

Verified Gallina Transformations in the Codegen Plugin for Rocq

Akira Tanaka

National Institute of Advanced Industrial Science and Technology (AIST)

2026-01-21 IPSJ PRO 157

Purpose

- ▶ Generation of practical C code from Rocq
- ▶ Program verification in Rocq with efficient execution in C

Rocq Proof Assistant (Formally Coq)

- ▶ It provides a pure, ML-like functional language called Gallina
- ▶ Various properties of functions written in Gallina can be verified
- ▶ Proofs are also represented in Gallina
- ▶ It includes an extraction plugin for generating OCaml code from Gallina

Codegen Plugin for Rocq

- ▶ We are developing a Rocq plugin that translates a subset of Gallina into C
- ▶ It aims to generate lower-level code than the extraction plugin
- ▶ <https://github.com/akr/codegen>

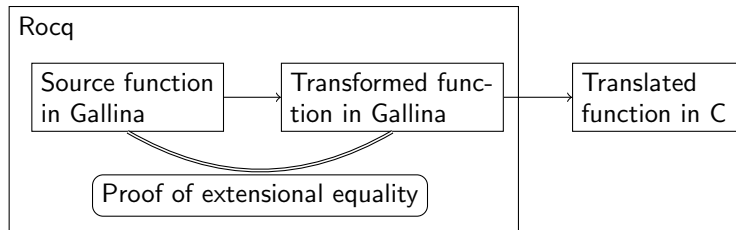
Two-Phase Translation

- ▶ Gallina-to-Gallina Transformation

Transforms functions into forms that are easy to translate into C: monomorphization, requiring application arguments to be variables, etc.

- ▶ Gallina-to-C Translation

Generates C code from monomorphic Gallina functions



Extensional Equality

- ▶ The results are equal for all arguments: $\forall args, f\ args = f'\ args$
- ▶ This property is commonly required for compilers
- ▶ For two Gallina functions f and f' , this can be proved using a standard Rocq proof.
(There is no need to define the semantics of Gallina or C on our own.)

Example: MathComp's xor function

Source function:

```
Definition addb (b : bool) : bool → bool :=  
  if b then negb else id.
```

Transformed function:

```
Definition addb' (b b2 : bool) : bool :=  
  if b then negb b2 else b2.
```

Translated function:

```
static bool addb(bool v1_b, bool v2_b) {  
  switch (v1_b) {  
    default:  
      return negb(v2_b);  
    case false:  
      return v2_b;  
  }  
}
```

Matchapp Transformation

addb' is obtained by following matchapp rule:

$(\text{match } e \text{ with } C_1 \Rightarrow t_1 \mid \dots \mid C_n \Rightarrow t_n \text{ end } \text{args})$

$\triangleright (\text{match } e \text{ with } C_1 \Rightarrow t_1 \text{ args} \mid \dots \mid C_n \Rightarrow t_n \text{ args end}).$

Without this transformation, C-level support for closures is required.

```
static bool addb(bool v1_b, bool v2_b) {
  closure c;
  switch (v1_b) {
    default:
      c = /* generate closure: fun (x:bool) ⇒ negb x */;
      break;
    case false:
      c = /* generate closure: fun (x:bool) ⇒ x */;
      break;
  }
  return /* call closure: c v2_b */;
}
```

Extensional Equality of `addb` and `addb'`

Manual proof is easy:

Lemma `addb_addb'` : $\forall (b \ b2 : \text{bool}), \text{addb } b \ b2 = \text{addb}' \ b \ b2$.

Proof.

```
  intros b b2.
```

```
  destruct b.
```

```
    reflexivity.
```

```
  reflexivity.
```

Qed.

Codegen generates this proof automatically.

Proof Term of Extensional Equality of addb and addb'

Produced proof term of the manual proof.

```
Definition addb_addb' :  $\forall$  b b2 : bool, addb b b2 = addb' b b2 :=  
  fun (b b2 : bool)  $\Rightarrow$   
    if b as b0 return (addb b0 b2 = addb' b0 b2) then  
      eq_refl (* constructor of eq (reflexivity generates it) *)  
    else  
      eq_refl.
```

Source function:

```
Definition addb (b : bool) : bool  $\rightarrow$  bool :=  
  if b then negb else id.
```

The proof term performs case analysis on b , in the same way as the source function.

Example of Recursive Function: MathComp's mem_seq

Source function (specialized to nat):

```
Notation xpred0 := (fun (x : nat) => false).
Notation xpredU1 := (fun (a : nat) (p : nat → bool) =>
  (fun (x : nat) => orb (Nat.eqb x a) (p x))).
Fixpoint mem_seq (s : list nat) : nat → bool :=
  match s with
  | nil => xpred0
  | cons y s' => xpredU1 y (mem_seq s')
  end.
```

Transformed function:

```
Fixpoint mem_seq2 (s : list nat) (n : nat) : bool :=
  match s with
  | nil => xpred0 n
  | cons y s' => xpredU1 y (mem_seq2 s') n
  end.
```

Extensional Equality of mem_seq and mem_seq2

Manual proof uses induction:

Lemma mem_seq2_valid (s : list nat) (n : nat):
 mem_seq s n = mem_seq2 s n.

Proof.

induction s; simpl; try rewrite IHs; **reflexivity**.

Qed.

The proof term use the induction principle, list_ind:

Definition mem_seq2_valid

: $\forall$ (s : list nat) (n : nat), mem_seq s n = mem_seq2 s n :=

fun (s : list nat) (n : nat) $\Rightarrow$

list_ind (**fun** s0 : list nat $\Rightarrow$ mem_seq s0 n = mem_seq2 s0 n)

(eq_refl : mem_seq nil n = mem_seq2 nil n)

(**fun** (a : nat) (s0 : list nat) (IHs : mem_seq s0 n = mem_seq2 s0 n)

eq_ind_r

(**fun** b : bool $\Rightarrow$

(Nat.eqb n a || b)%bool = (Nat.eqb n a || mem_seq2 s0 n)

eq_refl IHs

: mem_seq (a :: s0) n = mem_seq2 (a :: s0) n)

Induction Principle is not Usable in General

- ▶ Induction principle follows the recursive structure of the inductive type
A recursive function can follow a different structure
- ▶ Induction principle is not generally usable
Rocq's induction principle generation doesn't work well for mutually and nested inductive types

Codegen's Strategy to Prove Recursive Functions

- ▶ Codegen does not rely on induction principles generated by Rocq
instead, it uses fixpoint expressions directly
- ▶ The basic idea of automatic proof generation is to follow the recursion structure of the source function
- ▶ The generated proofs will pass Rocq's termination checker in the same way as the source function

Contents

Background

Gallina syntax and conversion rules

Verification of the Transformation

Results

After the Submission

Summary

Gallina Syntax

$t = x$	v-variable
f	f-variable
c	constant
C	constructor
T	type
$\lambda x:T. t$	abstraction
$t u$	application
$\text{let } x := t : T \text{ in } u$	let-in
$\text{match } t \text{ as } y \text{ in } V \bar{z} \text{ return } U$	conditional
$\text{with } \overline{C \ x:\bar{T} \Rightarrow u} \text{ end}$	
$\text{fix } \overline{f/k:T := t} \text{ with for } f_j$	fixpoint

Gallina Syntax (2)

- ▶ Gallina is similar to ML
- ▶ $\text{fix } \overline{f/k:T := t} \text{ for } f_j$ corresponds $\text{let rec } \overline{f:T := t} \text{ in } f_j$.
The number k specifies the position of the decreasing argument. To reduce the fixpoint, the head of the decreasing argument must be a constructor.
- ▶ A `match` expression has a `as-in-return` clause to specify a result type that depends on the scrutinee and on the index of the scrutinee's type

Conversion Rules

beta: $E[\Gamma] \vdash ((\lambda x. t) u) \triangleright t\{x/u\}$

delta-local: $\frac{(x := t) \in \Gamma}{E[\Gamma] \vdash x \triangleright t}$ delta-global: $\frac{(c := t) \in E}{E[\Gamma] \vdash c \triangleright t}$

zeta: $E[\Gamma] \vdash \text{let } x := t \text{ in } u \triangleright u\{x/t\}$

iota-match: $\frac{E[\Gamma] \vdash C_j \bar{a} \bar{b} : T \quad |a| = \text{NP}_T}{E[\Gamma] \vdash \text{match } (C_j \bar{a} \bar{b}) \text{ with } \bar{C} \bar{x} \Rightarrow \bar{t} \text{ end} \triangleright (\lambda \bar{x}j. t_j) \bar{b}}$

iota-fix: $\frac{u_{k_j} \equiv C \bar{a} \quad |u| = k_j}{E[\Gamma] \vdash (\text{fix } f/k := t \text{ for } f_j) \bar{u}}$
 $\triangleright t_j\{\overline{f/\text{fix } (f/k := t) \text{ for } f_j}\} \bar{u}$

eta-expansion: $\frac{E[\Gamma] \vdash t : \forall x:T. U}{E[\Gamma] \vdash t \triangleright \lambda x:T. (t x)}$

Conversion Rules (2)

- ▶ The conversion rules represent the execution of Gallina
- ▶ Two terms are convertible (or definitionally equal) if they can be transformed into the same term, up to renaming of bound variables, using the conversion rules
- ▶ The type checker does not distinguish between convertible terms

Proof Term of Extensional Equality of `addb` and `addb'` (2)

Definition `addb_addb'`

```
: ∀ b b2 : bool, addb b b2 = addb' b b2 :=  
fun (b b2 : bool) ⇒  
match b as b0 return addb b0 b2 = addb' b0 b2 with  
| true ⇒ @eq_refl bool (addb' true b2)  
| false ⇒ @eq_refl bool (addb' false b2)  
end
```

- ▶ The type of the `match` expression and of each branch is `addb b0 b2 = addb' b0 b2`
- ▶ For the type of the whole `match` expression, `b0` is replaced with `b`; this is the same as the return type
- ▶ In the `true` branch, `b0` is replaced with `true`.
`addb true b2 = addb' true b2` is convertible to
`negb b2 = negb b2`.
`@eq_refl bool (addb' true b2)` has the type
`negb b2 = negb b2`
- ▶ The `false` branch is analogous

Verification of Matchapp Transformation

$(\text{match } e \text{ with } C_1 \Rightarrow t_1 \mid \dots \mid C_n \Rightarrow t_n \text{ end } args)$

$\triangleright (\text{match } e \text{ with } C_1 \Rightarrow t_1 \text{ args} \mid \dots \mid C_n \Rightarrow t_n \text{ args end}).$

- ▶ The transformation can be verified by case analysis on e
- ▶ `eq_refl` suffices to prove each branch $t_i \text{ args} = t_i \text{ args}$

Contents

Background

Gallina syntax and conversion rules

Verification of the Transformation

Results

After the Submission

Summary

Verification of Transformations

Codegen has 10 verification rules

- ▶ verify-forall
- ▶ verify-refl
- ▶ verify-applyhyp
- ▶ verify-matchapp
- ▶ verify-fix
- ▶ verify-letin-generic, verify-letin-simple
- ▶ verify-apparg
- ▶ funext-ind, funext-fun

verify-forall

verify-forall:

$$\frac{E[\Gamma; vars] \vdash p : t = t'}{E[\Gamma] \vdash \lambda vars. p : \forall vars, t = t'}$$

- ▶ Equivalent to the `intros` tactic
- ▶ E denotes the global environment
- ▶ Γ denotes the local context

verify-refl

verify-refl:

$$\frac{E[\Gamma] \vdash t \stackrel{\text{conv}}{=} t'}{E[\Gamma] \vdash \text{eq_refl } t : t = t'}$$

- ▶ Equivalent to the **reflexivity** tactic
- ▶ $t \stackrel{\text{conv}}{=} t'$ means that t and t' are convertible

verify-applyhyp

verify-applyhyp:

$$\frac{(H : t \simeq t') \in \Gamma}{E[\Gamma] \vdash H \text{ args} : t \text{ args} = t' \text{ args}}$$

- ▶ $t \simeq t'$ means that t and t' are extensionally equal

verify-matchapp

verify-matchapp:

$$\frac{\overline{E[\Gamma] \vdash p : \forall \text{cargs}, \text{BZ}[\![t \text{ margs}]\!] = \text{BZ}[\![t' \text{ margs}']\!]}}{E[\Gamma] \vdash \text{match } x \text{ as } x' \text{ return} \\ \quad \text{match } x' \text{ with } \overline{C \text{ cargs} \Rightarrow t} \text{ end margs} \\ = \text{match } x' \text{ with } \overline{C \text{ cargs} \Rightarrow t'} \text{ end margs}' \\ \text{with } \overline{C \text{ cargs} \Rightarrow p \text{ cargs}} \text{ end} \\ : \quad \text{match } x \text{ with } \overline{C \text{ cargs} \Rightarrow t} \text{ end margs} \\ = \text{match } x \text{ with } \overline{C \text{ cargs} \Rightarrow t'} \text{ end margs}'}$$

where

$$\text{BZ}[\![t \text{ y } \bar{z}]\!] = \begin{cases} \text{BZ}[\![u\{x/y\} \bar{z}]\!] & \text{if } t \text{ is } \lambda x. u \\ \text{let } x := u \text{ in } \text{BZ}[\![u' \text{ y } \bar{z}]\!] & \text{if } t \text{ is } \text{let } x := u \text{ in } u' \\ t \text{ y } \bar{z} & \text{otherwise} \end{cases}$$

verify-fix

verify-fix:

$$\frac{\overline{E[\Gamma; \overline{IH : F \simeq F'}]} \vdash p : t\{\overline{f/F}\} \simeq t'\{\overline{f/F'}\}}{\overline{E[\Gamma] \vdash (\text{fix } \overline{IH/k : (F \simeq F')} := q \text{ for } IH_s) fargs} : F_s fargs = F'_s fargs}}$$

where

$$F_i := \text{fix } \overline{f/k : T} := t \text{ for } f_i \quad F'_i := \text{fix } \overline{f/k : T} := t' \text{ for } f_i$$

$$q_j := \lambda x_1 \dots x_{n_j}. \text{let } H := p_j \text{ in}$$

$$\text{match } x_{k_j} \text{ as } y \text{ return}$$

$$F_j x_1 \dots x_{k_j-1} y x_{k_j+1} \dots x_{n_j} = F'_j x_1 \dots x_{k_j-1} y x_{k_j+1} \dots x_{n_j}$$

with

$$\frac{C \text{ cargs} \Rightarrow H x_1 \dots x_{k_j-1} (C \text{ params}_j \text{ cargs}) x_{k_j+1} \dots x_{n_j}}{\text{end}}$$

end

verify-fix (non-mutual version)

verify-fix:

$$\frac{E[\Gamma; IH : F \simeq F'] \vdash p : t\{f/F\} \simeq t'\{f/F'\}}{E[\Gamma] \vdash (\text{fix } IH/k : (F \simeq F') := q) \text{ fargs} : F \text{ fargs} = F' \text{ fargs}}$$

where

$$F := \text{fix } f/k : T := t \quad F' := \text{fix } f/k : T := t'$$

$$q := \lambda x_1 \dots x_n. \text{let } H := p \text{ in}$$

$$\text{match } x_k \text{ as } y \text{ return}$$

$$F \ x_1 \dots x_{k-1} \ y \ x_{k+1} \dots x_n = F' \ x_1 \dots x_{k-1} \ y \ x_{k+1} \dots x_n$$

with

$$\frac{C \text{ cargs} \Rightarrow H \ x_1 \dots x_{k-1} \ (C \ \underline{\text{params}} \ \text{cargs}) \ \underline{x_{k+1} \dots x_n}}{\text{end}}$$

end

verify-letin-generic, verify-letin-simple

verify-letin-generic:

x' fresh

$E[\Gamma] \vdash p : t \simeq t'$

$E[\Gamma; (x := t); (x' := t'); (H : x \simeq x')] \vdash q : u \text{ args} = u' \{x/x'\} \text{ args}'$

$$\begin{aligned} E[\Gamma] \vdash & (\text{let } x := t \text{ in} \\ & \text{let } x' := t' \text{ in} \\ & \lambda(H : x \simeq x'). q) p \\ & : (\text{let } x := t \text{ in } u) \text{ args} = (\text{let } x := t' \text{ in } u') \text{ args}' \end{aligned}$$

verify-letin-simple:

$t \stackrel{\text{conv}}{=} t'$

$E[\Gamma; (x := t)] \vdash q : u \text{ args} = u' \text{ args}'$

$$E[\Gamma] \vdash (\text{let } x := t \text{ in } q)$$
$$: (\text{let } x := t \text{ in } u) \text{ args} = (\text{let } x := t' \text{ in } u') \text{ args}'$$

verify-apparg

verify-apparg:

t and t' are not applications

$E[\Gamma] \vdash x \not\equiv x'$

$E[\Gamma] \vdash p : x = x'$

$E[\Gamma] \vdash q : t \text{ hargs } x \text{ targs } = t' \text{ hargs } x \text{ targs}'$

rew p **in** $q : t \text{ hargs } x \text{ targs } = t' \text{ hargs } x' \text{ targs}'$

where

rew p **in** $q := \text{eq_ind } x (\lambda y. t \text{ hargs } x \text{ targs } = t' \text{ hargs } y \text{ targs}') q x$

funext-ind, funext-fun

funext-ind:

t and t' are not functions

$$\frac{E[\Gamma] \vdash p : t \simeq t'}{p : t = t'}$$

funext-fun:

t and t' are functions

$$\frac{E[\Gamma] \vdash p : \forall x, t x = t' x}{p : \text{functional_extensionality } t \ t' \ p : t = t'}$$

- ▶ `verify-letin-generic` introduces an extensional equality $x \simeq x'$, whereas `verify-apparg` requires $x = x'$
- ▶ `functional_extensionality` bridges this gap
- ▶ This is required for inductive values that contain function values

Results

- ▶ 227 basic functions from Rocq and MathComp were examined
- ▶ 4 functions require non-convertible proofs:
 - ▶ `addb` in `Corelib.ssr.ssrbool`
 - ▶ `locked` in `Corelib.ssr.ssreflect`
 - ▶ `mem_seq` in `mathcomp.ssreflect.seq`
 - ▶ `pop_succn` in `mathcomp.ssreflect.ssrnat`

Codegen can generate proofs for all of them

- ▶ Monomorphization failed for one function:
 - ▶ `list_power` in `Stdlib.Lists.List`
Extending `verify-fix` would address this issue

After Submission

I upgraded Rocq from version 9.0.0 to 9.1.0, and one Codegen test failed.

```
Definition div n d :=  
  match d with  
  | 0  $\Rightarrow$  0  
  | S d'  $\Rightarrow$   
    let fix f n :=  
      match n - d' with  
      | 0  $\Rightarrow$  fun _  $\Rightarrow$  0  
      | S n''  $\Rightarrow$  fun _  $\Rightarrow$  S (f n'')  
    end tt  
  in f n  
end.
```

The argument `tt` is moved into the `match` branches, but the generated proof is rejected by Rocq 9.1.0.

Bug Report and Response

This is a minimized version of the proof term generated by Codegen. It is accepted by Rocq 9.0.0 but rejected by 9.1.0.

```
Fixpoint f (n m : nat) {struct n} : nat :=  
  let g n := match n - m with  
    | 0  $\Rightarrow$  0  
    | S n'  $\Rightarrow$  f n' m  
  end  
  
  in  
  match n with  
  | 0  $\Rightarrow$  g 0  
  | S x'  $\Rightarrow$  g (S x')  
end.
```

<https://github.com/rocq-prover/rocq/issues/21452>
However, this change was intentional rather than a bug.

The Problem and Better Strategy

The verify-fix rule introduces a **match** expression to destruct and *reconstruct* the decreasing argument x_{k_j} .

$$\begin{array}{l} q_j := \lambda x_1 \dots x_{n_j}. \text{let } H := p_j \text{ in} \\ \quad \text{match } x_{k_j} \text{ as } y \text{ return} \\ \quad \quad F_j x_1 \dots x_{k_j-1} y x_{k_j+1} \dots x_{n_j} = F'_j x_1 \dots x_{k_j-1} y x_{k_j+1} \dots x_{n_j} \\ \quad \text{with} \\ \hline C \text{ cargs} \Rightarrow \underline{H x_1 \dots x_{k_j-1}} (\underline{C \text{ params}_j \text{ cargs}}) \underline{x_{k_j+1} \dots x_{n_j}} \\ \text{end} \end{array}$$

This was not good idea

The termination checker does not recognize the reconstructed term as being the same as the scrutinee.

Our Research Position on Proofs of Recursive Functions

- ▶ **Problem:** The termination checker is not well documented. The Rocq manual describes only the basic behavior, while the implementation is continuously evolving. (The MetaRocq project aims to reimplement the termination checker in Rocq, though.)
- ▶ [Barthe2002] generates induction principles from recursive functions, but does not support `match` expressions in non-tail positions
- ▶ Our approach provides limited support for `match` expressions in non-tail positions

Possible future directions:

- ▶ Identify all `match` expressions that destruct the decreasing argument, and move (and unify) them into a tail position
- ▶ Perform more extensive case analysis in `verify-fix`
- ▶ Extend the Rocq termination checker to support *reconstruction of the decreasing argument*

More Case Analysis in Verify-Fix

Rocq 9.1.0 accepts following definition

```
Fixpoint f (n m : nat) {struct n} : nat :=  
  let g n m := match n - m with  
    | 0  $\Rightarrow$  0  
    | S n'  $\Rightarrow$  f n' m  
  end  
  
in  
match n with  
| 0  $\Rightarrow$  g 0 m  
| S x'  $\Rightarrow$   
  (* case analysis on m *)  
  match m with  
  | 0  $\Rightarrow$  g (S x') 0  
  | S m'  $\Rightarrow$  g (S x') (S m')  
  end  
end.
```

The termination checker reduces the term, and the reconstruction appears to be eliminated internally?

Summary

- ▶ We are developing the Codegen plugin to translate Gallina into C
- ▶ Codegen performs Gallina-to-Gallina transformations to ease translation into C
- ▶ Codegen automatically generates correctness proofs for the Gallina-to-Gallina transformations
- ▶ The *matchapp* transformation in recursive functions is challenging
- ▶ We mimic the recursive structure of the source function in the proof, although this approach is not fully robust in Rocq 9.1.0