Reification of shallow-embedded DSLs in Coq with automated verification*

Vadim Zaliva
Carnegie Mellon University
vzaliva@cmu.edu

Matthieu Sozeau
Inria & IRIF, University Paris 7
matthieu.sozeau@inria.fr

Abstract

Shallow and deep embeddings of DSLs have their pros and cons. For example, 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. Thus, it might be useful to be able to switch from shallow to deep embedding while making sure the semantics of the embedded language is preserved. We will demonstrate a working approach for implementing and proving such conversion using TemplateCoq.

1 Introduction

In the course of our work on the HELIX system [3], we faced the problem of writing a certified compiler for a domain specific language called Σ-HCOL, which is shallow-embedded in the Gallina language of the Coq proof assistant [2]. The approach presented in this report is the result of a summer of 2018 collaborative visit to Inria, where the use of TemplateCoq [1] was first suggested by Matthieu Sozeau and successfully implemented and used in the HELIX project.

An application of this technique to reify Σ-HCOL can be found in the HELIX source code. For illustrative purposes in this paper, we use a simpler language of arithmetic expressions on natural numbers. It is shallow-embedded in Gallina and includes constants, bound variables, and three arithmetic operators: $+$, $-$, and $*$, the complete source for which can be found in the git repository https://github.com/vzaliva/CoqPL19-paper.

1.1 Example

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.
```

Listing 1. Expression in source language

The target language includes the same operators but will be defined by an inductive type of its AST:

```
Inductive NExpr: Type :=
| NVar   : ℕ → NExpr (* using de Bruijn indices *)
| NConst: ℕ → NExpr
| NPlus : NExpr → NExpr → NExpr
| NMinus: NExpr → NExpr → NExpr
| NMult : NExpr → NExpr → NExpr.
```

Listing 2. Target language type

The expression from Listing 1, translated to the target language, looks like:

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

Listing 3. Expression in target language

The purpose of the reification step is just to switch from shallow to deep embedding. Thus, the target language syntax is supposed to be close to the source language syntax. We just change the representation and enforce the scope of the shallow embedding to ensure that only the allowed subset of Gallina is used. This makes the translation implementation fairly straightforward. Additionally, this allows automatic proof of semantic preservation as shown below.

2 Translation

The translation process performs reification of a given expression, producing a translated expression in the target language. It is implemented as a *template program* named *reifyNExp*.

The input expressions are presented in a closed form, where all variables first need to be introduced via lambda bindings. Thus, the input expression could be either just $\mathbb{N}$ in the simplest case or an n-ary function of natural numbers, for example $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. The reification template program is hence polymorphic on the input type expression.

In addition to the input expression, *reifyNExp* 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, as discussed in the next section.

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

Listing 4. Reification program

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. The reason for a failure might be that the expression contains constructs which are legal in Gallina but not part of our embedded language. The translation is

*CoqPL'19 talk extended abstract

implemented in a straightforward way, traversing the Gallina AST of the original expression that Template-Coq quoting produces.

3 Semantic Preservation

The semantics of our source language is defined by Gallina. On the other hand, the inductive type representing the target language needs to be given its own semantics. We do this by providing an evaluation function for the deeply embededd terms. It takes an *evaluation context* which holds the current values of free variables, the expression being evaluated, and returns the result as a natural number or *None* in case of error.

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

The evaluation may fail, for example, if the expression references a variable not present in the evaluation context. As variables are represented by their indices in Γ, this will happen if an index is greater or equal to the length of Γ.

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.
```

Consequently, the lemma generated by *reifyNExp* for our example in Listing 1 will state semantic equivalence between its expression and that of Listing 3:

```
∀ 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)
```

The generated lemma still needs to be proven. Because the expressions in the original and target languages have the same structure, such proof can be automated. The general idea is to define and prove semantic equivalence lemmas for each pair of operators and then add them as hints into a hint database used with the *eauto* tactic:

```
Lemma NExpr_add_equiv (Γ: evalContext) {a b a' b' }:
  NExpr_term_equiv Γ a a' → NExpr_term_equiv Γ b b' →
  NExpr_term_equiv Γ (NPlus a b) (Nat.add a' b ').

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 ').

Lemma NExpr_const_equiv (Γ: evalContext) {v v' }:
  evalNexp Γ (NConst v) = Some v' →
  NExpr_term_equiv Γ (NConst v) v '.

Lemma NExpr_var_equiv (Γ: evalContext) {v x }:
  evalNexp Γ (NVar v) = Some x →
  NExpr_term_equiv Γ (NVar v) x.
```

```
Create HintDb NExprHints.
Hint Resolve NExpr_add_equiv NExpr_mul_equiv: NExprHints.
Hint Resolve NExpr_const_equiv NExpr_var_equiv: NExprHints.

Obligation Tactic := intros; simpl; eauto 99 with NExprHints.
Run TemplateProgram (reifyNExp "Ex1_def" "Ex1_lemma" Ex1 ).
```

The ASTs of the original and translated expressions and the semantic equivalence relations between them are shown in Figure 1.

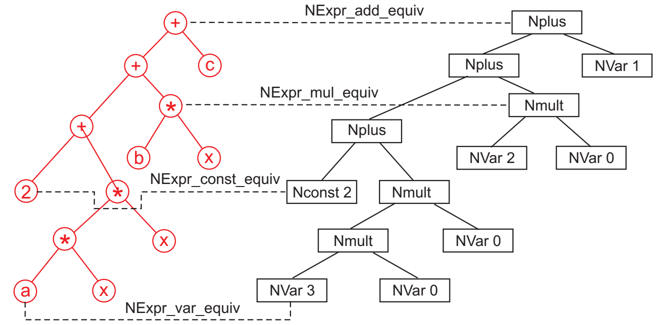

Figure 1. Semantics equivalence

4 Summary

Our approach could be summarized as follows. In order to translate a language shallowly embedded in Gallina into a deep embedding, complete the following steps:

1. Define an inductive type for the target language AST.
2. Implement an evaluation function for the target language.
3. Define a semantic equivalence relation between expressions in source and target languages.
4. Implement reification as a template program which generates an expression in the target language and a semantic preservation lemma.
5. Define and prove lemmas of semantic equivalence between operators of source and target languages and add them to the hints database.
6. Use *eauto* to prove automatically generated semantic preservation lemma.

We successfully applied this translation validation approach to the non-trivial Σ-HCOL language in HELIX, featuring binders, dependent types, and higher-order operators like map on fixed-sized vectors. We found it easy to implement compared to other means of reification like type classes or $\mathcal{L}_{tac}$ programming and recommend using it as a guide for switching from shallow to deep embedding of DSLs. In the presentation, we propose to present this case study in detail.

References

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP 2018-9th Conference on Interactive Theorem Proving*.

[2] The Coq Development Team. 2018. The Coq Proof Assistant, version 8.8.0. (Apr 2018). https://doi.org/10.5281/zenodo.1219885

[3] Vadim Zaliva and Franz Franchetti. 2018. 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 (FHPC 2018)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/3264738.3264739