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

Vadim Zaliva Franz Franchetti

Department of Electrical and Computer Engineering
Carnegie Mellon University

FHPC'18

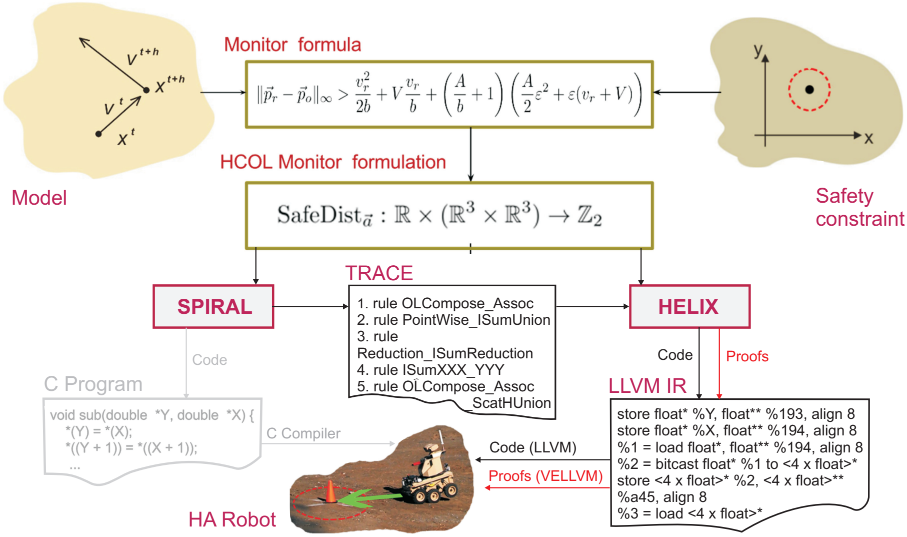
Outline

Introduction

Spiral and HELIX

- SPIRAL is a program generation system which 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 such features of target architecture as multiple cores, single-instruction multiple-data (SIMD) vector instruction sets, and deep memory hierarchies.
  It is developed by interdisciplinary team from CMU, ETH Zurich, Drexel, UIUC, and industry collaborators.
- HELIX is a CMU research project to bring the rigor of formal verification to SPIRAL.

Real-life Use-Case (Cyber-physical System)



$$\|\vec{p}_r - \vec{p}_o\|_\infty > \frac{v_r^2}{2b} + V\frac{v_r}{b} + \left(\frac{A}{b}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon(v_r+V)\right)$$

$$\mathrm{SafeDist}_{\vec{a}} : \mathbb{R}\times(\mathbb{R}^3\times\mathbb{R}^3)\to\mathbb{Z}_2$$

```
1. rule OLCompose_Assoc
2. rule PointWise_ISumUnion
3. rule
Reduction_ISumReduction
4. rule ISumXXX_YYY
5. rule OLCompose_Assoc
                    _ScatHUnion
```

```
void sub(double *Y, double *X) {
    *(Y) = *(X);
    *((Y + 1)) = *((X + 1));
    ...
```

```
store float* %Y, float** %193, align 8
store float* %X, float** %194, align 8
%1 = load float*, float** %194, align 8
%2 = bitcast float* %1 to <4 x float>*
store <4 x float>* %2, <4 x float>**
%a45, align 8
%3 = load <4 x float>*
```

Motivating Example

Motivating Example

Chebyshev distance

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

$$\mathrm{d}_\infty : \ \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \quad \text{with}$$
$$\mathrm{d}_\infty(\vec{a}, \vec{b}) = ||\vec{a} - \vec{b}||_\infty$$

Infinity norm

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

$$|| \cdot ||_\infty : \ \mathbb{R}^n \to \mathbb{R} \quad \text{with}$$
$$||\vec{x}||_\infty = \max_i |\vec{x}_i|$$

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* ($\mathbb{R}^m \times \mathbb{R}^n \cong \mathbb{R}^{m+n}$).

The *Chebyshev distance* and the *infinity norm HCOL* operators have the following types:

$$\texttt{ChebyshevDist}: \ \mathbb{R}^{2n} \rightarrow \mathbb{R}^1$$

$$\texttt{InfinityNorm}: \ \mathbb{R}^{n} \rightarrow \mathbb{R}^1$$

Some basic *HCOL* operators

Three more *HCOL* operators correspond to common functional programming primitives: `fold`, `map`, and `zipWith`:

$$\begin{aligned} \texttt{Reduce}_{f,z} &: \mathbb{R}^n \to \mathbb{R}^1 \\ \texttt{Map}_f &: \mathbb{R}^n \to \mathbb{R}^n \\ \texttt{Binop}_f &: \mathbb{R}^{2n} \to \mathbb{R}^n \end{aligned}$$

HCOL operators can be combined using functional composition, for which we will use infix notation: $A \circ B$.

Chebyshev distance breakdown in HCOL

We can write an *HCOL* expression for the Chebyshev distance as a composition of an `InfinityNorm` operator and an element-wise vector subtraction, expressed as `Binop` parameterized by a binary subtraction function ($\mathrm{sub} : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$):

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

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

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

From *HCOL* to 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:

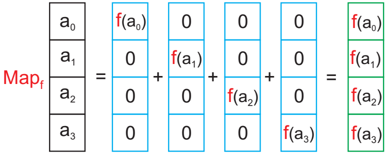
1. Iterative operators
2. Sparse vector data type

We will call such language Σ-*HCOL*.

In the next slides we will show simple example to demonstrate how sparsity and iterative operators interact.

Map as iterative sum

HCOL operator `Map` performs pointwise application of a function $f : \mathbb{R} \rightarrow \mathbb{R}$ to all elements of vector **a**. It could be represented as an iterative sum:



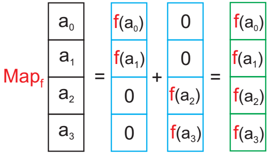
Which roughly corresponds to the following loop:

```
for(i=0;i<4;i++)
    f(src+i, dst+i);
```

Which requires 4 iterations.

Pointwise as a vectorized iterative sum

If we have a vectorized implementation of f with type $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ the sum will look like:



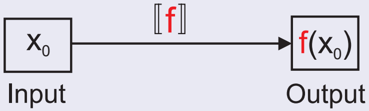
Which roughly corresponds to the following loop:

```
    for(i=0;i<2;i++)
        f(src+2*i, dst+2*i);
```

Which now requires only 2 iterations.

Lifting scalar functions

We use notation $[\![\cdot]\!]$ for the *HCOL atomic* operator, which lifts real-valued scalar functions to *HCOL* operators.

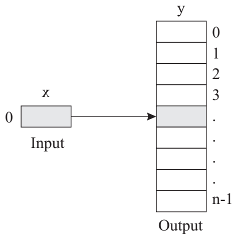


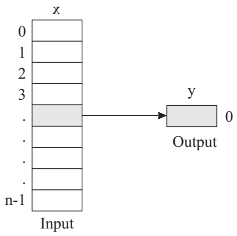
When lifting functions of multiple arguments, they are uncurried and their arguments are flattened into a vector. Thus, $f : \mathbb{R} \to \mathbb{R}$ is directly lifted to $[\![f]\!] : \mathbb{R}^1 \to \mathbb{R}^1$, but $g : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ becomes $[\![g]\!] : \mathbb{R}^2 \to \mathbb{R}^1$.

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:

$$\mathtt{Embed}_{n,i}: \ \mathbb{R}^1 \rightarrow \mathbb{R}^n$$
$$\mathtt{Pick}_{i}: \ \mathbb{R}^n \rightarrow \mathbb{R}^1$$



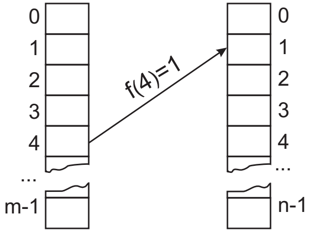


Index mapping functions

An index mapping function f has domain of natural numbers $\mathbb{N}$ in interval $[0, m)$ (denoted as $\mathbb{I}_m$) and the codomain of $\mathbb{N}$ in interval $[0, n)$ (denoted as $\mathbb{I}_n$):

$$f^{m \to n} : \mathbb{I}_m \to \mathbb{I}_n$$

Such function could be used to establish relation between indices of two vectors with respective sizes m and n.



Families of Index Mapping Functions

Function families

We define a *family* f of k index mapping functions as:

$$\forall j < k, \quad f_j^{m \to n} : \mathbb{I}_m \to \mathbb{I}_n$$

–jections

The family is called *injective* if it satisfies:

$$\forall n, \forall m, \forall i, \forall j, \quad f_n(i) = f_m(j) \implies (i = j) \land (n = m).$$

The family is called *surjective* if it satisfies:

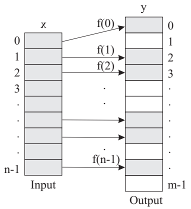
$$\forall j, \exists n, \exists i, f_n(i) = j.$$

The family is called *bijective* if it is both *injective* and *surjective*.

Generalizing `Embed` as `Scatter` operator

Given an *injective* index mapping function $f^{n \to m}$ the *scatter* operator $\texttt{Scat}_f : \mathbb{R}^n \to \mathbb{R}^m$ is defined as:

$$\mathbf{y} = \texttt{Scat}_f(\mathbf{x}) \iff \forall i < n,\ y_j = \begin{cases} x_i & \exists j < N,\ j = f(i), \\ \theta & otherwise. \end{cases}$$

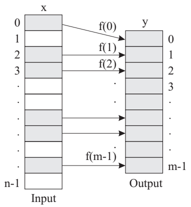


Function f must be *injective*. That ensures that every output vector element is assigned exactly once. Additionally, if f is *bijective* it is a *permutation*.

Generalizing `Pick` as `Gather` operator

Given an index mapping function $f^{m \to n}$ the *gather* operator $\texttt{Gath}_f : \mathbb{R}^n \to \mathbb{R}^m$ is defined as:

$$\mathbf{y} = \texttt{Gath}_f(\mathbf{x}) \iff \forall i < n,\ y_i = x_{f(i)}$$



If f is *injective* then every element of input vector will be sent to output vector at most once. Otherwise, some output vector elements can be repeated in the output vector.

Sparse Embedding

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

$$\texttt{Scat}_f \circ K \circ \texttt{Gath}_g$$

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*.

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:

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

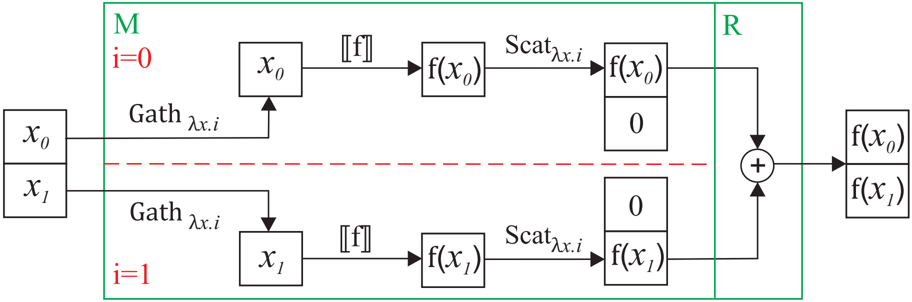
When evaluated, a *map-reduce* applies all family members with indices between 0 and $k-1$ (inclusive) to an input vector, and the resulting k vectors are folded element-wise using a binary function ($f : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$) and the initial value ($z : \mathbb{R}$).

Map-Reduce of Sparse Embedding

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

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

We use a family of *sparse embeddings* of $[\![f]\!]$ as a body of the *map-reduce*.



Chebyshev Σ-*HCOL* breakdown

Our *HCOL* expression 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.

$$
\begin{aligned}
&\texttt{Reduce}_{\max,0} \circ \texttt{Map}_{\text{abs}} \circ \texttt{Binop}_{\text{sub}} \\
&= \texttt{Reduce}_{\max,0} \circ \texttt{Map}_{\text{abs}} \circ \texttt{MR}_{n,+,0}(\lambda i.(\texttt{Scat}_{\lambda x.i} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i})) \\
&= \texttt{Reduce}_{\max,0} \circ \texttt{MR}_{n,+,0}(\lambda i.(\texttt{Map}_{\text{abs}} \circ \texttt{Scat}_{\lambda x.i} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i})) \\
&= \texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Reduce}_{\max,0} \circ \texttt{Map}_{\text{abs}} \circ \texttt{Scat}_{\lambda x.i} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i})) \\
&= \texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Reduce}_{\max,0} \circ \texttt{Scat}_{\lambda x.i} \circ \texttt{Map}_{\text{abs}} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i})) \\
&= \texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Map}_{\text{abs}} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i})) \\
&= \texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Binop}_{\lambda\, ab\, .\, |a-b|} \circ \texttt{Gath}_{\lambda x.xn+i})) \\
&= \texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Binop}_{\lambda\, ab\, .\, |a-b|} \circ (\texttt{MR}_{2,+,0}(\lambda j.(\texttt{Embed}_{2,j} \circ \texttt{Pick}_{i+jn})))))
\end{aligned}
$$

Breakdown Step 3

$$\texttt{Reduce}_{\max,0} \circ \texttt{MR}_{n,+,0}(\lambda i.(\texttt{Map}_{\text{abs}} \circ \texttt{Scat}_{\lambda x.i} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i})) =$$
$$\texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Reduce}_{\max,0} \circ \texttt{Map}_{\text{abs}} \circ \texttt{Scat}_{\lambda x.i} \circ \texttt{Binop}_{sub} \circ \texttt{Gath}_{\lambda x.xn+i}))$$

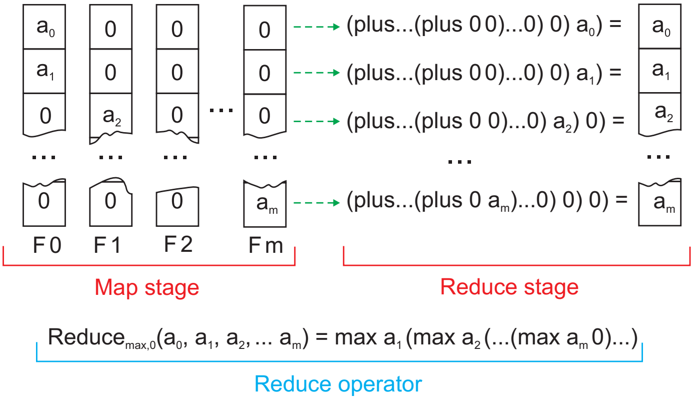
The corresponding Coq rewrite lemma

```
Theorem rewrite_Reduction_IReduction
  {i o n: ℕ}
  (op_family: @SHOperatorFamily Monoid_RthetaFlags i o n)
  `{uf_zero: MonUnit CarrierA} (* Common unit for both monoids *)
  `{f: SgOp CarrierA} (* 1st Monoid used in reduction *)
  `{P: SgPred CarrierA} (* the restriction *)
  `{f_mon: CommutativeRMonoid f uf_zero P} (* 2nd Monoid used in IUnion *)
  `{u: SgOp CarrierA}
  `{u_mon: CommutativeMonoid u uf_zero}
  (Uz: Apply_Family_Single_NonUnit_Per_Row _ op_family uf_zero)
  (Upoz: Apply_Family_Vforall_P _ (liftRthetaP P) op_family) :
      (liftM_HOperator Monoid_RthetaFlags (@HReduction _ f uf_zero))
        ∘ (@IUnion i o n u _ uf_zero op_family) =
      SafeCast (IReduction f uf_zero
            (UnSafeFamilyCast (SHOperatorFamilyCompose _
                (liftM_HOperator Monoid_RthetaFlags
                (@HReduction _ f uf_zero)) op_family))).
```

Step 3 – LHS

Let us consider left-hand side of the expression being rewritten:

$$\mathtt{Reduce}_{\max,0} \circ \mathtt{MR}_{n,+,0}(\lambda i.(F\ i)) = \mathtt{MR}_{n,\max,0}(\lambda i.(\mathtt{Reduce}_{\max,0} \circ (F\ i)))$$

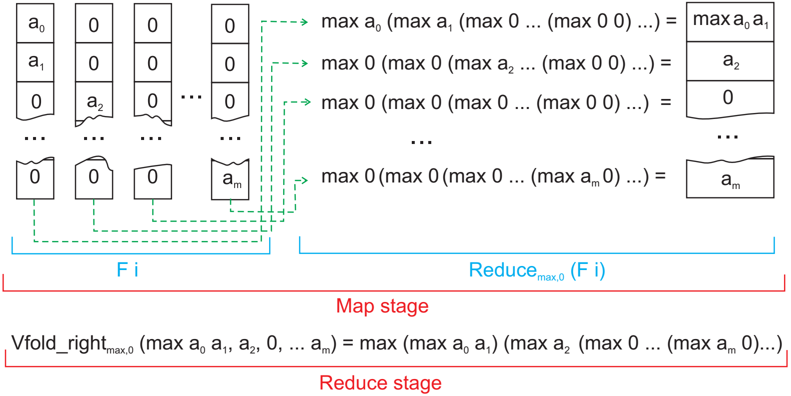


a_0 a_1 0 … 0
0 0 a_2 … 0
0 0 0 … 0
0 0 0 … a_m
F 0
F 1
F 2
F m
(plus...(plus 0 0)...0) 0) a_0) = a_0
(plus...(plus 0 0)...0) 0) a_1) = a_1
(plus...(plus 0 0)...0) a_2) 0) = a_2
(plus...(plus 0 a_m)...0) 0) 0) = a_m
Map stage
Reduce stage
$\mathrm{Reduce}_{\max,0}(a_0, a_1, a_2, \ldots a_m) = \max a_1 (\max a_2 (\ldots(\max a_m\ 0)\ldots)$
Reduce operator

Step 3 – RHS

Let us consider right-hand side of the expression being rewritten:

$$\texttt{Reduce}_{\max,0} \circ \texttt{MR}_{n,+,0}(\lambda i.(F\ i)) = \texttt{MR}_{n,\max,0}(\lambda i.(\texttt{Reduce}_{\max,0} \circ (F\ i)))$$



a_0
a_1
0
a_2
a_m
max a_0 (max a_1 (max 0 ... (max 0 0) ...) =
max 0 (max 0 (max a_2 ... (max 0 0) ...) =
max 0 (max 0 (max 0 ... (max 0 0) ...) =
max 0 (max 0 (max 0 ... (max a_m 0) ...) =
max a_0 a_1
F i
Reduce$_{\max,0}$ (F i)
Map stage
Vfold_right$_{\max,0}$ (max a_0 a_1, a_2, 0, ... a_m) = max (max a_0 a_1) (max a_2 (max 0 ... (max a_m 0)...)
Reduce stage

Step 3 – LHS RHS equality

The actual rewrite:

$$\mathtt{Reduce}_{\max,0} \circ \mathbb{MR}_{n,+,0}(\lambda i.(F\ i)) = \mathbb{MR}_{n,\max,0}(\lambda i.(\mathtt{Reduce}_{\max,0} \circ (F\ i)))$$

In our example could be reduced to equality:

$$\boxed{\begin{array}{l}\texttt{max}\ a_0\ \texttt{(max}\ a_1\ \texttt{(max}\ a_2\ \texttt{(... (max}\ a_m\ \texttt{0) ...) =}\\ \texttt{max (max}\ a_0\ a_1\texttt{) (max}\ a_2\ \texttt{(max 0 ... (max}\ a_m\ \texttt{0) ...).}\end{array}}$$

Which is correct as long as all a_i values are non-negative and max is commutative and associative.

Step 3 – observations

The actual rewrite:

$$\mathtt{Reduce}_{\max,0} \circ \mathtt{MR}_{n,+,0}(\lambda i.(F\ i)) = \mathtt{MR}_{n,\max,0}(\lambda i.(\mathtt{Reduce}_{\max,0} \circ (F\ i)))$$

Fold direction

`Reduce` is defined as a *right fold*, while the *reduce* step of `MR` is defined as a *left fold*:

```
Vfold_right f [a_1 ... a_n] b = f a_1 (f a_2 ... (f a_n b) ... ).
Vfold_left_rev f a [b_1 ... b_n] = f ... (f (f a b_n) b_{n-1}) ... b_1.
```

The rule will work only if $(\mathcal{T}, \max, 0)$ is a *commutative monoid*. It is not true for $\mathcal{T} = \mathbb{R}$ but it is true for $\mathcal{T} = \mathbb{R}^+$.

Sparsity

In the matrix produced by sequentially evaluating $F\ i$ for $i = 0 \ldots n-1$, each row has *at most* one non-zero element. For example, this is true if F is a `Scat` parametrized by *injective* familiy of index functions.

Step 3 – generalization

Generalized rewrite rule:

$$\texttt{Reduce}_{\mathrm{f},z} \circ \mathbb{MR}_{n,g,z}(\lambda i.(F\ i)) = \mathbb{MR}_{n,\mathrm{f},z}(\lambda i.(\texttt{Reduce}_{\mathrm{f},z} \circ (F\ i)))$$

Additional constraints:

1. In the matrix produced by sequentially evaluating $F\ i$ for $i = 0 \ldots n-1$, each row has *at most* one element not equal z.
2. In vectors produced evaluating $F\ i$ for any i all elements satisfy some predicate P.
3. $(\mathcal{T}, u, z)$ forms a commutative monoid.
4. $(\mathcal{T}, f, z, P)$ forms a *restricted* commutative monoid:
    - f is closed under P.
    - $(\mathcal{T}, u, z)$ is a commutative monoid for all $\mathcal{T}$ values which satisfy P.

Chebyshev Σ-*HCOL* code generation

The resulting expression presents Chebyshev distance in terms of two nested iterative computations and some simple arithmetic operations:

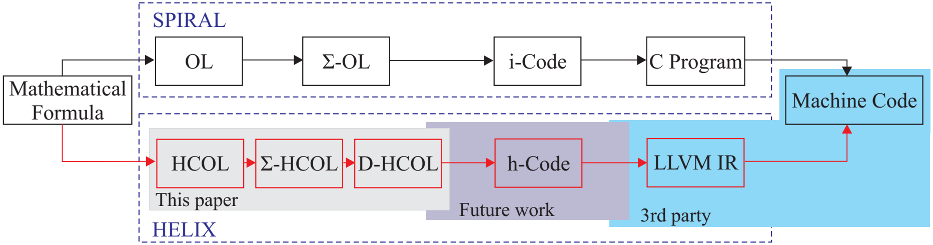
$$\mathtt{MR}_{n,\max,0}(\lambda i.(\mathtt{Binop}_{\lambda\, ab\,.\,|a-b|} \circ (\mathtt{MR}_{2,+,0}(\lambda j.(\mathtt{Embed}_{2,j} \circ \mathtt{Pick}_{i+jn})))))$$

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 the expression for $n = 3$ with optimizations turned off into the C code shown below:

```
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]); /* λab.|a − b| */
        y[0] = max(s, y[0]);
    }
  }
```

HELIX

From SPIRAL to HELIX



- *HCOL* formalization ✓
- *HCOL* rewriting correctness proofs ✓
- Σ-*HCOL* formalization ✓
- Σ-*HCOL* rewriting correctness proofs ✓
- *D-HCOL* formalization ✓
- Σ-*HCOL* to *D-HCOL* compiler correctness proofs ✓
- Σ-*HCOL* to *h-Code* verified compiler - *future work*
- *h-Code* to LLVM IR verified compiler - *future work*

HELIX languages summary

HELIX languages are embedded in Coq Proof assistant. The program is sequentially transformed from one language to another, and proof of all transformation stages guarantees semantic preservation.

$$\begin{tikzcd}\boxed{\text{HCOL}} \arrow[r] & \boxed{\Sigma\text{-HCOL}} \arrow[r] & \boxed{\text{D-HCOL}} \arrow[r] & \boxed{\text{h-Code}}\end{tikzcd}$$

	Abstraction	Embedding	Data	semantics	Proofs
HCOL	declarartive	shallow	dense vectors	equational	yes
Σ-HCOL	functional	shallow	sparse vectors	equational	yes
D-HCOL	functional	deep	dense vectors	equational	no (stripped)
h-Code	imperative	deep	memory arrays	operational	no

Sparsity

Why Sparsity Matters

Dense vectors are decomposed into iterative sums of sparse vectors. Multiple decompositions are possible. This allows applying a variety of algebraic transformations to reshape a computation to optimize for vectorization, parallelization, sequential memory access.

$$\mathrm{Map}_f \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ f(a_3) \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

$$\mathrm{Map}_f \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ f(a_3) \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

$$\mathrm{Map}_f \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ f(a_3) \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

$$\mathrm{Map}_f \xcancel{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ \frac{f(a_3)}{2} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{f(a_3)}{2} \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}}$$

Sparsity Constraints

$$\mathrm{Map}_f \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} f(a_0) \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ f(a_1) \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ f(a_2) \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ f(a_3) \end{pmatrix} = \begin{pmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{pmatrix}$$

- In such iterative sum, the addition has a special semantics:
  - Mathematically, the sparse values could be treated as zeroes.
  - Operationally, adding sparse and non-sparse values is an assignment.
- Certain constraints on structure of sparse vectors under iterative sums must be maintained.
- Tracking and enforcing such constraints in correctness proofs is difficult, as they are not adequately enforced by mathematical abstraction used.

Sparsity Requirements

We want our sparse vector formalization to meet the following requirements:

- Distinguish sparse and assigned cells
- Treat sparse cells as some “structural” value
- The “structural” value is not a constant (e.g. we may use 0 for addition but 1 for multiplication)
- In *sparse embedding* we should never combine two non-sparse elements. Such situation, if arise, we will call a *collision*
- Separate sparsity tracking from actual operations on values as they represent two different aspects of computation

Our Sparsity Approach

An overview of our sparse vector handing approach:

- Each value is tagged with two boolean flags: *is_struct* and *is_collision*
- The flags along with combining operator forms a *Monoid*.
- Depending on context one of the two monoid instances is used: with and without collision tracking
- Flags are tracked using *Writer Monad*
- Operations on values could not examine directly sparsity flags and thus could not depend on them
- Sparsity is automatically tracked by the monad. No implicit flags handling in operators' implementation
- Collisions are automatically detected and propagated by the monad

Monoid (Abstract algebra refresher)

Monoid

A *Monoid* $(\mathcal{A}, \oplus, \mathbf{0})$ is an algebraic structure which consists of:

- A Set $\mathcal{A}$
- A binary operation $\oplus : \mathcal{A} \to \mathcal{A} \to \mathcal{A}$ (AKA *mappend*).
- A special set element $\mathbf{0} \in \mathcal{A}$ (AKA *mzero*)

Monoid laws

A Monoid must satisfy the following *Monoid laws*:

- left identity: $\forall a \in \mathcal{A},\ \ \mathbf{0} \oplus a = a$
- right identity: $\forall a \in \mathcal{A},\ \ a \oplus \mathbf{0} = a$
- associativity: $\forall a, b, c \in \mathcal{A},\ \ (a \oplus b) \oplus c = a \oplus (b \oplus c)$

Flags Monoid

```
Record $\mathcal{R}_{\text{flags}}$ : Type := mk$\mathcal{R}_{\text{flags}}$ {is_struct: $\mathbb{B}$; is_collision: $\mathbb{B}$}.
```

```
Definition mzero := mkRthetaFlags $\top$ $\bot$.
Definition mappend (a b: $\mathcal{R}_{\text{flags}}$) : $\mathcal{R}_{\text{flags}}$ :=
    mkRthetaFlags
      (is_struct a && is_struct b)
      (is_collision a || is_collision b ||
                       (negb (is_struct a || is_struct b ))).
Definition Monoid_$\mathcal{R}_{\text{flags}}$ : Monoid $\mathcal{R}_{\text{flags}}$ := Build_Monoid mappend mzero.
```

The initial flags' value has structural flag *True* and collision flag *False*. The *mappend* operation combines the two sets of flags as follows. If one of operands is non-structural, the result is also non-structural. The collision flags are "sticky". Combining two non-structural elements, causes a collision. It could be proven that *monoid laws* are satisfied.

Monad in *Coq*

A simplified definition of *Monad* class from *Coq ExtLib* library:

```
Class Monad (m : Type → Type) : Type := {
    ret : ∀ {t : Type}, t → m t ;
    bind : ∀ {t u : Type}, m t → (t → m u) → m u
}.
```

m	is a *type constructor* that defines, for every underlying type, how to obtain a corresponding monadic type.
ret	is a *unit function* that injects a value in an underlying type to a value in the corresponding monadic type.
bind	is a *binding operation* used to link the operations in the pipeline.

WriterMonad

- One can think about *WriterMonad* as a product type $t \times s$ containing a *value* of type t and a *state* of type s. The state must be a *Monoid*.
- Monadic *ret* function constructs the new *WriterMonad* value by combining provided value with *mzero* state.
- Monadic *bind* operator allows to combine monadic values using user-provided functoin, and takes care of state tracking by combining states via *mappend*.
- In additon to *ret* and *bind* the following writer-specific functions are defined:

```
writer: ∀ s : Type, Monoid s → Type → Type
tell: ∀ (s: Type) (w: Monoid s), s → writer w ()
runWriter: ∀ (s t : Type) (w: Monoid s), writer w t → t×s
execWriter: ∀ (s t : Type) (w : Monoid s), writer w t → s
evalWriter: ∀ (s t : Type) (w : Monoid s), writer w t → t
```

Combining $\mathcal{R}_{\text{flags}}$ and *WriterMonad*

To track the flags while performing operations on $\mathbb{R}$ values we will use writer monad, parametrized by a monoid which defines how flags will be handled:

```
Definition Rθ := writer Monoid_Rflags ℝ.
```

To construct values of the type $\mathcal{R}_\theta$ we define two convenience functions:

```
Definition mkStruct (v:ℝ) : Rθ := ret v.
Definition mkValue (v:ℝ) : Rθ := tell (mkRflags ⊥ ⊥) ;; ret v.
```

Any unary or binary operation could be "lifted" to operate on monadic values using *liftM* or *liftM2* respectively:

```
liftM: (ℝ → ℝ) → (Rθ → Rθ)
liftM2: (ℝ → ℝ → ℝ) → (Rθ → Rθ → Rθ)
```

Sparse Operator Example

Now we can define a `Map` operator:

```
Definition Map (n: ℕ) (f: ℝ → ℝ) (v: vector ℛθ n): (vector ℛθ n)
    := vector.map (liftM f) v.
```

Key points:

- actual operation performing computations (f) is defined on $\mathbb{R}$
- all structural flags tracking is transparent
- a raw vector x could be passed as an argument by lifting it via *(vector.map mkValue x)*
- a vector of raw values could be extracted from the result x by simply applying *(vector.map evalWriter x)*.
- The resulting vector x could be checked for *collisions* using:

```
Definition vecNoCollision {n: ℕ} (v: vector ℛθ n) : Prop
    := vector.Forall (not ∘ is_collision ∘ execWriter) v
```

Iterative Operators

Iterative Operators – from dense to sparse

We have shown earlier that *Map* could be expressed as a summation:

$$\forall x \in \mathbb{R}^n, \quad \texttt{Map}_f\, x = \texttt{MR}_{n,+,0}(\lambda i.(\texttt{Scat}_{\lambda x.i} \circ [\![f]\!] \circ \texttt{Gath}_{\lambda x.i}))\, x$$

- This formulation is using dense vectors, without collision tracking. Now we would like to extend it to sparse vectors with collision tracking.
- It is using summation to combine elements. We would like to generalize it to other operations such as multiplication.

Scalar, Vector, and Operator Diamond

From arbitrary binary operation $\diamond : \mathcal{A} \to \mathcal{A} \to \mathcal{A}$ we can induce binary pointwise *vector diamond* operation:

$$\begin{aligned} &\vec{\diamond} : \mathcal{A}^n \to \mathcal{A}^n \to \mathcal{A}^n \\ &((a_0, a_1, \ldots, a_{n-1}), (b_0, b_1, \ldots, b_{n-1})) \mapsto \\ &(a_0 \diamond b_0, a_1 \diamond b_1, \ldots, a_{n-1} \diamond b_{n-1}) \end{aligned} \tag{2}$$

Next, we can define *operator diamond*:

$$\begin{aligned} &\mathring{\diamond} : (\mathcal{A}^n \to \mathcal{A}^m) \to (\mathcal{A}^n \to \mathcal{A}^m) \to (\mathcal{A}^n \to \mathcal{A}^m) \\ &(F, G) \mapsto (\mathbf{x} \mapsto F(\mathbf{x}) \, \vec{\diamond} \, G(\mathbf{x})) \end{aligned} \tag{3}$$

Iterative Diamond

Operator diamond in turn induces an *iterative diamond* operation for a family of n operators $F : \mathcal{A}^n \to \mathcal{A}^n$:

$$\mathop{\Diamond}_{i=0}^{n-1} F_i = F_1 \mathbin{\mathring{\diamond}} F_2 \mathbin{\mathring{\diamond}} \cdots \mathbin{\mathring{\diamond}} F_n$$

Or more formally, the recursive definition:

$$\begin{aligned} &\mathop{\Diamond}_{i=0}^{n-1} F_i : \mathcal{A}^n \to \mathcal{A}^n \\ &\mathbf{x} \mapsto \begin{cases} \mathbf{0}^n & \text{if } n = 0, \\ \left(F_{n-1} \mathbin{\mathring{\diamond}} \left(\mathop{\Diamond}_{j=0}^{n-2} F_j \right) \right) (\mathbf{x}) & \text{otherwise.} \end{cases} \end{aligned} \tag{4}$$

An additional requirement here is that the Set $\mathcal{A}$ forms a *Monoid* with identity element 0 of type $\mathcal{A}$ and binary associative operation $\diamond : \mathcal{A} \to \mathcal{A} \to \mathcal{A}$. The notation $\mathbf{0}^n$ denotes constant vector of identity elements of length n.

Iterative Sum with sparsity and collision tracking

Let us apply the *diamond* abstraction demonstrated in previous slides to $\mathcal{R}_\theta$ type (which represents $\mathbb{R}$ values with $\mathcal{R}_{\mathrm{flags}}$ state) and summation operator. To do so we specalize previous notation as follows:

Definition $\mathcal{A} := \mathcal{R}_\theta$.
Definition $\diamond :=$ `liftM2` $(+)$.
Definition $\mathbf{0}^n :=$ `vector.const (ret 0) n`.

This gives us a sparse, collision-tracking *Map*:

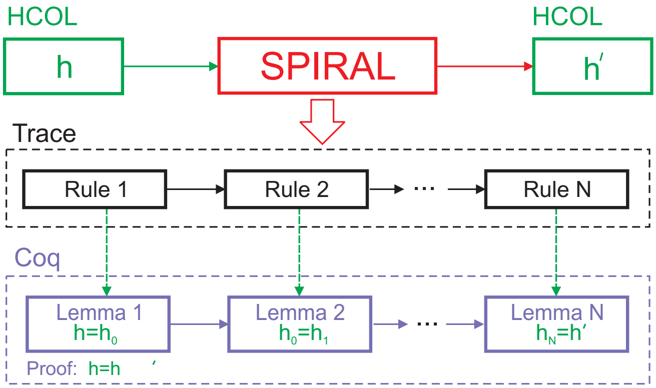
$$\mathtt{Map}_f \;= \mathop{\Diamond}_{j=0}^{n-1} \left(\mathtt{Scat}_{\lambda x.i} \circ [\![f]\!] \circ \mathtt{Gath}_{\lambda x.i}\right)$$

Verification

HELIX Verification Tasks

1. HELIX performs a series of program transformations (rewrites) in HCOL and Σ-*HCOL* languages. These transformations needs to be semantic-preserving.
2. The final Σ-*HCOL* expression must have certain structural properties.
3. Translation of Σ-*HCOL* to *D-HCOL* must preserve semantics.
4. Compilation of Σ-*HCOL* to *h-Code* must preserve semantics.
5. Compilation of *h-Code* to *LLVM IR* must preserve semantics.
6. *The correctness of the final compilation of LLVM IR to machine code will be guaranteed by a verified compiler such as VELLVM.*

Rewriting Rules Translation Validation



- "Translation Validation" vs full compiler verification approach.
- Each Σ-*HCOL* rewriting rule is a lemma.

Proving Rewriting Rules – Value Correctness

- Abstracting $\mathbb{R}$ as a *carrier type*
- Using *Setoid equality* relation defined on a carrier type wrapped in *WriterMonad*.
- The equality unwraps the *WriterMonad* and compare just values, ignoring the state (flags)
- Per-rule lemmas stating Extensional Setoid Equality of (compound) operators.
- Mixed embedding: Record containing operator function as well as additional properties such as *Setoid Morphism* (*Proper* instance)
- Value correctness reasoning in *transitional*.

Structural Properties

- Structural properties deal with sparsity flags and collision errors only. (They only examine the *state* of the *Writer Monad*).
- Operator *record* is extended to include finite sets of indices of non-sparse values of input and output vectors.
- The properties are expressed as a typeclass with following fields:
  1. Both *in_index_set* and out_index_set memberships are decideabe
  2. Only input elements with indices in *in_index_set* affect output
  3. Sufficiently (values in right places, no info on empty spaces) filled input vector guarantees properly (values are only where values expected) filled output vector
  4. Never generate values at sparse positions of output vector
  5. As long there are no collisions in expected non-sparse places, none is expected in nonsparce places on output
  6. Never generate collisions on sparse places
- Structural correctness reasoning in *compositional*.

Structural Properties typeclass

```
Class SHOperator_Facts {i o:ℕ} (f: @SHOperator i o) := {
  in_dec: FinNatSet_dec (in_indset f);
  out_dec: FinNatSet_dec (out_indset f);
  in_as_domain: ∀ x y,
    vec_equiv_at_set x y (in_indset f) → op f x = op f y;
  out_as_range: ∀ v,
    (∀  j (jc:j<i), in_indset f (mkFinNat jc) → Is_Val (Vnth v jc)) →
    (∀  j (jc:j<o), out_indset f (mkFinNat jc) → Is_Val (Vnth (op f v) jc));
  no_vals_at_sparse: ∀ v,
    (∀  j (jc:j<o), ¬ out_indset f (mkFinNat jc) → Is_Struct (Vnth (op f v) jc));
  no_coll_range: ∀ v,
    (∀  j (jc:j<i), in_indset f (mkFinNat jc) → Not_Collision (Vnth v jc)) →
    (∀  j (jc:j<o), out_indset f (mkFinNat jc) →
      Not_Collision (Vnth (op f v) jc));
  no_coll_at_sparse: ∀ v,
    (∀  j (jc:j<o), ¬ out_indset f (mkFinNat jc) →
      Not_Collision (Vnth (op f v) jc));
}.
```

D-HCOL language

After applying HELIX rewriting rules, Σ-*HCOL* expressions are compiled to lower-level language, called *D-HCOL*. This language differs from Σ-*HCOL* in a number of ways:

- There is one-to-one correspondance between Σ-*HCOL* and *D-HCOL* operators.
- *D-HCOL* contains a limited subset of operators compared to Σ-*HCOL*.
- *D-HCOL* is deep embedded in Coq, unlike Σ-*HCOL* which is shallow embedded.
- No sparsity tracking.
- No proofs as part of definitions.
- Using de Bruijn indices for variables.
- Evaluation function is defined which takes *D-HCOL* expression and environment Γ and evaluates it in this environment.
- A limited fixed set of *intirinsic* functions (e.g. $+, -, max$) is defined.

From Σ-*HCOL* to *D-HCOL*

- Compiler pass from Σ-*HCOL* to *D-HCOL* is implemented using *Template-Coq*
- Template program *reifySHCOL* takes an Σ-*HCOL* expression and produces two artefacts (or an error):
  1. A corresponding *D-HCOL* expression.
  2. A theorem, stating semantic equivalence of produced *D-HCOL* expression and the original Σ-*HCOL* expression.
- The semantic equivalence theorem is automatically proven by applying a sequence of semantic preservation lemmas (one per operator). This is possible because the expressions are structurally similar and there one-to-one correspondence between operators.
- An error occurs when Σ-*HCOL* expression contains operators which are not part of *D-HCOL*. Normally, rewriting rules ensure that never happens.

Summary

Summary

What have been done so far:

1. Formalization of *HCOL*, Σ-*HCOL*, and *D-HCOL* languages.
2. *HCOL* and Σ-*HCOL* rewriting proofs.
3. Sparse vectors, sparsity tracking.
4. Formalized operator structural properties
5. Proofs of structural properties of Σ-*HCOL* expressions.
6. Verified Σ-*HCOL* to *D-HCOL* compiler.

Next steps:

1. Σ-*HCOL* rewriting proof automation using SPIRAL trace
2. Formalzing *h-Code* (including operational semantics)
3. Linking *D-HCOL* semantics to *h-Code* semantics
4. Linking *h-Code* semantics to *LLVM IR* semantics
5. Dealing with *floating point* arithmetic.

For Further Reading I

Yves Bertot and Pierre Castéran.
Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions.
Springer, 2013.

Franz Franchetti, Yevgen Voronenko, and Markus Püschel.
Formal loop merging for signal transforms
PLDI, 2005

Franz Franchetti, Tze Meng Low, Stefan Mitsch, Juan Pablo Mendoza, Liangyan Gui, Liangyan, Amarin Phaosawasdi, David Padua, Soummya Kar, Jose MF Moura, Michael Franusich, et al.
High-Assurance SPIRAL: End-to-End Guarantees for Robot and Car Control
IEEE Control Systems, 2017

Contact

email: vzaliva@cmu.edu
twitter: @vzaliva
web: https://github.com/vzaliva/helix