Reification of shallow-embedded DSLs in Coq with automated verification

Vadim Zaliva [1],Matthieu Sozeau [2]

[1]Carnegie Mellon University
[2]Inria & IRIF, University Paris 7

CoqPL'19

Introduction

Shallow and Deep Embedding

There are several ways to embed a domain specific language (DSL) in a host language (HL):

Deep DSL's AST is represented as HL data structures and semantic is explicitly assigned to it.

Shallow DSL is a subset of HL and semantic is inherited from HL.

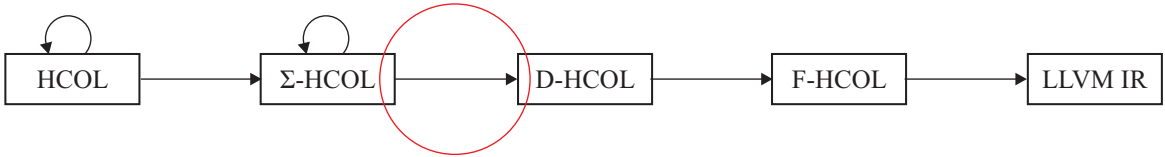
The shallow embedding is excellent for quick prototyping, as it allows quick extension or modification of the embedded language. Meanwhile, deep embeddings are better suited for code transformation and compilation.

Motivation: HELIX project

- HELIX is 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.
- HELIX is inspired by SPIRAL.
- Focuses on automatic translation 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 it's findings.
- Uses LLVM as machine code generation backend.
- Main application: Cyber-physical systems.

Motivation (contd.)

HELIX system uses transformation and translation steps involving several intermediate languages:



Σ-*HCOL* language is shallow embedded, while *D-HCOL* is deep embedded. We were looking for a translation technique between the two which is:

1. Automated - Can automatically translate arbitrary valid Σ-*HCOL* expressions.
2. Verified - Provides semantic preservation guarantees.

Example

An Example of Shallow Embedding in Coq

As an example let us consider a simple language of arithmetic expressions on natural numbers. It is shallow-embedded in Gallina and includes constants, bound variables, and three arithmetic operators: $+$, $-$, and $*$.

Provided that $a, b, c, x \in \mathbb{N}$, the expression below is an example of a valid expression in the source language:

```
2 + a*x*x + b*x + c.
```

An Example of Deep Embedding in Coq

A deep-embedded variant of the same language includes the same operators but will be defined by an inductive type of its AST:

```
Inductive NExpr: Type :=
| NVar  : ℕ → NExpr
| NConst: ℕ → NExpr
| NPlus : NExpr → NExpr → NExpr
| NMinus: NExpr → NExpr → NExpr
| NMult : NExpr → NExpr → NExpr.
```

Our sample expression, in deep-embedded target language, looks like:

```
NPlus (NPlus (NPlus (NConst 2)
  (NMult (NMult (NVar 3) (NVar 0)) (NVar 0)))
    (NMult (NVar 2) (NVar 0))) (NVar 1)
```

Template Coq

Template Coq

TemplateCoq is quoting library for Coq. It takes Coq terms and constructs a representation of their syntax tree as a Coq inductive data type:

```
Inductive term : Set :=
| tRel       : ℕ → term
| tVar       : ident → term (* For free variables (e.g. in a goal) *)
| tMeta      : ℕ → term
| tEvar      : ℕ → list term → term
| tSort      : universe → term
| tCast      : term → cast_kind → term → term
| tProd      : name → term (* the type *) → term → term
| tLambda    : name → term (* the type *) → term → term
| tLetIn     : name → term (* the term *) → term (* the type *) → term → term
| tApp       : term → list term → term
| tConst     : kername → universe_instance → term
| tInd       : inductive → universe_instance → term
| tConstruct : inductive → ℕ→ universe_instance → term
| tCase      : (inductive * ℕ) (* * of parameters *) → term (* type info *)
                → term (* discriminee *) → list (ℕ * term) (* branches *) → term
| tProj      : projection → term → term
| tFix       : mfixpoint term → ℕ→ term
| tCoFix     : mfixpoint term → ℕ→ term.
```

The Template Monad

```
Cumulative Inductive TemplateMonad@{t u} : Type@{t} → Type :=
(* Monadic operations *)
| tmReturn : ∀ {A:Type@{t}}, A → TemplateMonad A
| tmBind : ∀ {A B : Type@{t}}, TemplateMonad A → (A → TemplateMonad B) → TemplateMonad B
(* General commands *)
| tmPrint : ∀ {A:Type@{t}}, A → TemplateMonad unit
| tmFail : ∀ {A:Type@{t}}, string → TemplateMonad A
| tmEval : reductionStrategy → ∀ {A:Type@{t}}, A → TemplateMonad A
| tmDefinition : ident → ∀ {A:Type@{t}}, A → TemplateMonad A
| tmAxiom : ident → ∀ A : Type@{t}, TemplateMonad A
| tmLemma : ident → ∀ A : Type@{t}, TemplateMonad A
| tmFreshName : ident → TemplateMonad ident
| tmAbout : ident → TemplateMonad (option global_reference)
| tmCurrentModPath : unit → TemplateMonad string
(* Quoting and unquoting commands *)
| tmQuote : ∀ {A:Type@{t}}, A → TemplateMonad Ast.term
| tmQuoteRec : ∀ {A:Type@{t}}, A → TemplateMonad program
| tmQuoteInductive : kername → TemplateMonad mutual_inductive_body
| tmQuoteUniverses : unit → TemplateMonad uGraph.t
| tmQuoteConstant : kername → 𝔹 (* bypass opacity? *) → TemplateMonad constant_entry
| tmMkDefinition : ident → Ast.term → TemplateMonad unit
| tmMkInductive : mutual_inductive_entry → TemplateMonad unit
| tmUnquote : Ast.term → TemplateMonad typed_term@{u}
| tmUnquoteTyped : ∀ A : Type@{t}, Ast.term → TemplateMonad A
(* Typeclass registration and querying for an instance *)
| tmExistingInstance : ident → TemplateMonad unit
| tmInferInstance : ∀ A : Type@{t}, TemplateMonad (option A)
.
```

Approach

Reification Approach

- Implemented as *Template Program* named `reifyNExp`
- The input expressions are given in a closed form, where all variables first need to be introduced via lambda bindings. E.g. $\mathbb{N}$ or $\mathbb{N} \to \ldots \to \mathbb{N}$.
- `reifyNExp` also takes two names (as strings). The first is used as the name of a new definition, to which the expression in the target language will be bound. The second is the name to which the semantic preservation lemma will be bound.

```
Polymorphic Definition reifyNExp@{t u}
  {A:Type@{t}} (res_name lemma_name: string) (nexpr:A)
  : TemplateMonad@{t u} unit.
```

When executed, if successful, `reifyNExp` will create a new definition and new lemma with the given names. If not, it will fail with an error.

Reification Implementation

- All variables are converted into parameters via $\forall$.
- Gallina AST (as *quoted* by TemplateCoq) of the original expression is traversed and an expression in target language is generated.
- Variable names are converted to de Bruijn indices.

Our sample expression will be compiled to the following deep-embedded form:

```
NPlus (NPlus (NPlus (NConst 2)
    (NMult (NMult (NVar 3) (NVar 0)) (NVar 0)))
    (NMult (NVar 2) (NVar 0))) (NVar 1)
```

Semantic Preservation Approach

Semantics of our deep embedded language is defined by evaluation function:

```
Definition evalContext:Type := list ℕ.
Fixpoint evalNexp (Γ:evalContext) (e:NExpr): option ℕ.
```

The semantic preservation is expressed as a heterogeneous relation between expressions in the source and target languages:

```
Definition NExpr_term_equiv (Γ: evalContext)
(d: NExpr) (s: ℕ) : Prop := evalNexp Γ d = Some s.
```

For our sample expression the following lemma will be generated:

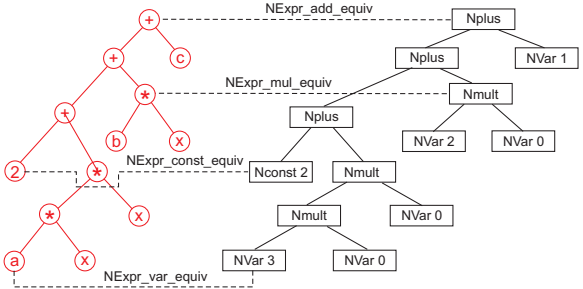
```
∀ x c b a : ℕ, NExpr_term_equiv [x; c; b; a]
    NPlus (NPlus (NPlus (NConst 2)
        (NMult (NMult (NVar 3) (NVar 0)) (NVar 0)))
        (NMult (NVar 2) (NVar 0))) (NVar 1)
    (2 + a * x * x + b * x + c)
```

Automated Proof of Semantic Preservation Lemma

The general idea is to define and prove semantic equivalence lemmas for each pair of operators. For example:

```
Lemma NExpr_mul_equiv (Γ: evalContext) {a b a' b' }:
  NExpr_term_equiv Γ a a' → NExpr_term_equiv Γ b b' →
  NExpr_term_equiv Γ (NMult a b) (Nat.mul a' b').
```

Because the expressions in the original and target languages have the same structure, such proof can be automated, for example by use of eauto tactic.



Case study

Practical use in HELIX system

The input is shallow embedded Σ-*HCOL* language:

- An operator language on finite-dimensional vectors.
- Includes meta-operators
- Includes binders.
- Operates on sparse vectors with optional collistion tracking.
- Uses logical parameters (of type `Prop`).

The output is deep embedded *D-HCOL* language:

- No logical parameters.
- Using de Bruijn indices for variables.
- Only a subset of Σ-*HCOL* operators are represented.
- Operates on dense vectors.
- A fixed set of *intirinsic* functions (e.g. $+, -, max$) is defined.

Σ-*HCOL* Example

```
dynwin_SHCOL1 =
λ a : vector CarrierA 3,
SafeCast (SHBinOp (IgnoreIndex2 Zless))
  ∘ HTSUMUnion plus
  (eUnion (le_S (le_n 1)) 0
    ∘ SafeCast (IReduction plus 0
      (SHFamilyOperatorCompose
        (λ  jf : FinNat 3,
          (SHPointwise (Fin1SwapIndex jf (mult_by_nth a)))
          ∘ (SHInductor (' jf) mult 1))
        (eT (GathH1_domain_bound_to_base_bound (h_bound_first_half 1 4))))))
  (eUnion (le_n 2) 0
    ∘ SafeCast (IReduction minmax.max 0
      (λ  jf : FinNat 2,
        (SHBinOp (λ  (i : FinNat 1) (a0 b : CarrierA),
                     IgnoreIndex abs i (Fin1SwapIndex2 jf (IgnoreIndex2 sub) i a0 b)))
        ∘ (UnSafeCast (ISumUnion (λ  jf0 : FinNat 2,
                                     eUnion (proj2_sig jf0) 0
                                     ∘ eT (h_index_map_compose_range_bound ...)))))))
  : vector CarrierA 3 → SHOperator
```

D-HCOL Example

```
dynwin_DSHCOL1 =
DSHCompose
  (DSHBinOp (AZless (AVar 1) (AVar 0)))
  (DSHHTSUMUnion (APlus (AVar 1) (AVar 0))
     (DSHCompose (DSHeUnion (NConst 0) CarrierAz)
        (DSHIReduction 3 (APlus (AVar 1) (AVar 0)) CarrierAz
           (DSHCompose
              (DSHCompose
                 (DSHPointwise (AMult (AVar 0) (ANth 3 (VVar 3) (NVar 2))))
                 (DSHInductor (NVar 0) (AMult (AVar 1) (AVar 0)) CarrierA1))
              (DSHeT (NConst 0)))))
     (DSHCompose (DSHeUnion (NConst 1) CarrierAz)
        (DSHIReduction 2 (AMax (AVar 1) (AVar 0)) CarrierAz
           (DSHCompose
              (DSHBinOp (AAbs (AMinus (AVar 1) (AVar 0))))
              (DSHIUnion 2 (APlus (AVar 1) (AVar 0)) CarrierAz
                 (DSHCompose
                    (DSHeUnion (NVar 0) CarrierAz)
                    (DSHeT
                       (NPlus (NPlus (NConst 1) (NMult (NVar 1)(NConst 1)))
                          (NMult (NVar 0) (NMult (NConst 2) (NConst 1))))))))))
     : DSHOperator (1 + 4) 1
```

Reifying dependently typed operators

D-HCOL operator type family is indexed by dimensions of input and output vectors:

```
Inductive DSHOperator: ℕ → ℕ→ Type
```

To deal with it we defined a record and cast function:

```
Record reifyResult := { rei_i: ℕ; rei_o: ℕ; rei_op: DSHOperator rei_i rei_o; }.
Definition castReifyResult (i o:ℕ) (rr:reifyResult): TemplateMonad (DSHOperator i o).
```

Our reification templete program returns `TemplateMonad reifyResult`. The reification of a function composition is implemented as:

```
| Some n_SHCompose, [fm ; i1 ; o2 ; o3 ; op1 ; op2] ⇒
  ni1 ← tmUnquoteTyped ℕ i1 ;; no2 ← tmUnquoteTyped ℕ o2 ;; no3 ← tmUnquoteTyped ℕ o3 ;;
  cop1' ← compileSHCOL vars op1 ;; cop2' ← compileSHCOL vars op2 ;;
  let '(_, _, cop1) := (cop1': varbindings*term*reifyResult) in
  let '(_, _, cop2) := (cop2': varbindings*term*reifyResult) in
  cop1 ← castReifyResult no2 no3 cop1 ;; cop2 ← castReifyResult ni1 no2 cop2 ;;
       tmReturn (vars, fm, {| rei_i:=ni1; rei_o:=no3; rei_op:=@DSHCompose ni1 no2 no3 cop1 cop2 |})
```

coq-switch plugin

While parsing Coq's AST we have to do a lot of string matching, like this:

```
Definition parse_SHCOL_Op_Name (s:string): option SHCOL_Op_Names :=
  if string_dec s "Helix.SigmaHCOL.SigmaHCOL.eUnion" then Some n_eUnion
  else if string_dec s "Helix.SigmaHCOL.SigmaHCOL.eT" then Some n_eT
    ...
  else None.
```

Writing manually such biolerplate code is tiresome and error-prone. We developed *coq-switch* plugin to to automate such tasks, for any decidable type:

```
Run TemplateProgram
  (mkSwitch string string_beq
            [("Helix.SigmaHCOL.SigmaHCOL.eUnion", "n_eUnion") ;
             ("Helix.SigmaHCOL.SigmaHCOL.eT"    , "n_eT"    ) ;
               ...]
            "SHCOL_Op_Names" "parse_SHCOL_Op_Name"
  ).
```

It will generate inductive type `SHCOL_Op_Names` with constructors `n_eUnion`, `n_eT` ... and a function: `parse_SHCOL_Op_Name`: `string` → `option SHCOL_Op_Names`.

Conclusions

Σ-*HCOL* to *D-HCOL* reification in numbers

- Development time: ~2 weeks
- Lines on code:

Spec	601
Proofs	830

- Execution time (for simple expression containing 24 Σ-*HCOL* operators):

Reification	4s
Proofs	1s

Questions?

Links:

- Full example: `https://github.com/vzaliva/CoqPL19-paper`
- TemplateCoq: `https://metacoq.github.io/metacoq/`
- *coq-switch* plugin `https://github.com/vzaliva/coq-switch`
- *"HELIX: a case study of a formal verification of high performance program generation."*, Vadim Zaliva, Franz Franchetti. FHPC 2018.

Contact:

- Vadim Zaliva: `http://www.crocodile.org/lord/`
- Matthieu Sozeau: `https://www.irif.fr/~sozeau/`

Backup slides

From Denotational to Operational Semantics