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

Vadim Zaliva[1][0000−0002−9145−3288], Ilia Zaichuk[2][0000−0003−1617−3259], and Franz Franchetti[1][0000−0002−3529−8973]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] Taras Shevchenko National University, Kyiv, Ukraine

Abstract. HELIX is a formally verified language and rewriting engine for generation of high-performance implementation for a variety of numerical algorithms. Based on the existing SPIRAL system, HELIX adds the rigor of formal verification of its correctness using the Coq proof assistant. It formally defines a series of domain-specific languages starting with *HCOL*, which represents a computation data flow. HELIX works by transforming the original program through a series of intermediate languages, culminating in LLVM IR. In this paper, we will focus on three intermediate languages and the formally verified translation between them. Translation between these three languages is non-trivial, because each subsequent language introduces lower-level abstractions, compared to the previous one. During these steps, we switch from pure-functional language using mixed embedding to a deep-embedded imperative one, also introducing a memory model, lexical scoping, monadic error handling, and transition from abstract algebraic datatype to floating-point numbers. We will demonstrate the design of these languages, the automatic reification between them, and automated proofs of semantic preservation, in Coq.

1 Introduction

With the current level of sophistication of hardware architectures, the problem of high-performance implementation of numerical algorithms becomes challenging for manual implementation even when using optimizing compilers and is often solved by specialized code generation systems, such as SPIRAL [13].

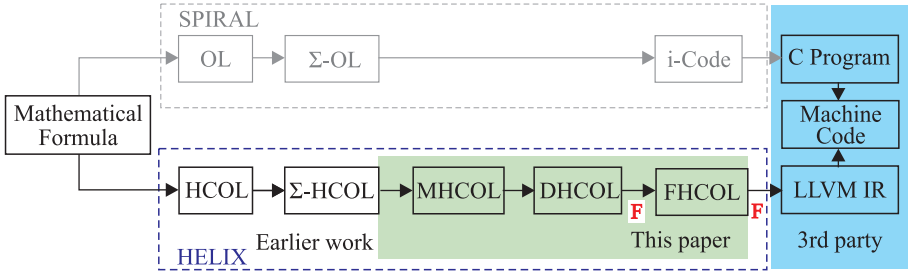
Developed over the last 20 years, the SPIRAL system has been used to generate, synthesize, and autotune programs and libraries. It works by translating rule-encoded high-level specifications of mathematical algorithms into highly optimized/library-grade implementations. SPIRAL has been used to formalize a variety of computational kernels from the signal and image processing domain, including graph algorithms, robotic vehicle control, software-defined radio (SDR), numerical solution of partial differential equations. SPIRAL is capable of generating code for multiple platforms ranging from mobile devices and multicore (desktop and server) processors to high-performance and supercomputing systems [7].

When SPIRAL is applied to generate high-performance libraries used in mission critical software, the question arises as to what kind of assurances could be made about the correctness of the generated code. The goal of HELIX, as a part of the High Assurance SPIRAL project [5,11], is to formally prove of the correctness of SPIRAL optimizations and code generation using Coq proof assistant.

Both SPIRAL and HELIX work by transforming an original formula through a series of intermediate languages, culminating in machine code, as shown in Figure 1. The translation steps correspond to different levels of abstraction:

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: *FHCOL* language)
5. Mainstream programming language code: (SPIRAL: C Program, HELIX: LLVM IR program)

The dataflow language is very close to mathematical notation and can represent a wide class of relevant mathematical formulae. As a first step, SPIRAL attempts to deconstruct the original expression into simpler expressions, which, combined by a function composition, represent a data-flow graph of the computation [8]. The resulting expression is then translated into another language, called *Σ-OL* which adds the implicit representation of iterative computations. Next, the *Σ-OL* expression is rewritten using a series of rewrite rules, driven by the extensive knowledge base of SPIRAL's optimization algorithms, into a shape which lends itself to generating the most efficient code for the target platform. Subsequently, an *Σ-OL* expression is compiled into an intermediate imperative language. By doing this, SPIRAL converts the dataflow graph into a sequence of loops and arithmetic operations. Finally, an intermediate imperative language representation, 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 expression.



Legend: F-*future work*

Fig. 1. SPIRAL/HELIX transformation stages

This paper describes the design of a part of the HELIX system, a formally verified version of SPIRAL. Implementing such a system broadly consists of defining and formalizing all intermediate languages, writing translators between them, and proving semantic preservation of each such translation. The scope of this paper is shown by the shaded box labeled "This paper" of Figure 1. The definition of *HCOL* and *Σ-HCOL* languages and proving correctness of their translation was addressed in previous papers: [16,17]. This paper focuses on *MHCOL*, *DHCOL*, and *FHCOL* languages. Unlike SPIRAL, HELIX uses LLVM IR instead of C as the last intermediate translation step. This decision was based on the ease of formally proving the semantics preservation of this step. Some features that SPIRAL relies upon still have no adequate formalization in CompCert [10], the most developed formally verified C compiler. On the other hand Vellvm [19] project provides almost all LLVM IR formalization we require. Even though we already developed the *FHCOL* to LLVM IR compiler, its design and verification will be addressed separately in future papers.
The main contributions we present in this paper are:

1. Demonstration of a working approach to semantics-preserving, two-step translation from a mixed-embedded dependently-typed purely functional language to a deep-embedded imperative one.
   (a) The initial translation of *Σ-HCOL* into an intermediate *MHCOL* language lowering the abstraction level by introducing lower-level data representation and error handling.
   (b) Subsequent compilation of *MHCOL* into *DHCOL* language. Formalization of memory model, type system, evaluation contexts, and small-step operational semantics of the target language. Mapping algebraic abstraction of partial computations via sparse vectors into independent memory updates. Formalizing and proving the notion of semantic preservation between *MHCOL*'s denotational semantics and *DHCOL*'s small step operational semantics.
2. Demonstration of a framework for switching from abstracted *reals* and *natural numbers* to IEEE 754 floating-point numbers and fixed bit-length machine integers. Our approach with two versions of the same language parameterized by different types provides a convenient framework for numerical stability and overflow-safety proofs.

2 The Approach

We start with *Σ-HCOL* language. It is a purely functional *operator language* with mixed embedding in Coq and is built around the concept of *operators* from multi-linear algebra, which are defined as maps on vector spaces [6]. It operates on finite length sparse vectors of abstract *carrier type*. Our intermediate goal is to compile it (via intermediate languages) into an imperative *DHCOL* program suitable for LLVM IR translation. This undertaking involves several distinct challenges:

1. Translating a purely functional program into imperative language.
2. Mapping the layout of *Σ-HCOL* data to *DHCOL* 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 *DHCOL* program.

In this paper, we will discuss how these challenges were addressed, what technical decisions were made, and the lessons learned. The translation is performed via series of intermediate languages, as shown in Figure 1.

With each step, the level of abstraction moves from purely mathematical operations on abstract algebraic types down towards LLVM IR instructions operating on registers and memory locations. The semantic preservation is proven from each language to the next one in the chain.

Our source *Σ-HCOL* [17] language is mixed-embedded in Coq and purely functional. The main data type is a finite length sparse vector of *carrier type* values. *Σ-HCOL* programs have no error handling, since potential error situations, like out-of-bounds vector index access, are eliminated by strong, dependent typing.

In *Σ-HCOL*, we use sparse vectors as an abstraction for partial computations. Each operator can perform a computation of some elements of a vector, leaving others undefined. To perform algebraic transformations on *Σ-HCOL* expressions, sparsity is rigorously tracked, but sparse (undefined) vector elements are assigned implicit "default" value [17]. While such default values are used in algebraic equational theory that supports underlying *Σ-HCOL* rewriting rules, they are not supposed to contribute to the final result, which must depend solely on dense (defined) elements of input vectors.

The 15 *Σ-HCOL* operators are listed with their informal descriptions below:

1. `IdOp` – no-op.
2. `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.
3. `Pick i` – Selects an element from the input vector at the given index and returns it as a single element vector.
4. `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.
5. `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.
6. `SHPointwise f` – Similar to the `map` function in Haskell.
7. `SHBinOp f` – Similar to the `map2` function in Haskell, applied to the first and the second half of the input vector.

8. `SHInductor n f` – Iteratively applies given function `f` to the input `n` times.
9. `liftM_HOperator hop` – "lifts" *HCOL* operators, so they can be used in *Σ-HCOL* expressions.
10. `HTSUMUnion sop1 sop2` – A higher-order operator applying two operators to the same input and combining their results (discussed in more detail below).
11. `SafeCast sop` – A higher-order operator, wrapping another *Σ-HCOL* operator. While it does not change the values, computed by the wrapped operator, it adds a monadic wrapper to track sparsity properties[3].
12. `UnSafeCast sop` – Similar to `SafeCast` but uses a different monadic wrapper[3].
13. `IUnion f (fam: {x:nat|x<n} → SHOperator)` – Iteratively applies indexed family of n operators to the input and combines their outputs elementwise using the given binary function `f`. This is an abstraction for parallel loops.
14. `IReduction f (fam: {x:nat|x<n} → SHOperator)` – Similar to `IUnion` but without assumption of non-overlapping sparsity.
15. `SHCompose sop1 sop2` – Functional composition of operators.

Let us consider more closely the example of the *Σ-HCOL* operator, `HTSUMUnion`. It is a higher-order operator parameterized by the two operators, `f` and `g`. Given an input vector, the operator applies them both to the vector and combines their results using *vector union*, as shown in Figure 2.

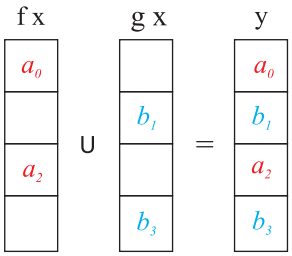


Fig. 2. `HTSUMUnion` in *Σ-HCOL*

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. The *structural correctness* property is an invariant of the previous steps in the HELIX processing chain, and the *Σ-HCOL* expressions that we are dealing with are guaranteed to satisfy it.

[3]The details of our monadic approach to sparsity tracking are out of scope of this paper, but discussed in detail in [17].

3 *MHCOL*: an Intermediate Language

Sparse vectors in *Σ-HCOL* are an algebraic abstraction for *memory blocks*. We make this explicit in an intermediate mixed-embedded language, *MHCOL* (M stands for *memory*). 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 values.

An examples of both representations is shown in Figure 3. It shows a sparse vector with three initialized elements, A, B, and C, and a dictionary with three corresponding keys.

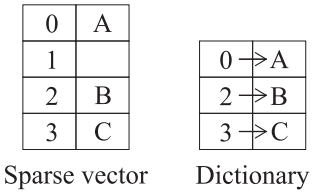


Fig. 3. Sparse vectors as dictionaries

`HTSUMUnion` in *MHCOL* is shown in Figure 4. Each of the two operators `f` and `g`, applied to the input vector `x`, produces a corresponding dictionary, and the two dictionaries have disjoint keys: $[0;2]$ and $[1;3]$, respectively. They are then combined into the final resulting dictionary `y`.

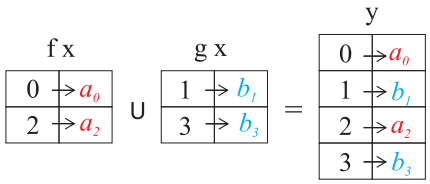


Fig. 4. `HTSUMUnion` in *MHCOL*

With this change of data representation, we move away from the algebraic nature of *Σ-HCOL* towards a lower-level representation. In this representation, an actual value must be associated with a key in a dictionary before it can be accessed. Trying to access an uninitialized key is an error. It means that *MHCOL* operators could return errors and thus have a type: mem_block $\rightarrow$ option mem_block. However, we prove that our translation of structurally correct *Σ-HCOL* programs produces *MHCOL* programs that do not err when applied to sufficiently initialized memory blocks.

Like in *Σ-HCOL*, we use *mixed embedding* [4] (a combination of *shallow* and *deep embedding*) to represent *MHCOL* operators. The following record type is used (we assume the reader is familiar with Coq):

```
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 dimensions of input and output memory blocks. The fields include: a function implementing the operation on memory blocks which can fail (returning `None`); a *proper morphism* [3] 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.

All *MHCOL* operator implementations must satisfy certain *memory safety* properties. We have formulated these properties as the typeclass, `MSHOperator_Facts`, and proven instances of it for all operators. This is a similar approach to what we took with *Σ-HCOL*, but the properties are different:

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 with index in `m_out_index_set` and must not assign a value to any element with index not in `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.

The semantic equality for a pair of *Σ-HCOL* and *MHCOL* operators is defined as the `SH_MSH_Operator_compat` typeclass. It ensures that they have the same dimensionality, the same input and output patterns (index sets), and structural correctness of *Σ-HCOL* and *MHCOL* operators (by 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 Sigma-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.

For regular operators, `SH_MSH_Operator_compat` instances could be proven directly. For higher-order operators, the proofs are predicated on `SH_MSH_Operator_compat` assumptions for all operators involved. Some operators may have additional prerequisites. For example, for `HTSUMUnion`, output index sets of `f` and `g` must be disjoint.

Translation from *Σ-HCOL* to *MHCOL*, is implemented using the Coq metaprogramming plugin, TemplateCoq [2]. For translated programs, we use proof automation to prove that `SH_MSH_Operator_compat` holds between the original and the compiled programs. This approach is known as *translation validation*.

4 *DHCOL*: an Imperative Language

The next language in the translation sequence is called *DHCOL* (D stands for *deep-embedded*). While *Σ-HCOL* and *MHCOL* have one-to-one correspondence between the operators, this no longer holds true for *Σ-HCOL* to *DHCOL*. *DHCOL* is a lower-level language, so each *MHCOL* operator is translated into not one but a sequence of *DHCOL* operators. Another distinction is that *DHCOL* is *deep embedded*. However, a more important difference is that it is no longer *functional* but *imperative*. The execution model assumes an *environment* (variables) and memory. The operators can have side effects, modifying the memory but not the environment.

The language design decisions for *DHCOL* were guided by the needs of an intermediate representation language between *MHCOL* and LLVM IR. It is not meant to be a general-purpose programming language and contains only features required to represent *DHCOL* programs. By constraining it in such way, we have kept it small and made it easier to prove statements about it.

Our earlier example, `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, as shown in Figure 5, without the risk of overwriting each others' results.

Finally, because we need to work with multiple memory blocks, we organize them into a *memory*, which is just a dictionary of memory blocks. Such a two-level hierarchical memory model is very similar to the memory model used in CompCert [10] and Vellvm [19].

In addition to memory, there is an *evaluation context* which holds all variables in scope. Variables are typed and can hold natural numbers, *carrier type* values, and memory pointers.

We provide *small-step operational semantics* of *DHCOL* by an `evalDSHOperator: evalContext → DSHOperator → memory → fuel → option (err memory)` function which, given an *evaluation context*, a memory state, and an operator, returns a modified memory state. It uses *fuel* to make it easier to prove that it always terminates. It should be noted that a semantic step is expressed as a transition between memory states. The environment stays

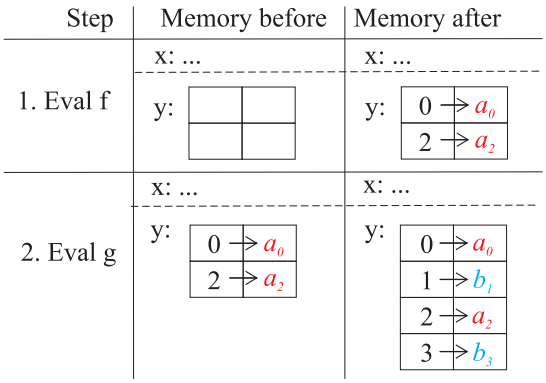


Fig. 5. `HTSUMUnion` in *DHCOL*

unchanged, and operator side effects are limited to modifying memory values. The reason for that is the design of the language; variables are statically scoped and are in *single static assignment* (SSA) form.

The reification of *MHCOL* to *DHCOL* is also implemented using Template Coq. In addition to operators, we translate arithmetic expressions on natural numbers and carrier type values. During expression translation, we enforce the restriction that only supported operations like + are allowed. A high-level reification and validation approach is discussed in [18].

The reification process is then invoked with the `Run TemplateProgram (reifySHCOL shcol "dhcol")` command in Coq. It translates Σ-*HCOL* expression `shcol` and creates a new definition with the name `dhcol` initialized with its *DHCOL* translation.

4.1 Definition

DHCOL expressions are made up of variables from an evaluation context (referenced by de Bruijn indices), operations (e.g., + and −), and constants. There are four types of expressions: `NExpr` - expressions on natural numbers; `AExpr` - expressions of carrier type values; `PExpr` - pointer expressions; and `MExpr` - memory block expressions. The first two, `NExpr` and `AExpr`, evaluate to natural and carrier type values, respectively. The latter two, `PExpr` and `MExpr`, both evaluate to memory blocks. The variables in the evaluation context are typed and could have one of three types: *natural numbers*, *carrier type* values, or *pointers to memory blocks*.

The are ten *DHCOL* operators, defined by an inductive type shown in Listing 1.1.

```
Inductive DSHOperator :=
| DSHNop (* no-op *)
| 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] *).
```

Listing 1.1. *DHCOL* operator type

4.2 Proof of Semantics Preservation

While the regular *MHCOL* operators translate to a single *DHCOL* instruction, the higher-order operators translate into a sequence of instructions, with placeholders filled with *DHCOL* translations of their respective parameters. For example, *MHCOL*'s `(MSHIReduction i o n z f op_family)` operator is compiled to the following *DHCOL* program:

```
DSHSeq
  (DSHMemInit o y_p z)
  (DSHAlloc o (DSHLoop n (DSHSeq dop_family (DSHMemMap2 o y_p' (PVar 1) y_p' df)))))
```

The parameters of `MSHIReduction` above are: the dimensions of the input and the output vectors (i and o respectively), the size n of the operator family `op_family`, and initialization value z. In *DHCOL*, `df` and `dop_family` correspond to `f` and `op_family`, respectively, translated to *DHCOL*.

Operators `DSHAlloc` and `DSHLoop` introduce two new variables: the pointer to newly allocated memory block and the loop index. Inside the loop they are referenced by their respective de Bruijn indices as `(PVar 1)` and `(PVar 0)`. The `dop_family` takes the loop index to access the family operator member to evaluate on each iteration which is then executed writing output to temporary memory block. The output of `MSHIReduction` is assumed to be written to a memory block referenced by variable `y_p`, and `y_p'` is the same variable with the de Bruijn index increased by two (to accommodate for the loop index and a new variable, holding a reference to the newly allocated temporary memory block).

We want to prove that our translation from *MHCOL* to *DHCOL* preserves the semantics. As with other HELIX languages, we use automated *translation validation* approach. To allow automatic proof of translation results, we need to

prove correctness lemmas for each *MHCOL* operator and its *DHCOL* representation. Then, these lemmas could be applied recursively, hierarchically descending the structure of the *MHCOL* reified expression.

The first step in the process is to formalize the notion of semantic equivalence between the purely functional language with denotational semantics (*MHCOL*) and the imperative language with operational semantics (*DHCOL*). Each *MHCOL* operator is a function $x \mapsto y$ where `x` and `y` are memory blocks[4]. It is a *pure function* without side effects, whose output `y` depends on `x` and other variables in scope. On the other hand, a *DHCOL* translation of this *MHCOL* operator is an imperative program that can read variables available in the *evaluation context*, and it also can read and modify the memory. One block from this memory will correspond to `x`, and some other block will correspond to `y`. Being a translation of a pure function, the operator can modify only `y`. The formalization of the class of *DHCOL* programs representing pure functions is expressed as `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:

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

Now, we can proceed to formulate the semantic equivalence between an *MHCOL* operator and a "pure" *DHCOL* program. Since the *MHCOL* part of this relation is a function, we need to universally quantify on all possible inputs. Since *DHCOL* operators read and modify memory, the input and output of this function must correspond to some existing memory blocks. In *DHCOL* memory, locations could be accessed via pointer variables only, so we state that there are two pointer variables in the evaluation context corresponding to the input and output memory block locations. For convenience, we define semantic equivalence as a type class parameterized by the respective *MHCOL* and *DHCOL* operators, the evaluation context, and by the name of the input and output pointer variables in this context. Additionally, the purity of the *DHCOL* operator must be guaranteed by providing a `DSH_pure` instance.

[4]We omit error handling for now.

```
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))); }.
```

In the listing above, `h_opt_opterr_c` deals with error handling. While `mem_op` has simple error reporting via option type, `evalDSHOperator` has two-level error handling, distinguishing between running out of fuel and other errors. The equality is defined if both operators err (for whatever reason) 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`.

Figure 6 shows the origin of these values in a case where no errors occur. Legend: σ is an evaluation context, `m` and $\mathtt{m}'$ are memory states before and after execution of the `evalDSHOperator`. The `ma` corresponds to a memory block in $\mathtt{m}'$ referenced by `y_p`. The `md` is the result of applying the *MHCOL* operator to `mx`.

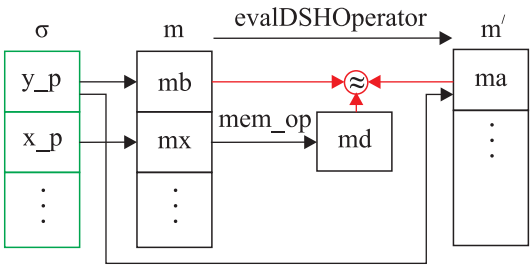


Fig. 6. *DHCOL* and *MHCOL* equality relation

To understand this relation, we must recall, that in *Σ-HCOL*, sparse vectors represent the results of partial computation. Sparse elements correspond to as yet uncomputed values, while dense elements are already computed. Performing a union of the resulting sparse vectors represents the combining of several partial computations. Replacing immutable vectors with mutable memory blocks allows us to replace the operation of combining computation results with a simple memory update. Following this reasoning, the result of the *MHCOL* operator application (called `md`, where "d" stands for *delta*) is a memory block contain-

ing values only at the indices that we need to update. The values at all other indices must remain unchanged. On the other hand, in *DHCOL*, we know the memory state before the operator evaluation and the updated state after it has been evaluated. Thus, `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

This relation is implemented via the element-wise relation, `MemOpDelta`, which is lifted to memory blocks as `SHCOL_DSHCOL_mem_block_equiv`:

```
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`.

Once we have proven `SH_MSH_Operator_compat` instances for all *MHCOL* operators and their corresponding *DHCOL* equivalents, we can automatically generate proof for the result of any *MHCOL* to *DHCOL* translation as an instance of this class top top-level *MHCOL* and *DHCOL* expressions. During this proof automation, we need to recursively descend on an *MHCOL* expression. The reason for this is that mapping between the two is not injective, and compiling two different *MHCOL* operators could result in similar *DHCOL* constructs not easily distinguishable by simple matching on the structure. Whereas *MHCOL* operators could be uniquely matched.

5 Connecting the Dots: from *FHCOL* to LLVM IR

Dealing with floating-point numbers presents a distinct set of challenges. Instead of introducing floating-point numbers from the very start, we work on an abstract data type (generalized reals) up to and including *DHCOL* language. Switching to floating-point numbers is done by introducing yet another intermediate language, *FHCOL* (F stands for *floating-point*), which operates on IEEE 754 numbers. *FHCOL* and *DHCOL* share the same memory model, but elements are now IEEE 754 *floats* instead of values of the abstract *carrier type*. This is still a higher level

than that of the Vellvm memory model, which deals with bytes. However, the *FHCOL* memory model can be unambiguously mapped to the Vellvm memory model.

We apply a similar treatment to integers used in arithmetic expressions to calculate memory indices. Since the indices are non-negative by definition, we represent the integers as *natural numbers* in all intermediate languages up to *FHCOL*, where we replace them with `int64`.

The translation between *DHCOL* and *FHCOL* is trivial, since both languages are implemented as module instances parameterized by different types (for floats and integers). They both a members of a language family indexed by numeric types. This approach allows us to easily define both language syntax and semantics without code duplication. It also allows some lemmas to proven for both languages generally. The translation from *DHCOL* to *FHCOL* is implemented in Gallina.

5.1 Correctness Proof using Numerical Analysis

The relation in the real domain between the results of the evaluation of the structurally similar expressions in these two languages could encapsulate all numerical analysis properties, such as error bounds and numerical stability.

For integers, we need to perform a very simple bounds check to avoid integer overflows in the arithmetic expressions. The ranges of arithmetic expressions could be estimated based on the ranges of their components. In *DHCOL*, these components are either constants or loop indices. Since loop indices are also bound by constant loop dimensions, overflow analysis is possible.

For floating-point numbers, we identified three approaches. None have yet been implemented, and all represent future research directions.

1. Offline Uncertainty Propagation. In the most general case, an uncertainty propagation approach could be applied [9]. Using it, we can estimate error bounds for compiled *FHCOL* expressions. Unfortunately, in many cases, the error bounds are too large to be useful for practical applications. This analysis could be very easily plugged into our verification framework at the *DHCOL* to *FHCOL* verification step.

2. Problem-specific Error Estimation. One of the primary intended uses of HELIX is to validate cyber-physical systems. In such settings, the problem domain could inform additional physical constraints which will allow us to provide stronger guarantees, such as tighter error bounds. Therefore, instead of trying to solve this problem in general, we will allow users to plug in their own reasoning, by providing a lemma which guarantees that for a particular expression and its value ranges, the floating-point approximation meets the user's given criteria. This analysis could be implemented by a user and plugged into our verification framework at the *DHCOL* to *FHCOL* verification step.

3. Online Uncertainty Propagation. This approach is similar to the Abstract Interpretation on interval domain, where the values are represented as intervals which are computed at runtime. The result of the computation is not a single value but an interval. Because it is computed for concrete values, it is usually much narrower than one estimated using offline uncertainty propagation. In some cases, it could be proven that it is smaller than the *machine epsilon* of the floating-point representation, collapsing the output interval into a single floating-point number. The price of this approach is that additional computations have to be performed at runtime affecting the performance. The unique advantage of HELIX here is that such computations could be compiled into highly efficient parallelized and vectorized code using SPIRALs optimizations. For this approach, we will introduce yet another language named *IHCOL* which replaces the abstract data type with an interval represented as two IEEE floating point numbers for its bounds. Compared to other approaches, implementing this approach will require greater changes to HELIX, including a new LLVM compiler and its verification.

5.2 Compiling to LLVM IR

Finally, *FHCOL* is compiled into LLVM IR language. A compiler from *FHCOL* to LLVM IR was implemented using Template Coq. To prove semantic equivalence of *FHCOL* programs and their IR translations, we rely on the VELLVM project, which provides formal semantics of IR. The proof involves interaction trees [15], and detailed discussion of it is beyond the scope of this paper.

5.3 Implementation Details and Related Work

There are also many smaller but useful language design and proof techniques not covered in this paper due to space constraints. Presently HELIX consists of 43,393 lines of Coq code. Examples of such techniques include proof automation, monadic error handling, meta-programming translation techniques, dealing with imperative programs with "holes" in the presence of de Bruijn indices (using resolvers), among others. Interested readers are encouraged to see the HELIX source code at https://github.com/vzaliva/helix.

There are several projects for certified compilation from functional to imperative languages. *CertiCoq* translates Gallina programs to CompCert's *Clight*. The goal is much more ambitious than ours, as the aim is to translate not a domain specific language like *Σ-HCOL* but a dependently typed general purpose language. Interestingly, all three guiding principles cited in [1] also apply to our approach. Some of their transformation steps could be related to ours. For example, going from dependently typed *Σ-HCOL* to *MHCOL*, we perform nominal *type erasure*. However, there are some differences at later stages. For example, unlike HELIX, they use a continuation-passing style representation. They prove compiler correctness, while we rely on automated translation validation.

Another related project is *CakeML*, which also targets a subset of a general-purpose language (Standard ML). Unlike HELIX, *CakeML* uses higher-order

logic (HOL) to specify *functional big-step semantics* [12]. There are similarities with our approach, as we also use a *definitional interpreter* [14] written in Gallina, to define the semantics of the higher-order language, *FHCOL*. The main differences are: small-step versus big-step semantics and translation validation versus certified compilation. Additionally our programs can not diverge, and while we technically use *fuel* to simplify termination checking, it is different from the *clock* usage in CakeML.

6 Conclusions and Future Work

Our successful completion of this penultimate verification step in the HELIX translation chain brings us one step closer to our final goal of completing HELIX, a formally verified end-to-end high performance code synthesis system based on SPIRAL.

The main contribution of this work is demonstration of an approach to verified translation from mixed-embedded purely functional operator language (Σ-*HCOL*) to deep-embedded imperative language (*DHCOL*) via an intermediate language (*MHCOL*). We have gradually introduced and proven: error handling, a memory model, and evaluation contexts. Our proof approach constrains each language program by a set of properties (*structural correctness* for Σ-*HCOL*, *memory safety* for *MHCOL*, and *purity* for *DHCOL*), using them to prove each translation step. Such properties carry some useful information from step to step in a generic but usable form. For example, when we introduce error handing in *MHCOL*, we prove a property under which no errors would occur in *MHCOL* programs translated from *structurally correct* Σ-*HCOL*. Similarly, in *DHCOL*, a program can modify arbitrary memory blocks, but we recognize a subset of *DHCOL* programs representing pure functions which have no side effects beyond modifying a single output memory block.

Future work will include the two main directions: First, completing the proof of translation chain by proving the last step of LLVM IR code generation. We made significant progress in this direction jointly with the Vellvm team and expect to finish and publish the results shortly. The second direction is related to floating-point related proofs using one of the strategies outlined in Section 5.1. We feel that we did most of preparatory work defining all languages and proof frameworks required to successfully complete this step. It must be noted that the remaining part of the work belongs mostly to the field of numerical analysis rather than formal methods and programming languages.

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: The Third International Workshop on Coq for Programming Languages (CoqPL) (2017)
2. Anand, A., Boulier, S., Cohen, C., Sozeau, M., Tabareau, N.: Towards certified meta-programming with typed Template-Coq. In: ITP 2018-9th Conference on Interactive Theorem Proving (2018)

3. Castéran, P., Sozeau, M.: A gentle introduction to type classes and relations in Coq. Tech. rep., Technical Report hal-00702455, version 1 (2012)
4. Chlipala, A.: Formal reasoning about programs (2017), http://adam.chlipala.net/frap
5. Franchetti, F., Low, T.M., Mitsch, S., Mendoza, J.P., Gui, L., Phaosawasdi, A., Padua, D., Kar, S., Moura, J.M.F., Franusich, M., Johnson, J., Platzer, A., Veloso, M.M.: High-assurance spiral: End-to-end guarantees for robot and car control. IEEE Control Systems **37**(2), 82–103 (April 2017). https://doi.org/10.1109/MCS.2016.2643244
6. Franchetti, F., de Mesmay, F., McFarlin, D., Püschel, M.: 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, pp. 385–410. Springer (2009)
7. Franchetti, F., Low, T.M., Popovici, T., Veras, R., Spampinato, D.G., Johnson, J., Püschel, M., Hoe, J.C., Moura, J.M.F.: SPIRAL: Extreme performance portability. Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code" **106**(11) (2018)
8. Franchetti, F., Voronenko, Y., Püschel, M.: Formal loop merging for signal transforms. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 315–326. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065048
9. Higham, N.J.: Accuracy and stability of numerical algorithms. Society for Industrial and Applied Mathematics, 2nd edn. (2002)
10. Leroy, X., Appel, A., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Tech. rep., INRIA (2012)
11. Low, T.M., Franchetti, F.: High assurance code generation for cyber-physical systems. In: IEEE International Symposium on High Assurance Systems Engineering (HASE) (2017)
12. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: European Symposium on Programming. pp. 589–615. Springer (2016)
13. Püschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: Spiral: Code generation for DSP transforms. Proceedings of the IEEE **93**(2), 232–275 (Feb 2005). https://doi.org/10.1109/JPROC.2004.840306
14. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference - Volume 2. p. 717740. ACM 72, Association for Computing Machinery, New York, NY, USA (1972). https://doi.org/10.1145/800194.805852, https://doi.org/10.1145/800194.805852
15. Xia, L.y., Zakowski, Y., He, P., Hur, C.K., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees. In: Proceedings of the 47th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL20). ACM, New York, NY, USA (2020)
16. Zaliva, V., Franchetti, F.: Formal verification of HCOL rewriting (2015), http://www.crocodile.org/lord/Formal_Verification_of_HCOL_Rewriting_FMCAD15.pdf
17. Zaliva, V., Franchetti, F.: 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. pp. 1–9. FHPC 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3264738.3264739, http://doi.acm.org/10.1145/3264738.3264739

18. Zaliva, V., Sozeau, M.: Reification of shallow-embedded DSLs in Coq with automated verification. CoqPL, Cascais, Portugal (2019), http://www.crocodile.org/lord/vzaliva-CoqPL19.pdf
19. Zhao, J.: Formalizing the SSA-based compiler for verified advanced program transformations. Ph.D. thesis