# HELIX: A Case Study of a Formal Verification of High Performance Program Generation


Vadim Zaliva
Electrical and Computer Engineering
Carnegie Mellon University
Moffet Field, CA, USA
vzaliva@cmu.edu

Franz Franchetti
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA
franzf@ece.cmu.edu



## Abstract

In this paper, we present HELIX, a formally verified operator language and rewriting engine for generation of high-performance implementation for a variety of linear algebra algorithms. Based on the existing SPIRAL system, HELIX adds the rigor of formal verification of its correctness using Coq proof assistant. It formally defines two domain-specific languages: *HCOL*, which represents a computation data flow and Σ-*HCOL*, which extends *HCOL* with iterative computations. A framework for automatically proving *semantic preservation* of expression rewriting for both languages is presented. The *structural properties* of the dataflow graph which allow efficient compilation are formalized, and a monadic approach to tracking them and to reasoning about *structural correctness* of Σ-*HCOL* expressions is presented.



***CCS Concepts*** • **Theory of computation → Algebraic language theory**; **Rewrite systems**; **Program semantics**; *Logic and verification*;

***Keywords*** rule rewriting, operator language, Coq, formal verification

**ACM Reference Format:**
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 (FHPC '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3264738.3264739





*FHPC '18, September 29, 2018, St. Louis, MO, USA*





ACM ISBN 978-1-4503-5813-2/18/09…$15.00
https://doi.org/10.1145/3264738.3264739


## 1 Introduction

With the current level of sophistication of hardware architectures, the problem of manually implementing high-performance numerical algorithms becomes challenging even when using optimizing compilers and is often solved by specialized code generation systems, such as SPIRAL [Püschel et al. 2005]. SPIRAL can generate high-performance implementation for a variety of linear algebra algorithms, such as discrete Fourier transform, discrete cosine transform, convolutions, and the discrete wavelet transform, optimizing for features of target architecture, such as multiple cores, single-instruction multiple-data (SIMD) vector instruction sets, and deep memory hierarchies.

While SPIRAL is used to generate high-performance libraries for mission critical software, users need assurances about the correctness of the generated code. The goal of HELIX, as a part of the High Assurance SPIRAL project [Franchetti et al. 2017; Low and Franchetti 2017], is formal proof of the correctness of SPIRAL optimizations and code generation using Coq proof assistant.

SPIRAL works by transforming an original program though a series of intermediate languages, culminating in machine code, as shown in Figure 1. The original SPIRAL input language is called *OL* [Franchetti et al. 2009], and it closely resembles mathematical formulae. As a first step, it "breaks down" an *OL* expression into one or more *OL* operators, which, glued together by a function composition, represent a data-flow graph of the computation [Franchetti et al. 2005]. The resulting expression is then translated into another language, called Σ-*OL*, which adds the implicit representation of iterative computations. Next, using a series of rewrite rules driven by the extensive knowledge base of SPIRAL's optimization algorithms, the Σ-*OL* expression gets rewritten into a shape which lends itself to the efficient code for the target platform. Subsequently, an Σ-*OL* expression is compiled into an intermediate imperative language, called *i-Code*. By doing this, SPIRAL converts the dataflow graph into a sequence of loops and arithmetic operations. Finally, the *i-Code*, after some additional transformations, yields a C program, which is compiled with an optimizing compiler, producing an executable high-performance machine code implementation of the original *OL* expression.

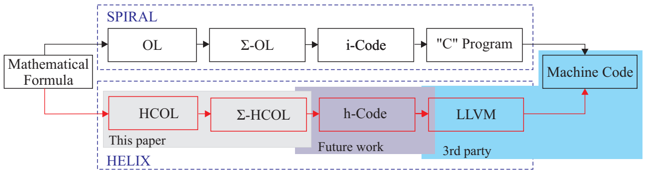



**Figure 1.** HELIX transformation stages

This paper describes HELIX, a system formalizing SPIRAL's *OL* and Σ-*OL* languages and providing correctness proofs of *OL* breakdown rules and Σ-*OL* rewriting rules. In the original SPIRAL system, both languages are loosely defined. For our work, we rigorously define two dialects of these languages, *HCOL* and Σ-*HCOL*.

We first resolve the question of the exact properties of the system we are formally proving. SPIRAL, being a DSL compiler, is expected to satisfy the *semantic preservation* property [Leroy 2009]. To perform platform specific optimizations, the *h-Code* generation step expects a Σ-*HCOL* expression in a certain shape adding the requirement of proving the *structural correctness* properties, which insure that shape. Proving correctness of such a system requires a novel approach, combining algebraic equational reasoning with compiler correctness proofs and proofs about computation dataflow structure.

## 2 Motivating Example

We first give an informal introduction of the *HCOL* language using a simple example to illustrate the main *HCOL* concepts to be more formally defined and elaborated in later sections. Sample implementation in Haskell is provided for illustration purposes in Appendix A.

As an example, we consider the *Chebyshev distance*, which is a metric defined on a vector space, induced by the infinity norm:

$$\begin{aligned} \mathrm{d}_\infty \colon\ & \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \quad \text{with} \\ \mathrm{d}_\infty(\vec{a}, \vec{b}) &= ||\vec{a} - \vec{b}||_\infty \end{aligned} \tag{1}$$

The *infinity norm* is a vector norm of a vector defined as:

$$\begin{aligned} ||\cdot||_\infty \colon\ & \mathbb{R}^n \to \mathbb{R} \quad \text{with} \\ ||\vec{x}||_\infty &= \max_i |\vec{x}_i| \end{aligned} \tag{2}$$

### 2.1 Chebyshev Distance in *HCOL*

*HCOL* operators are unary functions on real-valued finite-dimensional vectors. The scalar values are represented as single element vectors ($\mathbb{R} \cong \mathbb{R}^1$), and tuples of vectors are *flattened* ($\mathbb{R}^m \times \mathbb{R}^n \cong \mathbb{R}^{m+n}$). Thus, the *Chebyshev distance* and the *infinity norm HCOL* operators have the following types:

$$\texttt{ChebyshevDist} \colon\ \mathbb{R}^{2n} \to \mathbb{R}^1$$

$$\texttt{InfinityNorm} \colon\ \mathbb{R}^n \to \mathbb{R}^1$$

Three more *HCOL* operators correspond to common functional programming primitives: `fold`, `map`, and `zipWith` (see Appendix A for full definitions):

$$\texttt{Reduce}_{f,z} \colon\ \mathbb{R}^n \to \mathbb{R}^1 \tag{3}$$

$$\texttt{Map}_f \colon\ \mathbb{R}^n \to \mathbb{R}^n \tag{4}$$

$$\texttt{Binop}_f \colon\ \mathbb{R}^{2n} \to \mathbb{R}^n \tag{5}$$

*HCOL* operators can be combined using functional composition, for which we will use infix notation: $A \circ B$. Now, we can write an *HCOL* expression for the Chebyshev distance as a composition of an `InfinityNorm` operator and an elementwise vector subtraction, expressed as `Binop` parameterized by a binary subtraction function ($\mathrm{sub} : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$):

$$\texttt{ChebyshevDist} = \texttt{InfinityNorm} \circ \texttt{Binop}_{\mathrm{sub}} \tag{6}$$

In turn, an infinity norm can be broken down further into simpler operators resulting in the final *HCOL* expression for Chebyshev distance:

$$\texttt{ChebyshevDist} = \texttt{Reduce}_{\max,0} \circ \texttt{Map}_{\mathrm{abs}} \circ \texttt{Binop}_{\mathrm{sub}} \tag{7}$$

With this last step, we've transitioned from a high-level mathematical formula to an *HCOL* expression which operates on linear memory (vectors) and structurally represents the dataflow with granularity up to vectors.

### 2.2 Chebyshev Distance in Σ-*HCOL*

Most vector and matrix operations can be expressed as iterative computations on their elements. To generate efficient machine code for such computations, we transform our expressions into a form where these iterations will become explicit. For that, we extend the *HCOL* language in the following ways:

#### 2.2.1 Sparsity

An iterative computation on vectors can be viewed as superposition of computations performed during each step which processes only a subset of elements. The vector positions not used during an iteration step can be left empty. This can be presented naturally with sparse vectors. For example, a dense vector can be represented as the sum of columns of a diagonal sparse matrix, as shown in Figure 2. In this example, for simplicity, we use $\mathbb{R}^n$ type to represent sparse real-valued vectors of length $n$ and assume that sparse cells hold a special *structural zero* value, which is treated as regular 0 under addition. In later sections, we give a more formal treatment of how we represent and reason about sparsity in HELIX.

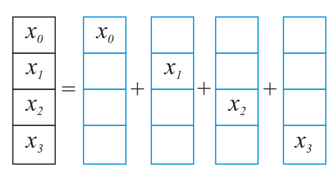


**Figure 2.** Dense vector as a sum of sparse vectors

It should be noted that these sparse representations are used only for verification of rewriting steps and do not affect generated code. All sparsity information will be "erased" along with proofs during compilation steps.

### 2.2.2 Data Partitioning and Re-assembly Operators

To switch between the dense and sparse representations of the data, we introduce additional operators which allow us to extract a subset of cells from a dense vector and embed them into a sparse one.

### 2.2.3 Higher-order Operators

To represent iterative computations over other operators, we need higher-order operators.

The *HCOL* language extended as described above is called Σ-*HCOL* and in the rest of this section, we present some of its operators.

***Lifting Scalar Functions*** We use notation $[\![\cdot]\!]$ for the *HCOL atomic* operator, which lifts real-valued scalar functions to *HCOL* operators. When lifting functions of multiple arguments, they are uncurried and their arguments are flattened into a vector. Thus, $\mathbb{R} \to \mathbb{R}$ is directly lifted to $\mathbb{R}^1 \to \mathbb{R}^1$, but $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$ becomes $\mathbb{R}^2 \to \mathbb{R}^1$.

***Embedding and Picking*** The `Embed` operator takes an element from a single-element vector and puts it at a specific index in a sparse vector of given length. The `Pick` operator does the opposite: it selects an element from the input vector at the given index and returns it as a single element vector:

$$\mathtt{Embed}_{n,i}: \ \mathbb{R}^1 \to \mathbb{R}^n \tag{8}$$

$$\mathtt{Pick}_i: \ \mathbb{R}^n \to \mathbb{R}^1 \tag{9}$$

***Gathering and Scattering*** Embedding and picking can be generalized where more than one element can be embedded or picked at once. The element selection is controlled by a user-provided *index mapping function*.

The `Scat` operator maps elements of the input vector to the elements of the output according to an index mapping function $f$. The mapping is *injective* but not necessarily *surjective*. That means the output vector could be sparse.

The `Gath` operator works in a similar manner except the index mapping function $f$ is used in the opposite direction: to map the output indices to the input ones.

$$\mathtt{Scat}_f: \ \mathbb{R}^n \to \mathbb{R}^m \tag{10}$$

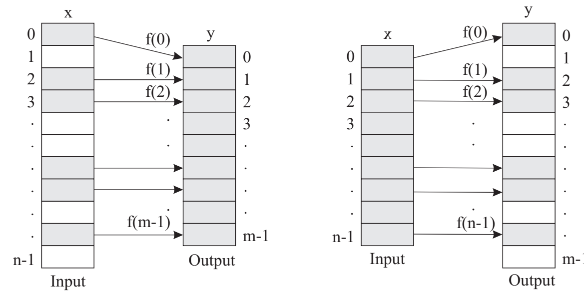
$$\mathtt{Gath}_f: \ \mathbb{R}^n \to \mathbb{R}^m \tag{11}$$

The `Gath` and `Scat` dataflows are shown in Figure 3.

### 2.2.4 Sparse Embedding

One class of *HCOL* expressions that we are particularly interested in has the following form:

$$\mathtt{Scat}_f \circ K \circ \mathtt{Gath}_g \tag{12}$$




**(a)** `Gath` **(b)** `Scat`

**Figure 3.** `Gath` and `Scat` dataflows

This form is called a *sparse embedding* of an operator $K$ (the *kernel*) and represents a step in iterative processing of a vector's elements. It corresponds to the body of a loop in which the *gather* picks the input vector's elements, which are then processed by $K$, and the results are then dispatched to appropriate positions in the output vector using the *scatter*. The function $f$ must be injective.

### 2.2.5 Map-Reduce

The higher-order *map-reduce* operator $\mathtt{MR}_{k,f,z}$ takes an indexed family of operators (a function which for each given index value returns an operator, typically a *sparse embedding*) and produces a new operator. It has the following type:

$$\mathtt{MR}_{k,f,z}: \ (\mathbb{N} \to (\mathbb{R}^n \to \mathbb{R}^m)) \to \mathbb{R}^n \to \mathbb{R}^m \tag{13}$$

When evaluated, a *map-reduce* applies all family members with indices between 0 and $k-1$ (inclusive) to an input vector, and the resulting $k$ vectors are folded element-wise using a binary function ($f : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$) and the initial value ($z : \mathbb{R}$).

A simple example applies a function $f$ to all elements of a vector of size 2:

$$\mathtt{MR}_{2,+,0}(\lambda i.(\mathtt{Scat}_{\lambda x.i} \circ [\![f]\!] \circ \mathtt{Gath}_{\lambda x.i})) \tag{14}$$

We use a family of *sparse embeddings* of $[\![f]\!]$ as a body of the *map-reduce*. The dataflow of expression (14) is shown in Figure 4.

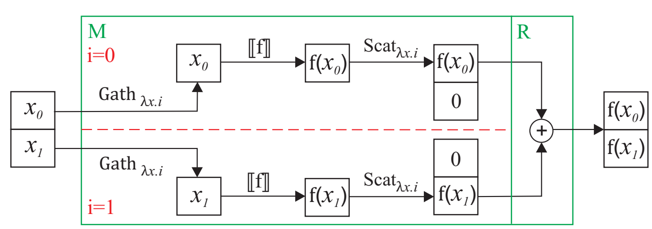



**Figure 4.** Map-Reduce of a Sparse Embedding of $f$

### 2.3 Chebyshev Distance Σ-*HCOL* Breakdown

We now demonstrate how *HCOL* expression (7) for Chebyshev Distance can be transformed via a series of rewriting steps into a Σ-*HCOL* form which exposes implicit iterations and is more suitable for compilation. During each step, one SPIRAL *rewriting rule* is applied, which replaces a part of the expression with another semantically equivalent expression.

$$\begin{aligned}
&\mathtt{Reduce}_{\max,0} \circ \mathtt{Map}_{\text{abs}} \circ \mathtt{Binop}_{\text{sub}} && (15)\\
&= \mathtt{Reduce}_{\max,0} \circ \mathtt{Map}_{\text{abs}} \circ \\
&\quad \mathtt{MR}_{n,+,0}(\lambda i.(\mathtt{Scat}_{\lambda x.i} \circ [\![\text{sub}]\!] \circ \mathtt{Gath}_{\lambda x.xn+i})) && (16)\\
&= \mathtt{Reduce}_{\max,0} \circ \mathtt{MR}_{n,+,0}(\lambda i. \\
&\quad (\mathtt{Scat}_{\lambda x.i} \circ [\![\text{abs}]\!] \circ [\![\text{sub}]\!] \circ \mathtt{Gath}_{\lambda x.xn+i})) && (17)\\
&= \mathtt{MR}\, n, \max, 0(\lambda i.(\mathtt{Reduce}_{\max,0} \circ \mathtt{Scat}_{\lambda x.i} \circ \\
&\quad [\![\text{abs}]\!] \circ [\![\text{sub}]\!] \circ \mathtt{Gath}_{\lambda x.xn+i})) && (18)\\
&= \mathtt{MR}_{n,\max,0}(\lambda i.([\![\text{abs}]\!] \circ [\![\text{sub}]\!] \circ \mathtt{Gath}_{\lambda x.xn+i})) && (19)\\
&= \mathtt{MR}_{n,\max,0}(\lambda i.([\![\lambda\, ab\ .\ |a-b|]\!] \circ \mathtt{Gath}_{\lambda x.xn+i})) && (20)\\
&= \mathtt{MR}_{n,\max,0}(\lambda i.([\![\lambda\, ab\ .\ |a-b|]\!] \circ \\
&\quad \mathtt{MR}_{2,+,0}(\lambda j.(\mathtt{Embed}_{2,j} \circ \mathtt{Pick}_{i+jn})))) && (21)
\end{aligned}$$

We start with expression (15), our final *HCOL* representation of Chebyshev distance, as in Equation (7). In (16), we expand the `Binop` operator onto an iterative map-reduce on sparse embedding of function sub. In (17), we combine `Map` and `MR` by moving `Map`'s operation inside `MR`'s sparse embedding kernel. Next, in (18), we move `Reduce` inside of the map-reduce iterator. In (19), we drop it altogether based on the fact that $\mathtt{Scat}_{\lambda x.i}$ will always produce a sparse vector with a single non-empty element. In (20), we merge two atomic operators into one. Finally, in (21), we expand `Gath` operating on a 2-element vector into a 2-step iterator processing each vector's element independently.

See Appendix A for the Haskell version of the complete example described in this section. The resulting expression (21) presents Chebyshev distance in terms of two nested iterative computations and some simple arithmetic operations. Each iterative map-reduce naturally translates to a loop, which allows compilation of this expression into an imperative program and subsequently into efficient machine code. For example, SPIRAL compiles expression (15) for $n = 3$ with optimizations turned off into the C code shown in Listing 1:

```
void chebyshev(float *y, float *x) {
    float s,t[2];
    y[0] = 0.0f;
    for(int i = 0; i <= 2; i++) { /* MR_{n,max,0} */
        for(int j = 0; j <= 1; j++) /* MR_{2,+,0} */
            t[j] = x[i + 3*j];
        s = abs(t[0] - t[1]); /* [[λ ab . |a − b|]] */
        y[0] = max(s, y[0]);
    }
}
```

**Listing 1.** SPIRAL-generated C Code for Chebyshev Distance

The C code above is generated for the most generic architecture and thus is not vectorized nor parallelized. However, from Σ-*HCOL* expressions, like (21), with implicit loops and the dataflow, SPIRAL can generate a very efficient machine code for various hardware architectures, taking advantage of vectorization and parallelization. See [Püschel et al. 2011] for details.

## 3 Defining *HCOL*

HELIX *HCOL* language is based on the SPIRAL *OL* language which was originally designed to represent linear algebra expressions on real or complex vectors. The primitive *OL* operators are functions from vectors to vectors. Higher-order operators, such as function composition, allow the building of more complex *HCOL* expressions.

Depending on the dimensionality of vectors an *OL* expression operates on, it could represent computation with different levels of granularity. By applying a set of rewriting rules, an *HCOL* expression could ultimately be "broken down" to the simplest form in which atomic operations are performed on scalar values, represented as single element vectors. An *HCOL* expression in such form can be directly mapped to a dataflow graph of the computation it represents.

*HCOL* is a shallow-embedded language in Coq proof assistant. All *HCOL* operators are represented as functions in [development team 2004] host language *Gallina*. Unlike *OL*, the data type is abstracted instead of using $\mathbb{R}$ or $\mathbb{C}$. In the rest of this section, we discuss the specifics of *HCOL* embedding in Coq.

The following are the data types used in *HCOL*:

The *Carrier Type*: This is an abstract representation of a numeric type, expressed in terms of its algebraic properties. Definitions and proofs formulated for it can be used, for example, on $\mathbb{R}$, $\mathbb{Q}$, or $\mathbb{Z}$, as they satisfy these properties. We denote the carrier type as $\mathcal{R}$. Algebraic properties are expressed using corresponding typeclass instances from the *MathClasses* library [Spitters and Van der Weegen 2011]. For example, we require that $\mathcal{R}$, along with corresponding operations, forms an algebraic ring, has total ordering, and has decideable *equality*.

Finite-dimensional Vectors: To represent vectors, we use the inductively-defined `Vector` type from Coq's standard library. Vector elements have type $\mathcal{R}$.

Finite natural numbers: To represent finite natural numbers, we use Coq's `sig` type. In this paper, we sometimes use the shorter notation $\mathbb{I}_n$ to denote $\{x : \mathbb{N} \mid x < n\}$.

The dimensions of input and output vectors of an *HCOL* operator are encoded as indices of the vector type family, and vector type $\mathcal{R}^n$ corresponds to `(vector R n)` in Coq. When constructing a complex *HCOL* expression, Coq's type system ensures that the input/output dimensions match.

# 4 Reasoning About *HCOL*

## 4.1 Semantic Preservation

When SPIRAL transforms *HCOL* expressions, as we demonstrated in Section 2, we want to ensure that the semantics are preserved. Our approach for proving semantic preservation of *HCOL* rewriting is described below and is based on Coq's *setoid rewriting* [Sozeau 2010] together with *HCOL* operator equational theory.

## 4.2 Equality

The definition of equality is essential for *HCOL* operator rewriting. The Coq default notion of equality (`eq`) is too restrictive for our purposes. For example, it doesn't allow us to work with rational numbers represented by non-reduced integer fractions. We would like to work on a carrier type equipped with an equivalence relation which is also called a *Setoid*.

Similarly, we define equality for vectors as a *pointwise relation*. From that follows the natural definition of the *HCOL* operator's *extensional equality*, which basically states that two operators $F$ and $G$ are equal if for all possible input vectors $x$, the values of $F\,x$ and $G\,x$ are also equal.

## 4.3 Rewriting

We define our semantics preservation property as an equivalence relation on *HCOL* expressions. To prove that *HCOL* expression $A$ could be transformed into *HCOL* expression $B$ while preserving its semantics, we need to prove $A = B$.

In the case of simple operators, we can just prove a lemma stating the equality of the two exact expressions. For complex expressions consisting of composition of multiple operators, such proof can be performed in a series of steps which can be automated. Each step corresponds to an application of a "rewriting rule" modifying a part of or a whole expression. For each rule, there is a lemma in the form $A = B$. It is applied using `setoid_rewrite` tactic, which looks in the current expression for patterns matching $A$ and replaces their occurrences with $B$. Because (=) is transitive, proving each rewriting step will guarantee the equality between the initial expression and the results of an application of a sequence of the rules. The rewrite rules in the HELIX library must be manually proven once but after that, these proofs can be reused to automatically prove the correctness of any sequence of their applications.

## 4.4 *HCOL* Semantics Preservation Verification Framework

To summarize, the components of our semantics preservation verification framework for *HCOL* rewriting were:

- We abstract the data type on which *HCOL* operates as carrier type $\mathcal{R}$.
- We assume an equivalence relation (=) on $\mathcal{R}$.
- We assume some algebraic properties of $\mathcal{R}$.
- We define (=) on vectors of $\mathcal{R}$ as a pointwise relation.
- We define *HCOL* operators as functions from vectors to vectors (of a carrier type) which are instances of `HOperator` typeclass.
- We define extensional equality of *HCOL* operators.
- We define rewriting rules as lemmas stating equality between *HCOL* expressions.

Using this framework, given the original and the final *HCOL* expressions, $h$ and $h'$, and the trace (a list) of rewriting rules applied to get from $h$ to $h'$, the HELIX *HCOL* rewriting proof engine can prove that an applied sequence of rewriting rules is *semantically preserving* and that $h = h'$.

This technique is known as a *translation validation*. A sequence of rewriting steps is generated outside of HELIX by the SPIRAL system. Instead of proving that SPIRAL will always transform an expression correctly, HELIX formally verifies the correctness of the produced results. Given that SPIRAL and HELIX use the same library of rewriting rules, the proof of a goal $h = h'$ is a sequence of applications of setoid rewrites using already proven per-rule lemmas from the HELIX library. We can automatically generate such proof from the trace and if Coq accepts it, the rewriting is proven correct. If, for some reason, the trace contains a non-semantically preserving rewriting sequence, Coq will not accept the proof.

# 5 Defining Σ-*HCOL*

HELIX Σ-*HCOL* language is based on the SPIRAL Σ-*OL* language. Like *HCOL*, Σ-*HCOL* is also embedded in Coq but with a *mixed embedding*, as discussed below. While *OL* and *HCOL* languages are declarative, Σ-*OL* and Σ-*HCOL* are purely functional.

*HCOL* operators can be used in Σ-*HCOL* expressions by wrapping them in a utility operator. Such "lifting" allows a temporary mixture of abstractions, corresponding to embedding mathematical formulae in a functional program. A Σ-*HCOL* expression can be gradually transferred to a purely functional form by applying a series of rewriting rules.

In iterative factorization of operations on vectors, each iteration represents a partial computation, and the resulting vector is sparse. In SPIRAL, the sparsity is implicit and represented by default values assigned to empty cells. In Σ-*HCOL*, we have implicit sparsity tracking and a special sparse vector type. Thus, while in SPIRAL, Σ-*OL* is a superset of *OL*, in HELIX, Σ-*HCOL* and *HCOL* are two distinct languages operating on different data types: sparse vs. dense vectors.

In the rest of this section, we discuss the specifics of Σ-*HCOL* formalization in Coq.

### 5.1 Sparse Vectors

As mentioned in Section 1, Σ-*HCOL* provides an implicit representation of iterative computations, such as applying a function to a vector's elements iteratively. A metaphor used is decomposition of dense vectors as a sum of sparse vectors (shown in Figure 2) combined with an `MR` operator (introduced in Section 2.2.5). Mathematically, the correctness of dense vector decomposition requires that the value held by the empty elements and the operation used to combine them form a monoid.

Decompositions differing in the number of vectors and locations of non-sparse values could represent a variety of memory access patterns. The particular class of decompositions of interest is the one where each is assigned to only one of the sparse vectors, as shown in Figure 2. In such a case, the *reduce* stage of the `MR` operator can be optimized out during code generation, and no actual summation needs to be performed. Thus, each non-sparse value generated represents a write to the output vector's element with the corresponding index. To detect violation of this form, it is sufficient to verify that, whenever we combine two cells during the *reduce* stage, at least one is empty. An attempt to combine two non-empty values, which we call *a collision*, signifies that the Σ-*HCOL* expression does not satisfy the collision-safe decomposition form we want.

Mathematically, as long as we assign a value of the monoid's identity element to the empty cells (for example, 0 in $\mathtt{MR}_{n,+,0}$), the summation decomposition will produce the correct results, but it will not allow us to distinguish empty cells from cells which happen to have 0 value and hence, we cannot detect collisions. Thus, we need to track the sparsity of vector elements separately from the actual values. We assume that sparse vector empty cells hold a nominal value, which we call a *structural value*.

Using this terminology, when combining vector elements pairwise, one of the values must be structural. If both values are non-structural, we have a *collision*, which indicates memory location being overwritten. Once occurred, the collision should be tracked down the computation graph, and any operation with a value produced as a result of a collision should be marked as colliding.

By this reasoning, we need our sparse vector formalization to meet the following requirements:

- Distinguish empty from assigned cells to detect collisions
- Customize the structural value of empty cells (e.g. 0 for addition, 1 for multiplication)
- Detect and track collisions
- Separate sparsity tracking from actual operations on values

We use a *Writer Monad* to track structural attributes. This monad provides a write-only state, which is tracked and updated along with computation. The state is accumulative, and its initial value `mzero` must form a monoid with the state update operation `mappend`. Our monad instance has an $\mathcal{R}$ as the value type and a set of flags stored in the `RthetaFlags` record as the state. We currently use just two boolean flags: one to represent whether the value is structural and another to track collisions which have already occurred.

Our default implementation of `mappend` combines the two sets of flags as follows: If at least one operands is non-structural, the result is also non-structural; combining two non-structural elements causes a collision; and the collision flag is sticky and propagates to the result as soon as one of the arguments has it set.

Writer Monad has an `mzero` value, which together with `mappend` forms a *monoid*. We define `mzero` as `RthetaFlags` with $is\_struct = true$ and $is\_collision = false$. We proved that the *monoid laws* are satisfied for `mappend` and `mzero`. Coq standard library do not have Monad type definitions, so in our development, we rely on the 3rd party library, *ExtLib* [Malecha et al. 2012].

Here we face a dilemma. In the *reduce* stage of `MR`, combining two non-structural elements is a collision. However, outside of `MR` as well as within its *map* stage, combining two dense values is a legitimate operation which should not trigger a collision. Even when collisions are not generated, we still need to track sparsity and preserve and propagate already raised collision flags. This can be achieved using a Writer Monad with the same value type, state type, and `mzero` value but equipped with a different `mappend` implementation. This operation is similar to default `mappend`, but it is collision-safe in that it never raises a new collision flag.

Now, we can define two new types for monadic $\mathcal{R}$ values: `Rtheta` and `RStheta`. They both share the same flags and track sparsity and collisions, but the former detects new collisions, while the latter does not. Technically these are implemented by instantiating Writer Monad for the same state and value types and the same `mzero` value but with two different `mappend` functions.

We can now use `Rtheta` or `RStheta` depending on whether we would like to permit collisions in the given context. We define conversion functions to switch between these two types, while preserving both the state and the value. Flag tracking is entirely separate from the operations on values since they represent two different aspects of computation. Our type system guarantees that flags and values are independent from each other. This allows separation of structural proofs from semantic preservation proofs.

### 5.2 Σ-*HCOL* Mixed Embedding

For Σ-*HCOL*, we use a different approach from *HCOL*, called *mixed embedding*, under which each operator is defined as a `Record` containing a shallow-embedded executable implementation in Gallina. Additionally, the record contains some other fields which are not required for execution but necessary for proving various properties of Σ-*HCOL* operators and expressions.

Two of such fields are `in_index_set` and `out_index_set` representing the operator's *sparsity contract*. They define the expected sparsity patterns of input vectors and guaranteed sparsity patterns of output vectors, respectively.

### 5.3 Σ-*HCOL* Operators

All Σ-*HCOL* operators are defined on sparse vectors, unlike *HCOL* operators which are defined on dense vectors. To work on sparse vectors and to track collisions, we defined Σ-*HCOL* variants of *HCOL* operators, such as `Binop`.

The composition of Σ-*HCOL* operators is defined as follows. The result of a composition of Σ-*HCOL* operators $f \circ g$ will be a new operator whose evaluation function will be a composition of the $f$ and $g$'s evaluation functions; the input dimensionality and `in_index_set` will be the same as in $g$; the output dimensionality and `out_index_set` will be the same as in $f$.

The final group of Σ-*HCOL* operators consists of higher-order operators representing iterative computations. An example of such an iterative operator is `MR` (*map-reduce*), which we introduced in Section 2.2.5. The `MR` works on sparse vectors but is parameterized by `Monoid RthetaFlags`, which allows this operator to be used with different collision-tracking policies.

## 6 Reasoning About Σ-*HCOL*

### 6.1 Semantic Preservation

The reasoning about semantic preservation during Σ-*HCOL* rewriting is similar in approach to our reasoning about *HCOL* in Section 4. The main difference is that Σ-*HCOL* operators work on sparse not dense vectors. However, for semantic preservation, we only care about equality of actual values ignoring the structural flags. This is achieved by defining custom equality relations for `Rtheta` and `RStheta` types. Comparing two monadic values of these types is defined as comparing their underlying values ignoring the state.

### 6.2 Structural Properties of Σ-*HCOL* Operators

We've defined the following structural properties which guarantee that a Σ-*HCOL* expression is in a form which is suitable for optimal and correct code generation:

1. The sparsity contract (`in_index_set` and `out_index_set` membership) is decideable
2. Only the values at indices from the `in_index_set` of the input vector affect output
3. A sufficiently filled input vector (values in correct places) guarantees a properly filled output vector (values only where expected)
4. Never generate values at sparse positions of the output vector
5. As long as there are no collisions at non-sparse locations of the input vector, none are produced at non-sparse locations in the output vector
6. Never generate collisions in sparse locations of the output vector

We've grouped them in a `SHOperator_Facts` type class and have proven its instances for all Σ-*HCOL* operators that we have defined. The proof of these properties for higher-order operators is *compositional*; as long as all operators involved are instances of `SHOperator_Facts`, it can be shown that all Σ-*HCOL* higher-order operators are also instances of `SHOperator_Facts`. That gives us a structural correctness proof "by construction" of any Σ-*HCOL* expression.

### 6.3 Additional Correctness Properties

In addition to semantic preservation and structural correctness, there are some additional properties which we want to verify for the final Σ-*HCOL* expression:

***Sparsity contract "subtyping"*** It guarantees that the resulting expression's `in_index_set` is included in the original expression's `in_index_set`, while the `out_index_set` of each expression is the same. This permits potential optimizations during rewriting, when indices of input vectors which were used by the original expression are no longer used by the resulting expression. This is also proven compositionally by constructing respective sparsity contracts of input and output expressions.

***Totality of the computation*** In general, Σ-*HCOL* operators work on sparse vectors. However, the sparsity is used only internally to represent partial computation. The whole composite computation should be total and take the dense input and produce the dense output. That means that for top-level Σ-*HCOL* expressions, we want to prove that both `in_index_set` and `out_index_set` are the full sets. This is proven compositionally as well, constructing respective sparsity contracts of input and output expressions.

## 7 Discussion

### 7.1 Status

Our goal is to formally verify the full pipeline shown in Figure 1 from mathematical expression to machine code. So far, we have formalized and proven *HCOL* and Σ-*HCOL* rewriting steps and translation between the two. The current development consists of about 5K lines of specification and 10K lines of proofs in Coq. Following the framework which we've developed, additional operators and rewriting rules could easily be added. We have not yet implemented the proof automation part, which would read the actual SPIRAL trace file and translate it to a sequence of tactic applications, but this step should be easy to implement, as there is 1-to-1 correspondence between rule applications in trace and tactic applications in automated proof. Automation of structural proofs could also be easily implemented in the future using LTAC proof automation for reasoning about finite sets.

### 7.2 Future Work

In future, we will prove the translation of Σ-*HCOL* to machine code via an intermediate language (*h-Code*). To do so we will use TemplateCoq [Anand et al. 2018] to reflect shallow embedded Σ-*HCOL* expressions to an intermediate deep-embedded form of Σ-*HCOL* language which then will be compiled to *h-Code*. We plan to develop a certified compiler from *h-Code* to LLVM IR. The sequence of translations is shown in Figure 1. We plan to prove correctness of Σ-*HCOL* to *h-Code* compilation and *h-Code* to LLVM IR. Proving correctness of IR to machine code compilation and low-level optimizations performed by the LLVM IR compiler are beyond the scope of this project and are being addressed by other parties, for example by the VELLVM project [Zhao et al. 2012, 2013] with whom we collaborate.

### 7.3 Related Work

This work falls under the category of *verified compilation*. The overall approach is discussed in [Leroy 2009], as implemented in CompCert compiler. A CertiCoq [Anand et al. 2017] approach is closer to ours as our language is shallow-embedded in Gallina. Another source of inspiration is CakeML [Kumar et al. 2014].

## A Chebyshev Distance Σ-*HCOL* expression in Haskell

```haskell
{-- Placeholder for structural zero --}
szero :: Float
szero = 0.0

{- Equation (3) -}
hReduce :: (a -> a -> a) -> a -> [a] -> [a]
hReduce f z = return . foldr f z

{- Equation (4) -}
hMap :: (a -> a) -> [a] -> [a]
hMap f = map f

{- Equation (5) -}
hBinop :: Int -> (a -> a -> a) -> [a] -> [a]
hBinop n f x = zipWith f a b
    where (a,b) = splitAt n x

{- Equation (8) -}
hEmbed :: Int -> Int -> [Float] -> [Float]
hEmbed n i (x:nil) = replicate i szero ++ [x] ++
                     (replicate (n-i-1) szero)

{- Equation (9) -}
hPick :: Int -> [a] -> [a]
hPick i x = [x !! i]

{- Converts a binary function to HCOL operator-}
hAtomic :: (a -> a -> a) -> [a] -> [a]
hAtomic f (a:b:nil) = [f a b]

{- Equation (10) -}
hScat :: Int -> (Int -> Int) -> [Float] -> [Float]
hScat m f x =
    let f' y d = if d == length x then Nothing
                 else if f d == y then Just d
                      else f' y (d+1)
    in map (\i -> case f' i 0 of
                    Nothing -> szero
                    Just j -> x!!j
           ) [0..m-1]

{- Equation (11) -}
hGath :: Int -> (Int -> Int) -> [a] -> [a]
hGath m f x = map (\i -> x!!(f i)) [0..m-1]

{- Equation (13) -}
hMR :: Int -> (a -> a -> a) -> a ->
       (Int -> ([a] -> [a])) -> [a] -> [a]
hMR k f z fam x = foldr (zipWith f) (repeat z)
                  (map (\i -> fam i x) [0..k-1])

{-- Equation (21) --}
chebyshev n = hMR n max 0.0 (\i ->
    hAtomic (\a b -> abs (a -b))
    .
    hMR 2 (+) 0.0
        (\ j -> (hEmbed 2 j) . (hPick (i+j*n))))
```

## Acknowledgments

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0291. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

## References

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*.

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP 2018-9th Conference on Interactive Theorem Proving*.

The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

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 (DSL WC) (Lecture Notes in Computer Science)*, Vol. 5658. Springer, 385–410.

F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Franusich, J. Johnson, A. Platzer, and M. M. Veloso. 2017. High-Assurance SPIRAL: End-to-End Guarantees for Robot and Car Control. *IEEE Control Systems* 37, 2 (April 2017), 82–103. https://doi.org/10.1109/MCS.2016.2643244

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 (PLDI '05)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/1065010.1065048

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Tze-Meng Low and Franz Franchetti. 2017. High Assurance Code Generation for Cyber-Physical Systems. In *IEEE International Symposium on High Assurance Systems Engineering (HASE)*.

Gregory Malecha et al. 2012. ExtLib Coq library. https://github.com/coq-ext-lib/coq-ext-lib. (2012). Accessed: 2018-03-12.

Markus Püschel, Franz Franchetti, and Yevgen Voronenko. 2011. *Spiral*. Springer US, Boston, MA, 1920–1933. https://doi.org/10.1007/978-0-387-09766-4_244

M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

Matthieu Sozeau. 2010. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning* (2010), 1–12.

Bas Spitters and Eelis Van der Weegen. 2011. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science* 21, 4 (2011), 795–825.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-Based Optimizations for LLVM. In *Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*.