Coq to C Translation with Partial Evaluation

Akira Tanaka

National Institute of Advanced Industrial Science and Technology (AIST)

2021-01-18 PEPM 2021 (online)
Purpose

- Practical C code generation from Coq
- Program verification in Coq and efficient execution in C
Coq Proof Assistant

- It provides the pure functional ML-like language, Gallina
- We can verify various properties of functions written in Gallina
- It has the extraction plugin to generate OCaml code from Gallina
We are developing a Coq plugin to translate a Gallina subset to C.

It intends to generate low-level code generation unlike the extraction plugin.

https://github.com/akr/codegen

Two-phase translation:

- Gallina to Gallina Transformation
  This includes partial evaluation
  This transformation is easily verifiable

- Gallina to C Translation
  C code generation for monomorphic Gallina function
Basic Idea

Gallina and C (and most imperative languages) shares basic features:

- function definition
- function invocation
- conditional
- variable declaration and its initialization
- variable reference
- recursive function

We can translate a Gallina subset to C without an overhead.
Mandatory Features for Low-level Programming

Our initial motivation is succinct data structures
It needs low-level features:

- various C types: 64 bit integer, SIMD register, etc.
  → Gallina inductive types are mapped to C types

- operators (+, -, *, etc.) and
  builtin functions (__builtin_popcount, etc.)
  → Gallina applications are mapped to C function calls:
    $f \ x$ in Gallina is translated to $f(x)$ in C
    $f$ can be implemented as a macro or builtin function

- loop without stack consumption (but Gallina has no loops)
  → We guarantee tail recursion elimination

These features enable us to generate low-level C functions from
monomorphic Gallina functions
Good-to-Have Features

Although we can implement monomorphic functions in Gallina but automatic transformations reduce the programmer’s effort

- Monomorphization for polymorphic functions
- Dependent type elimination for complex type computation
- Partial evaluation generalizes them

We implement a partial evaluation as Gallina to Gallina transformations
Contents

Background

Monomorphic Translation Example

Partial Evaluation with Stock Coq?

Translation of Codegen

Codegen Reduction Rules

Summary
Power Function: Gallina to Gallina

Fixpoint pow (a b : nat) : nat :=
  match b with
  | 0 ⇒ 1
  | S b’ ⇒ a * pow a b’
  end.

↓ application arguments into variables to ease code generation

Fixpoint s_pow (v1_a v2_b : nat) : nat :=
  match v2_b with
  | 0 ⇒ let v3_n := 0 in
    S v3_n
  | S v4_b_ ⇒ let v5_n := s_pow v1_a v4_b_ in
    Nat.mul v1_a v5_n
  end
Power Function: Gallina to C

```gallina
Fixpoint s_pow (v1_a v2_b : nat) : nat :=
  match v2_b with
  | 0 ⇒ let v3_n := 0 in S v3_n
  | S v4_b_ ⇒ let v5_n := s_pow v1_a v4_b_ in Nat.mul v1_a v5_n
  end

↓

static nat pow(nat v1_a, nat v2_b) {
  nat v3_n, v4_b_, v5_n;
  switch (sw_nat(v2_b)) {
    default: v3_n = 0(); return S(v3_n);
    case S_tag: v4_b_ = pred(v2_b);
    v5_n = pow(v1_a, v4_b_);
    return mul(v1_a, v5_n);
  }
}
```
User-Defined nat Implementation in C

- We can choose any implementation for nat in C
- nat implementation using uint64_t

```c
#include <stdint.h>
typedef uint64_t nat;
#define O() 0
#define S(n) ((n)+1)
#define sw_nat(n) ((n) == 0)
#define S_tag 0
#define pred(n) ((n)-1)
#define mul(x,y) ((x) * (y))
```

- uint64_t for nat works if overflow does not occur
  We provide monadification plugin for Coq for verification about overflow
  https://github.com/akr/monadification
Translation of Tail Recursion

\((^a^b * c)\)

```
(* a^b * c *)
Fixpoint powmul a b c :=
  match b with
  | 0 ⇒ c
  | S b' ⇒
    powmul a b' (a * c)
  end.
```

Tail recursion elimination for loop without stack consumption

```
static nat powmul(nat v1_a, nat v2_b, nat v3_c) {
  nat v4_b_, v5_n;
  entry_powmul:
  switch (sw_nat(v2_b)) {
    default: return v3_c;
    case S_tag:
      v4_b_ = pred(v2_b);
      v5_n = mul(v1_a, v3_c);
      v2_b = v4_b_;  
      v3_c = v5_n;
      goto entry_powmul;
    }
  }
```
Verification of Gallina to Gallina Transformation

We can verify \( s \_\text{pow} \) easily in Coq:

\[
\text{Goal } \text{pow} = s \_\text{pow}.
\]
\[
\text{Proof. } \text{reflexivity}. \text{ Qed.}
\]

This guarantees \( \text{pow} \) and \( s \_\text{pow} \) returns the same value for all arguments.
Contents

Background

Monomorphic Translation Example

Partial Evaluation with Stock Coq?

Translation of Codegen

Codegen Reduction Rules

Summary
We Want Partial Evaluation

- Implementing monomorphic functions is a tiring task
- We want monomorphization for polymorphic functions
- Monomorphization can be considered as specialization with respect to type arguments in Gallina (Type arguments are usual arguments since Gallina is a dependently typed language)
- Partial evaluation solves it (and more)
Partial Evaluation Example

\[
pow(a, b) = \begin{cases} 
1 & b = 0 \\
 a \times pow(a, b - 1) & b > 0 
\end{cases}
\]

\[f(x) = \ldots pow(x, 3)\ldots\]

Specialization of pow with respect to \(b = 3\)

\[
pow3(a) = pow(a, 3) = a \times a \times a \times 1
\]

\[f(x) = \ldots pow3(x)\ldots\]

f(x) would run faster
Fixpoint pow a b :=
  match b with
  | O  ⇒ 1
  | S b' ⇒ a * pow a b'
  end.
Definition pow3 a :=
  Eval cbv beta iota delta [pow] in pow a 3.
(* Same as Definition pow3 a := a * (a * (a * 1)). *)

Definition c := Eval cbv beta iota delta [pow] in t.
defines c with t reduced with beta and iota reductions, and
delta (unfolding) pow using call-by-value (cbv) strategy.

The reductions eliminate static computation (recursion and
match-expression) well

Problem 1: The reductions can duplicate computation

Problem 2: No automatic mechanism to replace call sites
Problem 1: Computation Duplication

- The reductions may duplicate computation:

```coq
Definition pow_2x_3 x :=
    Eval cbv beta iota delta [pow] in pow (x + x) 3.
(* Same as Definition pow_2x_3 x :=
    (x + x) * ((x + x) * ((x + x) * 1)). *)
```

The adding function is invoked only once in `pow (x + x) 3` but 3 times in `pow_2x_3` in the strict evaluation.

- It’s because beta reduction `((\lambda x : T. t) u \triangleright t{\{x/u\}})` can copy the argument `u` of the application.

- We don’t want to duplicate computation since it can make program much slower

  `t{\{x/u\}}` means a term in which `x` in term `t` is replaced by `u`. [Coq reference manual]
Problem 2: Call Site Replacement

- Coq has no feature to replace functions already defined

```coq
Fixpoint pow a b :=
  match b with
  | O ⇒ 1
  | S b' ⇒ a * pow a b'
  end.

Definition f x := ... pow x 3 ...

Definition pow3 a :=
  Eval cbv beta iota delta [pow] in pow a 3.
```

- We cannot redefine $f$ in Coq
- The extraction plugin cannot generate the code of $f$ to use $\text{pow3}$
1. A user defines a function `pow` and `f`  
2. A user specifies the second argument of `pow` is static  
3. Codegen transforms `f`  
   ▶ Codegen finds that `pow` is called with the second argument of 3  
   ▶ Codegen defines `p_pow3`  
     
     \textit{Definition} \quad p\_pow3 a \ := \ pow a \ 3.  
   ▶ Codegen defines `s_f` using `p_pow3`  
4. Codegen transforms `p_pow3`  
   ▶ Codegen defines `s_pow3`  
     
     \textit{Definition} \quad s\_pow3 a \ := \ldots  
   ▶ Codegen verifies `p_pow3 = s_pow3`  
5. Codegen generates C function `pow3` from `s_pow3`  
6. Codegen generates C function `f` from `s_f`  
   The invocation of `p_pow3` is translated to the invocation of `pow3`  
   (Problem 2, call site replacement, is solved)
Specialization of pow

```coq
Fixpoint pow (a b : nat) : nat :=
  match b with
  | O ⇒ 1
  | S b' ⇒ a * pow a b'
  end.

↓ specialize with respect to b = 3

Definition p_pow3 (a : nat) : nat := pow a 3.

Definition s_pow3 (v1_a : nat) : nat :=
  let v2_n := O in
  let v3_n := S v2_n in
  let v4_n := Nat.mul v1_a v3_n in
  let v5_n := Nat.mul v1_a v4_n in
  Nat.mul v1_a v5_n.

```
Convertible Transformations

We define Gallina to Gallina transformation as several steps

1. Inlining
2. V-normalization: Make application arguments and match item variables
3. S-normalization: Simplification
4. Replace call sites with specialized functions
5. Unused let-in Deletion
6. Argument completion to avoid partial application
7. C Variable Allocation

These steps transform a term convertible for verification with **reflexivity**

We explain V-normalization and S-normalization
Gallina Term

\[ t, u = x \quad \text{variable} \]
\[ | \ c \quad \text{constant} \]
\[ | \ C \quad \text{constructor} \]
\[ | \ T \quad \text{type} \]
\[ | \ \lambda x : T. \ t \quad \text{abstraction} \]
\[ | \ t \ u \quad \text{application} \]
\[ | \ \text{let} \ x := t : T \ \text{in} \ u \quad \text{let-in} \]
\[ | \ \text{match} \ t_0 \ \text{with} \ (C_i \Rightarrow t_i)_{i=1}^{h} \ \text{end} \quad \text{conditional} \]
\[ | \ \text{fix} \ (f_i/k_i : T_i := t_i)_{i=1}^{h} \ \text{for} \ f_j \quad \text{fixpoint} \]

Note: We omit details of types.
Actual Gallina syntax permits any term as a type because it is dependently typed.

We ignore Var, Meta, Evar because they are not used in complete program. Int and Float are considered as constants. Prod, Ind and Sort are considered as types. Cast is ignored because it can be eliminated immediately. CoFix is ignored because lazy-evaluation is not suitable to C. Proj is ignored because it is similar to \text{match}. 
Difference with Actual Gallina Term

Our Gallina syntax is more concise than actual Gallina:

- \( \lambda x : T. t \) means \( \text{fun} (x : T) \Rightarrow t \)
- \( \text{let } x := t : T \text{ in } u \) means \( \text{let } x : T := t \text{ in } u \)
- \( \text{fix } (f_i/k_i : T_i := \lambda x_{i1} : T_{i1} \cdots t_i)_{i=1 \ldots h} \text{ for } f_j \) means
  \[ \text{fix } f_1 (x_{11} : T_{11}) \cdots \{ \text{struct } x_{1k_1} \} := t_1 \]
  with ...
  with \( f_h (x_{h1} : T_{h1}) \cdots \{ \text{struct } x_{hk_h} \} := t_h \text{ for } f_j \)
- \( \text{match } t_0 \text{ with } (C_i \Rightarrow \lambda x_{i1} \cdots t_i)_{i=1 \ldots h} \text{ end } \) means
  \( \text{match } t_0 \text{ with } \)
  \( | C_1 x_{11} \cdots \Rightarrow t_1 \)
  \( | \ldots \)
  \( | C_h x_{h1} \cdots \Rightarrow t_h \)
  \( \text{end} \)

We ignore \texttt{as-in-return} clauses because they are not used in reductions
Conversion Rules

reflexivity tactic checks two terms are confluent by these rules

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

delta-local: \[ (x := t) \in \Gamma \]
\[ E[\Gamma] \vdash x \triangleright t \]

delta-global: \[ (c := t) \in E \]
\[ E[\Gamma] \vdash c \triangleright t \]

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

\[ E[\Gamma] \vdash C_\ j \ u_1 \ldots u_{p+m} : T \]

iota-match: \( p \) is the number of parameters of the inductive type \( T \)
\[ E[\Gamma] \vdash \text{match (} C_\ j \ u_1 \ldots u_{p+m} \text{) with } (C_i \Rightarrow t_i)_{i=1\ldots h} \text{ end} \]
\[ \triangleright t_j \ u_{p+1} \ldots u_{p+m} \]

\[ u_{kj} = C \ u'_1 \ldots u'_m \]

iota-fix: \[ E[\Gamma] \vdash (\text{fix (} f_i/k_i := t_i)i=1\ldots h \text{ for } f_j) \ u_1 \ldots u_{kj} \]
\[ \triangleright t_j\{f_k/\text{fix (} f_i/k_i := t_i)i=1\ldots h \text{ for } f_k\}_{k=1\ldots h} \ u_1 \ldots u_{kj} \]

eta expansion:
\[ E[\Gamma] \vdash t : \forall x : \ T. \ U \]
\[ E[\Gamma] \vdash t \triangleright \lambda x : \ T. \ (t \ x) \]

(\( t\{x/u\} \) means a term in which \( x \) in term \( t \) is replaced by \( u \). [Coq reference manual])
Evaluation Strategy for Codegen

- The six reduction rules of the conversion rules (beta, delta-local, delta-global, zeta, iota-match, and iota-fix) defines the execution of Gallina.
- Gallina itself can use any evaluation strategy.
- We use strict evaluation strategy as C: Application arguments are evaluated before the function call.
- A partial application does not call the function because there is no partial application in C. (Partial application will generate a closure when we support closures in future.)
Computation Size

- We do not want to transform functions slower
- We define “computation size” as the number of `match`-expression evaluated in run time
  - computation size is a rough approximation of running time
  - it includes loop count because Gallina recursion needs `match`-expression to obtain a subterm of a decreasing argument
- Our transformations does not increase computation size

Note: Computation size does not mean code size
V-Normal Form

We use V-normal form for our transformations

V-normal form restricts Gallina terms that
(1) application arguments and (2) match items to variables

\[ t = x \mid c \mid C \mid T \mid \lambda x : T. \ t \mid \text{let } x := t : T \ \text{in} \ u \]

\[ \mid \text{fix } (f_i/k_i : T_i := t_i)_{i=1}^{h} \ \text{for } f_j \]

\[ \mid t \ x \]

\[ \mid \text{match } x \ \text{with } (C_i \Rightarrow t_i)_{i=1}^{h} \ \text{end} \]

V-normalization transforms Gallina to V-normal form:

- ▶ \[ t_0 \ x_1 \ldots x_{i-1} \ t_i \ t_{i+1} \ldots t_n \]
  - ▶ \text{let } x_i := t_i \ \text{in } t_0 \ x_1 \ldots x_{i-1} \ x_i \ t_{i+1} \ldots t_n

- ▶ \text{match } t_0 \ \text{with } (C_i \Rightarrow t_i)_{i=1}^{h} \ \text{end}
  - ▶ \text{let } x_0 := t_0 \ \text{in match } x_0 \ \text{with } (C_i \Rightarrow t_i)_{i=1}^{h} \ \text{end}

Note: V-normal form is similar to A-normal form [Flanagan1993] but let-binding (\text{t of let } x:=t:T \ \text{in } u) and function position of application (\text{t of } t \ x) can be any V-normal term
S-Reductions: Reduction Rules without Computation Duplication

We define reduction rules similar to the conversion rules but without computation duplication

- beta-var
- delta-var
- delta-fun
- zeta-flat
- zeta-app
- zeta-del
- iota-match-var
- iota-fix-var
Beta May Duplicate Computation

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

Problem: \((\lambda x. \; x + x) \; (x \times x) \triangleright (x \times x) + (x \times x)\)

Solution: V-normal form restrict arguments as variables

Copying variables does not cause computation duplication because evaluation of a variable does not contain evaluation of match
Beta May Expose Computation in Partial Application

Problem: \((\lambda x. \text{match } x \text{ with } \text{tt } \Rightarrow \lambda y. \ t \ \text{end}) \ z \ \triangleright \ \text{match } z \ \text{with } \text{tt } \Rightarrow \lambda y. \ t \ \text{end}\)

- The evaluation of the former has no evaluation of \text{match} (It generates a closure because it is a partial application)
- The evaluation of the latter does cause an evaluation of \text{match}
- So computation size increases

Solution: We apply beta reduction if one of the following is satisfied

- it is not a partial application i.e. the result is an inductive type (full application evaluates function body anyway)
- the abstraction body is an abstraction or fixpoint (evaluation of abstraction and fixpoint is closure generation thus it has no evaluation of \text{match})

Note: the second condition is added after the camera-ready
Beta-Var Reduction

\[
\begin{align*}
0 < n & \quad E[\Gamma] \vdash (\lambda x. \ t) \ y_1 \ldots y_n : T \\
(T \text{ is an inductive type}) \text{ or } (t \text{ is an abstraction or fixpoint}) & \quad E[\Gamma] \vdash (\lambda x. \ t) \ y_1 \ldots y_n \triangleright t\{x/y_1\} \ y_2 \ldots y_n
\end{align*}
\]

- Since this rule is a restricted beta reduction, convertibility is preserved
Zeta May Duplicate Computation

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

Example: \( \text{let } x := y \times y \text{ in } x + x \triangleright (y \times y) + (y \times y) \)

Solution: We apply zeta only for moving or removing an expression ("moving" is combination of zeta reduction and zeta expansion)

\[ \text{zeta-flat: } E[\Gamma] \vdash \text{let } y := (\text{let } x := t_1 \text{ in } t_2) \text{ in } t_0 \]
\[ \triangleright \text{let } x := t_1 \text{ in } (\text{let } y := t_2 \text{ in } t_0) \]

\[ \text{zeta-app: } E[\Gamma] \vdash (\text{let } x_0 := t \text{ in } u)\ x_1 \ldots x_n \]
\[ \triangleright \text{let } x_0 := t \text{ in } (u\ x_1 \ldots x_n) \]

\[ \text{zeta-del: } x \text{ does not occur in } u \]
\[ E[\Gamma] \vdash \text{let } x := t \text{ in } u \triangleright u \]
Delta-Local May Duplicate Computation and May Break V-Normal Form

\[
\begin{align*}
\text{delta-local:} & \quad \frac{(x := t) \in \Gamma}{E[\Gamma] \vdash x \triangleright t}
\end{align*}
\]

$\Gamma$ is a local context

It contains $(x := t)$ if $x$ is occurs in $u$ of $\text{let } x := t \text{ in } u$

Example: $\text{let } x := y \ast y \text{ in } x + x$

$\triangleright \text{ let } x := y \ast y \text{ in } (y \ast y) + x$

$\triangleright \text{ let } x := y \ast y \text{ in } (y \ast y) + (y \ast y)$

Solution: We apply delta-local reduction if one of the following is satisfied

- $t$ is a variable
- Evaluation of $t$ has no computation and $x$ occur in a function position of application
Delta-Var and Delta-Fun Reduction

- \( t \) is a variable:
  - Since an evaluation of a variable has no computation, copying it does not increase computation size
  - Replacing a variable with a variable does not break V-normal form

\[
\text{delta-var: } \quad (x := y) \in \Gamma \\
E[\Gamma] \vdash x \triangleright y
\]

- Evaluation of \( t \) has no computation and \( x \) occur in a function position of application:
  - Since an evaluation of \( t \) has no computation, copying it does not increase computation size
  - Function position is not restricted by V-normal form

\[
0 \leq p \quad 0 < q \quad (f := t \ x_1 \ldots \ x_p) \in \Gamma \\
E[\Gamma] \vdash f \ y_1 \ldots y_q \triangleright t \ x_1 \ldots x_p \ y_1 \ldots y_q
\]
Iota-Match Conflicts with V-Normal Form

\[ E[\Gamma] \vdash C_j \ u_1 \ldots \ u_{p+m} : T \]

\[ \text{iota-match:} \quad \frac{p \text{ is the number of parameters of the inductive type } T}{E[\Gamma] \vdash \text{match } (C_j \ u_1 \ldots \ u_{p+m}) \text{ with } (C_i \Rightarrow t_i)_{i=1\ldots h} \text{ end} \]

\[ \triangleright \ t_j \ u_{p+1} \ldots \ u_{p+m} \]

Problems:

- match item must be a variable in V-normal form
- \( u_{p+1} \ldots u_{p+m} \) may have computation

Solutions:

- We examine the local context for the match item
- The constructor application arguments must be variables

\[ (x := C_j \ y_1 \ldots y_{p+m} : T) \in \Gamma \]

\[ \text{iota-match-var:} \quad \frac{p \text{ is the number of parameters of the inductive type } T}{E[\Gamma] \vdash \text{match } x \text{ with } (C_i \Rightarrow t_i)_{i=1\ldots h} \text{ end} \]

\[ \triangleright \ t_j \ y_{p+1} \ldots y_{p+m} \]
Iota-Fix Conflicts with V-Normal Form and May Break V-Normal Form

\[ u_{kj} = C \ u'_1 \ldots u'_m \]

\[ E[\Gamma] \vdash (\text{fix} (f_i/k_i := t_i)_{i=1 \ldots h} \text{ for } f_j) \ u_1 \ldots u_{kj} \]

\[ \triangleright \ t_j\{f_k/\text{fix} (f_i/k_i := t_i)_{i=1 \ldots h} \text{ for } f_k\}_{k=1 \ldots h} \ u_1 \ldots u_{kj} \]

Problems:

- \text{iota-fix needs the decreasing argument constructor form but it is not possible in V-normal form}
- \text{iota-fix replaces } f_k \text{ with fixpoints which may break V-normal form}

Solutions:

- We examine the local context for the decreasing argument
- We introduce let-in expressions for the fixpoints
- Also, we prohibit partial application (same as beta-var)
Iota-Fix-Var Reduction

\( (x_{k_j} := C \ y_1 \ldots y_m) \in \Gamma \quad f'_1 \ldots f'_h \) are fresh variables

\[
E[\Gamma] \vdash (\text{fix} \ (f_i/k_i := t_i)_{i=1}^h \text{ for } f_j) \ x_1 \ldots x_n : T
\]

\( T \) is an inductive type

\[
E[\Gamma] \vdash (\text{fix} \ (f_i/k_i := t_i)_{i=1}^h \text{ for } f_j) \ x_1 \ldots x_n \triangleright
\]

let \( f'_1 := \text{fix} \ (f_i/k_i := t_i)_{i=1}^h \text{ for } f_1 \text{ in } \ldots \)

let \( f'_h := \text{fix} \ (f_i/k_i := t_i)_{i=1}^h \text{ for } f_h \text{ in } \)

\( t_j\{f_k/f'_k\}_{k=1}^h \ x_1 \ldots x_n \)
Summary

- Codegen implements partial evaluation using Gallina to Gallina transformation
- The partial evaluation does not duplicate computation
- This transformation can be verified easily
- The partial evaluation also be used for monomorphization and dependent type elimination

Future work:
- Support downward funarg (restricted closure)
- Support proof elimination
Extra Slides
The partial evaluation can cause exponential code bloat. Static computation of $2^n$ causes exponential code bloat because $\text{nat}$ is Peano’s naturals. This is unavoidable as far as we provide general partial evaluation.

We can avoid exponential code bloat by sacrificing general partial evaluation: disabling delta-fun and iota-fix-var. In this case, monomorphization is still possible (because it does not need them).
Monomorphization of List.rev

List.rev is defined as follows:
(The type parameter $A$ is a usual argument because Gallina is a dependently-typed language)

**Definition** \( \text{rev} := \text{fun (A : Type)} \Rightarrow \)

\[
\text{fix rev (l : list A) : list A :=}
\]

\[
\text{match l with}
\]

\[
\mid \text{nil} \Rightarrow \text{nil}
\]

\[
\mid x :: l' \Rightarrow \text{rev l'} ++ x :: \text{nil}
\]

\[
\text{end.}
\]

We want a monomorphic version of List.rev for nat:

**Definition** \( \text{rev Nat} := \)

\[
\text{fix rev (l : list nat) : list nat :=}
\]

\[
\text{match l with}
\]

\[
\mid \text{nil} \Rightarrow \text{nil}
\]

\[
\mid x :: l' \Rightarrow \text{rev l'} ++ x :: \text{nil}
\]

\[
\text{end.}
\]
Monomorphization is Beta-Reduction

Monomorphization can be considered as beta reduction:

\[ \text{rev nat} \]
\[ = (\text{fun } (A : \text{Type}) \Rightarrow \text{fix } \text{rev} \ldots) \text{nat} \quad \text{(delta-global)} \]
\[ = (\text{fix } \text{rev} (l : \text{list } A) : \text{list } A := \ldots)(A/\text{nat}) \quad \text{(beta)} \]
\[ = \text{fix } \text{rev} (l : \text{list } \text{nat}) : \text{list } \text{nat} := \ldots \quad \text{(substitution)} \]
\[ = \text{rev} \_\text{nat} \]
When the partial evaluation compute types statically, we can eliminate dependent types

```
Fixpoint sprintf_type (fmt : string) : Type :=
  match fmt with
  | EmptyString ⇒ buffer
  | String "\%\"%char (String "d\"%char rest) ⇒ nat → sprintf_type rest
  | String "\%\"%char (String "b\"%char rest) ⇒ bool → sprintf_type rest
  | String "\%\"%char (String "s\"%char rest) ⇒ string → sprintf_type rest
  | String "\%\"%char EmptyString ⇒ buffer
  | String _ rest ⇒ sprintf_type rest end.

Fixpoint sprintf (buf : buffer) (fmt : string) : sprintf_type fmt :=
  match fmt return sprintf_type fmt with
  | EmptyString ⇒ buf
  | String "\%\"%char (String "d\"%char rest) ⇒ fun (n : nat) ⇒ sprintf (buf_addnat buf n) rest
  | String "\%\"%char (String "b\"%char rest) ⇒ fun (b : bool) ⇒ sprintf (buf_addbool buf b) rest
  | String "\%\"%char (String "s\"%char rest) ⇒ fun (s : string) ⇒ sprintf (buf_addstrbuf s) rest
  | String "\%\"%char (String ch rest) ⇒ sprintf (buf_addch (buf_addch buf "\") ch) rest
  | String "\%\"%char EmptyString ⇒ buf_addch buf "\"%"%char
  | String ch rest ⇒ sprintf (buf_addch buf ch) rest end.
```

Compute sprintf (Buf "") "\%d + \%d = \%d" 3 4 7.
(* = Buf "3 + 4 = 7" *)
Dependent Type Elimination

`sprintf` specialized with respect to the format string "\"x=\%d\"\".

**Definition** `s_sprintf_x_eq_nat v1_buf v2_n :=

```ocaml
let v3_b := false in let v4_b := false in let v5_b := false in let v6_b := true in
let v7_b := true in let v8_b := true in let v9_b := true in let v10_b := false in
let v11_a := Ascii v3_b v4_b v5_b v6_b v7_b v8_b v9_b v10_b in
let v12_b := true in let v13_b := false in let v14_b := true in let v15_b := true in
let v16_b := true in let v17_b := true in let v18_b := false in let v19_b := false in
let v20_a := Ascii v12_b v13_b v14_b v15_b v16_b v17_b v18_b v19_b in
let v21_b := buf_addch v1_buf v11_a in let v22_b := buf_addch v21_b v20_a in
let v23_b := buf_addnat v22_b v2_n in v23_b
```

```c
typedef unsigned char ascii;
#define Ascii(b0,b1,b2,b3,b4,b5,b6,b7) \
((b0) | (b1) << 1 | (b2) << 2 | (b3) << 3 | (b4) << 4 | (b5) << 5 | (b6) << 6 | (b7) << 7)
```

```c
static buffer sprintf_x_eq_nat(buffer v1_buf, nat v2_n) {
    bool v3_b, v4_b, v5_b, v6_b, v7_b, v8_b, v9_b, v10_b; ascii v11_a;
    bool v12_b, v13_b, v14_b, v15_b, v16_b, v17_b, v18_b, v19_b; ascii v20_a;
    buffer v21_b, v22_b, v23_b;
    /* v11_a = 'x'; */
    v3_b = false; v4_b = false; v5_b = false; v6_b = true;
    v7_b = true; v8_b = true; v9_b = true; v10_b = false;
    v11_a = Ascii(v3_b, v4_b, v5_b, v6_b, v7_b, v8_b, v9_b, v10_b);
    /* v20_a = '='; */
    v12_b = true; v13_b = false; v14_b = true; v15_b = true;
    v16_b = true; v17_b = true; v18_b = false; v19_b = false;
    v20_a = Ascii(v12_b, v13_b, v14_b, v15_b, v16_b, v17_b, v18_b, v19_b);
    v21_b = buf_addch(v1_buf, v11_a); v22_b = buf_addch(v21_b, v20_a); v23_b = buf_addnat(v22_b, v2_n);
    return v23_b;
}
```
Cleaner Code Generation for `match`

```plaintext
CodeGen Inductive Match nat ⇒ "" | 0 ⇒ "case 0"
    | S ⇒ "default" "pred".
CodeGen Constant 0 ⇒ "0".
CodeGen Primitive S ⇒ "succ".

static nat pow(nat v1_x, nat v2_y) {
    nat v3_n, v4_z, v5_n;
    switch (v2_y) {
        case 0:
            v3_n = 0;
            return succ(v3_n);
        default:
            v4_z = pred(v2_y);
            v5_n = pow(v1_x, v4_z);
            return mul(v1_x, v5_n);
    }
}
```
A-Normal Form, K-Normal Form, and V-Normal Form

- A-normal form [Flanagan1993] restricts let-binding bind neither let nor match:
  
  ```
  let v := (let ... in t) and let v := match ... end in t
  ```

  Arguments of application and match item must be variables. A-normal form also restricts a function position \( f(f x_1 \ldots x_n) \) as a variable or primitive function.


- V-normal form allows any V-normal term at a function position.

- V-normal form is useful to represent an equivalent of a loop in C as \( (\text{fix} \ldots) x_1 \ldots x_n \).
Limitation of Codegen

It is possible the convertible transformation retains a Gallina term that Codegen cannot generate C functions

▶ Type computation
▶ Closure generation

We have a plan to implement restricted closures (downward funarg), though
Iota-Match Example

\[ E[\Gamma] \vdash C_j \ u_1 \ldots u_{p+m} : T \]

\[ \text{iota-match: } \frac{p \text{ is the number of parameters of the inductive type } T}{E[\Gamma] \vdash \text{match } (C_j \ u_1 \ldots u_{p+m}) \text{ with } (C_i \Rightarrow t_i)_{i=1\ldots h} \text{ end}} \]
\[ \triangleright t_j \ u_{p+1} \ldots u_{p+m} \]

▶ The definition of list

\[ \text{Inductive list (A : Type) : Type :=} \]
\[ | \text{nil : list A} \]
\[ | \text{cons : A } \rightarrow \text{ list A } \rightarrow \text{ list A} \]

▶ list : Type \rightarrow Type

list has one parameter, A (p = 1)

▶ cons : \forall (A : Type), A \rightarrow list A \rightarrow list A

cons has two members (m = 2)
cons has three arguments (p + m = 3)

\[ \text{match } @ \text{cons nat 1 nil with } (\text{nil } \Rightarrow t_1) (\text{cons } \Rightarrow t_2) \text{ end} \]
\[ \triangleright t_2 \ 1 \ \text{nil} \]