Semantics Lifting for Scientific Kernels

Naifeng Zhang
Carnegie Mellon University
naifengz@cmu.edu

Franz Franchetti
Carnegie Mellon University
franzf@andrew.cmu.edu

Abstract

Modern compiler and code generation systems excel at lowering high-level specifications to optimized implementations. In contrast, the reverse direction, recovering high-level semantics from low-level code, remains relatively unexplored. This reverse direction, which can be viewed as *semantics lifting*, would enable verification and re-optimization of both generated and legacy code. Although lifting is undecidable in general, many scientific kernels exhibit strong mathematical structure that makes the problem tractable. We present an automatic semantics lifting framework built on SPIRAL, a mathematically grounded code generation system, for structured scientific kernels. By reversing SPIRAL's code generation passes, our approach lifts source code through successive abstraction layers to recover its mathematical semantics. We implement a fully automated lifting pipeline and evaluate it on GPT-generated fast Fourier transform kernels, successfully recovering the transform specification and detecting semantic errors.

1 Problem and Motivation

Asymmetry in code generation. Research in code generation exhibits a long-standing asymmetry. In scientific and high-performance computing, extensive effort has been devoted to translating high-level mathematical specifications into optimized low-level implementations. Over the past decades, domain-specific compilers and code generation systems such as TVM [3] and SPIRAL [5], and domain-specific languages (DSLs) like Halide [19] and TACO [11] have demonstrated how high-level specifications can be systematically lowered into highly optimized code. In contrast, the reverse direction has received far less attention: given a low-level implementation, recovering the high-level mathematical specification implemented by the code. A concrete example is determining that a piece of source code implements a fast Fourier transform (FFT). This problem can be viewed as *semantics lifting* [23]: deriving mathematical semantics from source code.

Opportunities. Solving this problem would enable two important capabilities. First, verification for correctness. Scientific kernels are increasingly produced not only by human developers but also by large language models (LLMs). Although such code may be syntactically valid, it can contain subtle semantic errors. Lifting implementations to high-level specifications provides a principled way to verify their semantic correctness before deployment in safety-critical environments. Second, re-optimization for performance. Many widely used scientific kernels exist as legacy hand-optimized implementations targeting older architectures, and their original mathematical specifications may no longer be available. Although modern compilers can apply local optimizations to such code, they generally do not recover the underlying mathematical semantics. Semantics lifting recovers this high-level specification to enable algorithmic re-synthesis across today's increasingly diverse computing landscape.

```c
#define M_PI 3.14159265358979323846
void fft_recursive(double* data, int n) {
    if (n <= 1) return;
    double* even = (double*)malloc(n * sizeof(double));
    double* odd = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n / 2; ++i) {
        even[2 * i] = data[4 * i]; even[2 * i + 1] = data[4 * i + 1];
        odd[2 * i] = data[4 * i + 2]; odd[2 * i + 1] = data[4 * i + 3]; }
    fft_recursive(even, n / 2); fft_recursive(odd, n / 2);
    for (int i = 0; i < n / 2; ++i) {
        double theta = -2.0 * M_PI * i / n;
        double wr = cos(theta); double wi = sin(theta);
        double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
        double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
        data[2 * i] = even[2 * i] + real;
        data[2 * i + 1] = even[2 * i + 1] + imag;
        data[2 * (i + n / 2)] = even[2 * i] - real;
        data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag; }
    free(even); free(odd); }
```

Listing 1: GPT-generated recursive FFT in C [23].

Challenges. Recovering semantics from arbitrary programs is fundamentally difficult. Rice's theorem [20] states that determining non-trivial semantic properties of arbitrary programs is undecidable in general. Existing lifting approaches [1, 10, 18] therefore focus on recovering structural summaries of programs, such as loop invariants. While useful, these techniques typically recover program structure rather than algorithmic semantics and often struggle with features common in scientific kernels, including floating-point arithmetic and transcendental functions, as shown in Listing 1.

Key insight. Many scientific kernels possess strong mathematical structure that makes the problem tractable. Kernels such as the FFT and convolution have well-defined mathematical specifications and a structured algorithmic space. Furthermore, domain-specific code generation systems like SPIRAL capture decades of domain knowledge as rewrite rules for generating highly optimized implementations. This observation leads to a key insight: if the rewrite rules that lower high-level specifications to implementations establish *equivalences across representations*, these rules can be reversed to lift implementations expressible within the same representational framework back to their algebraic form. Semantics lifting can therefore be formulated as a structured search within the code generator's algebraic framework rather than an open-ended attempt to infer arbitrary program semantics. In this work, we implement an automated semantics lifting framework based on SPIRAL, a mathematically grounded code generation system that provides the aforementioned properties. Our framework lifts low-level scientific kernels through multiple abstraction levels within SPIRAL, ultimately recovering their mathematical specifications. While still evolving, this work provides a practical realization of semantics lifting for structured scientific kernels.

2 Background and Related Work

The SPIRAL system. SPIRAL [5, 17] is a domain-specific code generation system that uses mathematical formalisms (particularly the Kronecker product formalism [9, 21]) to translate high-level specifications into highly optimized implementations across different hardware architectures. Over two decades of development, SPIRAL

has expanded from generating signal processing algorithms to supporting a broader class of applications including sparse and dense numerical computations and cryptographic kernels. SPIRAL models computations as compositions of structured operators such as tensor products, permutations, and diagonal matrices, an algebraic representation that makes it well suited for semantics lifting.

Abstraction layers in SPIRAL. SPIRAL organizes code generation through multiple intermediate representations (IRs) [7, 13, 22]. The highest level is the mathematical specification provided by the user. This specification is expressed in the signal processing language (SPL) [22], where it can be expanded within SPL into compositions of algebraic operators that capture algorithms. The next layer, Σ-SPL [7], represents loop-level structure and describes how SPL expressions are realized through iterative computation, enabling optimizations such as loop merging. The lowest layer is internal code (icode), an abstract representation close to actual implementations, from which SPIRAL can unparse to different programming languages. Transitions between these layers are performed through a large set of rewrite rules.

The FFT algorithm. The discrete Fourier transform (DFT) is a fundamental tool in science and engineering, widely used in areas such as signal processing, spectral analysis, and machine learning. As a structured yet non-trivial linear transform, it serves as an ideal example to demonstrate the capabilities of our semantics lifting framework. Efficient computation of the DFT is achieved through the FFT, a family of algorithms that reduce the time complexity of an n-point transform from $O(n^2)$ to $O(n \log n)$. In SPIRAL, DFT is viewed as a matrix-vector multiplication:

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

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. One of the most widely used FFT algorithms, the recursive Cooley-Tukey FFT algorithm [4], is expressed in SPL 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 [6].

Related work. Prior work has explored recovering higher-level representations of programs from lower-level implementations, particularly for structured numerical kernels such as stencils and tensor operations. Verified lifting techniques [1, 10, 18] use SMT solvers to synthesize program summaries in the form of loop invariants and post-conditions. Helium [15] adopts a dynamic, trace-driven approach to translate x86 binaries implementing stencil kernels into Halide [19]. Polygeist [16] raises C programs to MLIR dialects [12] by recovering loop structure using the polyhedral model. Multi-Level Tactics [2], an MLIR-based approach, uses pattern matching to recover higher-level abstractions, focusing primarily on lifting matrix multiplications. C2TACO [14] lifts dense tensor code to the TACO DSL [11] using automatically generated input-output examples to infer operator structure from source code. These systems generally recover structural or operator-level representations of programs rather than the *mathematical semantics* of the underlying algorithm. Moreover, many struggle with features common in scientific kernels such as floating-point arithmetic, transcendental functions, and recursion. In contrast, our work reverses an existing domain-specific code generation hierarchy to recover the underlying mathematical semantics implemented by complex scientific kernels. Recent work [23] outlined a conceptual roadmap for extending code generators like SPIRAL for semantics lifting, whereas this work presents a concrete system that realizes this vision and performs lifting automatically without human intervention.

3 Approach and Uniqueness

The key insight behind our semantics lifting framework is that SPIRAL's code generation is based on applying mathematically well-defined rewrite rules that preserve semantic equivalence. As these rules establish equivalence between representations, they can be applied in both directions, which allows semantics lifting to be formulated as the inverse of code generation.

To illustrate this idea, consider the rewrite rule shown in Equation 3, originally introduced by Franchetti et al. [7]. This rule rewrites a tensor product of matrix A with an identity matrix into an iterative gather (G)-compute-scatter (S) formulation, which can later be compiled into loop-based implementations. ι_n and $(j)_k$ represent index computations and their definitions are omitted due to space constraints.

$$A_n \otimes \mathrm{I}_k \;\; \rightleftarrows \;\; \sum_{j=0}^{k-1} \mathrm{S}_{\iota_n \otimes (j)_k}\, A_n\, \mathrm{G}_{\iota_n \otimes (j)_k} \tag{3}$$

During code generation, this rule is applied from left to right (→) to expand a high-level abstraction into a lower-level loop structure. For semantics lifting, as the rule preserves mathematical equivalence, we apply it in the reverse direction (←): by pattern matching the gather-compute-scatter structure with associated indices, we recover the corresponding tensor product representation. More generally, lifting becomes a structured search within SPIRAL's well-defined algebraic space, where low-level program constructs are progressively mapped to higher-level operators.

The rule above is one example among hundreds of rewrite rules in SPIRAL that preserve equivalences across different representations. By reversing these rules, implementations can be lifted back to their algebraic form, recovering the mathematical specification of kernels that fall within SPIRAL's representational framework. Building on this insight, we develop a semantics lifting framework that inverts SPIRAL's code generation pipeline to reconstruct the high-level specification implemented by source code.

Uniqueness. This work is unique in two key aspects. First, the methodology of reversing a mature domain-specific code generation hierarchy to recover high-level mathematical semantics from low-level implementations. Second, the demonstration of automatic lifting of GPT-generated complex scientific kernels to their high-level semantics, enabling verification and re-optimization of both generated and legacy code.

Although our work builds on SPIRAL, the methodology is not inherently tied to this system and could extend to other code generators that provide semantics-preserving rewrite systems. Since the current framework is tightly coupled with SPIRAL, it targets kernels whose implementations are expressible within SPIRAL's representational framework through the three IRs discussed in Section 2. This framework covers a broad class of scientific kernels including FFTs, basic linear algebra subprograms (BLAS) operations, and stencils. In the remainder of this section, we provide a high-level overview of how semantics lifting is automated across these abstraction layers using Listing 1 as an example. Detailed

definitions of the notations and operators used in the code snippets below can be found in prior works on SPIRAL [5, 17].

Parsing: source code to icode. The goal of this step is to eliminate programming-language-specific syntax and translate the program into icode, a unified IR used by SPIRAL. This step is implemented via a Clang-based parser. We first obtain the abstract syntax tree (AST) of the source program using Clang and then traverse the AST to construct the corresponding icode. For example, lines 15-16 in Listing 1 are translated into the icode snippet shown below.

```
1 loop(_i, _n/2, decl([_theta, _wr, _wi, _real, _imag], chain( ...
2     assign(nth(_d, V(2)*_i), nth(_even, V(2)*_i) + _real),
3     assign(nth(_d, V(2)*_i+V(1)), nth(_even, V(2)*_i+V(1)) + _imag), ... )))
```

Imposing a normal form: icode to Σ-SPL. The goal of this step is to identify the read-compute-write pattern of loop bodies and express it using the Σ-SPL representation. Translating icode to Σ-SPL requires several analyses, including range analysis and data layout analysis. We first impose a normal form on the loop body, assuming that each iteration follows a gather-compute-scatter structure. Under this assumption, the loop body can be represented as the composition of three matrices corresponding to the gather, compute, and scatter stages. Moreover, in the code snippet above, the indexing pattern consistently gathers elements at $2i$ and $2i + 1$ and writes the results back to $2i$ and $2i + 1$. This pattern indicates that the data is stored in an interleaved complex number format, where the real and imaginary components are adjacent in memory (denoted by `RC` in SPIRAL). Based on this indexing pattern and the gather-compute-scatter structure, we derive the following Σ-SPL from the icode snippet above:

```
1 RC(ISum(i, n/2,
2     Scat(fTensor(fId(2), fBase(n/2, i)))) * F(2) *
3     Gath(fTensor(fId(2), fBase(n/2, i))) * Diag(Tw1(n, n/2, -1)))
```

where `F(2)` denotes the DFT_2 matrix and `Tw1` represents the twiddle factors.

Recovering algebraic operators: Σ-SPL to SPL. As discussed at the beginning of Section 3, we reverse SPIRAL's code generation process by applying Rule 3 from right to left. Applying this rule to lines 1-3 of the Σ-SPL snippet above lifts the representation to `Tensor(F(2), I(n/2))`. Consequently, we obtain the following SPL representation:

```
1 RC(Tensor(F(2), I(n/2)) * Diag(Tw1(n, n/2, -1)))
```

This corresponds to the term $(\mathrm{DFT}_m \otimes \mathrm{I}_k)\, \mathrm{T}^n_k$ in Equation 2 when $m = 2$ and $k = n/2$.

Induction and lookup: SPL to specification. The final step combines the lifted SPL representations of all components extracted from Listing 1 (the walkthrough above lifts only one component of the FFT). The resulting SPL expression is:

```
1 RC(COND(n <= 1, I(1),
2     Tensor(F(2), I(n/2)) * Diag(Tw1(n, n/2, -1))) *
3     Tensor(I(2), fft_recursive(n/2)) * L(n, 2))
```

where `COND` denotes conditional evaluation. Although the SPL representation above exposes the program structure, it is not yet sufficient to conclude that the code implements an FFT. To derive the semantics, we must show that the representation corresponds to an FFT for any valid input size n. As the lifted SPL representation is recursive, we first reason about the base case by statically analyzing the smallest instance, $n = 1$, which results in the 1-point DFT. In this example, the base case is explicitly written as a conditional statement due to the recursive implementation. We then apply inductive reasoning within SPIRAL's algebraic framework together with its knowledge base to show that, for larger values of n, the recursive structure implements a DFT and matches the Cooley-Tukey FFT decomposition (Equation 2). The final output is shown in the next section.

4 Results and Contributions

Lifting LLM-generated FFT. We evaluate our lifting framework on a GPT-generated C implementation of the FFT, shown in Listing 1. Without any human intervention, the semantic lifting system automatically derives that the source code implements a DFT in the complex domain via the recursive Cooley-Tukey FFT algorithm:

```
$ spiral-lift fft.c
    ...
    Recovered specification: RC(DFT(n))
    Recovered algorithm: Recursive Cooley-Tukey FFT
```

To validate the result, we generate test inputs using FFTW [8] and statistically verify that the recovered specification matches the behavior of the source code. To the best of our knowledge, this demonstrates the first end-to-end automated recovery of high-level semantics for a non-trivial scientific kernel by reversing a domain-specific code generation system.

Bug detection. To evaluate the robustness of the system, we introduce small perturbations to the code, such as replacing `2*i` with `4/2*i` or swapping the order of two independent instructions. In both cases, the lifting process still succeeds, indicating that the system is robust to semantically equivalent variations and does not produce false positives in this setting. In contrast, modifying the access pattern from `2*i` to `4*i` causes the lifting to fail, as expected. Such indexing errors are common in LLM-generated code and can be difficult to diagnose due to the complexity of scientific kernels.

Input Variations	Lifting Output	Outcome
Original (Listing 1)	`RC(DFT(n))`	Verified
Reordered independent instructions	`RC(DFT(n))`	Verified
Semantically equivalent indexing	`RC(DFT(n))`	Verified
Incorrect index access	No SPL match	Bug detected

Limitations and future work. In the current prototype, although the lifting process is fully automated, the overall lifting strategy (e.g., selection and ordering of rewrite rules) is guided by manually designed heuristics. Automating the exploration of the lifting search space is part of our ongoing work. Moreover, we are currently extending the framework to support a broader range of FFT implementations as well as other scientific kernels such as BLAS operations and stencil computations, to further improve the robustness of the lifting framework. With a robust semantics lifting framework, our work enables two immediate future research directions. First, once a high-level specification is recovered, we can reuse the SPIRAL lowering pipeline to generate optimized implementations for different architectures or parse the lifted representation to utilize other code generators and DSLs, improving performance portability and extending the lifetime of legacy code. Second, the lifting trace provides structured feedback for LLMs by revealing which parts of generated code correspond to valid algorithmic constructs and which fail to lift. This provides symbolic feedback to neural code generators and suggests a pathway towards neuro-symbolic code generation.

Acknowledgments

The author would like to thank the shepherd and the anonymous reviewers for their constructive feedback. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0025645 and CW62109, and PRISM, a center in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by the Defense Advanced Research Projects Agency (DARPA). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the U.S. Department of Energy and DARPA.

References

[1] Sahil Bhatia, Sumer Kohli, Sanjit A Seshia, and Alvin Cheung. 2023. Building code transpilers for domain-specific languages using program synthesis (experience paper). In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 38–1.

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

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 578–594.

[4] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[5] 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.

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

[7] 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.

[8] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 3. Ieee, 1381–1384.

[9] 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.

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

[11] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[12] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[13] 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.

[14] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael FP O'Boyle. 2023. C2taco: Lifting tensor code to taco. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 42–56.

[15] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 391–402.

[16] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.

[17] 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.

[18] 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).

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

[20] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

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

[22] 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.

[23] Naifeng Zhang, Sanil Rao, Mike Franusich, and Franz Franchetti. 2025. Towards Semantics Lifting for Scientific Computing: A Case Study on FFT. *arXiv preprint arXiv:2501.09201* (2025).