Verified Translation Between Purely Functional and Imperative Domain Specific Languages in HELIX

Vadim Zaliva[1] Ilia Zaichuk[2] Franz Franchetti[1]

[1]Carnegie Mellon University, Pittsburgh, PA, USA

[2]Taras Shevchenko National University, Kyiv, Ukraine

VSTTE'20

July 2020

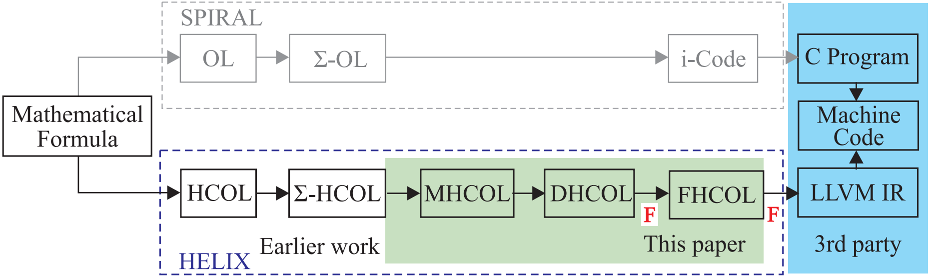
SPIRAL (foundation and inspiration)

- A program generation system which can generate high-performance code for a variety of linear algebra algorithms, such as discrete Fourier transform, discrete cosine transform, convolutions, and the discrete wavelet transform.
- It is developed since year 2000 by interdisciplinary team from CMU, ETH Zurich, Drexel, UIUC, and industry collaborators.
- It optimizes for multiple cores, single-instruction multiple-data (SIMD) vector instruction sets, and deep memory hierarchies.
- Main focus on linear operators.
- Footed in linear algebra and matrix theory.
- Written in GAP language with numerous extensions in C.
- Uses C compiler as machine code generation backend.
- Main application: Digital Signal Processing.

HELIX (our work)

- HELIX is inspired by SPIRAL.
- Focuses on automatic translation of a class of mathematical expressions to code.
- Revealing implicit iteration constructs and re-shaping them to match target platform parallelizm and vectorization capabilities.
- Rigorously defined and formally verified.
- Implemented in Coq proof assistant.
- Allows non-linear operators.
- Presently, uses SPIRAL as an optimization oracle, but we verify its findings.
- Uses LLVM as machine code generation backend.

Spiral and HELIX



1. Mathematical formula
2. The dataflow (SPIRAL: *OL* language, HELIX: *HCOL* language)
3. The dataflow with implicit loops: (SPIRAL: Σ-*OL* language, HELIX: Σ-*HCOL* language
4. Imperative program: (SPIRAL: iCode language, HELIX: *F-HCOL* language)
5. Mainstream programming language code: (SPIRAL: C Program, HELIX: LLVM IR program)

Approach

1. Translating a purely functional program into imperative language.
2. Mapping the layout of Σ-*HCOL* data to *D-HCOL* memory and variables.
3. Mapping Σ-*HCOL* sparse vector abstraction to partially initialized memory blocks.
4. Switching from mixed to deep embedding.
5. Handling errors.
6. Switching from *carrier type* to IEEE 754 floating-point numbers.
7. Switching from natural numbers to fixed bit-length machine integers.
8. Proving semantic equivalence between the original Σ-*HCOL* expression and the generated *D-HCOL* program.

Σ-*HCOL* Language

1. Purely functional
2. Statically typed
3. The main data type is a finite length sparse vector of *carrier type* values.
4. No error handling, since potential error situations, like out-of-bounds vector index access, are eliminated by strong, dependent typing.
5. Mixed-embedded in Coq

Σ-*HCOL* operators (basic)

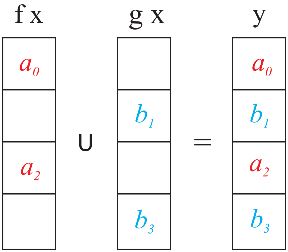
- `IdOp` – no-op.
- `Embed i n` – Takes an element from a single-element input vector and puts it at a specific index in a sparse vector of given length.
- `Pick i` – Selects an element from the input vector at the given index and returns it as a single element vector.
- `Scatter f` – Maps elements of 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.
- `Gather f` – Works in a similar manner to `Scatter`, except the index mapping function f is used in the opposite direction – to map the output indices to the input ones.
- `SHPointwise f` – Similar to the `map` function in Haskell.
- `SHBinOp f` – Similar to the `map2` function in Haskell, applied to the first and the second half of the input vector.
- `SHInductor n f` – Iteratively applies given function `f` to the input `n` times.

Σ-*HCOL* operators (higher-order)

- `liftM_HOperator hop` – “lifts” *HCOL* operators, so they can be used in Σ-*HCOL* expressions.
- `HTSUMUnion sop1 sop2` – A higher-order operator applying two operators to the same input and combining their results (discussed in more detail below).
- `SafeCast sop` – A higher-order operator, wrapping another Σ-*HCOL* operator. While not changing the values computed by the wrapped operator, it adds a monadic wrapper to track sparsity properties.
- `UnSafeCast sop` – Similar to `SafeCast` but uses a different monadic wrapper.
- `IUnion f (fam: {x:nat | x<n}→SHOperator)` – Iteratively applies indexed family of n operators to the input and combines their outputs element-wise using the given binary function `f`. This is an abstraction for parallel loops.
- `IReduction f (fam: {x:nat | x<n}→SHOperator)` – Similar to `IUnion` but without assumption of non-overlapping sparsity.
- `SHCompose sop1 sop2` – Functional composition of operators.

HTSUMUnion operator example

It is a higher-order operator parameterized by two operators, `f` and `g`. Given an input vector, `HTSUMUnion` applies them both to the vector and combines their results using vector union.



In *structurally correct Σ-HCOL* expression, it is guaranteed (proven) that both inputs to such a union will have disjoint sparsity patterns which guarantees that we will never try to combine two non-sparse elements.

MHCOL Language

1. 1-to-1 correspondence with Σ-*HCOL* by replacing vectors with *memory blocks*
2. "M" stands for *memory*
3. Purely functional
4. Dynamically typed (sizes of mem blocks not enforced)
5. The main data type is a memory block of *carrier type* values.
6. Has error handling
7. Mixed-embedded in Coq

Vectors as memory blocks

Sparse vectors in Σ-*HCOL* are an algebraic abstraction for *memory blocks*. Each memory block is represented as a dictionary in which the keys are memory offsets and the values are memory values of a carrier type. There is no mapping for keys corresponding to sparse elements.

0	A
1	
2	B
3	C

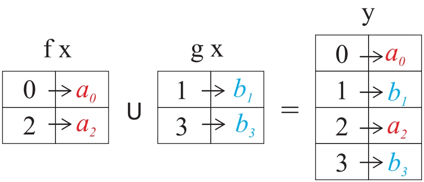
Sparse vector

0 →	A
2 →	B
3 →	C

Dictionary

HTSUMUnion in *MHCOL*

Each of the two operators `f` and `g`, applied to the input vector `x`, produces a corresponding dictionary, and the two dictionaries have disjoint key sets: $[0; 2]$ and $[1; 3]$, respectively. They are then combined into the final resulting dictionary `y`.



MHCOL mixed embedding

The following record type is used to define an operator:

```
Record MSHOperator {i o: ℕ} : Type := mkMSHOperator {
  mem_op: mem_block → option mem_block;
  mem_op_proper: Proper ((equiv) ⟹ (equiv)) mem_op;
  m_in_index_set: FinNatSet i;
  m_out_index_set: FinNatSet o; }.
```

It is indexed by the dimensions of the input and output memory blocks. The fields include: a function implementing the operation on memory blocks which can fail (returning `None`); a *proper morphism* instance for this function with respect to the setoid equality `equiv` (required because the carrier type is abstract); and the two sets which define input and output memory access patterns.

MHCOL memory safety properties

All *MHCOL* operator implementations must satisfy the following *memory safety* properties:

1. When applied to a memory block with all memory cells in `m_in_index_set` mapped to values, `mem_op` will not return an error.
2. The `mem_op` must assign a value to each element in `m_out_index_set` and must not assign a value to any element outside of `m_out_index_set`.
3. The output block of `mem_op` is guaranteed to contain no values at indices outside of operators' declared output size.

We have formulated these properties as a typeclass, `MSHOperator_Facts`, and proven instances of it for all operators.

Σ-*HCOL* to *MHCOL* semantics preservation

The semantic equality for a pair of Σ-*HCOL* and *MHCOL* operators is defined as the SH_MSH_Operator_compat typeclass. It ensures that two operators have the same dimensionality, the same input and output patterns (index sets), and are both structurally correct (through the presence of respective SHOperator_Facts and MSHOperator_Facts instances). In addition to these properties, the main semantic equivalence statement to be proven is:

```
mem_vec_preservation:
∀ (x:svector i),
(∀  (j: ℕ) (jc: j < i), in_index_set sop (mkFinNat jc) → Is_Val (Vnth x jc))
→
Some (svector_to_mem_block (op sop x)) = mem_op mop (svector_to_mem_block x)
```

Informally it could be stated as:

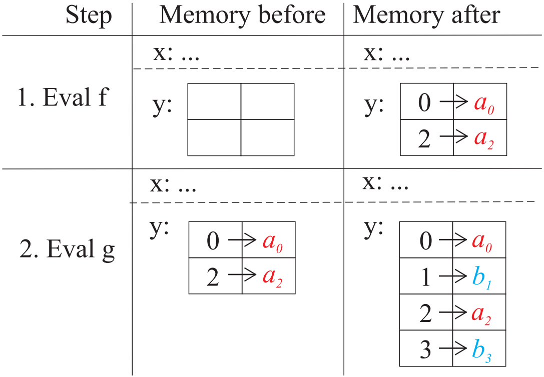
> *For any vector which complies with the input sparsity contract of the Σ-HCOL operator, an application of the MHCOL operator to such vector, converted to a memory block, must succeed and return a memory block which must be equal to the memory block produced by converting the result of the Σ-HCOL operator.*

DHCOL Language

1. Imperative.
2. The execution model assumes an *environment* (variables) and memory.
    1. Lexically scoped environment variables are in SSA form.
    2. The operators can modify memory.
3. Each *MHCOL* operator is translated into not one but a sequence of *D-HCOL* operators.
4. Has error handling.
5. Has *operators* and *expressions*.
6. Equipped with *big-step* operational semantics.
7. Deep-embedded in Coq.

HTSUMUnion in *DHCOL*

Our earlier example, the `HTSUMUnion` operator, could be viewed imperatively as a sequential execution of two operators and a combination of their results. Since output key index sets are guaranteed not to overlap, these operators could be computed independently (or even in parallel) and could write to the same output dictionary, without the risk of overwriting each others' results.



DHCOL operators

```
Inductive DSHOperator :=
| DSHAssign (src dst: MemVarRef) (* memory cell assignment *)
| DSHIMap (n: ℕ) (x_p y_p: PExpr) (f: AExpr) (* indexed [map] *)
| DSHBinOp (n: ℕ) (x_p y_p: PExpr) (f: AExpr) (* [map2] on two halfs of [x_p] *
| DSHMemMap2 (n: ℕ) (x0_p x1_p y_p: PExpr) (f: AExpr) (* [map2] *)
(* recursive application of [f]: *)
| DSHPower (n:NExpr) (src dst: MemVarRef) (f: AExpr) (initial: CT.t)
(* evaluate [body] [n] times. Loop index will be bound during body
eval: *)
| DSHLoop (n:ℕ) (body: DSHOperator)
(* allocates new uninitialized memory block and makes the pointer to it
available in evaluation context at De Bruijn index 0 while the
[body] is evaluated: *)
| DSHAlloc (size:NT.t) (body: DSHOperator)
(* initialize memory block indices [0-size] with given value. *)
| DSHMemInit (size:NT.t) (y_p: PExpr) (value: CT.t)
(* copy memory blocks. Overwrites output block values, if present: *)
| DSHMemCopy (size:NT.t) (x_p y_p: PExpr)
| DSHSeq (f g: DSHOperator) (* execute [g] after [f] *).
```

“Pure” *DHCOL* programs

Each *MHCOL* operator is a function $x \mapsto y$ where x and y are memory blocks. It is a *pure function* without side effects, whose output y depends on x and other variables in scope. A *DHCOL* translation of this *MHCOL* operator is an imperative program. One memory block will correspond to x, and some other block will correspond to y. The formalization of the class of *DHCOL* programs representing pure functions is expressed as the DSH_pure typeclass:

```
Class DSH_pure  (d: DSHOperator) (y: PExpr) := {
  mem_stable: forall σ m m′ fuel,
    evalDSHOperator σ d m fuel = Some (inr m′) ->
      forall k, mem_block_exists k m <-> mem_block_exists k m′;
  mem_write_safe: forall σ m m′ fuel,
    evalDSHOperator σ d m fuel = Some (inr m′) ->
      (forall y_i , evalPexp σ y = inr y_i -> memory_equiv_except m m′ y_i)
}.
```

It has the following two properties:

1. *memory stability* states that the operator does not free or allocate any memory blocks
2. *memory safety* states that the operator modifies only the memory block referenced by the pointer variable y, which must be valid in σ.

MHCOL to *DHCOL* semantics preservation

Now we can proceed to formulate the semantic equivalence between an *MHCOL* operator and a "pure" *DHCOL* program.

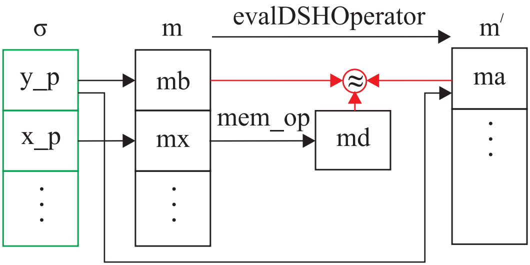
```
Class MSH_DSH_compat
      {i o: ℕ} (σ: evalContext) (m: memory)
      (mop: @MSHOperator i o) (dop: DSHOperator)
      (x_p y_p: PExpr) `{DSH_pure dop y_p} := {
      eval_equiv: ∀ (mx mb: mem_block),
          (lookup_Pexp σ m x_p = inr mx) → (lookup_Pexp σ m y_p = inr mb) →
          (h_opt_opterr_c
             (λ  md m' ⇒ err_p (λ  ma ⇒ SHCOL_DSHCOL_mem_block_equiv mb ma md)
                             (lookup_Pexp σ m' y_p))
             (mem_op mop mx)
             (evalDSHOperator σ dop m (estimateFuel dop))); }.
```

The equality is defined if both operators err or both succeed, in which case, their results must satisfy a provided sub-relation. The sub-relation (expressed via lambda) does additional error handling via `err_p` to ensure that `y_p` lookup succeeds in $\mathtt{m}'$. Finally, the equality is reduced to the predicate `SHCOL_DSHCOL_mem_block_equiv` relating `mb`, `ma`, and `md`.

MHCOL and *DHCOL* equality relation

`SHCOL_DSHCOL_mem_block_equiv` represents the relation between:

- `mb` - memory state of the output block before *DHCOL* execution
- `ma` - memory state of the output block after *DHCOL* execution
- `md` - values of changed output block elements after *MHCOL* evaluation



MemOpDelta relation on memory blocks

```
Definition SHCOL_DSHCOL_mem_block_equiv (mb ma md: mem_block) : Prop
:= ∀ i, MemOpDelta
       (mem_lookup i mb)
       (mem_lookup i ma)
       (mem_lookup i md).

Inductive MemOpDelta (b a d: option CarrierA) : Prop :=
| MemPreserved: is_None d → b = a → MemOpDelta b a d
| MemExpected: is_Some d → a = d → MemOpDelta b a d
```

Informally, it could be stated as:

> *For all memory indices in* `md` *where a value is present, the value at the same index in* `ma` *should be the same. For indices not set in* `md`*, the value in* `ma` *should remain as it was in* `mb`*.*

Future work

1. *FHCOL* specialization of *DHCOL* with machine floating-point and fixed-length integer types (done but out of scope of this paper)
2. *DHCOL* to *FHCOL* translation correctness proof using numerical analysis (future work)
3. *DHCOL* to *LLVM IR* compiler (done)
4. *DHCOL* to *LLVM IR* compiler correctness proof (paper submitted)

Questions?

- Project Page: spiral.net/software/helix.html
- GitHub github.com/vzaliva/helix
- Vadim Zaliva ✉ vzaliva@cmu.edu @vzaliva