

12-1-2015

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

David Mohr

Follow this and additional works at: https://digitalrepository.unm.edu/cs_etds

Recommended Citation

Mohr, David. "Stella: A Python-based Domain-Specific Language for Simulations." (2015). https://digitalrepository.unm.edu/cs_etds/12

This Dissertation is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Computer Science ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

David Mohr

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Darko Stefanovic, Chairperson

Manuel Hermenegildo

Lydia Tapia

Matthew Lakin

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

by

David Mohr

M.S., The University of New Mexico, 2010

B.S.C.S., The University of Texas-Pan American, 2006

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December 2015

©2015, David Mohr

Dedication

To all the open source and free software developers.

Acknowledgments

First of all I would like to thank my advisor Darko Stefanovic. I am very grateful that he gave me the freedom to find my own project, something that truly interests me, and then guided me to turn the small initial idea into the mature project that now lies in your (virtual) hands.

My committee, Manuel Hermenegildo, Lydia Tapia, and Matthew Lakin, were a pleasure to work with. Thank you for your time and effort!

I would also like to thank my lab mates Mark Olah and Oleg Semenov, who were fun to work with, fun to hang out with, and whose work gave rise to the idea for my project. And my friend Drew Levin for reviewing this document, and for keeping me steadily supplied with coffee when time was getting tight.

My project involved some heavy implementation effort, and I would like to acknowledge that it would not have been possible to prove my point without the excellent work done before me. Thank you for all the libraries and programs, little and small, that I used, and which were made graciously available for free by their authors.

This material is based upon work supported by the National Science Foundation under grants CCF-0829896, CDI-1028238, and CCF-1422840.

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

by

David Mohr

M.S., The University of New Mexico, 2010

B.S.C.S., The University of Texas-Pan American, 2006

Ph.D., Computer Science, University of New Mexico, 2015

Abstract

STELLA is a domain-specific language that (1) has single thread performance competitive with low-level languages, (2) supports object-oriented programming (OOP) to properly structure the code, and (3) is very easy to use. Instead of prototyping in a high-level language and then rewriting in a lower-level language, STELLA is embedded in Python, is transparently usable, retains some OOP features, compiles to machine code, and executes at speed similar to C. STELLA's source code is compatible with Python, and allows easy integration of C libraries. Its features are focused on the needs of scientific simulations. Other projects to speed up Python focus on easy integration, and smaller critical sections. In contrast, STELLA supports translating larger programs in their entirety, and does not allow interaction with the Python run-time, to ensure predictable performance. My experience developing STELLA shows that by carefully selecting language features, high run-time performance can be achieved in a high-level language that has in practice very few restrictions.

Contents

List of Figures	xiv
List of Tables	xv
0 Preface	1
0.1 On Prototyping	1
0.2 On Lower-Level Languages	3
0.3 On Object-Oriented Programming	5
0.4 Many Shoes that Don't Fit	6
0.5 Creating a Domain-Specific Language	8
1 Introduction	10
1.1 Motivation	10
1.2 Performance-Critical Sections	13
1.3 The STELLA Language	15
1.4 Related Work	17

Contents

1.4.1	Speeding up all of Python	17
1.4.2	Speeding up some of Python	19
1.4.3	Other language projects	22
1.5	Contributions	23
2	Example	24
2.1	Pre-Processing Stage	24
2.2	DSL Stage	27
2.3	Post-Processing Stage	28
2.4	OOP Extension	29
3	Design	31
3.1	Language Characteristics	32
3.2	Embedding	36
3.3	Syntax and Semantics	37
3.4	Syntactic Sugar	38
3.5	Typing	40
3.5.1	Static Typing	40
3.5.2	Types	41
3.6	Arrays	41
3.7	Typing Rules	43

Contents

3.8	Object-Oriented Programming	44
3.9	Variable Scope	47
3.10	Optimization Passes	48
3.10.1	Deviations from Python Semantics	48
3.11	Interfacing with C	49
4	Implementation	51
4.1	Dependencies	51
4.2	LLVM Primer	52
4.3	Intermediate Representation	54
4.4	Analysis	55
4.4.1	Function Objects	56
4.4.2	Disassembly	57
4.4.3	Rewrite	58
4.4.4	Intra-procedural Control Flow	58
4.4.5	Stack Unwinding	59
4.4.6	Type Analysis	60
4.4.7	Inter-procedural Control Flow Discovery	61
4.5	Intrinsics	62
4.6	Types	62
4.7	Code Generation and Execution	63

Contents

4.7.1	C libraries	64
4.7.2	Data Transfer	65
4.8	Verification	66
4.9	Debugging	67
4.10	Bytecodes	68
4.10.1	Basic Language Support	68
4.10.2	Arithmetic	69
4.10.3	Memory Interaction	71
4.10.4	Control Flow	72
4.10.5	Placeholders	73
5	Evaluation	75
5.1	Benchmark Description	76
5.2	Language Design	77
5.2.1	Required Program Modifications	77
5.2.2	Summary of Source Changes	79
5.3	Compilation Time	80
5.4	Performance	83
6	Conclusion	85
6.1	Future Work	86

Contents

A Benchmark Details	90
A.1 Fibonacci	90
A.1.1 Python and Stella Source	91
A.1.2 C Source	91
A.2 1D Spider	92
A.2.1 Python and Stella Source	92
A.2.2 Change Summary	95
A.2.3 C Source	95
A.3 nbody	99
A.3.1 Python Original	99
A.3.2 Modified for Stella	102
A.3.3 Change Summary	107
A.3.4 C Source	109
A.4 heat	112
A.4.1 Python Original	112
A.4.2 Modified for Stella	122
A.4.3 Change Summary	133
A.4.4 C Source	134
A.5 Performance per C Compiler	142
B Miscellaneous Source Code	143

Contents

B.1	Exploratory Program Details	143
B.1.1	GenericSpiderSim	144
B.2	mtpy	153
B.3	Example Program SpiderSemiInfinite1D	154
B.4	Example Helper Code	156
B.5	Example Program SpiderSemiInfinite1D-Fpt	158
C	STELLA Source Code	160
C.1	stella/analysis.py	160
C.2	stella/tp.py	172
C.3	stella/intrinsics/python.py	209
C.4	stella/intrinsics/_init_.py	210
C.5	stella/_version.py	218
C.6	stella/codegen.py	229
C.7	stella/ir.py	234
C.8	stella/bytecode.py	252
C.9	stella/test/debug_gc.py	292
C.10	stella/test/errors.py	294
C.11	stella/test/external_func.py	296
C.12	stella/test/benchmark.py	298
C.13	stella/test/basicmath.py	306

Contents

C.14 stella/test/virtnet_utils.py	310
C.15 stella/test/langconstr.py	312
C.16 stella/test/si111s_struct.py	330
C.17 stella/test/objects.py	334
C.18 stella/test/confest.py	350
C.19 stella/test/si111s_globals.py	352
C.20 stella/test/typing.py	357
C.21 stella/test/__init__.py	360
C.22 stella/storage.py	363
C.23 stella/exc.py	366
C.24 stella/utils.py	368
C.25 stella/__init__.py	374

List of Figures

1.1	Core simulation structure	14
3.1	Memory layout for single inheritance	46
3.2	Possible memory layout for multiple inheritance	46
4.1	Analysis overview	55
5.1	Comparison of compilation times	82

List of Tables

1.1	Comparison of exploratory simulation throughput	12
5.1	Benchmark run-times compared	83
A.1	Benchmark run-times of individual C compilers	142

Preface

First I would like to offer some context to how my dissertation came to be. This is a personal narrative for the interested reader. However, the information herein is not required for understanding the main body of this manuscript.

0.1 On Prototyping

The journey for this work started with a simple assignment: Our research lab had a kinetic Monte Carlo (KMC) simulation [11], written in Python, for an abstract model that we were investigating at the time [44]. Even though each individual simulation seemingly ran fast, all together it would have taken weeks to collect the number of samples that we needed to get statistically valid results. At the time I was investigating running similar simulations on a GPU¹, so I already had my C hat on and volunteered to provide a faster implementation by translating the simulation to C.

Initially I looked at the simulation and thought: “This will be easy; it is a simple code base, which shouldn’t take me more than a day to rewrite.” Memory does not serve me

¹Running multiple simulations in parallel on a GPU turned out to be ineffective due to excessive memory access.

Chapter 0. Preface

well here, but I do remember that it took several times longer. The process of translating a program from a higher-level language to a lower-level one does require some thought: Python provided so much functionality that isn't present off the shelf in C, and therefore I had to think about implementation details more often than anticipated. Then there are types—which are straightforward in most cases, but since I was not the original author, I did not know by heart what each variable was supposed to hold and had to frequently search the Python program to find where the variable in question was initialized to verify its type. While I was doing all of this, I was frequently thinking: “This step could be automated, and that step, too, could be automated.” I think any respectable computer scientist dislikes doing work by hand that could be done by a program.

Another frequent thought was that some Python features are nothing but syntactic sugar, but that this sugar both saved me from having to implement common tasks by myself (e.g. 2D array indexing), and that it made the code much easier to read. These features were a reminder that high-level languages really make the programmer more productive, but also that this comes at the price of slow run-time performance. And amongst the general purpose high-level languages this is particularly true for Python.

Once finished, the simulation ran blazingly fast when compared to the Python implementation (see also Table 1.1). The stellar performance we gained turned out to be more than just a convenience: it was a real necessity. Collecting the data for all combinations of the simulation parameters *still* took several days, even though the sample collection was running in parallel on all our computers. The speed-up of the C implementation is not only explained by language differences. During the rewriting process I applied obvious optimizations to make the program more efficient. This seemed natural, because I had the finished prototype and thus knew exactly what functionality was required. The lower-level nature of C makes it easy to implement optimizations by omitting the flexibility that Python automatically provides, but wasn't actually used in the program at hand.

The flip side is that the source code became more specific to the simulation at hand.

This is particularly annoying for research software. Since we must use exploratory work to determine which simulations are worth spending CPU time on, the specific nature of the C rewriting makes it difficult to reuse that effort later on. There are two options: 1) either we maintain the Python prototype as the reference implementation, 2) or we try to make the C version more flexible. Both have significant drawbacks. If we keep working with the Python version, then once the exploratory work is done significant effort may have to be spent to adapt the previous C rewrite to the current simulation. This is also an error-prone process because changes are likely to be necessary in many places. On the other hand, making the C version more flexible requires significant effort because of the low-level nature of C. This also poses the danger that optimizations will be lost while generalizing the code. One of the reasons that structuring is difficult in C is that there is no native support for modern structuring techniques, like object-oriented programming.

0.2 On Lower-Level Languages

Computers have become so fast that for many tasks the deciding factor to choose a programming language is not its run-time performance. This is partially due to the fact that CPUs run at such high clock rates that other limiting factors exist, such as input/output, network communication, user interaction, or even memory access times. Instead, the decision is often made based on library availability as well as tool and IDE support. This has made high-level scripting languages, such as Python, very popular because in many situations their overhead is hardly noticeable. Furthermore, the high-level programming approach has made it possible for many scientists and other users with little or no training in computer science to also write programs. I argue that this is another reason why simply rewriting programs in lower-level languages like C is not an ideal solution, because it is not immediately available to those user groups – low-level rewrites would then have to be out-sourced to other groups or individuals, which creates a new set of challenges and

Chapter 0. Preface

comes at a much higher price.

The usage scenario matters, too. I don't mind spending the time to write low-level code when I know that my program will be run by thousands of people around the world. When the program has a large user base, then in some sense the effort on its programming becomes amortized by the accumulated CPU time that is being saved. Scientific code, however, is often written to calculate some answer (e.g., a simulation provides insight into some behavior) and is not widely distributed outside of the research group where it was developed. While this code can still consume a significant amount of CPU time, it does lessen the incentive to develop and maintain lower-level code.

The convenience of higher-level languages is especially felt when it comes to the pre- or post-processing of the data. Lower-level languages usually write the resulting data out to disk, and then a second program, often written in a higher-level language, parses the data, processes it, and outputs graphs or other metrics. When working in a high-level language, there is no interfacing or data conversion overhead for these tasks.

A different option is then to implement the performance critical part in a lower-level language, but integrate it as a library into the higher-level language. Then pre- and post-processing remains seamless, and this is in fact the recommended strategy when Python code is not performing well [31]. Yet this approach again utilizes C which brings with it the shortcomings of lower-level languages already discussed.

Transitioning from one run-time system, Python, to another, the optimizing language, can also be costly: in my experiments Python *with* NumPy arrays [51] turned out to be slower than the implementation which uses native Python, despite the fact that NumPy was written to speed up numerical computation in Python. The operations in the benchmark manipulate only small data and not large matrices, so that the transition cost is not amortized by the highly efficient array implementation that NumPy provides. This is very similar to only rewriting a part of a simulation in a lower-level language.

There is also the issue with code reuse: while working on a topic a scientist often creates many different simulations, which are similar but explore different properties. 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 possible to (easily) use an object-oriented programming (OOP) style. Of course one can write C programs in an OOP-style, the excellent GObject² library is a prime example. But it is also very cumbersome to use because often long code templates are required to perform basic tasks. Its main use is as a lightweight base for the GUI toolkit GTK. Since desktop applications often run in the background, the low memory footprint and careful resource usage that is possible with GObject/GTK can make the higher programming effort worthwhile. In contrast, scientific applications are expected to use all available resources, and favor faster development times.

0.3 On Object-Oriented Programming

I believe that it is important to have modern code structuring techniques available, and object-oriented programming has a long history [40]. In fact, writing simulations gave rise to object-oriented programming in the first place [9]. It encourages the programmer to structure the program by designing classes, each of which roughly corresponds to a real-world object or abstract entity. This allows a separation of concerns, i.e., decoupling functionality without strong dependencies. Thus it enables better code reuse, and easier introduction of specializations. Inheritance is another essential feature of OOP: a new class is based on an existing class, its *parent*, but has additional functionality and/or deviates from the way the parent behaves (*overriding*).

²<https://developer.gnome.org/gobject/stable/>

Optimizing OOP has been a research topic for many years, e.g., [18, 9, 14, 19], and remains an active research area. OOP features often require support at run-time, therefore they can potentially be detrimental to performance. The programmer may then end up “optimizing” a program by rewriting the source code to avoid OOP—but this comes at the significant loss of modularity and functionality. In my opinion this is an entirely undesirable course of action, because I think OOP allows the programmer to follow good software engineering disciplines. “We should not optimize prematurely” stipulated Knuth [30], but optimizing in later stages makes it a challenge to keep code compatible with different simulations. OOP makes it easier to properly structure optimizations: more specific ones can be implemented by subclasses that override the universal functions. Then they are then isolated from the more general case while using them remains transparent to the remainder of the code.

OOP is not the only methodology to structure code, e.g., functional programming offers an alternative which can be used to achieve the same goals of code reuse and to allow specialization. Python does support functional programming, but the support is not complete [28]. In most Python programs functional programming is not used for the main structure of a program, but instead for smaller tasks within a program that is structured in a procedural or object-oriented style.

0.4 Many Shoes that Don’t Fit

Now let me take a step back to the beginning of my work. I had the original Python simulation, a C version, and had decided on requirements for a programming language that I outlined in the previous two sections. After starting to run the C simulations storing the resulting data became an issue. I decided that the HDF5 format [48] would be the appropriate solution, and began to integrate it into the C program. HDF5 has many bindings, therefore it would be easy to integrate it into the program for data analysis, which was

Chapter 0. Preface

written in Python. But it quickly became apparent that even now there was a duplication of effort because I was accessing the same data structure from two languages and when the file format changed they would get out of sync.

So I thought that maybe a better solution would be to integrate the C simulation directly into Python. Normally C modules for Python must implement the standard Python interpreter (CPython) API, which naturally has a higher learning curve. Cython [7] makes this process much easier: with some static type annotations it automatically generates the appropriate C code to allow calling the C methods from Python. Now the existing simulation was comfortably usable, but the approach had required much manual labor. How could I avoid writing future simulations in low-level code, but get fast run-time performance while having OOP available, and easily integrate into an existing software ecosystem?

Because of the Python original, it was natural to experiment with alternative Python run-times first, since this would involve the least effort. PyPy [41] is a just-in-time compiler for Python. Since its main drawback, the incompatibility with C modules did not apply to the pure Python simulation, it seemed like a good choice to improve performance. However, it ran virtually exactly as fast as CPython! This bad result made me abandon PyPy very quickly. I briefly examined RPython [5], but since it shipped only as part of the PyPy source, and had little usage instructions, I was not able to actually try running the simulation with it.

I knew from previous work that Java's HotSpot VM had seen many speed improvements in recent years, so it seemed worthwhile to try to use Jython[27], which implements the Python language on top of the JVM. But the results were similar to PyPy: it ran virtually at the same speed, and therefore was an uninteresting alternative. The last Python implementation that I was able to find was Shedskin [6]: it transforms Python code into C++, including its own implementation of the Python standard library. While not all default modules had been implemented yet, the support was complete enough to try to run our simulation. It did not compile, and instead produced pages of C++ template error

messages that I was not able to decipher to pinpoint the source of the issue. Shedskin illustrates the problem with source-to-source transformations without sufficient semantic validation, although a simple bug would manifest itself in similar ways.

Instead of trying to get Shedskin to work, I chose to manually translate the simulation to C++. I was interested in how fast C++ itself is, and would at the same time establish a baseline for the performance of Shedskin. C++, compiled with gcc, performed much better than any Python implementation, which was to be expected. But its performance still fell short by a factor of 2 when compared to the C implementation. That is still too slow.

0.5 Creating a Domain-Specific Language

When no existing solution came even close to the performance that we required, or had the programming language features that we would like to have, I decided to create a new domain-specific language (DSL), STELLA. In contrast to speeding up all of Python, I thought that a more restrictive subset of Python could be successfully implemented in a way that satisfied our needs. For this new language to be actually useful, it would have to fulfill strict performance requirements. For example, given the slow performance of pure Python code, a $100\times$ speed-up surely is impressive. But if the C version runs about $200\times$ as fast, this discrepancy would *still* be a significant hurdle to adoption. If the data collection needs to run two weeks instead of one, spending two days to rewrite the simulation in C will look like the better option. Therefore STELLA's design reflects this reality: I created the language with the mindset that if it compiles, the programmer should be able to have the same level of assurance about the run-time performance as if he had written it in C. Obviously this does not mean that one can't write slow programs in STELLA – one certainly can write slow programs in C. But there should be as little hidden costs as possible and every individual statement should have a constant cost.

Chapter 0. Preface

This hints at static typing, as opposed to the dynamic typing that Python employs. While the latter gives the programmer increased flexibility, it also results in run-time type checking, which is exactly one of these hidden costs that I want to avoid. Static typing also has some advantages compared with dynamic typing: It can catch errors early that otherwise would only show up at run-time. A real example is misspelling a variable name when returning the simulation results to the post-processing. In Python this would mean that all results are lost, and the CPU time is wasted³. While some tools exist that can perform simple static analysis on Python code⁴, and can catch some errors, they still fail in many situations. Therefore static typing adds additional safety, while simultaneously being easier to implement.

This goal of “constant cost” also means that the language can’t support the dynamic features that Python offers: modifying objects at run-time, executing code contained in a string (i.e., `eval()`), or being able to treat virtually everything as a first-class value. This is restrictive, but also an effective tool to achieve the performance goals. And the restrictions are mitigated in two ways: by being implemented directly in Python, the DSL can perform automatic data transfer, there is no need for additional glue as there would be if implementing directly in a lower-level language. Secondly, by reusing the Python syntax STELLA leverages many of the facilities that Python provides in a transparent manner. It creates its static code based on a snapshot of the program at the time the DSL is invoked. Therefore the full power of the programming language is available when it is needed the most: for initialization, data management [36], post processing, etc. And the simulation core can run with the full speed of a C-like implementation in STELLA, simply by avoiding some language features.

³The problem of a misspelled variable name is also reduced by utilities like `iPythonNotebook`[38], which automatically save intermediate results. However, if this occurs in the middle of the computation, then static typing still solves the problem more elegantly and efficiently.

⁴Example static analysis for Python are `pyflakes`[2], `PyChecker`[1], and `pylint`[3].

Introduction

1.1 Motivation

Scientists who write custom domain-specific simulations to explore a mathematical model of natural phenomena, such as Kinetic Monte Carlo (KMC) simulations [11], face a difficult choice when they must select a programming language. High *execution speed* is of great importance since the stochastic simulation process requires many executions of the same program to yield statistically significant results. This hints at C or Fortran as the language of choice, since it is usually the lowest level a scientist is willing to program in. 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 easy to use an object-oriented programming (OOP) style. Lastly, another software engineering discipline that is important but difficult to follow when writing in C is *properly structuring optimizations*. We should not optimize prematurely [30], but optimizing in later stages makes it a

challenge to keep the code compatible with different simulations.

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 simulation in C. This takes advantage of the high productivity HLSLs allow but does not solve the code reuse issues raised above. 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 [51, 33, 49], not least because it encompasses rich libraries for numerical computation (e.g., NumPy and other SciPy projects), data plotting (e.g., matplotlib), and analysis (e.g. pandas). For Python it is the recommended practice to refactor the performance-critical code into a separate module and then re-implement that in a lower-level language [31]. This may be a feasible approach for software that gets widely distributed and reused by many since then the additional effort is amortized. When the critical section is small, then this is sound advice even for scientific projects. However, there are classes of programs where the critical section is broad, such as KMC simulations (see Section 1.2). In that case re-implementing the performance-critical section can be tantamount to rewriting the whole program. Then, in many ways, it turns into nothing else but using Python as a prototyping language: there is a rewriting cost and maintenance issues as noted above.

It may seem that an alternative would be to refactor the critical section not as a whole, but piece by piece, into a lower-level language. This approach may make the lower-level code more manageable, but it will also introduce many transitions into and out of the scripting language at run-time. This can be very costly, for example because the data representation is different: Python uses boxed scalars, whereas lower-level languages consistently use machine types. This ends up negating much of the speed benefit.

Therefore, it is desirable to stay within Python, where the full array of modern lan-

Implementation	Variant	Throughput	Speedup	Slowdown
NumPy	generic	0.00319	1.0	1793
Jython	generic	0.00855	2.7	669
CPython	generic	0.00861	2.7	664
PyPy	generic	0.00882	2.8	648
CPython	2D	0.01659	5.2	344
C++	generic	0.27894	87.5	20
C	generic	0.89286	280.1	6
C	2D opt	5.71429	1792.6	1

Table 1.1: Throughput of different implementations of the exploratory program “GenericSpiderSim” in simulations per second. The benchmark is a KMC simulation from [44]. *Speedup* compares the throughput against the slowest implementation, *Slowdown* compares it against the fastest. See Appendix B.1 for some more details.

guage features is available, but unfortunately the performance penalty is prohibitive, running up to three orders of magnitude. Table 1.1 explores this based on a KMC simulation. It is common knowledge that the performance of native Python code is not sufficient for computationally intensive code [31]. Not surprisingly, then, several existing approaches to speeding up Python code have been developed, some of which are also shown in the Table 1.1. While they do provide a benefit, they all come rather short of the C-like performance that is desired (see related work in Section 1.4). In summary, the Achilles’ heel of all these approaches to speeding up Python is that they either aim to support the complete language, integrate so well with Python that calls into the slow run-time happen inadvertently, or focus mainly on parallel execution:

- It is inherently difficult to develop a compiler for a feature-rich and dynamic language such as Python. This is particularly true when it has to create very efficient machine code to enable the high execution speed that is required. While it may be possible to create a very effective optimizing compiler for the complete Python language, this would require an extraordinarily large engineering effort.
- Transparent integration with Python is the ability to use native Python data types and

functions when the DSL does not support the features that the simulation is trying to use. This approach has two main advantages. It is convenient for the programmer, because using the DSL does not restrict him in any way compared to fully-fledged Python. It also makes it easy to optimize small sections of the program, with Python functionality interspersed. On the other hand, transparent integration has inherent drawbacks; the convenience, by definition, means that the programmer does not know when he is using Python functionality and when the DSL's. Therefore it is easily possible to trigger unintended performance penalties by using features that are actually handled by the Python run-time.

- Given today's hardware, e.g., multi-core CPUs, GPUs, or high-performance computing clusters, there is a strong focus on parallel computation in many research projects on scientific computing. However, not all algorithms are parallelizable—some are inherently sequential. Therefore continued research into optimizing single-thread performance, such as presented in this dissertation, continues to be beneficial. More so, stochastic simulations are actually *embarrassingly parallel*; many samples have to be collected to yield statistically significant results, and these samples must be independent as otherwise the statistics will be skewed. Therefore we achieve the least overhead by focusing on single-thread performance, while utilizing any available CPU with independent runs of the simulation.

1.2 Performance-Critical Sections

The previous section mentioned that some programs have a broad critical section. Here one such program is examined where it is more difficult to rewrite the performance-critical section in a lower level language: Kinetic Monte Carlo simulations. For many programs 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.

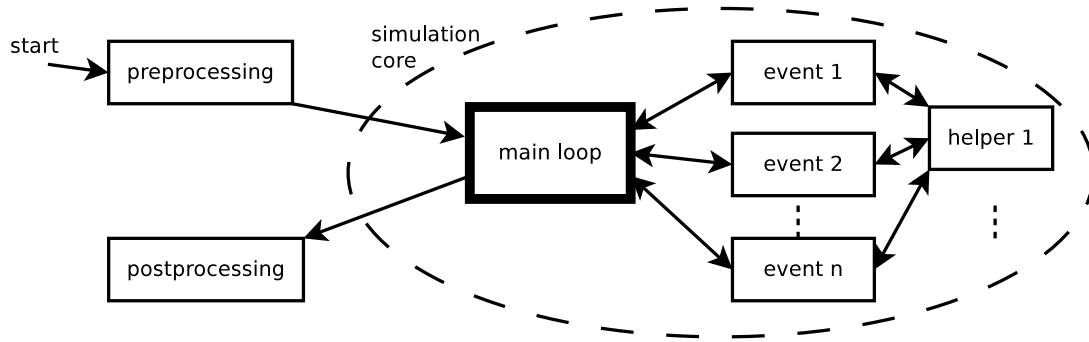


Figure 1.1: Simplified structure of a typical simulation core.

Consider the structure of a typical simulation (Figure 1.1): there will be a main loop, which keeps track of the progress of the simulation, and often also decides what event is currently being simulated. Then, depending on the event, it will call some function to evaluate the effects. These functions may be calling some helper functions. It is easy to see that the main loop will be called very frequently. But if only the main loop is rewritten, then the program will constantly transition from the high-level to the lower-level language and vice versa, as the main loop will be constantly calling the event handling functions.

An attempt at a workaround would be to implement only the most frequently called subset of functions, and the main loop, in the lower-level language. 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!

That transitioning from one run-time system, Python, to another, the optimizing language, can be costly is visible in Table 1.1 in the *NumPy* row: it is slower than the implementation which uses just Python, despite the fact that NumPy was written to speed up numerical computation in Python. The operations in the “GenericSpiderSim” benchmark manipulate only small data and not, e.g., large matrices, so that the transition cost is not amortized by the large speedup that NumPy is able to provide when operating on larger data sets. This is a very similar situation to when only part of a simulation is rewritten in

a lower-level language.

1.3 The STELLA Language

Since no existing solution fulfilled the requirements, the domain-specific language (DSL) STELLA with features selected for custom simulations was developed for this dissertation. With the reduced feature set execution speed can be a priority, allowing the DSL to be compiled into native code rather than being interpreted. Modern DSLs are usually allowed to make calls into the host language. But since the host language, Python, is slow, this would make it too easy for the programmer to unintentionally slow down the execution of STELLA programs. In fact, all expensive language features are avoided in STELLA, resulting in a C-like language where it is obvious 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. Static typing has been employed by other projects already, e.g., Cython, and [mypy](#). This loss of functionality will be compensated in part by staging the execution [46]: STELLA code is constructed within 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: parsing command-line parameters, configuration, and state files, accessing databases, and coordinating the simulation execution are just some examples.

Properly structuring simulations and frameworks should be done with OOP, therefore this support is deemed essential. Full OOP support is expensive: it takes a lot of 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. Additionally, the dynamic features of Python can

Chapter 1. Introduction

be used to assemble classes and objects at run-time. At compilation time the OOP is reduced to procedural-style code. The benefit is that OOP can be used as an engineering tool to solve the code reuse and optimization/specialization structuring problems.

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 libraries that are available in the Python ecosystem, because now the complete language is available again.

There are some inherent trade-offs in this design: Even though it shares Python's syntax, STELLA is a different language, and some adjustments are necessary for existing Python simulations before they are valid STELLA programs. However, for scientific simulations these restrictions have particularly low impact:

- Even if it were possible to make use of the Python run-time, this would in all situations be ill-advised because of the performance impact it would have in the simulation core. Therefore the inability to make calls into the Python run-time is a small draw-back in practice.
- The same argument can be made for implementing expensive language features natively in STELLA; if they were available, they should be avoided because of the negative impact on performance.
- Memory preallocation is a common optimization for performance critical code. The programmer performs this change anyway; in STELLA it is simply required from the outset. Note that this does not impact the programmer during the development phase, because at that time the simulation can be run within Python.

In practice only small changes were required. The simplicity of the language and familiar OOP patterns make it extremely easy to learn the language.

1.4 Related Work

The multitude of projects trying to improve the performance of Python are evidence to the real-world need to speed-up Python programs. Below existing projects, and their relevance to STELLA, are discussed. Note that the reference implementation of Python is called *CPython*.

The presented solutions either try to optimize the complete language, are incomplete, do not come close to the performance that we required, or they focus on special execution environments, e.g., parallelization or execution on the GPU. Yet parallel execution is not very useful when collecting samples is already *embarrassingly parallel*¹. Automatic execution on a GPU is an exciting field, but also does not always yield a speed-up [26]. Some parallelizable programs are not well suited for running on the GPU [10].

1.4.1 Speeding up all of Python

PyPy [41] 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. A flexible type system and type inference is used at each layer. PyPy's goal is to support languages as varied as C/Posix, .NET/CLI, and the JVM with reasonable efficiency. Today the project has implemented a JIT compiler for Python. One downside is that it is incompatible with CPython's modules written in foreign languages, so that, e.g., at present NumPy is not completely working within PyPy.

The PyPy project continuously updates their [benchmark results](#). It shows that there is a significant improvement over CPython in many cases. PyPy implements the complete Python language. While in special cases it is possible to get C-like performance, the

¹*Embarrassingly parallel* is an odd, but established term for problems that are inherently parallel, i.e., it is trivial to split the computation among multiple CPUs.

specification of Python makes it difficult for the programmer to know how to program to be able to achieve such high performance execution.

Falcon [39] is an optimizing bytecode interpreter for Python. Written mainly in C++ it is executed from within CPython and translates the Python bytecode, which is based on a stack machine, into their custom register based bytecodes. This allows more aggressive optimizations to be applied. The Falcon authors' conclusion is that there are language elements of Python that make it difficult to optimize: Dynamic typing and the overloading capabilities result in the possibility of side effects for almost every instruction. This is the reason why STELLA is a more specialized language which uses static typing, and which does not cover the complete Python language. The speed-up compared with CPython is a maximum of 200%, and on average 25% [39], which is not a sufficient improvement.

Unladen-Swallow [50] was started by Google employees as a branch of CPython to generally speed up the default Python run-time. It is unclear how much progress was made, since the project stalled shortly after its inception. One of the main authors [29] wrote that the project failed for a multitude of reasons; among others the lack of optimizations for dynamic languages within LLVM [34]. STELLA has a much smaller scope than Unladen-Swallow, which attempted to optimize the complete language. Since STELLA is without dynamic features it interacts much better than Unladen-Swallow with the optimizations currently implemented in LLVM.

Shedskin [6] is an experimental Python to C++ compiler. It supports all statically typed Python programs. The standard library needs to be re-implemented and at present 25 commonly used modules are available. The result is either a standalone program, or a Python module. In both cases the code runs independent from the CPython run-time, instead using the supplied standard library modules. The reported speed-up compared to CPython varies greatly (from 2 to 200 times faster than CPython). The implementation is not mature yet, and errors are difficult to interpret since they occur at the C++ level.

Unfortunately the benchmark program “SpiderSemiInfinite1D” did not successfully run within Shedskin. It was unclear what caused the problem, since there was an issue preventing successful compilation of the resulting C++ program. This shows the difficulty of working with complete Python source code, even if it is explicitly restricted by the project, as well as the importance of a clear semantic validation during the translation process.

Jython [27] is an alternative implementation of the CPython run-time which compiles Python programs and runs on the JVM. Jython programs benefit from the optimizations implemented in the JVM, and the ability to easily interface with existing Java programs. This can result in a significant speed-up if the JIT compiler is able to optimize the code well. But many features of Python need to be emulated as custom code since Java’s bytecodes do not provide a convenient interface. This emulation is the likely reason why there is no performance improvement compared to plain Python.

1.4.2 Speeding up some of Python

NumPy [51] is a popular Python package for numerical computation. Its most prominent feature is the highly optimized N -dimensional array data type and many common operations on it. As previously discussed in Section 0.2, the performance degradation compared to plain Python for my comparison program is most likely caused by the frequent calls of NumPy functions while spending only very little time inside of NumPy.

Cython [7] focuses on easy integration of C libraries, as well as optimizing small program sections, e.g., single functions. There are several ways in which Cython can be used. It can handle arbitrary Python code, which it translates to C code that uses Python’s internal data types and processing functions – in essence statically compiling the parsing of the Python source code. Thus semantics are exactly preserved, but also the speed-up is often minimal since only the interpreting overhead is eliminated. Cython also

Chapter 1. Introduction

introduces a language extension to Python in which types are statically declared. This allows a translation to pure C code if only C variables are involved in a computation.

In Cython, usage of the Python run-time is transparent. This makes it easy to combine C libraries and Python code, since the Cython program is but only a Python module written in C, which has been automatically generated. But usage of the Python run-time can also happen unintentionally: either by accident, or because of a lack of knowledge of the language. STELLA has the opposite goal: isolation from the Python run-time. This is a restriction but also gives the programmer assurance that the Python run-time cannot introduce a slow-down. Cython does use a different syntax to directly translate to C, while STELLA will be completely embedded. Cython cannot use staging within Python, since the language is compiled prior to loading Python by an external C compiler. Also, since control can flow arbitrarily between Cython and Python, it is not easily possible, if possible at all, to do automatic type inference.

Cython does support OO programming with its “extension types”. These allow the programmer to define classes with members that have C data types. The classes are implementing the Python C-interface, so that they act like native Python classes, and can be used from within the CPython run-time. Since STELLA does not support interaction with the Python run-time, its OOP support has fewer requirements. It implements a more lightweight model which favors execution speed instead.

Numba [37] 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: dictionaries, list comprehensions, function default arguments. OOP is supported and tries to integrate with Python. Numba allows easy calling of Python code as well as native libraries. While the compilation is triggered at run-time, Numba inspects the source code to generate the Python AST and then translate it to LLVM

Chapter 1. Introduction

IR. A commercial port² exists, which adds a CUDA back-end. The current implementation focuses on translating single functions.

Numba focuses on numeric computations, but supports enough Python to enable bigger programs to be compiled. However, when compiling and using larger programs, that consist of several functions or classes, my experiments with using Numba were unsuccessful due to a bug. The seamless Python integration makes it difficult to judge at what time Numba code is executed or when Python is in control again.

RPython [5] 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 not allowed.

The language was designed particularly for implementing PyPy. While it is a complete language, RPython is not fully ready for general use. 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.

Weave [43] is a subproject of **SciPy** and includes C/C++ as strings in Python, and then compiles them on the fly. Interaction with the CPython run-time is possible by using the C module interface. There is very little documentation of the project, but the obvious trade-off applies: pure C code will run fast, interaction with the Python run-time will make it slow, and code structuring techniques remain difficult to use when programming in C.

Composing programs by manipulating source code as strings is generally not a good idea [46, 42]. This would be similar to macro programming in C, which is usually considered unsafe except in the most simple cases because of the lack of typing. Using Weave means the programmer has to understand both C and the C interface of Python.

²[NumbaPro](#) is a commercial offering by the same company that backs the Numba development.

In contrast STELLA is using the AST of Python, which is a typed representation and guaranteed to be correct, since it is not created externally but rather by a live CPython instance. This provides tighter integration, increases compatibility, and means the programmer is only confronted with one kind of syntax – Python’s.

SEJITS [12] describes a general approach to selectively specialize a program. It has a strong focus on alternative hardware targets, e.g., multi-core, 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 [13] is an implementation of SEJITS for specializing Python code to a CUDA back-end and hence focuses strictly on parallel processing. Technically the simulation core is also a *selective* specialization, the focus in my work is to enable a complete section of the code to be translated.

1.4.3 Other language projects

C++ [52] first appeared in 1983 and is a mature language with highly optimized compilers. But it is also a very large language with many features. C++ programs can certainly perform as well as C programs, since C++ includes similar low-level facilities as C. The program in Table 1.1 is only a prototype and not necessarily completely optimized. This shows, however, that C++ programs can have significant overhead if not implemented very carefully. And, although C++ is a high-level language, it is not as easy to use as Python.

“Squeak, a practical Smalltalk written in itself” [24], used the programming language **Slang** to translate a subset of Smalltalk to C. It is a very small language, omitting many features such as blocks or objects. Slang is similar to STELLA in that it focuses on being able to implement all language features efficiently. However, its main purpose is the implementation of the Squeak VM, and does not provide enough features to easily translate whole programs.

Terra [20] is a language which builds on the scripting language Lua [23]. The design-

ers recognize the importance of being able to execute code independently from the host language run-time and implements Lua-style basic OOP. It differs by focusing on multi-stage execution, is a more general framework, and requires source changes to write a Terra program.

Julia [8] is a new programming language specifically designed for scientific computing. It is a high-level and dynamic language, and supports distributed computing. Julia provides interoperability by allowing calls into C, and also into Python. It supports object-oriented programming. Julia is JIT-compiled and provides performance that is reported to be competitive with C programs, however, OOP is not part of their benchmark suite. Julia therefore can offer similar performance, but does not provide the seamless integration into Python. It is unclear how well its object-oriented programming support performs.

1.5 Contributions

This dissertation contributes a new embedded domain-specific language, STELLA, initially aimed at writing scientific simulations within Python, which is 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 STELLA. This separation from the slow run-time will make it easy 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. STELLA implements basic OOP patterns to enable easy code reuse as well as specializations, without compromising run-time performance, by rewriting the OOP patterns in the compilation process.

STELLA achieves all these goals: the Python code requires minimal modification to be compatible, OOP is available, and it runs at speed comparable to the corresponding C version.

Example

In this example, a detailed explanation and execution of a STELLA simulation is performed. For simplicity the example program “SpiderSemiInfinite1D” is used instead of the STELLA implementation of the benchmark program of Table 1.1. Still, some detail is necessary to properly demonstrate the capabilities and different execution stages of the DSL. The example program is presented in a logical “execution” order to clearly illustrate when STELLA is active and how it interacts with Python. The comments explaining the significant features of the program are interspersed. The reader can find the complete listing in Appendix B.3.

The program is a KMC simulation of a molecular walker, nick-named “spider”, which performs a random walk on a 1D surface. It has only one leg, and it changes the state of the surface while it walks.

2.1 Pre-Processing Stage

First the standard Python interpreter is invoked.

Chapter 2. Example

```
1 #!/usr/bin/env python3
2 """
3 Semi-infinite 1D strip with a single spider.
4 """
5
6 import mtpy # cython wrapper around mtwist
7 from math import log, exp
```

The header is regular Python code. Several standard and custom modules are imported.

```
16 class Settings(virtnet_utils.Settings):
17     def setDefaults(self):
18         self.settings = {
19             'seed'      : [int(time.time()), int],
20             'r'         : [0.1, float],
21             'koffp'     : [1.0, float],
22             'K'         : [10, int],
23             'rununtiltime' : [1e3, float],
24             'elapsedTime': [self.elapsedTime, lambda x:x],
25         }
```

This class extends a library class to customize it for the current experiment. It is a small hand-written class to handle command-line arguments in a convenient manner—which is an example of functionality easily implemented in Python, but much more time consuming to add in lower-level languages.

```
92 def test():
93     settings = Settings()
94     settings['seed'] = 1368223681
95     expected = [0, 0, 1, 1, 3, 2, 7, 21, 32, 9]
96     actual = Simulation(settings).run()
97     # convert back to python list for easier comparison
98     assert(list(actual) == expected)
```

Chapter 2. Example

This function is part of the source code, but is only invoked when the tests are run. It verifies that the simulation is correct.

```
100 def main(argv):
101     settings = Settings(argv)
102     print("#", settings)
103     results = Simulation(settings).run()
104     print(results)
105
106 if __name__ == '__main__':
107     main(sys.argv[1:])
```

This is the main method, where the program execution starts. Note that on line 101 the Settings class, which was presented above, is instantiated. On line 103 an instance of Simulation is created (shown below) and on the following line the STELLA program is started by invoking the run() method (see the next section).

```
27 class Simulation(object):
28     EXPSTART = 0.2
29     def __init__(self, params):
30         self.K = params['K']
31         self.rununtiltime = params['rununtiltime']
32         mtpy.seed(params['seed'])
33         self.koffp = params['koffp']
34         self.kcat = params['r']
35
36         self.delta =
37             ↪ (log(self.rununtiltime)-log(self.EXPSTART))/float(self.K-1)
38         self.leg = 0
39         self.substrate = 0
40         self.obs_i = 0
41         self.observations = zeros(shape=self.K, dtype=int)
```

Here the Simulation instance is initialized. This is still executed within the regular

Python interpreter.

2.2 DSL Stage

```
74 @stella.wrap
75 def run(self):
76     self.t = 0.0;
77     self.next_obs_time = self.getNextObsTime();
78
79     while self.obs_i < self.K and self.t < self.rununtiltime:
80         if self.leg < self.substrate:
81             R = self.koffp
82         else:
83             R = self.kcat
84         self.t += mtpy.exp(R)
85
86         while self.isNextObservation():
87             self.makeObservation()
88
89         self.step()
90     return self.observations
```

The `@stella.wrap` decorator invokes the STELLA library, which makes `run()` the *entry method*. The code, shown here, is executed as the compiled STELLA program, even though it appears here as regular Python code.

Flow analysis will discover the following methods (of class `Simulation`) since they are called from `run()`, and then include them in the translation and execution of the STELLA program.

```
42 def makeObservation(self):
43     """Called from run()"""
44     self.observations[self.obs_i] = self.leg
45     self.obs_i += 1
46
47     self.next_obs_time = self.getNextObsTime()
48
49 def getNextObsTime(self):
```

Chapter 2. Example

```
50     """Called from run()"""
51     if self.obs_i == 0:
52         return self.EXPSTART
53     if self.obs_i==self.K-1:
54         return self.rununtiltime;
55
56     return exp(log(self.EXPSTART)+self.delta*self.obs_i);
57
58 def step(self):
59     """Called from run()"""
60     if self.leg == 0:
61         self.leg += 1
62     else:
63         u1 = mtpy.uniform()
64         if u1 < 0.5:
65             self.leg -= 1
66         else:
67             self.leg += 1
68     if self.leg == self.substrate:
69         self.substrate += 1
70
71 def isNextObservation(self):
72     return self.t > self.next_obs_time and self.obs_i < self.K
```

When `run()` returns, the DSL stage ends, and control returns to the Python run-time.

2.3 Post-Processing Stage

Returning back to `main()` from the call in line 103 to running within the Python interpreter:

```
103 results = Simulation(settings).run()
104 print (results)
```

Finally, to keep the example short, the post-processing stage simply prints the results

(line 104) instead of doing analysis, aggregation, or visualization.

2.4 OOP Extension

The following code segment shows how OOP is used to create another experiment, “SpiderSemiInfiniteID-Fpt”, on the same topic, to record different measurements. The complete source code can be found in Appendix B.5.

```
8 class SimulationFpt(Simulation):
9     def __init__(self, params):
10         Simulation.__init__(self, params)
11         self.observations = zeros(shape=self.K, dtype=float)
12
13     def getNextObsTime(self):
14         return 0.0
15
16     def isNextObservation(self):
17         return self.leg > self.obs_i
18
19     def makeObservation(self):
20         self.observations[self.obs_i] = self.t
21         self.obs_i += 1
```

The class `SimulationFpt` extends the class `Simulation` from the previous example.

On line 11 `observations` is initialized with a different data-type `float`, since the previous experiment saved its results as integers.

On lines 13 – 14 `getNextObsTime` is reduced to a minimal form since it will still be called by the simulation logic (which remains the same) but is not actually used in this experiment. The optimizer could then later remove this call at compilation time.

On lines 16 – 17 a new condition for when to make observations is defined.

Chapter 2. Example

On lines 20 – 21 the current simulation time is saved as observation data (the previous experiment saved the distance instead).

The main simulation loop `run()`, which is the entrance method for the STELLA program, is unchanged and hence is not listed here (see Sect. 2.2 for reference). But from the DSL point of view the translation mechanism is exactly the same: Python introspection will simply return the bytecode of the overridden method instead of the original one. Hence the sub-classing does not explicitly need to be taken into account.

Note that this example was designed to be short, readable, and to fit into the structure of the previous experiment.

Design

Chapter 1 explained how the ideas that drove the development of STELLA came to be, but the language must be defined more precisely. STELLA was developed to be an option for implementing high-performance simulations in the Python ecosystem without switching to a lower-level language. There are three goals that drive STELLA's design, in order of precedence:

1. Performance comparable to C
2. Object oriented programming support
3. Ease of use

The last goal is in itself vague, but it is a good summary of the design aspect. Since Python is a high-level language, STELLA can only achieve the first goal if executing individual statements has a constant cost – just like in C. This is naturally at odds with more complex language features. Another view on the goals is then to 1) implement as many features as possible with constant cost, 2) support enough Python idioms such that only minimal modifications to the Python source code for execution in STELLA are required.

Chapter 3. Design

There are several reasons why Python is slow: It is a very dynamic language, where many features can be customized on a per-object basis. Combined with the dynamic typing this results in method look-ups for many operations [39]. Python also is a high-level language with many features. It is easier to write an interpreter for a feature-rich language, but interpreting is inherently slower running a compiled binary. Additionally, a complex language makes it difficult to optimize any kind of implementation.

Performance is of great concern: therefore compiling the language is an easy design choice. As such, 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* what STELLA is meant to be. Since the initial focus is on supporting efficient execution of scientific simulations, a classification as a domain-specific language is more accurate. Nonetheless, the “Python subset” point of view helps to understand how the language looks and feels (see also Chapter 2 discussing an example program). Note that the focus on simulations does not mean that STELLA cannot also be used for other tasks, but it may not offer some convenient features that programmers are used to from other languages.

3.1 Language Characteristics

STELLA is a statically typed, just-in-time compiled language which is embedded in Python. Usually domain-specific languages provide a higher level of abstraction, but STELLA neither raises nor lowers the abstraction. Instead it preserves the Python semantics as much as possible, while restricting the available features to be able to compile the DSL to machine code which could also have been created by a C program.

Staging The program initially runs inside the standard Python interpreter. User code is executed with all language features being available, e.g., to compute constants, dynamically assemble classes and objects, and load initial data. All libraries are usable at

Chapter 3. Design

this point since the DSL is not yet active. After the DSL finishes execution, the program returns to the same Python stage.

This model is not limited to one DSL stage per program, rather STELLA can be invoked as needed. The standard use case, however, will be just one DSL execution since that minimizes the overhead, i.e., analysis and compilation.

Embedding STELLA can roughly be classified as a *shallow embedding* [21]: it uses the native Python syntax, but is executed separately and does not necessarily preserve the structure of the terms. Instead of using decorators or function calls for language features, STELLA operates directly on the internal representation, the Python bytecodes. However, not all features of Python are supported; only those that can be implemented without an inherent run-time overhead are available. For example, basic types are unboxed, there is no `eval()`, and an object's structure cannot be dynamically manipulated.

The use of the Python bytecodes makes the approach very flexible: modules can be used as appropriate, objects can be dynamically assembled, and libraries can be distributed in native Python manner. As long as the called functions only use valid STELLA bytecodes, Python will pre-process it in the expected way, because it is handled by the unmodified Python interpreter.

Restricted features STELLA supports the static parts of Python, but avoids dynamic features because they result in unexpected run-time overhead. It is unrealistic to be able to support a complex and dynamic language and expect it to perform as well as programs written in C.

Semantics The semantics of the language features that are supported remain as close to Python's semantics as possible.

Syntactic sugar Basic Python statements are naturally required to be implemented. But there is also a lot of syntactic sugar which, by definition, is not essential, i.e., the same effect can be achieved through other language constructs [32]. Yet in an effort to be as

Chapter 3. Design

compatible as possible, STELLA implements some of these features because they make the language easier to use: both for new programs, and to make it easier to rewrite existing programs for STELLA.

Predictable performance The supported language features of Python are selected based on a simple criterion: does the language 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. C, in contrast, is lower-level and every individual statement has a cost that is easy to gauge. This is an important feature when writing high-speed implementations. Note that this is naturally no guarantee for fast performance; algorithmic properties are just as important and need to always be considered by the programmer. C libraries can be easily used, since they are often used to provide high-speed implementations.

Memory management A hidden cost in scripting languages is the frequency of memory allocation and deallocation. Since STELLA focuses on high-performance simulations, it is reasonable to preallocate the required memory, and to discourage strongly from allocating memory within the DSL. This means there is some loss of convenience, but will result in better performing code.

Invocation Using the DSL is non-invasive: The existing Python method is wrapped, e.g., using a decorator. Once run, the necessary data is automatically made available to STELLA, the code is generated and executed, and the resulting data is passed back to Python.

Intermediate Representation Introspection is used to transform the Python bytecodes into an intermediate representation for STELLA (IR-S). Some higher-level bytecodes are immediately replaced with a combination of lower-level representations, and others are later rewritten.

Chapter 3. Design

The Python bytecodes are stack-based. The stack locations are propagated and registers assigned to them. This makes it easier to perform the necessary manipulation of the IR-S, and lets STELLA to generate code more easily in later stages.

Control Flow STELLA programs are not allowed to call Python functions for performance reasons. Therefore all called functions must also be STELLA functions and are automatically added to the program. This has two advantages: the programmer only has to modify the program in exactly one place, and all of the performance critical parts are completely executed in STELLA. They are discovered automatically, and an exception is raised if any called function is incompatible.

Type Analysis Using the register-based intermediate format, types are automatically inferred. Since the types of all parameters are known, the remaining types can be deduced automatically by examining the instructions. This is a very simple form of abstract interpretation [15].

OOP The static typing makes it easy to generate code for object-oriented patterns that is procedural. The receiver of bound method calls is passed explicitly as a parameter, very similar to the way in which Python makes the programmer declare the first parameter of a method. Attribute access then turns into accessing a structure's field. This translation mechanism seems basic, but is actually quite powerful since it leverages the available type information and reuses the staging environment for features like inheritance and method resolution order.

Code Generation STELLA uses LLVM [34], a proven compiler toolkit, to make sure that high-quality machine code is generated. Given all the preparatory work, i.e., instructions with typed registers, it is easy to generate the LLVM intermediate representation (LLVM IR). The generated code is very similar to the LLVM IR generated by *clang*, so the default LLVM optimizations apply well.

Focus on Single-Thread Performance Stochastic simulations are in fact *embar-*

rassingly parallel: each sample that needs to be collected does not communicate with other samples, because they must be statistically independent. Since no effort is required to parallelize the simulation (aside from data management, but that is a different research area [36]), STELLA is focused on the most beneficial factor for this problem domain: excellent single-thread performance.

3.2 Embedding

STELLA is embedded in Python. Most related approaches favor easy integration, and make it transparent to run any Python function or feature. However, this means the language is not focused on self-contained execution, and also causes the programmer to unintentionally invoke the Python run-time which inadvertently slows down the program execution. STELLA deliberately chooses the exact opposite. It guarantees that during the DSL execution there is *no interaction* with the Python run-time. This is restrictive: Python cannot contribute features that STELLA has not natively implemented, and any Python modules implemented in C cannot be directly used since they are written for the CPython API. This is similar to run-times like *PyPy* [41]. The advantage, however, is that STELLA ensures that the complete core of the program is executed by the DSL and no unexpected slow-downs will occur.

On the other hand, STELLA's types are compatible with C. Therefore it is easy to integrate direct calls to C functions. Since C libraries are compiled, the programmer is expected to know the run-time behavior of the used libraries, and as such there is no unexpected overhead.

3.3 Syntax and Semantics

As STELLA is an embedded DSL, there is no special syntax to describe: it is the Python syntax. Unlike DSLs with a higher abstraction level, there is no special API either. It is the Python statements themselves that are interpreted for the DSL. STELLA even lets the Python interpreter create and process the AST. It then utilizes the lower level representation, the Python bytecodes. This approach has several advantages:

- A bytecode has a very specific function. This granularity makes it easy to implement matching functionality.
- One single bytecode can be produced while compiling a variety of source level statements. Therefore there is a synergy effect once it is implemented and more of the Python language can be supported.
- During the creation of the bytecodes the Python interpreter encodes parts of its knowledge of the language. This supports the goal to retain as much Python behavior as possible—if CPython performs the function, then it is inherently correct.

There is also one downside: the reliance on the CPython bytecodes makes it more difficult to integrate STELLA with alternative language run-times such as PyPy.

Python is a language without a formal specification; it is defined by its reference implementation, CPython. STELLA's goal is to stay as close to the Python semantics as possible. Therefore STELLA is also a language that is defined by its implementation. Even though there is no formal specification, the whole language is described here and the user can expect that if a construct is supported and is not mentioned in Section 3.10.1, it will behave as if it was run in CPython. This is confirmed by the tests described in Section 4.8.

The fact that STELLA is defined by the implemented bytecodes has the unfortunate side effect that it is not easy to describe the supported features at the Python source-level.

The implementation will be discussed in Chapter 4, but significant features which the prototype does not support are: lambda expressions, slicing of arrays, comprehensions of lists, sets, and dictionaries; generators, yield, del, nonlocal, assert, import, and nested functions. Note that all these features are available in the pre-processing stage. For example, the programmer can use `import` then, and the resulting binding is subsequently available within the STELLA part of the program.

3.4 Syntactic Sugar

STELLA implements some Python features which are syntactic sugar: they are not essential, but they are important for both Python compatibility as well as readability. The latter is an important goal of the Python language: source code is written once, but often read many times. Therefore clarity of the source code is important to maintain the overall spirit of Python.

```
1 /* C */
2 #define get(s,x,y, sz) ((s)[(x)*sz+(y)])
3 d2Tdx = (fl*get(M,y,left, ySz) + fr*get(M,y,right,ySz) -
  ↪ 2.0*get(M,y,x,ySz)) / gridsz2;
```

```
1 # Python
2 d2Tdx = (fl*M[y,left] + fr*M[y,right] - 2.0*M[y,x]) / gridsz2
```

Listing 1: Comparison of 2D array indexing in C, Python

Consider the subscript operator in Listing 1 as an example. NumPy arrays use it to implement 2D array access in a very transparent manner. The same code is supported in STELLA. This is not possible in a low-level language such as C, where the same functionality can obviously be implemented but it must be used without the syntactic sugar: Here the function call (or macro) `get` is much less readable.

Chapter 3. Design

```
1 def complicated(array, option1=False, option2=True, doValidate=False,
  ↪ transpose=True):
2     # ...
3
4 # calling the function while specifying only some parameters
5 complicated(a, option2=False, doValidate=True)
```

```
1 #define option1_default 0
2 #define option2_default 1
3 void complicatedNoOptions(int **array) {
4     complicated(array, option1_default, option2_default, 0, 1);
5 }
6 void complicatedSomeOptions(int **array, int option1, int doValidate) {
7     complicated(array, option1, option2_default, doValidate, 1);
8 }
9 void complicated(int **array, int option1, int option2, int doValidate,
  ↪ int transpose) { /* ... */ }
```

Listing 2: Default values for function arguments compared in Python (above), C (below)

Another very useful Python feature is function default arguments. See Listing 2 for an example. While the same functionality can be implemented by using a variety of helper functions to supply the default values, the syntactic sugar combines everything relevant in one central location, the function definition. The body, where the default value will be consumed, follows immediately afterwards. A similar purpose is fulfilled by keyword arguments: it is a simple mechanism that allows flexible function calls.

Here it can be seen how much more verbose the C variant is, and it does not even provide all possible combinations of parameters: not only does it take much more effort to write, but also it is cumbersome to read.

Python offers much more functionality, e.g., it can use a dictionary to fill in keyword options during a function call. While this is also convenient, it is a dynamic feature that is therefore not implemented in STELLA.

3.5 Typing

The Python language uses dynamic typing and encourages a style that is called [duck typing](#): instead of explicit type checking, programs are encouraged to assume that assumptions hold, and catch exceptions if they don't. While STELLA does not have complete exception support, it does work well with the polymorphic approach: duck typing results in Python code that does not usually check types, so when STELLA is invoked it will evaluate the code with respect to the types used in the present invocation. If the given object implements all methods that the code calls, then it is compatible and the code will compile.

3.5.1 Static Typing

Since STELLA avoids dynamic features there must not be any dynamic type checks: everything, i.e., the presence of attributes or methods, and their types, is checked statically at compile time. That means every variable, parameter, and return value must be attributed exactly one type, or an exception will be raised. Note that the duck typing approach makes it possible to use the same method but with differently typed parameters in several contexts, e.g., different invocations of STELLA. This is the same flexibility that Python provides.

Static typing is desirable as an error checking mechanism, too. Since it is seamless to prototype in Python, unrestricted by any type rules, the programmer does not lose any flexibility. Subsequently the finished simulation can run in STELLA without further type annotations, and the programmer can be certain that the computationally expensive simulation core will execute without any type errors at run-time.

3.5.2 Types

STELLA uses machine types, exactly as C, to gain the most efficient run-time behavior. This should be familiar to any programmer with some low-level experience and also gives STELLA ABI compatibility with C. The typing rules are similar to Python.

Scalars Integers, booleans and floats are implemented as unboxed machine types for efficiency reasons.

Tuples Tuples are a prevalent data structure in Python: an immutable list containing heterogeneous data. They are fully supported in STELLA as they are equivalent to (anonymous) structures.

Arrays The array is an important data type in scientific applications. Arrays of scalar values are required to be NumPy arrays. They are a popular choice in existing scientific programs, which means no type conversion is required to pass them to STELLA. NumPy arrays do not support complex types such as objects¹. Instead, STELLA uses regular Python lists for elements with complex types. For both array types, no resize operations are implemented within the DSL. See Section 3.6 for more details.

Objects Python objects are represented as C-like structures. In Python all of the objects data are stored as attributes–member variables as well as methods. In STELLA only the data is present in the structure as method resolution is performed at compile time. See Section 3.8 for more details.

3.6 Arrays

Arrays typically hold the majority of the data in scientific programs, therefore it is very important that this data type can be transparently used in STELLA without any structural

¹NumPy arrays can be used for complex data structures, but not for Python objects.

conversions. NumPy arrays fulfill this requirement, and the implementation is discussed in Section 4.7.2.

Dependent typing is used for arrays [54], therefore the type also encodes the length of the array. The array length can be easily retrieved because arrays are constructed within Python and their length can simply be queried during the analysis. Recall that no array resizing is allowed in STELLA since this would constitute a dynamic language feature, and therefore the array length is a constant.

STELLA does *not* automatically add bounds checks to array indexing because this violates the design principle of constant cost. The Java language did initially have array bounds check on every single array subscript operation which led to disastrous performance for numerical code. The HotSpot Java VM is also an example for the behavior that STELLA wants to avoid—It implements array bounds check elimination, but since the optimization has preconditions that are hidden from the programmer, there were unexpected situations when programs suddenly performed abysmally. However, omitting bounds checks can lead to crashes at run-time. But errors of this kind can be eliminated during prototyping within Python. When possible, i.e. when the index is constant, STELLA will check the bounds. Consider the following example code:

```
1 def subscript1(a, i):
2     return a[i]
3
4 def subscript2(a):
5     return a[42]
```

In the first function, `subscript1`, the index `i` cannot be evaluated at compile time and therefore out of bounds access is possible. In the second function, `subscript2`, the constant index is immediately compared against `a`'s length and the program will not compile if it is out of bounds.

Consider the common Python idiom to iterate over the complete array:

```
1 for x in a:
2     foo(x)
```

Here the array length is used for the iteration, and naturally no out of bounds access is possible.

Lastly the array length is used when the `len()` intrinsic is called, which means that Python code can remain idiomatic and manual optimizations such as the following are not necessary:

```
1 # original
2 for i in range(len(a)):
3     foo(a[i]*len(a))
4
5 # ill-conceived manual optimization
6 manually_factored_len = len(a)
7 for i in range(manually_factored_len):
8     foo(a[i]*manually_factored_len)
```

In fact, “calling” `len` instead using a “pre-computed” variable is more efficient before the LLVM optimization passes are applied: The former is a constant from the outset, while for the latter the optimizer has to discover that the variable is not modified and can in fact be replaced by a constant.

3.7 Typing Rules

Python, similar to C, automatically converts integers into floating point numbers when combined in a computation. Therefore types are organized in a lattice: bottom is no type.

Only integer types are automatically promoted to float. Every other type is promoted to top (untypable). Since the type rules form a lattice, it is ensured that the type analysis is bounded: every function can be analyzed at most 3 times, the height of the lattice.

Arrays, as well as lists, are compatible if their member type matches (contents are of homogeneous type), and are of equal length. Note that this is simpler than general dependent typing, and is immediately decidable, because the length is always of constant size.

The current typing rules of objects are extremely simple: The class must be the same for two objects to be of the same type. Subclass relationships are not currently taken into account, see Section 3.8 for more details.

3.8 Object-Oriented Programming

Python lacks 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, but Section 1.3 explained why OOP support is important in general.

When the DSL is invoked, introspection is used 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 a dictionary, scalar values in Python are boxed). Therefore the attribute contents need to be transferred from Python to the STELLA representation. At the end of the DSL execution, the resulting values are copied back into the corresponding Python object. Since this is constant overhead, it will be amortized by the many attribute accesses during the execution of the DSL program.

Chapter 3. Design

Note that the object introspection leverages Python's flexible abilities to construct objects, including multiple inheritance (mix-ins), and staying true to Python's method resolution order. All of these features behave exactly as in Python because CPython handles them. Within STELLA the behavior is much simpler, but this is often not a limiting factor, in part because of the powerful construction mechanisms.

All access of object attributes 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, or virtual method call look-ups.

The receiver of the method calls is conveniently a named parameter in Python, typically the name `self`. STELLA then turns method calls into a regular function call which passes the structure representing the object as the first parameter.

Typing rules are currently very simple and make any two distinct classes incompatible, i.e., their class hierarchy is not taken into account. This could certainly be refined, but a model identical to Python's cannot be implemented without adding hidden overhead due to Python's support for multiple inheritance.

First consider the single inheritance tree in Figure 3.1a. This hierarchy can be represented with the structure shown in Figure 3.1b. The memory layout presented there allows low-level code accessing structure `classA` member `x` to also be compatible with the layout of `classB` or `classC`, since the member `x` always has offset 0 in memory. This is just as it should be, since B and C are also A's.

A memory layout with similar accessibility is not possible for multiple inheritance. Consider the class hierarchy in Figure 3.2a, and the corresponding low-level representation in Figure 3.2b. Here D's `x` has an offset of 0, but so does E's `y`. There is no memory layout of F that is compatible with both functions that accept D instances and other functions that accept E instances. It should be possible to pass an instance of F to a function designed for

```

1 class A:
2     x = 1
3
4 class B(A):
5     y = 2
6
7 class C(A):
8     z = 3

```

(a) Class hierarchy in Python

```

1 struct classA {
2     int x;
3 };
4 struct classB {
5     struct classA base;
6     int y;
7 };
8 struct classC {
9     struct classA base;
10    int z;
11 };

```

(b) Corresponding low-level accessible data layout

Figure 3.1: Single inheritance class tree in Python, and the corresponding low-level memory layout. For clarity the structures are presented in C.

E, since F is a subclass of E. But the memory layout would mean that an access of offset 0 in instances of F mistakenly return x instead of y. Since STELLA uses static typing, and avoids dynamic type checks, this cannot be supported.

```

1 class D:
2     x = 1
3
4 class E:
5     y = 2
6
7 class F(D,E):
8     z = 3

```

(a) Class hierarchy in Python

```

1 struct classD {
2     int x;
3 };
4 struct classE {
5     int y;
6 };
7 struct classF {
8     struct classA baseD;
9     struct classA baseE;
10    int z;
11 };

```

(b) One possible low-level data layout

Figure 3.2: Multiple inheritance class graph in Python, and one possible low-level memory layout. For clarity the structures are presented in C.

There is, however, a more restrictive form of multiple inheritance called *mix-in*: a class which does not define any data members, but only provides functionality in terms of methods. STELLA could support this restrictive form since then no memory layout issues arise. The STELLA compiler would have to verify that in the presence of multiple inheritance at most one base class contains non-function attributes.

Therefore it is a trade-off: if STELLA supported only *mix-in*-style multiple inheritance, then subtyping relations could be modeled. On the other hand, the current prototype does not implement subtypes. It does allow all valid Python class constructs, including those by using full multiple inheritance, to be used in STELLA. The memory layout is not compatible, but that is acceptable because subtypes are not compatible. Going back to Figure 3.2, if only instances of class F are present in the program, this limitation is not restrictive at all.

3.9 Variable Scope

Scoping rules are simple: any variable is either local, or global. This decision is made by Python itself since different bytecodes are used for each (recall that the bytecodes are created by the Python run-time and not by STELLA). Local variables are visible in the entire function. Globals, by definition, are visible everywhere. These rules are only a subset of Python's scoping rules, but they are compatible.

Python has additional rules for nested blocks and module-level visibility. Neither is supported in STELLA: nested blocks, e.g., nested function definitions, require creating closures on-the-fly, which is a dynamic feature and therefore not part of the language. STELLA can access functions in other modules, but it does not currently support a module scope and cannot access variables defined in other modules. A module scope could be supported, but has not yet been implemented.

3.10 Optimization Passes

STELLA ensures fast performance at run-time through its design and feature selection. It does not itself implement an optimization pass (with the exception of rewriting for loops discussed in Section 4.4.3). Instead it relies on the infrastructure that LLVM provides. If optimization is requested when STELLA is invoked, the corresponding LLVM pass will be run before execution.

While the optimization passes that LLVM currently implements may not be optimal for dynamic languages such as Python [29], they are an exact fit for STELLA, a static language without dynamic features. STELLA generates LLVM IR that is very similar to the LLVM IR generated by *clang*. To achieve this goal, STELLA required some minimal semantic changes compared to Python, which are explained in the next Section.

3.10.1 Deviations from Python Semantics

In order to preserve the goal of C-like efficiency, deviations from the Python semantics were required. These differences will not have a major impact on most programmes. The exceptions are:

1. In STELLA the basic types are machine types, while in Python they are boxed. Thus STELLA is implemented in same way as C is implemented, and hence the semantics of the basic arithmetic types will be the same as in C. This causes differences in, e.g., overflow or underflow of a variable because Python then automatically switches to arbitrary precision math. Arbitrary precision math is complex and therefore much slower. This behavior can cause performance issues even in programs that run at acceptable performance in CPython. Therefore transparently switching to arbitrary precision math is unacceptable in STELLA.

2. Modulo always has the sign of the dividend, unlike Python where it has the sign of the divisor.
3. 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 its sign.
4. There is no array bounds check. As discussed in Section 3.5.2, this would add unreasonable overhead. Even with bounds checks and a corresponding analysis to lift the checks, there would always be situations where the analysis fails and thus unexpected slowdowns could not be avoided.
5. STELLA uses static typing, as discussed in Section 3.5.1.

3.11 Interfacing with C

One of the design goals of STELLA is that if a program compiles, there is a certain assurance that it will perform well (see Section 1.3). Among other design decisions this is ensured by avoiding all interaction with the Python run-time. C libraries, however, are often used to provide high-speed implementations, and since they also use machine types, it is a good fit for STELLA and can easily provide access to them.

Recall that all STELLA programs are also valid Python programs. This compatibility is important because it is used for testing and debugging (see Section 4.8). Therefore interfacing with C must maintain the Python compatibility, which can be achieved by generating a Python module which is a light-weight wrapper around the C library: it makes each C function available with an identical name, and accepts the same arguments. Such a wrapper can be generated semi-automatically, see Section 4.7 for more details.

Chapter 3. Design

At compile time, all module references are checked to see if they are backed by a C library. If so, then instead of trying to disassemble the functions accessible in Python, STELLA will create a special object with a reference to the module and function, and the correct type signature. When the program is run in Python, the wrapper makes the C function available to the Python run-time. When the program is run in STELLA, then the wrapper is skipped, and the backing C function is directly called without additional overhead.

Implementation

This chapter describes the implementation of the design that was presented in Chapter 3. This is significant, because, just like Python, STELLA is defined by its implementation.

STELLA is a Python library written in Python, as opposed to implementations in other languages, e.g. C++, which interface with CPython through its API. First, libraries used will be explained along with the function they fulfill. STELLA uses its own intermediate representation, which are the main building blocks of the implementation. The program analysis and its phases successively prepare the intermediate representation for the code generation. Afterwards the translated program is executed. Only some minimal work is necessary before control is returned back to the user program.

4.1 Dependencies

It is rare, if not impossible, to implement complex software today without the use of third party software. Here the libraries are introduced, and their use is briefly discussed.

LLVM Compiler construction toolkit based on the LLVM intermediate representation (LLVM IR), a typed assembly language. It offers both a regular ahead-of-time

Chapter 4. Implementation

compiler, as well as a JIT compiler. It includes a set of optimization passes, which can be used by both compiler implementations. See Section 4.2 for more details.

llvmlite Python bindings to LLVM. It provides a lightweight wrapper around the C++ LLVM interface and creates a text version of the LLVM IR for further processing. It operates in-memory and can call the LLVM JIT compiler.

NumPy Well-known package for scientific computing with support for high-performance arrays and matrices. Its data types are accessible from Python, but are backed by a memory structure that can be directly accessed from low-level languages. It is technically only required if the user program uses arrays. No effort has been made to turn this into an optional dependency, because array use is expected in virtually all scientific programs.

pytest An excellent, easy to use, testing tool for Python. Only required for running the tests or the benchmarks.

ctypes A module out of the standard library for Python. The main features used are the construction of C-style signatures, allocation of memory in C-compatible layouts, and transparent access from Python to this memory.

4.2 LLVM Primer

The compiler construction toolchain LLVM [34] is a general compiler framework with an initial focus on C and C++. LLVM provides an intermediate language, the LLVM IR, which has both a textual representation and a binary format. It is a typed but low-level language. LLVM comes with optimization passes and a back-end code generator for several platforms. It is a very popular choice recently due to its fast compilation times and high quality machine code generators.

Chapter 4. Implementation

The decision to use LLVM was made very early on, and therefore the whole implementation is tailored towards processing the Python bytecodes with the goal of generating LLVM IR. This clear objective shaped several stages of the implementation, therefore I briefly discuss the relevant aspects of LLVM here. For further information I refer to LLVM’s [documentation](#), in particular the [tutorial](#)¹.

The LLVM IR is a low-level representation, similar to assembly language, but it is completely typed. It is not only used as the input to the LLVM ecosystem but also as its intermediate format. The simplicity of each instruction, the availability of type information, and static single assignment (SSA) enable a multitude of optimizations to be performed. LLVM IR uses an unlimited register model, since automatic register assignments today generally is more effective than manual register management.

Each function is composed of a series of code blocks. A code block is a sequence of instructions with exactly one entry and one exit point—the beginning and the end of the block, respectively. All control flow between blocks must be explicit.

Single static assignment [16] means that each register can only be written to exactly once. Reads are not affected. This makes it necessary to introduce so called “Phi nodes” to merge values based on control flow to the current code block, since it is not possible to rewrite a register. SSA simplifies the analysis and rewriting of the IR that optimization passes perform.

LLVM has been immensely successful and is used by various other high-profile projects such as nVidia’s CUDA [22], various Apple software products², OpenCL [25], and GHC [47].

¹The predecessor to the *llvmlite* library, *llvmpy*, included a [Python version](#) of the same tutorial.

²Apple itself does not advertise that it uses LLVM, however, this fact is commonly known, and documented on the [llvm website](#).

4.3 Intermediate Representation

STELLA uses its own intermediate representation. The term is potentially confusing because LLVM also has its own intermediate representation, which is the output of the code generation phase. Therefore in the remainder of this document the STELLA intermediate representation is referred to as *IR-S* (Intermediate representation – STELLA) while the LLVM intermediate representation is referred to as *LLVM IR*.

The IR-S is composed of objects that inherit from the `stell.ir.IR` class, each of which can be categorized as either a Python bytecode, a LLVM helper, or a rewrite helper.

Python Bytecodes Each Python bytecode is represented in the IR-S as an individual object instantiated from a class of the identical name. This allows automatic mapping of bytecodes to IR-S classes. Inheritance is used to implement common mechanics in only once in a parent class, e.g., for jumps which differ only in the condition (*jump if false* versus *jump if true*). The bytecodes are discussed in more detail in Section 4.10.

LLVM helper The LLVM IR is a model with unlimited registers but each register has to follow the SSA format. Therefore *phi nodes*, which represent SSA's ϕ -functions [16], become necessary to instruct LLVM which value should be chosen based on the last branch. The `PhiNode` class creates an IR-S object which acts as a jump destination and records the origins.

Rewrite Helper Python for loops are rewritten in lower-level terms to guarantee optimal performance (see Section 4.4.3). Therefore the `ForLoop` class implements a for loop and is used to record its parameters.

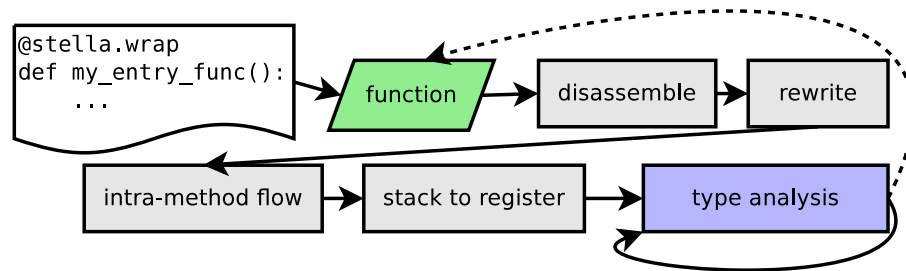


Figure 4.1: A high-level overview of the analysis steps for a function. It starts at the *entry function* that the programmer marked. Other called functions are automatically discovered during the type analysis, and added to the queue to get analyzed as well.

4.4 Analysis

The analysis starts at the *entry function* (see Figure 4.1). The programmer needs to decorate only this one function that starts the simulation core. The remainder of the analysis, including finding the call graph, is performed automatically. This is convenient, but it also serves a more important purpose: the programmer can easily move code into and out of the DSL section of the program, and it also allows the same function to be called from both Python and STELLA—a feature which makes it easy to write tests validating the semantics.

The first step of the analysis is to use Python’s introspection facilities 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.

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, e.g., for-loops, as a sequence of lower-level bytecodes. 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. This representation is more efficient to process, and also makes it easier

Chapter 4. Implementation

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 [39]. Therefore the values on the stack are transformed into a register representation, which makes it easier to perform the manipulations and to generate the LLVM IR, which is also based on registers. Note that at this point only the flow of information is represented, but not yet the exact values which the bytecodes operate on. That is because type information is required in some situations, and it only becomes available in the next stage.

The last stage 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. Function calls interrupt this analysis if the return type is not yet known, since further type deductions depend on it.

Note that STELLA does not yet implement a dedicated optimization phase, since even without one the goal of C-like performance was achieved (see Section 5.4). It does heavily rely on the optimization passes built into LLVM.

4.4.1 Function Objects

The fundamental unit of the analysis is the function, and a single `Function` object will be created for each Python function (or method) discovered during the analysis phase. All analysis phases are implemented as methods that mutate the state of this object.

There is a global queue of functions that need to be analyzed (the *global analysis queue*). This queue initially only contains the entry function. During the analysis, newly discovered functions (see Section 4.4.7) are added to the queue, as well as functions that need to be analyzed again at a later time (e.g., due to incomplete type information). The analysis continues until this queue is empty.

Chapter 4. Implementation

The analysis of an individual function is separated into distinct phases. The first time STELLA encounters a function, all phases will be run. On subsequent encounters, only the type analysis is repeated—since this is the only part that is subject to change.

4.4.2 Disassembly

The disassembly code is based on Python’s own internal disassembly mechanism³, since it is an implementation detail that is not heavily documented.

In LLVM, parameters are passed in registers. This is inconvenient, because registers have to follow the SSA format, while Python parameters are treated as variables, and therefore are mutable. To treat parameters equal to regular variables, they are renamed, and then a stack location is created that bears the original parameter name. This stack location can then be treated as any other local variable. This may seem like an inefficient choice, but LLVM has a very good built-in optimization pass to remove unnecessary stack locations.

The disassembly function then creates the IR-S objects (as introduced in Section 4.3), and adds information which is directly contained in the Python bytecodes: arguments, variable names, and jump target information. Debugging information, i.e., the line of the source code from which the bytecode originated, is also transferred to the IR-S objects. If a bytecode is unsupported, an exception will be raised, the analysis is aborted, and control returns to Python. The exception is discussed in more detail in Section 4.9.

³The standard module `dis` is implemented in native Python code and served as a basis for the disassembly phase.

4.4.3 Rewrite

Most Python features are translatable directly from the bytecode, because each bytecode has a very fine-grained functionality. But there are some situations which are more complex, and are easier to translate by making some structural changes first. This rewriting is done directly after the disassembly, and therefore can be considered a peephole optimization since it only takes local information into account. Currently this is only used for for-loops.

Python handles for-loops by creating an iterator and using it to traverse the given object. Iterators can either be objects, or use `yield`, which essentially creates a continuation. While this is a very flexible approach, 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. Generally STELLA tries to take full advantage of the LLVM optimizer (see Section 3.10), but since the general for-loop semantics are complex, removing the overhead through an optimization pass would be difficult to perform while avoiding all corner cases. The Python bytecodes explicitly record for-loops, since they are represented with a dedicated bytecode, but once the for-loop semantics are translated to the lower-level LLVM IR, this information is lost. This is the only situation where STELLA directly applies optimizations. Here it is necessary because simulations usually use one or more for-loops at the very core of the simulation, and therefore an efficient implementation is essential to be able to attain the desired speed.

4.4.4 Intra-procedural Control Flow

The information about the control flow is necessarily present in the Python bytecodes, since this is all the information that the Python interpreter has to correctly run the program.

Chapter 4. Implementation

It was added verbatim to the IR-S objects during disassembly. What follows is a translation from indirect, and implicit, information into an explicit representation:

1. If a verbatim index for a destination bytecode is present, then the target bytecode is looked up, and added to the IR-S object as a direct pointer.
2. Many control flow diverting bytecodes, e.g., `JUMP_IF_TRUE_OR_POP`, jump to a target, or fall through to the next bytecode. Since LLVM requires all control flow to be explicit, the implicit fall-through is now converted into the same explicit format the indirect jump target is translated to, a direct pointer to the next bytecode.
3. LLVM only allows control flow to occur at the end of a code block, and the control diversion must be to the beginning of another code block. Therefore if a bytecode is the target of a jump, it must be the start of a code block. Consequently, the previous bytecode is the end of the code block, and since all control flow in LLVM is explicit, a simple jump must be added. This is straightforward to decide given the explicit control flow information.
4. If there is more than one incoming jump at the current bytecode, then a phi node must be added. The current bytecode operates on the value on top of the stack, but this value could now come from more than one control flow path. Since registers can only be written to once, as demanded by the SSA, a phi node is required to select the correct register based on the control-flow information at run-time. Phi nodes are a part of the LLVM IR, and do not require further processing on STELLA's side.

4.4.5 Stack Unwinding

The LLVM IR operates on an unlimited number of registers, and requires instructions to be in the single static assignment format (SSA). This makes it in principle very easy to transform the stack values to registers by assigning a new register to every stack location.

Chapter 4. Implementation

In reality it is slightly more complicated because some Python bytecodes places results on the stack that STELLA needs to handle at compile time. Unlike most results, these are not LLVM registers. Therefore the transformation of stack values is then split into two parts. The stack operations are replayed but instead of immediately creating registers, the analysis keeps track of which IR-S object put the value into a stack location. The creation of registers is deferred until the type analysis phase, because at that time enough information is available to decide whether to create a register, or to create a different STELLA representation, e.g., a reference to a method or an intrinsic.

The stack evaluation operates on a queue of bytecodes. When there is no control flow change, the next bytecode is added to the queue. If there are control flow deviations, then the bytecode returns all possible paths, together with the stack that each path needs to be evaluated on, i.e., the stacks are duplicated for each path. The paths are then added to the queue unless both the stack is empty, and the bytecode has already been evaluated before. Such paths are not relevant for unwinding the stack.

In some cases Python generates code that is unreachable. The last statement of any function will be a “return None”, even when there is no control flow path to this bytecode. This is ignored, because None is only a valid return value for objects—it would violate the static typing for other return types if evaluated in later stages.

4.4.6 Type Analysis

Every IR-S object contains its typing rules and evaluates them based on the types of its operands. This is a very simple form of abstract interpretation, since no operations are actually executed. The typing rules match those of Python, with the exceptions mentioned in Section 3.10.1. Since the stack unwinding has already determined the operands for each bytecode, the type analysis can now simply iterate sequentially over the list of bytecodes.

Similar to the stack unwinding, unreachable bytecodes are skipped.

Chapter 4. Implementation

For an individual bytecode the procedure is as follows. If it has operands, it will retrieve the result object for each one from the bytecodes that the stack unwinding determined. Then the current bytecode can evaluate its rules, and create its result object. The result is either an IR-S object representing an LLVM register, or an different class of IR-S object.

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).

The type analysis is immediately repeated when types are widened, e.g., during type coercion when arithmetic operations have different operand types. When a function is called where the return type is not yet known, the analysis is interrupted. The current function is then added to the global analysis queue behind the called function (see Section 4.4.7), so that when it is next evaluated, the called function return type will be known.

4.4.7 Inter-procedural Control Flow Discovery

This is not a distinct phase, but rather is performed automatically as part of the type analysis (see Section 4.4.6). At any time that a function is referenced, usually during a function call, the state of the STELLA Function object is checked. If it is not completely analyzed yet, or the return type is not yet know, then this function is added to the global analysis queue. This ensures that all referenced functions are analyzed by STELLA.

This relatively simple control flow analysis is completely sufficient since there are no dynamic ways to call a function, such as “eval(“func(‘dynamically assembled’)”)” in regular Python.

References to external functions, such as part of a C module, do not need to be an-

alyzed: their types can be directly looked up, and that is sufficient information for the STELLA type analysis, as well as for the actual code generation (see Section 3.11).

4.5 Intrinsic

Intrinsic are functions in Python which are not implemented as regular function calls in STELLA. There are several kinds of intrinsic:

1. Python functions which can be computed at compile time, e.g., “len()” (see Section 3.6). These calls turn into constant values.
2. Functions mapped to LLVM intrinsic, e.g., *math.log*. They still require a function call, but LLVM requires a special declaration for these functions.
3. Casts in Python are function calls. In STELLA they are implemented as native LLVM instructions.
4. Special constructs. This is currently only a placeholder for an Exception object. Since new objects in Python are created with a function call, the Exception in STELLA is implemented as an intrinsic function. Currently only the special combination of “creating an exception, and then immediately raising it” is valid in STELLA. This support improves the Python compatibility, even though it is only a no-op due to missing features in the *llvmlite* interface to LLVM.

4.6 Types

The supported types are implemented as follows:

Chapter 4. Implementation

Scalar Boolean are represented as 8-bit integers since that is the minimum size of memory operations on x86 that is well defined. Integer and floating point numbers are presently defined to be 64 bits wide. This choice seems appropriate for scientific applications where accuracy is a concern.

Array An array can only contain scalar types, and must be preallocated as a NumPy array during the pre-processing phase. Therefore it is created by a ctypes cast to a pointer of the corresponding member type. Subscripts use LLVM's GEP instruction to calculate the correct memory offset.

List A list must contain objects of uniform type. The list is created during the data transfer phase, while still in Python, and is therefore memory managed by the Python run-time. The machine type is therefore "pointer to a pointer of an object structure", and subscripts also use the GEP instruction.

Tuple A tuple is implemented using an anonymous structure (termed `literal structure` by LLVM). It is supported by STELLA, and is allocated on the stack at run-time. Therefore a tuple is automatically released when the function returns without additional memory management procedures (and their corresponding overhead). Subscripts use the LLVM instructions `insert_value` and `extract_value`.

4.7 Code Generation and Execution

The last phase for STELLA is to generate the LLVM IR. The different analysis phases reorganized the code, cross-referenced the bytcodes, placed the operands in registers, and typed them. Everything works towards this goal, and therefore it is now straight forward to generate LLVM IR.

The LLVM JIT compiler, at the time of writing, does not support executing methods with arguments. This means a stub method must be generated with the actual parameters

Chapter 4. Implementation

compiled in as constants. The stub method then calls the entry function and passes the parameters.

The next steps are executed for each function that is part of the current STELLA program:

C modules External C modules are handled completely separate from STELLA functions, and will be discussed separately in Section 4.7.1.

LLVM blocks Given the list of IR-S objects, the first goal is to create blocks. The requirements for blocks are described in Section 4.2, and are easy to generate based on the control flow information. A linear iteration over the bytecodes is sufficient, and a new block is created and stored in the current bytecode whenever there is an incoming jump.

Emitting LLVM IR With the blocks in place, generating the LLVM IR again requires a linear iteration over the list of bytecodes. Each bytecode now executes its own generation code, which directly creates the LLVM IR by calling into the *llvmlite* library.

Optimization The LLVM optimizer is invoked with the programmer-specified optimization level.

Compilation and Execution Once LLVM's IR is completely assembled, the *MCJIT* component is used to just-in-time compile and execute the program. This is handled by *llvmlite* and LLVM.

4.7.1 C libraries

There is no need to generate code, or translate external C modules in other ways. As described in Section 3.11, some preparatory work is required ahead of time. The C library must be made accessible in Python, which is easily done using Cython [7]. The wrapper needs to provide type annotations, so that they can be used for the static typing. STELLA

Chapter 4. Implementation

designates a special function in the module to return this information in a dictionary. Note that this function is most easily provided by the Cython wrapper and hence does not have to be present in the original C 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. Currently this is done by hand, since I could not find an API to extract this information from Cython. In the future, an alternative for automatic interface generation could be provided by the [cffi](#) library, which directly parses C source code.

4.7.2 Data Transfer

Section 3.5.2 already laid out the representation of the different data types. Scalar values are re-created as LLVM constants and thus do not require data transfer. NumPy arrays conveniently offer access to their intermediate representation, which is compatible with C-style array layouts, and can be directly used in STELLA as a pointer to the same memory which is being used in Python. No data transfer is necessary.

Objects, on the other hand, have a memory layout which is geared towards Python. This layout does not allow direct access from lower-level code, e.g., due to the boxing of scalar values. Therefore 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. If the added memory size became an issue, another representation would need to be added to STELLA, e.g., an implementation of objects which offers convenient low-level access. However, this would come at a cost of sizable implementation work and increased execution time inside of Python.

Chapter 4. Implementation

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 as described above. Then a list of pointers to these shadow objects is created.

The data transfer implementation can be divided into three phases:

Discovery The objects must be accessible from within STELLA, or no transfer would be necessary in the first place. First we iterate over all globals discovered during the analysis, and transfer their contents. Then recall that the receiver of a method invocation is passed as the first argument. Thus all remaining data that accessible to the STELLA program is necessarily reachable through one of the entry function parameters.

Pre-Execution A shadow structure is created. For objects, STELLA selects all attributes of the Python original that contains data, i.e., is not a method. With the data attributes, and their corresponding types, a C-compatible data type is allocated with *ctypes*, which can then be accessed transparently from Python. Then the content is copied to the shadow structure. Scalar data can be directly assigned to, and any other types are processed in this same manner, and then assigned to the shadow structure's member. For other types the shadow structure is created as mentioned above.

Post-Execution The same discovery process iterates over all parameters to transfer the data back from the shadow structure into the original Python data structures. Because of the *ctypes* interface, the transfer is now simply the assignment in the reverse direction.

4.8 Verification

One goal of my project is that each DSL code fragment should have the same semantics as its literal equivalent in Python (with the exceptions defined in Section 3.10.1). Ideally, this would be ensured via formal verification, e.g., as it is possible with the Filet-o-Fish

Chapter 4. Implementation

(FoF) framework [17]. That framework introduces a semantic language, FoF, which is a safe abstraction of C. Since that language is purely functional, equational reasoning can be used to prove correctness for a DSL implemented in FoF. Unfortunately, Python does not have a specification but rather is defined by its reference implementation. These facts mean that formal verification would be a major project on its own, outside the scope of this dissertation.

However, STELLA was developed using the test-driven development methodology. For any feature that was implemented, first a functional test was written. Here the fact that all STELLA programs are also valid Python programs is important: each test runs exactly the same code first in the Python interpreter, then using the DSL compiler, and checks that the result is the same. This gives the user a high assurance that STELLA does not have bugs in the language features that are supported.

The tests were an essential tool during the development of STELLA. Major parts of the internal structure had to be rewritten several times to be able to support additional Python features. Any such undertaking without a complete test suite would be difficult to accomplish, because it is very easy to break existing code when focusing on one single issue at a time in a complex code base.

4.9 Debugging

When an error occurs, e.g., an unsupported Python feature is encountered, a regular back trace into the STELLA library is generated, but also annotated with the information the debugging information from the bytecode that triggered the error. Both pieces of information are important: 1) the back trace is necessary for fixing bugs in STELLA itself, or for extending it with more features, and 2) the user program source code location is important for the programmer when STELLA is not at fault.

Chapter 4. Implementation

There is no special debugging support for STELLA programs aside from this error reporting during compilation. Since the semantics are so close to Python's, the programmer can easily execute the program in Python, and use Python's excellent debugging facilities, e.g., `pdb`. Then once the bug has been corrected, the code can again be run in STELLA.

The other possibility is a bug in STELLA itself which causes wrong code to be generated. This kind of bug is inherently difficult to find, which is why STELLA uses so many tests for verification (see Section 4.8). It is easily possible to inspect the LLVM IR that STELLA generates. If additional debugging information became necessary, then the code generator could annotate the LLVM IR with meta-data to track the bytecode or source line they belong to. This would aid debugging with lower-level tools, e.g., `gdb`.

4.10 Bytecodes

The bytecodes that STELLA supports implement a core part of the functionality. They are a part of the IR-S introduced in Section 4.3. A brief description highlighting interesting behavior follows. All bytecodes are disassembled from the pre-compiled binary upon which CPython operates. Some bytecodes have arguments, which are also encoded in the binary blob, and looked up during the disassembly phase in one of the constant sections.

4.10.1 Basic Language Support

These bytecodes do not fall into a broader category but implement important basic functionality.

`LOAD_CONST` does not translate to code, instead it is used to generate a result at compile time, an LLVM constant with the value specified by its argument.

COMPARE_OP implements all comparison operators in Python. Which operator in particular is specified by an argument, and is equivalent in name to the LLVM comparison operators. The operands are taken from the stack, and in mixed integer and floating point comparisons cast to the more general type. The comparison is then performed by the LLVM `fcmp_oreder` or `icmp_signed` instruction, depending on the types involved. Since booleans are implemented as 8-bit integers, they are treated as such. The result of LLVM comparison operators is a 1-bit integer, which must be extended to 8-bit to be consistent with STELLA booleans.

RETURN_VALUE takes a value off the stack, and returns it. The returned type is unified with the current return type of the function. The code generated depends on the function return type: For `void`, the special `ret_void` LLVM function is used, otherwise a regular `return` instruction is generated. When the stack value is of object type, returning “None” results in the value `null` being returned. For other types, the stack value is returned, and returning “None” is not valid.

CALL_FUNCTION has one argument, which are two numbers packed into one integer (the number of positional arguments and the number of keyword arguments). First the function to be called is taken off the stack. Then for each positional argument, an item is taken off the stack. For each keyword argument, two items are taken off the stack: the argument name, and its value. Keyword arguments are reduced to regular positional arguments: The keyword arguments are combined with the function default values at compile time, and added to the list of positional arguments. This list is then used for the actual call.

4.10.2 Arithmetic

The bytecodes that implement the basic arithmetic operations.

BinaryOp The binary operators (`BINARY_ADD`, `BINARY_SUBTRACT`, `BINARY_MULTIPLY`, `BINARY_MODULO`) all operate in a similar fashion. There are no

Chapter 4. Implementation

arguments; the operands are taken from the stack. First they need to be unified in type, i.e., if both operands are not of the same type, the one of the less general type needs to be cast to the more general one. Then the operation is performed, and the result placed in a register onto the stack.

BINARY_POWER is used for the power operator (**). The stack interaction is identical to the other binary operators, but the translation requires more attention. Both LLVM functions `llvm.pow`, `llvm.powi` require the first argument to be a floating point number, so integers must be cast first. Integer exponents must be only 32bit wide, so STELLA's integers need to be truncated from their 64bit width. After calling the function the Python semantics are that when both the base and the exponent are integers, the result is also an integer. LLVM's power always returns a floating point number, therefore the result must be cast to an integer. See Section 3.10.1 for a discussion about negative integer exponents.

BINARY_FLOOR_DIVIDE is used for the “//” operator. The stack interaction is identical to the other binary operators. The Python semantics are implemented by a floating point division, followed by a call to `llvm.floor`. This requires integer operands to be converted to float first. And if both operands were integers originally, the result is cast back to an integer.

BINARY_TRUE_DIVIDE implements true division (the “/” operator), which always yields a floating point number. Therefore floating point division is used and integer operands are cast to floats first. The stack interaction is identical to the other binary operators.

INPLACE_* (`INPLACE_ADD`, `INPLACE_SUBTRACT`, `INPLACE_MULTIPLY`, `INPLACE_TRUE_DIVIDE`, `INPLACE_FLOOR_DIVIDE`, `INPLACE_MODULO`) These binary operands are implemented identical to their **BINARY_*** version, since the Python semantics are the same for scalar values, and the calling the corresponding *magic methods* of objects is not yet supported.

The bit-wise operators (**BINARY_AND**, **BINARY_OR**, **BINARY_XOR**) are implemented similar to the binary operators above, and use the corresponding LLVM instruction.

UNARY_NOT takes one value off the stack, and puts the result back onto the stack. Integer and float arguments are cast to boolean first. The actual operation is implemented as an XOR with the constant 1.

UNARY_NEGATIVE takes the operand off the stack, and places the result back onto the stack. It is implemented for integers or floating point numbers and subtracts the operand from zero.

4.10.3 Memory Interaction

The bytecodes which generally interact with memory.

LOAD_FAST takes the variable name as an argument. It loads the variable from memory into a register by placing it on the stack. The type of the register will be the same as the type of the variable.

STORE_FAST takes the variable name as an argument, and takes a register from the stack. It then stores the register in memory. The type of the memory location is unified with the type of the register.

LOAD_GLOBAL loads a global value, and places it onto the stack. A global can be a variety of items: a function, a module, a variable, an intrinsic, or a cast specification. A register is created only for global variables; for the remaining items their intermediate representation is placed onto the stack for further processing by the consuming bytecodes.

LOAD_ATTR loads the attribute specified by its argument, a string. The container object the attribute is loaded from is retrieved from the stack, and the result pushed back

Chapter 4. Implementation

onto the stack. Structures can either contain values, which are placed in registers, or methods, which are represented as IR-S objects. Arrays support the shape attribute, which contains the array dimensions. This is looked up in the array type, and is a constant value. The result is then placed onto the stack.

STORE_ATTR takes two items from the stack, the container and the value. The attribute name is the argument. The value is then placed into the structure at the offset specified by the attribute name, which is statically looked up in the container's type.

STORE_SUBSCR takes three values off the stack, the value to be stored, the container, and the index. The actual implementation depends on the type of the container (see Section 4.6).

BINARY_SUBSCR takes two values off the stack, the container and the index, and pushes the result back on the stack. The actual implementation depends on the type of the container (see Section 4.6).

UNPACK_SEQUENCE takes the sequence to unpack from the stack. The number of elements to extract, n , is directly supplied as an argument. The container item must be subscriptable. It places n items back onto the stack. This is semantically equivalent to n subscript operations with increasing indices.

BUILD_TUPLE creates a tuple on the stack by taking values off the stack. The number of items is supplied as an argument. These values are then subsequently inserted into the newly created tuple.

4.10.4 Control Flow

The bytecodes listed here manipulate the control flow of the program. Since LLVM requires all control flow to be explicit, each bytecode is marked whether it has the option to “fall through” to the next bytecode in the list, so that it can be properly accounted for

Chapter 4. Implementation

during the intra-procedural flow analysis (see Section 4.4.4).

SETUP_LOOP marks the beginning of a for-loop. Its argument is the relative offset to the end of the for-loop. No processing is done for this bytecode itself, instead it is used in the rewriting phase (see Section 4.4.3) and then removed.

Unconditional jumps (`JUMP_ABSOLUTE`, `JUMP_FORWARD`) both take one argument, the absolute location, or the relative offset, to jump to. The relative offset is immediately converted to an absolute location during disassembly; therefore their implementation is identically. The control flow is diverted to the destination bytecode, and there is no fall through.

Conditional jumps (`JUMP_IF_FALSE_OR_POP`, `JUMP_IF_TRUE_OR_POP`, `POP_JUMP_IF_FALSE`, `POP_JUMP_IF_TRUE`) takes the condition value off the stack. There are two independent variants for conditional jumps: when the condition is taken off the stack, and whether to jump when the condition true or false. For the first two variants (`JUMP_IF*_OR_POP`) the condition is only popped when the no jump takes place, while for the other two variants (`POP_JUMP_IF*`) an additional value is popped from the stack when a jump takes place. All variants fall through when the condition is not satisfied. Satisfied means that the condition is equal to `TRUE` or `FALSE`, as contained in the name of the bytecode.

4.10.5 Placeholders

These bytecodes are not directly involved in the code generation, but are recognized by the disassembly phase. Most of these bytecodes contain information which is used during the analysis. The remainder serves a function for CPython's stack machine but is redundant for STELLA, and removed during one of the analysis phases.

Chapter 4. Implementation

POP_BLOCK takes no arguments, and does not interact with the stack. It marks the end of a block, which is used, for example, to mark the end of a for-loop.

GET_ITER is not used in STELLA and removed during the rewrite process.

FOR_ITER is removed during the rewrite process. Its argument is the end of the for-loop body, however this information is not directly used since the `SETUP_LOOP` already contains a reference to the next bytecode after the loop ends.

POP_TOP removes one value from the stack.

DUP_TOP duplicates the value on top of the stack.

DUP_TOP_TWO takes two values off the stack, and then pushes both of them them back onto the stack twice in the same order.

ROT_TWO switches the top two values on top of the stack.

ROT_THREE takes three values off the stack, then puts them back in the following order: third, first, second.

RAISE_VARARGS calls the LLVM intrinsic `llvm.trap` to abort the program. This is at present not implemented in the *llvmlite* library and therefore leads to an error. See Section 6.1 for how this could be properly implemented in the future.

Evaluation

Some assumptions that were made at the beginning of this work turned out to be wrong. It was much more difficult than expected to obtain benchmark simulations written in Python because most people seem to only produce a rough prototype in Python before transitioning to another language. Then the Python version is abandoned. On the other hand, this is exactly a problem that STELLA can solve! So while this decreases the candidate pool for benchmarks, it also is a validating statement that no comparable solution exists today.

Four different benchmark programs were collected and evaluated in three ways: 1) How fast do they run, compared with the C version? 2) How many source code changes were necessary to be STELLA compatible? 3) How long does it take to compile the STELLA program?

Generally when scientists require a high-speed implementation, C is very often the language of choice. C is a very mature lower-level language¹ with highly optimized compilers. STELLA focuses on single-threaded execution, which matches C's native execution model. Together with the author's knowledge of C, it seemed like a natural choice to com-

¹At the time that C was introduced, it was actually considered a high-level language. However, over time as newer languages have become even more feature-full, I now classify C as a lower-level language.

Chapter 5. Evaluation

pare the performance against. All benchmarks were originally implemented in Python, if necessary modified for STELLA, and then translated by hand to C. The complete sources for all versions can be found in Appendix A.

Benchmark setup All benchmarks were run on an AMD FXTM-4100 CPU (Zambezi, 3.6GHz) with 16GB RAM, running Debian Linux (sid), kernel 4.0.0-2-amd64, Python 3.4.3, LLVM 3.5, gcc 4.9.3, clang 3.4, and STELLA revision 000f3bb5058c535bdf2bbb26e63af82381f84483. The C programs were compiled at -O3. STELLA used LLVM's optimization level 3.

Verification For each benchmark the end results of all versions are compared to verify that they did compute exactly the same answer, and therefore did the same amount of work. A difference of less than 10^{-7} was deemed a rounding error, and therefore did not constitute a different result.

The C benchmarks print their results as text, which is then parsed by Python. The verification is then performed by directly comparing Python data structures for equality. All benchmarks had the same results across each implementation/run-time. This implies that for the stochastic benchmark the random number generator was initialized with identical seeds, and hence all variants performed exactly the same amount of work.

5.1 Benchmark Description

The benchmarks used for the evaluation are:

Fibonacci This micro-benchmark recursively calculates a large Fibonacci number. Therefore the cost of function calls dominates the cost of arithmetic operations.

nbody The Debian project has an ongoing [programming language benchmark game](#). The [nbody benchmark](#) 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.

1D-spider This benchmark is a stochastic simulation in which a spider performs a random walk on a semi-infinite 1D surface.

heat Simulation of heat transfer by iteratively computing the finite difference to approximate the differential equations for heat transfer.

5.2 Language Design

The program design is evaluated by comparing the STELLA programs with their original Python version.

5.2.1 Required Program Modifications

For the purpose of this discussion the changes to the source code is summarized, because I feel that is more illustrative. However, a comparison showing actual source snippets can also be found in Appendix A.

Fibonacci This is a short and simple benchmark. No modification was necessary to run this function in STELLA.

1D spider This benchmark required only one simple modification. In STELLA objects are fixed, therefore it is not valid to initialize attributes inside of the simulation. In the original program some variables were initialized at the beginning of the `run()` method (time `self.t` and the time for the next observation `self.next_obs_time`). For STELLA this initialization was moved to the object constructor `__init__()`.

nbody The `combinations()` function pre-computes all combinations of bodies. It

Chapter 5. Evaluation

was removed and replaced by a nested for loop to iterate over the bodies on the fly, because STELLA does not yet support tuples as list elements. This is only a cosmetic change.

The nbody benchmark originally was written in a very compact style. The solar system was represented as a dictionary, associating each body with a name. The individual body was encoded as a tuple containing the position, the velocity, and the mass. Position and velocity were lists. This data structure is one choice but it is not easy to read. For example, you need to consult the code processing the data structure to see the variable names, since they are not encoded in the data structure itself.

As discussed in Section 3.1, STELLA does not support dictionaries, but for this benchmark the dictionary does not need to be removed: the dictionary is set to name the individual bodies, but the actual core iterates over the bodies as a list. Thus the dictionary is only used in the pre-processing phase and can remain unchanged. The benchmark was rewritten to use an object for each body of the solar system. While this is a larger change in the source code, it does not change the algorithmic properties of the main loop. The code becomes much more readable as a result. The original version executes more quickly within CPython, since it unpacks a tuple and then accesses the body attributes as local variables. The STELLA rewrite always accesses the object attribute, which is a slow process in Python due to the dynamic look-up process. Note that this change has no impact on the benchmark results in Section 5.1, since the reference point for comparison is the C version.

The next change is much smaller. The implementation does not support arbitrary expressions in calls to the range function. So the expression “range(j+1)” had to be rewritten as “m=j+1; range(m)”. This is only a cosmetic change and could be avoided by improving STELLA’s loop support.

The last change is related to the version of nbody that this benchmark is based on. It computes $\sqrt{(dx^2 + dy^2 + dz^2)^3}$, which is easy to implement in Python as $(dx^2 + dy^2 +$

Chapter 5. Evaluation

$dz^2)^{1.5}$. The latter does reduce the number of operations, but is in reality more costly: it invokes the `pow` function. The literal implementation executes much more efficiently since it avoids the costly `pow` call if multiplication is used to implement raising to the power of three. This appears to be true even for Python, since an [alternative version](#) of the benchmark changes the computation to also avoid `pow`. Therefore this change is not required for the STELLA implementation, as STELLA can call `pow`, but was included anyway to make the comparison with the C implementation more fair (which also uses `sqrt` and multiplication to avoid raising to the power of 1.5).

heat The original simulation used a GUI to continuously update a visual of the state of the simulation. The GUI class also contained the simulation code. Since GUI programs are not a focus of STELLA, the GUI was separated from the rest of the program, which is generally good practice.

Right now it is not possible to have a STELLA program continuously update a GUI, because the GUI runs inside the Python run-time and interaction with the run-time is expressibly forbidden. But properly separating GUI from the simulation core enables the programmer to optionally update the GUI, e.g., when prototyping on smaller simulation sizes, and then turn off the GUI for production runs with large simulations in STELLA.

The second change was to remove I/O from the simulation core, and instead store the values that would have been printed into an array for intermediate storage. This array is then output at the end of the simulation, when Python resumes the program execution. Again, this is generally good practice, and can be important when, e.g., benchmarking a simulation.

5.2.2 Summary of Source Changes

The changes that the surveyed programs required to run in STELLA can be categorized into two kinds, those expected by the language design and those required due to the state

Chapter 5. Evaluation

of the implementation. Some of the former were expected from the beginning, and the latter were not prevented by any design issues but simply by a lack of time to provide a more complete feature set.

I/O Input and output were required to be performed by Python from the outset. This is best practice anyway, unless the dataset exceeds main memory—a rare event for the simulations that STELLA is targeting.

Inability to use libraries It is not possible to add the simulation code to a GUI object, which is provided by a library. Again, this is a good practice anyway, and an expected change by the language design.

Cosmetic changes Lists containing tuples are simply not implemented yet. Aside from the fact that it is not particularly useful to pre-compute a nested for-loop that is only used twice, this could easily be added to STELLA with a little more implementation work.

5.3 Compilation Time

Recent developments in programming languages have shown a renewed interest in short compilation times, e.g., as the language *Go*², or the LLVM project, show. STELLA has even more reasons to provide fast compilation time: CPython is an interpreter, and therefore has no compilation overhead. So for STELLA to not only have similar semantics as Python, but to also feel similar in spirit to the programmer, it should feature fast compilation times. The faster the compilation is, the more seamless the experience will be for the programmer.

The compilation time is compared with the time it takes to compile the C sources that were used for the benchmarks. It is important to note, however, there is no one-

²Go, the programming language started by Google, lists “fast compilation” as one of its first goals in the [FAQ](#).

Chapter 5. Evaluation

to-one correspondence: STELLA takes the already parsed and processed bytecodes, and transforms them into LLVM IR. Compiling a C program does involve more work: The C compiler still has to do all the front-end work of opening the source code, parsing the text, creating an abstract syntax tree, and compiling assembly code. Normally it then first calls the assembler to create machine code, then calls the linker, and finally writes an executable to disk.

The following C compilation pipeline was used to create results that were more easily comparable to the compilation that STELLA performs presented in Figure 5.1:

1. The C compiler is called with `-E`, which results in pre-processed C source code. The result is saved in memory, and piped to the next stage.
2. The compiler is called with the options `-x cpp-output -S` so that it reads the preprocessed source code, and produces assembly language. This result is again stored in memory. Only this stage is timed, and compared with STELLA.
3. The last call uses the options `-x assembler` to read in the assembly code, produce machine code and link a final executable. Since this stage is not timed, the result is stored on disk for later execution.

Another difference is that STELLA is invoked from within Python—there is no process spawning, or I/O overhead. This was mitigated as much as possible by spawning the C compiler process first, and then interacting with it through standard input and output pipes. Therefore disk I/O is eliminated, and the process spawn time is excluded as well. Still, the C compiler and STELLA do slightly different work, because there is no interface to shortcut the C compiler front-end. These differences cannot be quantified exactly, but the comparison should still provide valuable insight into STELLA’s compilation overhead. The absolute times are also a useful measure by themselves.

On the other hand, the transformation of LLVM IR to machine code is performed in-

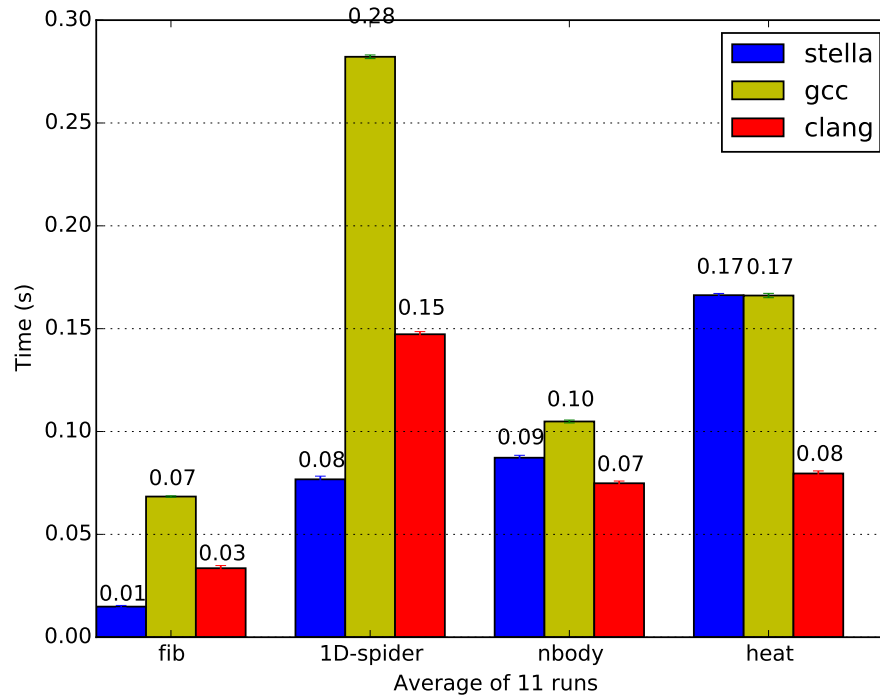


Figure 5.1: Compilation time comparison: C compilers from preprocessed source code to assemble, and STELLA from Python bytecodes to LLVM IR. I/O is eliminated by operating in memory. Error bars represent the standard error of the average runs.

ternally by LLVM’s JIT compiler, and is therefore included in the run-time of the STELLA benchmarks presented in Section 5.1.

Figure 5.1 shows the results. As the smallest benchmark, *fib*’s compilation time is very short, but nonetheless repeatable. For all programs STELLA is faster, or equally fast, as the C compiler. LLVM’s own clang stays true to its reputation to be faster than gcc. The C compiler does have some difficulties with the analysis of the 1D-spider benchmark, since it takes twice as long to compile as heat, while heat has almost twice the number of lines. STELLA’s compilation time, however, is not affected.

Benchmark	Compiler	Avg. run-time (s)	Std. Error	Slow-down
fib	stella	16.948	0.010	-
	C (avg)	21.861	1.632	1.29
	stella*	4.038	0.013	-
	python	314.298	5.165	77.84
1D-spider	C (avg)	98.696	0.401	-
	stella	111.123	0.696	1.13
	stella*	9.283	0.041	-
	python	245.912	5.019	26.49
nbody	stella	16.145	0.038	-
	C (avg)	20.937	0.969	1.30
	stella*	1.621	0.005	-
	python	302.785	2.671	186.77
heat	C (avg)	12.882	0.047	-
	stella	15.880	0.014	1.23
	stella*	0.959	0.002	-
	python	320.016	1.628	333.59

Table 5.1: The run-time of the benchmarks, smaller is better. The table is sorted by average run-time (of 10 runs), so the fastest is listed first, and the relative slow-down is always in reference to the fastest.

Python is so slow in comparison to STELLA that a separate set of reduced parameters was used. Python is only compared to the STELLA performance (marked by a *).

5.4 Performance

STELLA performs competitively with the C implementations as Table 5.1 shows. It is important to note that the STELLA run-time includes the JIT compilation cost, but excludes the time STELLA takes to create the LLVM IR. The excluded compilation time has already been discussed in Section 5.3.

The performance of Python was so much slower that it was necessary to re-run the benchmarks with reduced parameters to get an acceptable Python run-time. The parameters that each benchmark program used for the performance evaluation were:

Chapter 5. Evaluation

Fibonacci calculates the 48th number in the Fibonacci sequence for comparing to C, and 45th for comparing to Python.

1D-spider walks until 1.2×10^9 time has passed for comparing to C, and 10^8 when comparing to Python. Time is an abstract measure in this simulation.

nbody advances the system for 10^8 steps when comparing to C, and 10^7 steps when comparing to Python.

heat calculates 50,000 iterations when comparing to C, and 3000 iterations when comparing to Python.

Two C compilers were used, gcc and clang. Since the performance variance between the two was relatively high (but non-uniform), STELLA is compared to the average performance of the two (see Appendix A.5 for the individual results).

Table 5.1 is sorted by lowest run-time, and the slow-down is always in reference to the fastest implementation. The averages are over 10 runs on identical inputs (and identical random number generator state). On average STELLA is 7% faster than C. Without the micro-benchmark STELLA is 0.1% slower than C. If the fastest C version is used for every benchmark, then STELLA is 12% slower. On the other hand, if the slowest C version is used for every benchmark, then STELLA is 26% faster.

Conclusion

STELLA is a new embedded domain-specific language initially aimed at writing scientific simulations within Python. It is a compiled language, and executes at C-like speed. The novel approach goes against the trend of complete integration, meaning that it will not be possible to call Python code from within the DSL. 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: the OOP patterns are rewritten in the compilation process.

At the core the language is relatively simple. But this simplicity is what allows STELLA to generate such highly efficient code that it is in many cases on par with the performance of C! The embedding and seamless execution make it very easy to use STELLA, which is important because then Python can augment functionality when performance is not critical without much programmer effort. Another important component is the dual use of the source code. Since the programs are also valid Python code, they can still be prototyped and debugged in Python. The difference from other prototyping approaches is that for putting the code into production no lengthy rewrite process is required. On average just

Chapter 6. Conclusion

a few source changes are necessary, and in some cases the code will even run untouched. When problems do occur, the programmer can just run the identical code in Python to comfortably debug it. In some sense this is a manual version of the deoptimization performed by run-time systems like Truffle [53]. STELLA is less automated, but offers higher predictability.

The benchmarks show that STELLA's performance is competitive with programs that were rewritten in C by hand. The benchmark programs only required minimal modification to remove unsupported language features, and some of them were only due to limitations of the current implementation. And finally, the compilation time was short in absolute terms, less than 0.2s in all cases, which is competitive with the time it takes to compile C programs.

This dissertation demonstrates that it is easy to write very fast simulations while still having OOP available and many of the comforts of modern high-level languages. STELLA makes it possible for the scientist to focus more on the problem at hand. It is a challenge to write well-structured and well-performing simulations, which can be a major cost and time factor. By using STELLA it is possible for scientists to skip some of the tedious work involved with lower-level languages, as well as work with an easy to use platform that gives him many of the high-level tools often expected today. This does not only benefit computer scientists but also scientists with less traditional training in programming.

The implementation is published at <https://github.com/squisher/stella>.

6.1 Future Work

A programming language can always be improved. For STELLA the goals are particularly easy, supporting more Python features while respecting the goal to produce fast code. Theoretically the sky is the limit, as improved static analysis can always provide even more

Chapter 6. Conclusion

insight. That in turns allows generating code adhering to the principle of constant cost for features that aren't currently supported. A concrete list of what would be investigated next is:

Magic methods More object-oriented features could be supported, in particular the so-called *magic methods* would be useful syntactic sugar: the purpose of these features is mainly convenience, although there are exceptions. For example, the `__getitem__` and `__setitem__` method names allow the programmer to use the subscription operator on an object. It is then possible that subscripting does not have a constant cost. At first glance this may contradict Section 3.1. But because the programmer implemented these functions, he is aware of the cost, and can maintain standard Python programming styles better.

Exceptions General *exception handling* is likely to incur too much run-time overhead, but it is possible to use exceptions to abort the DSL execution and to report errors back to Python. So a “try: ...catch: ...” block would not be supported, but the STELLA library could install a special mechanism to catch errors, and then raise an exception from within Python.

Supporting functional features Python implements a number of functional programming features, e.g., list comprehensions, anonymous functions, function references. Some of these features are not dynamic in nature, but do involve memory management. An improved static analysis might be able to determine the bounds of the operations and insert static memory management code for the programmer. It seems likely that this will not be possible in all situations. Further research is necessary to identify when static bounds can be determined and when not; and a clear description of applicability is a must so that the programmer can easily use the supported functional features.

Supporting more types The most essential types `int` and `float` are available. Support could be added for `string` types, although care must be taken to provide a clean implementation that does not involve too much memory management. Python 3 supports

Chapter 6. Conclusion

native complex types, which would potentially be useful to support.

Fine grained numerical types Python does not have different widths of data types, i.e., only *int* exists and no *int32* or *uint8*. The current implementation translates to the most general machine type, i.e., the widest one. In certain situations this can have a significant effect on the performance, since much more memory is used than required. Static, or dynamic, analysis [45] can determine the bound in some cases, and then select a narrower type for some variables. NumPy also supports narrower type definitions, which could be used for manual annotation (i.e., using *mypy*).

Subtypes Section 3.8 explains why STELLA does not model subtyping relationships. It would be possible to extend the type system to model subtypes, and continue to accept objects constructed using full multiple inheritance—classes constructed that way would simply be incompatible with their parent classes.

Distributed computing Python does not have native support for multicore or distributed computing. It should be possible to integrate some support akin to the standard library module *multiprocessing*. Whether STELLA would need to implement this support itself, or can operate under the Python implementation’s umbrella is yet to be determined.

IPC Using multiple processes could also lead to new communication channels between Python and STELLA. Efficient and flexible IPC mechanisms such as *zeromq* [4] could be used to exchange data with a Python interpreter running in a different process. This could enable STELLA to be used in very different application domains aside from scientific simulations where computationally intensive tasks are performed but more frequent interaction with Python is required.

LLVM optimization passes The LLVM IR is designed particularly to allow advanced analysis and optimization to be performed. I would investigate if STELLA programs have patterns in the generated LLVM IR which could benefit from custom LLVM optimization passes.

Chapter 6. Conclusion

Improve NumPy support Currently only NumPy's array type is used. It would be an interesting opportunity to investigate integrating more NumPy features, e.g., numerical evaluation of integrals. This task mainly depends on the internal data representation of NumPy, and its C API.

Benchmark Details

This section lists the source code of some programs for reference. This serves as an additional perspective on the benchmark programs although the discussion in the main text mentions all relevant source snippets. Please note that all source code has been reformatted to fit the maximum possible line length in this document.

This is particularly true for the programs used to evaluate STELLA. The main text (Section 5.2) discusses the changes to the Python programs that were necessary to have them run in STELLA. The changes listed here show the relevant source code. The differences are still summarized for clarity. The benchmarks in Section 5.1 compare the performance of STELLA against the C version of the programs. For reference, the source code is included here as well.

The source code for the “GenericSpiderSim” program used in the initial exploratory work.

A.1 Fibonacci

This is a baseline benchmark that does not require further discussion.

A.1.1 Python and Stella Source

```
1 def fib(x):
2     if x <= 2:
3         return 1
4     return fib(x - 1) + fib(x - 2)
```

A.1.2 C Source

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 long long fib(long long x) {
5     if (x <= 2) {
6         return 1;
7     } else {
8         return fib(x-1) + fib(x-2);
9     }
10 }
11
12 int main(int argc, char ** argv) {
13     long long r = 0;
14     const int {{x_init}};
15
16     r += fib(x);
17
18     printf ("%lld\n", r);
19     exit (0);
20 }
```

A.2 1D Spider

A.2.1 Python and Stella Source

Only the STELLA version of the code is presented here because it differs from the Python original only in one line, and remains valid Python code. See Section A.2.2 below for the difference between the versions.

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
12 # implied. See the License for the specific language governing
13 # permissions and limitations under the License.
14 """
15 Semi-infinite 1D strip with a single spider.
16 """
17
18 import mtpy # cython wrapper around mtwist
19 from math import log, exp
20 import time
21 from numpy import zeros
22
23 import virtnet_utils
24
25
26 class Settings(virtnet_utils.Settings):
27     def setDefaults(self):
28         self.settings = {
29             'seed': [int(time.time()), int],
30             'r': [0.1, float],
```

Appendix A. Benchmark Details

```
31         'koffp': [1.0, float],
32         'K': [10, int],
33         'rununtiltime': [1e3, float],
34         'elapsedTime': [self.elapsedTime, lambda x:x],
35     }
36
37
38 def mtpy_exp(p):
39     u = 1.0 - mtpy.mt_drand()
40     return -log(u) / p
41
42
43 class Simulation(object):
44     EXPSTART = 0.2
45
46     def __init__(self, params):
47         self.K = params['K']
48         self.rununtiltime = params['rununtiltime']
49         mtpy.mt_seed32new(params['seed'])
50         self.koffp = params['koffp']
51         self.kcat = params['r']
52
53         self.delta = ((log(self.rununtiltime) - log(self.EXPSTART)) /
54                      float(self.K - 1))
55         self.leg = 0
56         self.substrate = 0
57         self.obs_i = 0
58         self.observations = zeros(shape=self.K, dtype=int)
59         # The following initializations are added here for Stella
60         self.t = 0.0
61         self.next_obs_time = 0.0
62
63     def __str__(self):
64         return "{}: {}>".format(super().__str__()[:-1],
65                                self.observations)
66
67     def __eq__(self, o):
68         assert isinstance(o, self.__class__)
69         return (self.observations == o.observations).all()
70
71     def makeObservation(self):
72         """Called from run()"""
73         self.observations[self.obs_i] = self.leg
```

Appendix A. Benchmark Details

```
74         self.obs_i += 1
75
76         self.next_obs_time = self.getNextObsTime()
77
78     def getNextObsTime(self):
79         """Called from run()"""
80         if self.obs_i == 0:
81             return self.EXPSTART
82         if self.obs_i == self.K - 1:
83             return self.rununtiltime
84
85         return exp(log(self.EXPSTART) + self.delta * self.obs_i)
86
87     def step(self):
88         """Called from run()"""
89         if self.leg == 0:
90             self.leg += 1
91         else:
92             u1 = mtpy.mt_drand()
93             if u1 < 0.5:
94                 self.leg -= 1
95             else:
96                 self.leg += 1
97         if self.leg == self.substrate:
98             self.substrate += 1
99
100     def isNextObservation(self):
101         return self.t > self.next_obs_time and self.obs_i < self.K
102
103     def run(self):
104         self.t = 0.0
105         self.next_obs_time = self.getNextObsTime()
106
107         # Stella: Declaring R here is not necessary in Python
108         R = 0.0
109
110         while self.obs_i < self.K and self.t < self.rununtiltime:
111             if self.leg < self.substrate:
112                 R = self.koffp
113             else:
114                 R = self.kcat
115             self.t += mtpy_exp(R)
116
```

Appendix A. Benchmark Details

```
117         while self.isNextObservation():
118             self.makeObservation()
119
120         self.step()
121
122
123 if __name__ == '__main__':
124     s = Settings()
125     sim_py = Simulation(s)
126     sim_py.run()
127
128     print(sim_py.observations)
```

A.2.2 Change Summary

Consider the following attribute initialization:

```
1         self.t = 0.0
2         self.next_obs_time = 0.0
```

This was originally performed at the start of the `run()` method. But that is not valid in STELLA since the attributes of `self` are fixed once the DSL starts. Therefore the solution is to move the initialization out of `run()` and into the object initialization method `__init__()`

A.2.3 C Source

```
1 /*
2  * Copyright 2013-2015 David Mohr
3  *
4  * Licensed under the Apache License, Version 2.0 (the "License");
5  * you may not use this file except in compliance with the License.
6  * You may obtain a copy of the License at
7  *
```

Appendix A. Benchmark Details

```
8  *      http://www.apache.org/licenses/LICENSE-2.0
9  *
10 * Unless required by applicable law or agreed to in writing, software
11 * distributed under the License is distributed on an "AS IS" BASIS,
12 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
13 * implied. See the License for the specific language governing
14 * permissions and limitations under the License.
15 */
16 #include <stdio.h>
17 #include <stdlib.h>
18 #define lint // get rid of unused variable warning in mtwist.c
19 #include "../mtpy/mtwist-1.1/mtwist.c"
20 #include <math.h>
21
22 typedef struct {
23     int K;
24     double rununtiltime;
25     int seed;
26     double koffp;
27     double kcat;
28
29     double t;
30     double delta;
31     int leg;
32     int substrate;
33     int obs_i;
34     int * observations;
35     double next_obs_time;
36 } spider_t;
37 const double EXPSTART = 0.2;
38
39 double uniform() {
40     return mt_drand();
41 }
42
43 double mtpy_exp(double p) {
44     double u = 1.0 - uniform();
45     return -log(u)/p;
46 }
47
48 double getNextObsTime(spider_t *sp) {
49     //""""Called from run()""""
50     //global obs_i, EXPSTART, rununtiltime, delta
```


Appendix A. Benchmark Details

```
51
52     if (sp->obs_i == 0) {
53         return EXPSTART;
54     }
55     if (sp->obs_i==sp->K-1) {
56         return sp->rununtiltime;
57     }
58
59     return exp(log(EXPSTART)+sp->delta*sp->obs_i);
60 }
61
62 void makeObservation(spider_t *sp) {
63     sp->observations[sp->obs_i] = sp->leg;
64     sp->obs_i += 1;
65
66     sp->next_obs_time = getNextObsTime(sp);
67 }
68
69 void step(spider_t *sp) {
70     ///""Called from run()""
71     ///global leg, substrate
72     if (sp->leg == 0)
73         sp->leg += 1;
74     else {
75         double u1 = uniform();
76         if (u1 < 0.5)
77             sp->leg -= 1;
78         else
79             sp->leg += 1;
80     }
81     if (sp->leg == sp->substrate)
82         sp->substrate += 1;
83 }
84
85 int isNextObservation(spider_t *sp) {
86     return sp->t > sp->next_obs_time && sp->obs_i < sp->K;
87 }
88
89 void run(spider_t *sp) {
90     sp->next_obs_time = getNextObsTime(sp);
91
92     double R = 0.0;
93     while (sp->obs_i < sp->K && sp->t < sp->rununtiltime) {
```

Appendix A. Benchmark Details

```
94     if (sp->leg < sp->substrate)
95         R = sp->koffp;
96     else
97         R = sp->kcat;
98     sp->t += mtpy_exp(R);
99
100    while (isNextObservation(sp)) {
101        makeObservation(sp);
102    }
103
104    step(sp);
105 }
106 }
107
108 void init(spider_t *sp) {
109     sp->K = 10;
110     sp->{{rununtiltime_init}};
111     sp->{{seed_init}};
112     sp->koffp = 1.0;
113     sp->kcat = 0.1;
114     sp->t = 0.0;
115     sp->leg = 0;
116     sp->substrate = 0;
117     sp->obs_i = 0;
118     sp->delta = (log(sp->rununtiltime)-log(EXPSTART)) \
119                 /(double)(sp->K-1);
120     sp->observations = (int *) malloc (sizeof(int) * sp->K);
121
122     mt_seed32new(sp->seed);
123 }
124
125 int main(int argc, char ** argv) {
126     spider_t sp;
127
128     init(&sp);
129
130     run(&sp);
131
132     int i;
133     printf ("[");
134     for (i=0; i<sp.K-1; i++) {
135         printf ("%d ", sp.observations[i]);
136     }
```

Appendix A. Benchmark Details

```
137     printf ("%d]\n", sp.observations[i]);
138
139     free(sp.observations);
140
141     exit (0);
142 }
```

A.3 nbody

A.3.1 Python Original

```
1 #!/usr/bin/env python3
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 #     http://www.apache.org/licenses/LICENSE-2.0
9 #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
13 # implied. See the License for the specific language governing
14 # permissions and limitations under the License.
15 # The Computer Language Benchmarks Game
16 # http://benchmarksgame.alioth.debian.org/
17 #
18 # originally by Kevin Carson
19 # modified by Tupteq, Fredrik Johansson, and Daniel Nanz
20 # modified by Maciej Fijalkowski
21 # 2to3
22
23 import sys
24 import math
25
26 def combinations(l):
27     result = []
```

Appendix A. Benchmark Details

```
28     for x in range(len(l) - 1):
29         ls = l[x+1:]
30         for y in ls:
31             result.append((l[x],y))
32     return result
33
34 PI = 3.14159265358979323
35 SOLAR_MASS = 4 * PI * PI
36 DAYS_PER_YEAR = 365.24
37
38 BODIES = {
39     'sun': ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),
40
41     'jupiter': ([4.84143144246472090e+00,
42                 -1.16032004402742839e+00,
43                 -1.03622044471123109e-01],
44                [1.66007664274403694e-03 * DAYS_PER_YEAR,
45                 7.69901118419740425e-03 * DAYS_PER_YEAR,
46                 -6.90460016972063023e-05 * DAYS_PER_YEAR],
47                9.54791938424326609e-04 * SOLAR_MASS),
48
49     'saturn': ([8.34336671824457987e+00,
50                4.12479856412430479e+00,
51                -4.03523417114321381e-01],
52               [-2.76742510726862411e-03 * DAYS_PER_YEAR,
53                4.99852801234917238e-03 * DAYS_PER_YEAR,
54                2.30417297573763929e-05 * DAYS_PER_YEAR],
55               2.85885980666130812e-04 * SOLAR_MASS),
56
57     'uranus': ([1.28943695621391310e+01,
58                -1.51111514016986312e+01,
59                -2.23307578892655734e-01],
60               [2.96460137564761618e-03 * DAYS_PER_YEAR,
61                2.37847173959480950e-03 * DAYS_PER_YEAR,
62                -2.96589568540237556e-05 * DAYS_PER_YEAR],
63               4.36624404335156298e-05 * SOLAR_MASS),
64
65     'neptune': ([1.53796971148509165e+01,
66                 -2.59193146099879641e+01,
67                 1.79258772950371181e-01],
68                [2.68067772490389322e-03 * DAYS_PER_YEAR,
69                 1.62824170038242295e-03 * DAYS_PER_YEAR,
70                 -9.51592254519715870e-05 * DAYS_PER_YEAR],
```

Appendix A. Benchmark Details

```
71             5.15138902046611451e-05 * SOLAR_MASS) }
72
73
74 SYSTEM = list(BODIES.values())
75 PAIRS = combinations(SYSTEM)
76
77
78 def advance(dt, n, bodies=SYSTEM, pairs=PAIRS):
79
80     for i in range(n):
81         for (([x1, y1, z1], v1, m1),
82              ([x2, y2, z2], v2, m2)) in pairs:
83             dx = x1 - x2
84             dy = y1 - y2
85             dz = z1 - z2
86
87             # dist = math.sqrt(dx * dx + dy * dy + dz * dz)
88             # mag = dt / (dist*dist*dist)
89             mag = dt * ((dx * dx + dy * dy + dz * dz) ** (-1.5))
90
91             b1m = m1 * mag
92             b2m = m2 * mag
93             v1[0] -= dx * b2m
94             v1[1] -= dy * b2m
95             v1[2] -= dz * b2m
96             v2[0] += dx * b1m
97             v2[1] += dy * b1m
98             v2[2] += dz * b1m
99         for (r, [vx, vy, vz], m) in bodies:
100             r[0] += dt * vx
101             r[1] += dt * vy
102             r[2] += dt * vz
103
104
105 def report_energy(bodies=SYSTEM, pairs=PAIRS, e=0.0):
106
107     for (([x1, y1, z1], v1, m1),
108          ([x2, y2, z2], v2, m2)) in pairs:
109         dx = x1 - x2
110         dy = y1 - y2
111         dz = z1 - z2
112         e -= (m1 * m2) / ((dx * dx + dy * dy + dz * dz) ** 0.5)
113     for (r, [vx, vy, vz], m) in bodies:
```

Appendix A. Benchmark Details

```
114         e += m * (vx * vx + vy * vy + vz * vz) / 2.
115     print("%.9f" % e)
116
117 def offset_momentum(ref, bodies=SYSTEM, px=0.0, py=0.0, pz=0.0):
118
119     for (r, [vx, vy, vz], m) in bodies:
120         px -= vx * m
121         py -= vy * m
122         pz -= vz * m
123     (r, v, m) = ref
124     v[0] = px / m
125     v[1] = py / m
126     v[2] = pz / m
127
128 def main(n, ref='sun'):
129     offset_momentum(BODIES[ref])
130     report_energy()
131     advance(0.01, n)
132     report_energy()
133
134 if __name__ == '__main__':
135     main(int(sys.argv[1]))
```

A.3.2 Modified for Stella

```
1 #!/usr/bin/env python3
2 # The Computer Language Benchmarks Game
3 # http://benchmarksgame.alioth.debian.org/
4 #
5 # originally by Kevin Carson
6 # modified by Tupteq, Fredrik Johansson, and Daniel Nanz
7 # modified by Maciej Fijalkowski
8 # 2to3
9 # modified by David Mohr
10 #
11 ##
12 # This is a specific instance of the Open Source Initiative (OSI) BSD
13 # license template: http://www.opensource.org/licenses/bsd-license.php
14 ##
15 # Revised BSD license
```

Appendix A. Benchmark Details

```
16 #
17 # Copyright © 2004-2008 Brent Fulgham, 2005-2015 Isaac Gouy,
18 #           2015 David Mohr
19 #
20 # All rights reserved.
21 #
22 # Redistribution and use in source and binary forms, with or without
23 # modification, are permitted provided that the following conditions
24 # are met:
25 #
26 # - Redistributions of source code must retain the above copyright
27 #   notice, this list of conditions and the following disclaimer.
28 # - Redistributions in binary form must reproduce the above copyright
29 #   notice, this list of conditions and the following disclaimer in
30 #   the documentation and/or other materials provided with the
31 #   distribution.
32 # - Neither the name of "The Computer Language Benchmarks Game" nor
33 #   the name of "The Computer Language Shootout Benchmarks" nor the
34 #   names of its contributors may be used to endorse or promote
35 #   products derived from this software without specific prior written
36 #   permission.
37 #
38 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
39 # "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
40 # LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
41 # FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
42 # COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
43 # INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
44 # BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
45 # LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
46 # CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
47 # LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
48 # ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
49 # POSSIBILITY OF SUCH DAMAGE.
50
51
52 import sys
53 import copy
54 import math
55 try:
56     from . import mark, unimplemented
57     parametrize = mark.parametrize
58 except SystemError:
```

Appendix A. Benchmark Details

```
59     def unimplemented(f):
60         return f
61
62     def parametrize(*args):
63         return unimplemented
64
65
66     PI = 3.14159265358979323
67     SOLAR_MASS = 4 * PI * PI
68     DAYS_PER_YEAR = 365.24
69
70     DELTA = 0.0000001
71
72
73     class Body(object):
74         def __init__(self, p, v, mass):
75             self.x, self.y, self.z = p
76             self.vx, self.vy, self.vz = v
77             self.mass = mass
78
79         def __repr__(self):
80             v = (self.x, self.y, self.z,
81                 self.vx, self.vy, self.vz,
82                 self.mass)
83             return "Body([{} , {} , {} ]->[{} , {} , {} ]@{})".format(v)
84
85         def diff(self, o):
86             for a in ['x', 'y', 'z', 'vx', 'vy', 'vz', 'mass']:
87                 me = getattr(self, a)
88                 it = getattr(o, a)
89                 if abs(me - it) >= DELTA:
90                     v = (a, me, it, me - it, DELTA)
91                     raise Exception('{}: {} - {} = {} > {}'.format(v))
92
93
94     BODIES = {
95         'sun': Body([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),
96
97         'jupiter': Body([4.84143144246472090e+00,
98                         -1.16032004402742839e+00,
99                         -1.03622044471123109e-01],
100                        [1.66007664274403694e-03 * DAYS_PER_YEAR,
101                        7.69901118419740425e-03 * DAYS_PER_YEAR,
```


Appendix A. Benchmark Details

```
102             -6.90460016972063023e-05 * DAYS_PER_YEAR],
103             9.54791938424326609e-04 * SOLAR_MASS),
104
105     'saturn': Body([8.34336671824457987e+00,
106                   4.12479856412430479e+00,
107                   -4.03523417114321381e-01],
108                   [-2.76742510726862411e-03 * DAYS_PER_YEAR,
109                   4.99852801234917238e-03 * DAYS_PER_YEAR,
110                   2.30417297573763929e-05 * DAYS_PER_YEAR],
111                   2.85885980666130812e-04 * SOLAR_MASS),
112
113     'uranus': Body([1.28943695621391310e+01,
114                   -1.51111514016986312e+01,
115                   -2.23307578892655734e-01],
116                   [2.96460137564761618e-03 * DAYS_PER_YEAR,
117                   2.37847173959480950e-03 * DAYS_PER_YEAR,
118                   -2.96589568540237556e-05 * DAYS_PER_YEAR],
119                   4.36624404335156298e-05 * SOLAR_MASS),
120
121     'neptune': Body([1.53796971148509165e+01,
122                    -2.59193146099879641e+01,
123                    1.79258772950371181e-01],
124                    [2.68067772490389322e-03 * DAYS_PER_YEAR,
125                    1.62824170038242295e-03 * DAYS_PER_YEAR,
126                    -9.51592254519715870e-05 * DAYS_PER_YEAR],
127                    5.15138902046611451e-05 * SOLAR_MASS),
128 }
129
130 SYSTEM = list(BODIES.values())
131
132 def advance(dt, n, bodies):
133     for i in range(n):
134         for j in range(len(bodies)):
135             m = j+1 # Stella workaround
136             for k in range(m, len(bodies)):
137                 dx = bodies[j].x - bodies[k].x
138                 dy = bodies[j].y - bodies[k].y
139                 dz = bodies[j].z - bodies[k].z
140
141                 # This is extremely slow because of pow (**)
142                 # mag = dt * ((dx * dx + dy * dy + dz * dz) ** (-1.5))
143                 dist = math.sqrt(dx * dx + dy * dy + dz * dz)
144                 mag = dt / (dist * dist * dist)
```

Appendix A. Benchmark Details

```
145
146         b1m = bodies[j].mass * mag
147         b2m = bodies[k].mass * mag
148         bodies[j].vx -= dx * b2m
149         bodies[j].vy -= dy * b2m
150         bodies[j].vz -= dz * b2m
151         bodies[k].vx += dx * b1m
152         bodies[k].vy += dy * b1m
153         bodies[k].vz += dz * b1m
154     for j in range(len(bodies)):
155         bodies[j].x += dt * bodies[j].vx
156         bodies[j].y += dt * bodies[j].vy
157         bodies[j].z += dt * bodies[j].vz
158
159 def calculate_energy(bodies, e=0.0):
160     for j in range(len(bodies)):
161         m = j+1 # Stella workaround
162         for k in range(m, len(bodies)):
163             dx = bodies[j].x - bodies[k].x
164             dy = bodies[j].y - bodies[k].y
165             dz = bodies[j].z - bodies[k].z
166             e -= ((bodies[j].mass * bodies[k].mass) /
167                  ((dx * dx + dy * dy + dz * dz) ** 0.5))
168     for i in range(len(bodies)):
169         e += bodies[i].mass * (bodies[i].vx * bodies[i].vx +
170                               bodies[i].vy * bodies[i].vy +
171                               bodies[i].vz * bodies[i].vz) / 2.
172     return e
173
174 def report_energy(bodies, e=0.0):
175     print("%.9f" % calculate_energy(bodies, e))
176
177 def offset_momentum(ref, bodies, px=0.0, py=0.0, pz=0.0):
178     for i in range(len(bodies)):
179         px -= bodies[i].vx * bodies[i].mass
180         py -= bodies[i].vy * bodies[i].mass
181         pz -= bodies[i].vz * bodies[i].mass
182     ref.vx = px / ref.mass
183     ref.vy = py / ref.mass
184     ref.vz = pz / ref.mass
185
186 def init():
187     system = copy.deepcopy(SYSTEM)
```

Appendix A. Benchmark Details

```
188     offset_momentum(system[0], system)
189     return system
190
191 def main(n, wrapper=lambda x: x):
192     system = init()
193     report_energy(system)
194     r = wrapper(advance)(0.01, n, system)
195     report_energy(system)
196     return r
197
198 if __name__ == '__main__':
199     main(int(sys.argv[1]))
```

A.3.3 Change Summary

Data structure Instead of the original:

```
1 BODIES = {
2     'sun': ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),
3
4     'jupiter': ([4.84143144246472090e+00,
5                 -1.16032004402742839e+00,
6                 -1.03622044471123109e-01],
7                 [1.66007664274403694e-03 * DAYS_PER_YEAR,
8                 7.69901118419740425e-03 * DAYS_PER_YEAR,
9                 -6.90460016972063023e-05 * DAYS_PER_YEAR],
10                9.54791938424326609e-04 * SOLAR_MASS),
11     # ...
12 }
```

STELLA uses:

```
1 class Body(object):
2     def __init__(self, p, v, mass):
3         (self.x, self.y, self.z) = p
4         (self.vx, self.vy, self.vz) = v
```

Appendix A. Benchmark Details

```
5     self.mass = mass
6
7
8 BODIES = {
9     'sun': Body([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),
10
11     'jupiter': Body([4.84143144246472090e+00,
12                     -1.16032004402742839e+00,
13                     -1.03622044471123109e-01],
14                     [1.66007664274403694e-03 * DAYS_PER_YEAR,
15                     7.69901118419740425e-03 * DAYS_PER_YEAR,
16                     -6.90460016972063023e-05 * DAYS_PER_YEAR],
17                     9.54791938424326609e-04 * SOLAR_MASS),
18     # ...
```

The loop changes accordingly from

```
1 pairs = combinations(SYSTEM)
2 # ...
3     for (([x1, y1, z1], v1, m1),
4          ([x2, y2, z2], v2, m2)) in pairs:
5         # ...
```

to the STELLA variant

```
1 bodies = SYSTEM
2 # ...
3     for j in range(len(bodies)):
4         m = j+1
5         for k in range(m, len(bodies)):
6             # ...
```

range Line 4 above was added because in STELLA the range expression on line 5 does not support “j+1” as an argument.

pow The original computation

Appendix A. Benchmark Details

```
1 mag = dt * ((dx * dx + dy * dy + dz * dz) ** (-1.5))
```

now avoids the pow operator “**” by using the equivalent

```
1 dist = math.sqrt(dx * dx + dy * dy + dz * dz)
2 mag = dt / (dist * dist * dist)
```

A.3.4 C Source

```
1 /* The Computer Language Benchmarks Game
2  * http://benchmarksgame.alioth.debian.org/
3  *
4  * contributed by Christoph Bauer
5  *
6  */
7
8 #include <math.h>
9 #include <stdio.h>
10 #include <stdlib.h>
11
12 #define pi 3.141592653589793
13 #define solar_mass (4 * pi * pi)
14 #define days_per_year 365.24
15
16 struct planet {
17     double x, y, z;
18     double vx, vy, vz;
19     double mass;
20 };
21
22 void advance(int nbodies, struct planet * bodies, double dt)
23 {
24     int i, j;
25
26     for (i = 0; i < nbodies; i++) {
27         struct planet * b = &(bodies[i]);
```

Appendix A. Benchmark Details

```
28     for (j = i + 1; j < nbodies; j++) {
29         struct planet * b2 = &(bodies[j]);
30         double dx = b->x - b2->x;
31         double dy = b->y - b2->y;
32         double dz = b->z - b2->z;
33         double distance = sqrt(dx * dx + dy * dy + dz * dz);
34         double mag = dt / (distance * distance * distance);
35         b->vx -= dx * b2->mass * mag;
36         b->vy -= dy * b2->mass * mag;
37         b->vz -= dz * b2->mass * mag;
38         b2->vx += dx * b->mass * mag;
39         b2->vy += dy * b->mass * mag;
40         b2->vz += dz * b->mass * mag;
41     }
42 }
43 for (i = 0; i < nbodies; i++) {
44     struct planet * b = &(bodies[i]);
45     b->x += dt * b->vx;
46     b->y += dt * b->vy;
47     b->z += dt * b->vz;
48 }
49 }
50
51 double energy(int nbodies, struct planet * bodies)
52 {
53     double e;
54     int i, j;
55
56     e = 0.0;
57     for (i = 0; i < nbodies; i++) {
58         struct planet * b = &(bodies[i]);
59         e += 0.5 * b->mass * (b->vx * b->vx
60                             + b->vy * b->vy
61                             + b->vz * b->vz);
62         for (j = i + 1; j < nbodies; j++) {
63             struct planet * b2 = &(bodies[j]);
64             double dx = b->x - b2->x;
65             double dy = b->y - b2->y;
66             double dz = b->z - b2->z;
67             double distance = sqrt(dx * dx + dy * dy + dz * dz);
68             e -= (b->mass * b2->mass) / distance;
69         }
70     }
```

Appendix A. Benchmark Details

```
71 return e;
72 }
73
74 void offset_momentum(int nbodies, struct planet * bodies)
75 {
76     double px = 0.0, py = 0.0, pz = 0.0;
77     int i;
78     for (i = 0; i < nbodies; i++) {
79         px += bodies[i].vx * bodies[i].mass;
80         py += bodies[i].vy * bodies[i].mass;
81         pz += bodies[i].vz * bodies[i].mass;
82     }
83     bodies[0].vx = - px / solar_mass;
84     bodies[0].vy = - py / solar_mass;
85     bodies[0].vz = - pz / solar_mass;
86 }
87
88 #define NBODIES 5
89 struct planet bodies[NBODIES] = {
90     { /* sun */
91         0, 0, 0, 0, 0, 0, solar_mass
92     },
93     { /* jupiter */
94         4.84143144246472090e+00,
95         -1.16032004402742839e+00,
96         -1.03622044471123109e-01,
97         1.66007664274403694e-03 * days_per_year,
98         7.69901118419740425e-03 * days_per_year,
99         -6.90460016972063023e-05 * days_per_year,
100        9.54791938424326609e-04 * solar_mass
101     },
102     { /* saturn */
103         8.34336671824457987e+00,
104         4.12479856412430479e+00,
105         -4.03523417114321381e-01,
106         -2.76742510726862411e-03 * days_per_year,
107         4.99852801234917238e-03 * days_per_year,
108         2.30417297573763929e-05 * days_per_year,
109         2.85885980666130812e-04 * solar_mass
110     },
111     { /* uranus */
112         1.28943695621391310e+01,
113         -1.51111514016986312e+01,
```

Appendix A. Benchmark Details

```
114     -2.23307578892655734e-01,  
115     2.96460137564761618e-03 * days_per_year,  
116     2.37847173959480950e-03 * days_per_year,  
117     -2.96589568540237556e-05 * days_per_year,  
118     4.36624404335156298e-05 * solar_mass  
119 },  
120 {                                     /* neptune */  
121     1.53796971148509165e+01,  
122     -2.59193146099879641e+01,  
123     1.79258772950371181e-01,  
124     2.68067772490389322e-03 * days_per_year,  
125     1.62824170038242295e-03 * days_per_year,  
126     -9.51592254519715870e-05 * days_per_year,  
127     5.15138902046611451e-05 * solar_mass  
128 }  
129 };  
130  
131 int main(int argc, char ** argv)  
132 {  
133     int {{n_init}};  
134     float {{dt_init}};  
135     int i;  
136  
137     offset_momentum(NBODIES, bodies);  
138     printf (".9f\n", energy(NBODIES, bodies));  
139     for (i = 1; i <= n; i++)  
140         advance(NBODIES, bodies, dt);  
141     printf (".9f\n", energy(NBODIES, bodies));  
142     return 0;  
143 }
```

A.4 heat

A.4.1 Python Original

```
1 #####  
2 # Heat equation finite difference  
3 # visualize with Qt4
```


Appendix A. Benchmark Details

```
4 # AP Ruymgaart 4/14/2015
5 #####
6 import time, sys, os, math
7 import random as rnd
8 from PyQt4 import QtGui, QtCore
9 from PySide.QtCore import *
10 import PySide
11 from PySide.QtGui import *
12 import numpy as np
13
14 #####
15 """
16  $d/dx [k(x) d/dx [u(x)]] = d k(x)/dx * d u(x)/dx + k(x) \text{Lapl } u$ 
17  $d/dx [f g] = f'g + f g'$ 
18 """
19 #####
20 #####
21 def AppendLog(logfile, sz):
22     f = open(logfile, "a+")
23     f.write(sz)
24     f.close()
25
26 #####
27 def Diffusion2D(M, K, x,y, bPeriodic, gridsz):
28
29     xSz = M.shape[1]
30     ySz = M.shape[0]
31
32     !-- boundary
33     left = x-1;
34     right = x+1;
35     up = y+1;
36     down = y-1;
37
38     fl = 1.0
39     fr = 1.0
40     fu = 1.0
41     fd = 1.0
42
43     if (bPeriodic):
44         !-- periodic boundary
45
46         !-- x
```

Appendix A. Benchmark Details

```
47     if (right >= xSz):
48         right -= xSz;
49     if (left < 0):
50         left += xSz;
51
52     #-- y
53     if (up >= ySz):
54         up -= ySz;
55     if (down < 0):
56         down += ySz;
57 else:
58     #-- Neumann (d2Tdx) )
59
60     #-- x
61     if (right >= xSz):
62         right = left;
63         #fl = fr = 0.5
64     if (left < 0):
65         left = right;
66         #fl = fr = 0.5
67
68     #-- y
69     if (up >= ySz):
70         up = down;
71         #fu = fd = 0.5
72     if (down < 0):
73         down = up;
74         #fu = fd = 0.5
75
76
77 gridsz2 = gridsz * gridsz
78
79 #-- diagonal of the Hessian:
80 d2Tdx = (fl*M[y,left] + fr*M[y,right] - 2.0*M[y,x]) / gridsz2
81 d2Tdy = (fu*M[up,x] + fd*M[down,x] - 2.0*M[y,x]) / gridsz2
82
83 #-- Laplacian (trace of the Hessian)
84 L = d2Tdx + d2Tdy
85
86 #-- add the term coming from the variable k (gradient dot k)
87 dot = 0.0
88
89 dKdx = (K[y,left] - K[y,x])/gridsz
```

Appendix A. Benchmark Details

```
90     dTdx = (M[y,left] - M[y,x])/gridsz
91
92     dKdy = (K[up,x] - K[y,x])/gridsz
93     dTdy = (M[up,x] - M[y,x])/gridsz
94
95     dot = dKdx * dTdx + dKdy * dTdy
96
97     du = K[y,x]*L + dot;
98
99     return du
100
101
102 #####
103 class Frame(QMainWindow):
104
105     #-----
106     def __init__(self, label):
107
108         super(Frame, self).__init__()
109         label.setMouseTracking(True)
110         QMainWindow.setCentralWidget(self, label)
111
112         self.xSize = 100
113         self.ySize = 100
114
115         self.xm = 1.95
116         self.ym = 0.3
117         self.gridSize = self.xm/self.xSize
118         self.gridArea = self.gridSize*self.gridSize
119         self.gridVolume = self.gridSize*self.gridSize*self.gridSize
120         self.sourceVolume = 0.0
121
122         self.U = np.zeros((1, 1))
123         self.Source = np.zeros((1, 1))
124         self.Sink = np.zeros((1, 1))
125         self.K = np.zeros((1, 1))
126
127         self.src = 1.0
128         self.sinktemp = -5.0
129
130         self.scale = 4
131
132         self.dt = 0.04
```

Appendix A. Benchmark Details

```
133     self.nsteps = 2000
134     self.paintsteps = 10
135     self.time = 0.0
136
137     self.border = 100
138
139     self.bmpK = ""
140     self.bmpSource = ""
141     self.bmpSink = ""
142     self.bBMP = False
143
144
145     self.uMax = 100.0
146     self.uMin = -100.0
147     self.uRange = self.uMax #- self.uMin
148     self.uTotal = 0.0
149
150     self.bPeriodic = True
151
152     self.mouseX = 0
153     self.mouseY = 0
154
155
156     #-----
157     def SetSizes(self, szx, szy):
158
159         self.xSize = szx
160         self.ySize = szy
161
162         self.setGeometry(40, 40, self.xSize*self.scale + self.border,
163                         self.ySize*self.scale + self.border)
164
165         self.U = np.zeros((self.ySize, self.xSize))
166         self.Source = np.zeros((self.ySize, self.xSize))
167         self.Sink = np.zeros((self.ySize, self.xSize))
168         self.K = np.zeros((self.ySize, self.xSize))
169
170         self.xm = float(self.xSize) * 0.003
171         self.ym = float(self.ySize) * 0.003
172
173         self.gridSize = self.xm/self.xSize
174         self.gridArea = self.gridSize*self.gridSize
175         self.gridVolume = self.gridSize*self.gridSize*self.gridSize
```

Appendix A. Benchmark Details

```
176
177
178     if abs(self.gridSize - self.ym/self.ySize) > 0.00001:
179         print "SIZE ERROR, grid not square", self.xm, self.ym, \
180             self.ym/self.ySize, self.gridSize
181         exit()
182
183     sz = "\
184     ##### SIZES #####\n\
185     x=%4.1f, y=%4.1f (m)\n\
186     x=%5d, y=%5d (tiles)\n\
187     grid cell size=%f (m)\n\
188     grid cell area=%f (m^2) grid cell volume=%10.9f (m^3)\n\
189     total volume=%f (m^3)\n\
190     " % (self.xm, self.ym, self.xSize, self.ySize, self.gridSize,
191         self.gridArea, self.gridVolume,
192         self.gridVolume*self.xSize*self.ySize)
193     print sz
194     AppendLog("simulation.log", sz)
195
196
197     sz = "\
198     ##### INTEGRATION #####\n\
199     number steps=%d\n\
200     timestep=%f (s)\n" % (self.nsteps, self.dt)
201     print sz
202     AppendLog("simulation.log", sz)
203
204
205     sz = "\
206     ##### START #####\n"
207     AppendLog("simulation.log", sz)
208
209
210     #-----
211     # NOTE: NEED + and - limits (max,min) to be symmetric
212     #-- zero = black
213     #-- below zero, blue (vary brightness)
214     #-- from zero up, transition from black to green to red
215     #--
216     def HeatColor(self, u):
217
218     intvl = [0, self.uMax]
```

Appendix A. Benchmark Details

```
219         intvl8bit = [0,255]
220
221         R = 0
222         G = 0
223         B = 0
224
225         t = abs(u)
226         v8bit = t * 10
227         if (v8bit > 255):
228             v8bit = 255
229
230         G = 255 - abs(v8bit)
231
232         if (u > 0):
233             R = v8bit
234             B = G
235         else:
236             B = v8bit
237             R = G
238
239
240         return QColor(R,G,B)
241
242     #-----
243     def GetHeat(self, n):
244
245         #--
246         self.uTotal = 0.0
247         self.uMax = -10000.0
248         self.uMin = -1.0 * self.uMax
249
250         for x in range(self.xSize):
251             for y in range(self.ySize):
252
253
254                 du = Diffusion2D(self.U, self.K, x,y, self.bPeriodic,
255                                 self.gridSize)
256                 self.uTotal += self.U[y,x]
257
258                 #-- timestep
259                 self.U[y,x] += du * self.dt
260
261                 #-- Sources and Sinks
```

Appendix A. Benchmark Details

```
262         self.U[y,x] += self.Source[y,x] * self.dt
263         self.U[y,x] -= self.Sink[y,x] * self.dt
264
265
266         if (self.U[y,x] > self.uMax):
267             self.uMax = self.U[y,x]
268         if (self.U[y,x] < self.uMin):
269             self.uMin = self.U[y,x]
270
271
272         self.time = self.dt * float(n)
273         sz = "%08d %12.11f %12.11f %4.1f %4.1f\n" % (n, self.time,
274                                                     self.uTotal,
275                                                     self.uMax,
276                                                     self.uMin)
277         print sz
278         AppendLog("simulation.log", sz)
279
280
281         #-----
282         def paintEvent(self, event):
283
284             painter = QPainter(self)
285             sz = "H=%10.1f time=%10.9f " % (self.uTotal, self.time)
286             painter.drawText(15,15,sz)
287             relX = self.mouseX - self.border/2
288             relY = self.mouseY - self.border/2
289             relX /= self.scale
290             relY /= self.scale
291             sz = "invalid point"
292             if ((relX > 0) and (relX < self.xSize)):
293                 if ((relY > 0) and (relY < self.ySize)):
294                     v = (relX, relY, self.U[relY][relX],
295                         self.K[relY][relX], self.Source[relY][relX])
296                     painter.drawText(250,15,sz)
297                     sz = "x=%d y=%d T=%9.8f k=%9.8f Src=%9.8f" % v
298
299             for x in range(self.xSize):
300                 for y in range(self.ySize):
301
302                     col = self.HeatColor(self.U[y,x])
303                     painter.fillRect(QRectF(x*self.scale+self.border/2,
304                                             y*self.scale+self.border/2,
```

Appendix A. Benchmark Details

```
305             self.scale, self.scale), col)
306
307     #-----
308     def run(self):
309
310         for n in range(self.nsteps):
311
312             self.GetHeat(n * self.paintsteps)
313             self.update()
314             QApplication.processEvents()
315             time.sleep(0.01)
316
317     #-----
318     def mousePressEvent(self, event):
319
320         self.mouseX = event.x()
321         self.mouseY = event.y()
322
323
324     #####
325     if __name__ == '__main__':
326         example = QApplication(sys.argv)
327         label = QLabel()
328         frm = Frame(label)
329         frm.setMouseTracking(True)
330
331         frm.show()
332         frm.raise_()
333
334     #---- VERY SIMPLE KEYWORD PARSER ----
335     inFile = open("simulation.txt","r");
336     data = inFile.readlines()
337     inFile.close()
338     for line in data:
339         elms = line.split()
340         if (len(elms)):
341             if (elms[0][0] != '#'):
342                 key = elms[0]
343                 if key == 'dt':
344                     frm.dt = float(elms[2])
345                     print "SETTING dt", frm.dt
346
347                 if key == 'steps':
```


Appendix A. Benchmark Details

```
348         frm.nsteps = int(elms[2])
349         print "SETTING steps", frm.nsteps
350
351     if key == 'paintsteps':
352         frm.paintsteps = int(elms[2])
353         print "SETTING paintsteps", frm.paintsteps
354
355     if key == 'src':
356         frm.src = float(elms[2])
357         print "SETTING src (K/s)", frm.src
358
359     if key == 'sinktemp':
360         frm.sinktemp = float(elms[2])
361         print "SETTING sinktemp ", frm.sinktemp
362
363
364     if key == 'scale':
365         frm.scale = int(elms[2])
366         print "SETTING scale", frm.scale
367
368
369     if key == 'grid':
370         frm.xSize = int(elms[2])
371         frm.ySize = int(elms[3])
372         frm.SetSizes(frm.xSize, frm.ySize)
373         for y in range(frm.ySize):
374             for x in range(frm.xSize):
375                 frm.K[y, x] = 1.0
376         print "SETTING grid", frm.xSize, frm.ySize
377
378
379     if key == 'source':
380         frm.Source[int(elms[2])][int(elms[1])] = \
381             float(elms[3])
382
383     if key == 'temp':
384         frm.U[int(elms[2])][int(elms[1])] = float(elms[3])
385
386     if key == 'alpha':
387         frm.K[int(elms[2])][int(elms[1])] = float(elms[3])
388
389
390     frm.run()
```

Appendix A. Benchmark Details

391 sys.exit(example.exec_())

A.4.2 Modified for Stella

```
1 # Copyright 2013-2015 AP Ruymgaart, David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
12 # implied. See the License for the specific language governing
13 # permissions and limitations under the License.
14 #####
15 # Heat equation finite difference
16 # visualize with Qt4
17 # AP Ruymgaart 4/14/2015
18 #####
19 gui = False
20
21 import time, sys, os, os.path
22 try:
23     from . import unimplemented
24 except SystemError:
25     def unimplemented(f):
26         return f
27 try:
28     from PySide.QtCore import *
29     from PySide.QtGui import *
30 except ImportError:
31     # fake stuff
32     gui = False
33
34     class QMainWindow(object):
35         pass
36
```

Appendix A. Benchmark Details

```
37 import numpy as np
38
39
40 #####
41 """
42  $d/dx [k(x) d/dx [u(x)]] = d k(x)/dx * d u(x)/dx + k(x) \text{Lapl } u$ 
43  $d/dx [f g] = f'g + f g'$ 
44
45 """
46 #####
47
48 log = os.path.join(os.path.dirname(__file__), 'simulation.log')
49
50 def AppendLog(logfile, sz):
51     f = open(logfile, "a+")
52     f.write(sz)
53     f.close()
54
55 #####
56 def WriteLog(sz):
57     f = open(log, "a+")
58     f.write(sz)
59     f.close()
60
61 #####
62 def Diffusion2D(M, K, x,y, bPeriodic, gridsz):
63
64     xSz = M.shape[1]
65     ySz = M.shape[0]
66
67     #-- boundary
68     left = x-1
69     right = x+1
70     up = y+1
71     down = y-1
72
73     fl = 1.0
74     fr = 1.0
75     fu = 1.0
76     fd = 1.0
77
78     if (bPeriodic):
79         #-- periodic boundary
```

Appendix A. Benchmark Details

```
80
81     -- x
82     if (right >= xSz):
83         right -= xSz
84     if (left < 0):
85         left += xSz
86
87     -- y
88     if (up >= ySz):
89         up -= ySz
90     if (down < 0):
91         down += ySz
92 else:
93     -- Neumann (d2Tdx) 
94
95     -- x
96     if (right >= xSz):
97         right = left
98         #fl = fr = 0.5
99     if (left < 0):
100        left = right
101        #fl = fr = 0.5
102
103     -- y
104     if (up >= ySz):
105         up = down
106         #fu = fd = 0.5
107     if (down < 0):
108         down = up
109         #fu = fd = 0.5
110
111
112 gridsz2 = gridsz * gridsz
113
114 -- diagonal of the Hessian:
115 d2Tdx = (fl*M[y,left] + fr*M[y,right] - 2.0*M[y,x]) / gridsz2
116 d2Tdy = (fu*M[up,x] + fd*M[down,x] - 2.0*M[y,x]) / gridsz2
117
118 -- Laplacian (trace of the Hessian)
119 L = d2Tdx + d2Tdy
120
121 -- add the term coming from the variable k (gradient dot k)
122 dot = 0.0
```

Appendix A. Benchmark Details

```
123
124     dKdx = (K[y,left] - K[y,x])/gridsz
125     dTdx = (M[y,left] - M[y,x])/gridsz
126
127     dKdy = (K[up,x] - K[y,x])/gridsz
128     dTdy = (M[up,x] - M[y,x])/gridsz
129
130     dot = dKdx * dTdx + dKdy * dTdy
131
132     du = K[y,x]*L + dot
133     #if y == 10 and x == 9:
134     #     print ("du={}".format(du))
135
136     return du
137
138
139 #####
140 class Frame(QMainWindow):
141     def __init__(self, label, sim):
142
143         super(Frame, self).__init__()
144         label.setMouseTracking(True)
145         QMainWindow.setCentralWidget(self, label)
146         self.sim = sim
147
148         self.mouseX = 0
149         self.mouseY = 0
150
151     def SetSizes(self, szx, szy):
152         self.setGeometry(40, 40,
153                         self.sim.xSize*self.sim.scale+self.sim.border,
154                         self.sim.ySize*self.sim.scale+self.sim.border)
155
156     #-----
157     def paintEvent(self, event):
158
159         painter = QPainter(self)
160         sz = "H=%10.1f time=%10.9f " % (self.sim.uTotal, self.sim.time)
161         painter.drawText(15,15,sz)
162         relX = self.mouseX - self.sim.border/2
163         relY = self.mouseY - self.sim.border/2
164         relX /= self.sim.scale
165         relY /= self.sim.scale
```

Appendix A. Benchmark Details

```
166     sz = "invalid point"
167     if ((relX > 0) and (relX < self.sim.xSize)):
168         if ((relY > 0) and (relY < self.sim.ySize)):
169             v = (relX, relY,
170                 self.sim.U[relY][relX],
171                 self.sim.K[relY][relX],
172                 self.sim.Source[relY][relX])
173             sz = "x=%d y=%d  T=%9.8f      k=%9.8f      Src=%9.8f" % v
174     painter.drawText(250,15,sz)
175
176     for x in range(self.sim.xSize):
177         for y in range(self.sim.ySize):
178
179             col = self.HeatColor(self.sim.U[y,x])
180             r = QRectF(x*self.sim.scale+self.sim.border/2,
181                       y*self.sim.scale+self.sim.border/2,
182                       self.sim.scale, self.sim.scale)
183             painter.fillRect(r, col)
184
185     #-----
186     # NOTE: NEED + and - limits (max,min) to be symmetric
187     #-- zero = black
188     #-- below zero, blue (vary brightness)
189     #-- from zero up, transition from black to green to red
190     #--
191     def HeatColor(self, u):
192
193         #intval = [0, self.uMax]
194         #intval8bit = [0,255]
195
196         R = 0
197         G = 0
198         B = 0
199
200         t = abs(u)
201         v8bit = t * 10
202         if (v8bit > 255):
203             v8bit = 255
204
205         G = 255 - abs(v8bit)
206
207         if (u > 0):
208             R = v8bit
```

Appendix A. Benchmark Details

```
209         B = G
210     else:
211         B = v8bit
212         R = G
213
214
215     return QColor(R,G,B)
216
217     #-----
218     def run(self):
219         """
220         HACK: instead of passing in a function to sim.run(), this is an
221         adoptation of sim.run() for the GUI.
222         """
223
224         for n in range(self.sim.nsteps):
225
226             self.sim.GetHeat(n * self.sim.paintsteps)
227             self.update()
228             QApplication.processEvents()
229             time.sleep(0.01)
230
231         print ("Done.")
232
233     def run_no_sleep(self):
234         """
235         HACK: instead of passing in a function to sim.run(), this is an
236         adoptation of sim.run() for the GUI.
237         """
238
239         for n in range(self.sim.nsteps):
240
241             self.sim.GetHeat(n * self.sim.paintsteps)
242             self.update()
243             QApplication.processEvents()
244
245         print ("Done.")
246
247     #-----
248     def mouseMoveEvent(self, event):
249
250         self.mouseX = event.x()
251         self.mouseY = event.y()
```

Appendix A. Benchmark Details

```
252
253
254 class Sim(object):
255     def __init__(self):
256         self.xSize = 100
257         self.ySize = 100
258
259         self.xm = 1.95
260         self.ym = 0.3
261         self.gridSize = self.xm/self.xSize
262         self.gridArea = self.gridSize*self.gridSize
263         self.gridVolume = self.gridSize*self.gridSize*self.gridSize
264         self.sourceVolume = 0.0
265
266         self.U = np.zeros((1, 1))
267         self.Source = np.zeros((1, 1))
268         self.Sink = np.zeros((1, 1))
269         self.K = np.zeros((1, 1))
270
271         self.src = 1.0
272         self.sinktemp = -5.0
273
274         self.scale = 4
275
276         self.dt = 0.04
277         self.set_nsteps(2000)
278         self.paintsteps = 10
279         self.time = 0.0
280
281         self.border = 100
282
283         self.uMax = 100.0
284         self.uMin = -100.0
285         self.uRange = self.uMax #- self.uMin
286         self.uTotal = 0.0
287
288         self.bPeriodic = True
289
290
291     def set_nsteps(self, nsteps):
292         self.nsteps = nsteps
293         self.observations = np.zeros((nsteps, 5))
294
```


Appendix A. Benchmark Details

```
295 #-----
296 def SetSizes(self, szx, szy, p=False):
297
298     self.xSize = szx
299     self.ySize = szy
300
301     self.U = np.zeros((self.ySize, self.xSize))
302     self.Source = np.zeros((self.ySize, self.xSize))
303     self.Sink = np.zeros((self.ySize, self.xSize))
304     self.K = np.zeros((self.ySize, self.xSize))
305
306     self.xm = float(self.xSize) * 0.003
307     self.ym = float(self.ySize) * 0.003
308
309     self.gridSize = self.xm/self.xSize
310     self.gridArea = self.gridSize*self.gridSize
311     self.gridVolume = self.gridSize*self.gridSize*self.gridSize
312
313
314     if abs(self.gridSize - self.ym/self.ySize) > 0.00001:
315         print ("SIZE ERROR, grid not square", self.xm, self.ym,
316               self.ym/self.ySize, self.gridSize)
317         exit()
318
319     sz = """
320 ##### SIZES #####
321 x=%4.1f, y=%4.1f (m)
322 x=%5d, y=%5d (tiles)
323 grid cell size=%f (m)
324 grid cell area=%f (m^2) grid cell volume=%10.9f (m^3)
325 total volume=%f (m^3)
326 """ % (self.xm, self.ym, self.xSize, self.ySize, self.gridSize,
327         self.gridArea, self.gridVolume,
328         self.gridVolume*self.xSize*self.ySize)
329     if p:
330         print (sz)
331     WriteLog(sz)
332
333
334     sz = """
335 ##### INTEGRATION #####
336 number steps=%d
337 timestep=%f (s)
```

Appendix A. Benchmark Details

```
338 """ % (self.nsteps, self.dt)
339     if p:
340         print (sz)
341         WriteLog(sz)
342
343
344     sz = """
345 ##### START #####
346 """
347     WriteLog(sz)
348
349
350 #-----
351 def GetHeat(self, n):
352
353     #--
354     self.uTotal = 0.0
355     self.uMax = -10000.0
356     self.uMin = -1.0 * self.uMax
357
358     for x in range(self.xSize):
359         for y in range(self.ySize):
360
361             du = Diffusion2D(self.U, self.K, x,y, self.bPeriodic,
362                             self.gridSize)
363             self.uTotal += self.U[y,x]
364
365             #-- timestep
366             self.U[y,x] += du * self.dt
367
368             #-- Sources and Sinks
369             self.U[y,x] += self.Source[y,x] * self.dt
370             self.U[y,x] -= self.Sink[y,x] * self.dt
371
372
373             if (self.U[y,x] > self.uMax):
374                 self.uMax = self.U[y,x]
375             if (self.U[y,x] < self.uMin):
376                 self.uMin = self.U[y,x]
377
378     self.time = self.dt * float(n)
379
380
```

Appendix A. Benchmark Details

```
381 #-----
382 def run(self):
383
384     for n in range(self.nsteps):
385
386         self.GetHeat(n * self.paintsteps)
387         self.observations[n, 0] = n
388         self.observations[n, 1] = self.time
389         self.observations[n, 2] = self.uTotal
390         self.observations[n, 3] = self.uMax
391         self.observations[n, 4] = self.uMin
392
393
394 def process_config(sim, fn="heat_settings.txt", p=False):
395     #---- VERY SIMPLE KEYWORD PARSER ----
396     inFile = open(os.path.join(os.path.dirname(__file__), fn), "r")
397     data = inFile.readlines()
398     inFile.close()
399     for line in data:
400         elms = line.split()
401         if (len(elms)):
402             if (elms[0][0] != '#'):
403                 key = elms[0]
404                 if key == 'dt':
405                     sim.dt = float(elms[2])
406                     if p:
407                         print ("SETTING dt", sim.dt)
408
409                 if key == 'steps':
410                     sim.set_nsteps(int(elms[2]))
411                     if p:
412                         print ("SETTING steps", sim.nsteps)
413
414                 if key == 'paintsteps':
415                     sim.paintsteps = int(elms[2])
416                     if p:
417                         print ("SETTING paintsteps", sim.paintsteps)
418
419                 if key == 'src':
420                     sim.src = float(elms[2])
421                     if p:
422                         print ("SETTING src (K/s)", sim.src)
423
```

Appendix A. Benchmark Details

```
424         if key == 'sinktemp':
425             sim.sinktemp = float(elms[2])
426             if p:
427                 print ("SETTING sinktemp ", sim.sinktemp)
428
429
430         if key == 'scale':
431             sim.scale = int(elms[2])
432             if p:
433                 print ("SETTING scale", sim.scale)
434
435
436         if key == 'grid':
437             sim.xSize = int(elms[2])
438             sim.ySize = int(elms[3])
439             sim.SetSizes(sim.xSize, sim.ySize, p)
440             for y in range(sim.ySize):
441                 for x in range(sim.xSize):
442                     sim.K[y, x] = 1.0
443             if p:
444                 print ("SETTING grid", sim.xSize, sim.ySize)
445
446
447         f_elms3 = float(elms[3])
448         if key == 'source':
449             sim.Source[int(elms[2])][int(elms[1])] = f_elms3
450
451         if key == 'temp':
452             sim.U[int(elms[2])][int(elms[1])] = f_elms3
453
454         if key == 'alpha':
455             sim.K[int(elms[2])][int(elms[1])] = f_elms3
456
457
458     def format_result(sim):
459         for n, time_, uTotal, uMax, uMin in sim.observations:
460             print ("%08d %12.11f %12.11f %4.1f %4.1f" % (n, time_, uTotal,
461                                                         uMax, uMin))
462
463     #####
464     if __name__ == '__main__':
465         sim = Sim()
466         if gui:
```

Appendix A. Benchmark Details

```
467     example = QApplication(sys.argv)
468     label = QLabel()
469     frm = Frame(label, sim)
470     frm.setMouseTracking(True)
471
472     frm.show()
473     frm.raise_()
474
475     process_config(sim, p=False)
476
477     if gui:
478         frm.SetSizes(sim.xSize, sim.ySize)
479         frm.run_no_sleep()
480         sys.exit(example.exec_())
481     else:
482         sim.run()
483         format_result(sim)
```

A.4.3 Change Summary

The combined simulation and GUI class

```
1 class Frame(QMainWindow):
2
3     #-----
4     def __init__(self, label):
5
6         super(Frame, self).__init__()
7         label.setMouseTracking(True)
8         QMainWindow.setCentralWidget(self, label)
9
10        self.xSize = 100
11        self.ySize = 100
12
13        self.xm = 1.95
14        self.ym = 0.3
15
16        #...
```

Appendix A. Benchmark Details

gets separated into two classes:

```
1 class Frame(QMainWindow):
2     def __init__(self, label, sim):
3
4         super(Frame, self).__init__()
5         label.setMouseTracking(True)
6         QMainWindow.setCentralWidget(self, label)
7         self.sim = sim
8     #...
9
10 class Sim(object):
11     def __init__(self):
12         self.xSize = 100
13         self.ySize = 100
14
15         self.xm = 1.95
16         self.ym = 0.3
17     #...
```

The GUI class retains a reference to the simulation object. Only the GUI-specific code, e.g., `setSize()`, stays in `Frame`, and the GUI now makes calls into the simulation code.

A.4.4 C Source

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <math.h>
5 #include <string.h>
6
7 #define True 1
8 #define False 0
9 #define bool uint8_t
10
11 #define sfs_get(s,x,y) \
12     ((s)[(x)*dim_x + (y)])
```

Appendix A. Benchmark Details

```
13
14 double* idx(double *a, int x, int y, int sx, int sy) {
15     //return a+(y*sy+x);
16     return &a[x*sy+y];
17 }
18 double get(double *a, int x, int y, int sx, int sy) {
19     return *idx(a, x, y, sx, sy);
20 }
21 double set(double *a, int x, int y, int sx, int sy, double v) {
22     return (*idx(a, x, y, sx, sy) = v);
23 }
24
25 typedef struct {
26     int xSize;
27     int ySize;
28
29     double xm;
30     double ym;
31     double gridSize;
32     double gridArea;
33     double gridVolume;
34     //int sourceVolume;
35
36     double *U;
37     double *Source;
38     double *Sink;
39     double *K;
40
41     double src;
42     double sinktemp;
43
44     int scale;
45
46     double dt;
47     int paintsteps;
48     double time;
49
50     int nsteps;
51     double *observations;
52
53     int border;
54
55     double uMax;
```

Appendix A. Benchmark Details

```
56     double uMin;
57     double uRange;
58     double uTotal;
59
60     bool bPeriodic;
61 } tSim;
62
63 double * np_array(int szx, int szy, double init)
64 {
65     double *r;
66     int sz;
67
68     sz = szx * szy * sizeof(double);
69     r = (double *) malloc(sz);
70     if (init == 0)
71         memset(r, '\0', sz);
72     else for (int i=0; i<szx; i++)
73         for (int j=0; j<szy; j++)
74             set(r, i, j, szx, szy, init);
75     return r;
76 }
77 #define np_zeros(szx, szy) np_array(szx, szy, 0)
78 #define np_ones(szx, szy) np_array(szx, szy, 1)
79
80 /* **-*-* */
81
82 double Diffusion2D(double *M, double *K, int x, int y,
83                   int xSz, int ySz, bool bPeriodic, double gridsz)
84 {
85     //-- boundary
86     int left = x-1;
87     int right = x+1;
88     int up = y+1;
89     int down = y-1;
90
91     double fl = 1.0;
92     double fr = 1.0;
93     double fu = 1.0;
94     double fd = 1.0;
95
96     if (bPeriodic) {
97         //-- periodic boundary
98
```


Appendix A. Benchmark Details

```
99     //-- x
100    if (right >= xSz)
101        right -= xSz;
102    if (left < 0)
103        left += xSz;
104
105    //-- y
106    if (up >= ySz)
107        up -= ySz;
108    if (down < 0)
109        down += ySz;
110 } else {
111     //-- Neumann (d2Tdx)
112
113     //-- x
114     if (right >= xSz)
115         right = left;
116         //fl = fr = 0.5
117     if (left < 0)
118         left = right;
119         //fl = fr = 0.5
120
121     //-- y
122     if (up >= ySz)
123         up = down;
124         //fu = fd = 0.5
125     if (down < 0)
126         down = up;
127         //fu = fd = 0.5
128 }
129
130 double gridsz2 = gridsz * gridsz;
131
132 //-- diagonal of the Hessian:
133 double d2Tdx = (fl*get(M, y,left, xSz, ySz)
134                + fr*get(M, y,right, xSz, ySz)
135                - 2.0*get(M, y,x, xSz, ySz)) / gridsz2;
136 double d2Tdy = (fu*get(M, up,x, xSz, ySz)
137                + fd*get(M, down,x, xSz, ySz)
138                - 2.0*get(M, y,x, xSz, ySz)) / gridsz2;
139
140 //-- Laplacian (trace of the Hessian)
141 double L = d2Tdx + d2Tdy;
```

Appendix A. Benchmark Details

```
142
143     //-- add the term coming from the variable k (gradient dot k)
144     double dot = 0.0;
145
146     double dKdx = (get(K, y,left, xSz, ySz) - get(K, y,x, xSz, ySz))
147                   /gridsz;
148     double dTdx = (get(M, y,left, xSz, ySz) - get(M, y,x, xSz, ySz))
149                   /gridsz;
150
151     double dKdy = (get(K, up,x, xSz, ySz) - get(K, y,x, xSz, ySz))
152                   /gridsz;
153     double dTdy = (get(M, up,x, xSz, ySz) - get(M, y,x, xSz, ySz))
154                   /gridsz;
155
156     dot = dKdx * dTdx + dKdy * dTdy;
157
158     double du = get(K, y,x, xSz, ySz)*L + dot;
159
160     /*
161     if (y == 10 && x == 9) {
162         printf ("du=%f\n", du);
163     }
164     */
165
166     return du;
167 }
168
169 void set_nsteps(tSim *self, int nsteps)
170 {
171     self->nsteps = nsteps;
172     self->observations = np_zeros(nsteps, 5);
173 }
174
175 void SetSizes(tSim *self, int szx, int szy)
176 {
177     self->xSize = szx;
178     self->ySize = szy;
179
180     self->U = np_zeros(self->ySize, self->xSize);
181     self->Source = np_zeros(self->ySize, self->xSize);
182     self->Sink = np_zeros(self->ySize, self->xSize);
183     self->K = np_ones(self->ySize, self->xSize);
184
```

Appendix A. Benchmark Details

```
185 self->xm = ((float)self->xSize) * 0.003;
186 self->ym = ((float)self->ySize) * 0.003;
187
188 self->gridSize = self->xm/self->xSize;
189 self->gridArea = self->gridSize*self->gridSize;
190 self->gridVolume = self->gridSize*self->gridSize*self->gridSize;
191
192 if (fabs(self->gridSize - self->ym/self->ySize) > 0.00001) {
193     printf ("SIZE ERROR, grid not square %f %f %f %f\n", self->xm,
194           self->ym, self->ym/self->ySize, self->gridSize);
195     exit(1);
196 }
197 }
198
199 void __init__(tSim *self)
200 {
201     SetSizes(self, 100, 100);
202
203     /*
204     self->sourceVolume = 0.0;
205
206     self->U = np.zeros((1, 1));
207     self->Source = np.zeros((1, 1));
208     self->Sink = np.zeros((1, 1));
209     self->K = np.zeros((1, 1));
210     */
211     int {{nsteps_init}};
212
213     self->dt = 0.000000002; // 0.04;
214     set_nsteps(self, nsteps);
215     self->paintsteps = 1000; // 10;
216     self->src = 0.4255; // 1.0;
217     self->sinktemp = -15; // -5.0;
218     self->scale = 4;
219
220     /* note that the original source uses the order 'y, x' */
221     set(self->Source, 10, 10, self->xSize, self->ySize, 10000000.0);
222     set(self->Source, 40, 10, self->xSize, self->ySize, -10000000.0);
223     set(self->Source, 26, 65, self->xSize, self->ySize, 10000000.0);
224     set(self->Source, 36, 80, self->xSize, self->ySize, -10000000.0);
225     // temp
226     set(self->U, 10, 10, self->xSize, self->ySize, 50);
227     set(self->U, 40, 40, self->xSize, self->ySize, -50);
```

Appendix A. Benchmark Details

```
228 // alpha
229 set(self->K, 10, 10, self->xSize, self->ySize, 30);
230 set(self->K, 10, 11, self->xSize, self->ySize, 30);
231 set(self->K, 10, 12, self->xSize, self->ySize, 30);
232 set(self->K, 11, 10, self->xSize, self->ySize, 30);
233 set(self->K, 11, 11, self->xSize, self->ySize, 30);
234 set(self->K, 11, 12, self->xSize, self->ySize, 30);
235 set(self->K, 12, 10, self->xSize, self->ySize, 30);
236 set(self->K, 12, 11, self->xSize, self->ySize, 30);
237 set(self->K, 12, 12, self->xSize, self->ySize, 30);
238
239 self->time = 0.0;
240 self->border = 100;
241 self->uMax = 100.0;
242 self->uMin = -100.0;
243 self->uRange = self->uMax; //- self->uMin;
244 self->uTotal = 0.0;
245 self->bPeriodic = True;
246 }
247
248 void GetHeat(tSim *self, int n)
249 {
250 //--
251 self->uTotal = 0.0;
252 self->uMax = -10000.0;
253 self->uMin = -1.0 * self->uMax;
254
255 for (int x=0; x<self->xSize; x++) {
256     for (int y=0; y<self->ySize; y++) {
257
258         double du = Diffusion2D(self->U, self->K, x,y, self->xSize,
259                                 self->ySize, self->bPeriodic, self->gridSize);
260         self->uTotal += get(self->U, y,x,
261                             self->xSize, self->ySize);
262
263         //-- timestep
264         *idx(self->U, y,x, self->xSize, self->ySize) \
265             += du * self->dt;
266
267         //-- Sources and Sinks
268         *idx(self->U, y,x, self->xSize, self->ySize) \
269             += get(self->Source, y,x, self->xSize, self->ySize) \
270                 * self->dt;
```

Appendix A. Benchmark Details

```
271         *idx(self->U, y,x, self->xSize, self->ySize) \
272         -= get(self->Sink, y,x, self->xSize, self->ySize) \
273         * self->dt;
274
275         double Uyx = get(self->U, y,x, self->xSize, self->ySize);
276         if (Uyx > self->uMax)
277             self->uMax = Uyx;
278         if (Uyx < self->uMin)
279             self->uMin = Uyx;
280     }
281 }
282
283 self->time = self->dt * (float)n;
284 }
285
286 void run(tSim *self)
287 {
288     for (int n=0; n<self->nsteps; n++) {
289
290         GetHeat(self, n * self->paintsteps);
291         set(self->observations, n, 0, self->nsteps, 5, n);
292         set(self->observations, n, 1, self->nsteps, 5, self->time);
293         set(self->observations, n, 2, self->nsteps, 5, self->uTotal);
294         set(self->observations, n, 3, self->nsteps, 5, self->uMax);
295         set(self->observations, n, 4, self->nsteps, 5, self->uMin);
296     }
297 }
298
299 int main(int argc, char **argv)
300 {
301     tSim sim;
302
303     __init__(&sim);
304
305     run(&sim);
306
307     /* report result */
308     tSim *self = &sim;
309     for (int i=0; i<self->nsteps; i++) {
310         double n = get(sim.observations, i, 0, self->nsteps, 5);
311         double time = get(sim.observations, i, 1, self->nsteps, 5);
312         double uTotal = get(sim.observations, i, 2, self->nsteps, 5);
313         double uMax = get(sim.observations, i, 3, self->nsteps, 5);
```

Appendix A. Benchmark Details

Benchmark	Compiler	Avg. run-time (s)	Std. Error	Slow-down
fib	gcc	14.208	0.017	-
	stella	16.948	0.010	1.19
	clang	29.515	0.033	2.08
1D-spider	clang	97.140	0.252	-
	gcc	100.253	0.372	1.03
	stella	111.123	0.696	1.14
nbody	stella	16.145	0.038	-
	gcc	16.611	0.057	1.03
	clang	25.263	0.594	1.56
heat	gcc	12.677	0.024	-
	clang	13.087	0.024	1.03
	stella	15.880	0.014	1.25

Table A.1: The run-time of the benchmarks, smaller is better. The table is sorted by average run-time (of 10 runs), so the fastest is listed first, and the relative slow-down is always in reference to the fastest.

```
314     double uMin = get(sim.observations, i, 4, self->nsteps, 5);
315     printf("%08.0f %12.11f %12.11f %4.1f %4.1f\n", n, time,
316           uTotal, uMax, uMin);
317 }
318
319 return 0;
320 }
```

A.5 Performance per C Compiler

Table A.1 shows the performance of each individual C compiler. This is the data that was used to compute the average performance shown in Table 5.1. The disparity between the gcc and clang is surprising, but not the focus of this dissertation.

Miscellaneous Source Code

Some source code of interest does not fall in the other categories and is listed here.

B.1 Exploratory Program Details

The benchmark “GenericSpiderSim” presented in Table 1.1 is a KMC simulation of continuous-time Markov process models of molecular spiders [44]. A spider is placed at the origin of a surface, and is then simulated walking over it.

The *generic* variant written in Python generalizes the code to handle arbitrary dimensions; a spider can have arbitrarily many legs; and the code can handle multiple spiders. This structure is representative of a source code the scientist would ideally work with: it is generic, reusable, but still needs to run fast. The complete source code is listed in Appendix B.1.1. The *NumPy* implementation uses NumPy arrays instead of Python lists. *NumPy* was used since it adds support for multidimensional arrays, and generally provides a great speed-up for numerical computations. Minor adjustments were necessary to run the generic Python version in the *Jython* and *PyPy* run-times (see Section 1.4), but there were no significant changes to the simulation.

Appendix B. Miscellaneous Source Code

C 2D optimized is production code, from Ref.[44], which was specialized by hand; it is limited to 2D using 1 spider with 2 legs. For comparison purposes we translated the generic Python version to C and C++, as well as optimized the Python version for 2D. The *generic C* version shows the potential speed-up of hand optimization, while the C++ version compares native object-oriented programming support against C code.

B.1.1 GenericSpiderSim

This is the simulation which was used for the initial evaluation of my work. Variants of this source code were used for Table 1.1.

```
1 #!/usr/bin/env python3
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 # http://www.apache.org/licenses/LICENSE-2.0
9 #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 import mtpy # cython wrapper around mtwist
17 from math import log
18 import sys
19 from copy import deepcopy
20 from numpy import zeros, copy
21 try:
22     from .virtnet_utils import Settings
23 except ValueError:
24     from test.virtnet_utils import Settings
25
26
```


Appendix B. Miscellaneous Source Code

```
27 ##### HELPERS #####
28
29 class Rnd(object):
30     @staticmethod
31     def seed(s):
32         #random.seed(s)
33         mtpy.mt_seed32new(s)
34
35     @staticmethod
36     def uniform():
37         #return random.random()
38         return mtpy.mt_drand()
39
40     @staticmethod
41     def exp(p):
42         u = 1.0 - mtpy.mt_drand()
43         return -log(u)/p
44
45
46 class SimObj(object):
47     @classmethod
48     def make(klass, n, initializer):
49         r = []
50         for i in range(n):
51             r.append(klass(next(initializer)))
52         return r
53
54     @classmethod
55     def pick(klass, l):
56         prob = Rnd.uniform()
57         return l[int(prob*len(l))]
58
59
60 class Point(object):
61     def __init__(self, pos = None):
62         if pos == None:
63             self.pos = zeros(shape=Point.dim)
64         elif type(pos) == Point:
65             self.pos = copy(pos.pos)
66         else:
67             self.pos = pos
68
69     def getPos(self):
```

Appendix B. Miscellaneous Source Code

```
70     return self.pos
71
72     def reldist(self, pos2):
73         p1 = self.pos
74         p2 = pos2.getPos()
75         rel = [x-y for x,y in zip(p1, p2)]
76         return pow(sum([pow(x,Point.dim) for x in rel]), 1.0/Point.dim)
77
78     def add(self, p2):
79         self.pos += p2
80     def addPos(self, p2):
81         pos2 = p2.getPos()
82         for i in range(len(self.pos)):
83             self.pos[i] += pos2[i]
84     def addToDim(self, dim, val):
85         self.pos[dim] += val
86
87     #def sub(self, p2):
88     #     self.pos -= p2
89
90     def div(self, p2):
91         #self.pos /= p2
92         self.pos = [x/p2 for x in self.pos]
93
94     #def mul(self, p2):
95     #     self.pos *= p2
96     def __repr__(self):
97         return self.pos.__repr__()
98
99
100 #### MAIN OBJECTS ####
101
102 class Leg(SimObj):
103     def __init__(self, params):
104         self.dim = params['dim']
105         self.pos = params['pos']
106         self.r = params['r']
107         self.no = params['no']
108
109         params['surface'].putDown(self.pos)
110
111     def __repr__(self):
112         return "Leg {0}".format(self.no)
```

Appendix B. Miscellaneous Source Code

```
113
114 def move(self, spider, surface):
115     # TODO gait check, extend possible moves?
116     moves = []
117     for d1 in range(surface.dim):
118         pos = Point(self.pos)
119         pos.addToDim(d1, -1)
120         if not surface.isOccupied(pos) and spider.gaitOk(pos, self):
121             moves.append(pos)
122
123         pos = Point(self.pos)
124         pos.addToDim(d1, 1)
125         if not surface.isOccupied(pos) and spider.gaitOk(pos, self):
126             moves.append(pos)
127     if len(moves) == 0:
128         raise Exception("No moves -- this shouldn't happen")
129     move = moves[int(Rnd.uniform() * len(moves))]
130
131     surface.pickUp(self.pos)
132     self.r = surface.putDown(move)
133     #print ("# Moving to {0}".format(move))
134
135     self.pos = move
136
137 def getRate(self):
138     return self.r
139 def getPosition(self):
140     return self.pos
141 def getDistance(self, pos):
142     return self.pos.reldist(pos)
143
144 @classmethod
145 def pick(klass, legs):
146     prob = Rnd.uniform()
147     R = 0.0
148     for leg in legs:
149         R += leg.getRate()
150     Ridx = R * prob
151     for leg in legs:
152         r = leg.getRate()
153         if Ridx < r:
154             #print ("# Using {0}".format(leg))
155             return leg
```

Appendix B. Miscellaneous Source Code

```
156         else:
157             Ridx -= r
158             raise Exception ("No leg was picked")
159
160
161 class Spider(SimObj):
162     def __init__(self, params):
163         def legInit():
164             pos = Point(Point.center)
165             i = 0
166             while True:
167                 yield {'r': params['r'], 'pos': Point(pos), 'dim':
168                     → params['dim'], 'surface': params['surface'], 'no': i}
169                 pos.addToDim(0,1)
170                 i += 1
171
172         self.legs = Leg.make(params['nlegs'], legInit())
173         self.nlegs = params['nlegs']
174         self.gait = params['gait']
175
176     def getLegs(self):
177         return self.legs
178
179     def getRate(self):
180         r = 0.0
181         for leg in self.legs:
182             r += leg.getRate()
183         return r
184
185     def getDistance(self):
186         pos = Point()
187         for leg in self.legs:
188             pos.addPos(leg.getPosition())
189         pos.div(self.nlegs)
190         return pos.reldist(Point.center)
191
192     def gaitOk(self, pos, leg):
193         otherlegs = [l for l in self.legs if l != leg]
194         #assert len(otherlegs) == self.params['nlegs'] - 1
195         for oleg in otherlegs:
196             if oleg.getDistance(pos) > self.gait:
197                 return False
198         return True
```

Appendix B. Miscellaneous Source Code

```
198
199     @classmethod
200     def pick(klass, spiders):
201         # FIXME: compatability with the optimized C version
202         #assert len(spiders) == 1
203         return spiders[0]
204
205
206 class Surface(object):
207     (substrate, product, occupied) = range(3)
208     def __init__(self, params):
209         self.dim = params['dim']
210         self.r = params['r']
211         self.koff = params['koff']
212         self.s = zeros(shape=[params['center']*2 for d in
213             ↪ range(self.dim)], dtype=int)
214
215     def __getitem__ (self, idx):
216         a = self.s
217         for i in range(len(idx)):
218             try:
219                 a = a[idx[i]]
220             except IndexError:
221                 raise Exception ("Index {0:s}[{1:d}] out of
222                 ↪ range".format(idx, i ))
223         return a
224
225     def __setitem__ (self, idx, value):
226         a = self.s
227         for i in range(len(idx)-1):
228             a = a[idx[i]]
229         a[idx[len(idx)-1]] = value
230
231     def isOccupied(self, idx):
232         idx = tuple(idx.getPos())
233         return self[idx] & self.occupied
234
235     def pickUp(self, idx):
236         idx = tuple(idx.getPos())
237         if not (self[idx] & self.product):
238             self[idx] += self.product
239         self[idx] -= self.occupied
240
```

Appendix B. Miscellaneous Source Code

```
239 def putDown(self, idx):
240     idx = tuple(idx.getPos())
241     try:
242         self[idx] += self.occupied
243     except IndexError:
244         # this shouldn't happen
245         print ("Error: {0} is out of range".format(idx))
246         raise
247     if self[idx] & self.product:
248         return self.koff
249     else:
250         return self.r
251
252
253
254 class Simulation(object):
255     def __init__(self, params):
256         Rnd.seed(params['seed'])
257
258         params['center'] = params['radius'] + 2
259
260         Point.dim = params['dim']
261         Point.center = Point([params['center'] for x in
262                               ↪ range(params['dim'])])
263
264         self.surface = Surface(params)
265     def spiderInit():
266         init_params = deepcopy(params)
267         init_params['surface'] = self.surface
268         while True:
269             yield init_params
270     self.spiders = Spider.make(params['nspiders'], spiderInit())
271     max_observations = 15 # pre-allocate space for observations, this
272                          ↪ is arbitrary and only limited by memory
273     self.observations = zeros(max_observations, dtype=float)
274     self.obs_i = 0
275     self.radius = params['radius']
276     self.t = 0.0 # added for Stella
277     self.nextObsDist = 1
278
279     def end(self):
280         return self.nextObsDist > self.radius or self.obs_i >=
281                ↪ len(self.observations)
```

Appendix B. Miscellaneous Source Code

```
279
280 def isNewObservation(self, spider):
281     return spider.getDistance() >= self.nextObsDist
282
283 def getHeader(self):
284     return "# sim_time"
285
286 def observe(self, spider):
287     #dist = spider.getDistance()
288     #print ("{t:.4f} {dist:.1f} {secs:.2f}".format(t=self.t,
289     ↪ dist=dist, secs=self.params['elapsedTime']()))
290     self.observations[self.obs_i] = self.t
291     self.obs_i += 1
292
293 def __eq__(self, o):
294     return (self.observations == o.observations).all()
295
296 def run(self):
297     # self.t = 0 # removed for Stella, widening of self.t not yet
298     ↪ supported
299     self.nextObsDist = 1
300     while not self.end():
301         spider = Spider.pick(self.spiders)
302         #print ("Moving spider {0}".format(spider))
303
304         self.t += Rnd.exp(spider.getRate())
305
306         leg = Leg.pick(spider.getLegs())
307         #print ("Moving leg {0}".format(leg))
308
309         leg.move(spider, self.surface)
310
311         if self.isNewObservation(spider):
312             self.observe(spider)
313             self.nextObsDist += 1
314
315 def verify_results():
316     """
317     This test assures that the simulation computes exactly the same result
318     ↪ as the C version.
319     """
```

Appendix B. Miscellaneous Source Code

```
319 seed = 1368048967
320 exp_results = [(9.041135791947408, 1.118033988749895),
  ↪ (19.667526963260286, 2.0615528128088303), (43.08767021796176,
  ↪ 3.0413812651491097), (119.12422328354563, 4.301162633521313),
  ↪ (235.72355926459574, 5.315072906367325), (244.49982281025584,
  ↪ 6.020797289396148), (252.74106914186763, 7.0710678118654755),
  ↪ (298.1731077872496, 8.06225774829855), (383.2934150062547,
  ↪ 9.013878188659973), (412.1063434596637, 10.012492197250394)]
321 exp_times = list(map(lambda x: x[0], exp_results))
322
323 settings = Settings()
324 settings['seed'] = seed
325
326 sim = Simulation(settings)
327 sim.run()
328
329 actual_results = sim.observations
330 for e_t, a_t in zip(exp_times, actual_results):
331     assert (e_t == a_t).all()
332
333
334 def main(argv):
335     settings = Settings(argv)
336     print("## {0}".format(settings))
337
338     sim = Simulation(settings)
339     sim.run()
340
341     print (sim.getHeader())
342     for t in sim.observations:
343         print ("{t:.4f}".format(t=t))
344
345     #print ([(x[0], x[1]) for x in results])
346
347
348 if __name__ == '__main__':
349     main(sys.argv[1:])
```

B.2 mtpy

The library *mtpy* is a simple interface to the highly optimized Mersenne Twister [35] implementation in C by Geoff Kuenning. This is Cython source code (*.pyx*), which is a language extension to Python.

```
1 #
2 # Copyright 2013-2015 David Mohr
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU Lesser General Public License as
6 #   → published by
7 # the Free Software Foundation, either version 3 of the License, or
8 # (at your option) any later version.
9 #
10 # This program is distributed in the hope that it will be useful,
11 # but WITHOUT ANY WARRANTY; without even the implied warranty of
12 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13 # GNU General Public License for more details.
14 #
15 # You should have received a copy of the GNU Lesser General Public
16 #   → License
17 # along with this program. If not, see <http://www.gnu.org/licenses/>.
18
19 cdef extern from "mtwist-1.1/mtwist.c":
20     double c_mt_drnd "mt_drnd" ()
21     void c_mt_seed32new "mt_seed32new" (unsigned int)
22
23 def getCSignatures():
24     """There should be a way to retrieve this info from cython, but I
25     → couldn't find it"""
26     import ctypes
27     return {
28         'mt_drnd': (ctypes.c_double, []),
29         'mt_seed32new': (None, [ctypes.c_uint32])
30     }
31
32 def mt_drnd():
33     return c_mt_drnd()
```

Appendix B. Miscellaneous Source Code

```
31
32 def mt_seed32new(s):
33     c_mt_seed32new (s)
```

B.3 Example Program SpiderSemiInfinite1D

The source code for the example discussed in Chapter 2.

```
1  #!/usr/bin/env python3
2  """
3  Semi-infinite 1D strip with a single spider.
4  """
5
6  import mtpy # cython wrapper around mtwist
7  from math import log, exp
8  import pdb
9  import sys
10 import time
11 from copy import deepcopy
12 import virtnet_utils
13 import stella
14 from numpy import zeros
15
16 class Settings(virtnet_utils.Settings):
17     def setDefaults(self):
18         self.settings = {
19             'seed'      : [int(time.time()), int],
20             'r'         : [0.1, float],
21             'koffp'     : [1.0, float],
22             'K'         : [10, int],
23             'rununtiltime' : [1e3, float],
24             'elapsedTime': [self.elapsedTime, lambda x:x],
25         }
26
27 class Simulation(object):
28     EXPSTART = 0.2
29     def __init__(self, params):
30         self.K = params['K']
```

Appendix B. Miscellaneous Source Code

```
31     self.rununtiltime = params['rununtiltime']
32     mtpy.seed(params['seed'])
33     self.koffp = params['koffp']
34     self.kcat = params['r']
35
36     self.delta =
37     ↪ (log(self.rununtiltime)-log(self.EXPSTART))/float(self.K-1)
38     self.leg = 0
39     self.substrate = 0
40     self.obs_i = 0
41     self.observations = zeros(shape=self.K, dtype=int)
42
43     def makeObservation(self):
44         """Called from run()"""
45         self.observations[self.obs_i] = self.leg
46         self.obs_i += 1
47
48         self.next_obs_time = self.getNextObsTime()
49
50     def getNextObsTime(self):
51         """Called from run()"""
52         if self.obs_i == 0:
53             return self.EXPSTART
54         if self.obs_i==self.K-1:
55             return self.rununtiltime;
56
57         return exp(log(self.EXPSTART)+self.delta*self.obs_i);
58
59     def step(self):
60         """Called from run()"""
61         if self.leg == 0:
62             self.leg += 1
63         else:
64             u1 = mtpy.uniform()
65             if u1 < 0.5:
66                 self.leg -= 1
67             else:
68                 self.leg += 1
69         if self.leg == self.substrate:
70             self.substrate += 1
71
72     def isNextObservation(self):
73         return self.t > self.next_obs_time and self.obs_i < self.K
```

Appendix B. Miscellaneous Source Code

```
73
74 @stella.wrap
75 def run(self):
76     self.t = 0.0;
77     self.next_obs_time = self.getNextObsTime();
78
79     while self.obs_i < self.K and self.t < self.rununtiltime:
80         if self.leg < self.substrate:
81             R = self.koffp
82         else:
83             R = self.kcat
84         self.t += mtpy.exp(R)
85
86         while self.isNextObservation():
87             self.makeObservation()
88
89         self.step()
90     return self.observations
91
92 def test():
93     settings = Settings()
94     settings['seed'] = 1368223681
95     expected = [0, 0, 1, 1, 3, 2, 7, 21, 32, 9]
96     actual = Simulation(settings).run()
97     # convert back to python list for easier comparison
98     assert(list(actual) == expected)
99
100 def main(argv):
101     settings = Settings(argv)
102     print("#", settings)
103     results = Simulation(settings).run()
104     print(results)
105
106 if __name__ == '__main__':
107     main(sys.argv[1:])
```

B.4 Example Helper Code

The source code for the library used by the example from Chapter 2.

Appendix B. Miscellaneous Source Code

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 from types import FunctionType
15 import time
16
17 class Settings(object):
18     def setDefaults(self):
19         self.settings = {
20             'seed'      : [int(time.time()), int],
21             'r'         : [0.1, float],
22             'koff'      : [1.0, float],
23             'radius'    : [10, int],
24             'nlegs'     : [2, int],
25             'gait'      : [2, int],
26             'dim'       : [2, int],
27             'nspiders'  : [1, int],      # not completely functional
28                                     → yet
29             'elapsedTime': [self.elapsedTime, lambda x:x],
30         }
31     def elapsedTime(self):
32         return time.time() - self.start_time
33
34     def __init__(self, argv = []):
35         self.start_time = time.time()
36
37         self.setDefaults()
38
39         if isinstance(argv, dict):
40             for k, v in argv.items():
41                 self[k] = v
42         else:
```

Appendix B. Miscellaneous Source Code

```
42         # parse command line arguments to overwrite the defaults
43         for key, _, val in [s.partition('=') for s in argv]:
44             self[key] = val
45
46     def __setitem__(self, k, v):
47         if k in self.settings:
48             self.settings[k][0] = self.settings[k][1](v)
49         else:
50             self.settings[k] = [v, type(v)]
51
52     def __getitem__(self, k):
53         return self.settings[k][0]
54
55     def __str__(self):
56         r = '{'
57         for k, (v, type_) in self.settings.items():
58             if isinstance(type_, FunctionType):
59                 continue
60             r += str(k) + ':' + str(v) + ', '
61         return r[:-2] + '}'
62
63     def dsl(f):
64         return f
```

B.5 Example Program SpiderSemiInfinite1D-Fpt

The source code for the object-oriented programming example discussed in Section 2.4. It extends the program listed in Section B.3.

```
1 #!/usr/bin/env python3
2 """
3 Semi-infinite 1D strip with a single spider measuring first passage time.
4 """
5
6 from sill1s_python import *
7
8 class SimulationFpt(Simulation):
```

Appendix B. Miscellaneous Source Code

```
9  def __init__(self, params):
10     Simulation.__init__(self, params)
11     self.observations = zeros(shape=self.K, dtype=float)
12
13  def getNextObsTime(self):
14     return 0.0
15
16  def isNextObservation(self):
17     return self.leg > self.obs_i
18
19  def makeObservation(self):
20     self.observations[self.obs_i] = self.t
21     self.obs_i += 1
22
23  def test():
24     from numpy import array
25     settings = Settings()
26     settings['seed'] = 1368637527
27     expected = array([ 6.47391565, 22.70745085, 35.85235243, 64.4477626,
28     ↪ 65.87662277,
29     ↪ 71.05396571, 76.79803361, 81.62789233, 84.24541638,
30     ↪ 87.40575155])
31
32     eps = 0.0001
33     actual = SimulationFpt(settings).run()
34     # inexact comparison for floating point numbers
35     assert((abs(actual-expected) < eps).all())
36
37  def main(argv):
38     settings = Settings(argv)
39     print("#", settings)
40     results = SimulationFpt(settings).run()
41     print(results)
42
43  if __name__ == '__main__':
44     main(sys.argv[1:])
```

STELLA Source Code

The complete and formatted source code of the STELLA implementation. It can also be easily downloaded or viewed online at [github](#). These sources represent revision 000f3bb5058c535bdf2bbb26e63af82381f84483.

C.1 stella/analysis.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import dis
15 import logging
16 import inspect
```


Appendix C. STELLA Source Code

```
17
18 from . import exc
19 from . import bytecode
20 from . import ir
21 from . import tp
22 from . import utils
23
24
25 class DebugInfo(object):
26     line = None
27     filename = None
28
29     def __init__(self, filename, line):
30         self.line = line
31         self.filename = filename
32
33     def __str__(self):
34         return self.filename + ':' + str(self.line)
35
36
37 class Function(object):
38     funcs = {}
39     analysis_count = 0
40
41     @classmethod
42     def clearCache(klass):
43         klass.funcs.clear()
44
45     @classmethod
46     def get(klass, f, module):
47         if isinstance(f, ir.FunctionRef):
48             impl = f.function
49         elif isinstance(f, ir.Function):
50             impl = f
51         else:
52             raise exc.TypeError("{} is not a Function, it has type
53             ↪ {}".format(f, type(f)))
54
55         logging.debug("Function.get({0}|{1}, {2})".format(
56             impl, id(impl), module))
57         try:
58             return klass.funcs[(impl, module)]
59         except KeyError:
```

Appendix C. STELLA Source Code

```
59         self = klass(impl, module)
60         klass.funcs[(impl, module)] = self
61         return self
62
63     def __init__(self, impl, module):
64         self.bytecodes = None # pointer to the first bytecode
65         self.labels = {}
66         self.incoming_jumps = {}
67
68         self.f = impl.pyFunc()
69         self.impl = impl
70         self.module = module
71
72         self.log = logging.getLogger(str(self))
73         self.todo = utils.Stack("Todo", log=self.log, quiet=True)
74         logging.info("Analyzing {0}".format(self))
75
76     def __str__(self):
77         return str(self.impl)
78
79     def __repr__(self):
80         return "{:~}>".format(super().__repr__()[:-1], self)
81
82     def retype(self, go=True):
83         """Immediately retype this function if go is True"""
84         if isinstance(go, tuple):
85             # extract the widening
86             go = go[0]
87         if go:
88             self.analyze_again = True
89
90     def add_incoming_jump(self, target_bc, source_bc):
91         if target_bc in self.incoming_jumps:
92             self.incoming_jumps[target_bc].append(source_bc)
93         else:
94             self.incoming_jumps[target_bc] = [source_bc]
95
96     def addLabel(self, bc):
97         """Remove replaceLocation() below?"""
98         self.replaceLocation(bc)
99
100    def replaceLocation(self, bc):
101        """Assume that bc.loc points to the new location already."""
```

Appendix C. STELLA Source Code

```
102     self.labels[bc.loc] = bc
103
104     def rewrite(self):
105         self.bytecodes.printAll(self.log)
106         self.log.debug("Rewriting (peephole optimizations) " + '-'*40)
107         for bc in self.bytecodes:
108             try:
109                 if isinstance(bc, bytecode.FOR_ITER):
110                     for_loop = bytecode.ForLoop(self, bc.debuginfo)
111                     for_loop.basicSetup(bc)
112                     for_loop.rewrite(self)
113                     self.replaceLocation(for_loop)
114             except exc.StellaException as e:
115                 e.addDebug(bc.debuginfo)
116                 raise
117
118         self.bytecodes.printAll(self.log)
119
120     def intraflow(self):
121         self.log.debug("Building Intra-Flowgraph " + '-'*40)
122         for bc in self.bytecodes:
123             try:
124                 if isinstance(bc, bytecode.Jump):
125                     if bc.processFallThrough():
126                         self.add_incoming_jump(bc.linearNext(), bc)
127                 if isinstance(bc, bytecode.HasTarget):
128                     target_bc = self.labels[bc.target_label]
129                     bc.setTargetBytecode(target_bc)
130                     self.add_incoming_jump(target_bc, bc)
131             except exc.StellaException as e:
132                 e.addDebug(bc.debuginfo)
133                 raise
134
135         for bc in self.bytecodes:
136             try:
137                 if bc in self.incoming_jumps:
138                     bc_prev = bc.linearPrev()
139                     # TODO Ugly -- blocks aren't transparent enough
140                     if isinstance(bc_prev, utils.Block):
141                         bc_prev = bc_prev.blockContent()
142                     if bc_prev and not isinstance(bc_prev,
143                                                    ↪ utils.BlockTerminal):
144                         bc_ = bytecode.Jump(self, bc.debuginfo)
```

Appendix C. STELLA Source Code

```
144         bc_.loc = ''
145         bc_.setTargetBytecode(bc)
146         bc_.setTarget(bc.loc) # for printing purposes
147         ↪ only
148         bc.insert_before(bc_)
149         self.add_incoming_jump(bc, bc_)
150
151         self.log.debug("IF ADD " + bc_.locStr())
152
153     if len(self.incoming_jumps[bc]) > 1:
154         bc_ = ir.PhiNode(self.impl, bc.debuginfo)
155         bc_.loc = bc.loc # for printing purposes only
156
157         bc.insert_before(bc_)
158
159         # Move jumps over to the PhiNode
160         if bc in self.incoming_jumps:
161             self.incoming_jumps[bc_] =
162             ↪ self.incoming_jumps[bc]
163             for bc__ in self.incoming_jumps[bc_]:
164                 bc__.setTargetBytecode(bc_)
165             del self.incoming_jumps[bc]
166
167         self.log.debug("IF ADD " + bc_.locStr())
168     except exc.StellaException as e:
169         e.addDebug(bc.debuginfo)
170         raise
171
172 def stack_to_register(self):
173     self.log.debug("Stack->Register Conversion " + '-'*40)
174     stack = utils.Stack(log=self.log)
175     self.todo.push((self.bytecodes, stack))
176     eevald = set()
177
178     # For the STORE_FAST of the argument(s)
179     for arg in reversed(self.impl.arg_transfer):
180         arg_bc = bytecode.ResultOnlyBytecode(self.impl, None)
181         arg_bc.result = self.impl.getRegister('__param_' + arg)
182         stack.push(arg_bc)
183
184     while not self.todo.empty():
185         (bc, stack) = self.todo.pop()
```

Appendix C. STELLA Source Code

```
185         if isinstance(bc, utils.Block):
186             bc = bc.blockContent()
187
188         r = bc.stack_eval(self.impl, stack)
189         eeval.add(bc)
190         if r is None:
191             # default case: no control flow diversion, just continue
192             #   → with
193             # the next instruction in the list
194             # Note: the 'and not' part is a basic form of dead code
195             # elimination. This is used to drop unreachable "return
196             #   → None"
197             # which are implicitly added by Python to the end of
198             #   → functions.
199             # TODO is this the proper way to handle those returns? Any
200             #   → side
201             # effects?
202             # NEEDS REVIEW See also codegen.Program.__init__
203             if bc.linearNext() and not isinstance(bc,
204             #   → utils.BlockTerminal):
205                 # the PhiNode swallows different control flow paths,
206                 # therefore do not evaluate beyond more than once
207                 if not (isinstance(bc, ir.PhiNode) and
208                 bc.linearNext() in eeval):
209                     self.todo.push((bc.linearNext(), stack))
210
211                 if isinstance(bc, utils.Block):
212                     # the next instruction after the block is now
213                     #   → already
214                     # on the todo list, but first lets work inside the
215                     #   → block
216                     self.todo.push((bc.blockContent(), stack))
217             else:
218                 if bc.linearNext() and not self.todo.contains(lambda
219                 #   → x: x[0] == bc.linearNext()):
220                     # Dead code. It used to be necessary to continue
221                     #   → here, but this is no longer
222                     # the case.
223                     self.log.debug("Next instruction is not directly
224                     #   → reachable, aborting")
225                 else:
226                     assert stack.empty()
227             else:
```

Appendix C. STELLA Source Code

```
218         # there is (one or more) control flow changes, add them
           ↪ all to
219         # the todo list
220         assert isinstance(r, list)
221         for (bc_, stack_) in r:
222             # don't go back to a bytecode that we already
           ↪ evaluated
223             # if there are no changes on the stack
224             if stack_.empty() and bc_ in evaled:
225                 continue
226             self.todo.push((bc_, stack_))
227
228     def type_analysis(self):
229         self.log.debug("Type Analysis " + '-'*40)
230         self.todo.push(self.bytecodes)
231
232         i = 0
233         reachable = True
234         while not self.todo.empty():
235             assert len(self.todo) == 1 # TODO could avoid todo queue here
236             self.log.debug("Type analysis iteration {0}".format(i))
237             self.analyze_again = False
238             bc_list = self.todo.pop()
239
240             for bc in bc_list:
241                 try:
242                     if reachable:
243                         abort = bc.type_eval(self)
244                         self.log.debug("TYPE'D " + bc.locStr())
245                         if isinstance(bc, bytecode.RETURN_VALUE):
246
247                             ↪ self.retype(self.impl.result.unify_type(bc.result.type,
248
                                                                           ↪ bc.debuginfo))
249
250                     else:
251                         self.log.debug("UNTYPE " + bc.locStr() + " --
252                             ↪ unreachable")
253                         bc.reachable = False
254
255                 if isinstance(bc, utils.BlockTerminal) and \
256                     bc.linearNext() is not None and \
257                     bc.linearNext() not in self.incoming_jumps:
```

Appendix C. STELLA Source Code

```
255         # Python does generate unreachable code which is
           ↪ then
256         # followed by reachable code. This allows us to
           ↪ jump
257         # over the section that needs to be ignored
258         self.log.debug("Unreachable {0}, but
           ↪ continuing".format(bc.linearNext()))
259         reachable = False
260     elif bc.linearNext() in self.incoming_jumps:
261         # Once there is an incoming jump, the following
           ↪ code is
262         # reachable again
263         reachable = True
264
265     if abort:
266         self.log.debug("Aborting typing, resuming later.")
267         self.log.debug("{!r}".format(self.todo))
268         self.impl.analyzeAgain()
269         break
270     except exc.StellaException as e:
271         e.addDebug(bc.debuginfo)
272         raise
273
274     if self.analyze_again:
275         self.todo.push(bc_list)
276
277     if i > 3:
278         raise Exception("Stopping after {0} type analysis
           ↪ iterations (failsafe)".format(i))
279     i += 1
280
281     self.log.debug("returning type " + str(self.impl.result.type))
282
283     def analyzeCall(self, args, kwargs):
284         self.log.debug("analysis.Function id " + str(id(self)))
285         if not self.impl.analyzed:
286             self.impl.setupArgs(args, kwargs)
287
288             self.log.debug("Analysis of " + self.impl.nameAndType())
289
290             self.disassemble()
291
292             self.rewrite()
```

Appendix C. STELLA Source Code

```
293
294     self.intraflow()
295
296     self.bytecodes.printAll(self.log)
297
298     self.stack_to_register()
299
300     self.type_analysis()
301
302     self.impl.bytecodes = self.bytecodes
303     self.impl.incoming_jumps = self.incoming_jumps
304 else:
305     self.log.debug("Re-typing " + self.impl.nameAndType())
306
307     self.type_analysis()
308
309 def disassemble(self):
310     """Disassemble a code object."""
311     self.log.debug("Disassembling -----")
312
313     self.last_bc = None
314
315     # Store arguments in memory locations for uniformity
316     for arg in self.impl.arg_transfer:
317         # TODO Patch up di?
318         di = None
319         bc = bytecode.STORE_FAST(self.impl, di)
320         bc.addLocalName(self.impl, arg)
321         if self.last_bc is None:
322             self.bytecodes = self.last_bc = bc
323         else:
324             self.last_bc.insert_after(bc)
325             self.last_bc = bc
326         self.log.debug("DIS'D {0}".format(bc.locStr()))
327
328     co = self.f.__code__
329     code = co.co_code
330     labels = dis.findlabels(code)
331     linestarts = dict(dis.findlinestarts(co))
332     n = len(code)
333     i = 0
334     extended_arg = 0
335     free = None
```


Appendix C. STELLA Source Code

```
336     line = 0
337     self.blocks = utils.Stack('blocks')
338     while i < n:
339         op = code[i]
340         if i in linestarts:
341             line = linestarts[i]
342
343         di = DebugInfo(co.co_filename, line)
344
345         if extended_arg == 0 and op in bytecode.opconst:
346             bc = bytecode.opconst[op](self.impl, di)
347         else:
348             raise exc.UnsupportedOpcode(op, di)
349         bc.loc = i
350
351         if i in labels:
352             self.labels[i] = bc
353
354         # print(repr(i).rjust(4), end=' ')
355         # print(dis.opname[op].ljust(20), end=' ')
356         i = i + 1
357         try:
358             if op >= dis.HAVE_ARGUMENT:
359                 oparg = code[i] + code[i + 1]*256 + extended_arg
360                 extended_arg = 0
361                 i = i+2
362                 if op == dis.EXTENDED_ARG:
363                     extended_arg = oparg*65536
364
365                 if op in dis.hasconst:
366                     # print('(' + repr(co.co_consts[oparg]) + ')',
367                       #     end=' ')
368                     bc.addConst(co.co_consts[oparg])
369                 elif op in dis.hasname:
370                     # print('(' + co.co_names[oparg] + ')', end=' ')
371                     bc.addName(self.impl, co.co_names[oparg])
372                 elif op in dis.hasjrel:
373                     # print('(to ' + repr(i + oparg) + ')', end=' ')
374                     bc.setTarget(i + oparg)
375                 elif op in dis.hasjabs:
376                     # print(repr(oparg).rjust(5), end=' ')
377                     bc.setTarget(oparg)
378                 elif op in dis.haslocal:
```

Appendix C. STELLA Source Code

```
378         # print('(' + co.co_varnames[oparg] + ')', end='
379         ↪ ')
380         bc.addLocalName(self.impl, co.co_varnames[oparg])
381     elif op in dis.hascompare:
382         # print('(' + dis.cmp_op[oparg] + ')', end=' ')
383         bc.addCmp(dis.cmp_op[oparg])
384     elif op in dis.hasfree:
385         if free is None:
386             free = co.co_cellvars + co.co_freevars
387             # print('(' + free[oparg] + ')', end=' ')
388             raise exc.UnimplementedError('hasfree')
389         else:
390             bc.addRawArg(oparg)
391
392     self.log.debug("DIS'D {0}".format(bc.locStr()))
393 except exc.StellaException as e:
394     e.addDebug(di)
395     raise
396
397 if isinstance(bc, utils.BlockStart):
398     # Start of a block.
399     # The current bc gets added as the first within the block
400     block = utils.Block(bc)
401     self.blocks.push(block)
402     # Then handle the block as any regular bytecode
403     # so that it will be registered appropriately
404     bc = block
405     # Note the instance(bc, Block) below
406
407 if self.last_bc is None:
408     self.bytecodes = bc
409 else:
410     self.last_bc.insert_after(bc)
411 self.last_bc = bc
412
413 if isinstance(bc, utils.Block):
414     # Block is inserted, now switch back to appending to the
415     ↪ block
416     # content
417     self.last_bc = bc.blockContent()
418 elif isinstance(bc, utils.BlockEnd):
419     # Block end, install the block itself as last_bc
420     # so that the next instruction is added outside the block
```

Appendix C. STELLA Source Code

```
419         self.last_bc = self.blocks.pop()
420         # mark the instruction as being the last of the block
421         bc.blockEnd(self.last_bc)
422
423
424 def cleanup():
425     logging.debug("Cleaning up...")
426     Function.clearCache()
427     tp.destruct()
428
429
430 def main(f, args, kwargs):
431     # Clean up first since an internal failure may have prevented the
432     # destructors from running.
433     cleanup()
434
435     try:
436         module = ir.Module()
437         f_type = tp.get(f)
438         funcref = module.getFunctionRef(f_type)
439
440         if f_type.bound:
441             f_self = tp.wrapValue(f.__self__)
442             funcref.f_self = f_self
443
444         # TODO: why do I use wrapValue for args but Const for kwargs...?
445         const_kw = {}
446         for k, v in kwargs.items():
447             const_kw[k] = tp.Const(v)
448         funcref.makeEntry(list(map(tp.wrapValue, args)), const_kw)
449
450         f = Function.get(funcref, module)
451
452         wrapped_args = [tp.wrapValue(arg) for arg in args]
453         wrapped_kwargs = {k: tp.wrapValue(v) for k, v in kwargs.items()}
454
455     except exc.StellaException as e:
456         # An error occurred while preparing the entry call, so at this
457         # ↪ point
458         # it's best to attribute it to the caller
459         (frame, filename, line_number,
460          function_name, lines, index) =
461         ↪ inspect.getouterframes(inspect.currentframe())[2]
```

Appendix C. STELLA Source Code

```
460         debuginfo = DebugInfo(filename, line_number)
461         e.addDebug(debuginfo)
462         raise e
463
464     f.analyzeCall(wrapped_args, wrapped_kwargs)
465
466     while module.todoCount() > 0:
467         f.log.debug("called functions: {} ({}).format(module.todoList(),
468             ↪ module.todoCount())
469         # TODO add kwargs support!
470         (call_impl, call_args, call_kwargs) = module.todoNext()
471         call_f = Function.get(call_impl, module)
472         # TODO this limit currently needs to be raised for larger
473         ↪ programs.
474         #     that is an indicator that it does not truly represent the
475         ↪ analysis count, and needs
476         #     fixing
477         if call_f.analysis_count > 30:
478             # TODO: arbitrary limit, it would be better to check if the
479             ↪ return
480             # type changed or not
481             raise Exception("Stopping {} after {} call analysis iterations
482                 ↪ (failsafe)".format(
483                 call_f, call_f.analysis_count))
484         call_f.analyzeCall(call_args, call_kwargs)
485         call_f.analysis_count += 1
486     module.addDestruct(cleanup)
487     return module
```

C.2 stella/tp.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
```

Appendix C. STELLA Source Code

```
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import llvmlite.ir as ll
15 import numpy as np
16 import ctypes
17 import logging
18 import types
19 from abc import ABCMeta, abstractmethod
20 import inspect
21 from functools import reduce
22 import operator
23
24 from . import exc
25
26
27 class Type(metaclass=ABCMeta):
28     type_ = None
29     _llvm = None
30     ptr = 0
31     on_heap = False
32     complex_on_stack = False
33     req_transfer = False
34     ctype = False # init to false because None is a valid ctype
35
36     def makePointer(self, ensure=False):
37         """Modifies this type to be a pointer"""
38         if self.on_heap:
39             if ensure:
40                 self.ptr = 1
41             else:
42                 assert self.ptr == 0
43                 self.ptr += 1
44
45     def isReference(self):
46         return self.ptr > 0
47
48     def isUntyped(self):
49         return False
50
51     def dereference(self):
```

Appendix C. STELLA Source Code

```
52     if self.ptr == 1:
53         # TODO hackish!
54         return self
55     else:
56         raise exc.TypeError("Cannot dereference the non-reference type
57             ↪ {}".format(self))
58
59 @abstractmethod
60 def __str__(self):
61     pass
62
63 def llvmType(self, module):
64     # some types just come as a reference (e.g. external numpy array).
65     ↪ For
66     # references within stella use class Reference.
67     assert self.ptr <= 1
68     type_ = self._llvmType(module)
69     if self.ptr:
70         return type_.as_pointer()
71     else:
72         return type_
73
74 def null(self, module):
75     return ll.Constant(self.llvmType(module), None)
76
77 def _llvmType(self, module):
78     raise exc.TypeError(
79         "Cannot create llvm type for an unknown type. This should have
80             ↪ been caught earlier.")
81
82 def Ctype(self):
83     if type(self.ctype) == bool:
84         raise exc.TypeError(
85             "Cannot create ctype for an unknown type. This should have
86             ↪ been caught earlier.")
87     else:
88         return self.ctype
89
90 def getElementType(self, idx):
91     return NoType
92
93 @classmethod
94 def unpack(klass, val):
```

Appendix C. STELLA Source Code

```
91         return val
92
93
94 class UnknownType(Type):
95     def isUntyped(self):
96         return False
97
98     def __str__(self):
99         return '?'
100
101     def dereference(self):
102         return self
103
104
105 NoType = UnknownType()
106
107
108 class Reference(Type):
109     def __init__(self, type_):
110         self.type_ = type_
111         self.ptr = type_.ptr + 1
112
113     def llvmType(self, module):
114         type_ = self.type_.llvmType(module)
115         # for i in range(self.ptr):
116         return type_.as_pointer()
117
118     def dereference(self):
119         return self.type_
120
121     def __str__(self):
122         return '*{}'.format(self.type_)
123
124     def __repr__(self):
125         return '{} of {}>'.format(super().__repr__()[:-1], self.type_)
126
127     @property
128     def on_heap(self):
129         return self.type_.on_heap
130
131     def __eq__(self, o):
132         return self.ptr == o.ptr and self.type_ == o.type_
133
```

Appendix C. STELLA Source Code

```
134     def __ne__(self, other):
135         return not self.__eq__(other)
136
137
138 class Subscriptable(metaclass=ABCMeta):
139     """Mixin"""
140     @abstractmethod
141     def loadSubscript(cge, container, idx):
142         pass
143
144     @abstractmethod
145     def storeSubscript(cge, container, idx, value):
146         pass
147
148
149 class PyWrapper(Type):
150     """Wrap Python types, e.g. for the intrinsic zeros dtype parameter.
151
152     This allows passing types as first-class values, but is used only in
153     special circumstances like zeros().
154     """
155     def __init__(self, py):
156         self.py = py
157         self.bc = None
158         self.type = get_scalar(py)
159
160     def __str__(self):
161         return str(self.py)
162
163     def _llvmType(self, module):
164         raise exc.TypeError("Cannot create an LLVM type for Python type
165         ↪ {}".format(self.py))
166
167 class ScalarType(Type):
168     def __init__(self, name, type_, llvm, ctype, cast_map):
169         """
170         cast_map: key is the type to map from, value is the builder
171         ↪ function
172         name to call to convert to the type this object represents.
173         """
174         self.name = name
175         self.type_ = type_
```


Appendix C. STELLA Source Code

```
175     self.ctype = ctype
176     self._llvm = llvm
177     self.cast_map = cast_map
178
179     def llvmType(self, module=None):
180         return super().llvmType(module)
181
182     def _llvmType(self, module):
183         return self._llvm
184
185     def constant(self, value, cge=None):
186         return ll.Constant(self._llvm, value)
187
188     def cast(self, obj, cge, name=None):
189         pytype = obj.type.type_
190         try:
191             cast_name = self.cast_map[pytype]
192         except KeyError:
193             raise exc.TypeError("Cannot cast from {} to
194             ↪ {}".format(obj.type, self))
195         cast_f = getattr(cge.builder, cast_name)
196         return cast_f(obj.llvm, self.llvmType(cge.module), name)
197
198     def __str__(self):
199         return self.name
200
201     def __repr__(self):
202         return "<{0}:{1}>".format(str(type(self))[8:-2], self.name)
203
204 class BoolType(ScalarType):
205     cmp_map = {int: 'icmp_signed', float: 'fcmp_ordered'}
206
207     def __init__(self, name, type_, llvm, ctype):
208         super().__init__(name, type_, llvm, ctype, None)
209
210     def cast(self, obj, cge, name=None):
211         pytype = obj.type.type_
212         try:
213             cmp_name = self.cmp_map[pytype]
214         except KeyError:
215             raise exc.TypeError("Cannot cast from {} to
216             ↪ {}".format(obj.type, self))
```

Appendix C. STELLA Source Code

```
216         cmp_f = getattr(cge.builder, cmp_name)
217         cmp_llvm = cmp_f('!=', obj.llvm, obj.type.constant(0))
218         return cge.builder.zext(cmp_llvm, tp_bool)
219
220
221 tp_int = ll.IntType(64)
222 tp_int32 = ll.IntType(32) # needed for llvm operators
223 tp_double = ll.DoubleType()
224 # Note on booleans:
225 # llvm uses i1 for its conditionals, but i1 does not store well in memory.
226 # Therefore use i8 by default, and translate to i1 when before conditional
227 # instructions
228 tp_bool = ll.IntType(8)
229 tp_bool1 = ll.IntType(1)
230 tp_void = ll.VoidType()
231
232 Int = ScalarType(
233     "Int",
234     int, tp_int, ctypes.c_int64,
235     {float: 'fptosi', bool: 'zext'},
236 )
237 uInt = ScalarType( # TODO: unclear whether this is correct or not
238     "uInt",
239     int, tp_int32, ctypes.c_int32,
240     {},
241 )
242 Float = ScalarType(
243     "Float",
244     float, tp_double, ctypes.c_double,
245     {int: 'sitofp', bool: 'sitofp'},
246 )
247 Bool = BoolType(
248     "Bool",
249     bool, tp_bool, ctypes.c_bool,
250 )
251
252
253 def getIndex(i):
254     if type(i) == int:
255         return ll.Constant(tp_int32, i)
256     else:
257         raise exc.UnimplementedError("Unsupported index type
    ↪ {}".format(type(i)))
```

Appendix C. STELLA Source Code

```
258
259
260 def invalid_none_use(msg):
261     raise exc.StellaException(msg)
262
263
264 None_ = ScalarType(
265     "NONE",
266     type(None), tp_void, None,
267     {},
268 )
269 Void = None_ # TODO: Could there be differences later?
270 Str = ScalarType(
271     "Str",
272     str, None, ctypes.c_char_p,
273     {},
274 )
275
276 _pyscalars = {
277     int: Int,
278     float: Float,
279     bool: Bool,
280 }
281
282
283 def get_scalar(obj):
284     """obj can either be a value, or a type
285
286     Returns the Stella type for the given object"""
287     type_ = type(obj)
288     if type_ == type(int):
289         type_ = obj
290     elif type_ == PyWrapper:
291         type_ = obj.py
292
293     # HACK {
294     if type_ == type(None): # noqa
295         return None_
296     elif type_ == str:
297         return Str
298     # } HACK
299
300     try:
```

Appendix C. STELLA Source Code

```
301     return _pyscalars[type_]
302 except KeyError:
303     raise exc.TypeError("Invalid scalar type '{0}'".format(type_))
304
305
306 def supported_scalar(type_):
307     """type_ is either a Python type or a stella type!"""
308     try:
309         get_scalar(type_)
310         return True
311     except exc.TypeError:
312         # now check for stella scalar types
313         return type_ in _pyscalars.values()
314
315
316 def supported_scalar_name(name):
317     assert type(name) == str
318
319     types = map(lambda x: x.__name__, _pyscalars.keys())
320     return any([name == t for t in types])
321
322
323 class CType(object):
324     """
325     Dynamically create a ctype Structure.
326     """
327     _registry = {}
328
329     @classmethod
330     def getStruct(klass, name, fields=[]):
331         """
332         Creates a Structure with the given fields. Caches based on (name,
333         fields).
334         """
335         fields = tuple(fields)
336         if name not in klass._registry:
337             if fields:
338                 attribs = {'_fields_': fields}
339             else:
340                 attribs = {}
341             type_ = type(name, (ctypes.Structure, ), attribs)
342             klass._registry[name] = type_
343         return type_
```

Appendix C. STELLA Source Code

```
343         else:
344             struct = klass._registry[name]
345             if fields:
346                 assert fields == struct._fields_
347             return struct
348
349     @classmethod
350     def destruct(klass):
351         klass._registry.clear()
352
353
354     class StructType(Type):
355         on_heap = True
356         req_transfer = True
357
358         attrib_names = None
359         attrib_type = None
360         attrib_idx = None
361         _ctype = None
362         _llvmtyp = None
363         type_store = {} # Class variable
364
365     @classmethod
366     def fromObj(klass, obj):
367         if type(obj) == type(int):
368             # class
369             type_name = str(obj).split("'")[1] + ".__class__"
370         else:
371             # instance
372             type_name = str(type(obj)).split("'")[1]
373
374         # cache it early, which allows fields of this type to be resolved
375         # immediately
376         if type_name in klass.type_store:
377             return klass.type_store[type_name]
378
379         type_ = StructType(type_name)
380         type_.makePointer() # by default
381         klass.type_store[type_name] = type_
382
383         attrib_type = {}
384         attrib_idx = {}
```

Appendix C. STELLA Source Code

```
385     attrib_names = sorted(list(filter(lambda s: not
386         ↪ s.startswith('__'),
387                                     dir(obj))))
388     for name in attrib_names:
389         attrib = getattr(obj, name)
390         try:
391             a_type = get(attrib)
392         except exc.UnsupportedTypeError as e:
393             e.prepend(name, type(attrib))
394             raise e
395         if a_type.on_heap:
396             a_type.makePointer(True)
397         attrib_type[name] = a_type
398
399     # Sort attrib_names so that function types are after attribute
400     ↪ types.
401     # This allows me to keep them around, because even though they
402     ↪ aren't
403     # translated into the llvm struct, their presence does not mess up
404     ↪ the
405     # indices.
406     def funcs_last(n):
407         if isinstance(attrib_type[n], FunctionType):
408             return 1
409         else:
410             return 0
411
412     attrib_names = sorted(attrib_names, key=funcs_last)
413     i = 0
414     for name in attrib_names:
415         attrib_idx[name] = i
416         i += 1
417
418     type_.attrib_type = attrib_type
419     type_.attrib_idx = attrib_idx
420     type_.attrib_names = attrib_names
421
422     return type_
423
424 @classmethod
425 def destruct(klass):
426     klass.type_store.clear()
```

Appendix C. STELLA Source Code

```
424     def __init__(self, name):
425         self.name = name
426
427     def getMemberType(self, name):
428         return self.attrib_type[name]
429
430     def getMemberIdx(self, name):
431         return self.attrib_idx[name]
432
433     def _scalarAttributeNames(self):
434         return filter(lambda n: not isinstance(self.attrib_type[n],
435             ↪ FunctionType),
436             self.attrib_names)
437
438     def items(self):
439         """
440         Return (unordered) name, type tuples
441         """
442         # TODO turn this into an iterator?
443         return [(name, self.attrib_type[name]) for name in
444             ↪ self.attrib_names]
445
446     def _llvmType(self, module):
447         if self._llvmtype is not None:
448             return self._llvmtype
449
450         self._llvmtype =
451             ↪ module.llvm.context.get_identified_type(self.name)
452         assert self._llvmtype.is_opaque
453
454         llvm_types = []
455         for name in self._scalarAttributeNames():
456             type_ = self.attrib_type[name]
457             llvm_types.append(type_.llvmType(module))
458         self._llvmtype.set_body(*llvm_types)
459
460         return self._llvmtype
461
462     @property
463     def ctype(self):
464         if self._ctype:
465             return self._ctype
466         fields = []
```

Appendix C. STELLA Source Code

```
464     self._ctype = CType.getStruct("_" + self.name + "_transfer")
465     for name in self._scalarAttributeNames():
466         if isinstance(self.attrib_type[name], ListType):
467             fields.append((name,
468                 ↪ ctypes.POINTER(self.attrib_type[name].ctype)))
469         elif self.attrib_type[name] is self:
470             fields.append((name, ctypes.POINTER(self._ctype)))
471         else:
472             fields.append((name, self.attrib_type[name].ctype))
473     self._ctype._fields_ = fields
474     return self._ctype
475
476 def ctypeInit(self, value, transfer_value):
477     for name in self.attrib_names:
478         item = getattr(value, name)
479         if isinstance(item, np.ndarray):
480             # TODO: will this fail with float?
481             item = ctypes.cast(item.ctypes.data,
482                 ↪ ctypes.POINTER(ctypes.c_int))
483         elif isinstance(item, list):
484             l = List.fromObj(item)
485             l.ctypeInit()
486             item = ctypes.cast(ctypes.addressof(l.transfer_value),
487                 ↪ ctypes.POINTER(l.type.ctype))
488         elif self.attrib_type[name] is self:
489             item = ctypes.cast(ctypes.addressof(transfer_value),
490                 ↪ ctypes.POINTER(self.ctype))
491         elif not supported_scalar(type(item)) and not isinstance(item,
492             ↪ types.MethodType):
493             # struct, has to be the last check because everything is
494             ↪ an
495             # object in Python
496             item = wrapValue(item).transfer_value
497         setattr(transfer_value, name, item)
498
499 def constant(self, value, cge):
500     """Transfer values Python -> Stella"""
501     transfer_value = self.ctype()
502     # logging.debug("StructType.constant() {}/{:x} ->
503     ↪ {}/{:x}".format(self, id(self),
504     #
505     ↪ transfer_value,
```


Appendix C. STELLA Source Code

```
498     #
499     ↪ id(transfer_value))
500
501     assert self.ptr == 1
502
503     self.cTypeInit(value, transfer_value)
504
505     addr_llvm = Int.constant(int(ctypes.addressof(transfer_value)))
506     result_llvm = ll.Constant(tp_int,
507     ↪ addr_llvm).inttoptr(self.llvmType(cge.module))
508     # logging.debug("{} constant() transfer {}:{}".format(value,
509     ↪ transfer_value,
510     #
511     ↪ id(transfer_value)))
512     return (result_llvm, transfer_value)
513
514 def cType2Python(self, transfer_value, value):
515     for name in self._scalarAttributeNames():
516         item = getattr(transfer_value, name)
517         # TODO generalize!
518         if isinstance(self.attrib_type[name], List):
519             l = List.fromObj(item)
520             l.cType2Python(item)
521         elif not self.attrib_type[name].on_heap:
522             # TODO is this actually used?
523             setattr(value, name, item)
524
525 def resetReference(self):
526     """Special case: when a list of objects is allocated, then the
527     ↪ type is NOT a pointer type"""
528     self.ptr = 0
529
530 def __str__(self):
531     return "{}{}".format('*'*self.ptr, self.name)
532
533 def __repr__(self):
534     if self.attrib_type:
535         type_info = list(self.attrib_type.keys())
536     else:
537         type_info = '?'
538     return "<{}{}: {}>".format('*'*self.ptr, self.name, type_info)
539
540 def __eq__(self, other):
```

Appendix C. STELLA Source Code

```
536         return (type(self) == type(other)
537                and self.attrib_names == other.attrib_names
538                and self.attrib_type == other.attrib_type)
539
540     def __ne__(self, other):
541         return not self.__eq__(other)
542
543     @classmethod
544     def unpack(klass, val):
545         # logging.debug("{} / {} unpack()".format(val, id(val)))
546         # logging.debug("{} * {}".format(ctypes.addressof(val.contents)))
547         if (val):
548             addr = ctypes.addressof(val.contents)
549             return Struct.obj_store[addr].value
550         else:
551             # null pointer
552             return None
553
554
555     class TupleType(ScalarType, Subscriptable):
556         def __init__(self, types):
557             self.types = types
558
559         def __eq__(self, other):
560             return isinstance(other, self.__class__) and self.types ==
561                 ⇨ other.types
562
563         def __ne__(self, other):
564             return not self.__eq__(other)
565
566         @property
567         def len(self):
568             return len(self.types)
569
570         def _getConst(self, idx):
571             if isinstance(idx, int):
572                 return idx
573             elif isinstance(idx, Const):
574                 return idx.value
575             else:
576                 raise exc.TypeError("Tuple index must be constant, not
577                 ⇨ {}".format(type(idx)))
```

Appendix C. STELLA Source Code

```
577     def getElementType(self, idx):
578         val = self._getConst(idx)
579         if val >= len(self.types):
580             raise exc.IndexError("tuple index out of range")
581         return self.types[val]
582
583     def loadSubscript(self, cge, container, idx):
584         val = self._getConst(idx)
585         return cge.builder.extract_value(container.translate(cge), [val])
586
587     def storeSubscript(self, cge, container, idx, value):
588         # TODO Needs tests!
589         idx_val = self._getConst(idx)
590         cge.builder.insert_value(container.translate(cge),
591                                 ↪ value.translate(cge), idx_val)
592
593     def CType(self):
594         fields = [{"f{}".format(i), val.ctype} for i, val in
595                  ↪ enumerate(self.types)]
596         return CType.getStruct("__tuple__", fields)
597
598     @classmethod
599     def unpack(klass, val):
600         """
601         Convert a ctypes.Structure wrapper into a native tuple.
602         """
603         l = [getattr(val, n) for n, _ in val._fields_]
604         return tuple(l)
605
606     def _llvmType(self, module):
607         return ll.LiteralStructType([t.llvmType(module) for t in
608                                     ↪ self.types])
609
610     def constant(self, values, cge=None):
611         if not self._llvm:
612             self._llvm =
613                 ↪ ll.Constant.literal_struct([wrapValue(v).translate(cge)
614                                             ↪ for v in values])
615         return self._llvm
616
617     def __str__(self):
618         return "tuple, {} elems".format(len(self.types))
619
620
```

Appendix C. STELLA Source Code

```
615     def __repr__(self):
616         return "{}".format(", ".join([str(t) for t in self.types]))
617
618
619 class ArrayType(Type, Subscriptable):
620     type_ = NoType
621     shape = None
622     on_heap = True
623     ctype = ctypes.POINTER(ctypes.c_int) # TODO why is
        → ndarray.ctypes.data of type int?
624
625     @classmethod
626     def fromObj(klass, obj):
627         # TODO support more types
628         if obj.dtype == np.int64:
629             dtype = _pyscalars[int]
630         elif obj.dtype == np.float64:
631             dtype = _pyscalars[float]
632         else:
633             raise exc.UnimplementedError("Numpy array dtype {0} not (yet)
        → supported".format(
634                 obj.dtype))
635
636         # TODO: multidimensional arrays
637         shape = obj.shape
638
639         assert klass.isValidType(dtype)
640
641         try:
642             ndim = len(shape)
643         except TypeError:
644             ndim = 1
645
646         if ndim == 0:
647             raise exc.UnimplementedError("Array with zero dimensions is
        → not supported.")
648         elif ndim == 1:
649             return ArrayType(dtype, shape[0])
650         else:
651             return ArrayNdType(dtype, shape)
652
653     @classmethod
654     def isValidType(klass, type_):
```

Appendix C. STELLA Source Code

```
655     return type_ in _pyscalars.values()
656
657     def __init__(self, type_, shape):
658         self.type_ = type_
659         self.shape = shape
660
661     def _boundsCheck(self, idx):
662         """Check bounds, if possible. This is a compile time operation."""
663         if isinstance(idx, Const) and idx.value >= self.shape:
664             raise exc.IndexError("array index out of range")
665
666     def getElementType(self, idx):
667         self._boundsCheck(idx)
668         return self.type_
669
670     def _llvmType(self, module):
671         type_ = ll.ArrayType(self.type_.llvmType(module), self.shape)
672         return type_
673
674     def __str__(self):
675         return "{}{}[{}]".format('*'*self.ptr, self.type_, self.shape)
676
677     def __repr__(self):
678         return '<{}>'.format(self)
679
680     def loadSubscript(self, cge, container, idx):
681         self._boundsCheck(idx)
682         p = cge.builder.gep(container.translate(cge),
683                             [Int.constant(0), idx.translate(cge)],
684                             inbounds=True)
685         return cge.builder.load(p)
686
687     def cast(self, value, cge):
688         if value.type == self.type_:
689             return value.translate(cge)
690         if value.type == Int and self.type_ == Float:
691             return Cast(value, Float).translate(cge)
692         raise TypeError("Cannot store {} into an array of {}".format(
693             value.type, self.type_))
694
695     def storeSubscript(self, cge, container, idx, value):
696         self._boundsCheck(idx)
697         p = cge.builder.gep(
```

Appendix C. STELLA Source Code

```
698         container.translate(cge), [
699             Int.constant(0), idx.translate(cge)], inbounds=True)
700     val = self.cast(value, cge)
701     cge.builder.store(val, p)
702
703
704 class ArrayNdType(ArrayType):
705     def __init__(self, type_, shape):
706         super().__init__(type_, shape)
707
708     def _boundsCheck(self, idx):
709         """Check bounds, if possible. This is a compile time operation."""
710         if isinstance(idx, Const):
711             if isinstance(idx.type, TupleType):
712                 ndim = len(idx.value)
713             else:
714                 ndim = 1
715             if len(self.shape) != ndim:
716                 msg = "TODO: indexing with {} dimensions into an
717                     ↪ {}-dimensional array".format(
718                         ndim, len(self.shape))
719                 raise exc.TypeError(msg)
720             for i in range(len(self.shape)):
721                 if idx.value[i] >= self.shape[i]:
722                     msg = "array index {} out of range: {} >=
723                         ↪ {}".format(i, idx.value[i],
724                                     ↪ self.shape[i])
725                     raise exc.IndexError(msg)
726
727     def _llvmType(self, module):
728         type_ = ll.ArrayType(self.type_.llvmType(module),
729                             ↪ reduce(operator.mul, self.shape))
730         return type_
731
732     def _generateIndex(self, cge, idx):
733         # TODO: test with dim > 2
734         assert idx.type.len > 1
735         flat_idx = Const(0).translate(cge)
736         for i in range(idx.type.len-1):
737             idx_val = idx.type.loadSubscript(cge, idx, i)
738             flat_idx = cge.builder.add(flat_idx,
739                                     ↪ cge.builder.mul(idx_val,
```

Appendix C. STELLA Source Code

```
737
                                                    ↪ Const(self.shape[i+1]).trans
738
739     idx_val = idx.type.loadSubscript(cge, idx, idx.type.len-1)
740     flat_idx = cge.builder.add(flat_idx, idx_val)
741     return [Int.constant(0), flat_idx]
742
743 def loadSubscript(self, cge, container, idx):
744     self._boundsCheck(idx)
745     p = cge.builder.gep(container.translate(cge),
746                         self._generateIndex(cge, idx),
747                         inbounds=True)
748     return cge.builder.load(p)
749
750 def storeSubscript(self, cge, container, idx, value):
751     self._boundsCheck(idx)
752     p = cge.builder.gep(container.translate(cge),
753                         self._generateIndex(cge, idx),
754                         inbounds=True)
755     val = self.cast(value, cge)
756     cge.builder.store(val, p)
757
758
759 class ListType(ArrayType):
760     req_transfer = True
761     type_store = {} # Class variable
762
763     @classmethod
764     def fromObj(klass, obj):
765         # type checking: only continue if the list can be represented.
766         if len(obj) == 0:
767             msg = "Empty lists are not supported, because they are not
768                 ↪ typable"
769             raise exc.UnsupportedTypeError(msg)
770         type_ = type(obj[0])
771         for o in obj[1:]:
772             if type_ != type(o):
773                 msg = "List contains elements of type {} and type {}, but
774                     ↪ lists must not contain objects of more than one
775                     ↪ type".format( # noqa
776                         type_, type(o))
777                 raise exc.UnsupportedTypeError(msg)
```

Appendix C. STELLA Source Code

```
776     base_type = get(obj[0])
777     if not isinstance(base_type, StructType):
778         msg = "Python lists must contain objects, not {}". Use numpy
779             ↪ arrays for simple types".format(base_type) # noqa
780         raise exc.UnsupportedTypeError(msg)
781     base_type.resetReference()
782     # assert !klass.isValidType(dtype)
783
784     # type_name = "[{}]".format(str(type(obj[0])).split("'")[1])
785     type_ = klass(base_type, len(obj))
786     return type_
787
788 def getElementType(self, idx):
789     return Reference(super().getElementType(idx))
790
791 @classmethod
792 def destruct(klass):
793     klass.type_store.clear()
794
795 def __init__(self, base_type, shape):
796     super().__init__(base_type, shape)
797
798 def _llvmType(self, module):
799     mangled_name = str(self)
800
801     if mangled_name not in self.__class__.type_store:
802         type_ = ll.ArrayType(self.type_.llvmType(module), self.shape)
803         self.__class__.type_store[mangled_name] = type_
804         return type_
805     else:
806         return self.__class__.type_store[mangled_name]
807
808 def ctypeInit(self, value, transfer_value):
809     for i in range(len(value)):
810         self.type_.ctypeInit(value[i], transfer_value[i])
811
812 @property
813 def ctype(self):
814     return self.type_.ctype * self.shape
815
816 def __eq__(self, other):
817     return (type(self) == type(other)
818             and self.type_ == other.type_)
```


Appendix C. STELLA Source Code

```
818         and self.shape == other.shape)
819
820     def __ne__(self, other):
821         return not self.__eq__(other)
822
823     def loadSubscript(self, cge, container, idx):
824         # TODO address calculation is same as for ArrayType, unify?
825         p = cge.builder.gep(container.translate(cge),
826                             [Int.constant(0), idx.translate(cge)],
827                             inbounds=True)
828         return p
829
830
831     class Callable(metaclass=ABCMeta):
832         def combineArgs(self, args, kwargs):
833             """Combine concrete args and kwargs according to calling
834             ↪ conventions.
835
836             Precondition: Typing has been performed, so typeArgs already
837             ↪ ensures
838             that the correct number of arguments are provided.
839             """
840             return self.type._combineArgs(args, kwargs)
841
842         def call(self, cge, args, kw_args):
843             combined_args = self.combineArgs(args, kw_args)
844
845             return cge.builder.call(self.llvm, [arg.translate(cge) for arg in
846             ↪ combined_args])
847
848
849     class Foreign(object):
850         """Mixin: This is not a Python function. It does not need to get
851         ↪ analyzed."""
852         pass
853
854
855     class FunctionType(Type):
856         _registry = {}
857
858         @classmethod
859         def get(klass, obj, bound=None, builtin=False):
860             if bound:
```

Appendix C. STELLA Source Code

```
857         key = (str(bound), obj.__name__)
858     else:
859         key = obj
860
861     if key not in klass._registry:
862         klass._registry[key] = klass(obj, bound, builtin)
863
864     return klass._registry[key]
865
866 @classmethod
867 def destruct(klass):
868     klass._registry.clear()
869
870 def __init__(self, obj, bound=None, builtin=False):
871     """Type representing a function.
872
873     obj: Python function reference
874     bound: self if it is a method
875     builtin: True if e.g. len
876
877     Assumption: bound or builtin
878     """
879     self.name = obj.__name__
880     self._func = obj
881     self.bound = bound
882     self._builtin = builtin
883
884     self.readSignature(obj)
885
886 def pyFunc(self):
887     return self._func
888
889 @property
890 def bound(self):
891     """None if a regular function, returns the type of self if a bound
892     ↪ method
893
894     Note that unbound methods are not yet supported
895     """
896     # This is not very elegant, but correct. Self should always be a
897     # reference and never get passed in by value.
898     if self._bound and not self._bound.isReference():
899         return Reference(self._bound)
```

Appendix C. STELLA Source Code

```
899         return self._bound
900
901     @bound.setter
902     def bound(self, obj):
903         assert obj is None or isinstance(obj, Type)
904         self._bound = obj
905
906     @property
907     def builtin(self):
908         return self._builtin
909
910     arg_defaults = []
911     tp_defaults = []
912     arg_names = []
913     arg_types = []
914     def_offset = 0
915
916     def readSignature(self, f):
917         argspec = inspect.getargspec(f)
918         self.arg_names = argspec.args
919         self.arg_defaults = [Const(default) for default in
920             ↪ argspec.defaults or []]
921         self.tp_defaults = [d.type for d in self.arg_defaults]
922         self.def_offset = len(self.arg_names)-len(self.arg_defaults)
923
924     def typeArgs(self, tp_args, tp_kwargs):
925         # TODO store the result?
926
927         if self.bound:
928             tp_args.insert(0, self.bound)
929
930         num_args = len(tp_args)
931         if num_args+len(tp_kwargs) <
932             ↪ len(self.arg_names)-len(self.arg_defaults):
933             raise exc.TypeError("takes at least {0} argument(s) ({1}
934                 ↪ given)".format(
935                 len(self.arg_names)-len(self.arg_defaults),
936                 ↪ len(tp_args)+len(tp_kwargs)))
937         if num_args+len(tp_kwargs) > len(self.arg_names):
938             raise exc.TypeError("takes at most {0} argument(s) ({1}
939                 ↪ given)".format(
940                 len(self.arg_names), len(tp_args)))
941
942
```

Appendix C. STELLA Source Code

```
937     if len(self.arg_types) == 0:
938         self.arg_types = self._combineArgs(tp_args, tp_kwargs,
939             ↪ self.tp_defaults)
939     else:
940         # Already typed, so the supplied arguments must match what the
941             ↪ last
942         # call used.
943         supplied_args = self._combineArgs(tp_args, tp_kwargs,
944             ↪ self.tp_defaults)
945         for i, prototype, supplied in zip(range(len(supplied_args)),
946             self.arg_types,
947             supplied_args):
948             if prototype != supplied:
949                 raise exc.TypeError("Argument {} has type {}, but type
950                     ↪ {} was supplied".format(
951                         self.arg_names[i], prototype, supplied))
952     return self.arg_types
953
954 def _combineArgs(self, args, kwargs, defaults=None):
955     """Combine concrete or types of args and kwargs according to
956         ↪ calling conventions."""
957     if defaults is None:
958         defaults = self.arg_defaults
959     num_args = len(args)
960     r = [None] * len(self.arg_names)
961
962     # copy supplied regular arguments
963     for i in range(len(args)):
964         r[i] = args[i]
965
966     # set default values
967     for i in range(max(num_args, len(self.arg_names)-len(defaults)),
968         len(self.arg_names)):
969         r[i] = defaults[i-self.def_offset]
970
971     # insert kwargs
972     for k, v in kwargs.items():
973         try:
974             idx = self.arg_names.index(k)
975             if idx < num_args:
976                 raise exc.TypeError("got multiple values for keyword
977                     ↪ argument '{0}'".format(
978                         self.arg_names[idx]))
```

Appendix C. STELLA Source Code

```
974         r[idx] = v
975     except ValueError:
976         raise exc.TypeError("Function does not take an {0}
977             ↪ argument".format(k))
978
979     return r
980
981 def __str__(self):
982     if self._bound:
983         tp_name = str(self._bound)
984         return "<bound method {}.{}>".format(tp_name,
985             ↪ self._func.__name__)
986     else:
987         return "<function {}>".format(self._func.__name__)
988
989 @property
990 def fq(self):
991     """Returns the fully qualified type name."""
992     if self._bound:
993         # TODO this should probably always be a reference
994         if self.bound.isReference():
995             # bound is a reference, we don't want the * as part of the
996             ↪ name
997             return "{}.{}".format(str(self.bound)[1:], self.name)
998         else:
999             return "{}.{}".format(str(self.bound), self.name)
1000     else:
1001         return self.name
1002
1003 def __llvmType(self, module):
1004     raise exc.InternalError("This is an intermediate type presentation
1005         ↪ only!")
1006
1007 class IntrinsicType(FunctionType):
1008     def __init__(self, f, names, defaults):
1009         super().__init__(f, bound=None, builtin=True)
1010         self.arg_names = names
1011         self.arg_defaults = defaults
1012         self.def_offset = len(self.arg_names)-len(self.arg_defaults)
1013
1014     def readSignature(self, f):
1015         """The signature is built in for Ininsics. NOOP."""
```

Appendix C. STELLA Source Code

```
1013     pass
1014
1015
1016 class ExtFunctionType(FunctionType):
1017     def __init__(self, python, signature):
1018         super().__init__(python, builtin=True)
1019         ret, arg_types = signature
1020         self.return_type = from_ctype(ret)
1021         self.arg_types = list(map(from_ctype, arg_types))
1022         self.readSignature(None)
1023
1024     def __str__(self):
1025         return "<{} function({})>".format(self.return_type,
1026                                           ", ".join(zip(self.arg_types,
1027                                                         self.arg_names)))
1027
1028     def readSignature(self, f):
1029         # arg, inspect.getargspec(f) doesn't work for C/cython functions
1030         self.arg_names = ['arg{0}'.format(i) for i in range(len(self.arg_types))]
1031         self.arg_defaults = []
1032
1033     def getReturnType(self, args, kw_args):
1034         return self.return_type
1035
1036
1037 def llvm_to_py(type_, val):
1038     if type_ == Int:
1039         return val.as_int_signed()
1040     elif type_ == Float:
1041         return val.as_real(type_.llvmType())
1042     elif type_ == Bool:
1043         return bool(val.as_int())
1044     elif type_ is None_:
1045         return None
1046     else:
1047         raise exc.TypeError("Unknown type {0}".format(type_))
1048
1049
1050 def get(obj):
1051     """Resolve python object -> Stella type"""
1052     type_ = type(obj)
1053     if supported_scalar(type_):
1054         return get_scalar(type_)
```

Appendix C. STELLA Source Code

```
1055     elif type_ == np.ndarray:
1056         return ArrayType.fromObj(obj)
1057     elif type_ == list:
1058         return ListType.fromObj(obj)
1059     elif isinstance(obj, types.FunctionType):
1060         return FunctionType.get(obj)
1061     elif isinstance(obj, types.MethodType):
1062         return FunctionType.get(obj, bound=get(obj.__self__))
1063     elif isinstance(obj, types.BuiltinFunctionType):
1064         return FunctionType.get(obj, builtin=True)
1065     elif isinstance(obj, types.BuiltinMethodType):
1066         assert False and "TODO: This case has not been completely
           ↪ implemented"
1067         return FunctionType.get(obj, bound=True, builtin=True)
1068     elif isinstance(obj, tuple):
1069         return TupleType([get(e) for e in obj])
1070     else:
1071         # TODO: How to identify unsupported objects? Everything is an
           ↪ object...
1072         return StructType.fromObj(obj)
1073
1074
1075     _cscalars = {
1076         ctypes.c_double: Float,
1077         ctypes.c_uint: UInt,
1078         None: Void
1079     }
1080
1081
1082     def from_ctype(type_):
1083         assert type(type_) == type(ctypes.c_int) or type(type_) == type(None)
           ↪ # noqa
1084         return _cscalars[type_]
1085
1086
1087     # values which need to be destructed: Any complex type that is wrapped.
1088     _stella_values_ = []
1089
1090
1091     class Typable(object):
1092         type = NoType
1093         llvm = None
1094
```

Appendix C. STELLA Source Code

```
1095 def unify_type(self, tp2, debuginfo):
1096     """Returns: widened:bool, needs_cast:bool
1097     widened: this type changed
1098     needs_cast: tp2 needs to be cast to this type
1099     """
1100     tp1 = self.type
1101     if tp1 != NoType and tp2 != NoType and tp1.ptr != tp2.ptr:
1102         if tp2 is None_:
1103             # special case: return NULL
1104             return False, False
1105         else:
1106             raise exc.TypeError("Inconsistent pointers: {} does not
1107                 ↳ match {}".format(tp1, tp2),
1108                 debuginfo)
1109     if tp1 == tp2:
1110         pass
1111     elif tp1 == NoType:
1112         self.type = tp2
1113     elif tp2 == NoType:
1114         pass
1115     elif tp1 == Int and tp2 == Float:
1116         self.type = Float
1117         return True, False
1118     elif tp1 == Float and tp2 == Int:
1119         # Note that the type does not have to change here because
1120         ↳ Float is
1121         # already wider than Int
1122         return False, True
1123     elif isinstance(tp1, Reference) and isinstance(tp2, Reference):
1124         while isinstance(tp1, Reference) and isinstance(tp2,
1125             ↳ Reference):
1126             tp1 = tp1.type_
1127             tp2 = tp2.type_
1128         if tp1 != tp2:
1129             raise exc.TypeError("{} is not compatible with
1130                 ↳ {}".format(tp1, tp2), debuginfo)
1131     else:
1132         raise exc.TypeError("Unifying of types {} and {} (not yet)
1133             ↳ implemented".format(
1134                 tp1, tp2), debuginfo)
1135     return False, False
```


Appendix C. STELLA Source Code

```
1133     def llvmType(self, module):
1134         """Map from Python types to LLVM types."""
1135         return self.type.llvmType(module)
1136
1137     def translate(self, cge):
1138         # TODO assert self.llvm here? This fails with uninitialized values
1139         ↪ like
1140         # test.langconstr.new_global_var
1141         return self.llvm
1142
1143     def ctype2Python(self, cge):
1144         pass
1145
1146     def destruct(self):
1147         pass
1148
1149 class ImmutableType(object):
1150     def unify_type(self, tp2, debuginfo):
1151         raise TypeError("Type {} is immutable, it cannot be unified with
1152             ↪ {} at {}".format(
1153                 self.type, tp2, debuginfo))
1154
1155     def llvmType(self, module):
1156         """Map from Python types to LLVM types."""
1157         return self.type.llvmType(module)
1158
1159 class Const(Typable):
1160     value = None
1161
1162     def __init__(self, value):
1163         self.value = value
1164         try:
1165             if type(value) == tuple:
1166                 self.type = TupleType([get(v) for v in value])
1167             else:
1168                 self.type = get_scalar(value)
1169                 self.name = str(value)
1170         except exc.TypeError as e:
1171             self.name = "InvalidConst({0}, type={1})".format(value,
1172                 ↪ type(value))
1172             raise e
```

Appendix C. STELLA Source Code

```
1173
1174 def unify_type(self, tp2, debuginfo):
1175     r = super().unify_type(tp2, debuginfo)
1176     return r
1177
1178 def translate(self, cge):
1179     if self.type.on_heap:
1180         (self.llvm, self.transfer_value) =
1181             ↪ self.type.constant(self.value, cge)
1182     elif self.type is Bool:
1183         # llvmlite would output 'true' (which is i1) for True even
1184         ↪ when the type is i8
1185         self.llvm = self.type.constant(int(self.value), cge)
1186     else:
1187         self.llvm = self.type.constant(self.value, cge)
1188     return self.llvm
1189
1190 def __str__(self):
1191     return self.name
1192
1193 def __repr__(self):
1194     return self.__str__()
1195
1196 class NumpyArray(Typable):
1197     def __init__(self, array):
1198         assert isinstance(array, np.ndarray)
1199
1200         # TODO: multi-dimensional arrays
1201         self.type = Reference(ArrayType.fromObj(array))
1202         self.value = array
1203
1204     def translate(self, cge):
1205         ptr_int = self.value.ctypes.data
1206         ptr_int_llvm = Int.constant(ptr_int)
1207
1208         type_ = self.type.llvmType(cge.module)
1209         self.llvm = ll.Constant(tp_int, ptr_int_llvm).inttoptr(type_)
1210         return self.llvm
1211
1212     def __str__(self):
1213         return str(self.type)
```

Appendix C. STELLA Source Code

```
1214     def __repr__(self):
1215         return str(self)
1216
1217
1218     class Struct(Typable):
1219         obj_store = {} # Class variable, used to map returned objects back to
1220             ↪ the original Python
1221
1222         @classmethod
1223         def fromObj(klass, obj):
1224             """Only one Struct representation per Python object instance.
1225             """
1226             if not hasattr(obj, '__stella_wrapper__'):
1227                 obj.__stella_wrapper__ = Struct(obj)
1228                 _stella_values_.append(obj.__stella_wrapper__)
1229             assert isinstance(obj.__stella_wrapper__, klass)
1230             return obj.__stella_wrapper__
1231
1232         def __init__(self, obj):
1233             self.type = StructType.fromObj(obj)
1234             self.value = obj
1235             self.transfer_attributes = {}
1236             for name, type_ in self.type.items():
1237                 if type_.req_transfer:
1238                     attrib = getattr(obj, name)
1239                     if id(attrib) == id(obj):
1240                         # self-reference TODO does it even need a transfer
1241                         ↪ then?
1242                         # self.transfer_attributes[name] = self
1243                         pass
1244                     else:
1245                         self.transfer_attributes[name] = wrapValue(attrib)
1246
1247         def __str__(self):
1248             return str(self.type)
1249
1250         def __repr__(self):
1251             return repr(self.type)
1252
1253         def translate(self, cge):
1254             for wrapped in self.transfer_attributes.values():
1255                 wrapped.translate(cge)
1256             if not self.llvm:
```

Appendix C. STELLA Source Code

```
1255         (self.llvm, self.transfer_value) =
1256             ↪ self.type.constant(self.value, cge)
1257         # logging.debug("translate() of {}:
1258             ↪ *{:x}".format(self.transfer_value,
1259                 #
1260                 ↪ ctypes.addressof(self.transfer_value)))
1261         addr = ctypes.addressof(self.transfer_value)
1262         self.__class__.obj_store[addr] = self
1263         assert self.transfer_value
1264         return self.llvm
1265
1266 def ctype2Python(self, cge):
1267     """At the end of a Stella run, the struct's values need to be
1268     ↪ copied back
1269     into Python.
1270
1271     Please call self.destruct() afterwards.
1272     """
1273     self.type.ctype2Python(self.transfer_value, self.value)
1274     for wrapped in self.transfer_attributes.values():
1275         wrapped.ctype2Python(cge)
1276
1277 def destruct(self):
1278     # TODO why don't we always have a transfer_value?
1279     if hasattr(self, 'transfer_value'):
1280         # logging.debug("{} / {:x} destruct()".format(self,
1281             ↪ id(self.transfer_value)))
1282         addr = ctypes.addressof(self.transfer_value)
1283         del self.__class__.obj_store[addr]
1284         del self.transfer_value
1285         del self.value.__stella_wrapper__
1286
1287 class List(Typable):
1288     _registry = {} # Class variable
1289
1290     @classmethod
1291     def fromObj(klass, obj):
1292         """Only one Struct representation per Python object instance.
1293         """
1294         if id(obj) not in klass._registry:
1295             wrapped = klass(obj)
1296             klass._registry[id(obj)] = wrapped
```

Appendix C. STELLA Source Code

```
1293         _stella_values_.append(wrapped)
1294     return klass._registry[id(obj)]
1295
1296     @classmethod
1297     def destructList(klass):
1298         klass._registry.clear()
1299
1300     def __init__(self, obj):
1301         self.type = ListType.fromObj(obj)
1302         self.type.makePointer()
1303         self.value = obj
1304
1305         self.transfer_value = self.type.ctype()
1306
1307     def __str__(self):
1308         return str(self.type)
1309
1310     def __repr__(self):
1311         return repr(self.type)
1312
1313     def ctypeInit(self):
1314         self.type.ctypeInit(self.value, self.transfer_value)
1315
1316     def translate(self, cge):
1317         if self.llvm:
1318             return self.llvm
1319
1320         self.ctypeInit()
1321
1322         addr_llvm =
1323             ↪ Int.constant(int(ctypes.addressof(self.transfer_value)))
1324         self.llvm = ll.Constant(tp_int,
1325             ↪ addr_llvm).inttoptr(self.type.llvmType(cge.module))
1326         return self.llvm
1327
1328     def ctype2Python(self, cge):
1329         """At the end of a Stella run, all list elements need to get
1330         ↪ copied back
1331         into Python.
1332
1333         Please call self.destruct() afterwards.
1334         """
1335         for i in range(len(self.value)):
```

Appendix C. STELLA Source Code

```
1333         self.type.type_.ctype2Python(self.transfer_value[i],
1334         ↪ self.value[i])
1334
1335     def destruct(self):
1336         del self.transfer_value
1337
1338     def loadSubscript(self, cge, container, idx):
1339         p = cge.builder.gep(container.translate(cge),
1340         [Int.constant(0), idx.translate(cge)],
1341         inbounds=True)
1342         return p
1343
1344     def storeSubscript(self, cge, container, idx, value):
1345         p = cge.builder.gep(
1346         container.translate(cge), [
1347         Int.constant(0), idx.translate(cge)], inbounds=True)
1348         cge.builder.store(value.translate(cge), p)
1349
1350
1351     class Tuple(Typable):
1352         def unify_type(self, o, debuginfo):
1353             assert isinstance(o, TupleType)
1354             my_types = self.type.types
1355             if len(my_types) != len(o.types):
1356                 raise exc.TypeError("A {}-tuple is not compatible with a
1357                 ↪ {}-tuple".format(
1358                 len(my_types), len(o.types)))
1359             for i in range(len(my_types)):
1360                 # TODO: unify this with Typable.unify_type somehow?
1361                 tp1 = my_types[i]
1362                 tp2 = o.types[i]
1363                 if tp1 == NoType:
1364                     my_types[i] = tp2
1365                 elif tp2 == NoType:
1366                     pass
1367                 elif tp1 == Int and tp2 == Float:
1368                     my_types[i] = Float
1369                     return True, False
1370                 elif tp1 == Float and tp2 == Int:
1371                     # Note that the type does not have to change here because
1372                     ↪ Float is
1373                     # already wider than Int
1374                     return False, True
```

Appendix C. STELLA Source Code

```
1373         return False, False
1374
1375     def __init__(self, values):
1376         self.values = [wrapValue(v) for v in values]
1377         self.type = TupleType([v.type for v in self.values])
1378
1379     def __str__(self):
1380         return "(tuple, {} elems)".format(len(self.values))
1381
1382     def __repr__(self):
1383         return "({})".format(", ".join([str(v) for v in self.values]))
1384
1385     def translate(self, cge):
1386         if self.llvm:
1387             return self.llvm
1388
1389         # A struct is a constant and can only be initialize with
1390         ↪ constants.
1391         # Insert the non-constants afterwards
1392         init = []
1393         self.inserts = []
1394         for i, v in enumerate(self.values):
1395             if isinstance(v, Const):
1396                 init.append(v.translate(cge))
1397             else:
1398                 init.append(ll.Constant(v.type.llvmType(cge.module),
1399                                     ↪ None))
1400                 self.inserts.append((i, v))
1401
1402         self.llvm = ll.Constant.literal_struct(init)
1403         for i, v in self.inserts:
1404             self.llvm = cge.builder.insert_value(self.llvm,
1405                                                 ↪ v.translate(cge), i)
1406         return self.llvm
1407
1408     def wrapValue(value):
1409         if isinstance(value, Typable):
1410             # already wrapped, nothing to do
1411             return value
1412         type_ = type(value)
1413         if supported_scalar(type_) or type_ == tuple:
1414             return Const(value)
```

Appendix C. STELLA Source Code

```
1413     elif type_ == np.ndarray:
1414         return NumpyArray(value)
1415     elif type_ == list:
1416         return List.fromObj(value)
1417     else:
1418         return Struct.fromObj(value)
1419
1420
1421 class Cast(Typable):
1422     def __init__(self, obj, tp):
1423         self.obj = obj
1424         self.type = tp
1425         self.emitted = False
1426
1427         logging.debug("Casting {0} to {1}".format(self.obj.name,
1428             ↪ self.type))
1429         self.name = "{0}{1}".format(self.type, self.obj.name)
1430
1431     def translate(self, cge):
1432         """This is a special case:
1433         The .llvm attribute is set by bytecode that is being cast here.
1434         So save it in obj, and generate our own .llvm
1435         """
1436         if self.emitted:
1437             return self.llvm
1438         self.emitted = True
1439
1440         # TODO: HACK: instead of failing, let's make it a noop
1441         # I need to check WHY these casts are being created and if I can
1442         ↪ avoid
1443         # them
1444         if self.obj.type == self.type:
1445             self.llvm = self.obj.llvm
1446             return self.llvm
1447
1448         if isinstance(self.obj, Const):
1449             value = self.type.type_(self.obj.value)
1450             self.llvm = self.type.constant(value)
1451         else:
1452             self.llvm = self.type.cast(self.obj, cge, self.name)
1453         return self.llvm
1454
1455     @staticmethod
```


Appendix C. STELLA Source Code

```
1454     def translate_i1(obj, cge):
1455         assert obj.type is Bool
1456         if obj.name:
1457             name = '(i1)' + obj.name
1458         else:
1459             name = None
1460         return cge.builder.trunc(obj.translate(cge), tp_bool1, name)
1461
1462     def __str__(self):
1463         return self.name
1464
1465
1466 def destruct():
1467     global _stella_values_
1468     for value in _stella_values_:
1469         value.destruct()
1470     _stella_values_ = []
1471     FunctionType.destruct()
1472     StructType.destruct()
1473     ListType.destruct()
1474     CType.destruct()
1475     List.destructList()
```

C.3 stella/intrinsics/python.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 def zeros(shape=1, dtype=None):
```

Appendix C. STELLA Source Code

```
15     """
16     Emulate certain features of 'numpy.zeros'
17
18     Note:
19     * 'dtype' is ignored in Python, but will be interpreted in Stella.
20     * This is for testing only! Memory allocation (and deallocation) is
    → not
21     a feature of Stella at this point in time.
22     """
23     try:
24         dim = len(shape)
25         if dim == 1:
26             shape = shape[0]
27             raise TypeError()
28     except TypeError:
29         return [0 for i in range(shape)]
30
31     # here dim > 1, build up the inner most dimension
32     inner = [0 for i in range(shape[dim-1])]
33     for i in range(dim-2, -1, -1):
34         new_inner = [list(inner) for j in range(shape[i])]
35         inner = new_inner
36     return inner
```

C.4 stella/intrinsics/_init__.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
```

Appendix C. STELLA Source Code

```
14 """
15 Intrinsics
16 """
17
18 import sys
19 import math
20 from abc import abstractmethod
21 import builtins
22 import llvmlite.ir as ll
23
24 from . import python
25 from .. import tp, exc
26 from ..storage import Register
27 import numpy as np
28
29
30 class Intrinsic(tp.Foreign, tp.Callable):
31     py_func = None
32     arg_names = []
33     arg_defaults = []
34
35     def __init__(self):
36         self.type_ = tp.IntrinsicType(self.py_func, self.arg_names,
37             ↪ self.arg_defaults)
38
39     @abstractmethod
40     def call(self, cge, args, kw_args):
41         """args and kw_args are already added by a call through
42         ↪ addArgs()"""
43         pass
44
45     def getResult(self, func):
46         return Register(func)
47
48
49 class Zeros(Intrinsic):
50     py_func = python.zeros
51     arg_names = ['shape', 'dtype']
52     arg_defaults = [tp.PyWrapper(int)]
53
54     def getReturnType(self, args, kw_args):
55         combined = self.combineArgs(args, kw_args)
56         shape = combined[0].value
```

Appendix C. STELLA Source Code

```
55
56     # the Python type when passed around in Stella is the intrinsic
57     # function which is a subtype of Cast
58     assert isinstance(combined[1], Cast)
59     type_ = tp.get_scalar(combined[1].py_func)
60
61     if not tp.supported_scalar(type_):
62         raise exc.TypeError("Invalid array element type
63         ↪ {0}".format(type_))
64     atype = tp.ArrayType(type_, shape)
65     atype.complex_on_stack = True
66     atype.on_heap = False
67     return atype
68
69 def call(self, cge, args, kw_args):
70     type_ = self.getReturnType(args, kw_args).llvmType(cge.module)
71     return cge.builder.alloca(type_)
72
73 class Len(Intrinsic):
74     """
75     Determine the length of the array based on its type.
76     """
77     py_func = len
78     arg_names = ['obj']
79
80 def getReturnType(self, args, kw_args):
81     return tp.Int
82
83 def getResult(self, func):
84     # we need the reference to back-patch
85     self.result = tp.Const(-42)
86     return self.result
87
88 def call(self, cge, args, kw_args):
89     obj = args[0]
90     if obj.type.isReference():
91         type_ = obj.type.dereference()
92     else:
93         type_ = obj.type
94     if not isinstance(type_, tp.ArrayType):
95         raise exc.TypeError("Invalid array type {0}".format(obj.type))
96     self.result.value = type_.shape
```

Appendix C. STELLA Source Code

```
97         self.result.translate(cge)
98         return self.result.llvm
99
100
101 class Log(Intrinsic):
102     py_func = math.log
103     intr = 'llvm.log'
104     arg_names = ['x'] # TODO: , base
105
106     def getReturnType(self, args, kw_args):
107         return tp.Float
108
109     def call(self, cge, args, kw_args):
110         if args[0].type == tp.Int:
111             args[0] = tp.Cast(args[0], tp.Float)
112
113         # TODO llvmlite
114         llvm_f = cge.module.llvm.declare_intrinsic(self.intr,
115             ↪ [args[0].llvmType(cge.module)])
116         result = cge.builder.call(llvm_f, [args[0].translate(cge)])
117         return result
118
119 class Exp(Log):
120     py_func = math.exp
121     intr = 'llvm.exp'
122     arg_names = ['x']
123
124
125 class Pow(Intrinsic):
126     py_func = pow
127     arg_names = ['x', 'y']
128
129     def getReturnType(self, args, kw_args):
130         if args[0].type == tp.Int and args[1].type == tp.Int:
131             return tp.Int
132         else:
133             return tp.Float
134
135     def castResultInt(self, args):
136         return (isinstance(args[0], tp.Cast)
137             and args[0].obj.type == tp.Int
138             and args[1].type == tp.Int)
```

Appendix C. STELLA Source Code

```
139
140 def call(self, cge, args, kw_args):
141     # TODO unify with bytecode.BINARY_POWER
142
143     # llvm.pow[i]'s first argument always has to be float
144     arg = args[0]
145     if arg.type == tp.Int:
146         args[0] = tp.Cast(arg, tp.Float)
147
148     if args[1].type == tp.Int:
149         # powi takes a i32 argument
150         power = cge.builder.trunc(
151             args[1].translate(cge),
152             tp.tp_int32,
153             '(i32)' +
154             args[1].name)
155     else:
156         power = args[1].translate(cge)
157
158     if args[1].type == tp.Int:
159         intr = 'llvm.powi'
160     else:
161         intr = 'llvm.pow'
162
163     llvm_f = cge.module.llvm.declare_intrinsic(intr,
164     ↪ [args[0].llvmType(cge.module)])
165     pow_result = cge.builder.call(llvm_f, [args[0].translate(cge),
166     ↪ power])
167
168     if self.castResultInt(args):
169         # cast back to an integer
170         result = cge.builder.fptosi(pow_result,
171     ↪ tp.Int.llvmType(cge.module))
172     else:
173         result = pow_result
174
175     return result
176
177 class MathPow(Pow):
178     py_func = math.pow
179
180     def castResultInt(self, args):
```

Appendix C. STELLA Source Code

```
179         return False
180
181     def getReturnType(self, args, kw_args):
182         return tp.Float
183
184
185 class Sqrt(Log):
186     py_func = math.sqrt
187     intr = 'llvm.sqrt'
188     arg_names = ['x']
189
190
191 class Exception(Intrinsic):
192     """
193     NoOP since Exceptions are not objects in Stella.
194     """
195     py_func = builtins.Exception
196     arg_names = ['msg']
197
198     def getReturnType(self, args, kw_args):
199         return tp.Void
200
201     def getResult(self, func):
202         return Register(func)
203
204     def call(self, cge, args, kw_args):
205         pass
206
207     @staticmethod
208     def is_a(item):
209         return isinstance(item, type) and issubclass(item,
210             ↪ builtins.Exception)
211
212
213 class Cast(Intrinsic):
214     """
215     Abstract cast
216     """
217     arg_names = ['x']
218
219     def __init__(self):
220         super().__init__()
```

Appendix C. STELLA Source Code

```
221     def getReturnType(self, args, kw_args):
222         return self.stella_type
223
224     def call(self, cge, args, kw_args):
225         obj = args[0]
226         cast = tp.Cast(obj, self.stella_type)
227         return cast.translate(cge)
228
229
230 class Float(Cast):
231     stella_type = tp.Float
232     py_func = stella_type.type_
233
234
235 class Int(Cast):
236     stella_type = tp.Int
237     py_func = stella_type.type_
238
239
240 class Bool(Cast):
241     stella_type = tp.Bool
242     py_func = stella_type.type_
243
244
245 class Tuple(Intrinsic):
246     py_func = tuple
247     arg_names = ['iterable']
248
249     def __init__(self):
250         super().__init__()
251
252     def getReturnType(self, args, kw_args):
253         type_ = args[0].type.dereference()
254         assert isinstance(type_, tp.Subscriptable)
255         # there are no Nd tuples, accept only 1d
256         assert not isinstance(type_.shape, list)
257
258         ttype = tp.TupleType([type_.type_] * type_.shape)
259         return ttype
260
261     def call(self, cge, args, kw_args):
262         in_type = args[0].type.dereference()
263
```


Appendix C. STELLA Source Code

```
264     init = [ll.Constant(in_type.type_.llvmType(cge.module), None)] *
        ↪ in_type.shape
265
266     llvm = ll.Constant.literal_struct(init)
267
268     for i in range(in_type.shape):
269         val = in_type.loadSubscript(cge, args[0], tp.Const(i))
270         llvm = cge.builder.insert_value(llvm, val, i)
271
272     return llvm
273
274
275 casts = (int, float, bool, tuple)
276
277
278 def is_extra(item):
279     """Allow more flexible intrinsics detection than simple equality.
280
281     The example is Exception, where we want to catch subtypes as well.
282     """
283     # 'numpy.ndarray' in 'tuple' is broken, so work around it
284     if isinstance(item, np.ndarray):
285         return False
286     return any([f(item) for f in [Exception.is_a]]) or item in casts
287
288
289 def get(func):
290     if func in func2klass:
291         return func2klass[func]
292     elif Exception.is_a(func):
293         return Exception
294     else:
295         return None
296
297
298 func2klass = {}
299
300
301 # Get all concrete subclasses of Intrinsic and register them
302 for name in dir(sys.modules[__name__]):
303     klass = sys.modules[__name__].__dict__[name]
304     try:
```

Appendix C. STELLA Source Code

```
305     if isinstance(klass, Intrinsic) and len(klass.__abstractmethods__)
        ↪ == 0 and \
306         klass.py_func is not None:
307         func2klass[klass.py_func] = klass
308     except TypeError:
309         pass
```

C.5 stella/_version.py

```
1  # This file helps to compute a version number in source trees obtained
    ↪ from
2  # git-archive tarball (such as those provided by githubs download-from-tag
3  # feature). Distribution tarballs (built by setup.py sdist) and build
4  # directories (produced by setup.py build) will contain a much shorter
    ↪ file
5  # that just contains the computed version number.
6
7  # This file is released into the public domain. Generated by
8  # versioneer-0.15 (https://github.com/warner/python-versioneer)
9
10 import errno
11 import os
12 import re
13 import subprocess
14 import sys
15
16
17 def get_keywords():
18     # these strings will be replaced by git during git-archive.
19     # setup.py/versioneer.py will grep for the variable names, so they
    ↪ must
20     # each be defined on a line of their own. _version.py will just call
21     # get_keywords().
22     git_refnames = "$Format:%d$"
23     git_full = "$Format:%H$"
24     keywords = {"refnames": git_refnames, "full": git_full}
25     return keywords
26
27
```

Appendix C. STELLA Source Code

```
28 class VersioneerConfig:
29     pass
30
31
32 def get_config():
33     # these strings are filled in when 'setup.py versioneer' creates
34     # _version.py
35     cfg = VersioneerConfig()
36     cfg.VCS = "git"
37     cfg.style = "pep440"
38     cfg.tag_prefix = ""
39     cfg.parentdir_prefix = "stella-"
40     cfg.versionfile_source = "stella/_version.py"
41     cfg.verbose = False
42     return cfg
43
44
45 class NotThisMethod(Exception):
46     pass
47
48
49 LONG_VERSION_PY = {}
50 HANDLERS = {}
51
52
53 def register_vcs_handler(vcs, method): # decorator
54     def decorate(f):
55         if vcs not in HANDLERS:
56             HANDLERS[vcs] = {}
57             HANDLERS[vcs][method] = f
58         return f
59     return decorate
60
61
62 def run_command(commands, args, cwd=None, verbose=False,
63     ↪ hide_stderr=False):
64     assert isinstance(commands, list)
65     p = None
66     for c in commands:
67         try:
68             dispcmd = str([c] + args)
69             # remember shell=False, so use git.cmd on windows, not just
70             ↪ git
```

Appendix C. STELLA Source Code

```
69         p = subprocess.Popen([c] + args, cwd=cwd,
70                               ↪ stdout=subprocess.PIPE,
71                                   stderr=(subprocess.PIPE if hide_stderr
72                                           else None))
73         break
74     except EnvironmentError:
75         e = sys.exc_info()[1]
76         if e.errno == errno.ENOENT:
77             continue
78         if verbose:
79             print("unable to run %s" % dispcmd)
80             print(e)
81         return None
82     else:
83         if verbose:
84             print("unable to find command, tried %s" % (commands,))
85         return None
86     stdout = p.communicate()[0].strip()
87     if sys.version_info[0] >= 3:
88         stdout = stdout.decode()
89     if p.returncode != 0:
90         if verbose:
91             print("unable to run %s (error)" % dispcmd)
92         return None
93     return stdout
94
95 def versions_from_parentdir(parentdir_prefix, root, verbose):
96     # Source tarballs conventionally unpack into a directory that includes
97     # both the project name and a version string.
98     dirname = os.path.basename(root)
99     if not dirname.startswith(parentdir_prefix):
100         if verbose:
101             print("guessing rootdir is '%s', but '%s' doesn't start with "
102                  "prefix '%s'" % (root, dirname, parentdir_prefix))
103         raise NotThisMethod("rootdir doesn't start with parentdir_prefix")
104     return {"version": dirname[len(parentdir_prefix):],
105           "full-revisionid": None,
106           "dirty": False, "error": None}
107
108
109 @register_vcs_handler("git", "get_keywords")
110 def git_get_keywords(versionfile_abs):
```

Appendix C. STELLA Source Code

```
111 # the code embedded in _version.py can just fetch the value of these
112 # keywords. When used from setup.py, we don't want to import
    ↪ _version.py,
113 # so we do it with a regexp instead. This function is not used from
114 # _version.py.
115 keywords = {}
116 try:
117     f = open(versionfile_abs, "r")
118     for line in f.readlines():
119         if line.strip().startswith("git_refnames ="):
120             mo = re.search(r'=\s*"(.*)"', line)
121             if mo:
122                 keywords["refnames"] = mo.group(1)
123         if line.strip().startswith("git_full ="):
124             mo = re.search(r'=\s*"(.*)"', line)
125             if mo:
126                 keywords["full"] = mo.group(1)
127     f.close()
128 except EnvironmentError:
129     pass
130 return keywords
131
132
133 @register_vcs_handler("git", "keywords")
134 def git_versions_from_keywords(keywords, tag_prefix, verbose):
135     if not keywords:
136         raise NotThisMethod("no keywords at all, weird")
137     refnames = keywords["refnames"].strip()
138     if refnames.startswith("$Format"):
139         if verbose:
140             print("keywords are unexpanded, not using")
141         raise NotThisMethod("unexpanded keywords, not a git-archive
    ↪ tarball")
142     refs = set([r.strip() for r in refnames.strip "()".split(",")])
143     # starting in git-1.8.3, tags are listed as "tag: foo-1.0" instead of
144     # just "foo-1.0". If we see a "tag: " prefix, prefer those.
145     TAG = "tag: "
146     tags = set([r[len(TAG):] for r in refs if r.startswith(TAG)])
147     if not tags:
148         # Either we're using git < 1.8.3, or there really are no tags. We
    ↪ use
149         # a heuristic: assume all version tags have a digit. The old git
    ↪ %d
```

Appendix C. STELLA Source Code

```
150     # expansion behaves like git log --decorate=short and strips out
    ↪ the
151     # refs/heads/ and refs/tags/ prefixes that would let us
    ↪ distinguish
152     # between branches and tags. By ignoring refnames without digits,
    ↪ we
153     # filter out many common branch names like "release" and
154     # "stabilization", as well as "HEAD" and "master".
155     tags = set([r for r in refs if re.search(r'\d', r)])
156     if verbose:
157         print("discarding '%s', no digits" % ", ".join(refs-tags))
158     if verbose:
159         print("likely tags: %s" % ", ".join(sorted(tags)))
160     for ref in sorted(tags):
161         # sorting will prefer e.g. "2.0" over "2.0rc1"
162         if ref.startswith(tag_prefix):
163             r = ref[len(tag_prefix):]
164             if verbose:
165                 print("picking %s" % r)
166             return {"version": r,
167                    "full-revisionid": keywords["full"].strip(),
168                    "dirty": False, "error": None
169                    }
170     # no suitable tags, so version is "0+unknown", but full hex is still
    ↪ there
171     if verbose:
172         print("no suitable tags, using unknown + full revision id")
173     return {"version": "0+unknown",
174            "full-revisionid": keywords["full"].strip(),
175            "dirty": False, "error": "no suitable tags"}
176
177
178 @register_vcs_handler("git", "pieces_from_vcs")
179 def git_pieces_from_vcs(tag_prefix, root, verbose,
    ↪ run_command=run_command):
180     # this runs 'git' from the root of the source tree. This only gets
    ↪ called
181     # if the git-archive 'subst' keywords were *not* expanded, and
182     # _version.py hasn't already been rewritten with a short version
    ↪ string,
183     # meaning we're inside a checked out source tree.
184
185     if not os.path.exists(os.path.join(root, ".git")):
```

Appendix C. STELLA Source Code

```
186     if verbose:
187         print("no .git in %s" % root)
188         raise NotThisMethod("no .git directory")
189
190     GITS = ["git"]
191     if sys.platform == "win32":
192         GITS = ["git.cmd", "git.exe"]
193     # if there is a tag, this yields TAG-NUM-gHEX[-dirty]
194     # if there are no tags, this yields HEX[-dirty] (no NUM)
195     describe_out = run_command(GITS, ["describe", "--tags", "--dirty",
196                                     "--always", "--long"],
197                                cwd=root)
198     # --long was added in git-1.5.5
199     if describe_out is None:
200         raise NotThisMethod("'git describe' failed")
201     describe_out = describe_out.strip()
202     full_out = run_command(GITS, ["rev-parse", "HEAD"], cwd=root)
203     if full_out is None:
204         raise NotThisMethod("'git rev-parse' failed")
205     full_out = full_out.strip()
206
207     pieces = {}
208     pieces["long"] = full_out
209     pieces["short"] = full_out[:7] # maybe improved later
210     pieces["error"] = None
211
212     # parse describe_out. It will be like TAG-NUM-gHEX[-dirty] or
213     # ↪ HEX[-dirty]
214     # TAG might have hyphens.
215     git_describe = describe_out
216
217     # look for -dirty suffix
218     dirty = git_describe.endswith("-dirty")
219     pieces["dirty"] = dirty
220     if dirty:
221         git_describe = git_describe[:git_describe.rindex("-dirty")]
222
223     # now we have TAG-NUM-gHEX or HEX
224
225     if "-" in git_describe:
226         # TAG-NUM-gHEX
227         mo = re.search(r'^(.+)-(\d+)-g([0-9a-f]+)$', git_describe)
228         if not mo:
```

Appendix C. STELLA Source Code

```
228         # unparseable. Maybe git-describe is misbehaving?
229         pieces["error"] = ("unable to parse git-describe output: '%s'"
230                           % describe_out)
231         return pieces
232
233     # tag
234     full_tag = mo.group(1)
235     if not full_tag.startswith(tag_prefix):
236         if verbose:
237             fmt = "tag '%s' doesn't start with prefix '%s'"
238             print(fmt % (full_tag, tag_prefix))
239             pieces["error"] = ("tag '%s' doesn't start with prefix '%s'"
240                               % (full_tag, tag_prefix))
241         return pieces
242     pieces["closest-tag"] = full_tag[len(tag_prefix):]
243
244     # distance: number of commits since tag
245     pieces["distance"] = int(mo.group(2))
246
247     # commit: short hex revision ID
248     pieces["short"] = mo.group(3)
249
250 else:
251     # HEX: no tags
252     pieces["closest-tag"] = None
253     count_out = run_command(GITS, ["rev-list", "HEAD", "--count"],
254                             cwd=root)
255     pieces["distance"] = int(count_out) # total number of commits
256
257 return pieces
258
259
260 def plus_or_dot(pieces):
261     if "+" in pieces.get("closest-tag", ""):
262         return "."
263     return "+"
264
265
266 def render_pep440(pieces):
267     # now build up version string, with post-release "local version
268     # identifier". Our goal: TAG[+DISTANCE.gHEX[.dirty]] . Note that if
269     # you
270     # get a tagged build and then dirty it, you'll get TAG+0.gHEX.dirty
```


Appendix C. STELLA Source Code

```
270
271 # exceptions:
272 # 1: no tags. git_describe was just HEX.
    ↪ 0+untagged.DISTANCE.gHEX[.dirty]
273
274 if pieces["closest-tag"]:
275     rendered = pieces["closest-tag"]
276     if pieces["distance"] or pieces["dirty"]:
277         rendered += plus_or_dot(pieces)
278         rendered += "%d.g%s" % (pieces["distance"], pieces["short"])
279         if pieces["dirty"]:
280             rendered += ".dirty"
281     else:
282         # exception #1
283         rendered = "0+untagged.%d.g%s" % (pieces["distance"],
284                                           pieces["short"])
285         if pieces["dirty"]:
286             rendered += ".dirty"
287     return rendered
288
289
290 def render_pep440_pre(pieces):
291     # TAG[.post.devDISTANCE] . No -dirty
292
293     # exceptions:
294     # 1: no tags. 0.post.devDISTANCE
295
296     if pieces["closest-tag"]:
297         rendered = pieces["closest-tag"]
298         if pieces["distance"]:
299             rendered += ".post.dev%d" % pieces["distance"]
300     else:
301         # exception #1
302         rendered = "0.post.dev%d" % pieces["distance"]
303     return rendered
304
305
306 def render_pep440_post(pieces):
307     # TAG[.postDISTANCE[.dev0]+gHEX] . The ".dev0" means dirty. Note that
308     # .dev0 sorts backwards (a dirty tree will appear "older" than the
309     # corresponding clean one), but you shouldn't be releasing software
    ↪ with
310     # -dirty anyways.
```

Appendix C. STELLA Source Code

```
311
312 # exceptions:
313 # 1: no tags. 0.postDISTANCE[.dev0]
314
315 if pieces["closest-tag"]:
316     rendered = pieces["closest-tag"]
317     if pieces["distance"] or pieces["dirty"]:
318         rendered += ".post%d" % pieces["distance"]
319         if pieces["dirty"]:
320             rendered += ".dev0"
321         rendered += plus_or_dot(pieces)
322         rendered += "g%s" % pieces["short"]
323 else:
324     # exception #1
325     rendered = "0.post%d" % pieces["distance"]
326     if pieces["dirty"]:
327         rendered += ".dev0"
328     rendered += "+g%s" % pieces["short"]
329 return rendered
330
331
332 def render_pep440_old(pieces):
333     # TAG[.postDISTANCE[.dev0]] . The ".dev0" means dirty.
334
335     # exceptions:
336     # 1: no tags. 0.postDISTANCE[.dev0]
337
338     if pieces["closest-tag"]:
339         rendered = pieces["closest-tag"]
340         if pieces["distance"] or pieces["dirty"]:
341             rendered += ".post%d" % pieces["distance"]
342             if pieces["dirty"]:
343                 rendered += ".dev0"
344     else:
345         # exception #1
346         rendered = "0.post%d" % pieces["distance"]
347         if pieces["dirty"]:
348             rendered += ".dev0"
349     return rendered
350
351
352 def render_git_describe(pieces):
353     # TAG[-DISTANCE-gHEX][-dirty], like 'git describe --tags --dirty
```

Appendix C. STELLA Source Code

```
354     # --always'
355
356     # exceptions:
357     # 1: no tags. HEX[-dirty] (note: no 'g' prefix)
358
359     if pieces["closest-tag"]:
360         rendered = pieces["closest-tag"]
361         if pieces["distance"]:
362             rendered += "-%d-g%s" % (pieces["distance"], pieces["short"])
363     else:
364         # exception #1
365         rendered = pieces["short"]
366     if pieces["dirty"]:
367         rendered += "-dirty"
368     return rendered
369
370
371 def render_git_describe_long(pieces):
372     # TAG-DISTANCE-gHEX[-dirty], like 'git describe --tags --dirty
373     # --always -long'. The distance/hash is unconditional.
374
375     # exceptions:
376     # 1: no tags. HEX[-dirty] (note: no 'g' prefix)
377
378     if pieces["closest-tag"]:
379         rendered = pieces["closest-tag"]
380         rendered += "-%d-g%s" % (pieces["distance"], pieces["short"])
381     else:
382         # exception #1
383         rendered = pieces["short"]
384     if pieces["dirty"]:
385         rendered += "-dirty"
386     return rendered
387
388
389 def render(pieces, style):
390     if pieces["error"]:
391         return {"version": "unknown",
392                 "full-revisionid": pieces.get("long"),
393                 "dirty": None,
394                 "error": pieces["error"]}
395
396     if not style or style == "default":
```

Appendix C. STELLA Source Code

```
397     style = "pep440" # the default
398
399     if style == "pep440":
400         rendered = render_pep440(pieces)
401     elif style == "pep440-pre":
402         rendered = render_pep440_pre(pieces)
403     elif style == "pep440-post":
404         rendered = render_pep440_post(pieces)
405     elif style == "pep440-old":
406         rendered = render_pep440_old(pieces)
407     elif style == "git-describe":
408         rendered = render_git_describe(pieces)
409     elif style == "git-describe-long":
410         rendered = render_git_describe_long(pieces)
411     else:
412         raise ValueError("unknown style '%s'" % style)
413
414     return {"version": rendered, "full-revisionid": pieces["long"],
415           "dirty": pieces["dirty"], "error": None}
416
417
418 def get_versions():
419     # I am in _version.py, which lives at ROOT/VERSIONFILE_SOURCE. If we
420     ↪ have
421     # __file__, we can work backwards from there to the root. Some
422     # py2exe/bbfreeze/non-CPython implementations don't do __file__, in
423     ↪ which
424     # case we can only use expanded keywords.
425
426     cfg = get_config()
427     verbose = cfg.verbose
428
429     try:
430         return git_versions_from_keywords(get_keywords(), cfg.tag_prefix,
431                                         verbose)
432     except NotThisMethod:
433         pass
434
435     try:
436         root = os.path.realpath(__file__)
437         # versionfile_source is the relative path from the top of the
438         ↪ source
439         # tree (where the .git directory might live) to this file. Invert
```

Appendix C. STELLA Source Code

```
437     # this to find the root from __file__.  
438     for i in cfg.versionfile_source.split('/'):   
439         root = os.path.dirname(root)   
440     except NameError:   
441         return {"version": "0+unknown", "full-revisionid": None,   
442                 "dirty": None,   
443                 "error": "unable to find root of source tree"}   
444   
445     try:   
446         pieces = git_pieces_from_vcs(cfg.tag_prefix, root, verbose)   
447         return render(pieces, cfg.style)   
448     except NotThisMethod:   
449         pass   
450   
451     try:   
452         if cfg.parentdir_prefix:   
453             return versions_from_parentdir(cfg.parentdir_prefix, root,   
454                                           ↪ verbose)   
455     except NotThisMethod:   
456         pass   
457     return {"version": "0+unknown", "full-revisionid": None,   
458             "dirty": None,   
459             "error": "unable to compute version"}
```

C.6 stella/codegen.py

```
1 #!/usr/bin/env python   
2 # Copyright 2013-2015 David Mohr   
3 #   
4 # Licensed under the Apache License, Version 2.0 (the "License");   
5 # you may not use this file except in compliance with the License.   
6 # You may obtain a copy of the License at   
7 #   
8 #     http://www.apache.org/licenses/LICENSE-2.0   
9 #   
10 # Unless required by applicable law or agreed to in writing, software   
11 # distributed under the License is distributed on an "AS IS" BASIS,   
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

Appendix C. STELLA Source Code

```
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 import llvmlite.ir as ll
17 import llvmlite.binding as llvm
18
19 import logging
20 import time
21
22 from . import tp
23 from . import ir
24 from . import exc
25
26
27 class CGEnv(object):
28     module = None
29     builder = None
30
31
32 class Program(object):
33     def __init__(self, module):
34         llvm.initialize()
35         llvm.initialize_native_target()
36         llvm.initialize_native_asmprinter()
37
38         self.module = module
39         self.module.translate()
40
41         self.cge = CGEnv()
42         self.cge.module = module
43
44         self.llvm = self.makeStub()
45
46         for _, func in self.module.namestore.all(ir.Function):
47             self.blockAndCode(func)
48
49         self.target_machine =
50             ↪ llvm.Target.from_default_triple().create_target_machine()
51
52         logging.debug("Verifying... ")
53         self._llmod = None
54
55     def llmod(self):
```

Appendix C. STELLA Source Code

```
55     if not self._llmod:
56         self._llmod = llvm.parse_assembly(str(self.module.llvm))
57     return self._llmod
58
59     def blockAndCode(self, impl):
60         func = impl.llvm
61         # create blocks
62         bb = func.append_basic_block("entry")
63
64         for bc in impl.bytecodes:
65             if bc.discard:
66                 impl.remove(bc)
67                 impl.log.debug("BLOCK skipped {0}".format(bc))
68                 continue
69
70                 newblock = ''
71                 if bc in impl.incoming_jumps:
72                     assert not bc.block
73                     bc.block = func.append_basic_block(str(bc.loc))
74                     bb = bc.block
75                     newblock = ' NEW BLOCK (' + str(bc.loc) + ')'
76                 else:
77                     bc.block = bb
78                 impl.log.debug("BLOCK'D {0}{1}".format(bc, newblock))
79
80         for ext_module in self.module.getExternalModules():
81             ext_module.translate(self.module.llvm)
82
83         impl.log.debug("Printing all bytecodes:")
84         impl.bytecodes.printAll(impl.log)
85
86         impl.log.debug("Emitting code:")
87         bb = None
88         cge = self.cge
89         for bc in impl.bytecodes:
90             try:
91                 if bb != bc.block:
92                     # new basic block, use a new builder
93                     cge.builder = ll.IRBuilder(bc.block)
94
95                 if bc.reachable:
96                     bc.translate(cge)
97                 impl.log.debug("TRANS'D {0}".format(bc.locStr()))
```

Appendix C. STELLA Source Code

```
98         else:
99             # eliminate unreachable code, which may occur in the
            ↪ middle of a function
100             impl.log.debug("UNREACH {}".format(bc.locStr()))
101         except exc.StellaException as e:
102             e.addDebug(bc.debuginfo)
103             raise
104
105     def makeStub(self):
106         impl = self.module.entry
107         func_tp = ll.FunctionType(impl.result.type.llvmType(self.module),
            ↪ [])
108         func = ll.Function(self.module.llvm, func_tp,
            ↪ name=str(impl.function)+'__stub__')
109         bb = func.append_basic_block("entry")
110         builder = ll.IRBuilder(bb)
111         self.cge.builder = builder
112
113         for name, var in self.module.namestore.all(ir.GlobalVariable):
114             var.translate(self.cge)
115
116         llvm_args = [arg.translate(self.cge) for arg in
            ↪ self.module.entry_args]
117
118         call = builder.call(impl.llvm, llvm_args)
119
120         if impl.result.type is tp.Void:
121             builder.ret_void()
122         else:
123             builder.ret(call)
124         return func
125
126     def elapsed(self):
127         if self.start is None or self.end is None:
128             return None
129         return self.end - self.start
130
131     def optimize(self, opt):
132         if opt is not None:
133             logging.warn("Running optimizations level {0}...
            ↪ ".format(opt))
134
```


Appendix C. STELLA Source Code

```
135         # TODO was build_pass_managers(tm, opt=opt,
136         ↪ loop_vectorize=True, fpm=False)
137         pmb = llvm.create_pass_manager_builder()
138         pmb.opt_level = opt
139         pm = llvm.create_module_pass_manager()
140         pmb.populate(pm)
141         pm.run(self.llmod())
142
143     def destruct(self):
144         self.module.destruct()
145         del self.module
146
147     def __del__(self):
148         logging.debug("DEL {}: {}".format(repr(self), hasattr(self,
149         ↪ 'module')))
150
151     def run(self, stats):
152         logging.debug("Preparing execution...")
153
154         import ctypes
155         import llvmlite
156         import os
157
158         _lib_dir = os.path.dirname(llvm.ffi.__file__)
159         clib = ctypes.CDLL(os.path.join(_lib_dir,
160         ↪ llvmlite.utils.get_library_name()))
161         # Direct access as below mangles the name
162         # f = clib.__powidf2
163         f = getattr(clib, '__powidf2')
164         llvm.add_symbol('__powidf2', ctypes.cast(f,
165         ↪ ctypes.c_void_p).value)
166
167     with llvm.create_mcjit_compiler(self.llmod(), self.target_machine)
168     ↪ as ee:
169         ee.finalize_object()
170
171         entry = self.module.entry
172         ret_type = entry.result.type
173
174         logging.info("running {0}{1}".format(entry,
175         ↪ list(zip(entry.type._arg_types,
```

Appendix C. STELLA Source Code

```
171
                                                    → self.module.entry_args))
172
173     entry_ptr =
174         → ee.get_pointer_to_global(self.llmod().get_function(self.llvm.name))
175     ret_ctype = entry.result.type.Ctype()
176     if ret_type.on_heap:
177         ret_ctype = ctypes.POINTER(ret_ctype)
178     cfunc = ctypes.CFUNCTYPE(ret_ctype)(entry_ptr)
179
180     time_start = time.time()
181     retval = cfunc()
182     stats['elapsed'] = time.time() - time_start
183
184     for arg in self.module.entry_args:
185         arg.ctype2Python(self.cge) # may be a no-op if not necessary
186
187     retval = ret_type.unpack(retval)
188
189     logging.debug("Returning...")
190     self.destruct()
191
192     return retval
193
194     def getAssembly(self):
195         return self.target_machine.emit_assembly(self.llmod())
196
197     def getLlvmIR(self):
198         ret = self.module.getLlvmIR()
199
200         logging.debug("Returning...")
201         self.destruct()
202
203         return ret
```

C.7 stella/ir.py

```
1 # Copyright 2013-2015 David Mohr
2 #
```

Appendix C. STELLA Source Code

```
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import logging
15 import weakref
16 import types
17 import sys
18 from abc import ABCMeta, abstractmethod
19 import ctypes
20 import inspect
21
22 import llvmlite.ir as ll
23 import llvmlite.binding as llvm
24
25 from . import exc
26 from . import utils
27 from . import tp
28 from .storage import Register, StackLoc, GlobalVariable
29 from . import intrinsics
30
31
32 @utils.linkedlist
33 class IR(metaclass=ABCMeta):
34     args = None
35     bc_args = None
36     stack_bc = None
37     const_arg = None
38     result = None
39     debuginfo = None
40     llvm = None
41     block = None
42     loc = ''
43     discard = False
44     reachable = True
45
```

Appendix C. STELLA Source Code

```
46     def __init__(self, func, debuginfo):
47         self.debuginfo = debuginfo
48         self.bc_args = []
49         self.args = []
50
51     def addConst(self, arg):
52         self.const_arg = tp.wrapValue(arg)
53
54     def addRawArg(self, arg):
55         raise exc.UnimplementedError("{0}.addRawArg() is not
56         ↪ implemented".format(
57             self.__class__.__name__))
58
59     def addLocalName(self, func, name):
60         self.bc_args.append(func.getStackLoc(name))
61         # TODO: is a base implementation needed??
62
63     def cast(self, cge):
64         for arg in self.args:
65             if isinstance(arg, tp.Cast):
66                 arg.translate(cge)
67
68     def grab_stack(self):
69         """
70         ↪ Call first during type evaluation. Gets the results from the stack
71         and
72         adds them to args.
73         """
74         if self.stack_bc:
75             self.args = []
76             for arg in self.stack_bc:
77                 # TODO should arg.result always be a list?
78                 if isinstance(arg.result, list):
79                     result = arg.result.pop()
80                     # keep the result, we need it for retyping
81                     arg.result.insert(0, result)
82                 else:
83                     result = arg.result
84                     result.bc = arg
85                     self.args.append(result)
86
87     @abstractmethod
88     def stack_eval(self, func, stack):
```

Appendix C. STELLA Source Code

```
87         pass
88
89     @abstractmethod
90     def translate(self, cge):
91         pass
92
93     @abstractmethod
94     def type_eval(self, func):
95         pass
96
97     def __str__(self):
98         if self.discard:
99             b = '('
100            e = ')'
101        else:
102            b = e = ''
103
104        if hasattr(self, '_str_args'):
105            args = self._str_args
106        else:
107            args = ", ".join([str(v) for v in self.args])
108
109        return "{0}{1} {2} {3}{4}".format(
110            b,
111            self.__class__.__name__,
112            self.result,
113            args,
114            e)
115
116     def __repr__(self):
117         # TODO: are there reasons not to do this?
118         return self.__str__()
119
120     def locStr(self):
121         return "{0:3s} {1}".format(str(self.loc), str(self))
122
123     def equivalent(self, other):
124         """Equality but location independent.
125
126         This method may need to be overridden by the concrete
127         implementation.
128         """
129         return type(self) == type(other) and self.args == other.args
```

Appendix C. STELLA Source Code

```
129
130
131 class PhiNode(IR):
132     def __init__(self, func, debuginfo):
133         super().__init__(func, debuginfo)
134
135         self.blocks = []
136         self.stacked = False
137         self.stack_bc = []
138
139     def stack_eval(self, func, stack):
140         tos = stack.peek()
141
142         # sanity check: either there is always a tos or never
143         if self.stacked:
144             if tos and not self.result:
145                 raise exc.StellaException("Invalid bytecode sequence:
146                 ↪ unexpected tos")
147             if not tos and self.result:
148                 raise exc.StellaException("Invalid bytecode sequence:
149                 ↪ expected tos")
150
151         if tos:
152             if not self.result:
153                 self.result = Register(func)
154                 self.stack_bc.append(stack.pop())
155                 self.blocks.append(self.stack_bc[-1])
156                 stack.push(self)
157
158             self.stacked = True
159
160     def type_eval(self, func):
161         self.grab_stack()
162         if len(self.args) == 0:
163             return
164         for arg in self.args:
165             self.result.unify_type(arg.type, self.debuginfo)
166
167     def translate(self, cge):
168         if len(self.args) == 0:
169             return
170         phi = cge.builder.phi(self.result.llvmType(cge.module),
171                               ↪ self.result.name)
```

Appendix C. STELLA Source Code

```
169         for arg in self.args:
170             phi.add_incoming(arg.llvm, arg.bc.block)
171
172         self.result.llvm = phi
173
174
175 class Scope(object):
176     """
177     Used to add scope functionality to an object
178     """
179     def __init__(self, parent):
180         self.parent = parent
181         self.register_n = 0
182         self.registers = dict()
183         self.stacklocs = dict()
184
185     def newRegisterName(self):
186         n = str(self.register_n)
187         self.register_n += 1
188         return n
189
190     def getOrNewRegister(self, name):
191         if name not in self.registers:
192             self.registers[name] = Register(self, name)
193         return self.registers[name]
194
195     def getRegister(self, name):
196         if name not in self.registers:
197             raise exc.UndefinedError(name)
198         return self.registers[name]
199
200     def getOrNewStackLoc(self, name):
201         isnew = False
202         if name not in self.stacklocs:
203             self.stacklocs[name] = StackLoc(self, name)
204             isnew = True
205         return (self.stacklocs[name], isnew)
206
207     def getStackLoc(self, name):
208         if name not in self.stacklocs:
209             raise exc.UndefinedError(name)
210         return self.stacklocs[name]
211
```

Appendix C. STELLA Source Code

```
212
213 class Globals(object):
214     def __init__(self):
215         self.store = dict()
216
217     def __setitem__(self, key, value):
218         # TODO: should uniqueness be enforced here?
219         assert key not in self.store
220         self.store[key] = value
221
222     def __getitem__(self, key):
223         if key not in self.store:
224             raise exc.UndefinedGlobalError(key)
225         return self.store[key]
226
227     def all(self, tp=None):
228         if tp is None:
229             return self.store.items()
230         else:
231             return [(k, v) for k, v in self.store.items() if isinstance(v,
232                 → tp)]
233
234 class Module(object):
235     i = 0
236
237     def __init__(self):
238         super().__init__()
239         self._todo = []
240         self.entry = None
241         self.llvm = None
242         self.namestore = Globals()
243         self.external_modules = dict()
244         self._cleanup = []
245         self.log = logging.getLogger(str(self))
246
247     def _getFunction(self, item):
248         if isinstance(item, tp.FunctionType):
249             f_type = item
250         else:
251             f_type = tp.get(item)
252         try:
253             f = self.namestore[f_type.fq]
```


Appendix C. STELLA Source Code

```
254     except exc.UndefinedGlobalError:
255         f = Function(f_type, self)
256         self.namestore[f_type.fq] = f
257
258     return f
259
260 def getFunctionRef(self, item):
261     if isinstance(item, Function):
262         f = item
263     else:
264         f = self._getFunction(item)
265     if f.type_bound:
266         return BoundFunctionRef(f)
267     else:
268         return FunctionRef(f)
269
270 def makeEntry(self, funcref, args):
271     assert self.entry is None
272     self.entry = funcref
273     self.entry_args = args
274
275 def getExternalModule(self, mod):
276     if mod not in self.external_modules:
277         self.external_modules[mod] = ExtModule(mod)
278     return self.external_modules[mod]
279
280 def getExternalModules(self):
281     return self.external_modules.values()
282
283 def _wrapPython(self, key, item, module=None):
284     if isinstance(item, (types.FunctionType,
285                         types.BuiltinFunctionType)) or
286         ↪ intrinsics.is_extra(item):
287         intrinsic = intrinsics.get(item)
288
289         if intrinsic:
290             wrapped = intrinsic
291         elif module and hasattr(module, '__file__') and
292             ↪ module.__file__[-3:] == '.so':
293             ext_module = self.getExternalModule(module)
294             wrapped = ext_module.getFunctionRef(item)
295         else:
296             f = self._getFunction(item)
```

Appendix C. STELLA Source Code

```
295         wrapped = self.getFunctionRef(f)
296     elif isinstance(item, types.ModuleType):
297         # no need to wrap it, it will be used with self.loadExt()
298         wrapped = item
299     else:
300         # Assume it is a global variable
301         # TODO: is this safe? How do I catch types that aren't
302         #       ↪ supported
303         # without listing all valid types?
304         wrapped = GlobalVariable(key, item)
305
306     return wrapped
307
308 def loadExt(self, module, attr):
309     assert isinstance(module, types.ModuleType) and type(attr) == str
310
311     key = module.__name__ + '.' + attr
312     try:
313         wrapped = self.namestore[key]
314     except exc.UndefinedGlobalError as e:
315         if attr not in module.__dict__:
316             raise e
317
318         item = module.__dict__[attr]
319         if hasattr(module, '__file__') and module.__file__[-3:] ==
320             ↪ '.so' and \
321             type(item) == type(print):
322             # external module
323             pass
324         elif not isinstance(item, (types.FunctionType, type(print))):
325             raise exc.UnimplementedError(
326                 "Currently only Functions can be imported (not
327                 ↪ {0})".format(type(item)))
328         wrapped = self._wrapPython(key, item, module)
329         self.namestore[key] = wrapped
330
331     # instrinsic check: we need a new instance for every call!
332     # TODO: see also self.loadGlobal()
333     if inspect.isclass(wrapped) and isinstance(wrapped,
334         ↪ intrinsics.Intrinsic):
335         return wrapped()
336     else:
337         return wrapped
```

Appendix C. STELLA Source Code

```
334
335 def loadGlobal(self, func, key):
336     try:
337         wrapped = self.namestore[key]
338         if isinstance(wrapped, Function):
339             wrapped = self.getFunctionRef(wrapped)
340     except exc.UndefinedGlobalError as e:
341         # TODO: is the order Ok? Should globals come before builtins?
342         if key == 'len':
343             item = len
344         elif key in __builtins__:
345             item = __builtins__[key]
346         elif key in func.pyFunc().__globals__:
347             item = func.pyFunc().__globals__[key]
348         else:
349             raise e
350     wrapped = self._wrapPython(key, item)
351
352     # _wrapPython will create an entry for functions _only_
353     if not isinstance(wrapped, FunctionRef):
354         self.namestore[key] = wrapped
355
356     # instrinsic check: we need a new instance for every call!
357     # TODO: this may be required in _getFunction?
358     if inspect.isclass(wrapped) and issubclass(wrapped,
359         ↪ intrinsics.Intrinsic):
360         return wrapped()
361     else:
362         return wrapped
363
364 def newGlobal(self, func, name):
365     """"MUST ensure that loadGlobal() fails before calling this
366     ↪ function!"""
367     wrapped = GlobalVariable(name, None)
368     self.namestore[name] = wrapped
369     return wrapped
370
371 def functionCall(self, funcref, args, kwargs):
372     func = funcref.function
373     if isinstance(func, tp.Foreign):
374         # no need to analyze it
375         return
```

Appendix C. STELLA Source Code

```
375     if kwargs is None:
376         kwargs = {}
377     if not func.analyzed or func.result.type is tp.NoType:
378         self.todoAdd(funcref, args, kwargs)
379
380 def todoAdd(self, func, args, kwargs):
381     # If the function was already in the list to be analyzed, remove
382     ↪ it
383     # so that it is only present once at the end
384     # TODO look at args and kwargs
385     self._todo = list(filter(lambda t: t[0] != func, self._todo))
386     self._todo.append((func, args, kwargs))
387
388 def todoLastFunc(self, func):
389     if len(self._todo) > 0:
390         (f, _, _) = self._todo[-1]
391         if f == func:
392             return True
393         return False
394
395 def todoNext(self):
396     n = self._todo[0]
397     self._todo = self._todo[1:]
398     self.log.log(utils.VERBOSE, "current TODO function %s", n)
399     return n
400
401 def todoCount(self):
402     return len(self._todo)
403
404 def todoList(self):
405     return self._todo
406
407 def translate(self):
408     self.llvm = ll.Module('__stella__'+str(self.__class__.__i),
409     ↪ context=ll.context.Context())
410     self.__class__.__i += 1
411     for _, impl in self.namestore.all(Function):
412         impl.translate(self)
413
414 def destruct(self):
415     """Clean up this objects so that gc will succeed.
416     Function has a weakref back to Module, but something in Function
```

Appendix C. STELLA Source Code

```
416         confuses the gc algorithm and Module will never be collected while
417         fully intact.
418         """
419         logging.debug("destruct() of " + repr(self))
420
421         # destruct() can be called more than once
422         if hasattr(self, 'entry'):
423             del self.entry
424         if hasattr(self, 'entry_args'):
425             del self.entry_args
426
427         msg = []
428         while True:
429             try:
430                 d = self._cleanup.pop()
431                 msg.append(str(d))
432                 d()
433             except IndexError:
434                 break
435         if len(msg) > 0:
436             logging.debug("Called destructors: " + ", ".join(msg))
437
438     def addDestruct(self, d):
439         self._cleanup.append(d)
440
441     def __del__(self):
442         logging.debug("DEL " + repr(self))
443         self.destruct()
444
445     def getLlvmIR(self):
446         if self.llvm:
447             return str(self.llvm)
448         else:
449             return "<no code yet>"
450
451     def __str__(self):
452         return '__module__' + str(id(self))
453
454
455 class Function(Scope):
456     """
457     Represents the code of the function. Has to be unique for each source
458     instance.
```

Appendix C. STELLA Source Code

```
459     """
460     def __init__(self, type_, module):
461         # TODO: pass the module as the parent for scope
462         super().__init__(None)
463         if not isinstance(type_, tp.FunctionType):
464             type_ = tp.FunctionType.get(type_)
465         self.type_ = type_
466         self.name = type_.fq
467         self.result = Register(self, '__return__')
468
469         self.args = []
470
471         # Use a weak reference here because module has a reference to the
472         # function -- the cycle would prevent gc
473         self.module = weakref.proxy(module)
474
475         self.analyzed = False
476         self.log = logging.getLogger(str(self))
477
478     def pyFunc(self):
479         return self.type_.pyFunc()
480
481     def __str__(self):
482         return self.name
483
484     def __repr__(self):
485         return "{}:{}".format(super().__repr__()[:-1], self)
486
487     def nameAndType(self):
488         return self.name + "(" + str(self.args) + ")"
489
490     def getReturnType(self, args, kw_args):
491         return self.result.type
492
493     def analyzeAgain(self):
494         """Pushes the current function on the module's analysis todo
495         ↪ list"""
496         if not self.module.todoLastFunc(self):
497             self.module.todoAdd(self, None, None)
498
499     def loadGlobal(self, key):
500         return self.module.loadGlobal(self, key)
```

Appendix C. STELLA Source Code

```
501     def newGlobal(self, key):
502         return self.module.newGlobal(self, key)
503
504     def setupArgs(self, args, kwargs):
505         tp_args = [arg.type for arg in args]
506         tp_kwargs = {k: v.type for k, v in kwargs.items()}
507
508         combined = self.type_.typeArgs(tp_args, tp_kwargs)
509
510         self.arg_transfer = []
511
512         for i in range(len(combined)):
513             type_ = combined[i]
514
515             if type_.on_heap:
516                 # TODO: create superclass for complex types
517                 arg = self.getOrCreateRegister(self.type_.arg_names[i])
518                 arg.type = type_
519             else:
520                 name = self.type_.arg_names[i]
521                 arg = self.getOrCreateRegister('__param_'+name)
522                 arg.type = type_
523                 self.arg_transfer.append(name)
524
525             self.args.append(arg)
526
527         self.analyzed = True
528
529     def translate(self, module):
530         self.arg_types = [arg.llvmType(module) for arg in self.args]
531
532         func_tp = ll.FunctionType(self.result.type.llvmType(module),
533             ↪ self.arg_types)
534         self.llvm = ll.Function(module.llvm, func_tp, name=self.name)
535
536         for i in range(len(self.args)):
537             self.llvm.args[i].name = self.args[i].name
538             self.args[i].llvm = self.llvm.args[i]
539
540     def remove(self, bc):
541         # TODO: should any of these .next become .linearNext()?
542         if bc == self.bytecodes:
```

Appendix C. STELLA Source Code

```
543         self.bytecodes = bc.next
544
545     if bc in self.incoming_jumps:
546         bc_next = bc.next
547         if not bc_next and bc._block_parent:
548             bc_next = bc._block_parent.next
549             # _block_parent will be move with bc.remove() below
550         assert bc_next
551         self.incoming_jumps[bc_next] = self.incoming_jumps[bc]
552         for bc_ in self.incoming_jumps[bc_next]:
553             bc_.updateTargetBytecode(bc, bc_next)
554         del self.incoming_jumps[bc]
555     bc.remove()
556
557
558 class FunctionRef(tp.Callable):
559     def __init__(self, function):
560         self.function = function
561
562     def __str__(self):
563         return "<*function {}>".format(self.function)
564
565     def __repr__(self):
566         return "{}:{}".format(super().__repr__()[:-1],
567                                ↪ self.function.type_.fq)
568
569     @property
570     def type_(self):
571         """Convenience function to return the referenced function's
572         ↪ type."""
573         # TODO: this may be confusing if function references become first
574         ↪ class
575         # citizens!
576         return self.function.type_
577
578     def makeEntry(self, args, kwargs):
579         # TODO Verify that type checking occurred at this point
580         self.function.module.makeEntry(self, self.combineArgs(args,
581                                ↪ kwargs))
582
583     def getResult(self, func):
584         return Register(func)
585
586
```


Appendix C. STELLA Source Code

```
582     # TODO: Maybe the caller of the following functions should resolve to
583     # self.function instead of making a proxy call here.
584
585     def getReturnType(self, args, kw_args):
586         return self.function.getReturnType(args, kw_args)
587
588     @property
589     def result(self):
590         return self.function.result
591
592     @property
593     def llvm(self):
594         return self.function.llvm
595
596
597 class BoundFunctionRef(FunctionRef):
598     def __init__(self, function):
599         super().__init__(function)
600
601     def __str__(self):
602         return "<*bound method {} of {}>".format(self.function,
603             ↪ self.type_.bound)
604
605     @property
606     def self_type(self):
607         return self.type_.bound
608
609     @property
610     def f_self(self):
611         return self._f_self
612
613     @f_self.setter
614     def f_self(self, value):
615         # TODO: These should be throw-away objects. I want to know if they
616         ↪ live
617         # longer than expected.
618         assert not hasattr(self, '_f_self')
619         self._f_self = value
620
621     def combineArgs(self, args, kwargs):
622         full_args = [self.f_self] + args
623         return super().combineArgs(full_args, kwargs)
```

Appendix C. STELLA Source Code

```
623
624 class ExtModule(object):
625     python = None
626     signatures = {}
627     funcs = dict()
628
629     def __init__(self, python):
630         assert type(python) == type(sys)
631
632         self.python = python
633         self.signatures = python.getCSignatures()
634
635         for name, sig in self.signatures.items():
636             type_ = tp.ExtFunctionType(python, sig)
637             self.funcs[name] = ExtFunction(name, type_)
638         self.translated = False
639
640     def getFile(self):
641         return self.python.__file__
642
643     def getSymbols(self):
644         return self.signatures.keys()
645
646     def getSignatures(self):
647         return self.signatures.items()
648
649     def getFunction(self, f):
650         return self.funcs[f.__name__]
651
652     def getFunctionRef(self, f):
653         return ExtFunctionRef(self.funcs[f.__name__])
654
655     def __str__(self):
656         return str(self.python)
657
658     def translate(self, module):
659         if not self.translated:
660             self.translated = True
661             logging.debug("Adding external module
662                 ↪ {0}".format(self.python))
662             clib = ctypes.cdll.LoadLibrary(self.python.__file__)
663             for func in self.funcs.values():
664                 func.translate(clib, module)
```

Appendix C. STELLA Source Code

```
665
666
667 class ExtFunction(tp.Foreign):
668     llvm = None
669     analyzed = True
670     name = '?()'
671
672     def __init__(self, name, type_):
673         self.name = name
674         self.type_ = type_
675         self.result = Register(self, '__return__')
676
677     def __str__(self):
678         return self.name
679
680     def getReturnType(self, args, kw_args):
681         # TODO: Do we need to type self.result?
682         return self.type_.getReturnType(args, kw_args)
683
684     def translate(self, clib, module):
685         logging.debug("Adding external function {0}".format(self.name))
686         f = getattr(clib, self.name)
687         llvm.add_symbol(self.name, ctypes.cast(f, ctypes.c_void_p).value)
688
689         llvm_arg_types = [arg.llvmType(module) for arg in
690             ↪ self.type_.arg_types]
691
692         func_tp = ll.FunctionType(self.type_.return_type.llvmType(module),
693             ↪ llvm_arg_types)
694         self.llvm = ll.Function(module, func_tp, self.name)
695
696 class ExtFunctionRef(FunctionRef):
697     def makeEntry(self, args, kwargs):
698         msg = "External function {} cannot be the entry method for
699             ↪ Stella".format(self.function)
700         raise exc.TypeError(msg)
701
702     def call(self, cge, args, kw_args):
703         args = self.combineArgs(args, kw_args)
704
705         args_llvm = []
706         for arg, arg_type in zip(args, self.type_.arg_types):
```

Appendix C. STELLA Source Code

```
705         if arg.type != arg_type:
706             # TODO: trunc is not valid for all type combinations.
707             # Needs to be generalized.
708             llvm = cge.builder.trunc(arg.translate(cge),
709                                     arg_type.llvmType(cge.module),
710                                     '{0}{1}'.format(arg_type,
711                                                     ↪ arg.name))
712         else:
713             llvm = arg.llvm
714             args_llvm.append(llvm)
715     return cge.builder.call(self.function.llvm, args_llvm)
```

C.8 stella/bytecode.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import dis
15 import sys
16 import types
17 from abc import abstractproperty
18
19 from . import tp
20 from . import exc
21 from . import utils
22 from . import ir
23 from .storage import Register, StackLoc, GlobalVariable
24 from .tp import Cast, Const
```

Appendix C. STELLA Source Code

```
25 from .intrinsic import Intrinsic
26 from copy import copy
27
28
29 def pop_stack(n):
30     """
31     Decorator, it takes n items off the stack
32     and adds them as bytecode arguments.
33     """
34     def extract_n(f):
35         def extract_from_stack(self, func, stack):
36             args = []
37             for i in range(n):
38                 args.append(stack.pop())
39             args.reverse()
40
41             self.stack_bc = args
42             return f(self, func, stack)
43         return extract_from_stack
44     return extract_n
45
46
47 class Poison(object):
48     """
49     Require that this bytecode is rewritten by bailing out
50     if it is ever evaluated.
51
52     Note that if the child overrides all methods, this mixin will be
53     ↪ useless
54     and should be removed from the child.
55     """
56
57     def stack_eval(self, func, stack):
58         raise exc.UnimplementedError(
59             "{0} must be rewritten".format(
60                 self.__class__.__name__))
61
62     def translate(self, cge):
63         raise exc.UnimplementedError(
64             "{0} must be rewritten".format(
65                 self.__class__.__name__))
66
```

Appendix C. STELLA Source Code

```
67     def type_eval(self, func):
68         raise exc.UnimplementedError(
69             "{0} must be rewritten".format(
70                 self.__class__.__name__))
71
72
73 class Bytecode(ir.IR):
74     """
75     Parent class for all Python bytecodes
76     """
77     pass
78
79
80 class ResultOnlyBytecode(Poison, ir.IR):
81     """Only use this to inject values on the stack which did not originate
82     ↪ from
83     any real bytecode. This will only work at the beginning of a program
84     because otherwise the bytecode may be used as the origin of a branch.
85     """
86     def __init__(self, func, debuginfo):
87         super().__init__(func, debuginfo)
88
89 class LOAD_FAST(Bytecode):
90     def __init__(self, func, debuginfo):
91         super().__init__(func, debuginfo)
92         self.source = None
93
94     def addLocalName(self, func, name):
95         # TODO: crude?
96         try:
97             self.source = func.getRegister(name)
98         except exc.UndefinedError:
99             self.source = func.getStackLoc(name)
100
101     def addArg(self, arg):
102         assert self.source is None
103         self.source = arg
104
105     @property
106     def _str_args(self):
107         return str(self.source)
108
```

Appendix C. STELLA Source Code

```
109     def stack_eval(self, func, stack):
110         stack.push(self)
111
112     def type_eval(self, func):
113         self.grab_stack()
114         arg_type = self.source.type
115         if self.result is None:
116             type_ = type(self.source)
117             if type_ == StackLoc:
118                 self.result = Register(func.impl)
119             elif type_ == Register:
120                 self.result = self.source
121             else:
122                 raise exc.StellaException(
123                     "Invalid LOAD_FAST argument type '{0}'".format(type_))
124         if type(self.source) == StackLoc:
125             if arg_type.isReference():
126                 arg_type = arg_type.dereference()
127             self.result.unify_type(arg_type, self.debuginfo)
128
129     def translate(self, cge):
130         type_ = type(self.source)
131         if type_ == StackLoc:
132             self.result.llvm =
133                 ↪ cge.builder.load(self.source.translate(cge))
134         elif type_ == Register:
135             # nothing to load, it's a pseudo instruction in this case
136             pass
137
138     class STORE_FAST(Bytecode):
139         def __init__(self, func, debuginfo):
140             super().__init__(func, debuginfo)
141             self.new_allocate = False
142             self.needs_cast = False
143
144         def addLocalName(self, func, name):
145             # Python does not allocate new names, it just refers to them
146             (self.result, self.new_allocate) = func.getOrNewStackLoc(name)
147
148         def addArg(self, arg):
149             assert self.result is None
150             self.result = arg
```

Appendix C. STELLA Source Code

```
151
152 @pop_stack(1)
153 def stack_eval(self, func, stack):
154     pass
155
156 def type_eval(self, func):
157     self.grab_stack()
158     # func.retype(self.result.unify_type(self.args[1].type,
159     ↪ self.debuginfo))
160
161 arg = self.args[0]
162 if arg.type.complex_on_stack or arg.type.on_heap:
163     type_ = tp.Reference(arg.type)
164 else:
165     type_ = arg.type
166 widened, needs_cast = self.result.unify_type(type_,
167     ↪ self.debuginfo)
168 if widened:
169     # TODO: can I avoid a retype in some cases?
170     func.retype()
171 if needs_cast or self.needs_cast:
172     self.needs_cast = True
173     self.args[0] = Cast(arg, self.result.type)
174
175 def translate(self, cge):
176     self.cast(cge)
177     arg = self.args[0]
178     if self.new_allocate:
179         type_ = self.result.type
180         if type_.on_heap:
181             type_ = type_.dereference()
182             llvm_type = type_.llvmType(cge.module)
183             self.result.llvm = cge.builder.alloca(llvm_type,
184             ↪ name=self.result.name)
185     cge.builder.store(arg.translate(cge), self.result.translate(cge))
186
187 class STORE_GLOBAL(Bytecode):
188     def __init__(self, func, debuginfo):
189         super().__init__(func, debuginfo)
190
191     def addName(self, func, name):
192         # Python does not allocate new names, it just refers to them
```


Appendix C. STELLA Source Code

```
191     try:
192         self.result = func.loadGlobal(name)
193     except exc.UndefinedError:
194         self.result = func.newGlobal(name)
195
196     @pop_stack(1)
197     def stack_eval(self, func, stack):
198         pass
199
200     def type_eval(self, func):
201         self.grab_stack()
202         # func.retype(self.result.unify_type(self.args[1].type,
203         ↪ self.debuginfo))
204         arg = self.args[0]
205
206         if self.result.initial_value is None:
207             # This means we're defining a new variable
208             self.result.setInitialValue(arg)
209
210         widened, needs_cast = self.result.unify_type(arg.type,
211         ↪ self.debuginfo)
212         if widened:
213             # TODO: can I avoid a retype in some cases?
214             func.retype()
215         if needs_cast:
216             # TODO: side effect! Maybe that's for the best.
217             self.args[0] = Cast(arg, self.result.type)
218
219     def translate(self, cge):
220         # Assume that the global has been allocated already.
221         self.cast(cge)
222         cge.builder.store(self.args[0].translate(cge),
223         ↪ self.result.translate(cge))
224
225     class LOAD_CONST(Bytecode):
226         discard = True
227
228     def __init__(self, func, debuginfo):
229         super().__init__(func, debuginfo)
230
231     def addArg(self, arg):
232         assert self.const_arg is None
```

Appendix C. STELLA Source Code

```
231         self.const_arg = arg
232
233     def stack_eval(self, func, stack):
234         self.result = self.const_arg
235         stack.push(self)
236
237     def type_eval(self, func):
238         pass
239
240     def translate(self, cge):
241         pass
242
243
244 class BinaryOp(Bytecode):
245     def __init__(self, func, debuginfo):
246         super().__init__(func, debuginfo)
247         self.result = Register(func)
248         self.needs_cast = [False, False]
249
250     @pop_stack(2)
251     def stack_eval(self, func, stack):
252         stack.push(self)
253
254     def type_eval(self, func):
255         self.grab_stack()
256         for i in range(len(self.args)):
257             arg = self.args[i]
258             if arg.type == self.result.type:
259                 # a cast may have been necessary in the previous
260                 # iteration,
261                 # but now the argument may have changed type, so check
262                 # before
263                 # continuing
264                 self.needs_cast[i] = False
265             if self.needs_cast[i]:
266                 # install the cast before unify_type() because otherwise
267                 # we're
268                 # in an infinite loop retyping the function
269                 self.args[i] = Cast(arg, self.result.type)
270             widened, needs_cast = self.result.unify_type(arg.type,
271                 # self.debuginfo)
272             if widened:
273                 # TODO: can I avoid a retype in some cases?
```

Appendix C. STELLA Source Code

```
270         # It could definitely be smarter and retype the other
271         → parameter
272         # directly if need be.
273         func.retype()
274         if needs_cast:
275             self.needs_cast[i] = True
276             # install the cast here because we may not get re-typed
277             self.args[i] = Cast(arg, self.result.type)
278
279     def builderFuncName(self):
280         try:
281             return self.b_func[self.result.type]
282         except KeyError:
283             raise exc.TypeError(
284                 "{0} does not yet implement type {1}".format(
285                     self.__class__.__name__,
286                     self.result.type))
287
288     def translate(self, cge):
289         self.cast(cge)
290         f = getattr(cge.builder, self.builderFuncName())
291         self.result.llvm = f(
292             self.args[0].translate(cge),
293             self.args[1].translate(cge))
294
295     @abstractmethod
296     def b_func(self):
297         return {}
298
299     class BINARY_ADD(BinaryOp):
300         b_func = {tp.Float: 'fadd', tp.Int: 'add'}
301
302
303     class BINARY_SUBTRACT(BinaryOp):
304         b_func = {tp.Float: 'fsub', tp.Int: 'sub'}
305
306
307     class BINARY_MULTIPLY(BinaryOp):
308         b_func = {tp.Float: 'fmul', tp.Int: 'mul'}
309
310
311     class BINARY_MODULO(BinaryOp):
```

Appendix C. STELLA Source Code

```
312     b_func = {tp.Float: 'frem', tp.Int: 'srem'}
313
314
315 class BINARY_POWER(BinaryOp):
316     b_func = {tp.Float: 'llvm.pow', tp.Int: 'llvm.powi'}
317
318     def __init__(self, func, debuginfo):
319         super().__init__(func, debuginfo)
320         self.result = Register(func)
321
322     @pop_stack(2)
323     def stack_eval(self, func, stack):
324         stack.push(self)
325
326     def type_eval(self, func):
327         self.grab_stack()
328         # TODO if args[1] is int but negative, then the result will be
329         ↪ float, too!
330         super().type_eval(func)
331
332     def translate(self, cge):
333         # llvm.pow[i]'s first argument always has to be float
334         arg = self.args[0]
335         if arg.type == tp.Int:
336             self.args[0] = Cast(arg, tp.Float)
337
338         self.cast(cge)
339
340         if self.args[1].type == tp.Int:
341             # powi takes a i32 argument
342             power = cge.builder.trunc(
343                 self.args[1].translate(cge),
344                 tp.tp_int32,
345                 '(i32)' +
346                 self.args[1].name)
347         else:
348             power = self.args[1].translate(cge)
349
350         llvm_pow =
351             ↪ cge.module.llvm.declare_intrinsic(self.b_func[self.args[1].type],
352             ↪ [self.args[0].llvmType(cge.mod
```

Appendix C. STELLA Source Code

```
351     pow_result = cge.builder.call(llvm_pow,
352     ↪ [self.args[0].translate(cge), power])
353
354     if isinstance(self.args[0], Cast) and \
355         self.args[0].obj.type == tp.Int and self.args[1].type ==
356         ↪ tp.Int:
357         # cast back to an integer
358         self.result.llvm = cge.builder.fptosi(pow_result,
359         ↪ tp.Int.llvmType(cge.module))
360     else:
361         self.result.llvm = pow_result
362
363 class BINARY_FLOOR_DIVIDE(BinaryOp):
364     """Python compliant '//' operator.
365
366     Slow since it has to perform type conversions and floating point
367     ↪ division for integers"""
368     b_func = {
369         tp.Float: 'fdiv',
370         tp.Int: 'fdiv'} # NOT USED, but required to make it a concrete
371     ↪ class
372
373     def __init__(self, func, debuginfo):
374         super().__init__(func, debuginfo)
375
376     def type_eval(self, func):
377         self.grab_stack()
378         for arg in self.args:
379             widened, _ = self.result.unify_type(arg.type, self.debuginfo)
380             func.retype(widened)
381
382     def translate(self, cge):
383         is_int = all([arg.type == tp.Int for arg in self.args])
384         for i in range(len(self.args)):
385             if self.args[i].type != tp.Float:
386                 self.args[i] = Cast(self.args[i], tp.Float)
387         self.cast(cge)
388
389         tmp = cge.builder.fdiv(
390             self.args[0].translate(cge),
391             self.args[1].translate(cge))
392         llvm_floor = cge.module.llvm.declare_intrinsic('llvm.floor',
```

Appendix C. STELLA Source Code

```
389
390                                     ↪ [tp.Float.llvmType(cge.module)
self.result.llvm = cge.builder.call(llvm_floor, [tmp])
391
392 if is_int:
393     # TODO this may be superflous if both args got converted to
    ↪ float
394     # in the translation stage -> move toFloat partially to the
    # analysis stage.
395     self.result.llvm = cge.builder.fptosi(
396         self.result.translate(cge),
397         tp.Int.llvmType(cge.module),
398         "(int)" +
399         self.result.name)
400
401
402
403 class BINARY_TRUE_DIVIDE(BinaryOp):
404     b_func = {tp.Float: 'fdiv'}
405
406     def __init__(self, func, debuginfo):
407         super().__init__(func, debuginfo)
408         self.result = Register(func)
409
410     @pop_stack(2)
411     def stack_eval(self, func, stack):
412         stack.push(self)
413
414     def type_eval(self, func):
415         self.grab_stack()
416         # The result of '/', true division, is always a float
417         self.result.type = tp.Float
418         super().type_eval(func)
419
420
421 class INPLACE_ADD(BINARY_ADD):
422     pass
423
424
425 class INPLACE_SUBTRACT(BINARY_SUBTRACT):
426     pass
427
428
429 class INPLACE_MULTIPLY(BINARY_MULTIPLY):
```

Appendix C. STELLA Source Code

```
430     pass
431
432
433 class INPLACE_TRUE_DIVIDE(BINARY_TRUE_DIVIDE):
434     pass
435
436
437 class INPLACE_FLOOR_DIVIDE(BINARY_FLOOR_DIVIDE):
438     pass
439
440
441 class INPLACE_MODULO(BINARY_MODULO):
442     pass
443
444
445 class COMPARE_OP(Bytecode):
446     b_func = {tp.Float: 'fcmp_ordered', tp.Int: 'icmp_signed', tp.Bool:
447         ↪ 'icmp_signed'}
448     op = None
449
450     def __init__(self, func, debuginfo):
451         super().__init__(func, debuginfo)
452         self.result = Register(func)
453
454     def addCmp(self, op):
455         self.op = op
456
457     @pop_stack(2)
458     def stack_eval(self, func, stack):
459         stack.push(self)
460
461     def type_eval(self, func):
462         self.grab_stack()
463         self.result.type = tp.Bool
464
465         # upcast integers to float if required
466         if (self.args[0].type == tp.Int and self.args[1].type ==
467             ↪ tp.Float):
468             self.args[0] = Cast(self.args[0], tp.Float)
469         if (self.args[0].type == tp.Float and self.args[1].type ==
470             ↪ tp.Int):
471             self.args[1] = Cast(self.args[1], tp.Float)
```

Appendix C. STELLA Source Code

```
470     if (self.args[0].type != self.args[1].type and
471         self.args[0].type != tp.NoType and self.args[1].type !=
472         ↪ tp.NoType):
473         raise exc.TypeError(
474             "Comparing different types ({0} with {1})".format(
475                 self.args[0].type,
476                 self.args[1].type))
477
478 def translate(self, cge):
479     # assume both types are the same, see @stack_eval
480     type_ = self.args[0].type
481     if not self.args[0].type in self.b_func:
482         raise exc.UnimplementedError(type_)
483
484     f = getattr(cge.builder, self.b_func[type_])
485
486     llvm = f(self.op,
487             self.args[0].translate(cge),
488             self.args[1].translate(cge))
489     # the comparison returns i1 but we need to return an i8
490     self.result.llvm = cge.builder.zext(llvm, tp.Bool.llvmType(cge))
491
492 class RETURN_VALUE(utils.BlockTerminal, Bytecode):
493
494     def __init__(self, func, debuginfo):
495         super().__init__(func, debuginfo)
496         self.func = func
497
498     @pop_stack(1)
499     def stack_eval(self, func, stack):
500         pass
501
502     def type_eval(self, func):
503         self.grab_stack()
504         self.result = self.args[0]
505         for arg in self.args:
506             func.retype(self.result.unify_type(arg.type, self.debuginfo))
507
508     def translate(self, cge):
509         if self.result.type is tp.Void:
510             if self.func.result.type is tp.Void:
511                 cge.builder.ret_void()
```


Appendix C. STELLA Source Code

```
512         else:
513             cge.builder.ret(self.func.result.type.null(cge.module))
514     else:
515         cge.builder.ret(self.result.translate(cge))
516
517
518 class HasTarget(object):
519     target_label = None
520     target_bc = None
521
522     def setTargetBytecode(self, bc):
523         self.target_bc = bc
524
525     def updateTargetBytecode(self, old_bc, new_bc):
526         self.setTargetBytecode(new_bc)
527
528     def setTarget(self, label):
529         self.target_label = label
530
531     def __str__(self):
532         return "{0} {1} {2}".format(
533             self.__class__.__name__,
534             self.target_label,
535             ", ".join([str(v) for v in self.args]))
536
537
538 class Jump(utils.BlockTerminal, HasTarget, ir.IR):
539
540     def __init__(self, func, debuginfo):
541         super().__init__(func, debuginfo)
542
543     def processFallThrough(self):
544         return False
545
546     def stack_eval(self, func, stack):
547         return [(self.target_bc, stack)]
548
549     def type_eval(self, func):
550         self.grab_stack()
551
552     def translate(self, cge):
553         cge.builder.branch(self.target_bc.block)
554
```

Appendix C. STELLA Source Code

```
555
556 class Jump_if_X_or_pop(Jump):
557
558     def __init__(self, func, debuginfo):
559         super().__init__(func, debuginfo)
560
561     def processFallThrough(self):
562         self.fallthrough = self.next
563         return True
564
565     def updateTargetBytecode(self, old_bc, new_bc):
566         if old_bc == self.target_bc:
567             self.setTargetBytecode(new_bc)
568         else:
569             assert self.fallthrough == old_bc
570             self.fallthrough = new_bc
571
572     @pop_stack(1)
573     def stack_eval(self, func, stack):
574         stack2 = stack.clone()
575         r = []
576         # if X, push back onto stack and jump:
577         stack.push(self.stack_bc[0])
578         r.append((self.target_bc, stack))
579         # else continue with the next instruction (and keep the popped
580         ↪ value)
581         r.append((self.next, stack2))
582
583         return r
584
585 class JUMP_IF_FALSE_OR_POP(Jump_if_X_or_pop, Bytecode):
586
587     def __init__(self, func, debuginfo):
588         super().__init__(func, debuginfo)
589
590     def translate(self, cge):
591         cond = tp.Cast.translate_i1(self.args[0], cge)
592         cge.builder.cbranch(cond,
593                             self.next.block,
594                             self.target_bc.block)
595
596
```

Appendix C. STELLA Source Code

```
597 class JUMP_IF_TRUE_OR_POP(Jump_if_X_or_pop, Bytecode):
598
599     def __init__(self, func, debuginfo):
600         super().__init__(func, debuginfo)
601
602     def translate(self, cge):
603         cond = tp.Cast.translate_i1(self.args[0], cge)
604         cge.builder.cbranch(cond,
605                             self.target_bc.block,
606                             self.next.block)
607
608
609 class Pop_jump_if_X(Jump):
610
611     def __init__(self, func, debuginfo):
612         super().__init__(func, debuginfo)
613         self.additional_pops = 0
614
615     def processFallThrough(self):
616         self.fallthrough = self.next
617         return True
618
619     def updateTargetBytecode(self, old_bc, new_bc):
620         if old_bc == self.target_bc:
621             self.setTargetBytecode(new_bc)
622         else:
623             assert self.fallthrough == old_bc
624             self.fallthrough = new_bc
625
626     def additionalPop(self, i):
627         """Deviate from Python semantics: pop i more items off the stack
628         ↪ WHEN jumping.
629
630         Instead of the Python semantics to pop one value of the stack, pop
631 ↪ i more when jumping.
632         """
633         self.additional_pops = i
634
635     @pop_stack(1)
636     def stack_eval(self, func, stack):
637         r = []
638         # if X, jump
639         jump_stack = stack.clone()
```

Appendix C. STELLA Source Code

```
638     for i in range(self.additional_pops):
639         jump_stack.pop()
640     r.append((self.target_bc, jump_stack))
641     # else continue to the next instruction
642     r.append((self.next, stack))
643     # (pop happens in any case)
644
645     return r
646
647
648 class POP_JUMP_IF_FALSE(Pop_jump_if_X, Bytecode):
649
650     def __init__(self, func, debuginfo):
651         super().__init__(func, debuginfo)
652
653     def translate(self, cge):
654         cond = tp.Cast.translate_i1(self.args[0], cge)
655         cge.builder.cbranch(cond,
656                             self.next.block,
657                             self.target_bc.block)
658
659
660 class POP_JUMP_IF_TRUE(Pop_jump_if_X, Bytecode):
661
662     def __init__(self, func, debuginfo):
663         super().__init__(func, debuginfo)
664
665     def translate(self, cge):
666         cond = tp.Cast.translate_i1(self.args[0], cge)
667         cge.builder.cbranch(cond,
668                             self.target_bc.block,
669                             self.next.block)
670
671
672 class SETUP_LOOP(utils.BlockStart, HasTarget, Bytecode):
673     """
674     Will either be rewritten (for loop) or has no effect other than mark
675     ↪ the
676     start of a block (while loop).
677     """
678     discard = True
679
680     def __init__(self, func, debuginfo):
```

Appendix C. STELLA Source Code

```
680         super().__init__(func, debuginfo)
681
682     def stack_eval(self, func, stack):
683         pass
684
685     def translate(self, cge):
686         pass
687
688     def type_eval(self, func):
689         pass
690
691
692 class POP_BLOCK(utils.BlockEnd, Bytecode):
693     discard = True
694
695     def __init__(self, func, debuginfo):
696         super().__init__(func, debuginfo)
697
698     def stack_eval(self, func, stack):
699         pass
700
701     def translate(self, cge):
702         pass
703
704     def type_eval(self, func):
705         pass
706
707
708 class LOAD_GLOBAL(Bytecode):
709     var = None
710
711     def __init__(self, func, debuginfo):
712         super().__init__(func, debuginfo)
713
714     def addName(self, func, name):
715         self.args.append(name)
716
717     def stack_eval(self, func, stack):
718         stack.push(self)
719
720     def translate(self, cge):
721         if isinstance(self.var, ir.FunctionRef):
722             pass
```

Appendix C. STELLA Source Code

```
723     elif isinstance(self.var, GlobalVariable):
724         self.result.llvm = cge.builder.load(self.var.translate(cge))
725
726 def type_eval(self, func):
727     self.grab_stack()
728     if self.result is None:
729         self.var = func.impl.loadGlobal(self.args[0])
730         # TODO: remove these isinstance checks and just check for
731         # GlobalVariable else return directly?
732         if isinstance(self.var, ir.FunctionRef):
733             self.result = self.var
734         elif isinstance(self.var, types.ModuleType):
735             self.result = self.var
736         elif isinstance(self.var, type):
737             self.result = tp.PyWrapper(self.var)
738         elif isinstance(self.var, Intrinsic):
739             self.result = self.var
740         elif isinstance(self.var, GlobalVariable):
741             self.result = Register(func.impl)
742         else:
743             raise exc.UnimplementedError(
744                 "Unknown global type {}".format(
745                     type(self.var)))
746
747     if isinstance(self.var, GlobalVariable):
748         self.result.unify_type(self.var.type, self.debuginfo)
749
750
751 class LOAD_ATTR(Bytecode):
752     def __init__(self, func, debuginfo):
753         super().__init__(func, debuginfo)
754
755     def addName(self, func, name):
756         self.name = name
757
758     @property
759     def _str_args(self):
760         if len(self.args) > 0:
761             obj = self.args[0]
762         else:
763             obj = '?'
764         return "{}.{}".format(obj, self.name)
765
```

Appendix C. STELLA Source Code

```
766     @pop_stack(1)
767     def stack_eval(self, func, stack):
768         stack.push(self)
769
770     def type_eval(self, func):
771         self.grab_stack()
772
773         arg = self.args[0]
774         # TODO would it be better to move some of this into arg.type?
775
776         if isinstance(arg, types.ModuleType):
777             self.result = func.module.loadExt(arg, self.name)
778             self.discard = True
779             return
780
781         type_ = arg.type.dereference()
782         if isinstance(type_, tp.StructType):
783             try:
784                 attr_type =
785                     → arg.type.dereference().getMemberType(self.name)
786             except KeyError:
787                 raise exc AttributeError("Unknown field {} of type
788                     → {}".format(self.name,
789                                     self.debuginfo)
790                                     → arg.t)
791             if isinstance(attr_type, tp.FunctionType):
792                 self.result = func.module.getFunctionRef(attr_type)
793             else:
794                 if self.result is None:
795                     self.result = Register(func.impl)
796                     self.result.unify_type(attr_type, self.debuginfo)
797                 elif isinstance(type_, tp.ArrayType):
798                     if self.result is None:
799                         self.result = Register(func.impl)
800                         self.result.unify_type(tp.get(type_.shape), self.debuginfo)
801                     else:
802                         raise exc TypeError("Cannot load attribute {} from type
803                             → {}".format(self.name,
804                                             self.debuginfo)
805                                             → arg.t)
806
807                 self.debuginfo)
```

Appendix C. STELLA Source Code

```
804     def translate(self, cge):
805         arg = self.args[0]
806         if isinstance(arg, types.ModuleType):
807             return
808
809         type_ = arg.type.dereference()
810         if isinstance(type_, tp.StructType):
811             tp_attr = type_.getMemberType(self.name)
812             if isinstance(tp_attr, tp.FunctionType):
813                 self.result.f_self = arg
814                 return
815             idx = type_.getMemberIdx(self.name)
816             idx_llvm = tp.getIndex(idx)
817             struct_llvm = arg.translate(cge)
818             p = cge.builder.gep(struct_llvm, [tp.Int.constant(0),
819                                     ↪ idx_llvm], inbounds=True)
819             self.result.llvm = cge.builder.load(p)
820         elif isinstance(type_, tp.ArrayType):
821             val = tp.wrapValue(type_.shape)
822             self.result.llvm = val.translate(cge)
823         else:
824             raise exc.UnimplementedError(type(arg))
825
826     class STORE_ATTR(Bytecode):
827         def __init__(self, func, debuginfo):
828             super().__init__(func, debuginfo)
829             # TODO: Does the result have to be a register? Don't I only need
830             ↪ it for
831             # the llvm propagation?
832             self.result = Register(func)
833
834         def addName(self, func, name):
835             self.name = name
836
837         @pop_stack(2)
838         def stack_eval(self, func, stack):
839             pass
840
841         def type_eval(self, func):
842             self.grab_stack()
843             type_ = self.args[1].type.dereference()
844             if isinstance(type_, tp.StructType):
```


Appendix C. STELLA Source Code

```
845         member_type = type_.getMemberType(self.name)
846         arg_type = self.args[0].type
847         if member_type != arg_type:
848             if member_type == tp.Float and arg_type == tp.Int:
849                 self.args[0] = tp.Cast(self.args[0], tp.Float)
850                 return
851             # TODO would it speed up the algorithm if arg_type is set
852             → to be
853             # member_type here?
854             if arg_type == tp.NoType:
855                 # TODO dead code, not necessary anymore since function
856                 → calls interrupt typing
857                 return
858             raise exc.TypeError("Argument type {} incompatible with
859             → member type {}".format(
860                 arg_type, member_type))
861         else:
862             raise exc.UnimplementedError(
863                 "Cannot store attribute {} of an object with type
864                 → {}".format(
865                     self.name,
866                     type(self.args[1])))
867
868     def translate(self, cge):
869         if (isinstance(self.args[1], tp.Typable)
870             and isinstance(self.args[1].type.dereference(),
871                 → tp.StructType)):
872             struct_llvm = self.args[1].translate(cge)
873             idx = self.args[1].type.dereference().getMemberIdx(self.name)
874             idx_llvm = tp.getIndex(idx)
875             val_llvm = self.args[0].translate(cge)
876             p = cge.builder.gep(struct_llvm, [tp.Int.constant(0),
877                 → idx_llvm], inbounds=True)
878             self.result.llvm = cge.builder.store(val_llvm, p)
879         else:
880             raise exc.UnimplementedError(type(self.args[1]))
881
882     class CALL_FUNCTION(Bytecode):
883         def __init__(self, func, debuginfo):
884             super().__init__(func, debuginfo)
885
886         def addRawArg(self, arg):
```

Appendix C. STELLA Source Code

```
882     self.num_pos_args = arg & 0xFF
883     self.num_kw_args = (arg >> 8) & 0xFF
884     self.num_stack_args = self.num_pos_args + self.num_kw_args*2
885
886     @property
887     def _str_args(self):
888         return str(self.func)
889
890     def separateArgs(self):
891         self.func = self.args[0]
892         args = self.args[1:]
893
894         assert len(args) == self.num_stack_args
895
896         self.kw_args = {}
897         for i in range(self.num_kw_args):
898             # the list is reversed, so the value comes first
899             value = args.pop()
900             key = args.pop()
901             # key is a Const object, unwrap it
902             self.kw_args[key.value] = value
903
904         # remainder is positional
905         self.args = args
906
907     def stack_eval(self, func, stack):
908         self.stack_bc = []
909         for i in range(self.num_pos_args + 2*self.num_kw_args + 1):
910             arg = stack.pop()
911             self.stack_bc.append(arg)
912         self.stack_bc.reverse()
913
914         stack.push(self)
915
916     def type_eval(self, func):
917         self.grab_stack()
918         self.separateArgs()
919
920         if not isinstance(self.func, (ir.FunctionRef, Intrinsic,
921             ↪ ir.ExtFunctionRef)):
922             # we don't officially know yet that what we're calling is a
923             # function, so install a dummy result and redo the analysis
924             ↪ later
```

Appendix C. STELLA Source Code

```
923         # TODO dead code
924         func.impl.analyzeAgain()
925         self.result = Register(func.impl)
926         return
927
928     if self.result is None or self.result.type == tp.NoType:
929         self.result = self.func.getResult(func.impl)
930
931         if not isinstance(self.func, Intrinsic):
932             func.module.functionCall(self.func, self.args,
933                                     ↪ self.kw_args)
934
935     type_ = self.func.getReturnType(self.args, self.kw_args)
936     tp_change = self.result.unify_type(type_, self.debuginfo)
937
938     if self.result.type == tp.NoType:
939         # abort here because mostly everything downstream will be
940         ↪ unknown types
941         return True
942     else:
943         func.retype(tp_change)
944
945     def translate(self, cge):
946         self.result.llvm = self.func.call(
947             cge,
948             self.args,
949             self.kw_args)
950
951     class GET_ITER(Poison, Bytecode):
952         discard = True
953
954         def __init__(self, func, debuginfo):
955             super().__init__(func, debuginfo)
956
957     class FOR_ITER(Poison, HasTarget, Bytecode):
958         """WIP"""
959         discard = True
960
961         def __init__(self, func, debuginfo):
962             super().__init__(func, debuginfo)
```

Appendix C. STELLA Source Code

```
964
965
966 class JUMP_ABSOLUTE(Jump, Bytecode):
967
968     """WIP"""
969
970     def __init__(self, func, debuginfo):
971         super().__init__(func, debuginfo)
972
973
974 class JUMP_FORWARD(Jump, Bytecode):
975
976     """WIP"""
977
978     def __init__(self, func, debuginfo):
979         super().__init__(func, debuginfo)
980
981
982 class ForLoop(HasTarget, ir.IR):
983     def __init__(self, func, debuginfo):
984         super().__init__(func, debuginfo)
985         self.iterable = None
986
987     def setLoopVar(self, loop_var):
988         self.loop_var = loop_var
989
990     def setLimit(self, limit):
991         self.limit = limit
992
993     def setStart(self, start):
994         self.start = start
995
996     def setEndLoc(self, end_loc):
997         self.target_label = end_loc
998
999     def setTestLoc(self, loc):
1000         self.test_loc = loc
1001
1002     def setIterLoc(self, loc):
1003         """The location of FOR_ITER which may be referenced as 'restart
1004         ↳ loop'"""
1005         self.iter_loc = loc
```

Appendix C. STELLA Source Code

```
1006     def setIterable(self, iterable):
1007         """The LOAD of X which we are iterating over: for _ in X:"""
1008         self.iterable = iterable
1009
1010     def basicSetup(self, bc):
1011         iter_loc = bc.loc
1012         start = None
1013         iterable = None
1014
1015         cur = bc.prev
1016         if not isinstance(cur, GET_ITER):
1017             raise exc.UnimplementedError('unsupported for loop')
1018         cur.remove()
1019         cur = bc.prev
1020         if isinstance(cur, (LOAD_ATTR, LOAD_GLOBAL, LOAD_FAST)):
1021             iterable = cur
1022             limit = Const(0)
1023             cur.remove()
1024             cur = bc.prev
1025             if isinstance(iterable, LOAD_ATTR):
1026                 # LOAD_ATTR requires the object to load, and iterable.prev
1027                 # still refers to it
1028                 cur.remove()
1029
1030                 # iterable should point to the first instruction required
1031                 cur.next = iterable
1032                 iterable = cur
1033
1034             cur = bc.prev
1035         else:
1036             if not isinstance(cur, CALL_FUNCTION):
1037                 raise exc.UnimplementedError('unsupported for loop')
1038             cur.remove()
1039             cur = bc.prev
1040             # TODO: this if..elif should be more general!
1041             if isinstance(cur, LOAD_FAST):
1042                 limit = cur.source
1043                 cur.remove()
1044             elif isinstance(cur, LOAD_CONST):
1045                 limit = cur.const_arg
1046                 cur.remove()
1047             elif isinstance(cur, CALL_FUNCTION):
1048                 cur.remove()
```

Appendix C. STELLA Source Code

```
1049         limit = [cur]
1050         num_args = cur.num_stack_args+1  # +1 for the function
           ↪ name
1051         i = 0
1052         while i < num_args:
1053             cur = cur.prev
1054             # TODO: HACK. How to make this general and avoid
           ↪ duplicating
1055             # stack_eval() knowledge?
1056             if isinstance(cur, LOAD_ATTR):
1057                 # LOAD_ATTR has an argument; num_args is stack
           ↪ values NOT
1058                 # the number of bytecodes which i is counting
1059                 num_args += 1
1060                 cur.remove()
1061                 limit.append(cur)
1062
1063                 i += 1
1064             elif isinstance(cur, LOAD_ATTR):
1065                 limit = [cur, cur.prev]
1066                 cur.prev.remove()
1067                 cur.remove()
1068             else:
1069                 raise exc.UnimplementedError(
1070                     'unsupported for loop: limit {0}'.format(
1071                         type(cur)))
1072         cur = bc.prev
1073
1074         # this supports a start argument to range
1075         if isinstance(cur, LOAD_FAST) or isinstance(cur, LOAD_CONST):
1076             start = cur
1077             cur.remove()
1078             cur = bc.prev
1079
1080         if not isinstance(cur, SETUP_LOOP):
1081             if not isinstance(cur, LOAD_GLOBAL):
1082                 raise exc.UnimplementedError('unsupported for loop')
1083             cur.remove()
1084             cur = bc.prev
1085             if not isinstance(cur, SETUP_LOOP):
1086                 raise exc.UnimplementedError('unsupported for loop')
1087         end_loc = cur.target_label
1088
```

Appendix C. STELLA Source Code

```
1089     self.loc = cur.loc
1090     # TODO set location for self and transfer jumps!
1091     self.setIterable(iterable)
1092     self.setStart(start)
1093     self.setLimit(limit)
1094     self.setEndLoc(end_loc)
1095     self.setTestLoc(bc.loc)
1096     self.setIterLoc(iter_loc)
1097
1098     cur.insert_after(self)
1099     cur.remove()
1100
1101     cur = bc.next
1102     if not isinstance(cur, STORE_FAST):
1103         raise exc.UnimplementedError('unsupported for loop')
1104     loop_var = cur.result
1105     self.setLoopVar(loop_var)
1106     cur.remove()
1107
1108     bc.remove()
1109
1110 def rewrite(self, func):
1111     def load_loop_value(last, after=True):
1112         if isinstance(self.iterable.next, LOAD_ATTR):
1113             b = copy(self.iterable)
1114             if after:
1115                 last.insert_after(b)
1116                 last = b
1117             else:
1118                 last.insert_before(b)
1119             b = copy(self.iterable.next)
1120             if after:
1121                 self.iterable_attr = b
1122             if after:
1123                 last.insert_after(b)
1124                 last = b
1125             else:
1126                 last.insert_before(b)
1127         else:
1128             b = copy(self.iterable)
1129             if after:
1130                 last.insert_after(b)
1131                 last = b
```

Appendix C. STELLA Source Code

```
1132         else:
1133             last.insert_before(b)
1134
1135     b = LOAD_FAST(func.impl, self.debuginfo)
1136     b.addArg(self.loop_var)
1137     if after:
1138         last.insert_after(b)
1139         last = b
1140     else:
1141         last.insert_before(b)
1142
1143     b = BINARY_SUBSCR(func.impl, self.debuginfo)
1144     if after:
1145         last.insert_after(b)
1146         last = b
1147     else:
1148         last.insert_before(b)
1149
1150     b = STORE_FAST(func.impl, self.debuginfo)
1151     b.new_allocate = True
1152     b.addArg(self.loop_value)
1153     if after:
1154         last.insert_after(b)
1155         last = b
1156     else:
1157         last.insert_before(b)
1158
1159     return last
1160
1161     last = self
1162     (self.limit_minus_one, _) = func.impl.getOrNewStackLoc(
1163         str(self.test_loc) + "__limit")
1164     if self.iterable:
1165         self.loop_value = self.loop_var
1166         (self.loop_var, _) = func.impl.getOrNewStackLoc(
1167             self.loop_value.name + "__idx")
1168
1169     # init
1170     if self.start:
1171         b = self.start
1172     else:
1173         b = LOAD_CONST(func.impl, self.debuginfo)
1174         b.addArg(Const(0))
```


Appendix C. STELLA Source Code

```
1175     last.insert_after(b)
1176     last = b
1177
1178     b = STORE_FAST(func.impl, self.debuginfo)
1179     b.addArg(self.loop_var)
1180     b.new_allocate = True
1181     last.insert_after(b)
1182     last = b
1183
1184     # initial test
1185     b = LOAD_FAST(func.impl, self.debuginfo)
1186     b.addArg(self.loop_var)
1187     b.loc = self.test_loc
1188     func.replaceLocation(b)
1189     last.insert_after(b)
1190     last = b
1191
1192     if isinstance(self.limit, (StackLoc, Register)):
1193         b = LOAD_FAST(func.impl, self.debuginfo)
1194         b.addArg(self.limit)
1195         last.insert_after(b)
1196         last = b
1197     elif isinstance(self.limit, Const):
1198         b = LOAD_CONST(func.impl, self.debuginfo)
1199         b.addArg(self.limit)
1200         last.insert_after(b)
1201         last = b
1202     elif isinstance(self.limit, list):
1203         # limit is return value of a function call
1204         for b in reversed(self.limit):
1205             last.insert_after(b)
1206             last = b
1207         b = DUP_TOP(func.impl, self.debuginfo)
1208         last.insert_after(b)
1209         last = b
1210
1211         b = ROT_THREE(func.impl, self.debuginfo)
1212         last.insert_after(b)
1213         last = b
1214     else:
1215         raise exc.UnimplementedError(
1216             "Unsupported limit type {0}".format(
1217                 type(
```

Appendix C. STELLA Source Code

```
1218         self.limit)))
1219
1220     b = COMPARE_OP(func.impl, self.debuginfo)
1221     b.addCmp('>=')
1222     last.insert_after(b)
1223     last = b
1224
1225     b = POP_JUMP_IF_TRUE(func.impl, self.debuginfo)
1226     b.setTarget(self.target_label)
1227     if isinstance(self.limit, list):
1228         b.additionalPop(1)
1229     last.insert_after(b)
1230     last = b
1231
1232     # my_limit = limit -1
1233     if isinstance(self.limit, (StackLoc, Register)):
1234         b = LOAD_FAST(func.impl, self.debuginfo)
1235         b.addArg(self.limit)
1236         last.insert_after(b)
1237         last = b
1238     elif isinstance(self.limit, Const):
1239         b = LOAD_CONST(func.impl, self.debuginfo)
1240         b.addArg(self.limit)
1241         last.insert_after(b)
1242         last = b
1243     elif isinstance(self.limit, list):
1244         # Nothing to do, the value is already on the stack
1245         pass
1246     else:
1247         raise exc.UnimplementedError(
1248             "Unsupported limit type {0}".format(
1249                 type(
1250                     self.limit)))
1251
1252     b = LOAD_CONST(func.impl, self.debuginfo)
1253     b.addArg(Const(1))
1254     last.insert_after(b)
1255     last = b
1256
1257     b = BINARY_SUBTRACT(func.impl, self.debuginfo)
1258     last.insert_after(b)
1259     last = b
1260
```

Appendix C. STELLA Source Code

```
1261     b = STORE_FAST(func.impl, self.debuginfo)
1262     b.addArg(self.limit_minus_one)
1263     b.new_allocate = True
1264     last.insert_after(b)
1265     last = b
1266
1267     if self.iterable:
1268         last = load_loop_value(last)
1269
1270     # lbody, keep, find the end of it
1271     body_loc = b.linearNext().loc
1272     func.addLabel(b.linearNext())
1273
1274     jump_updates = []
1275     while b.next is not None:
1276         if isinstance(b, Jump) and b.target_label == self.iter_loc:
1277             jump_updates.append(b)
1278             b = b.next
1279     assert isinstance(b, utils.BlockEnd)
1280     jump_loc = b.loc
1281     last = b.prev
1282     b.remove()
1283
1284     # go back to the JUMP and switch locations
1285     loop_test_loc = last.loc
1286     last.loc = jump_loc
1287     func.replaceLocation(last)
1288
1289     for b in jump_updates:
1290         b.setTarget(loop_test_loc)
1291
1292     if last.linearPrev().equivalent(last) and isinstance(last,
1293     ↪ JUMP_ABSOLUTE):
1294         # Python seems to sometimes add a duplicate JUMP_ABSOLUTE at
1295         ↪ the
1296         # end of the loop. Remove it.
1297         last.linearPrev().remove()
1298
1299     # loop test
1300     # pdb.set_trace()
1301     b = LOAD_FAST(func.impl, self.debuginfo)
1302     b.addArg(self.loop_var)
1303     b.loc = loop_test_loc
```

Appendix C. STELLA Source Code

```
1302     func.replaceLocation(b)
1303     last.insert_before(b)
1304
1305     b = LOAD_FAST(func.impl, self.debuginfo)
1306     b.addArg(self.limit_minus_one)
1307     last.insert_before(b)
1308
1309     b = COMPARE_OP(func.impl, self.debuginfo)
1310     b.addCmp('>=')
1311     last.insert_before(b)
1312
1313     b = POP_JUMP_IF_TRUE(func.impl, self.debuginfo)
1314     b.setTarget(self.target_label)
1315     last.insert_before(b)
1316
1317     # increment
1318     b = LOAD_FAST(func.impl, self.debuginfo)
1319     b.addArg(self.loop_var)
1320     last.insert_before(b)
1321
1322     b = LOAD_CONST(func.impl, self.debuginfo)
1323     b.addArg(Const(1))
1324     last.insert_before(b)
1325
1326     b = INPLACE_ADD(func.impl, self.debuginfo)
1327     last.insert_before(b)
1328
1329     b = STORE_FAST(func.impl, self.debuginfo)
1330     b.addArg(self.loop_var)
1331     last.insert_before(b)
1332
1333     if self.iterable:
1334         load_loop_value(last, False)
1335
1336     # JUMP to COMPARE_OP is already part of the bytcodes
1337     last.setTarget(body_loc)
1338
1339     def stack_eval(self, func, stack):
1340         # self.result = func.getOrNewRegister(self.loop_var)
1341         # stack.push(self.result)
1342         pass
1343
1344     def translate(self, cge):
```

Appendix C. STELLA Source Code

```
1345     pass
1346
1347     def type_eval(self, func):
1348         self.grab_stack()
1349         if self.iterable:
1350             # TODO if we have an iterable, then we must populate the limit
1351             # here. Yet I am not sure how to detect when this was never
1352             # successful
1353             if isinstance(self.iterable.next, LOAD_ATTR):
1354                 if self.iterable_attr.result:
1355                     iterable = self.iterable_attr.result
1356                 else:
1357                     func.retype()
1358                 return
1359             else:
1360                 iterable = self.iterable.source
1361             if iterable.type != tp.NoType:
1362                 type_ = iterable.type.dereference()
1363                 self.limit.value = type_.shape
1364
1365
1366 class STORE_SUBSCR(Bytecode):
1367     def __init__(self, func, debuginfo):
1368         super().__init__(func, debuginfo)
1369
1370     @pop_stack(3)
1371     def stack_eval(self, func, stack):
1372         self.result = None
1373
1374     def type_eval(self, func):
1375         self.grab_stack()
1376
1377     def translate(self, cge):
1378         if self.args[1].type.isReference():
1379             type_ = self.args[1].type.dereference()
1380         else:
1381             type_ = self.args[1].type
1382         type_.storeSubscript(cge, self.args[1], self.args[2],
1383                               ↪ self.args[0])
1384
1385 class BINARY_SUBSCR(Bytecode):
1386     def __init__(self, func, debuginfo):
```

Appendix C. STELLA Source Code

```
1387         super().__init__(func, debuginfo)
1388         self.result = Register(func)
1389
1390     @pop_stack(2)
1391     def stack_eval(self, func, stack):
1392         stack.push(self)
1393
1394     def type_eval(self, func):
1395         self.grab_stack()
1396         if self.args[0].type.isReference():
1397             arg_type = self.args[0].type.dereference()
1398         else:
1399             arg_type = self.args[0].type
1400         if not isinstance(arg_type, tp.Subscriptable):
1401             raise exc.TypeError(
1402                 "Type must be subscriptable, but got {0}".format(
1403                     self.args[0].type))
1404         self.result.unify_type(
1405             arg_type.getElementType(self.args[1]),
1406             self.debuginfo)
1407
1408     def translate(self, cge):
1409         if self.args[0].type.isReference():
1410             type_ = self.args[0].type.dereference()
1411         else:
1412             type_ = self.args[0].type
1413         self.result.llvm = type_.loadSubscript(cge, self.args[0],
1414             ↪ self.args[1])
1414
1415
1416     class POP_TOP(Bytecode):
1417         discard = True
1418
1419     def __init__(self, func, debuginfo):
1420         super().__init__(func, debuginfo)
1421
1422     @pop_stack(1)
1423     def stack_eval(self, func, stack):
1424         pass
1425
1426     def type_eval(self, func):
1427         self.grab_stack()
1428
```

Appendix C. STELLA Source Code

```
1429     def translate(self, cge):
1430         pass
1431
1432
1433 class DUP_TOP(Bytecode):
1434     discard = True
1435
1436     def __init__(self, func, debuginfo):
1437         super().__init__(func, debuginfo)
1438
1439     @pop_stack(1)
1440     def stack_eval(self, func, stack):
1441         stack.push(self.stack_bc[0])
1442         stack.push(self.stack_bc[0])
1443
1444     def type_eval(self, func):
1445         self.grab_stack()
1446
1447     def translate(self, cge):
1448         pass
1449
1450
1451 class DUP_TOP_TWO(Bytecode):
1452     discard = True
1453
1454     def __init__(self, func, debuginfo):
1455         super().__init__(func, debuginfo)
1456
1457     @pop_stack(2)
1458     def stack_eval(self, func, stack):
1459         stack.push(self.stack_bc[0])
1460         stack.push(self.stack_bc[1])
1461         stack.push(self.stack_bc[0])
1462         stack.push(self.stack_bc[1])
1463
1464     def type_eval(self, func):
1465         self.grab_stack()
1466
1467     def translate(self, cge):
1468         pass
1469
1470
1471 class ROT_TWO(Bytecode, Poison):
```

Appendix C. STELLA Source Code

```
1472     discard = True
1473
1474     def __init__(self, func, debuginfo):
1475         super().__init__(func, debuginfo)
1476
1477     @pop_stack(2)
1478     def stack_eval(self, func, stack):
1479         stack.push(self.stack_bc[1])
1480         stack.push(self.stack_bc[0])
1481
1482     def type_eval(self, func):
1483         self.grab_stack()
1484
1485     def translate(self, cge):
1486         pass
1487
1488
1489 class ROT_THREE(Bytecode, Poison):
1490     discard = True
1491
1492     def __init__(self, func, debuginfo):
1493         super().__init__(func, debuginfo)
1494
1495     @pop_stack(3)
1496     def stack_eval(self, func, stack):
1497         stack.push(self.stack_bc[2])
1498         stack.push(self.stack_bc[0])
1499         stack.push(self.stack_bc[1])
1500
1501     def type_eval(self, func):
1502         self.grab_stack()
1503
1504     def translate(self, cge):
1505         pass
1506
1507
1508 class UNARY_NEGATIVE(Bytecode):
1509     b_func = {tp.Float: 'fsub', tp.Int: 'sub'}
1510
1511     def __init__(self, func, debuginfo):
1512         super().__init__(func, debuginfo)
1513         self.result = Register(func)
1514
```


Appendix C. STELLA Source Code

```
1515     @pop_stack(1)
1516     def stack_eval(self, func, stack):
1517         stack.push(self)
1518
1519     def type_eval(self, func):
1520         self.grab_stack()
1521         arg = self.args[0]
1522         self.result.unify_type(arg.type, self.debuginfo)
1523
1524     def builderFuncName(self):
1525         try:
1526             return self.b_func[self.result.type]
1527         except KeyError:
1528             raise exc.TypeError(
1529                 "{0} does not yet implement type {1}".format(
1530                     self.__class__.__name__,
1531                     self.result.type))
1532
1533     def translate(self, cge):
1534         self.cast(cge)
1535         f = getattr(cge.builder, self.builderFuncName())
1536         self.result.llvm = f(
1537             self.result.type.constant(0),
1538             self.args[0].translate(cge))
1539
1540
1541     class UNPACK_SEQUENCE(Bytecode):
1542         n = 0
1543
1544         def __init__(self, func, debuginfo):
1545             super().__init__(func, debuginfo)
1546
1547         def addRawArg(self, arg):
1548             self.n = arg
1549
1550     @pop_stack(1)
1551     def stack_eval(self, func, stack):
1552         self.result = []
1553         for i in range(self.n):
1554             reg = Register(func)
1555             stack.push(self)
1556             self.result.append(reg)
1557
```

Appendix C. STELLA Source Code

```
1558     def type_eval(self, func):
1559         self.grab_stack()
1560         i = 0
1561         for reg in reversed(self.result):
1562             reg.unify_type(self.args[0].type.getElementType(i),
1563                           ↪ self.debuginfo)
1564             i += 1
1565
1566     def translate(self, cge):
1567         if self.args[0].type.isReference():
1568             type_ = self.args[0].type.dereference()
1569         else:
1570             type_ = self.args[0].type
1571         i = 0
1572         for reg in reversed(self.result):
1573             reg.llvm = type_.loadSubscript(cge, self.args[0], i)
1574             i += 1
1575
1576     class BUILD_TUPLE(Bytecode):
1577         n = 0
1578
1579         def __init__(self, func, debuginfo):
1580             super().__init__(func, debuginfo)
1581
1582         def addRawArg(self, arg):
1583             self.n = arg
1584
1585         def stack_eval(self, func, stack):
1586             self.stack_bc = []
1587             for i in range(self.n):
1588                 self.stack_bc.append(stack.pop())
1589             stack.push(self)
1590
1591         def type_eval(self, func):
1592             self.grab_stack()
1593             self.args.reverse()
1594             if not self.result:
1595                 self.result = tp.Tuple(self.args)
1596             else:
1597                 self.result.unify_type(tp.TupleType([arg.type for arg in
1598                                                       ↪ self.args]),
1599                                       self.debuginfo)
```

Appendix C. STELLA Source Code

```
1599
1600     def translate(self, cge):
1601         self.result.translate(cge)
1602
1603
1604 class RAISE_VARARGS(Bytecode):
1605     """TODO will abort the program with a crash"""
1606     n = 0
1607
1608     def __init__(self, func, debuginfo):
1609         super().__init__(func, debuginfo)
1610
1611     def addRawArg(self, arg):
1612         self.n = arg
1613
1614     def stack_eval(self, func, stack):
1615         for i in range(self.n):
1616             stack.pop()
1617
1618     def type_eval(self, func):
1619         self.grab_stack()
1620
1621     def translate(self, cge):
1622         llvm_f = cge.module.llvm.declare_intrinsic('llvm.trap', [])
1623         cge.builder.call(llvm_f, [])
1624
1625
1626 class UNARY_NOT(Bytecode):
1627     def __init__(self, func, debuginfo):
1628         super().__init__(func, debuginfo)
1629         self.result = Register(func)
1630
1631     @pop_stack(1)
1632     def stack_eval(self, func, stack):
1633         stack.push(self)
1634
1635     def type_eval(self, func):
1636         self.grab_stack()
1637         arg = self.args[0]
1638         if arg.type in (tp.Int, tp.Float):
1639             self.args[0] = Cast(arg, tp.Bool)
1640
1641         self.result.unify_type(tp.Bool, self.debuginfo)
```

Appendix C. STELLA Source Code

```
1642
1643     def translate(self, cge):
1644         self.cast(cge)
1645         self.result.llvm = cge.builder.xor(
1646             tp.Bool.constant(1),
1647             self.args[0].translate(cge))
1648
1649
1650 class BINARY_AND(BinaryOp):
1651     b_func = {tp.Bool: 'and_', tp.Int: 'and_' }
1652
1653
1654 class BINARY_OR(BinaryOp):
1655     b_func = {tp.Bool: 'or_', tp.Int: 'or_' }
1656
1657
1658 class BINARY_XOR(BinaryOp):
1659     b_func = {tp.Bool: 'xor', tp.Int: 'xor' }
1660
1661
1662 opconst = {}
1663 # Get all concrete subclasses of Bytecode and register them
1664 for name in dir(sys.modules[__name__]):
1665     obj = sys.modules[__name__].__dict__[name]
1666     try:
1667         if issubclass(obj, Bytecode) and len(obj.__abstractmethods__) ==
1668             ↪ 0:
1669             opconst[dis.opmap[name]] = obj
1670     except TypeError:
1671         pass
```

C.9 stella/test/debug_gc.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
```

Appendix C. STELLA Source Code

```
7 # http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import gc
15 import stella
16 import stella.ir
17 from . import langconstr
18 # import types
19
20
21 def check():
22     print("Collecting {}".format(gc.collect()))
23
24     for obj in filter(lambda x: isinstance(x, stella.ir.Module),
25 ↪ gc.get_objects()):
26         print ('-'*48)
27         print("{} | {} : in={}, out={}".format(
28             str(obj), repr(obj), len(gc.get_referrers(obj)),
29             ↪ len(gc.get_referrers(obj))))
30         for r in gc.get_referrers(obj):
31             print(" < {}".format(type(r)))
32             if isinstance(r, list):
33                 if len(r) > 20:
34                     print(" ", len(r))
35                     continue
36                 print(" ", r)
37             # for rr in gc.get_referrers(r):
38             #     print(" < {}".format(type(rr)))
39             # for r in gc.get_referents(obj):
40             #     print(" > {}".format(type(r)))
41
42 # with bug
43 r = stella.wrap(langconstr.kwargs_call1)(1)
44 print(r)
45 # with bug
46 #r = stella.wrap(langconstr.call_void)()
47 #print(r)
48 # no bug
49 #r = stella.wrap(langconstr.array_alloc_use)()
```

Appendix C. STELLA Source Code

```
48 #print(r)
49
50 print ('='*78)
51 check()
52
53 print ('='*78)
54 print("Garbage: ", len(gc.garbage), any([isinstance(x, stella.ir.Module)
    ↪   for x in gc.garbage]))
55 for m in filter(lambda x: isinstance(x, stella.ir.Module), gc.garbage):
56     import pdb; pdb.set_trace() # XXX BREAKPOINT
57
58 print('-'*78)
59 check()
```

C.10 stella/test/errors.py

```
1 #!/usr/bin/env python
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 #     http://www.apache.org/licenses/LICENSE-2.0
9 #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 from stella import exc, wrap
17 from . import * # noqa
18 from stella.intrinsics.python import zeros
19
20
21 def undefined1():
22     if False:
23         r = 1
```

Appendix C. STELLA Source Code

```
24     return r
25
26
27 def undefined2():
28     if False:
29         x = 1
30     y = 0 + x # noqa
31     return True
32
33
34 def zeros_no_type():
35     a = zeros(5) # noqa
36
37
38 @mark.parametrize('f', [undefined1, undefined2])
39 def test_undefined(f):
40     make_exc_test(f, (), UnboundLocalError, exc.UnDEFINEDError)
41
42
43 def third(t):
44     return t[2]
45
46
47 def callThird():
48     t = (4, 2)
49     return third(t)
50
51
52 def array_alloc_const_index_out_of_bounds():
53     a = zeros(5, dtype=int)
54     a[5] = 42
55
56
57 def array_alloc_var_index_out_of_bounds():
58     """This tests causes a segmentation fault."""
59     a = zeros(5, dtype=int)
60     i = 5
61     a[i] = 42
62
63
64 @mark.parametrize('f', [callThird, array_alloc_const_index_out_of_bounds])
65 def test_indexerror(f):
66     make_exc_test(f, (), IndexError, exc.IndexError)
```

Appendix C. STELLA Source Code

```
67
68
69 @mark.parametrize('f', [array_alloc_var_index_out_of_bounds])
70 @unimplemented
71 def test_indexerror_segfault(f):
72     """Would crash"""
73     make_exc_test(f, (), IndexError, exc.IndexError)
74
75
76 class TestException(Exception):
77     pass
78
79
80 def raise_exc1():
81     raise TestException('foo')
82
83
84 def raise_exc2():
85     raise Exception('foo')
86
87
88 @mark.parametrize('f_exc', [(raise_exc1, TestException), (raise_exc2,
89     ↪ Exception)])
89 @unimplemented
90 def test_exception(f_exc):
91     """
92     Note: this isn't a real test. The NotImplementedError is thrown during
93     _compile-time_, not _run-time_!
94     """
95     f, exc = f_exc
96
97     with raises(exc):
98         f()
99
100     with raises(NotImplementedError):
101         wrap(f)()
```

C.11 stella/test/external_func.py

Appendix C. STELLA Source Code

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 from random import randint
15
16 import mtpy
17
18 from . import * # noqa
19
20
21 def seed_const():
22     mtpy.mt_seed32new(42)
23
24
25 def seed(s):
26     mtpy.mt_seed32new(s)
27
28
29 def drand_const():
30     mtpy.mt_seed32new(42)
31     return mtpy.mt_drand()
32
33
34 def drand(s):
35     mtpy.mt_seed32new(s)
36     return mtpy.mt_drand() + mtpy.mt_drand()
37
38
39 @mark.parametrize('f', [seed_const, drand_const])
40 def test1(f):
41     make_eq_test(f, ())
42
43
```

Appendix C. STELLA Source Code

```
44 @mark.parametrize('arg', single_args([1, 2, 42, 1823828, randint(1,
    ↪ 10000000),
45                                     randint(1, 10000000)]))
46 @mark.parametrize('f', [seed, drand])
47 def test2(f, arg):
48     make_eq_test(f, arg)
```

C.12 stella/test/benchmark.py

```
1 #!/usr/bin/env python
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 #     http://www.apache.org/licenses/LICENSE-2.0
9 #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 import os
17 import os.path
18 from subprocess import check_output, Popen, PIPE
19 import time
20 import functools
21 import numpy
22
23 import pystache
24
25 from . import * # noqa
26 import stella
27
28 opt = 3
29 min_speedup = 0.75
30
```

Appendix C. STELLA Source Code

```
31
32 def ccompile(fn, src, cc=None, flags={}):
33     """
34     Write the string src into the file fn, then compile it with -O{opt}
    ↪ and
35     return the executable name.
36     """
37     with open(fn, 'w') as f:
38         f.write(src)
39
40     if 'c' not in flags:
41         flags['c'] = []
42     if 'ld' not in flags:
43         flags['ld'] = []
44
45     if cc is None:
46         if 'CC' in os.environ:
47             CC = os.environ['CC']
48         else:
49             CC = 'gcc'
50     else:
51         CC = cc
52
53     (root, ext) = os.path.splitext(fn)
54     if os.path.exists(root):
55         os.unlink(root)
56     obj = root + ".o"
57     if os.path.exists(obj):
58         os.unlink(obj)
59     with open(fn, 'rb') as f:
60         sourcecode = f.read()
61
62     # the following three cmds are equivalent to
63     # [CC, '-Wall', '-O' + str(opt)] + flags + ['-o', root, fn]
64
65     cmd = [CC] + flags['c'] + ['-Wall', '-E', '-o', '-', '-']
66     print("Preprocessing: {0}".format(" ".join(cmd)))
67     p = Popen(cmd, stdin=PIPE, stdout=PIPE, stderr=PIPE)
68     preprocessed, serr = p.communicate(timeout=30, input=sourcecode)
69     assert (not serr or not serr.decode())
70
71     # start with C input, generate assembly
72     cmd = [CC, '-Wall'] + flags['c'] + ['-x', 'cpp-output', '-S',
```

Appendix C. STELLA Source Code

```
73         '-o' + str(opt), '-o', '-', '-']
74     print("Compiling to assembly: {0}".format(" ".join(cmd)))
75
76     p = Popen(cmd, stdin=PIPE, stdout=PIPE, stderr=PIPE)
77
78     time_start = time.time()
79     sout, serr = p.communicate(timeout=30, input=preprocessed)
80     elapsed = time.time() - time_start
81
82     assert not serr.decode()
83
84     cmd = [CC] + flags['ld'] + ['-o', root, '-x', 'assembler', '-']
85     print("Compiling to machine code & linking: {0}".format("
86     ↪ ".join(cmd)))
87     p = Popen(cmd, stdin=PIPE, stdout=PIPE, stderr=PIPE)
88     sout, serr = p.communicate(timeout=30, input=sout)
89     assert (not serr or not serr.decode()) and (not sout or not
90     ↪ sout.decode())
91
92     return root, elapsed
93
94 def bench_it(name, c_src, args, extended, parse_f, verify_f,
95             stella_f=None, full_f=None,
96             flags={}):
97     """args = {k=v, ...}
98     Args gets expanded to 'k'_init: 'k'='v' for the C template
99     """
100    if not stella_f and not full_f:
101        raise Exception(
102            "Either need to specify stella_f(*arg_value) or full_f(args,
103            ↪ stats)")
104
105    t_run = {}
106    t_compile = {}
107
108    c_args = {k+'_init': k+'='+str(v) for k, v in args.items()}
109    print("Doing {0}({1})".format(name, args))
110    src = pystache.render(c_src, **c_args)
111
112    if extended:
113        CCs = ['gcc', 'clang']
114    else:
```

Appendix C. STELLA Source Code

```
113     CCs = ['gcc']
114
115     results = {}
116
117     for cc in CCs:
118         exe, elapsed_compile = ccompile(__file__ + "." + name + ".c", src,
119             ↪ cc, flags)
120         t_compile[cc] = elapsed_compile
121
122         cmd = [exe]
123
124         print("Running C/{:}: {}".format(cc, " ".join(cmd)))
125         time_start = time.time()
126         out = check_output(cmd, universal_newlines=True)
127         print(out)
128         results[cc] = parse_f(out)
129         elapsed_c = time.time() - time_start
130         t_run[cc] = elapsed_c
131
132     print("Running Stella:")
133     stats = {}
134     wrapper_opts = {'debug': False, 'opt': opt, 'stats': stats}
135     if stella_f:
136         arg_values = args.values()
137         time_start = time.time()
138         res = stella.wrap(stella_f, **wrapper_opts)(*arg_values)
139         elapsed_stella = time.time() - time_start
140     else:
141         elapsed_stella, res = full_f(args, stella.wrap, wrapper_opts)
142
143     results['stella'] = res
144
145     t_run['stella'] = stats['elapsed']
146     # TODO no need to keep track of the combined time, is there?
147     # t_run['stella+compile'] = elapsed_stella
148     t_compile['stella'] = elapsed_stella - stats['elapsed']
149
150     if extended > 1:
151         print("\nRunning Python:")
152         if stella_f:
153             time_start = time.time()
154             res = stella_f(*[v for k, v in args.items()])
155             elapsed_py = time.time() - time_start
```

Appendix C. STELLA Source Code

```
155         else:
156             elapsed_py, res = full_f(args, time_stats, wrapper_opts)
157             t_run['python'] = elapsed_py
158             results['python'] = res
159
160
161         # verify results are identical
162         it = iter(results.keys())
163         k1 = next(it)
164         for k2 in it:
165             print('Verify:', k1, '==', k2)
166             verify_f(results[k1], results[k2])
167             k1 = k2
168
169
170         return {'run': t_run, 'compile': t_compile}
171
172
173 def fib_prepare(f):
174     @functools.wraps(f)
175     def prepare(args):
176         return (f, (args['x'], ), lambda r, x: r)
177     return prepare
178
179
180 def fib_parse(out):
181     print (out)
182     return int(out.strip())
183
184
185 def fib_verify(a, b):
186     assert a == b
187
188
189 def bench_fib(duration, extended):
190     from .langconstr import fib
191
192     args = {'x': duration}
193
194     return bench_vs_template(fib_prepare(fib), extended, 'fib', args,
195                             parse_f=fib_parse, verify_f=fib_verify)
196
197
```

Appendix C. STELLA Source Code

```
198 def bench_fib_nonrecursive(duration, extended):
199     from .langconstr import fib_nonrecursive
200
201     args = {'x': duration}
202
203     return bench_vs_template(fib_prepare(fib_nonrecursive), extended,
204         ↪ 'fib_nonrecursive', args,
205         parse_f=fib_parse, verify_f=fib_verify)
206
207 def bench_vs_template(prepare, extended, name, args, parse_f, verify_f,
208     ↪ flags={}):
209     fn = "{}/template.{}.{}.c".format(os.path.dirname(__file__),
210         os.path.basename(__file__),
211         name)
212
213     with open(fn) as f:
214         src = f.read()
215
216     def run_it(args, wrapper, wrapper_opts):
217         run_f, transfer, result_f = prepare(args)
218         if transfer is None:
219             transfer = []
220
221         time_start = time.time()
222         r = wrapper(run_f, **wrapper_opts)(*transfer)
223         elapsed_stella = time.time() - time_start
224
225         return elapsed_stella, result_f(r, *transfer)
226
227     return bench_it(name, src, args, extended, flags=flags, full_f=run_it,
228         parse_f=parse_f, verify_f=verify_f)
229
230 def bench_si111s(module, extended, suffix, duration):
231     def parse(out):
232         return numpy.array(list(map(float, out.strip()[1:-1].split(' '))))
233
234     def verify(a, b):
235         assert (a == b).all()
236
237     args = {'seed': int(time.time() * 100) % (2**32),
238         'rununtiltime': duration
239     }
```

Appendix C. STELLA Source Code

```
239     return bench_vs_template(module.prepare, extended, 'sillls_' + suffix,
    ↪     args,
240                                 flags={'ld': ['-lm']}),
241                                 parse_f=parse, verify_f=verify)
242
243
244 def bench_sillls_globals(duration, extended):
245     from . import sillls_globals
246     return bench_sillls(sillls_globals, extended, 'globals', duration)
247
248
249 def bench_sillls_struct(duration, extended):
250     from . import sillls_struct
251     return bench_sillls(sillls_struct, extended, 'struct', duration)
252
253
254 def bench_sillls_obj(duration, extended):
255     from . import sillls_obj
256     # reuse the 'struct' version of C since there is no native OO
257     return bench_sillls(sillls_obj, extended, 'struct', duration)
258
259
260 def bench_nbody(n, extended):
261     from . import nbody
262
263     def parse(out):
264         return list(map(float, out.strip().split('\n')))
265
266     def verify(a, b):
267         fmt = "{:8f}"
268         for x, y in zip(a, b):
269             assert fmt.format(x) == fmt.format(y)
270
271     args = {'n': n,
272            'dt': 0.01,
273            }
274     return bench_vs_template(nbody.prepare, extended, 'nbody', args,
    ↪     flags={'ld': ['-lm']}),
275                                 parse_f=parse, verify_f=verify)
276
277
278 def bench_heat(n, extended):
279     from . import heat
```


Appendix C. STELLA Source Code

```
280
281 def parse(out):
282     rows = out.strip().split('\n')
283     r = numpy.zeros(shape=(len(rows), 5))
284
285     for i, row in enumerate(rows):
286         for j, v in enumerate(row.split()):
287             r[i, j] = v
288
289     return r
290
291 def verify(a, b):
292     for i, row in enumerate(abs(a - b)):
293         assert (row < delta).all()
294
295     args = {'nsteps': n}
296     return bench_vs_template(heat.prepare, extended, 'heat', args,
297                             flags={'ld': ['-lm'], 'c': ['-std=c99']}},
298                             parse_f=parse, verify_f=verify)
299
300
301 def speedup(bench):
302     return bench['run']['gcc'] / bench['run']['stella']
303
304
305 @bench
306 def test_fib(bench_result, bench_opt, bench_ext):
307     duration = [30, 45, 48][bench_opt]
308     bench_result['fib'] = bench_fib(duration, bench_ext)
309     assert speedup(bench_result['fib']) >= min_speedup
310
311
312 @mark.skipif(True, reason="Runs too fast to be a useful benchmark")
313 def test_fib_nonrecursive(bench_result, bench_opt, bench_ext):
314     duration = [50, 150, 175][bench_opt]
315     bench_result['fib_nonrec'] = bench_fib_nonrecursive(duration,
316                                                         ↪ bench_ext)
317     assert speedup(bench_result['fib_nonrec']) >= min_speedup
318
319
320 sillis_durations = ['1e5', '1e8', '1.2e9']
321
```

Appendix C. STELLA Source Code

```
322 @bench
323 def test_si111s_globals(bench_result, bench_opt, bench_ext):
324     duration = si111s_durations[bench_opt]
325     bench_result['si111s_global'] = bench_si111s_globals(duration,
326     ↪ bench_ext)
327     assert speedup(bench_result['si111s_global']) >= min_speedup
328
329 @bench
330 def test_si111s_struct(bench_result, bench_opt, bench_ext):
331     duration = si111s_durations[bench_opt]
332     bench_result['si111s_struct'] = bench_si111s_struct(duration,
333     ↪ bench_ext)
334     assert speedup(bench_result['si111s_struct']) >= min_speedup
335
336 @bench
337 def test_si111s_obj(bench_result, bench_opt, bench_ext):
338     duration = si111s_durations[bench_opt]
339     bench_result['si111s_obj'] = bench_si111s_obj(duration, bench_ext)
340     assert speedup(bench_result['si111s_obj']) >= min_speedup
341
342
343 @bench
344 def test_nbody(bench_result, bench_opt, bench_ext):
345     duration = [250000, 10000000, 100000000][bench_opt]
346     bench_result['nbody'] = bench_nbody(duration, bench_ext)
347     assert speedup(bench_result['nbody']) >= min_speedup
348
349
350 @bench
351 def test_heat(bench_result, bench_opt, bench_ext):
352     duration = [13, 3000, 50000][bench_opt]
353     bench_result['heat'] = bench_heat(duration, bench_ext)
354     assert speedup(bench_result['heat']) >= min_speedup
```

C.13 stella/test/basicmath.py

Appendix C. STELLA Source Code

```
1 #!/usr/bin/env python
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 # http://www.apache.org/licenses/LICENSE-2.0
9 #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 from random import randint
17 from . import * # noqa
18 import math
19
20
21 def addition(a, b):
22     return a + b
23
24
25 def subtraction(a, b):
26     return a - b
27
28
29 def multiplication(a, b):
30     return a * b
31
32
33 def division(a, b):
34     return a / b
35
36
37 def floor_division(a, b):
38     return a // b
39
40
41 def modulo(a, b):
42     return a % b
43
```

Appendix C. STELLA Source Code

```
44
45 def power1(a, b):
46     return a ** b
47
48
49 def power2(a, b):
50     return pow(a, b)
51
52
53 def power3(a, b):
54     return math.pow(a, b)
55
56
57 def chained(a, b):
58     return (a - b) / b * a
59
60
61 def logarithm(x):
62     return math.log(x)
63
64
65 def exponential(x):
66     return math.exp(x)
67
68
69 def unary_neg(x):
70     return -x
71
72
73 def inplace(a, b):
74     x = a
75     x += b
76     x /= b
77     x -= b
78     x *= b
79     return x
80
81 arglist1 = [(-1, 0), (84, -42), (1.0, 1), (0, 1), (randint(0, 1000000),
82     ↪ randint(0, 1000000)),
83             (-1 * randint(0, 1000000), randint(0, 1000000))]
84
85 @mark.parametrize('args', arglist1)
```

Appendix C. STELLA Source Code

```
86 @mark.parametrize('f', [addition, subtraction, multiplication])
87 def test1(f, args):
88     make_eq_test(f, args)
89
90 arglist2 = [(0, 1), (5, 2), (5.2, 2), (4.0, 4), (-5, 2), (5.0, -2),
91             (3, 1.5), (randint(0, 1000000), randint(1, 1000000)), (341433,
92             ↪ 673069)]
93
94 @mark.parametrize('args', arglist2)
95 @mark.parametrize('f', [division, floor_division])
96 def test2(f, args):
97     make_delta_test(f, args)
98
99
100 @mark.parametrize('args', arglist2)
101 @mark.parametrize('f', [chained, inplace])
102 def test_accuracy(f, args):
103     """Note: Lower accuracy"""
104     make_delta_test(f, args, delta=1e-6)
105
106
107 @mark.parametrize('args', filter(lambda e: e[0] >= 0, arglist2))
108 def test_modulo(args):
109     """Note: Lower accuracy"""
110     make_delta_test(modulo, args, delta=1e-6)
111
112
113 @mark.parametrize('args', filter(lambda e: e[0] < 0, arglist2))
114 @mark.xfail(raises=AssertionError)
115 def test_semantics_modulo(args):
116     """Semantic difference:
117     ↪ Modulo always has the sign of the divisor in Python, unlike C where it
118     ↪ is
119     ↪ the sign of the dividend.
120     ↪ """
121     make_delta_test(modulo, args)
122
123 arglist3 = [(0, 42), (42, 0), (2, 5.0), (2.0, 5), (1.2, 2), (4, 7.5), (-4,
124 ↪ 2)]
125 @mark.parametrize('args', arglist3)
```

Appendix C. STELLA Source Code

```
126 @mark.parametrize('f', [power1, power2, power3])
127 def test3(f, args):
128     make_delta_test(f, args)
129
130
131 @mark.parametrize('args', [(4, -2)])
132 @mark.parametrize('f', [power1, power2, power3])
133 @mark.xfail(raises=AssertionError)
134 def test_semantics_power(f, args):
135     """Semantic difference:
136     4**2 returns an integer, but 4**-2 returns a float.
137     """
138     make_delta_test(f, args)
139
140
141 @mark.parametrize('args', single_args([1, 2, 42, 1.5, 7.9]))
142 @mark.parametrize('f', [logarithm, exponential])
143 def test4(f, args):
144     make_delta_test(f, args)
145
146
147 @mark.parametrize('args', single_args([1, 2, 42, 1.5, 7.9, randint(1,
148     ↪ 1000000), 0, -4, -999999]))
149 @mark.parametrize('f', [unary_neg])
150 def test5(f, args):
151     make_delta_test(f, args)
```

C.14 stella/test/virtnet_utils.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
```

Appendix C. STELLA Source Code

```
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 from types import FunctionType
15 import time
16
17 class Settings(object):
18     def setDefaults(self):
19         self.settings = {
20             'seed'      : [int(time.time()), int],
21             'r'         : [0.1, float],
22             'koff'      : [1.0, float],
23             'radius'   : [10, int],
24             'nlegs'    : [2, int],
25             'gait'     : [2, int],
26             'dim'      : [2, int],
27             'nspiders' : [1, int],      # not completely functional
                → yet
28             'elapsedTime':[self.elapsedTime, lambda x:x],
29         }
30     def elapsedTime(self):
31         return time.time() - self.start_time
32
33     def __init__(self, argv = []):
34         self.start_time = time.time()
35
36         self.setDefaults()
37
38         if isinstance(argv, dict):
39             for k, v in argv.items():
40                 self[k] = v
41         else:
42             # parse command line arguments to overwrite the defaults
43             for key, _, val in [s.partition('=') for s in argv]:
44                 self[key] = val
45
46     def __setitem__(self, k, v):
47         if k in self.settings:
48             self.settings[k][0] = self.settings[k][1](v)
49         else:
50             self.settings[k] = [v, type(v)]
51
52     def __getitem__(self, k):
```

Appendix C. STELLA Source Code

```
53         return self.settings[k][0]
54
55     def __str__(self):
56         r = '{'
57         for k,(v,type_) in self.settings.items():
58             if isinstance(type_, FunctionType):
59                 continue
60             r += str(k) + ':' + str(v) + ', '
61     return r[:-2] + '}'
```

C.15 stella/test/langconstr.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import numpy as np
15
16 from . import * # noqa
17 from stella.intrinsics.python import zeros
18 import stella
19 from .basicmath import addition, subtraction
20 from . import basicmath
21
22
23 def direct_assignment(x, y):
24     a = x
25     return a + y
26
27
```


Appendix C. STELLA Source Code

```
28 def simple_assignment(x, y):
29     a = x + y
30     return a
31
32
33 def return_const():
34     return 41
35
36
37 def assign_const():
38     r = 42
39     return r
40
41
42 def double_assignment(x, y):
43     a = x
44     b = 5 + y
45     a += b
46     return a
47
48
49 def double_cast(x, y):
50     a = x / y
51     b = y // x
52     return a + b
53
54
55 def simple_if(x):
56     if x:
57         return 0
58     else:
59         return 42
60
61
62 def simple_ifeq(x, y):
63     if x == y:
64         return 0
65     else:
66         return 42
67
68
69 def simple_ifeq_const(x):
70     if x == False: # noqa TODO: support 'is' here!
```

Appendix C. STELLA Source Code

```
71     return 0
72 else:
73     return 42
74
75
76 def op_not(x):
77     return not x
78
79
80 def for1(x):
81     r = 0
82     for i in range(x):
83         r += i
84     return r
85
86
87 def for2(x):
88     r = 0
89     s = 1
90     for i in range(x):
91         r += i
92         s *= 2
93     return r + s
94
95
96 def for_loop_var(x):
97     for i in range(x):
98         x = i
99     return x
100
101
102 def for3(a):
103     r = 0
104     for x in a:
105         r += x
106     return r
107
108
109 def while1(x):
110     r = 0
111     while x > 0:
112         r += x
113         x -= 1
```

Appendix C. STELLA Source Code

```
114     return r
115
116
117 def recursive(x):
118     if x <= 0:
119         return 1
120     else:
121         return x + recursive(x - 1)
122
123
124 def fib(x):
125     if x <= 2:
126         return 1
127     return fib(x - 1) + fib(x - 2)
128
129
130 def fib_nonrecursive(n):
131     if n == 0:
132         return 1
133     if n == 1:
134         return 1
135     grandparent = 1
136     parent = 1
137     me = 0 # required for stella only
138     for i in range(2, n):
139         me = parent + grandparent
140         grandparent = parent
141         parent = me
142     return me
143
144
145 def hof_f(n):
146     if n == 0:
147         return 1
148     else:
149         return n - hof_m(hof_f(n - 1))
150
151
152 def hof_m(n):
153     if n == 0:
154         return 0
155     else:
156         return n - hof_f(hof_m(n - 1))
```

Appendix C. STELLA Source Code

```
157
158
159 def and_(a, b):
160     return a and b
161
162
163 def or_(a, b):
164     return a or b
165
166 some_global = 0
167
168
169 def use_global():
170     global some_global
171     some_global = 0
172     x = 5
173     while some_global == 0:
174         x = global_test_worker(x)
175     return x
176
177
178 def global_test_worker(x):
179     global some_global
180     if x < 0:
181         some_global = 1
182     return x - 1
183
184
185 def new_global_const():
186     global prev_undefined
187     prev_undefined = 1
188
189
190 def new_global_var(x):
191     global prev_undefined
192     prev_undefined = x
193     return prev_undefined # TODO: / 2 fails!
194
195
196 def kwargs(a=0, b=1):
197     return a + b
198
199
```

Appendix C. STELLA Source Code

```
200 def kwargs_call1(x):
201     return kwargs(a=x)
202
203
204 def kwargs_call2(x):
205     return kwargs(b=x)
206
207
208 def kwargs_call3(x):
209     return kwargs(a=1, b=x)
210
211
212 def kwargs_call4(x):
213     return kwargs(a=x, b=x)
214
215
216 def return_without_init(x, y):
217     if y > 0:
218         return addition(x, y)
219     else:
220         return subtraction(x, y)
221
222
223 def ext_call(x):
224     return basicmath.subtraction(0, x)
225
226
227 def array_allocation():
228     a = zeros(5, dtype=int) # noqa
229     return 0
230
231
232 def array_allocation_reg():
233     """
234     Since memory allocation is not a focus right now,
235     this test will be skipped indefinitely.
236     """
237     l = 2
238     a = zeros(l, dtype=int) # noqa
239     return 0
240
241
242 def array_alloc_assignment():
```

Appendix C. STELLA Source Code

```
243     a = zeros(5, dtype=int)
244     i = 0
245     a[0] = i
246
247
248 def array_alloc_assignment2():
249     a = zeros(5, dtype=int)
250     for i in range(5):
251         a[i] = 42
252
253
254 def array_alloc_assignment3():
255     a = zeros(5, dtype=int)
256     for i in range(5):
257         a[i] = i + 1
258
259
260 def void():
261     pass
262
263
264 def call_void():
265     void()
266     return 1
267
268
269 def array_alloc_use():
270     a = zeros(5, dtype=int)
271     a[0] = 1
272     return a[0]
273
274
275 def array_alloc_use2():
276     a = zeros(5, dtype=int)
277     for i in range(5):
278         a[i] = i ** 2
279     r = 0
280     for i in range(5):
281         r += a[i]
282     return r
283
284
285 def array_len():
```

Appendix C. STELLA Source Code

```
286     a = zeros(5, dtype=int)
287     return len(a)
288
289
290 def numpy_array(a):
291     a[1] = 4
292     a[2] = 2
293     a[3] = -1
294
295
296 def numpy_assign(a):
297     b = a
298     b[1] = 4
299
300
301 def numpy_len_indirect(a):
302     l = len(a)
303     for i in range(l):
304         a[i] = i + 1
305
306
307 def numpy_len_direct(a):
308     for i in range(len(a)):
309         a[i] = i + 1
310
311
312 def numpy_passing(a):
313     a[0] = 3
314     a[2] = 1
315     numpy_receiving(a)
316
317
318 def numpy_receiving(a):
319     l = len(a)
320     for i in range(l):
321         if a[i] > 0:
322             a[i] += 1
323
324
325 def numpy_global():
326     global numpy_global_var
327     numpy_global_var[3] = 4
328     numpy_global_var[4] = 2
```

Appendix C. STELLA Source Code

```
329
330
331 def numpy_array2d1(a):
332     a[0, 0] = 1
333     a[0, 1] = 2
334     a[1, 0] = 3
335     a[1, 1] = 4
336
337
338 def numpy_array2d2(a):
339     return a[0, 0] * a[1, 1] + a[1, 0] * a[0, 1]
340
341
342 def numpy_array2d_for1(a):
343     r = 0
344     for i in range(2):
345         for j in range(2):
346             r += a[i, j]
347     return r
348
349
350 def numpy_array2d_shape(a):
351     return a.shape
352
353
354 def numpy_array2d_for2(a):
355     maxx = a.shape[0]
356     maxy = a.shape[1]
357     r = 0
358     for i in range(maxx):
359         for j in range(maxy):
360             r += 1
361     return r
362
363
364 def numpy_array2d_for3(a, b):
365     maxx = a.shape[0]
366     maxy = a.shape[1]
367     r = 0
368     for i in range(maxx):
369         for j in range(maxy):
370             r += 1
371             b[i, j] += r
```


Appendix C. STELLA Source Code

```
372     return r
373
374
375 def numpy_array2d_for4(a):
376     maxx = a.shape[0]
377     maxy = a.shape[1]
378     r = 0
379     for i in range(maxx):
380         for j in range(maxy):
381             r += a[i, j]
382     return r
383
384
385 def return_2():
386     return 2
387
388
389 def if_func_call():
390     return return_2() > 1
391
392
393 def numpy_func_limit(a):
394     for i in range(return_2()):
395         a[i] = i + 1
396
397
398 def return_tuple():
399     return (4, 2)
400
401
402 def first(t):
403     return t[0]
404
405
406 def callFirst():
407     t = (4, 2)
408     return first(t)
409
410
411 def second(t):
412     return t[1]
413
414
```

Appendix C. STELLA Source Code

```
415 def firstPlusSecond():
416     t = (4, 2)
417     return first(t) + second(t)
418
419
420 def getReturnedTuple1():
421     t = return_tuple()
422     return first(t)
423
424
425 def getReturnedTuple2():
426     x, _ = return_tuple()
427     return x
428
429
430 def switchTuple():
431     x, y = (1, 2)
432     y, x = x, y
433
434     return x - y
435
436
437 def createTuple1():
438     x = 1
439     t1 = (x, -1)
440     return t1
441
442
443 def createTuple2():
444     x = 7
445     t2 = (-2, x)
446     return t2
447
448
449 def createTuple3():
450     x = 1
451     t1 = (x, -1)
452     t2 = (t1[1], x)
453     return t1[0], t2[0]
454
455
456 def iterateTuple():
457     t = (4, 6, 8, 10)
```

Appendix C. STELLA Source Code

```
458     r = 0
459     for i in t:
460         r += i
461     return r
462
463
464 def addTuple(t):
465     return t[0] + t[1]
466
467
468 def bitwise_and(a, b):
469     return a & b
470
471
472 def bitwise_or(a, b):
473     return a | b
474
475
476 def bitwise_xor(a, b):
477     return a ^ b
478
479
480 def tuple_me(a):
481     return tuple(a)
482
483
484 def lt(x, y):
485     return x < y
486
487
488 def gt(x, y):
489     return x > y
490
491
492 def le(x, y):
493     return x <= y
494
495
496 def ge(x, y):
497     return x >= y
498
499
500 def ne(x, y):
```

Appendix C. STELLA Source Code

```
501     return x != y
502
503
504 def eq(x, y):
505     return x == y
506
507
508 ###
509
510 @mark.parametrize('args', [(40, 2), (43, -1), (41, 1)])
511 @mark.parametrize('f', [direct_assignment, simple_assignment,
512     ↪ double_assignment, double_cast,
513     ↪ return_without_init])
514 def test1(f, args):
515     make_eq_test(f, args)
516
517 @mark.parametrize('args', [(True, True), (True, False), (False, True),
518     ↪ (False, False)])
519 @mark.parametrize('f', [and_, or_])
520 def test2(f, args):
521     make_eq_test(f, args)
522
523 @mark.parametrize('arg', single_args([True, False]))
524 @mark.parametrize('f', [simple_if, simple_ifeq_const, op_not])
525 def test3(f, arg):
526     make_eq_test(f, arg)
527
528
529 @mark.parametrize('args', [(True, False), (True, True), (4, 2), (4.0,
530     ↪ 4.0)])
531 @mark.parametrize('f', [simple_ifeq])
532 def test4(f, args):
533     make_eq_test(f, args)
534
535 @mark.parametrize('f', [return_const, assign_const, use_global,
536     ↪ array_allocation,
537     ↪ array_alloc_assignment, array_alloc_assignment2,
538     ↪ array_alloc_assignment3,
539     ↪ void, call_void, array_alloc_use,
540     ↪ array_alloc_use2, array_len,
```

Appendix C. STELLA Source Code

```
538                                     if_func_call])
539 def test5(f):
540     make_eq_test(f, ())
541
542
543 @mark.parametrize('f', [array_allocation_reg])
544 @unimplemented
545 def test5b(f):
546     make_eq_test(f, ())
547
548
549 @mark.parametrize('arg', single_args([0, 1, 2, 3, 42, -1, -42]))
550 @mark.parametrize('f', [for1, for2, for_loop_var, while1, recursive,
    ↪ ext_call, kwargs_call1,
551                               kwargs_call2, kwargs_call3, kwargs_call4, op_not])
552 def test6(f, arg):
553     make_eq_test(f, arg)
554
555
556 @mark.parametrize('arg', single_args([0, 1, 2, 5, 8, -1, -3]))
557 @mark.parametrize('f', [fib, fib_nonrecursive])
558 def test7(f, arg):
559     make_eq_test(f, arg)
560
561
562 @mark.parametrize('f', [kwargs])
563 def test8(f):
564     make_eq_test(f, (1, 30))
565
566
567 @mark.parametrize('arg', single_args([0, 1, 2, 5, 8, 12]))
568 @mark.parametrize('f', [hof_f])
569 def test9(f, arg):
570     make_eq_test(f, arg)
571
572
573 @mark.parametrize('args', [{ 'a': 1}, { 'b': 2}, { 'a': 1, 'b': 0}, { 'b': 1,
    ↪ 'a': 0}, { 'a': 1.2},
574                               { 'b': -3}, {}])
575 def test10(args):
576     make_eq_kw_test(kwargs, args)
577
578
```

Appendix C. STELLA Source Code

```
579 @mark.parametrize('args', [{'c': 5}, {'b': -1, 'c': 5}])
580 @mark.xfail()
581 def test11(args):
582     make_eq_kw_test(kwargs, args)
583
584
585 @mark.parametrize('arg', single_args([np.zeros(5, dtype=int)]))
586 @mark.parametrize('f', [numpy_array, numpy_len_indirect, numpy_receiving,
    ↪ numpy_passing,
587                        numpy_len_direct, numpy_assign])
588 def test12(f, arg):
589     make_eq_test(f, arg)
590
591
592 @mark.parametrize('arg', single_args([np.zeros(5, dtype=int)]))
593 @mark.parametrize('f', [])
594 @unimplemented
595 def test12u(f, arg):
596     make_eq_test(f, arg)
597
598
599 def test13():
600     global numpy_global_var
601
602     orig = np.zeros(5, dtype=int)
603
604     numpy_global_var = np.array(orig)
605     py = numpy_global()
606     py_res = numpy_global_var
607
608     numpy_global_var = orig
609     st = stella.wrap(numpy_global)()
610     st_res = numpy_global_var
611
612     assert py == st
613     assert all(py_res == st_res)
614
615
616 def test13b():
617     """Global scalars are currently not updated in Python when their value
    ↪ changes in Stella"""
618     global some_global
619
```

Appendix C. STELLA Source Code

```
620     some_global = 0
621     py = use_global()
622     assert some_global == 1
623
624     some_global = 0
625     st = stella.wrap(use_global)()
626     assert some_global == 0
627
628     assert py == st
629
630
631 def test13c():
632     """Defining a new (i.e. not in Python initialized) global variable
633
634     and initialize it with a constant
635     """
636     global prev_undefined
637
638     assert 'prev_undefined' not in globals()
639     py = new_global_const()
640     assert 'prev_undefined' in globals()
641
642     del prev_undefined
643     assert 'prev_undefined' not in globals()
644     st = stella.wrap(new_global_const)()
645     # Note: currently no variable updates are transferred back to Python
646     assert 'prev_undefined' not in globals()
647
648     assert py == st
649
650
651 def test13d():
652     """Defining a new (i.e. not in Python initialized) global variable
653
654     and initialize it with another variable
655     """
656     global prev_undefined
657
658     assert 'prev_undefined' not in globals()
659     py = new_global_var(42)
660     assert 'prev_undefined' in globals()
661
662     del prev_undefined
```

Appendix C. STELLA Source Code

```
663     assert 'prev_undefined' not in globals()
664     st = stella.wrap(new_global_var)(42)
665     # Note: currently no variable updates are transferred back to Python
666     assert 'prev_undefined' not in globals()
667
668     assert py == st
669
670
671     @mark.parametrize('f', [callFirst, firstPlusSecond, getReturnedTuple1,
672                             getReturnedTuple2, return_tuple, switchTuple,
673                             createTuple1, createTuple2, createTuple3])
674     def test14(f):
675         make_eq_test(f, ())
676
677
678     @mark.parametrize('f', [iterateTuple])
679     @unimplemented
680     def test14_u(f):
681         make_eq_test(f, ())
682
683
684     @mark.parametrize('arg', single_args([(10, 20), (4.0, 2.0), (13.0, 14)]))
685     @mark.parametrize('f', [addTuple])
686     def test15(f, arg):
687         make_eq_test(f, arg)
688
689
690     @mark.parametrize('arg', single_args([np.array([1, 2, 5, 7]),
691     ↪ np.array([-1, -2, 0, 45]),
692                                         np.array([1.0, 9.0, -3.14, 0.0001,
693     ↪ 11111.0])]))
694
695     @mark.parametrize('f', [for3])
696     def test16(f, arg):
697         make_numpy_eq_test(f, arg)
698
699
700     array2d_args = single_args([np.zeros((2, 2), dtype=int),
701                                np.array([[4, 3], [2, -1]]),
702                                np.array([[1.5, 2.5, 5.5], [-3.3, -5.7,
703     ↪ 1.1]]),
704                                np.array([[42.0, 4.2], [5, 7], [0, 123]])
705                                ])
706
```


Appendix C. STELLA Source Code

```
703
704 @mark.parametrize('arg', array2d_args)
705 @mark.parametrize('f', [numpy_array2d1, numpy_array2d2,
706     ↪ numpy_array2d_for1, numpy_array2d_for2,
707     ↪ numpy_array2d_for4])
708 def test17(f, arg):
709     make_numpy_eq_test(f, arg)
710
711 @mark.parametrize('arg', array2d_args)
712 @mark.parametrize('f', [])
713 @unimplemented
714 def test17u(f, arg):
715     make_numpy_eq_test(f, arg)
716
717
718 @mark.parametrize('arg', array2d_args)
719 @mark.parametrize('f', [numpy_array2d_for3])
720 def test18(f, arg):
721     arg2 = np.zeros(arg[0].shape)
722     make_numpy_eq_test(f, (arg[0], arg2))
723
724
725 @mark.parametrize('args', [(40, 2), (43, 1), (42, 3), (0, 0), (2, 2), (3,
726     ↪ 3), (3, 4), (4, 7),
727     ↪ (True, True), (True, False), (False, False),
728     ↪ (False, True)])
729 @mark.parametrize('f', [bitwise_and, bitwise_or, bitwise_xor])
730 def test19(f, args):
731     make_eq_test(f, args)
732
733 # TODO Who needs arrays longer than 2?
734 # @mark.parametrize('arg', single_args([np.zeros(5, dtype=int),
735     ↪ np.zeros(3), np.array([1, 2, 42]),
736     ↪ np.array([0.0, 3.0])]))
737 # @mark.parametrize('arg', single_args([np.zeros(2, dtype=int), np.zeros(2),
738     ↪ np.array([1, 42]),
739     ↪ np.array([0.0, 3.0])]))
740 @mark.parametrize('f', [tuple_me])
741 def test20(f, arg):
742     make_numpy_eq_test(f, arg)
```

Appendix C. STELLA Source Code

```
741
742 @mark.parametrize('args', [(40, 2), (43, 1), (42, 3), (0, 0), (2, 2), (3,
    ↪ 3), (3, 4), (4, 7),
743                               (1.0, 0), (1.2, 2.0), (1, 2.3)])
744 @mark.parametrize('f', [lt, gt, eq, le, ge, ne])
745 def test19(f, args):
746     make_eq_test(f, args)
```

C.16 stella/test/sillls_struct.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import time
15 from math import log, exp
16 from random import randint
17
18 import numpy as np
19
20 from . import * # noqa
21 import mtpy
22 import stella
23 from . import virtnet_utils
24
25 EXPSTART = 0.2
26 class Spider(object):
27     def __init__(self, params, observations):
28         self.K = params['K']
29         self.rununtiltime = params['rununtiltime']
```

Appendix C. STELLA Source Code

```
30     mtpy.mt_seed32new(params['seed'])
31     self.koffp = params['koffp']
32     self.kcat = params['r']
33
34     self.delta = (log(self.rununtiltime) - log(EXPSTART)) /
35     ↪ float(self.K - 1)
36     self.leg = 0
37     self.substrate = 0
38     self.obs_i = 0
39     self.observations = observations
40     # LANG: Init below required before entering stella!
41     # TODO: Static analysis could discover the use in the original
42     ↪ location
43     self.t = 0.0
44     self.next_obs_time = 0.0
45 def __eq__(self, other):
46     return ((self.observations == other.observations).all() and
47             self.obs_i == other.obs_i and
48             self.t == other.t and
49             self.next_obs_time == other.next_obs_time)
50
51 def __str__(self):
52     return "{:~:~}>".format(super().__str__()[:-1], self.observations)
53
54 def uniform():
55     return mtpy.mt_drand()
56
57 def mtpy_exp(p):
58     u = 1.0 - uniform()
59     return -log(u) / p
60
61
62 def makeObservation(sp):
63     """Called from run()"""
64     sp.observations[sp.obs_i] = sp.leg
65     sp.obs_i += 1
66
67     sp.next_obs_time = getNextObsTime(sp)
68
69
70 def getNextObsTime(sp):
```

Appendix C. STELLA Source Code

```
71     """Called from run()"""
72     if sp.obs_i == 0:
73         return EXPSTART
74     if sp.obs_i == sp.K - 1:
75         return sp.rununtiltime
76
77     return exp(log(EXPSTART) + sp.delta * sp.obs_i)
78
79
80 def step(sp):
81     """Called from run()"""
82     if sp.leg == 0:
83         sp.leg += 1
84     else:
85         u1 = uniform()
86         if u1 < 0.5:
87             sp.leg -= 1
88         else:
89             sp.leg += 1
90     if sp.leg == sp.substrate:
91         sp.substrate += 1
92
93
94 def isNextObservation(sp):
95     return sp.t > sp.next_obs_time and sp.obs_i < sp.K
96
97
98 def run(sp):
99     # LANG: Init below moved to Spider.__init__
100    #sp.t = 0.0
101    sp.next_obs_time = getNextObsTime(sp)
102
103    # TODO: Declaring R here is not necessary in Python! But llvm needs a
104    # it because otherwise the definition of R does not dominate the use
105    ↪ below.
106    R = 0.0
107    while sp.obs_i < sp.K and sp.t < sp.rununtiltime:
108        if sp.leg < sp.substrate:
109            R = sp.koffp
110        else:
111            R = sp.kcat
112            sp.t += mtpy_exp(R)
```

Appendix C. STELLA Source Code

```
113         while isNextObservation(sp):
114             makeObservation(sp)
115
116         step(sp)
117
118
119 class Settings(virtnet_utils.Settings):
120     def setDefaults(self):
121         self.settings = {
122             'seed': [int(time.time()), int],
123             'r': [0.1, float],
124             'koffp': [1.0, float],
125             'K': [10, int],
126             'rununtiltime': [1e3, float],
127             'elapsedTime': [self.elapsedTime, lambda x:x],
128         }
129
130
131 def prototype(params):
132     s = Settings(params)
133
134     py = np.zeros(shape=s['K'], dtype=int)
135     sp_py = Spider(s, py)
136     run(sp_py)
137
138     st = np.zeros(shape=s['K'], dtype=int)
139     sp_st = Spider(s, st)
140     stella.wrap(run)(sp_st)
141
142     assert id(sp_py.observations) != id(sp_st.observations)
143     assert sp_py == sp_st
144
145
146 def prepare(args):
147     params = Settings([k+'='+str(v) for k, v in args.items()])
148     sp_py = Spider(params, np.zeros(shape=params['K'], dtype=int))
149
150     def get_results(r, sp):
151         print (sp.observations)
152         return sp.observations
153
154     return (run, (sp_py, ), get_results)
155
```

Appendix C. STELLA Source Code

```
156 @mark.parametrize('args', [['seed=42'], ['seed=63'], ['seed=123456'],
157                             ['rununtiltime=1e4', 'seed=494727'],
158                             ['seed={}'.format(randint(1, 100000))]])
159 def test1(args):
160     prototype(args)
161
162 timed = timeit(prototype, verbose=True)
163
164
165 def bench1():
166     timed(['seed=42', 'rununtiltime=1e8'])
```

C.17 stella/test/objects.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import numpy as np
15
16 from . import * # noqa
17 import stella
18
19
20 class B(object):
21     x = 0
22     y = 0
23
24     def __init__(self, x=1, y=2):
25         self.x = x
```

Appendix C. STELLA Source Code

```
26     self.y = y
27
28     def __eq__(self, other):
29         return self.x == other.x and self.y == other.y
30
31     def __ne__(self, other):
32         return not self.__eq__(other)
33
34     def __repr__(self):
35         return "{}:{}", {}>".format(str(type(self))[:-1], self.x, self.y)
36
37
38 class C(object):
39     """
40     %"class 'test.objects.C'<Int*6>_Int" = type { [6 x i64]*, i64 }
41     """
42     def __init__(self, obj, i=0):
43         if isinstance(obj, int):
44             self.a = np.zeros(shape=obj, dtype=int)
45             self.a[0] = 42
46         else:
47             self.a = np.array(obj)
48         self.i = i
49
50     def __eq__(self, other):
51         return self.i == other.i and (self.a == other.a).all()
52
53     def __ne__(self, other):
54         return not self.__eq__(other)
55
56     def __repr__(self):
57         return "{}:{}", {}>".format(str(type(self))[:-1], self.i, self.a)
58
59
60 class D(object):
61     z = 0
62     a = 0
63     y = 0.0
64     g = 0.0
65
66     def __init__(self):
67         pass
68
```

Appendix C. STELLA Source Code

```
69     def __eq__(self, other):
70         return (self.z == other.z and
71               self.a == other.a and
72               self.y == other.y and
73               self.g == other.g)
74
75     def __ne__(self, other):
76         return not self.__eq__(other)
77
78     def __repr__(self):
79         return "{}: {}>".format(str(type(self))[:-1], [self.z, self.a,
80               ↪ self.y, self.g])
81
82 class E(object):
83     def __init__(self, x=0):
84         self.x = x
85
86     def inc(self, p=1):
87         self.x += p
88         return self.x
89
90     def __eq__(self, other):
91         return (self.x == other.x)
92
93     def __ne__(self, other):
94         return not self.__eq__(other)
95
96     def __repr__(self):
97         return "{}[x={}]".format(str(type(self))[8:-2], self.x)
98
99
100 class F(object):
101     def __init__(self, l):
102         self.l = l
103
104     def __eq__(self, other):
105         return (self.l == other.l)
106
107     def __ne__(self, other):
108         return not self.__eq__(other)
109
110     def __repr__(self):
```


Appendix C. STELLA Source Code

```
111         return "{}[1={}]".format(str(type(self))[8:-2], self.l)
112
113
114 class G(B):
115     def __init__(self, x=1, y=2):
116         super().__init__(x, y)
117
118
119 class H(object):
120     def __init__(self, a, b, c):
121         self.es = [E(a), E(b), E(c)]
122         self.i = 0
123
124     def next(self):
125         e = self.es[self.i]
126         self.i = (self.i + 1) % len(self.es)
127         return e
128
129     def __eq__(self, other):
130         return self.es == other.es and self.i == other.i
131
132     def __ne__(self, other):
133         return not self.__eq__(other)
134
135     def __repr__(self):
136         return "H{{es: {}, i:{}}}".format(self.es, self.i)
137
138
139 class J(object):
140     def __init__(self, t):
141         self.t = t
142
143     def __eq__(self, other):
144         return self.t == other.t
145
146     def __ne__(self, other):
147         return not self.__eq__(other)
148
149     def __repr__(self):
150         return "J{{t: {}}}".format(self.t)
151
152
153 G.origin = G(0, 0)
```

Appendix C. STELLA Source Code

```
154
155
156 class K(object):
157     def notstatic():
158         return 1
159
160     @staticmethod
161     def static():
162         return 2
163
164
165 def justPassing(a):
166     x = 1 # noqa
167
168
169 def cmpAttrib(a):
170     return a.x == a.y
171
172
173 def setAttrib(a):
174     a.x = 42
175
176
177 def setAttribFloat(a):
178     a.x = 42.0
179
180
181 def setUnknownAttrib(a):
182     a.z = 42
183
184
185 def getAttrib(a):
186     return a.x
187
188
189 def addAttribs(a):
190     return a.x + a.y
191
192
193 def returnUnknownAttrib(a):
194     return a.z
195
196
```

Appendix C. STELLA Source Code

```
197 def getAndSetAttrib1(a):
198     a.x *= a.y
199
200
201 def getAndSetAttrib2(a):
202     a.x -= 1
203
204
205 def callBoundMethod(e):
206     e.inc()
207     return e.x
208
209
210 def callBoundMethod2(e):
211     e.inc(42)
212
213
214 def callBoundMethod3(e, x):
215     e.inc(x)
216
217
218 def callBoundMethodTwice(e, x):
219     e.inc(x)
220     e.inc(x)
221
222
223 def callBoundMethodOnTwo(e1, e2):
224     e1.inc(1)
225     e2.inc(2)
226
227
228 def objList1(l):
229     return l[0].x + l[1].x
230
231
232 def objList2(l):
233     r = 0
234     for i in range(len(l)):
235         r += l[i].x
236     return r
237
238
239 def objList3(l):
```

Appendix C. STELLA Source Code

```
240     r = 0
241     for i in range(len(l)):
242         for j in range(len(l)):
243             r += l[j].x + i
244     return r
245
246
247 def objList4(l):
248     for i in range(len(l)):
249         l[i].x = i
250
251
252 def first(l):
253     return l[0]
254
255
256 def objList5(l):
257     o = first(l)
258     return o.x
259
260
261 def objContainingList1(f):
262     return f.l[0].x + f.l[1].x
263
264
265 def objContainingList2(f):
266     r = 0
267     for i in range(len(f.l)):
268         r += f.l[i].x
269     return r
270
271
272 def objContainingList3(f):
273     for i in range(len(f.l)):
274         f.l[i].x = i
275
276
277 def selfRef(g):
278     return ((g.x - G.origin.x)**2 + (g.y - G.origin.y)**2)**0.5
279
280
281 def nextB(b):
282     return b.x == b.next.x and b.y == b.next.y
```

Appendix C. STELLA Source Code

```
283
284
285 def getObjThenUse(h):
286     e = h.next()
287     return e.x
288
289
290 def getObjThenCall(h):
291     e = h.next()
292     return e.inc()
293
294
295 def forObjAttr(c):
296     r = 1
297     for x in c.a:
298         r *= x
299     return r
300
301
302 def forObjAttrRange(c):
303     c.i = len(c.a)
304     r = 1
305     for x in range(c.i):
306         r += x
307     return r
308
309
310 def getFirstArrayValue(c):
311     return c.a[0]
312
313
314 def getSomeArrayValue(c, i):
315     return c.a[i]
316
317
318 def sumC(c):
319     for i in range(len(c.a)):
320         c.i += c.a[i]
321
322
323 def returnObj(o):
324     return o
325
```

Appendix C. STELLA Source Code

```
326
327 def select(item, truth):
328     """This will only work when item is a pointer"""
329     if truth:
330         return item
331     else:
332         return None
333
334
335 def objTuple1(j):
336     return j.t[0]
337
338 args1 = [(1, 1), (24, 42), (0.0, 1.0), (1.0, 1.0), (3.0, 0.0)]
339
340
341 @mark.parametrize('f', [justPassing, addAttribs, getAttrib])
342 @mark.parametrize('args', args1)
343 def test_no_mutation(f, args):
344     b1 = B(*args)
345     b2 = B(*args)
346
347     assert b1 == b2
348     py = f(b1)
349     st = stella.wrap(f)(b2)
350
351     assert b1 == b2 and py == st
352
353
354 @mark.parametrize('f', [])
355 @unimplemented
356 def test_no_mutation_u(f):
357     b1 = B()
358     b2 = B()
359
360     assert b1 == b2
361     py = f(b1)
362     st = stella.wrap(f)(b2)
363
364     assert b1 == b2 and py == st
365
366
367 @mark.parametrize('f', [setAttrib])
368 def test_mutation(f):
```

Appendix C. STELLA Source Code

```
369     b1 = B()
370     b2 = B()
371
372     assert b1 == b2
373     py = f(b1)
374     st = stella.wrap(f)(b2)
375
376     assert b1 != B() and b1 == b2 and py == st
377
378
379     @mark.parametrize('f', [setAttribFloat])
380     @mark.xfail(raises=TypeError)
381     def test_mutation_f(f):
382         """
383         The opposite, setting an int when the struct member is float does not
384         raise a TypeError since the int will be promoted to a float.
385         """
386         b1 = B()
387         b2 = B()
388
389         assert b1 == b2
390         py = f(b1)
391         st = stella.wrap(f)(b2)
392
393         assert b1 != B() and b1 == b2 and py == st
394
395
396     @mark.parametrize('args', args1)
397     @mark.parametrize('f', [cmpAttrib, getAndSetAttrib1, getAndSetAttrib2])
398     def test_mutation2(f, args):
399         b1 = B(*args)
400         b2 = B(*args)
401
402         assert b1 == b2
403         py = f(b1)
404         st = stella.wrap(f)(b2)
405
406         assert b1 == b2 and py == st
407
408
409     @mark.parametrize('args', [])
410     @mark.parametrize('f', [])
411     @unimplemented
```

Appendix C. STELLA Source Code

```
412 def test_mutation2_u(f, args):
413     b1 = B(*args)
414     b2 = B(*args)
415
416     assert b1 == b2
417     py = f(b1)
418     st = stella.wrap(f)(b2)
419
420     assert b1 == b2 and py == st
421
422
423 @mark.parametrize('f', [returnUnknownAttrib])
424 @mark.xfail(raises=AttributeError)
425 def test_mutation2_f(f):
426     b1 = B()
427     b2 = B()
428
429     assert b1 == b2
430     py = f(b1)
431     st = stella.wrap(f)(b2)
432
433     assert b1 == b2 and py == st
434
435
436 args2 = [(1, 2, 3, 4), (1.0, 2.0, 3.0)]
437 args3 = list(zip(args2, [0, 0.0]))
438
439
440 @mark.parametrize('f', [getFirstArrayValue, sumC])
441 @mark.parametrize('args', args3)
442 def test_no_mutation2(f, args):
443     b1 = C(*args)
444     b2 = C(*args)
445
446     assert b1 == b2
447     py = f(b1)
448     st = stella.wrap(f)(b2)
449
450     assert b1 == b2 and py == st
451
452
453 @mark.parametrize('f', [forObjAttr, forObjAttrRange])
454 @mark.parametrize('args', args2)
```


Appendix C. STELLA Source Code

```
455 def test_no_mutation2_u(f, args):
456     b1 = C(args)
457     b2 = C(args)
458
459     assert b1 == b2
460     py = f(b1)
461     st = stella.wrap(f)(b2)
462
463     assert b1 == b2 and py == st
464
465
466 @mark.parametrize('f', [getSomeArrayValue])
467 @mark.parametrize('args', args2)
468 def test_no_mutation3(f, args):
469     b1 = C(args)
470     b2 = C(args)
471
472     assert b1 == b2
473     py = f(b1, 1)
474     st = stella.wrap(f)(b2, 1)
475
476     assert b1 == b2 and py == st
477
478
479 def manipulate_d1(d):
480     d.z = 1
481     d.a = 2
482     d.y = 3.0
483     d.g = 4.0
484
485
486 def pass_struct(d):
487     manipulate_d1(d)
488
489
490 @mark.parametrize('f', [manipulate_d1, pass_struct])
491 def test_mutation3(f):
492     b1 = D()
493     b2 = D()
494
495     assert b1 == b2
496     py = f(b1)
497     st = stella.wrap(f)(b2)
```

Appendix C. STELLA Source Code

```
498
499     assert b1 == b2 and py == st
500
501
502 @mark.parametrize('f', [callBoundMethod, callBoundMethod2])
503 def test_mutation4(f):
504     e1 = E()
505     e2 = E()
506
507     assert e1 == e2
508
509     py = f(e1)
510     st = stella.wrap(f)(e2)
511
512     assert e1 == e2 and py == st
513
514
515 @mark.parametrize('f', [callBoundMethod3, callBoundMethodTwice])
516 @mark.parametrize('arg', [0, -1, 5])
517 def test_mutation5(f, arg):
518     e1 = E()
519     e2 = E()
520
521     assert e1 == e2
522
523     py = f(e1, arg)
524     st = stella.wrap(f)(e2, arg)
525
526     assert e1 == e2 and py == st
527
528
529 @mark.parametrize('f', [callBoundMethodOnTwo])
530 def test_mutation6(f):
531     e1 = E()
532     e2 = E()
533     e3 = E()
534     e4 = E()
535
536     assert e1 == e2 and e3 == e4
537
538     py = f(e1, e3)
539     st = stella.wrap(f)(e2, e4)
540
```

Appendix C. STELLA Source Code

```
541     assert e1 == e2 and e3 == e4 and py == st
542
543
544 @mark.parametrize('f', [objList1, objList2, objList3, objList5])
545 def test_no_mutation7(f):
546     l1 = [E(4), E(1)]
547     l2 = [E(4), E(1)]
548
549     py = f(l1)
550     st = stella.wrap(f)(l2)
551
552     assert l1 == l2 and py == st
553
554
555 @mark.parametrize('f', [objList4])
556 def test_mutation7(f):
557     l1 = [E(4), E(1)]
558     l2 = [E(4), E(1)]
559
560     py = f(l1)
561     st = stella.wrap(f)(l2)
562
563     assert l1 == l2 and py == st
564
565
566 @mark.parametrize('f', [objContainingList1, objContainingList2])
567 def test_no_mutation8(f):
568     l1 = [E(2), E(5)]
569     l2 = [E(2), E(5)]
570     f1 = F(l1)
571     f2 = F(l2)
572
573     py = f(f1)
574     st = stella.wrap(f)(f2)
575
576     assert f1 == f2 and py == st
577
578
579 @mark.parametrize('f', [objContainingList3])
580 def test_mutation8(f):
581     l1 = [E(2), E(5)]
582     l2 = [E(2), E(5)]
583     f1 = F(l1)
```

Appendix C. STELLA Source Code

```
584     f2 = F(12)
585
586     py = f(f1)
587     st = stella.wrap(f)(f2)
588
589     assert f1 == f2 and py == st
590
591
592 args3 = [(4, 8), (9.0, 27.0)]
593
594
595 @mark.parametrize('f', [nextB])
596 @mark.parametrize('args', args3)
597 def test_no_mutation9(f, args):
598     b1 = B(*args)
599     b2 = B(*args)
600
601     b1.next = b1
602     b2.next = b2
603
604     assert b1 == b2
605     py = f(b1)
606     st = stella.wrap(f)(b2)
607
608     assert b1 == b2 and py == st
609
610
611 @mark.parametrize('f', [selfRef])
612 @mark.parametrize('args', args3)
613 def test_no_mutation10(f, args):
614     b1 = G(*args)
615     b2 = G(*args)
616
617     assert b1 == b2
618     py = f(b1)
619     st = stella.wrap(f)(b2)
620
621     assert b1 == b2 and py == st
622
623
624 @mark.parametrize('f', [getObjThenUse, getObjThenCall])
625 def test1(f):
626     b1 = H(1, 2, 3)
```

Appendix C. STELLA Source Code

```
627     b2 = H(1, 2, 3)
628
629     assert b1 == b2
630     py = f(b1)
631     st = stella.wrap(f)(b2)
632
633     assert b1 == b2 and py == st
634
635
636     @mark.parametrize('f', [])
637     @unimplemented
638     def test_no_mutation11u(f):
639         b1 = H(1, 2, 3)
640         b2 = H(1, 2, 3)
641
642         assert b1 == b2
643         py = f(b1)
644         st = stella.wrap(f)(b2)
645
646         assert b1 == b2 and py == st
647
648
649     @mark.parametrize('f', [returnObj])
650     def test_no_mutation12(f):
651         b1 = H(1, 2, 3)
652         b2 = H(1, 2, 3)
653
654         assert b1 == b2
655         py = f(b1)
656         st = stella.wrap(f)(b2)
657
658         assert b1 == b2 and py == st
659
660
661     @mark.parametrize('f', [select])
662     @mark.parametrize('arg', [True, False])
663     def test_no_mutation13(f, arg):
664         b1 = H(1, 2, 3)
665         b2 = H(1, 2, 3)
666
667         assert b1 == b2
668         py = f(b1, arg)
669         st = stella.wrap(f)(b2, arg)
```

Appendix C. STELLA Source Code

```
670
671     assert b1 == b2 and py == st
672
673
674 @mark.parametrize('f', [ObjTuple1])
675 @mark.parametrize('arg', [(42, -1)])
676 @unimplemented
677 def test_no_mutation14(f, arg):
678     b1 = J(arg)
679     b2 = J(arg)
680
681     assert b1 == b2
682     py = f(b1, arg)
683     st = stella.wrap(f)(b2, arg)
684
685     assert b1 == b2 and py == st
686
687
688 @mark.parametrize('f', [K.static])
689 @unimplemented
690 def test_no_mutation15(f):
691     py = f()
692     st = stella.wrap(f)()
693
694     assert b1 == b2 and py == st
```

C.18 stella/test/confstest.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

Appendix C. STELLA Source Code

```
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import pytest
15 from collections import defaultdict
16
17
18 def pytest_addoption(parser):
19     parser.addoption('-B', "--bench", action="store",
20                     type=str, default=False,
21                     help="run benchmark tests: veryshort, short, or
22                          ↪ long")
23     parser.addoption('-E', "--extended-bench", action="count",
24                     default=False,
25                     help="run also extended benchmark tests: in Python,
26                          ↪ and with clang")
27
28
29 results = defaultdict(dict)
30
31 @pytest.fixture(scope="module")
32 def bench_result():
33     return results
34
35 def pytest_runtest_setup(item):
36     if 'bench' in item.keywords and not item.config.getoption("--bench"):
37         pytest.skip("need --bench option to run")
38
39
40 def pytest_configure(config):
41     bench = config.getoption("--bench")
42     if bench not in (False, 'short', 'long', 'veryshort', 's', 'l', 'v'):
43         raise Exception("Invalid --bench option: " + bench)
44
45
46 def save_results():
47     import pickle
48     with open('timings.pickle', 'wb') as f:
49         pickle.dump(results, f)
50
51
52 def pytest_terminal_summary(terminalreporter):
```

Appendix C. STELLA Source Code

```
53     tr = terminalreporter
54     if not tr.config.getoption("--bench"):
55         return
56     lines = []
57     if results:
58         name_width = max(map(len, results.keys())) + 2
59         save_results()
60     else:
61         # TODO we were aborted, display a notice?
62         name_width = 2
63     for benchmark, type_times in sorted(results.items()):
64         type_width = max(map(len, type_times.keys())) + 2
65         for b_type, times in sorted(type_times.items()):
66             r = []
67             s = []
68             for impl, t in times.items():
69                 r.append('{:0.3f}s'.format(impl, t))
70                 if not impl.startswith('stella'):
71                     s.append('{:0.2f}x '.format('f'.rjust(len(impl)), t
72                                     ↪ /
73                                     times['stella']))
74             else:
75                 s.append(' ' * len(r[-1]))
76         lines.append("{} {} {}".format(benchmark.ljust(name_width),
77                                     b_type.ljust(type_width), '
78                                     ↪ '.join(r)))
79         lines.append("{} {} {}".format(' '.ljust(name_width),
80                                     ' '.ljust(type_width), '
81                                     ↪ '.join(s)))
82     if len(lines) > 0:
83         tr.write_line('-'*len(lines[0]), yellow=True)
84     for line in lines:
85         tr.write_line(line)
```

C.19 stella/test/si111s_globals.py

Appendix C. STELLA Source Code

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import time
15 from math import log, exp
16 from random import randint
17
18 import numpy as np
19
20 from . import * # noqa
21 import mtpy
22 import stella
23 from . import virtnet_utils
24
25 EXPSTART = 0.2
26
27
28 def prepare(args):
29     global K, rununtiltime, koffp, kcat, delta, leg, substrate, obs_i,
30         ↪ observations
31     params = Settings(args)
32
33     K = params['K']
34     rununtiltime = params['rununtiltime']
35     mtpy.mt_seed32new(params['seed'])
36     koffp = params['koffp']
37     kcat = params['r']
38
39     delta = (log(rununtiltime) - log(EXPSTART)) / float(K - 1)
40     leg = 0
41     substrate = 0
42     obs_i = 0
43     observations = np.zeros(shape=K, dtype=int)
```

Appendix C. STELLA Source Code

```
43
44     def get_results(r):
45         print (observations)
46         return observations
47
48     return (run, (), get_results)
49
50
51 def uniform():
52     return mtpy.mt_drand()
53
54
55 def mtpy_exp(p):
56     u = 1.0 - uniform()
57     return -log(u) / p
58
59
60 def makeObservation():
61     """Called from run()"""
62     global observations, leg, obs_i, next_obs_time
63     observations[obs_i] = leg
64     obs_i += 1
65
66     next_obs_time = getNextObsTime()
67
68
69 def getNextObsTime():
70     """Called from run()"""
71     global obs_i, EXPSTART, rununtiltime, delta
72     if obs_i == 0:
73         return EXPSTART
74     if obs_i == K - 1:
75         return rununtiltime
76
77     return exp(log(EXPSTART) + delta * obs_i)
78
79
80 def step():
81     """Called from run()"""
82     global leg, substrate
83     if leg == 0:
84         leg += 1
85     else:
```

Appendix C. STELLA Source Code

```
86         u1 = uniform()
87         if u1 < 0.5:
88             leg -= 1
89         else:
90             leg += 1
91     if leg == substrate:
92         substrate += 1
93
94
95 def isNextObservation():
96     global t, next_obs_time, obs_i, K
97     return t > next_obs_time and obs_i < K
98
99
100 def run():
101     global t, next_obs_time, obs_i, K, rununtiltime, leg, substrate
102     t = 0.0
103     next_obs_time = getNextObsTime()
104
105     # TODO: Declaring R here is not necessary in Python! But llum needs a
106     # it because otherwise the definition of R does not dominate the use
107     ↪ below.
108     R = 0.0
109     while obs_i < K and t < rununtiltime:
110         if leg < substrate:
111             R = koffp
112         else:
113             R = kcat
114         t += mtpy_exp(R)
115
116         while isNextObservation():
117             makeObservation()
118
119     step()
120
121 class BaseSettings(object):
122
123     def setDefaults(self):
124         self.settings = {
125             'seed': [int(time.time()), int],
126             'r': [0.1, float],
127             'koff': [1.0, float],
```

Appendix C. STELLA Source Code

```
128         'radius': [10, int],
129         'nlegs': [2, int],
130         'gait': [2, int],
131         'dim': [2, int],
132         'nspiders': [1, int],      # not completely functional yet
133         'elapsedTime': [self.elapsedTime, lambda x:x],
134     }
135
136     def elapsedTime(self):
137         return time.time() - self.start_time
138
139     def __init__(self, argv=[]):
140         self.start_time = time.time()
141
142         self.setDefaults()
143
144         # parse command line arguments to overwrite the defaults
145         for key, _, val in [s.partition('=') for s in argv]:
146             self[key] = val
147
148     def __setitem__(self, k, v):
149         if k in self.settings:
150             self.settings[k][0] = self.settings[k][1](v)
151         else:
152             self.settings[k] = [v, type(v)]
153
154     def __getitem__(self, k):
155         return self.settings[k][0]
156
157     def __str__(self):
158         r = '{'
159         for k, (v, type_) in self.settings.items():
160             if isinstance(type_, FunctionType):
161                 continue
162             r += str(k) + ':' + str(v) + ', '
163         return r[:-2] + '}'
164
165
166 class Settings(virtnet_utils.Settings):
167
168     def setDefaults(self):
169         self.settings = {
170             'seed': [int(time.time()), int],
```

Appendix C. STELLA Source Code

```
171         'r': [0.1, float],
172         'koffp': [1.0, float],
173         'K': [10, int],
174         'rununtiltime': [1e3, float],
175         'elapsedTime': [self.elapsedTime, lambda x:x],
176     }
177
178
179 def prototype(params):
180     prepare(params)
181     run()
182     py = np.array(observations) # save the global result variable
183
184     prepare(params)
185     stella.wrap(run)()
186     assert id(py) != id(observations)
187     st = observations
188
189     assert all(py == st)
190
191
192 @mark.parametrize('args', [['seed=42'], ['seed=63'], ['seed=123456'],
193                             ['rununtiltime=1e4', 'seed=494727'],
194                             ['seed={}'.format(randint(1, 100000))]])
195 def test1(args):
196     prototype(args)
197
198 timed = timeit(prototype, verbose=True)
199
200
201 def bench1():
202     timed(['seed=42', 'rununtiltime=1e8'])
```

C.20 stella/test/typing.py

```
1 #!/usr/bin/env python
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
```

Appendix C. STELLA Source Code

```
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 #     http://www.apache.org/licenses/LICENSE-2.0
9 #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 from random import randint
17 import numpy as np
18 from . import * # noqa
19
20
21 def return_bool():
22     return True
23
24
25 def return_arg(x):
26     return x
27
28
29 def numpy_return_element(a):
30     return a[2]
31
32
33 def equality(a, b):
34     return a == b
35
36
37 def cast_float(x):
38     return float(x)
39
40
41 def cast_int(x):
42     return int(x)
43
44
45 def cast_bool(x):
46     return bool(x)
47
```

Appendix C. STELLA Source Code

```
48
49 def test1():
50     make_eq_test(return_bool, ())
51
52
53 @mark.parametrize('arg', single_args([True, False, 0, 1, 42.0, -42.5]))
54 def test2(arg):
55     make_eq_test(return_arg, arg)
56
57
58 @mark.parametrize('args', [(True, True), (1, 1), (42.0, 42.0), (1, 2),
59     ↪ (2.0, -2.0), (True, False),
60     ↪ (randint(0, 10000000), randint(-10000,
61     ↪ 1000000))])
62
63 def test3(args):
64     make_eq_test(equality, args)
65
66
67 @mark.parametrize('args', [(False, 1), (False, 0), (True, 1), (42.0,
68     ↪ True), (1, 1.0),
69     ↪ (randint(0, 10000000), float(randint(-10000,
70     ↪ 1000000)))]])
71
72 @mark.xfail()
73 def test3fail(args):
74     make_eq_test(equality, args)
75
76
77
78
79 @mark.parametrize('args', single_args([np.zeros(5, dtype=int),
80     ↪ np.array([1, 2, 3, 4, 5],
81     ↪ dtype=int)]))
82
83 @mark.parametrize('f', [numpy_return_element])
84 @mark.xfail()
85 def test4fail(f, args):
86     make_eq_test(f, args)
87
88
89
90
91 @mark.parametrize('f', [cast_float, cast_int, cast_bool])
92 @mark.parametrize('args', single_args([True, False, 1, 42, -3, -5.5, 0, 2,
93     ↪ 3,
94     ↪ 4, 5, 3.14, randint(0, 10000000)]))
95
96 def test5(f, args):
97     make_eq_test(f, args)
98
99
100
```

Appendix C. STELLA Source Code

```
85
86 @mark.parametrize('f', [])
87 @mark.parametrize('args', single_args([1, 42, -3, -5.5, 0, 3.14,
    ↪ randint(0, 10000000)]))
88 @unimplemented
89 def test5u(f, args):
90     make_eq_test(f, args)
```

C.21 stella/test/__init__.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import numpy as np
15 from functools import wraps
16 import time
17
18 from stella import wrap
19 import pytest
20 from pytest import mark
21 from pytest import raises
22
23
24 delta = 1e-7
25
26
27 def single_args(l):
28     return list(map(lambda x: (x,), l))
29
```


Appendix C. STELLA Source Code

```
30
31 def make_eq_test(f, args):
32     args1 = []
33     args2 = []
34     for a in args:
35         if type(a) == np.ndarray:
36             args1.append(np.copy(a))
37             args2.append(np.copy(a))
38         else:
39             args1.append(a)
40             args2.append(a)
41     x = f(*args1)
42     y = wrap(f)(*args2)
43     assert x == y and type(x) == type(y)
44
45
46 def make_numpy_eq_test(f, args):
47     """
48     TODO stella right now won't return numpy types.
49     This test will treat them as equal to the python counterparts.
50     """
51     args1 = []
52     args2 = []
53     for a in args:
54         if type(a) == np.ndarray:
55             args1.append(np.copy(a))
56             args2.append(np.copy(a))
57         else:
58             args1.append(a)
59             args2.append(a)
60     x = f(*args1)
61     y = wrap(f)(*args2)
62
63     type_x = type(x)
64     for type_name in ('int', 'float'):
65         if type(x).__name__.startswith(type_name):
66             type_x = __builtins__[type_name]
67
68     assert x == y and type_x == type(y)
69
70
71 def make_eq_kw_test(f, args):
72     x = f(**args)
```

Appendix C. STELLA Source Code

```
73     y = wrap(f)(**args)
74     assert x == y and type(x) == type(y)
75
76
77 def make_delta_test(f, args, delta=delta):
78     x = f(*args)
79     y = wrap(f)(*args)
80     assert x - y < delta and type(x) == type(y)
81
82
83 def make_exc_test(f, args, py_exc, stella_exc):
84     with raises(py_exc):
85         x = f(*args) # noqa
86
87     with raises(stella_exc):
88         y = wrap(f)(*args) # noqa
89
90     assert True
91
92
93 unimplemented = mark.xfail(reason="Unimplemented", run=False)
94 bench = mark.bench
95
96
97 @pytest.fixture
98 def bench_opt(request):
99     opt = request.config.getoption("--bench")
100     if opt in ('l', 'long'):
101         return 2
102     elif opt in ('s', 'short'):
103         return 1
104     else:
105         return 0
106
107
108 @pytest.fixture
109 def bench_ext(request):
110     opt = request.config.getoption("--extended-bench")
111     return opt
112
113
114 def timeit(f, verbose=False):
115     @wraps(f)
```

Appendix C. STELLA Source Code

```
116     def wrapper(*args, **kw_args):
117         start = time.time()
118         r = f(*args, **kw_args)
119         end = time.time()
120         if verbose:
121             print("{0}({1}, {2}) took {3:0.2f}s".format(
122                 f.__name__, args, kw_args, end - start))
123         else:
124             print("{:0.2f}s".format(end - start))
125         return r
126     return wrapper
127
128
129 def time_stats(f, stats=None, **kwargs):
130     @wraps(f)
131     def wrapper(*args, **kw_args):
132         start = time.time()
133         r = f(*args, **kw_args)
134         end = time.time()
135         stats['elapsed'] = end - start
136         return r
137     return wrapper
138
139
140 @pytest.fixture
141 def report():
142     pass
```

C.22 stella/storage.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
```

Appendix C. STELLA Source Code

```
10 # distributed under the License is distributed on an "AS IS" BASIS,  
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
12 # See the License for the specific language governing permissions and  
13 # limitations under the License.  
14 from . import tp  
15 import llvmlite.ir as ll  
16  
17  
18 class Register(tp.Typable):  
19     name = None  
20  
21     def __init__(self, func, name=None):  
22         super().__init__()  
23         if name:  
24             assert type(name) == str  
25             self.name = name  
26         else:  
27             self.name = func.newRegisterName()  
28  
29     def __str__(self):  
30         return "{0}<{1}>".format(self.name, self.type)  
31  
32     def __repr__(self):  
33         return self.name  
34  
35  
36 class StackLoc(tp.Typable):  
37     name = None  
38  
39     def __init__(self, func, name):  
40         super().__init__()  
41         self.name = name  
42  
43     def __str__(self):  
44         return "%{0}<{1}>".format(self.name, self.type)  
45  
46     def __repr__(self):  
47         return self.name  
48  
49  
50 class GlobalVariable(tp.Typable):  
51     name = None  
52     initial_value = None
```

Appendix C. STELLA Source Code

```
53
54 def __init__(self, name, initial_value=None):
55     super().__init__()
56     self.name = name
57     if initial_value is not None:
58         self.setInitialValue(initial_value)
59
60 def setInitialValue(self, initial_value):
61     if isinstance(initial_value, tp.Typable):
62         self.initial_value = initial_value
63     else:
64         self.initial_value = tp.wrapValue(initial_value)
65     self.type = self.initial_value.type
66     self.type.makePointer(True)
67
68 def __str__(self):
69     return "+{0}<{1}>".format(self.name, self.type)
70
71 def __repr__(self):
72     return self.name
73
74 def translate(self, cge):
75     if self.llvm:
76         return self.llvm
77
78     self.llvm = ll.GlobalVariable(cge.module.llvm,
79     ↪ self.llvmType(cge.module), self.name)
80     # TODO: this condition is too complicated and likely means that my
81     # code is not working consistently with the attribute
82     llvm_init = None
83     if (hasattr(self.initial_value, 'llvm')
84         and self.initial_value is not None):
85         llvm_init = self.initial_value.translate(cge)
86
87     if llvm_init is None:
88         self.llvm.initializer =
89         ↪ ll.Constant(self.initial_value.type.llvmType(cge.module),
90                       ll.Undefined)
91
92     else:
93         self.llvm.initializer = llvm_init
94
95     return self.llvm
```

C.23 stella/exc.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import dis
15
16
17 class StellaException(Exception):
18     def __init__(self, msg, debuginfo=None):
19         super().__init__(msg)
20
21         self.addDebug(debuginfo)
22
23     def addDebug(self, debuginfo):
24         if debuginfo:
25             self.debuginfo = debuginfo
26
27     def __str__(self):
28         if hasattr(self, 'debuginfo'):
29             return '{0} at {1}'.format(super().__str__(), self.debuginfo)
30         else:
31             return super().__str__()
32
33
34 class UnsupportedOpcode(StellaException):
35     def __init__(self, op, debuginfo):
36         super().__init__(dis.opname[op])
37         self.addDebug(debuginfo)
38
39
```

Appendix C. STELLA Source Code

```
40 class UnsupportedTypeError(StellaException, TypeError):
41     def __init__(self, msg, debuginfo=None):
42         self.name_stack = []
43         self.type_stack = []
44         super().__init__(msg)
45
46         self.addDebug(debuginfo)
47
48     def prepend(self, name, type):
49         self.name_stack.append(name)
50         self.type_stack.append(type)
51
52     def __str__(self):
53         fields = ".".join(reversed(self.name_stack))
54         if fields:
55             fields += ': '
56         return fields + super().__str__()
57
58
59 class TypeError(StellaException, TypeError):
60     def __init__(self, msg, debuginfo=None):
61         super().__init__(msg)
62
63         self.addDebug(debuginfo)
64
65
66 class UnimplementedError(StellaException):
67     pass
68
69
70 class UndefinedError(StellaException):
71     pass
72
73
74 class UndefinedGlobalError(UndefinedError):
75     pass
76
77
78 class InternalError(StellaException):
79     pass
80
81
82 class WrongNumberOfArgsError(StellaException):
```

Appendix C. STELLA Source Code

```
83     pass
84
85
86 class AttributeError(StellaException, AttributeError):
87     def __init__(self, msg, debuginfo=None):
88         super().__init__(msg)
89
90         self.addDebug(debuginfo)
91
92
93 class IndexError(StellaException, IndexError):
94     def __init__(self, msg, debuginfo=None):
95         super().__init__(msg)
96
97         self.addDebug(debuginfo)
```

C.24 stella/utils.py

```
1 # Copyright 2013-2015 David Mohr
2 #
3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 import logging
15
16 # log level value for logging
17 VERBOSE = 25
18
19
20 class Stack(object):
21     backend = None
```


Appendix C. STELLA Source Code

```
22
23 def __init__(self, name="Stack", log=None, quiet=False):
24     self.backend = []
25     self.name = name
26     self.quiet = quiet
27     if log is None:
28         self.log = logging
29     else:
30         self.log = log
31
32 def __str__(self):
33     return "[" + self.name + "(" + str(len(self.backend)) + "]"
34
35 def __repr__(self):
36     return "[" + self.name + "=" + ", ".join([str(x) for x in
37         ↪ self.backend]) + "]"
38
39 def _log_debug(self, *args):
40     if not self.quiet:
41         self.log.debug(*args)
42
43 def push(self, item):
44     self._log_debug "[" + self.name + "] Pushing " + str(item)
45     self.backend.append(item)
46
47 def pop(self):
48     item = self.backend.pop()
49     self._log_debug "[" + self.name + "] Popping " + str(item)
50     return item
51
52 def __len__(self):
53     return len(self.backend)
54
55 def peek(self):
56     if len(self.backend) > 0:
57         return self.backend[-1]
58     else:
59         return None
60
61 def empty(self):
62     return len(self.backend) == 0
63
64 def clone(self):
```

Appendix C. STELLA Source Code

```
64     s = self.__class__(self.name, self.log, self.quiet)
65     s.backend = [x for x in self.backend]
66     return s
67
68     def contains(self, cond):
69         for item in self.backend:
70             if cond(item):
71                 return True
72         return False
73
74
75 class LinkedListIter(object):
76
77     def __init__(self, start):
78         self.next = start
79         self.stack = Stack('iter')
80
81     def __iter__(self):
82         return self
83
84     def __next__(self):
85         if self.next is None:
86             if not self.stack.empty():
87                 self.next = self.stack.pop()
88                 return self.__next__()
89             raise StopIteration()
90
91         if isinstance(self.next, Block):
92             self.stack.push(self.next.next)
93             self.next = self.next._block_start
94             return self.__next__()
95
96         current = self.next
97         self.next = self.next.next
98         return current
99
100
101 def linkedlist(klass):
102     klass.next = None
103     klass.prev = None
104     klass._block_parent = None
105
106     def __iter__(self):
```

Appendix C. STELLA Source Code

```
107     return LinkedListIter(self)
108     klass.__iter__ = __iter__
109
110     def printAll(self, log=None):
111         """Debugging: print all IRs in this list"""
112
113         if log is None:
114             log = logging
115
116         # find the first bytecode
117         bc_start = self
118         while True:
119             while bc_start.prev is not None:
120                 bc_start = bc_start.prev
121             if bc_start._block_parent is None:
122                 break
123             else:
124                 bc_start = bc_start._block_parent
125
126         for bc in bc_start:
127             # logging.debug(str(bc))
128             log.debug(bc.locStr())
129     klass.printAll = printAll
130
131     def insert_after(self, bc):
132         """Insert bc after self.
133
134         Note: block start and end are not adjusted here! They're only
135 ↪ checked at remove()"""
136         bc.next = self.next
137         if bc.next:
138             # TODO is this sufficient for the end of a block?
139             bc.next.prev = bc
140             self.next = bc
141             bc.prev = self
142     klass.insert_after = insert_after
143
144     def insert_before(self, bc):
145         """Insert bc before self.
146
147         Note: block start and end are not adjusted here! They're only
148 ↪ checked at remove()"""
149         bc.prev = self.prev
```

Appendix C. STELLA Source Code

```
148     bc.next = self
149
150     if not bc.prev and self._block_parent:
151         bc._block_parent = self._block_parent
152         self._block_parent = None
153         bc._block_parent._block_start = bc
154     else:
155         bc.prev.next = bc
156         self.prev = bc
157     klass.insert_before = insert_before
158
159     def remove(self):
160         if self.next:
161             self.next.prev = self.prev
162             if self.blockStart():
163                 # Move the block start attribute over to the next
164                 self.next.blockStart(self.blockStart())
165         if self.prev:
166             self.prev.next = self.next
167             if self.blockEnd():
168                 # Move the block end attribute over to the prev
169                 self.prev.blockEnd(self.blockEnd())
170     klass.remove = remove
171
172     def blockStart(self, new_parent=None):
173         """Get the block parent, or set a new block parent."""
174         if new_parent is None:
175             return self._block_parent
176
177         # Update the block's start
178         new_parent._block_start = self
179         # Remember the block
180         self._block_parent = new_parent
181     klass.blockStart = blockStart
182
183     def blockEnd(self, new_parent=None):
184         """Get the block parent, or set a new block parent."""
185         if new_parent is None:
186             return self._block_parent
187
188         # Update the block's end
189         new_parent._block_end = self
190         # Remember the block
```

Appendix C. STELLA Source Code

```
191     self._block_parent = new_parent
192     klass.blockEnd = blockEnd
193
194     def linearNext(self):
195         """Move to the next bytecode, transparently handling blocks"""
196         # TODO should this be its own iterator?
197         if self.next is None:
198             if self._block_parent:
199                 return self._block_parent.linearNext()
200             else:
201                 return None
202         if isinstance(self.next, Block):
203             return self.next.blockContent()
204         return self.next
205     klass.linearNext = linearNext
206
207     def linearPrev(self):
208         """Move to the previous bytecode, transparently handling blocks"""
209         # TODO should this be its own iterator?
210         if self.prev is None:
211             if self._block_parent:
212                 return self._block_parent.prev
213             else:
214                 return None
215         if isinstance(self.prev, Block):
216             return self.prev._block_end
217         return self.prev
218     klass.linearPrev = linearPrev
219
220     return klass
221
222
223 @linkedlist
224 class Block(object):
225
226     """A block is a nested list of bytecodes."""
227     _block_start = None
228     _block_end = None
229
230     def __init__(self, bc):
231         self._block_start = bc
232         bc._block_parent = self
233
```

Appendix C. STELLA Source Code

```
234     def blockContent(self):
235         return self._block_start
236
237
238 @linkedlist
239 class BlockStart(object):
240
241     """Marks the start of a block of nested bytecodes.
242
243     Enables checks via multiple inheritance."""
244     pass
245
246
247 class BlockEnd(object):
248
249     """Marks the end of a block of nested bytecodes.
250
251     Enables checks via multiple inheritance."""
252     pass
253
254
255 class BlockTerminal(object):
256
257     """
258     Marker class for instructions which terminate a block.
259     """
260     pass
```

C.25 stella/__init__.py

```
1 #!/usr/bin/env python
2 # Copyright 2013-2015 David Mohr
3 #
4 # Licensed under the Apache License, Version 2.0 (the "License");
5 # you may not use this file except in compliance with the License.
6 # You may obtain a copy of the License at
7 #
8 #     http://www.apache.org/licenses/LICENSE-2.0
9 #
```

Appendix C. STELLA Source Code

```
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15 import logging
16 import faulthandler
17
18 from . import analysis
19 from . import codegen
20 from . import utils
21
22 _f = open('faulthandler.err', 'w')
23 faulthandler.enable(_f)
24 logging.addLevelName(utils.VERBOSE, 'VERBOSE')
25
26
27 def logLevel(name='VERBOSE'):
28     if name == 'VERBOSE':
29         # custom log level
30         logging.getLogger().setLevel(utils.VERBOSE)
31     else:
32         try:
33             logging.getLogger().setLevel(getattr(logging, name))
34         except AttributeError:
35             raise AttributeError("Invalid log level {}".format(name))
36
37
38 def wrap(f, debug=False, p=False, ir=False, lazy=False, opt=None,
39         stats=None):
40     """
41     Parameters:
42         bool debug: increase the log level to DEBUG
43         bool p: print the LLVM IR instead of executing the program
44         mixed ir: return the LLVM IR if True, or save to file if a str.
45         bool lazy: construct the Stella representation and return the
46         ↪ object without
47         any action.
48         int opt: specify an optimization level for LLVM (usually 1-4)
49         dict stats: if a dict is passed in, then a detailed split of
50         ↪ execution
51         time will be stored in this parameter
```

Appendix C. STELLA Source Code

```
50     Unless lazy is specified, a callable will be returned which can be
↳   executed
51     in place of 'f'. Lazy returns the generated .codegen.Program object .
52     """
53
54     if debug:
55         logLevel('DEBUG')
56
57     def run(*args, **kwargs):
58         if stats is None:
59             pass_stats = {}
60         else:
61             pass_stats = stats
62
63         module = analysis.main(f, args, kwargs)
64         prog = codegen.Program(module)
65
66         prog.optimize(opt)
67
68         if lazy:
69             return prog
70         elif ir is True:
71             return prog.getLlvmIR()
72         elif type(ir) == str:
73             print("Writing LLVM IR to {}.{}".format(ir))
74             with open(ir, 'w') as fh:
75                 fh.write(prog.getLlvmIR())
76             return
77         elif p:
78             print(prog.getLlvmIR())
79         else:
80             return prog.run(pass_stats)
81     return run
82
83
84 def run_tests(args=None):
85     import pytest
86     import os.path
87     try:
88         from . import test
89     except SystemError:
90         from stella import test
91     if args is None:
```


Appendix C. STELLA Source Code

```
92     args = os.path.dirname(test.__file__)
93     pytest.main(args)
94
95     # for convenience register the Python intrinsics directly in the stella
96     # namespace TODO maybe this isn't the best idea? It may be confusing. On
97     ↪ the
98     # other hand, I don't plan to add more directly to the stella module.
99     # from .intrinsic.python import *
100     from ._version import get_versions
101     __version__ = get_versions()['version']
102     del get_versions
```

References

- [1] PyChecker. <http://pychecker.sourceforge.net/>.
- [2] pyflakes. <https://github.com/pyflakes/pyflakes>.
- [3] pylint. <http://www.pylint.org/>.
- [4] F. Akgul. *ZeroMQ*. Packt Publishing, 2013.
- [5] 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.
- [6] H. Ardo, B. Blais, P. Boddie, et al. shedskin, June 2013. URL <http://code.google.com/p/shedskin/>. Retrieved 2013-06-26.
- [7] 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.
- [8] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. 1411.1607, November 2014.
- [9] A. P. Black. Object-oriented programming: Some history, and challenges for the next fifty years. *Inf. Comput.*, 231:3–20, Oct. 2013. ISSN 0890-5401.

REFERENCES

- [10] R. Bordawekar, U. Bondhugula, and R. Rao. Believe It or Not!: Multi-core CPUs can match GPU performance for a FLOP-intensive application! In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 537–538, New York, NY, USA, 2010. ACM.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM.
- [15] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991. ISSN 0164-0925.

REFERENCES

- [17] 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.
- [18] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. G. Olthoff, editor, *ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer, 1995.
- [19] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.
- [20] 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.
- [21] J. Gibbons and N. Wu. Folding Domain-specific Languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM.
- [22] V. Grover, A. Kerr, and S. Lee. Plang: Ptx frontend for llvm. In *LLVM Developers' Meeting*, 2009.
- [23] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996. ISSN 1097-024X.

REFERENCES

- [24] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 318–326, New York, NY, USA, 1997. ACM.
- [25] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, et al. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, pages 1–34, 2014.
- [26] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for cpu-gpu architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM.
- [27] Jython. Jython, 2000. URL <http://www.jython.org/>. Retrieved 2015-06-27.
- [28] A. Kachayev. Functional programming with Python, 2012. URL <http://ua.pycon.org/static/talks/kachayev/>.
- [29] R. Kleckner. Unladen Swallow Retrospective. *QINSB is not a Software Blog*, March 2011. Retrieved 2013-06-04.
- [30] 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".
- [31] 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.
- [32] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, <http://comjnl.oxfordjournals.org/content/6/4/308.full.pdf+html>, 1964.

REFERENCES

- [33] 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.
- [34] 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.
- [35] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998. ISSN 1049-3301.
- [36] M. J. Olah, D. Mohr, and D. Stefanovic. Representing uniqueness constraints in object-relational mapping - the natural entity framework. In *TOOLS Europe*, 2012.
- [37] T. Oliphant, A. Valverde, D. Christensen, et al. Numba, 2012. URL <http://numba.pydata.org/>. Retrieved 2013-07-18.
- [38] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615.
- [39] R. Power and A. Rubinsteyn. How fast can we make interpreted Python? *ArXiv e-prints*, 1306.6047, June 2013.
- [40] T. Rentsch. Object oriented programming. *SIGPLAN Not.*, 17(9):51–57, Sept. 1982. ISSN 0362-1340.
- [41] 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.

REFERENCES

- [42] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6): 121–130, 2012.
- [43] SciPy. Weave, 2014. URL <http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html>. Retrieved 2015-06-10.
- [44] O. Semenov, D. Mohr, and D. Stefanovic. First-passage properties of molecular spiders. *Phys. Rev. E*, 88:012724, Jul 2013.
- [45] D. Stefanovic and M. Martonosi. On availability of bit-narrow operations in general-purpose applications. In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, pages 412–421, London, UK, UK, 2000. Springer-Verlag.
- [46] W. Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.
- [47] D. A. Terei and M. M. Chakravarty. An llvm backend for GHC. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM.
- [48] The HDF Group. Hierarchical Data Format, version 5, 1997-2015. <http://www.hdfgroup.org/HDF5/>.
- [49] M. J. Turk and B. D. Smith. High-Performance Astrophysical Simulations and Analysis with Python. *ArXiv e-prints*, 1112.4482, Dec. 2011.
- [50] Unladen Swallow. Unladen Swallow, 2010. URL <http://code.google.com/p/unladen-swallow/>. Retrieved 2015-06-09.
- [51] 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.

REFERENCES

- [52] Wikipedia. C++. URL <http://en.wikipedia.org/wiki/C%2B%2B>. Retrieved 2013-09-05.
- [53] C. Wimmer and T. Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*, pages 13–14, New York, NY, USA, 2012. ACM.
- [54] H. Xi and F. Pfennig. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257. ACM Press, 1998.