Towards Semantics Lifting for Scientific Computing: A Case Study on FFT

Naifeng Zhang
Carnegie Mellon University
Pittsburgh, USA
naifengz@cmu.edu

Sanil Rao
Carnegie Mellon University
Pittsburgh, USA
sanilr@andrew.cmu.edu

Mike Franusich
SpiralGen, Inc.
Pittsburgh, USA
mike.franusich@spiralgen.com

Franz Franchetti
Carnegie Mellon University
Pittsburgh, USA
franzf@andrew.cmu.edu

Abstract

The rise of automated code generation tools, such as large language models (LLMs), has introduced new challenges in ensuring the correctness and efficiency of scientific software, particularly in complex kernels, where numerical stability, domain-specific optimizations, and precise floating-point arithmetic are critical. We propose a stepwise semantics lifting approach using an extended SPIRAL framework with symbolic execution and theorem proving to statically derive high-level code semantics from LLM-generated kernels. This method establishes a structured path for verifying the source code's correctness via a step-by-step lifting procedure to high-level specification. We conducted preliminary tests on the feasibility of this approach by successfully lifting GPT-generated fast Fourier transform code to high-level specifications.

Keywords: Semantics lifting, static analysis, scientific computing, fast Fourier transform, SPIRAL

1 Introduction

The growing adoption of neural-based code generation tools, such as large language models (LLMs), presents significant challenges in ensuring the correctness and efficiency of scientific software [10]. Although LLM-generated code may be syntactically valid, it often falls short of meeting the rigorous correctness and performance standards required for complex scientific kernels. Scientific computing demands numerical stability, domain-specific optimizations, and accurate floating-point arithmetic, which are challenging to achieve in code generated without domain expertise. By deriving the semantics of generated kernels statically (at compile time) for cases with unknown (runtime) size parameters, we can identify potential bugs, inefficiencies, and performance bottlenecks before deploying the code. This statically derived information can also be fed back into neural-based code generation tools to iteratively improve the generated code. However, scientific computing poses unique challenges for static analysis tools. Accurate handling of floating-point arithmetic requires managing rounding errors and numerical precision, while pointers, recursion, and transcendental functions like sine and cosine further complicate static analysis. To address these issues, this work proposes *stepwise semantics lifting* as an early-stage experimental solution within constrained boundaries. We develop a novel extension to the SPIRAL system [6, 16], which is equipped with symbolic execution and theorem-proving capabilities. Through an LLVM-to-SPIRAL parser, we import LLM-generated scientific kernels into SPIRAL and derive their semantics using SPIRAL's formal framework and engine.

Contributions. To summarize, this paper makes the following contributions:

1. An experimental approach, stepwise semantics lifting, for statically extracting high-level semantics from scientific kernels.
2. An end-to-end demonstration of the proposed approach by lifting GPT-generated fast Fourier transform code to its high-level specification.

2 Background

In this section, we provide background on SPIRAL, a formal code generation system, and the target scientific kernel: the fast Fourier transform (FFT).

The SPIRAL system. The SPIRAL system [16] originated as an automatic performance-tuning system for signal processing algorithms, particularly focusing on FFT algorithms. This focus stemmed from the availability of a formal framework (the Kronecker product formalism [13, 19]), which enables capturing and manipulating FFT algorithms in high-level mathematical representations. Over time, this representation was generalized to encompass a broader range of algorithms [5], including both sparse and dense mathematical computations. SPIRAL has thus evolved into a comprehensive code generation system, capable of taking high-level specifications and producing optimized implementations for target platforms.

SPIRAL dialects. The SPIRAL system consists of three main components used in a stepwise code generation process: i) Signal Processing Language (SPL) [21], ii) Σ-SPL [9], and iii) internal code (icode) [6]. SPL, the top-level domain-specific language (DSL), describes the mathematical semantics of kernels and the functional data flow of the target algorithm. The lower-level DSL, Σ-SPL, captures loop abstractions, while icode serves as an abstract code representation adaptable to different code syntaxes. As shown in Figure 1, each layer of abstraction is connected by a rewrite system that applies recursive descent followed by confluent term rewriting. For further details, we refer readers to the respective citations.

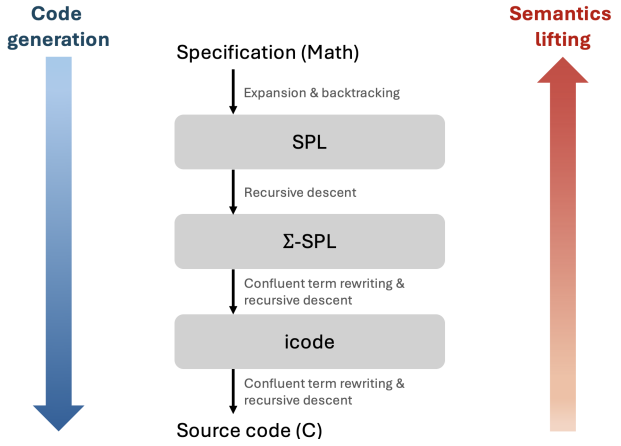

Figure 1. Overview of proposed semantics lifting procedure via SPIRAL. We propose to reverse the well-established code generation (i.e., lowering) process [6, 9, 15, 16, 21] to stepwisely lift the semantics of the source code.

Formal guarantees of SPIRAL. SPIRAL is built on top of the GAP computer algebra system [18], enabling localized correctness checks during rule application between abstraction layers. This allows verification that the left-hand and right-hand sides of each rewrite rule are equivalent at every step. Previous work, namely HELIX [22], has demonstrated that SPIRAL's algebraic guarantees can be extended to provide theorem prover-level assurances.

The FFT algorithm. The discrete Fourier transform (DFT) is a fundamental tool in science and engineering, playing a key role in areas such as signal processing, spectral analysis, communications, machine learning, and finance. The FFT is a class of efficient algorithms for computing the DFT. While a direct computation of the DFT for an n-element vector requires $O(n^2)$ operations, the FFT reduces this complexity to $O(n \log n)$.

FFT algorithms in SPL. In SPIRAL, linear transforms are represented as matrix-vector multiplications. For example, the DFT definition is viewed as a matrix-vector product:

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [\omega_n^{kl}]_{0 \le k,l < n}, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. There are several FFT algorithms for computing the DFT. Using SPL, we can define one of the most widely adopted FFT algorithms, the recursive Cookley-Tukey FFT algorithm, as

$$\mathrm{DFT}_n = (\mathrm{DFT}_m \otimes \mathrm{I}_k)\, \mathrm{T}^n_k (\mathrm{I}_m \otimes \mathrm{DFT}_k)\, \mathrm{L}^n_m, \ n = mk, \tag{2}$$

where I is the identity matrix, T is the twiddle matrix and L is the stride permutation matrix. L^{mk}_m permutes the elements of the input vector as $im + j \mapsto jk + i, 0 \le i < k, 0 \le j < m$ [8].

3 Stepwise Semantics Lifting

Our proposed approach explores the feasibility of lifting scientific kernels by reversing SPIRAL's rule-based code generation (i.e., lowering) process to transform source code into the highest achievable abstraction level, which is illustrated in Figure 1. Rather than employing linear rewriting as in the lowering process, lifting is structured as a search problem, where each step identifies the sequence of rule applications necessary to increase the abstraction level.

We demonstrate our approach through an end-to-end lifting of FFT code generated by GPT-4 [1], as presented in Listing 1. We have developed a custom parser that extracts the code's abstract syntax tree (AST) and converts it to SPIRAL's intermediate representation, icode. We omit the discussion of this parser due to space constraints.

```
1  #include <math.h>
2  #include <stdlib.h>
3  #define M_PI 3.14159265358979323846
4  void fft_recursive(double* data, int n) {
5      if (n <= 1) return;
6      // Allocate temporary storage for half-size FFTs.
7      double* even = (double*)malloc(n * sizeof(double));
8      double* odd = (double*)malloc(n * sizeof(double));
9      for (int i = 0; i < n / 2; ++i) {
10         even[2 * i] = data[4 * i];
11         even[2 * i + 1] = data[4 * i + 1];
12         odd[2 * i] = data[4 * i + 2];
13         odd[2 * i + 1] = data[4 * i + 3];
14     }
15     // Recursively compute FFTs.
16     fft_recursive(even, n / 2);
17     fft_recursive(odd, n / 2);
18     for (int i = 0; i < n / 2; ++i) {
19         double theta = -2.0 * M_PI * i / n;
20         double wr = cos(theta);
21         double wi = sin(theta);
22         // Twiddle factor multiplication.
23         double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
24         double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
25         data[2 * i] = even[2 * i] + real;
26         data[2 * i + 1] = even[2 * i + 1] + imag;
27         data[2 * (i + n / 2)] = even[2 * i] - real;
28         data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag;
29     }
30     // Cleanup.
31     free(even);
32     free(odd);
33 }
```

Listing 1. GPT-generated recursive FFT in C, which implements Equation 2.

Internal code to Σ-SPL. The provided example contains two loop bodies. We will now demonstrate how to lift lines 9-14 to Σ-SPL. To capture the access patterns in the code block, we utilize the gather and scatter formalism within SPIRAL [15]. In general, any read and write access can be formalized as an outer product of an $n \times 1$ and a $1 \times n$ standard basis vector:

$$e^{n\times 1}_{s(j)} \cdot e^{1\times n}_{g(j)},$$

where $e^{n\times 1}_k$ (resp. $e^{1\times n}_k$) is an $n \times 1$ (resp. $1 \times n$) vector with a one at the k^{th} position and zeros elsewhere. For the next step, we first need to conduct a range analysis on `data`, which is a pointer passed into the function without explicitly specified ranges. Now, line 10 turns into

$$e^{n\times 1}_{2j} \cdot e^{1\times 2n}_{4j}.$$

We apply a similar analysis to lines 11-13 to obtain B_j, and then we can write the entire loop (lines 9-14) as

$$\sum_{j=0}^{n/2-1} B_j, \quad B_j = \sum_{i=0}^{3} e^{n\times 1}_{2j+(i \bmod 2)} \cdot e^{1\times 2n}_{4j+i}. \tag{3}$$

Equation 3 can be directly captured by the iterative sum operator in Σ-SPL [9].

Σ-SPL to SPL. To lift Σ-SPL to SPL for the given example, we recognize the pattern of interleaved real and imaginary formats in this step, as B_j gathers from the same source and writes to two outputs twice, with an index offset of 1. The interleaved format of complex numbers is represented by the $\overline{(\cdot)}$ operator in SPIRAL [7] (which corresponds to `RC()` in Listing 2). Therefore, we can constrain the search problem to permutations of a complex vector. By definition, a stride-2 permutation reads with a stride of two and writes continuously, which aligns with the behavior observed in our target code block (lines 10-13). Consequently, the output from this stage is

$$\overline{\mathrm{L}^n_2}, \tag{4}$$

which, in SPL, represents a stride-2 permutation for a vector of n complex numbers.

Induction on SPL objects. After lifting all loop bodies to SPL expressions, we need to combine all lifted SPL components to form a complete SPL expression. In our example, we demonstrate how to lift lines 10-13 to $\overline{\mathrm{L}^n_2}$; similar principles apply to lifting lines 18-29, which results in $\overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\,\mathrm{T}^n_{n/2}}$. Therefore, we lift the code in Listing 1 to the representation in Listing 2.

```
1 void fft_recursive(double* data, int n) {
2     splfunc(concat_array(even, odd), data, i, RC(L(n, 2)));
3     fft_recursive(even, n / 2);
4     fft_recursive(odd, n / 2);
5     splfunc(data, concat_array(even, odd), i,
6         RC(Tensor(F(2), I(n/2)) * Diag(n, n/2)));
7 }
```

Listing 2. GPT-generated recursive FFT partially represented by SPL.

Since the FFT is defined recursively in this example, in SPL, we can write Listing 2 as

$$M_n = \overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\,\mathrm{T}^n_{n/2}}(\mathrm{I}_2 \otimes M_{n/2})\overline{\mathrm{L}^n_2}, \tag{5}$$

where M_n is an $n \times n$ matrix.

Now we can symbolically execute the entire program by setting $n = 2$. As FFT is a linear transform, we can derive the transformation matrix by combining the coefficients of each element in the result vector. By equivalence matching this matrix, we can verify that the generated C code implements a $\overline{\mathrm{DFT}_2}$ when $n = 2$. Given the base case, we can then induct on Equation 5 and derive that $M_n = \overline{\mathrm{DFT}_n}$. Therefore, we can consolidate all components into a single SPL expression:

$$\overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\,\mathrm{T}^n_{n/2}}\,\overline{(\mathrm{I}_2 \otimes \mathrm{DFT}_{n/2})\mathrm{L}^n_2}. \tag{6}$$

Note that the SPL expression, by definition, is evaluated from right to left. Thus, Equation 6 aligns precisely with the sequence of operations in Listing 1 and 2.

SPL to mathematical specification. This marks the final step of the lifting process. In this phase, we match the derived SPL expression with existing entries in SPIRAL's knowledge base through pattern matching. SPIRAL contains a wide range of linear transform algorithms, particularly within the FFT family [6]. In our example, the derived SPL (Equation 6) corresponds to Equation 2 when $m = 2$, $k = n/2$, and inputs are complex numbers. Hence, we can conclude that, with computer algebra system-level guarantees, the source code successfully implements the Cooley-Tukey FFT algorithm recursively. We write the final output as $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$.

Towards lifting multilinear operations. Our proposed approach can be extended to other scientific kernels, such as axpy [3], a multilinear operation that takes in two vectors of the same length, scales one vector by a constant α and adds it pointwise to the other vector. We can represent axpy for two vectors of length n in Σ-SPL as follows:

$$e_j^{n\times 1} \cdot [1, \alpha] \cdot \mathrm{I}_2 \otimes e_j^{n\times 1}. \tag{7}$$

We can then identify the scaling and reduction patterns within the axpy operation and lift Equation 7 to SPL as

$$[\mathrm{I}_n \mid \alpha\,\mathrm{I}_n]. \tag{8}$$

4 Related Work

The field of verified lifting [14, 17] utilizes SMT solvers to search for and verify program summaries (in the form of loop invariants and post-conditions) that correspond to the source code. However, existing efforts in this domain, which focus on stencil and tensor computations, fall short for scientific kernels due to limitations with floating-point numbers, pointers, and recursive functions [17]. Our approach shares the spirit of Stephen Wolfram's integration of Wolfram|Alpha with GPT [20], but leverages a more robust domain theory [5] suited to the complexity of floating-point numerical software, a domain typically hard to standard formal methods [2, 12]. Chelini et al. [4] similarly attempt to reverse the compilation process for scientific applications but restrict their approach to lifting matrix multiplications using the `Affine` dialect to `Linalg` dialect in MLIR.

5 Conclusion

In this work, we propose stepwise semantics lifting for scientific kernels and demonstrate a preliminary end-to-end lifting from LLM-generated C code to high-level semantics that recognizes the source code as a recursive FFT implementation. The problem of deriving the formal semantics of numerical code may indeed be unsolvable in general. However, for the class of algorithms that the SPIRAL system addresses—a well-defined subset of floating-point computational science and engineering algorithms—the problem becomes tractable. We aim to evaluate the proposed approach on a wider range of algorithms, focusing on numerical routines with data-independent control flows. Additionally, enhancing LLM code generation for scientific kernels by integrating SPIRAL-lifted information with human-guided prompts [11] could be a promising future direction.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1127353 and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-FOA-0002460. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the U.S. Department of Energy. Franz Franchetti was partially supported as the Kavčić-Moura Professor of Electrical and Computer Engineering.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Geoff Barrett. 1989. Formal methods applied to a floating-point number system. *IEEE transactions on software engineering* 15, 5 (1989), 611–621.

[3] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[4] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, and Nicolas Vasilache. 2020. MultiLevel Tactics: Lifting loops in MLIR. (2020).

[5] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 385–409.

[6] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.

[7] Franz Franchetti and Markus Puschel. 2002. A SIMD vectorizing compiler for digital signal processing algorithms. In *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 7–pp.

[8] Franz Franchetti and Markus Püschel. 2011. Fast fourier transform. *Encyclopedia of Parallel Computing. Springer* (2011), 51.

[9] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2005. Formal loop merging for signal transforms. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 315–326.

[10] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. 2023. Evaluation of OpenAI Codex for HPC parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*. 136–144.

[11] William F Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. 2024. Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience* (2024), e8269.

[12] John Harrison. 2006. Floating-point verification using theorem proving. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 211–242.

[13] Jeremy R Johnson, Robert W Johnson, Domingo Rodriguez, and Richard Tolimieri. 1990. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems and Signal Processing* 9 (1990), 449–500.

[14] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.

[15] Tze Meng Low and Franz Franchetti. 2017. High assurance code generation for cyber-physical systems. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 104–111.

[16] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[17] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A Seshia, and Alvin Cheung. 2024. Tenspiler: A Verified Lifting-Based Compiler for Tensor Operations. *arXiv preprint arXiv:2404.18249* (2024).

[18] The GAP Group. 2024. *GAP – Groups, Algorithms, and Programming, Version 4.13.1.* https://www.gap-system.org

[19] Charles Van Loan. 1992. *Computational frameworks for the fast Fourier transform*. SIAM.

[20] Stephen Wolfram. 2023. Chatgpt gets its' wolfram superpowers'. *Recuperado de https://writings. stephenwolfram. com/2023/03/chatgpt-gets-its-wolfram-superpowers* (2023).

[21] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. 2001. SPL: A language and compiler for DSP algorithms. *ACM SIGPLAN Notices* 36, 5 (2001), 298–308.

[22] Vadim Zaliva and Franz Franchetti. 2018. HELIX: a case study of a formal verification of high performance program generation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. 1–9.