Formal Verification of HCOL Rewriting

Vadim Zaliva and Franz Franchetti
Carnegie Mellon University

June 7, 2015

***Abstract*—The SPIRAL system allows users to synthesize resource-efficient, platform-adapted implementations of controllers for vehicular systems. It uses Hybrid Control Operator Language (HCOL) which takes advantage of advanced mathematical constructs to express controller specification, which is transformed through a series of steps into highly-optimized code for a target platform in a language, such as C or C++. An ongoing effort, part of which is described in this paper, is aimed at providing high-assurance guarantees for SPIRAL-generated controller implementations using a formal-methods based approach using the Coq proof assistant.**

I. Introduction

This work was done in the scope of a High-Assurance Cyber Military Systems (HACMS) DARPA project. The goals were to synthesize executable code for the control system of a robot satisfying certain safety and security properties and to produce machine-checkable proofs assuring that this code implements functional specification. We use the Coq proof assistant[1] to interactively develop machine-checked proofs.

A high-level approach for synthesizing high-assurance code for vehicular systems is discussed in [2]. The authors show how a vehicular control system could be specified in HCOL language, allowing automatic code generation. Moreover, the paper summarizes how HCOL specifications are transformed to the code via a series of steps, and it outlines directions for how each of these steps can be formally verified. Following these directions, our work thus far verifies the HCOL rewriting step.

Leroy et al. in [3] formally define *semantic preservation*, which states that the source and compiled programs (s and c respectively) must have exactly the same observable behaviours b. They use notation $s \approx c$ to express such semantic preservation. *Translation validation* is one of approaches, which we used to prove semantics preservation of SPIRAL's HCOL transformations (modelled as `Comp` function). With this approach, the compiler is complemented by a *validator* – a predicate function `Validate(s,c)` which validates $s \approx c$ after the compilation. Thus, under *translation validation*, instead of validating the compiler, we just validate the compilation results[3]:

$$\forall s\ c, \mathrm{Validate}(s, c) = \mathbf{true} \Longrightarrow s \approx c \tag{1}$$

Additionally, the correctness of the validator needs to be established.

To verify the SPIRAL HCOL rewriting subsystem, we first formalize HCOL in the Coq proof assistant as described in Section II. One of the challenges is the choice of representation for data types on which operators are defined. Using $\mathbb{R}$ as a main carrier type is too limiting, as many HCOL operators can be applied to a variety of types, such as *integers* or *complex numbers*. Moreover, at later stages of formalization of the SPIRAL processing chain, we have to ensure validity of our proofs for hardware types, such as *IEEE float* or *double*.

Because most of SPIRAL's HCOL transformations are algebraic in nature, we have taken abstract algebra as the foundation for formalization of operator types. Coq does not include comprehensive abstract algebra definitions as a part of its standard library, so we used the *MathClasses* library described in [4]. In this library, algebraic structures are represented as interfaces, expressed as Coq *type classes*, which mandate what operations the types should provide and what proofs they need to supply.

II. HCOL Formalization

A. HCOL Overview

HCOL allows us to express matrix and vector products in compact form using *operators*. Below is a brief overview of the HCOL language and definitions of some HCOL operators along with a sample SPIRAL rewriting rule.

a) Basic HCOL operators: are functions which take a vector of reals and return a vector of reals. The mathematical definitions of some of the operators are shown below. In formalization and actual implementation, all input and output values are coerced to vectors via implicit isomorphisms such as $\mathbb{R}^m \times \mathbb{R}^n \cong \mathbb{R}^{m+n}$ and $\mathbb{R} \cong \mathbb{R}^1$.

$$\mathrm{Polynomial}_{n,(\mathbf{a}\in\mathbb{R}^n)} : \mathbb{R} \to \mathbb{R} : \quad x \mapsto \sum_{i=0}^{n} \mathbf{a}_i x^i$$

$$\mathrm{ScalarProd}_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} : \quad (\mathbf{x}, \mathbf{y}) \mapsto \sum_{i=0}^{n-1} \mathbf{x}_i \mathbf{y}_i$$

$$\begin{aligned} \mathrm{Reduction}_{n,f,\mathrm{id}} &: \mathbb{R}^n \to \mathbb{R} : \\ &\quad \mathbf{x} \mapsto f(\mathbf{x}_{n-1}, f(\mathbf{x}_{n-2}, f(\dots f(\mathbf{x}_0, \mathrm{id})\dots) \\ \mathrm{Pointwise}_{n,f} &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n : \\ &\quad (\mathbf{x}, \mathbf{y}) \mapsto f(\mathbf{x}_0, \mathbf{y}_0) \oplus \dots \oplus f(\mathbf{x}_{n-1}, \mathbf{y}_{n-1}) \end{aligned}$$

b) Higher Order Operators: allow us to construct new operators by combining existing ones. An example of such an operator is the *function composition* of two operators:

$$(\circ) : (S \to R) \times (D \to S) \to (D \to R) : \quad (f, g) \mapsto f \circ g$$

Breakdown Rules: The core of SPIRAL is the rule-rewriting system which rewrites HCOL expressions according to *breakdown rules*. Each rule is expressed in the form $A \to B$ which means, whenever expression A is encountered, it can be replaced with expression B. Below is an example of a SPIRAL breakdown rule:

$$\mathrm{ScalarProd}_n \to \mathrm{Reduction}_{n,(a,b)\mapsto a+b,0} \circ \mathrm{Pointwise}_{n,(a,b)\mapsto ab} \tag{2}$$

It represents the scalar product operator as a function composition of the reduction and pointwise operators.

B. HCOL Operators in Coq

HCOL operators are defined as Coq functions. While in SPIRAL implementation all operators use type $\mathbb{R}$, in our Coq definitions, we have defined operators on a generic type, stipulating for each of them required properties of such type via *type classes*.

For example, the definitions of Polynomial operator is shown below:

```
Fixpoint EvalPolynomial {n} `{SemiRing A}
          (a: vector A n) (x:A) : A :=
  match a with
      nil ⇒ 0
    | cons a0 p a' ⇒ a0 + (x × (EvalPolynomial a' x))
  end.
```

It requires carrier type A to be a *semiring*, and the definition has type $A^n \to A \to A$.

C. Abstract Syntax

An HCOL expression can be represented by an Abstract Syntax Tree (AST) where leaves correspond to *basic operators* while internal nodes correspond to *higher order operators*. An AST for a subset of HCOL expressions can be formalized in Coq using the following *inductive type*:

```
Inductive HOperator : nat → nat → Type :=
| HOReduction: ∀ m (f: A→A→A)
   `{pF: !Proper ((=) ==> (=) ==> (=)) f} (id:A), HOperator m 1
| HOPointWise: ∀ n (f:A→A→A)
   `{pF: !Proper ((=) ==> (=) ==> (=)) f}, HOperator (n+n) n
| HOScalarProd: ∀ {k:nat}, HOperator (k+k) 1
| HOEvalPolynomial: ∀ {n} (a:vector A n), HOperator 1 1
| HOCompose: ∀ m {k} n, HOperator k n → HOperator m k
   → HOperator m n.
```

Type constructors correspond to individual HCOL operators, both basic, such as HOScalarProd, as well as higher order, such as HOCompose.

D. Semantics

Using the type we defined, any given HCOL expression could be represented in Coq as an object of type HOperator. The semantics of these objects are defined via an evaluation function, which takes an HOperator object and an input vector and returns the resulting vector:

```
evalHCOL: ∀ {m n}, HOperator m n → vector A m → vector A n.
```

The implementation of this function is mostly straightforward mapping between HOperator constructors and our functions implementing HCOL operators. To be able to evaluate any operator, evalHCOL is defined on a carrier type A, which must be an instance of all type classes mandated by the implementations of individual operators.

III. Validating HCOL rewriting

Given the original and transformed HCOL expressions (s and c respectively per Equation 1), we verify SPIRAL HCOL rewriting by defining the *Validate* function as an HCOL operator equivalence, as discussed below.

A. Equality

The Coq default notion of equality (eq function or $=$ notation) provides *definitional equality* which requires objects' internal structures to be the same in order for them to declared equal. However, such a notion of equality is too restrictive. For example, it doesn't allow us to work with rational numbers represented by non-reduced integer fractions, i.e. $\frac{2}{2} \neq \frac{4}{4}$. Thus, we use a generalized equivalence relation which we can define for a particular type. In other words, we work on a type equipped with an equivalence relation.

Type class Equiv defines the *equal* relation for a given type. Its subclass, Setoid, additionally requires this relation to be an *equivalence relation* by requiring proofs that it is transitive, commutative, and reflexive.

After we mandate our carrier type to be a Setoid, we prove that vectors of this type are also Setoids and define the equality of HCOL operators:

```
Global Instance HCOL_equiv {i o:nat}: Equiv (HOperator i
o) := fun a b ⇒ ∀ (x:vector A i), evalHCOL a x = evalHCOL b x.
```

Informally, it could be described as follows: two operators a and b are equal if for any input vector x the values of evalHCOL a x and evalHCOL b x are also equal.

Finally, we can declare our HOperator type to be a Setoid by using the equality definition above and by proving transitivity, commutativity, and reflexivity.

B. Proving Breakdown Rules

During the first stage of SPIRAL processing, an HCOL specification is transformed into an optimized form via a series of term rewriting steps. To satisfy high-assurance requirements, we need to produce a machine-checkable proof that these rewriting steps have no effect on the semantics of the specification.

We now have all the tools which allow us to formalize HCOL breakdown rules in Coq. We can express each rule as a lemma stating equality of two operators. For example, the following HCOL breakdown rule from Equation 2 could be expressed as the following Coq lemma:

```
Lemma breakdown_ScalarProd: ∀ {h:nat},
      HOScalarProd (h:=h) =
      HOCompose _ _
               (HOReduction _ (+) 0)
               (HOPointWise _ (.*.)).
```

Each such lemma corresponds to a breakdown rule that needs to be proven. Because HCOL operator equality is transitive, we can easily prove equality of the source and result of a chain of breakdown rule applications by proving breakdown rule lemmas for each step.

IV. Results

We applied our formalization and validation approach to validate the HCOL rewriting part of the SPIRAL which were used to generate implementation of the safety monitor which provides collision safety guarantees for the *Landshark* robot.

The rewriting of the constraint operator was validated by proving 76 lemmas describing 7 breakdown rules and took 2,138 lines of Coq code. Although the initial formalization of HCOL covered only a modest subset of the language, it laid the foundation and validated a methodology for proceeding with the full HCOL formalization.

References

[1] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2012, version 8.4. [Online]. Available: http://coq.inria.fr

[2] F. Franchetti, A. Sandryhaila, and J. R. Johnson, "High assurance SPIRAL," in *SPIE 9091, Signal Process. Sensor/Information Fusion, Target Recognit.*, I. Kadar, Ed., Jun. 2014, p. 90911O.

[3] X. Leroy, "Formal verification of a realistic compiler," p. 107, 2009.

[4] B. Spitters and E. van der Weegen, "Type Classes for Mathematics in Type Theory," *Math. Struct. Comput. Sci.*, vol. 21, no. 04, pp. 795–825, Jul. 2011.