SPIRAL Σ-HCOL formalization

Vadim Zaliva   Franz Franchetti

Department of Electrical and Computer Engineering
Carnegie Mellon University

CMU Seminar, May 2017

[1]

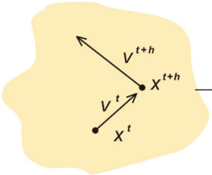
[1] This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0291. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Formal Methods

"Formal Methods refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems"

`http://shemesh.larc.nasa.gov/fm/fm-what.html`

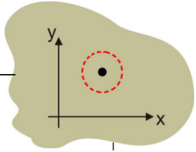
A Motivating Example



Model

Monitor formula

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



Safety constraint

HCOL Monitor formulation

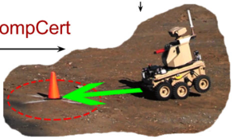
$$\mathrm{SafeDist}_{\vec{a}} : \mathbb{R} \times (\mathbb{R}^3 \times \mathbb{R}^3) \to \mathbb{Z}_2$$

SPIRAL

Generated Implementation

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

CompCert

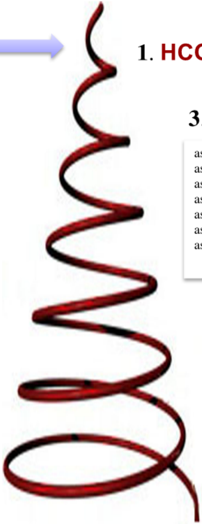


HA Robot

Under the hood – SPIRAL

Mathematical Formula

$$x^{t+h} \approx x^t + hv^{t+h}$$



1. HCOL

$$x^{t+h} \approx [I_2|hI_2](\, x^t \;\oplus v^{t+h})$$

2. Sigma-HCOL

```
SUM(
 Scat(fId(2)) *
 Prm(fId(2)) *
 Gath(H(4, 2, 0, 1)),
 ScatAcc(fId(2)) *
Scale(0.10000000000000001,
  Prm(fId(2))
 ) * Gath(H(4, 2, 2, 1)))
```

3. i-Code

```
assign(deref(Y), deref(X)),
assign(deref(add(Y, V(1))), deref(add(X, V(1)))),
assign(deref(Y), deref(add(X, V(2)))),
assign(deref(add(Y, V(1))), deref(add(X, V(3)))),
assign(deref(Y), mul(V(0.10000000000000001), deref(Y))),
assign(deref(Y), add(deref(Y), deref(Y))),
assign(deref(add(Y, V(1))), add(deref(add(Y, V(1))), deref(add(Y, V(1)))))
```

4. "C" Code

```
void sub(double  *Y, double  *X) {
   *(Y) = *(X);
   *((Y + 1)) = *((X + 1));
   *(Y) = *((X + 2));
   *((Y + 1)) = *((X + 3));
   *(Y) = (0.10000000000000001**(Y));
*(Y) = (*(Y) + *(Y));
   *((Y + 1)) = (*((Y + 1)) + *((Y + 1)));
```

Machine Code

```
movsd    (%rsi), %xmm0
movsd    %xmm0, (%rdi)
movsd    8(%rsi), %xmm0
movsd    %xmm0, 8(%rdi)
movsd    16(%rsi), %xmm0
movsd    %xmm0, (%rdi)
movhpd   24(%rsi), %xmm0
addpd    %xmm0, %xmm0
movupd   %xmm0, (%rdi)
popq     %rbp
```

Scope and Status

- Physical meaning - *out of scope*
- HCOL formalization - *done*
- HCOL correctness proofs - *done*
- Σ-HCOL formalization - this presentation
- Σ-HCOL correctness proofs - *work in progress*
- *i-Code* correctness proofs - *future work*
- *C* and machine code correctness proofs - *future work*

Pointwise as iterative sum

A pointwise application of a function $f^1 : \mathbb{R} \to \mathbb{R}$ to all elements of vector **a** could be represented as an iterative sum:

$$f^1 \,\$ \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ f(a_3) \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

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

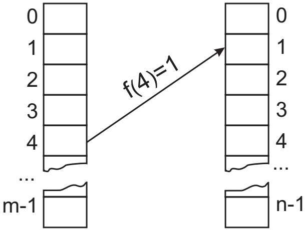
$$f^2 \,\$ \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ f(a_3) \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

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, for example, could be used to establish relation between indices of two vectors with respective sizes m and n.



Families of Index Mapping Functions

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

$$\forall j < k, \quad f_j^{m \to n} : \mathbb{I}_m \to \mathbb{I}_n \tag{1}$$

The family is called *injective* if it satisfies:

$$\forall n, \forall m, \forall i, \forall j, \quad f_n(i) = f_m(j) \implies (i = j) \land (n = m). \tag{2}$$

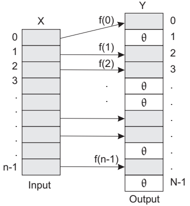
The family is called *surjective* if it satisfies:

$$\forall j, \exists n, \exists i, f_n(i) = j. \tag{3}$$

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

Scatter operator

Scatter operator's data flow:



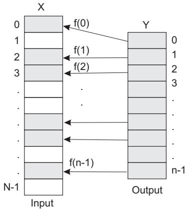
Given an *injective* index mapping function $f^{n \to N}$ the *Scatter* operator $\mathsf{S}_f : \mathbb{R}^n \to \mathbb{R}^N$ is defined as:

$$\mathbf{y} = \mathsf{S}_f(\mathbf{x}) \iff \forall i < n,\ y_j = \begin{cases} x_i & \exists j < N,\ j = f(i), \\ \theta & otherwise. \end{cases} \tag{4}$$

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*. If f is a *partial function* some elements of input vector will not be copied to the output.

Gather operator

Gather operator's data flow:



Given an index mapping function $f^{n \to N}$ the *Gather* operator $\mathrm{G}_f : \mathbb{R}^N \to \mathbb{R}^n$ is defined as:

$$\mathbf{y} = \mathrm{G}_f(\mathbf{x}) \iff \forall i < n,\ y_i = x_{f(i)} \tag{5}$$

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. If f is *bijective* and consequently $n = N$, then *Gather* is a *permutation*.

More operators

Atomic operator

Any binary function $f : \mathbb{R} \to \mathbb{R}$ could be lifted to become a standalone operator using *Atomic* operator:

$$\begin{aligned} \mathsf{A}_f : \mathbb{R}^1 &\to \mathbb{R}^1 \\ [x] &\mapsto [f(x)] \end{aligned} \tag{6}$$

Pointwise operator

We define *Pointwise* operator on vectors of dimensionality n for a family of functions f_i as:

$$\begin{aligned} \mathsf{P}^n_{f_i} : \mathbb{R}^n &\to \mathbb{R}^n \\ (x_0, x_1, \ldots, x_{n-1}) &\mapsto \left(f_0(x_0), f_1(x_1), \ldots, f_{n-1}(x_{n-1})\right) \end{aligned} \tag{7}$$

Pointwise as iterative sum

It could be shown that *Pointwise* operator could be expressed as a summation:

$$\mathsf{P}^n_{f_j} = \sum_{j=0}^{n-1} \mathsf{S}_{(j)_n} \circ A_{f_j} \circ \mathsf{G}_{(j)_n} \tag{8}$$

- Empty elements in sparse vectors are interpreted as zeros
- By $(j)_n$ we denote constant function: $\mathbb{I}_n \to \mathbb{I}_1$ with the value j.
- We will call the summand a *Sparse Embedding*

Sparsity Requirements

In general, the vectors we are dealing with are *sparse*. For example *Scatter* produces a vector with missing values. To prove Σ-HCOL language properties we need our sparse vector formalization to meet following requirements:

- distinguish empty and assigned cells
- treat empty cells as some "default" value
- such default value could depend on the context (e.g. 0 for addition but 1 for multiplication)
- in case of *SparseEmbedding* we should never attempt to combine two non-sparse elements. This type of error we will call a *collision*
- ideally, we would like to separate as much as possible sparsity tracking from actual operations on values as they represent two different aspects of computation

Sparsity Approach

An overview of our sparse vector handing approach:

- Implemented in *Coq* Proof Assistant
- Using Coq-Ext-Lib library
- Each value is tagged with two boolean flags: *is_struct* and *is_collision*
- Flags' structure forms a *Monoid* which governs how they combine
- Falags are tracked using *Writer Monad*
- Operations on values could not examine directly sparsity flags and thus could not depend on them

Monoid (Abstract algebra refresher)

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*)

Which 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

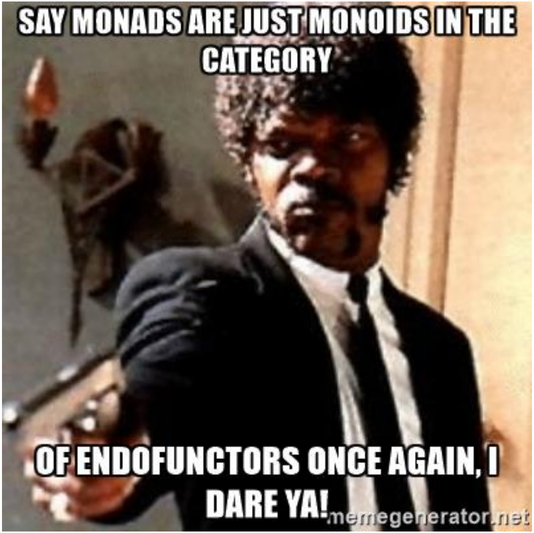
```
Record R_flags : Type := mkR_flags {is_struct: B; is_collision: B}.
```

$\texttt{Record}\ \mathcal{R}_{\text{flags}} : \texttt{Type} := \texttt{mk}\mathcal{R}_{\text{flags}}\ \{\texttt{is\_struct}: \mathbb{B};\ \texttt{is\_collision}: \mathbb{B}\}.$

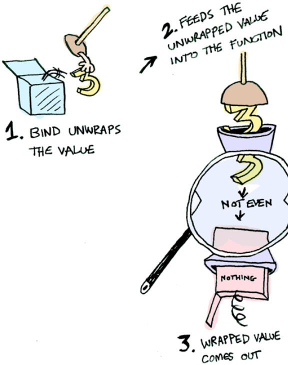
```
Definition mzero := mkRthetaFlags ⊤ ⊥.
Definition mappend (a b: R_flags) : R_flags :=
    mkRthetaFlags
      (is_struct a && is_struct b)
      (is_collision a || is_collision b ||
                     (negb (is_struct a || is_struct b))).
Definition Monoid_R_flags : Monoid R_flags := Build_Monoid 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.

What is a moand?



Monad intuition



"Monads apply a function that returns a wrapped value to a wrapped value" [2]

[2] Image credit: `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

Monad in *Coq*

A simplified[3] definition of *Monad* class from *Coq ExtLib*:

```
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 called a *type constructor*

ret
: "wraps" a value into a monad

bind
: takes a wrapped value, a function which returns a wrapped value and returns a wrapped value

[3] Removed universe polymorphism

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 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: ∀ (m: Type → Type) {Monad m} (T U: Type),
    (T → U) → (m T → m U)

liftM2: ∀ (m: Type → Type) {Monad m} (T U V: Type),
    (T → U → V) → (m T → m U → m V)
```

Sparse Operator Example

Now we can define an operator:

```
Definition Pointwise (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 ret x)*
- a vector of raw values could be extracted from the result x by simply applying *(vector.map evalWriter x)*.
- Correctness condition on the resulting vector x could checked using:

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

Iterative Operators – from dense to sparse

We have shown earlier that *Pointwise* operator on $\mathbb{R}^n$ could be expressed as a summation:

$$\mathsf{P}^n_{f_j}\, x = \sum_{j=0}^{n-1} \left(\mathsf{S}_{(j)_n} \circ A_{f_j} \circ \mathsf{G}_{(j)_n}\, x \right)$$

- This formulation was using dense vectors, without collision tracking. Now we would like to extend it to sparse vectors with collision tracking.
- It was using summation to combine elements. We would like to generalize it to other operations such as multiplication.
- We would like to generalize this notation to *iterative operators* using pointfree notation.

Operator Families

Similarly to as how we defined a family of index functions earlier we define a *family* F of k operators:

$$\forall j < k, \quad F_j : \mathcal{B}^m \to \mathcal{D}^n \tag{9}$$

- All operators in the family have the same type.
- Instead of $\mathbb{R}$ we use abstract types $\mathcal{B}$ and $\mathcal{D}$.
- In subsequent slides we will use uppercase calligraphic letters to denote abstract types.

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:

$$
\begin{aligned}
&\vec{\diamond} : \mathcal{A}^n \to \mathcal{A}^n \to \mathcal{A}^n \\
&((a_0, a_1, \ldots, a_{n-1}), (b_0, b_1, \ldots, b_{n-1})) \mapsto \\
&(a_0 \diamond b_0, a_1 \diamond b_1, \ldots, a_{n-1} \diamond b_{n-1})
\end{aligned} \tag{10}
$$

Next, we can define *operator diamond*:

$$
\begin{aligned}
&\mathring{\diamond} : (\mathcal{A}^n \to \mathcal{A}^m) \to (\mathcal{A}^n \to \mathcal{A}^m) \to (\mathcal{A}^n \to \mathcal{A}^m) \\
&(F, G) \mapsto (\mathbf{x} \mapsto F(\mathbf{x}) \,\vec{\diamond}\, G(\mathbf{x}))
\end{aligned} \tag{11}
$$

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:

$$\begin{aligned}
&\mathop{\Diamond}_{i=0}^{n-1} F_i : \mathcal{A}^n \to \mathcal{A}^n \\
&\mathbf{x} \mapsto \begin{cases} \mathbf{0}^n & \text{if } n = 0, \\ \left(F_{n-1} \mathbin{\mathring{\diamond}} \left(\mathop{\Diamond}_{j=0}^{n-2} F_j\right)\right)(\mathbf{x}) & \text{otherwise.} \end{cases}
\end{aligned} \tag{12}$$

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 $\mathcal{R}_{\text{flags}}$ state) and summation operator. To do so we specalize previous notation as follows:

```
Definition 𝒜 := ℛθ.
Definition ⋄ := liftM2 (+).
Definition ⋄⃗ := vector.map2 ⋄. (* generic *)
Definition ⋄̊ f g := λ x ⇒ (f x) ⋄⃗ (g x).  (* generic *)
Definition 0ⁿ := vector.const (ret 0) n.
```

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

$$\text{Pointwise}_{n,f} = \mathop{\Diamond}_{j=0}^{n-1} \left(\mathsf{S}_{(j)_n} \circ A_{f_j} \circ \mathsf{G}_{(j)_n} \right) \tag{13}$$

Summary

What have been formalized so far in Coq:

- Index functions and their families
- Σ-HCOL operators and their families
- Sparse vectors handling
- Collision tracking
- Generalized notion of iterative operators

Next steps:

- Proof techniques for proving structural properties
- Σ-HCOL rewriting proof automation using *compilation validation* approach in Coq, taking into account both value and structural correctness.

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