interpret since they occur at the C++ level.

Speeding up some of Python: Cython [3] allows easy calling of C libraries as well as Python code by generating C source code, which is then compiled into a CPython module. Cython can either faithfully translate arbitrary Python code into C using the Python run-time, or be used as a language extension. Then it is very similar to Python but declares types statically. This allows a translation to pure C code if only C variables are involved in a computation. Cython does support OO programming with its "extension types". Overall its goal is to provide transparent integration with Python, which makes it difficult to predict program performance.

Numba [16] is a compiler for Python with support for the scientific software stack, in particular NumPy. It uses LLVM to compile to machine code, and integrates seamlessly into Python. Numba restricts the supported language features so that many features are unavailable at the present time. OOP is supported, and tries to integrate with Python. Numba allows easy calling of Python code as well as native libraries. Even though compilation happens at run-time, Numba inspects the source code. The current implementation focuses on translating single functions.

RPython [1] is a restricted subset of Python aimed at executing efficiently on a virtual machine (VM) built for statically typed languages. It was developed for the PyPy project. The program is generated by a bootstrapping full Python interpreter; the RPython code is then generated from the live objects, and translated into a back-end language, e.g., JVM or CLI. Programs must be statically typable and some dynamic features are disallowed. The focus on removing dynamic language features is somewhat different from STELLA, as it is aimed at better VM support and not necessarily efficient execution in general.

SEJITS [5] describes a general approach to selectively specialize a program. It has a strong focus on alternative hardware targets, e.g., multicore, GPUs. In contrast to our work SEJITS also explicitly integrates with the host language, allowing calls and interaction with the slow Python run-time. Copperhead [6] is an implementation of SEJITS for specializing Python code to a CUDA back-end and hence focuses strictly on parallel processing.

Other language projects: Terra [8] is a language which builds on lua. It recognizes the importance of being able to execute code independently from the host language run-time and implements lua-style basic OOP. In contrast to STELLA it focuses on multi-stage execution, is a more general framework, and requires more extensive source changes.

6. CONCLUSION AND FUTURE WORK

STELLA is a new embedded domain-specific language initially aimed at writing scientific simulations within Python, which will be compiled and executed at C-like speed. The approach goes against the trend of complete integration, meaning that it is not possible to call Python code from within the DSL. However, this separation makes it easier for the programmer to write fast programs.

It is also crucial to have modern code management practices available in the form of object-oriented programming, so important OOP patterns are implemented to enable easy code reuse as well as specializations. This support does not compromise run-time performance, as the OOP patterns are rewritten during compilation.

Our prototype implementation succeeds at enabling programmers to write very fast simulations while still having OOP available and many of the comforts of modern high-level languages. In other words, STELLA enables scientist-programmers to work within an easy-to-use platform which gives them many of the modern tools they expect, to avoid much of the tedium of lower-level languages, and to focus their efforts on the science rather than the management of code. The finished prototype implementation is available at https://github.com/squisher/stella.

There are plentiful opportunities for future work. Dictionaries and sets are common types used in Python programs. Fully supporting these types would be against the spirit of STELLA, since using these types adds hidden costs to programs. However, more limited operations should be supported. E.g., constructing a dictionary or set in in Python, and then providing a frozen interface in STELLA. Additional object-oriented features could be supported. In particular the so-called *magic methods*, e.g., for custom subscription implementation, would be useful syntactic sugar. While general *exception*

handling is likely to incur too much run-time overhead, some limited functionality to abort the DSL execution and report errors back to Python would be helpful. Some *functional features*, such as list comprehension, could be supported with an improved static analysis. Fine-grained numeric data type detection, e.g., shorter-width or unsigned integers, could reduce the memory footprint and possibly also speed up arithmetic operations, but would require an improved static analysis and enhanced intermediate type information.

7. ACKNOWLEDGMENTS

We would like to thank Peter Ruymgaart for providing the heat benchmark. This material is based upon work supported by the National Science Foundation under grant CCF-1422840.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *OOPSLA 2007 Proceedings and Companion, DLS'07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64. ACM, 2007.

[2] H. Ardo, B. Blais, P. Boddie, et al. Shedskin, June 2013. URL http://code.google.com/p/shedskin/. Retrieved 2013-06-26.

[3] S. Behnel, R. Bradshaw, C. Citro, et al. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, March–April 2011. ISSN 1521-9615.

[4] A. Bortz, M. Kalos, and J. Lebowitz. A new algorithm for Monte Carlo simulation of Ising spin systems. *Journal of Computational Physics*, 17(1):10 – 18, 1975. ISSN 0021-9991.

[5] B. Catanzaro, S. Kamil, Y. Lee, et al. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009.

[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Principles and Practices of Parallel Programming, PPoPP'11*, pages 47–56, 2011.

[7] P.-E. Dagand, A. Baumann, and T. Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *Proceedings of the Fifth Workshop on Programming Languages and Operating Systems*, PLOS '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM.

[8] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[9] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25): 2340–2361, http://dx.doi.org/10.1021/j100540a008, 1977.

[10] R. Kleckner. Unladen Swallow Retrospective. *QINSB is not a Software Blog*, March 2011. Retrieved 2013-06-04.

[11] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6:261–301, 1974. "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil".

[12] P. Krill. Van Rossum: Python is not too slow, March 2012. URL http://www.infoworld.com/d/application-development/van-rossum-python-not-too-slow-188715. Retrieved 2013-04-09.

[13] H. P. Langtangen. *Python Scripting for Computational Science [electronic resource] / edited by Hans Petter Langtangen.* Texts in Computational Science and Engineering: 3. Berlin, Heidelberg : Springer Berlin Heidelberg, 3rd edition, 2008.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] M. J. Olah, D. Mohr, and D. Stefanovic. Representing uniqueness constraints in object-relational mapping - the natural entity framework. In *TOOLS Europe*, 2012.

[16] T. Oliphant, A. Valverde, D. Christensen, et al. Numba, 2012. URL http://numba.pydata.org/. Retrieved 2013-07-18.

[17] R. Power and A. Rubinsteyn. How fast can we make interpreted Python? *ArXiv e-prints*, 1306.6047, June 2013.

[18] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[19] O. Semenov, D. Mohr, and D. Stefanovic. First-passage properties of molecular spiders. *Phys. Rev. E*, 88:012724, Jul 2013.

[20] M. J. Turk and B. D. Smith. High-Performance Astrophysical Simulations and Analysis with Python. *ArXiv e-prints*, 1112.4482, Dec. 2011.

[21] Unladen Swallow. Unladen Swallow, 2010. URL http://code.google.com/p/unladen-swallow/. Retrieved 2015-06-09.

[22] S. van der Walt, S. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011. ISSN 1521-9615.