Coq to C Translation with Partial Evaluation

Akira Tanaka
tanaka-akira@aist.go.jp
National Institute of Advanced Industrial Science and Technology (AIST)
Tsukuba, Ibaraki, Japan

Abstract

Coq proof assistant can be used to prove various properties of programs written in the Gallina language. It is also possible to translate Gallina programs to OCaml programs. However, OCaml is not suitable for low-level programs. Therefore, we are developing a Coq plugin for Gallina to C translation. This plugin transforms functions written in Gallina into a form as close to C as possible within Gallina. This transformation includes partial evaluation, which improves execution efficiency and eliminates polymorphism and dependent types. We can easily verify in Coq that this transformation does not change the execution result, and thus it is highly reliable. And Gallina functions after this transformation can be easily translated to C.

CCS Concepts: • **Software and its engineering → Compilers**; *Software verification.*

Keywords: Coq, C, Gallina, translator, compiler, verification, partial evaluation, tail recursion

ACM Reference Format:
Akira Tanaka. 2021. Coq to C Translation with Partial Evaluation. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3441296.3441394

1 Introduction

We are developing a Gallina to C translator, Codegen[1] plugin for Coq proof assistant [10], which can generate low-level C programs. The initial target of Codegen is libraries that handle complex data structures such as succinct data structures [8]. To achieve this, we must be able to easily use various low-level features such as 64 bit integers, built-in functions for CPU instructions, hardware specific types such as for SIMD registers, and so on. In other words, the goal is to be able to use features that are available for usual C programs from Gallina programs.

Gallina is a purely functional ML-like language provided by Coq and is used for programs and proofs. However it shares several features with many imperative languages including C.

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

With these features, basic programming is possible. Therefore, it is possible to translate a Gallina subset of these features into a C subset to achieve translation without overheads.

But there are also gaps between Gallina and C.

1. Gallina has type polymorphism, but C does not.
2. Gallina has dependent types, but C does not.
3. Gallina can use any evaluation strategy, but C is strict.
4. Gallina uses curried functions, but C does not.
5. Gallina has first-class functions, but C does not.
6. Gallina's data types (such as Peano's naturals defined as an inductive type) are suitable for proof but its naive implementation is too inefficient. C has efficient data types such as 64 bit integer.
7. All Gallina programs terminate but C does not.
8. Gallina uses only immutable values but C uses mutable values.
9. Gallina does not require to release memory but C requires.
10. Gallina does not have a loop, but C does.

In this paper, we describe how we used partial evaluation to address the first two of these gaps in the translator from Gallina to C. Gallina achieves type polymorphism and dependent types as type computation. We eliminate them by computing them statically.

We also describe several Gallina transformations to make Gallina functions suitable to translate to C. The result of these transformations (including partial evaluation) is easily verified by Coq because they are convertible.

The 3rd and 4th gaps relate evaluation strategy. Codegen generates C functions that implement the strict evaluation of Gallina terms. It does not violate Gallina semantics because Gallina can use any evaluation strategy. We assume that a

[1] https://github.com/akr/codegen

PEPM '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8305-9/21/01…$15.00
https://doi.org/10.1145/3441296.3441394

Gallina function body is evaluated after all arguments have been given, just as in C. This means that the evaluation of a partial application evaluates the function position and the arguments but does not evaluate the body of the function.

Codegen eliminates tail recursion using `goto` to address the last gap. Codegen can generate a C function that contains multiple loops and nested loops from Gallina terms which contain fixpoints at non-tail positions and nested fixpoints. Codegen also supports mutually tail-recursive Gallina functions by generating a single C function that contains multiple function bodies. Although `goto` is not usable for indirect calls, direct calls are enough to represent loops.

Solutions to the other gaps are not the subject of this paper and are summarized in Appendix A.

Section 2 explains that Coq has a feature similar to partial evaluation but it has problems. Section 3 explains the whole structure of our translation. Section 4 explains the transformations in Gallina terms. Section 5 explains C code generation. Section 6 explains several examples. We describe related work in Section 7 and conclude in Section 8.

2 Partial Evaluation with Stock Coq?

Gallina syntax is shown in Figure 1. Gallina has conversion rules shown in Figure 2 to test the equality of terms. If two terms are confluent by the conversion rules, they are said to be convertible.

The six reductions in the conversion rules define the evaluation of Gallina. Since the reductions are strong normalization, any reduction can be applied to any subterm in any order. Coq also has a command to define a reduced term as a new constant.

These features are similar to partial evaluation. However, it has two problems: (1) the reductions can duplicate computations to produce inefficient programs; (2) Coq cannot replace a function and its static arguments with a specialized function.

2.1 Reduction of `pow`

We can define a power function in Coq.

```
Fixpoint pow x y := match y with
                    | 0 ⇒ 1
                    | S y' ⇒ x * pow x y'
                    end.
```

We can define the normal form of `pow x 3` with reductions (beta, iota-match, iota-fix, and delta-global for the constant `pow`) using call-by-value strategy.

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

This definition is the same as follows.

```
Definition pow3 x := x * (x * (x * 1)).
```

The result has no recursive call and conditional. It should run efficiently.

$$
\begin{array}{rr}
t = x & \text{variable} \\
\mid c & \text{constant} \\
\mid C & \text{constructor} \\
\mid T & \text{type} \\
\mid \lambda x{:}T.\ t & \text{abstraction} \\
\mid t\ u & \text{application} \\
\mid \texttt{let}\ x := t : T\ \texttt{in}\ u & \text{let-in} \\
\mid \texttt{match}\ t_0\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end} & \text{conditional} \\
\mid \texttt{fix}\ (f_i/k_i : T_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j & \text{fixpoint}
\end{array}
$$

Note:

- u represents a term as t.
  y, z, and f represent a variable as x.
  U represents a type as T.
- We write $(\cdots((t\ u_1)\ u_2)\cdots\ u_n)$ as $t\ u_1 \ldots u_n$.
- k is an integer.
  k_i for fixpoint specify the decreasing argument for f_i.
- If it is unambiguous, we omit type annotations for the sake of simplicity. We also omit k_i in fixpoints if they are not used.
- We omitted the elimination predicate (`as-in-return` clause of `match`-expression). It is not used in reductions.
- `match`-branches t_i are functions that take the constructor members (constructor arguments without inductive type parameters). This formalism is taken from CIC [11].
- We omitted the detail of the types. Actual Gallina permits any Gallina term which evaluates to a type.

Figure 1. Gallina syntax.

But Coq has no feature to replace `pow` invocations in the environment. We cannot replace `pow z 3` with `pow3 z` in functions already defined. If we have a function `f` that uses `pow z 3` and generate OCaml definition of `f` using the Extraction plugin, the generated function uses `pow`, not `pow3`.

Another problem is that the reduction rules can duplicate computations.

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

This is the same as follows.

```
Definition pow_2x_3 x :=
  (x + x) * ((x + x) * ((x + x) * 1)).
```

This result contains 3 times of `x + x`. Thus, it is not as efficient as we want.

2.2 Verification of `pow3`

We can verify that `pow x 3` and `pow3 x` are equal in Coq.

```
Goal ∀ x, pow x 3 = pow3 x. Proof. reflexivity. Qed.
```

This equality means that evaluation of them produces the same value. The `reflexivity` tactic confirms that `pow x 3` and `pow3 x` are convertible.

$$\text{beta: } E[\Gamma] \vdash ((\lambda x.\ t)\ u) \triangleright t\{x/u\} \qquad \text{delta-local: } \frac{(x := t) \in \Gamma}{E[\Gamma] \vdash x \triangleright t} \qquad \text{delta-global: } \frac{(c := t) \in E}{E[\Gamma] \vdash c \triangleright t}$$

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

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

$$\text{iota-fix: } \frac{u_{k_j} = C\ u'_1 \ldots u'_m}{E[\Gamma] \vdash (\texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j)\ u_1 \ldots u_{k_j} \triangleright t_j\{f_k/\texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_k\}_{k=1\ldots h}\ u_1 \ldots u_{k_j}}$$

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

Note:

- The rules shown here are reductions, except the eta expansion.
- $t\{x/u\}$ means a term in which x in term t is replaced by u. This notation is taken from the Coq reference manual [11].
- Variables cannot conflict because Coq uses de Bruijn's indexes to represent variables.
- E is a global environment which is a list of global assumptions ($c{:}T$), global definitions ($c := t{:}T$), and inductive definitions (Ind $[p]$ $(\Gamma_I := \Gamma_C)$).
- Γ is a local context which is a list of local assumptions ($x{:}T$) and local definitions ($x := t{:}T$). The local assumptions represent variables bounded by outer abstractions and fixpoints. The local definitions represent variables bounded by outer let-in.
- If it is unambiguous, we omit type annotations in these definitions for the sake of simplicity.
- Iota-match reduces `match @cons nat 1 nil with (nil ⇒ t1) (cons ⇒ t2) end` to t_2 `1 nil` because `list` has one parameter ($p = 1$) and `cons` has two members ($m = 2$).

Figure 2. Gallina conversion rules.

3 Translation from Gallina to C

We implement the Gallina to C translation with the following two kinds of translation phases.

- Convertible transformations
- C code generation

The former transforms Gallina functions in a convertible way. The latter translates Gallina functions to C functions.

We implement translation phases in the former as much as possible. This allows us to verify with Coq that these transformations do not change the result of the function.

A user can configure the translation as follows.

- constant, constructor and inductive type name mappings between Gallina and C (Gallina names are used as-is[2] by default)
- zero-argument constructors to translate without parenthesis (Codegen generates parenthesis by default but possible to translate `true` to `true` instead of `true()` and `O` to `0`)
- code snippets to translate `match`-expression to `switch`-statement (they are generated from the inductive type and constructor names by default)
- which the arguments are static or dynamic (only type arguments are static by default)
- functions to inline (no functions to inline by default)
- which types are linear (no linear types by default)

[2]Invalid characters for a C identifier are replaced by _.

4 Convertible Transformations

Codegen defines two constants with the convertible transformations: terms before and after the transformations.

For example, when a user specifies the second argument of `pow` static and Codegen find an invocation of `pow` with the actual argument `3`, Codegen defines two constants[3]:

```
Definition p_pow3 x := pow x 3. (* before trans. *)
Definition s_pow3 x := ... (* after trans. *)
```

The convertible transformations generate the latter (prefixed with `s_`) from the former (prefixed with `p_`). Codegen verifies the convertibility of them when the latter is defined.

Codegen applies seven convertible transformations described in Section 4.2 to Section 4.8.

4.1 Computation Size

We define the convertible transformations that do not duplicate computations. For this purpose, we define "computation size" as an approximation of running time[4].

We define the computation size as the number of `match`-expression evaluated at run time. The size estimates a loop count. Because a loop is represented by recursion in Gallina and recursive functions must use `match`-expression to obtain a subterm of the decreasing argument for a recursive call.

[3]We use longer names in actual implementation.

[4]The computation size is not an approximation of code size. If we want to avoid exponential code bloat, we can disable delta-fun, iota-fix-var, and iota-fix-var'. This makes complex partial evaluation impossible but monomorphization is still possible.

$$\text{zeta-arg: } \frac{E[\Gamma] \vdash t_i : T_i \quad t_0 \text{ is not an application} \quad t_i \text{ is not a variable} \quad x_i \text{ is a fresh variable}}{E[\Gamma] \vdash t_0\ x_1 \ldots x_{i-1}\ t_i\ t_{i+1} \ldots t_n \triangleright \texttt{let}\ x_i := t_i : T_i\ \texttt{in}\ t_0\ x_1 \ldots x_{i-1}\ x_i\ t_{i+1} \ldots t_n}$$

$$\text{zeta-item: } \frac{E[\Gamma] \vdash t_0 : T_0 \quad t_0 \text{ is not a variable} \quad x_0 \text{ is a fresh variable}}{E[\Gamma] \vdash \texttt{match}\ t_0\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end} \triangleright \texttt{let}\ x_0 := t_0 : T_0\ \texttt{in}\ \texttt{match}\ x_0\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end}}$$

Figure 3. V-reductions.

$$\begin{aligned} t = {} & x \mid c \mid C \mid T \mid \lambda x{:}T.\ t \mid \texttt{let}\ x := t : T\ \texttt{in}\ u \\ & \mid \texttt{fix}\ (f_i/k_i : T_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \\ & \mid t\ x && \leftarrow (1) \\ & \mid \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end} && \leftarrow (2) \end{aligned}$$

Figure 4. V-normal form restricts Gallina terms that (1) application arguments and (2) match items to variables.

4.2 Inlining

Codegen inlines user-specified functions using delta-global reduction. (The target of specialization, `pow` in `p_pow3` for example, is inlined automatically.) This phase does not apply further reductions such as beta reduction.

Since delta-global is part of the conversion rules, this transformation is convertible.

We assume functions inlined are defined as terms formed as an abstraction or a fixpoint. In this case, inlining does not change the computation size because the evaluation of them does not cause an evaluation of `match`-expression.

4.3 V-Normalization

Codegen transforms arguments of application and match items as variables as A-normal form [4]. We call the result of this transformation V-normal form. V-normal form permits more expressions at binding expressions in let-in and function positions in applications than A-normal form. Figure 3 shows V-reductions to generate V-normal form shown in Figure 4.

V-reductions introduce let-in expressions. V-reductions produce a convertible term because they are zeta expansion.

V-reductions move an expression from an application argument and a match item. This moves computations but does not change the computation size.

4.4 S-Normalization

We define S-reductions in Figure 5 to simplify V-normal terms. S-reductions are the reduction rules for the partial evaluation without computation duplication.

S-reductions resemble the conversion rules. Especially, they are terminating because iota-fix-var (and iota-fix-var') has the same guard condition of iota-fix.

beta-var. The beta-var reduction is a restricted version of the beta reduction. It applies the beta reduction when a beta redex occurs in the function position of a nested application that returns an inductive value. This avoids to reduce partial applications and follows our strict evaluation strategy that invokes a function with all arguments. The arguments of the beta-var redex are variables because of V-normal form. It does not duplicate computation because the argument does not contain a `match`-expression.

delta-var. The delta-var reduction is a restricted version of the delta-local reduction. It can be applied only when the local definition maps the variable to another variable. This happens when the outer let-in bounds the variable to another variable. Since a variable does not contain a `match`-expression, delta-var reduction does not change the computation size.

delta-fun. The delta-fun reduction is another restricted version of the delta-local reduction. It can be applied only when the variable f occurs in the function position of an application, $f\ y_1 \ldots y_q$ $(0 < q)$, and the bounded expression, $t\ x_1 \ldots x_p$ $(0 \leq p)$, has no computation. The evaluation of the function t has no computation because there is a premise to avoid `match`-expression. The evaluation of the arguments, $x_1 \ldots x_p$, has no computation because they are just variables. $t\ x_1 \ldots x_p$ is a partial application because it is the function of $f\ y_1 \ldots y_q$ and $0 < q$. The evaluation of $t\ x_1 \ldots x_p$ does not evaluate the function body of t because our strict evaluation strategy does not invoke the function of a partial application. Thus, the bounded expression has no computation and the delta-fun reduction does not change the computation size.

zeta-flat. The zeta-flat reduction commutes let-in expression to make any let-in expression does not bind a let-in expression. This reduction can be considered as a combination of zeta reduction to delete the let-in expression for x and zeta expansion to restore it in the different position. The evaluation of the term before and after the reduction is both that evaluation of t_1, t_2, t_0. Thus, it does not change the computation size.

zeta-app. The zeta-app reduction moves a let-in binding from the function position of an application outside of the application. This reduction can be considered as a combination of zeta reduction and zeta expansion such as zeta-flat reduction. This reduction moves t but does not change the computation size.

$$\text{beta-var: } \frac{0 < n \quad E[\Gamma] \vdash (\lambda x.\ t)\ y_1 \ldots y_n : T \quad T \text{ is an inductive type}}{E[\Gamma] \vdash (\lambda x.\ t)\ y_1 \ldots y_n \ \triangleright\ t\{x/y_1\}\ y_2 \ldots y_n} \qquad \text{delta-var: } \frac{(x := y) \in \Gamma}{E[\Gamma] \vdash x \ \triangleright\ y}$$

$$\text{delta-fun: } \frac{0 \le p \quad 0 < q \quad (f := t\ x_1 \ldots x_p) \in \Gamma \quad t \text{ is one of } x, c, C, \lambda x.\ u, \texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j}{E[\Gamma] \vdash f\ y_1 \ldots y_q \ \triangleright\ t\ x_1 \ldots x_p\ y_1 \ldots y_q}$$

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

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

$$\text{iota-match-var: } \frac{(x := C_j\ y_1 \ldots y_{p+m} : T) \in \Gamma \quad p \text{ is the number of parameters of the inductive type } T}{E[\Gamma] \vdash \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end} \ \triangleright\ t_j\ y_{p+1} \ldots y_{p+m}}$$

$$\text{iota-fix-var: } \frac{\begin{array}{c}(x_{k_j} := C\ y_1 \ldots y_m) \in \Gamma \quad f'_1 \ldots f'_h \text{ are fresh variables} \\ E[\Gamma] \vdash (\texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j)\ x_1 \ldots x_n : T \quad T \text{ is an inductive type}\end{array}}{\begin{array}{l}E[\Gamma] \vdash (\texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j)\ x_1 \ldots x_n \ \triangleright \\ \quad \texttt{let}\ f'_1 := \texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_1\ \texttt{in}\ \ldots\ \texttt{let}\ f'_h := \texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_h\ \texttt{in} \\ \quad t_j\{f_k/f'_k\}_{k=1\ldots h}\ x_1 \ldots x_n\end{array}}$$

$$\text{iota-fix-var': } \frac{\begin{array}{c}(x_{k_j} := C\ y_1 \ldots y_m) \in \Gamma \\ (f'_1 := \texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_1) \in \Gamma\ \ldots\ (f'_h := \texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_h) \in \Gamma \\ E[\Gamma] \vdash (\texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j)\ x_1 \ldots x_n : T \quad T \text{ is an inductive type}\end{array}}{E[\Gamma] \vdash (\texttt{fix}\ (f_i/k_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j)\ x_1 \ldots x_n \ \triangleright\ t_j\{f_k/f'_k\}_{k=1\ldots h}\ x_1 \ldots x_n}$$

Figure 5. S-reductions.

iota-match-var. The iota-match-var reduction reduces a `match`-expression to one of its branches if the constructor for the match item is known. Since the match item is always a variable in V-normal form, the premise of this reduction examines the local definition. We require that the local definition contains a constructor application with arguments as variables. This requirement makes it possible to use the constructor members in the reduced expression without duplicating computation. This reduction can be considered as a combination of delta-local reduction to make the match item constructor form and iota-match reduction. It decreases the computation size because it removes the evaluation of the `match`-expression at run time.

iota-fix-var. The iota-fix-var reduction reduces a fixpoint application that returns an inductive value. We require all arguments, not only until the decreasing arguments, to avoid reducing partial applications. This reduction introduces let-in bindings for the fixpoints to preserve V-normal form. Since the decreasing argument is a variable in V-normal form, the premise examines the local definition. Thus, this reduction can be considered as a combination of delta-local reduction to make the decreasing argument constructor form, iota-fix reduction, delta-local expansion to restore the decreasing argument, and zeta expansions to extract the fixpoints substituted by the iota-fix reduction. Since the introduced let-in bindings have no computation (because they are fixpoints without arguments), this reduction does not change the computation size.

iota-fix-var'. The iota-fix-var' reduction is the same as iota-fix-var reduction but reuses already bounded fixpoints. This reduction is optional. This reduction can be considered as a combination of the conversion rules the same as iota-fix-var except that delta-local expansions are used instead of the last zeta expansions. It has the property the same as iota-fix-var reduction except that the size of intermediate terms are smaller.

4.5 Call Site Replacement

Codegen replaces a function and its static arguments with a specialized function. When new values for static arguments are found, a new specialized function (before transformations prefixed by `p_`) is defined in the global environment.

The values of static arguments are obtained by normalizing the actual static arguments (variables) using the reductions in the conversion rules. Codegen requires that these normal forms have no free variables.

This transformation does not start the convertible translations for other functions and does not define simplified functions (prefixed with `s_`) for the other functions. Thus, Codegen can transform one function at a time.

This transformation can be considered as delta-global expansion addition to the reductions for normalizing the static arguments. If a dynamic argument exists before a static argument such as `pow`, beta expansion is also used before the delta-global expansion: when the normal form of `three` is `3`, `pow z three` is transformed to `(fun x ⇒ pow x 3) z` and `(fun x ⇒ pow x 3)` is replaced with the constant `p_pow3`.

$$\text{zeta-del: } \frac{x \text{ does not occur in } u}{E[\Gamma] \vdash \texttt{let}\ x := t\ \texttt{in}\ u \triangleright u}$$

Figure 6. Unused let-in deletion.

Although delta-global expansion introduces new constant in the global environment, convertibility test still works because it can use delta-global reduction.

It does not change the computation size because the function position is a constant and the static arguments are variables and they do not contain `match`-expression.

4.6 Unused let-in Deletion

This transformation deletes unused let-in expressions as shown in Figure 6. It is applied after call site replacement because call site replacement removes variable references such as `three` in the example in Section 4.5.

This reduction is a restricted zeta reduction.

This reduction removes the binding expression t in let-in expression. Therefore, it can decrease the computation size when an evaluation of t evaluates a `match`-expression. But it does not increase the computation size.

4.7 Argument Completion

Codegen applies eta expansion to avoid partial applications because C has no partial application. This makes all C variables[5] explicit in Gallina. It simplifies the arguments generated by eta expansion by moving the arguments into a body of let-in expression and beta-var reduction. Thus, this transformation can be considered as a combination of eta expansion, zeta-app reduction, and beta-var reduction.

It also transforms `match`-branches into nested abstractions corresponding to the members of constructors. This transformation is also eta expansion.

This transformation does not change the computation size because we apply eta expansion only when it does not delay evaluation. Since a function body is evaluated after all arguments are given, we can apply eta expansion for the function body (top-level functions and functions for fixpoints). If a constructor has k members, the corresponding `match`-branch is evaluated with k arguments so we can apply k times of eta expansion for the branch.

The detail of this transformation is shown in Appendix B.

4.8 C Variable Allocation

This transformation allocates C variables for Gallina variables. It generates unique names that are valid in C. It may allocate one C variable for multiple Gallina variables when they are always bound to the same value. For example, `y` is always the same as `z` in `match x with tt ⇒ fun y ⇒ y end z`. We formalized it in Appendix C to generates a variable mapping from Gallina to C.

Codegen implements this allocation by renaming variables in Gallina terms. This "renaming" does not change de Bruijn's indexes but changes variable names. Therefore, we can inspect the renaming by printing the result of the convertible transformations. This is the reason that we define this as one of the convertible transformations. This transformation does not affect the evaluation of Gallina terms.

4.9 Well-Typedness of the Transformations

The transformations produce convertible terms because they are combinations of the conversion rules. However, several transformations use the reverse of the conversion rules. V-normalization uses zeta expansion. S-normalization uses zeta expansion and delta-local expansion. Call site replacement uses delta-global expansion and beta expansion. Argument completion uses zeta expansion. Since the conversion rules restore the original term, they do not break convertibility. Furthermore, zeta expansion, delta-local expansion, and delta-global expansion produces a well-typed term from a well-typed term. They replace an expression with a new variable or constant which is convertible to the original expression. Since the Gallina typing rules do not distinguish convertible terms, they produce a well-typed term. The beta expansion can produce an ill-typed term for a dependently typed term [11] but we focus non-dependent terms after S-normalization.

5 C Code Generation

Codegen generates C code from the result of the convertible transformations. However Codegen gives up code generation if the result still contains type computations[6] or closure generation. We describe the subset of Gallina that can be translated to C in Appendix D[7].

This translation is easy because:

- No complex subexpressions because of V-normal form (like A-normal form)
- No type computations and closure generation
- No partial applications and C variables are already allocated

The convertible transformations may retain lambda abstractions not only at the top-level of a function. Codegen can generate C code for them as far as arguments are given lexically outside of them. For example, Codegen can generate C code for `match x with tt ⇒ fun y ⇒ y end z`. Since

[5] The temporary variables for parallel assignments are the exception.

[6] The type computation includes type polymorphism, polymorphic recursion, and dependent types

[7] This is the intermediate representation for Codegen, not the definition of the source Gallina terms for Codegen. We do not define the source Gallina terms for Codegen because the convertible transformations may remove arbitrary complex terms. For example, Codegen can handle `match` u `with true ⇒ 0 | false ⇒` t `end` for any t, as far as u is reduced to `true`. It is not possible to determine that without actually reducing u.

Codegen does not support closures yet, lambda abstractions that have no such actual arguments are not supported.

Codegen needs one more complicated task, tail recursion elimination. Codegen guarantees it because Gallina has no loops and we need a reliable way to describe loops that do not consume the stack. Since C compilers do not guarantee tail recursion elimination, Codegen must do it.

We show the code generation in Figure 7 for non-tail position and Figure 8 for tail position. Figure 9 defines auxiliary functions for them. Top-level function generation is described in Appendix E.

Codegen translates a recursive call in a tail position using `goto`. This is the traditional translation method and allows us to generate a loop in a tail position.

We also generate a loop in a non-tail position. We want this feature to generate loops for inlined small higher-order functions such as `fold_left`. Such functions are used not only in tail positions. Thus, we generate a loop from a fixpoint application in a non-tail position. However, not all of them can be a loop.

Whether a fixpoint application in a non-tail position can be a loop is determined by TR in Figure 10. TR determines that a fixpoint can be a loop if all recursive calls to the functions bounded by the fixpoint are translated to `goto`. This is equivalent to a C compiler determining that a recursive function cannot be inlined, but a non-recursive function can be inlined, even if it contains `goto`.

Codegen can generate a C function that contains multiple Gallina functions. It is used to generate mutually recursive functions with `goto`. See Appendix E for details.

6 Translation Examples

6.1 Convertible Transformations without Duplicating Computations

The convertible transformations do not duplicate computations. Therefore, we can specialize `pow_2x_3` in Section 2.1 properly. The convertible transformations in Codegen define `s_pow_2x_3` as follows:

```
Definition s_pow_2x_3 v1_x :=
  let v2_n := v1_x + v1_x in
  let v3_n := 0 in
  let v4_n := S v3_n in
  let v5_n := v2_n * v4_n in
  let v6_n := v2_n * v5_n in
  v2_n * v6_n.
```

The addition occurs only once and is not duplicated.

6.2 Specialized Function Application Generation

Codegen generates a specialized function that invokes other specialized functions. Assume the following definition and the second argument of `pow` is static.

```
Definition f x y := pow x 3 + pow y 3.
```

The convertible transformation for `f` generates `p_pow3` and `s_f` as follows. (`p_f` is not generated because `f` has no static arguments.)

```
Definition p_pow3 x := pow x 3.
Definition s_f v1_x v2_y :=
  let v3_n := p_pow3 v1_x in
  let v4_n := p_pow3 v2_y in
  v3_n + v4_n.
```

`s_pow3` can be generated by applying the convertible transformations to `p_pow3`.

```
Definition s_pow3 v1_x :=
  let v2_n := 0 in
  let v3_n := S v2_n in
  let v4_n := v1_x * v3_n in
  let v5_n := v1_x * v4_n in
  v1_x * v5_n
```

Codegen generates C functions for `pow3` and `f` from `s_pow3` and `s_f` as in Figure 11. This result shows that Codegen generates a specialized function that invokes another specialized function: the invocation of `pow` in `f` is translated to the invocation of `pow3`. This is different from Coq reductions with the Extraction plugin described in Section 2.1.

These functions depend on a C implementation of the inductive type and primitive functions. If a user wants to implement `nat` by `uint64_t`, the implementation in Figure 12 is possible[8].

6.3 Monomorphization of `List.rev`

The convertible transformations are usable for monomorphization (type polymorphism elimination). Since Gallina is a dependently typed language, types are first-class values and they can be passed as arguments. If a type is passed as an argument and it is used for the type of a later argument, the function is polymorphic. Thus, a beta reduction can eliminate a type argument. (If a fixpoint has a type argument, we cannot monomorphize it. This means Codegen cannot eliminate polymorphic recursion.)

Coq has a list reversal function `rev` in the standard library. It is defined as follows[10].

```
Definition rev := fun (A : Type) ⇒
                  fix rev (l : list A) : list A :=
                    match l with
                    | nil ⇒ nil
                    | x :: l' ⇒ rev l' ++ x :: nil
                    end.
```

[8] `nat` and `uint64_t` are not isomorphic because `nat` has infinitely many elements and `uint64_t` has a finite number of elements. We provide another Coq plugin, monadification[9] [9] to translate a function to use option monad. We can use it to verify that this mismatch does not cause problems when overflow does not occur.

[9] https://github.com/akr/monadification

[10] The standard library uses `Section` to define this term.

$\mathrm{A}_K[\![t\ /\ x_1 \ldots x_n]\!]$ generates C code for $t\ x_1 \ldots x_n$ in a non-tail position. The result expression is passed to K. $K(e)$ = "$v = e$;" in simple situations.

$\mathrm{A}_K[\![x\ /\ ]\!] = K(\text{“}x\text{”})$

$\mathrm{A}_K[\![x\ /\ x_1 \ldots x_n]\!] = \text{“ passign(fvars}'[\![x]\!], x_1 \ldots x_n)$ `goto entry_`x`;"` $\quad n > 0 \wedge x$ is bounded by a fixpoint $\wedge\ x \in TR$

$\mathrm{A}_K[\![x\ /\ x_1 \ldots x_n]\!] = K(\text{“}x(y_1, \ldots, y_o, x_1, \ldots, x_n)\text{”})$ $\quad n > 0 \wedge x$ is bounded by a fixpoint $\wedge\ x \notin TR$

$\mathrm{A}_K[\![c\ /\ x_1 \ldots x_n]\!] = K(\text{“}c(x_1, \ldots, x_n)\text{”})$ $\quad n \geq 0$

$\mathrm{A}_K[\![C\ /\ x_1 \ldots x_n]\!] = K(\text{“}C(x_1, \ldots, x_n)\text{”})$ $\quad n \geq 0$

$\mathrm{A}_K[\![t\ x_0\ /\ x_1 \ldots x_n]\!] = \mathrm{A}_K[\![t\ /\ x_0\ x_1 \ldots x_n]\!]$

$\mathrm{A}_K[\![$`let` $x :=\ t_1$ `in` $t_2\ /\ x_1 \ldots x_n]\!] = \text{“ A}_{K'}[\![t_1\ /\ ]\!]\ \mathrm{A}_K[\![t_2\ /\ x_1 \ldots x_n]\!]\text{”}$ $\quad$ where $\quad K'(e) = \text{“}x = e\text{;”}$

$\mathrm{A}_K[\![\lambda x.\ t\ /\ x_1\ x_2 \ldots x_n]\!] = \mathrm{A}_K[\![t\ /\ x_2 \ldots x_n]\!]$ $\quad$ (x and x_1 are mapped to the same C variable)

$\mathrm{A}_K[\![$`match` x `with` $(C_i \Rightarrow \lambda y_{i1} \ldots \lambda y_{i\,\mathrm{NM}_{C_i}}.\ t_i)_{i=1\ldots h}$ `end` $/\ x_1 \ldots x_n]\!] =$ $\quad$ where $\quad x : T$

"`switch (`$swfunc_T(x)$`) {`
...
$caselabel_{C_i}$:
$y_{i1} = get\_member_{C_i 1}(x);\ \ldots;\ y_{i\,\mathrm{NM}_{C_i}} = get\_member_{C_i\,\mathrm{NM}_{C_i}}(x);$
$\mathrm{A}_K[\![t_i\ /\ x_1 \ldots x_n]\!]$
`break;`
...
`}`"

$\mathrm{A}_K[\![$`fix` $(f_i := t_i)_{i=1\ldots h}$ `for` $f_j\ /\ x_1 \ldots x_n]\!] =$ $\quad f_j \in TR$

" passign(fvars$[\![t_j]\!], x_1 \ldots x_n)$ $\quad$ where $\quad K'(e) =$

$\mathrm{GENBODY}^{\mathrm{AT}}_{K'}[\![$`fix` $(f_i := t_i)_{i=1\ldots h}$ `for` $f_j]\!]$ $\quad \begin{cases} K(e) & K(e) \text{ contains } \texttt{goto} \\ \text{“}K(e)\ \texttt{goto exit\_}f_j\texttt{;”} & \text{otherwise} \end{cases}$

`exit_`f_j:"

$\mathrm{A}_K[\![$`fix` $(f_i := t_i)_{i=1\ldots h}$ `for` $f_j\ /\ x_1 \ldots x_n]\!] =$ $\quad f_j \notin TR$

"$K(f_j(y_1, \ldots, y_o, x_1 \ldots, x_n))$
`goto skip_`f_j`;`
$\mathrm{GENBODY}^{\mathrm{AN}}[\![$`fix` $(f_i := t_i)_{i=1\ldots h}$ `for` $f_j]\!]$
`skip_`f_j:"

Note:

- "..." means a string. A string can contain characters in typewriter font and expressions starting in italic or roman font. The former is preserved as-is. The latter embeds the value of the expression (with name translation from Gallina to C).
- Gallina types, constants, and constructors have corresponding (user-configurable) C names and they are implicitly translated. Gallina variables are translated by the mapping defined in Section 4.8.
- $TR = \mathrm{TR}[\![t]\!]_n$ where the translating function is defined as `Definition` $c := t$ and t is an n-arguments function.
- NM_C is the number of the members of the constructor C (the number of arguments without the parameters for the inductive type): $\mathrm{NM}_C = m$ if $C : T_1 \to \cdots \to T_p \to T_{p+1} \to \cdots \to T_{p+m} \to T_0$ and T_0 is an inductive type which has p parameters.
- $swfunc_T$, $caselabel_{C_i}$, and $get\_member_{C_i j}$ are defined by a user to translate `match`-expressions for the inductive type T.
- passign$(y_1 \ldots y_n, x_1 \ldots x_n)$ is a parallel assignment. It is translated to a sequence of assignments to assign $x_1 \ldots x_n$ into $y_1 \ldots y_n$. It may require temporary variables.
- $y_1, \ldots, y_o$ are the outer variables of the fixpoint.
- We do not define $\mathrm{A}_K[\![\lambda x.\ t\ /\ ]\!]$ because we do not support closures yet.
- Actual Codegen generates $\mathrm{GENBODY}^{\mathrm{AN}}[\![]\!]$ in a different position to avoid the label `skip_`f_j and `goto skip_`f_j`;`.

Figure 7. Translation to C for a non-tail position.

$\mathrm{B}_K[\![t\ /\ x_1 \ldots x_n]\!]$ generates C code for $t\ x_1 \ldots x_n$ in a tail position. The result expression is passed to K.
$K(e)$ = "`return` e;" in simple situations.

$$\mathrm{B}_K[\![x\ /\ ]\!] = K(\text{“}x\text{”})$$
$$\mathrm{B}_K[\![x\ /\ x_1 \ldots x_n]\!] = \text{“ passign(fvars}'[\![x]\!], x_1 \ldots x_n) \texttt{ goto entry\_}x\text{;”} \qquad n > 0 \wedge x \text{ is bounded by a fixpoint}$$
$$\mathrm{B}_K[\![c\ /\ x_1 \ldots x_n]\!] = K(\text{“}c(x_1, \ldots, x_n)\text{”}) \qquad n \geq 0$$
$$\mathrm{B}_K[\![C\ /\ x_1 \ldots x_n]\!] = K(\text{“}C(x_1, \ldots, x_n)\text{”}) \qquad n \geq 0$$
$$\mathrm{B}_K[\![t\ x_0\ /\ x_1 \ldots x_n]\!] = \mathrm{B}_K[\![t\ /\ x_0\ x_1 \ldots x_n]\!]$$
$$\mathrm{B}_K[\![\texttt{let } x := t_1 \texttt{ in } t_2\ /\ x_1 \ldots x_n]\!] = \text{“ A}_{K'}[\![t_1\ /\ ]\!]\ \mathrm{B}_K[\![t_2\ /\ x_1 \ldots x_n]\!]\text{”} \qquad \text{where} \quad K'(e) = \text{“}x = e\text{;”}$$
$$\mathrm{B}_K[\![\lambda x.\ t\ /\ x_1\ x_2 \ldots x_n]\!] = \mathrm{B}_K[\![t\ /\ x_2 \ldots x_n]\!] \qquad (x \text{ and } x_1 \text{ are mapped to the same C variable})$$
$$\mathrm{B}_K[\![\texttt{match } x \texttt{ with } (C_i \Rightarrow \lambda y_{i1} \ldots \lambda y_{i\,\mathrm{NM}_{C_i}}.\ t_i)_{i=1\ldots h} \texttt{ end}\ /\ x_1 \ldots x_n]\!] = \qquad \text{where} \quad x : T$$
“`switch (`$swfunc_T(x)$`) {`
...
$caselabel_{C_i}$: $y_{i1} = get\_member_{C_i 1}(x);\ \ldots;\ y_{i\,\mathrm{NM}_{C_i}} = get\_member_{C_i\,\mathrm{NM}_{C_i}}(x);$
$\mathrm{B}_K[\![t_i\ /\ x_1 \ldots x_n]\!]$
...
`}`”

$$\mathrm{B}_K[\![\texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j\ /\ x_1 \ldots x_n]\!] =$$
“ passign(fvars$[\![t_j]\!], x_1 \ldots x_n$)
$\mathrm{GENBODY}^{\mathrm{B}}_K[\![\texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j]\!]$”

Note: We do not define $\mathrm{B}_K[\![\lambda x.\ t\ /\ ]\!]$ because a tail position cannot be a function after the argument completion.

Figure 8. Translation to C for a tail position.

$$\mathrm{fvars}[\![t]\!] = \begin{cases} \text{“}x;\ \mathrm{fvars}[\![u]\!]\text{”} & t = \lambda x.\ u \\ \mathrm{fvars}[\![t_j]\!] & t = \texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j \\ \text{“”} & \text{otherwise} \end{cases}$$

$$\mathrm{fvars}'[\![f_i]\!] = \mathrm{fvars}[\![t_i]\!] \quad \text{for functions bounded by } \texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j$$

$$\mathrm{GENBODY}^{\mathrm{AT}}_K[\![t]\!] = \begin{cases} \mathrm{GENBODY}^{\mathrm{AT}}_K[\![u]\!] & t = \lambda x.\ u \\ \text{“}\texttt{entry\_}f_i\texttt{:}\ \mathrm{GENBODY}^{\mathrm{AT}}_K[\![t_i]\!]\text{”} & t = \texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j \\ \quad \text{for } i = j, 1, \ldots, (j-1), (j+1), \ldots, h & \\ \mathrm{A}_K[\![t\ /\ ]\!] & \text{otherwise} \end{cases}$$

$$\mathrm{GENBODY}^{\mathrm{AN}}[\![t]\!] = \begin{cases} \mathrm{GENBODY}^{\mathrm{AN}}[\![u]\!] & t = \lambda x.\ u \\ \text{“}\texttt{entry\_}f_i\texttt{:}\ \mathrm{GENBODY}^{\mathrm{AN}}[\![t_i]\!]\text{”} & t = \texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j \\ \quad \text{for } i = 1, \ldots, h & \\ \mathrm{B}_K[\![t\ /\ ]\!] & \text{otherwise} \quad \text{where } t : T \quad K(e) = \text{“}\texttt{*(}T\texttt{*)ret = }e\texttt{; return;”} \end{cases}$$

$$\mathrm{GENBODY}^{\mathrm{B}}_K[\![t]\!] = \begin{cases} \mathrm{GENBODY}^{\mathrm{B}}_K[\![u]\!] & t = \lambda x.\ u \\ \text{“}\texttt{entry\_}f_i\texttt{:}\ \mathrm{GENBODY}^{\mathrm{B}}_K[\![t_i]\!]\text{”} & t = \texttt{fix } (f_i := t_i)_{i=1\ldots h} \texttt{ for } f_j \\ \quad \text{for } i = j, 1, \ldots, (j-1), (j+1), \ldots, h & \\ \mathrm{B}_K[\![t\ /\ ]\!] & \text{otherwise} \end{cases}$$

Note:

- fvars and fvars′ returns a list of variables: $x_1; \ldots; x_n;$. For simplicity, we omit ";" if not ambiguous.
- "$g(i)$" for $i = j_1, \ldots, j_n$ means "$g(j_1)\ \ldots\ g(j_n)$".

Figure 9. Auxiliary functions for translation to C.

$\mathrm{TR}[\![t]\!]_n$ is the first element of $\mathrm{RNT}[\![t]\!]_n$.
$(R, N, T) = \mathrm{RNT}[\![t]\!]_n$ classify variables in t assuming that it is called with n arguments as $t\ x_1 \ldots x_n$:

R : tail-recursive fixpoint bounded functions that do not need to be real functions

N : free variables at non-tail positions of t

T : free variables at tail positions of t

"tail position" is extended to the function position of the application at a tail position.
R distinguishes fixpoint bounded functions translatable without actual functions (but with `goto`) or not.

$$\mathrm{TR}[\![t]\!]_n = R \quad \text{where} \quad (R, N, T) = \mathrm{RNT}[\![t]\!]_n$$

$$\mathrm{RNT}[\![t]\!]_n = \begin{cases} (\varnothing, \varnothing, \{x\}) & t = x \\ (\varnothing, \varnothing, \varnothing) & t = c \vee t = C \\ (R, N \cup \{x\}, T) & t = u\ x \quad \text{where} \quad (R, N, T) = \mathrm{RNT}[\![u]\!]_{n+1} \\ (R_1 \cup R_2, N_1 \cup T_1 \cup N_2 - \{x\}, T_2 - \{x\}) & t = \texttt{let}\ x := t_1\ \texttt{in}\ t_2 \\ & \text{where} \quad (R_1, N_1, T_1) = \mathrm{RNT}[\![t_1]\!]_0 \quad (R_2, N_2, T_2) = \mathrm{RNT}[\![t_2]\!]_n \\ \left(\bigcup_{i=1}^h R_i, \bigcup_{i=1}^h N_i, \bigcup_{i=1}^h T_i\right) & t = \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end} \quad \text{where } (R_i, N_i, T_i) = \mathrm{RNT}[\![t_i]\!]_{\mathrm{NM}_{C_i}+n} \\ (R, N - \{x\}, T - \{x\}) & t = \lambda x.\ u \wedge n > 0 \quad \text{where} \quad (R, N, T) = \mathrm{RNT}[\![u]\!]_{n-1} \\ (R, (N \cup T) - \{x\}, \varnothing) & t = \lambda x.\ u \wedge n = 0 \quad \text{where} \quad (R, N, T) = \mathrm{RNT}[\![u]\!]_{\mathrm{NA}_u} \quad \text{(closure)} \\ \left(\bigcup_{i=1}^h R_i \cup \{f_1, \ldots, f_h\},\ \bigcup_{i=1}^h N_i - \{f_1, \ldots, f_h\},\ \bigcup_{i=1}^h T_i - \{f_1, \ldots, f_h\}\right) & t = \texttt{fix}\ (f_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \wedge \\ & n = \mathrm{NA}_t \wedge \left(\bigcup_{i=1}^h N_i \cap \{f_1, \ldots, f_h\} = \varnothing\right) \\ & \text{where} \quad (R_i, N_i, T_i) = \mathrm{RNT}[\![t_i]\!]_{\mathrm{NA}_{t_i}} \\ \left(\bigcup_{i=1}^h R_i,\ \bigcup_{i=1}^h (N_i \cup T_i) - \{f_1, \ldots, f_h\},\ \varnothing\right) & t = \texttt{fix}\ (f_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \ \wedge \quad \text{(real function, maybe closure)} \\ & \neg\left(n = \mathrm{NA}_t \wedge \left(\bigcup_{i=1}^h N_i \cap \{f_1, \ldots, f_h\} = \varnothing\right)\right) \\ & \text{where} \quad (R_i, N_i, T_i) = \mathrm{RNT}[\![t_i]\!]_{\mathrm{NA}_{t_i}} \end{cases}$$

Note:

- NA_t is the number of arguments of t: $\mathrm{NA}_t = m$ if $t : T_1 \to \cdots \to T_m \to T_0$ and T_0 is an inductive type.
- The variables in t are unique. Codegen uses de Bruijn's indexes for N and T; the variables renamed by Section 4.8 for R.

Figure 10. Detection of tail recursive fixpoint bounded functions.

```
static nat pow3(nat v1_x) {
  nat v2_n; nat v3_n; nat v4_n; nat v5_n;
  v2_n = O();
  v3_n = S(v2_n);
  v4_n = mul(v1_x, v3_n);
  v5_n = mul(v1_x, v4_n);
  return mul(v1_x, v5_n);
}

static nat f(nat v1_x, nat v2_y) {
  nat v3_n; nat v4_n;
  v3_n = pow3(v1_x);
  v4_n = pow3(v2_y);
  return add(v3_n, v4_n);
}
```

Figure 11. `pow3` and `f` generated by Codegen.

```
typedef uint64_t nat;
#define O() 0
#define S(n) ((n)+1)
#define add(x,y) ((x) + (y))
#define mul(x,y) ((x) * (y))
```

Figure 12. `nat` implementation using `uint64_t`.

We can monomorphize `rev` by the convertible transformations as Figure 16. `s_rev_bool` is a monomorphic function without type arguments.

Codegen generates a C function as Figure 17 for it. This function depends on the implementation of `bool` type (`bool` type), `list bool` type (`list_bool` type; `lb_nil` and `lb_cons` for constructors; `lb_is_nil`, `lb_head`, and `lb_tail` for `match`-expression implementation), and the implementation of `app` (`++` operator) for `list bool` type (`app_bool`). See Appendix F for details.

```
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 (String _ rest) ⇒ 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_addstr buf 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.
```

Figure 13. `sprintf` definition using dependent type.

```
Definition s_sprintf_x_eq_nat v1_buf v2_n :=
  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
```

- `Ascii` is the constructor of `ascii` type defined in the standard library of Coq.
- `v11_a` and `v20_a` are the characters `"x"` and `"="` (0x78 and 0x3d in ASCII).

Figure 14. `sprintf` specialized with respect to the format string `"x=%d"`.

```
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)
static buffer sprintf_x_eq_nat(buffer v1_buf, nat v2_n) {
  bool v3_b; bool v4_b; bool v5_b; bool v6_b; bool v7_b; bool v8_b; bool v9_b; bool v10_b; ascii v11_a;
  bool v12_b; bool v13_b; bool v14_b; bool v15_b; bool v16_b; bool v17_b; bool v18_b; bool v19_b; ascii v20_a;
  buffer v21_b; buffer v22_b; buffer v23_b;
  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);
  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;
}
```

- `ascii` type and `Ascii` macro are a user-defined implementation of `ascii` type.

Figure 15. C code generation of `sprintf` specialized with respect to the format string `"x=%d"`.

```
Definition p_rev_bool := rev bool.
Definition s_rev_bool :=
  fix rev_bool (v1_l : list bool) : list bool :=
    match v1_l with
    | nil ⇒ lb_nil
    | v2_x :: v3_l_ ⇒
        let v4_l := rev_bool v3_l_ in
        let v5_l := lb_nil in
        let v6_l := lb_cons v2_x v5_l in
        p_app_bool v4_l v6_l
    end
```

Figure 16. The monomorphized `rev` function.

```
static list_bool rev_bool(list_bool v1_l)
{ bool v2_x; list_bool v3_l_; list_bool v4_l;
  list_bool v5_l; list_bool v6_l;
  switch (lb_is_nil(v1_l))
  { default: return lb_nil;
    case 0:  v2_x = lb_head(v1_l);
             v3_l_ = lb_tail(v1_l);
             v4_l = rev_bool(v3_l_);
             v5_l = lb_nil;
             v6_l = lb_cons(v2_x, v5_l);
             return app_bool(v4_l, v6_l);
  } }
```

Figure 17. Generated C code for `rev bool`.

6.4 Dependent Type Elimination of `sprintf`

Gallina can use dependent types to define functions which type depends on (non-type) arguments. For example, `sprintf` in Figure 13 is a function that the second argument `fmt` defines types of subsequent arguments. The types of subsequent arguments are computed by `sprintf_type`.

Codegen can specialize `sprintf` function to eliminate dependent types. Codegen specializes `sprintf` with respect to the format string `"x=%d"` as in Figure 14. The result does not contain dependent types. The result is very long because all bits of all characters in the format string are bound by variables. Figure 15 shows the C function generated for the specialized `sprintf` function.

`sprintf` takes a buffer as the first argument and returns a buffer that formatted string is appended. `buffer` type is expected to be declared as linear and the `buf_addch` function modifies the buffer destructively. See Appendix G for details.

7 Related Work

The Extraction plugin [6] is a standard plugin of Coq. It translates Gallina to OCaml, Haskell, and Scheme. It translates one Gallina function to one function in a target language and has no feature to specialize functions. When a Gallina function uses a dependent type which is not typable in OCaml, `Obj.magic` is used. This requires a uniform data representation where all values can be represented by a single type (single word, typically) at run time. Codegen does not have this requirement and any specific representation is usable for each inductive type.

CertiCoq [1] and Œuf [7] are compilers for Gallina to generate executable code using CompCert [5]. They use a uniform data representation and have no specialization.

A-normal form [4], K-normal form [2], and our V-normal form restricts application arguments as variables. They also restrict condition expressions as variables. A-normal form prohibits a let-in and conditional expression in a let-in binding but K-normal form and V-normal form permit it. A-normal form restricts function position with values (constant, variable, lambda abstraction) and primitive operations. K-normal form restricts function position with variables (including `letrec`-bounded functions). V-normal form permits in function position any V-normal term including let-in, conditional expression, and fixpoint. This permissibility of V-normal form is important to represent a loop using an application which function position is a fixpoint.

Continuation Passing Style (CPS) is another choice for an intermediate language of compilers. CPS and V-normal form restricts arguments as variables. The naive CPS translation produces many administrative redexes that can be reduced at compile time [3]. Our V-reduction restricts arguments a term as variables and S-reduction simplifies the term. The difference between them is V-reduction does not introduce extra constructs if an argument is already a variable and S-reduction reduces redexes not only administrative redexes.

8 Conclusion

We described that conversion rules-based partial evaluation is possible and its result is easily verifiable in Coq. The partial evaluation is usable for monomorphization and dependent type elimination. We formalized the partial evaluation as reductions over a subset of Gallina term, V-normal form.

Codegen implements the partial evaluation and generates C functions. We defined several transformations to make a result of the partial evaluation close to C to ease C code generation. These transformations are also convertible as the partial evaluation and its result is easily verifiable.

The C code generation in Codegen guarantees tail recursion elimination. It eliminates tail recursion in a tail position and also eliminates tail recursion in a non-tail position if possible.

Acknowledgments

This research was partially supported by the JSPS Bilateral Joint Research (Number: 20203001)/Inria AYAME Program Project FLAVOR.

A Summary of Solutions to the Gaps between Gallina and C

- Gallina has type polymorphism, but C does not.
  → Codegen uses partial evaluation to eliminate type polymorphism if possible.
- Gallina has dependent types, but C does not.
  → Codegen uses partial evaluation to eliminate dependent types if possible.
- Gallina can use any evaluation strategy, but C is strict.
  → Codegen applies the strict evaluation strategy to Gallina terms.
- Gallina uses curried functions, but C does not.
  → Codegen applies the semantics that evaluates a function body after all arguments are given.
- Gallina has first-class functions, but C does not.
  → We have a plan to implement restricted closures (downward funarg).
- Gallina's data types (such as Peano's naturals defined as an inductive type) are suitable for proof but its naive implementation is too inefficient. C has efficient data types such as 64 bit integer.
  → Codegen is data representation agnostic. A user can customize the implementation of inductive types.
- All Gallina programs terminate but C does not.
  → Our target is library functions and most of them terminate.
- Gallina uses only immutable values but C uses mutable values.
  → Local variables are modifiable at tail recursive calls which are translated to assignments and `goto`. Also, Codegen has a linearity checker for destructive updates on heap structures safely. It checks the linearity of variables of user-specified types: one variable is used exactly once. It guarantees that destructive update is invisible from Gallina program.
- Gallina does not require to release memory but C requires.
  → The linearity checker guarantees to consume a linear variable. It can be used to force a user to invoke a memory freeing function.
- Gallina does not have a loop, but C does.
  → Codegen implements reliable tail recursion elimination.

B Details of Argument Completion

Figure 18 explains the detail of argument completion.

C Formalized C Variable Allocation

Figure 19 and Figure 20 formalizes C variable allocation. Figure 19 defines the syntax of a mapping from Gallina variables to C variables. Figure 20 generates a variable mapping for a Gallina term.

D The Gallina Subset for C Code Generation

Our C code generator can generate a C function if $E[] \vdash_{\mathrm{f}} t$. Figure 21 defines $E[\Gamma] \vdash_{\mathrm{f}} t$ and $E[\Gamma] \vdash_{\mathrm{b}} t$. $E[\Gamma] \vdash_{\mathrm{f}} t$ determines that t is valid function for our C code generator. $E[\Gamma] \vdash_{\mathrm{b}} t$ determines that t is valid body for our C code generator.

E Top-Level C Function Generation

Figure 22, Figure 23, and Figure 24 explains the top-level function generation.

Figure 22 generates C definitions for a Gallina function that needs multiple C functions. Figure 23 generates a C function for a Gallina function that needs only one C function. Figure 24 chooses them. Assuming a Gallina function c is defined as `Definition` $c := t$, it needs multiple functions when a function bounded by a fixpoint in t is called as a function (not `goto`) and c itself is not usable to call the function.

For example, `half` function defined as follows is mutually defined with `uphalf` function. `half` invokes `uphalf` in a tail position but `uphalf` invokes `half` in a non-tail position.

```
Definition half :=
  fix half n :=
    match n with O ⇒ O | S m ⇒ uphalf m end
  with uphalf n :=
    match n with O ⇒ O | S m ⇒ S (half m) end
  for half.
```

Codegen generates `half` as a single function that contains the bodies of both functions. This is because all recursive calls are translatable to `goto` or a function call to the top-level function (`half`): `half` invokes `uphalf` using `goto`; `uphalf` invokes `half` using a usual function call.

```
static nat half(nat v1_n) {
  nat v2_m, v3_n, v4_m, v5_n;
  switch (v1_n) {
    case 0:  return 0;
    default: v2_m = pred(v1_n);
             v3_n = v2_m;
             goto entry_uphalf;
  }
  entry_uphalf:
  switch (v3_n) {
    case 0:  return 0;
    default: v4_m = pred(v3_n);
             v5_n = half(v4_m);
             return succ(v5_n);
  }
}
```

- $\mathrm{F}[\![t]\!]$ considers t to be a top-level function or a closure-generating expression. F transforms t to be a nested abstraction expression that takes all arguments. Fixpoint expressions are allowed outside or between the abstractions.
- $\mathrm{BR}[\![t]\!]_{m,q}$ transforms a branch of a match expression into a nested abstraction expression that takes constructor members. Fixpoint expressions are not allowed.
- $\mathrm{E}[\![t\ /\ x_1 \ldots x_p]\!]_q$ is a term convertible with $t\ x_1 \ldots x_p$ that does not contain a partial application. $\mathrm{E}[\![t\ /\ x_1 \ldots x_p]\!]_q$ traverses t while tracking the arguments for t to find closure-generating expressions. The number of arguments given to t is $p + q$. The first p arguments are $x_1 \ldots x_p$ and they can be the argument of beta-var redex. The last q arguments cannot be the argument of beta-var redex.

$$\mathrm{F}[\![t]\!] = \begin{cases} \lambda x.\ \mathrm{F}[\![u]\!] & t = \lambda x.\ u \\ \texttt{fix}\ (f_i := \mathrm{F}[\![t_i]\!])_{i=1\ldots h}\ \texttt{for}\ f_j & t = \texttt{fix}\ (f_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \\ \lambda x_1 \ldots \lambda x_m.\ \mathrm{E}[\![t\ /\ x_1 \ldots x_m]\!]_0 & \text{otherwise} \quad \text{(eta expansion)} \end{cases}$$

where $t : T_1 \to \cdots \to T_m \to T_0$
T_0 is an inductive type
$x_1 \ldots x_m$ are fresh variables

$$\mathrm{BR}[\![t]\!]_{m,q} = \begin{cases} \mathrm{E}[\![t\ /\ ]\!]_q & m = 0 \\ \lambda x.\ \mathrm{BR}[\![u]\!]_{m-1,q} & m > 0\ \wedge\ t = \lambda x.\ u \\ \lambda x_1 \ldots \lambda x_m.\ \mathrm{E}[\![t\ /\ x_1 \ldots x_m]\!]_q & \text{otherwise} \quad \text{(eta expansion)} \end{cases}$$

where $x_1 \ldots x_m$ are fresh variables

$$\mathrm{E}[\![t\ /\ x_1 \ldots x_p]\!]_q = \begin{cases} x & t = x \wedge p = q = 0 \\ x\ x_1 \ldots x_p & t = x \wedge \neg(p = q = 0) \wedge r = 0 \\ \mathrm{F}[\![x\ x_1 \ldots x_p]\!] & t = x \wedge \neg(p = q = 0) \wedge r > 0 \\ t\ x_1 \ldots x_p & (t = c \vee t = C) \wedge r = 0 \\ \mathrm{F}[\![t\ x_1 \ldots x_p]\!] & (t = c \vee t = C) \wedge r > 0 \\ \mathrm{F}[\![t]\!] & t = \lambda x.\ u \wedge p = q = 0 \\ \mathrm{E}[\![u\{x/x_1\}\ /\ x_2 \ldots x_p]\!]_q & t = \lambda x.\ u \wedge p > 0 \wedge r = 0 \quad \text{(beta-var)} \\ \lambda x.\ \mathrm{E}[\![u\ /\ ]\!]_{p+q-1} & t = \lambda x.\ u \wedge ((p = 0 \wedge q > 0) \vee r > 0) \\ \mathrm{E}[\![u\ /\ x_0\ x_1 \ldots x_p]\!]_q & t = u\ x_0 \\ \texttt{let}\ x := \mathrm{E}[\![t_1\ /\ ]\!]_0\ \texttt{in}\ \mathrm{E}[\![t_2\ /\ x_1 \ldots x_p]\!]_q & t = \texttt{let}\ x := t_1\ \texttt{in}\ t_2 \quad \text{(zeta-app)} \\ \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow \mathrm{BR}[\![t_i]\!]_{\mathrm{NM}_{C_i},p+q})_{i=1\ldots h}\ \texttt{end}\ x_1 \ldots x_p & t = \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow t_i)_{i=1\ldots h}\ \texttt{end} \\ (\texttt{fix}\ (f_i := \mathrm{F}[\![t_i]\!])_{i=1\ldots h}\ \texttt{for}\ f_j)\ x_1 \ldots x_p & t = \texttt{fix}\ (f_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \end{cases}$$

where $t : T_1 \to \cdots \to T_p \to T_{p+1} \to \cdots \to T_{p+q} \to T_{p+q+1} \to \cdots \to T_{p+q+r} \to T_0$
T_0 is an inductive type

Note: This transformation assumes t is not dependently typed: no type terms and no dependent `match`-expressions.

Figure 18. Argument completion.

x : Gallina variable
v : C variable
$V = \mathit{empty} \mid x \mapsto v \mid V; V$

Figure 19. Variable mapping from Gallina to C.

F `list bool` Implementation

`list bool` type can be implemented in C as Figure 25. This implementation uses `malloc()` but not `free()`. Thus a conservative GC is required.

G `buffer` Implementation

`buffer` type can be implemented in C as Figure 26. `unit` type is also implemented.

`make_buffer` allocates a buffer, `buf_addch` adds a character to the buffer destructively, and `free_buffer` deallocates the buffer.

`buf_addch` and `free_buffer` have side effects but it is not visible from Gallina as far as `buffer` type is used linearly. Codegen causes an error for a non-linear use of `buffer` type variables when `buffer` type is declared as linear.

$\mathrm{CV}[\![t \,/\, x_1 \ldots x_n]\!]_V$ is the variable mapping of the variables declared in t.
$x_1 \ldots x_n$ are arguments for t. V is the variable mapping for variables declared outside.

$$\mathrm{CV}[\![t \,/\, x_1 \ldots x_n]\!]_V = \begin{cases} \mathit{empty} & t = x \vee t = c \vee t = C \vee t = T \\ \mathrm{CV}[\![u \,/\,]\!]_{V;M}; M & t = \lambda x.\, u \wedge n = 0 \quad \text{where } M = x \mapsto v \\ \mathrm{CV}[\![u \,/\, x_2 \ldots x_n]\!]_{V;M}; M & t = \lambda x.\, u \wedge n > 0 \quad \text{where } M = x \mapsto V(x_1) \\ \mathrm{CV}[\![u \,/\, x_0\ x_1 \ldots x_n]\!]_V & t = u\ x_0 \\ \mathrm{CV}[\![t_1 \,/\,]\!]_V; \mathrm{CV}[\![t_2 \,/\, x_1 \ldots x_n]\!]_{V;M}; M & t = \texttt{let}\ x := t_1\ \texttt{in}\ t_2 \quad \text{where } M = x \mapsto v \\ (\mathrm{CV}[\![t_1 \,/\, x_1 \ldots x_n]\!]_{V;M_1}; \ldots; & t = \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow \lambda y_{i1} \ldots \lambda y_{i\,\mathrm{NM}_{C_i}}.\, t_i)_{i=1\ldots h}\ \texttt{end} \\ \quad \mathrm{CV}[\![t_h \,/\, x_1 \ldots x_n]\!]_{V;M_h}; & \text{where } M_i = y_{i1} \mapsto v_{i1}; \ldots; y_{i\,\mathrm{NM}_{C_i}} \mapsto v_{i\,\mathrm{NM}_{C_i}} \\ \quad M_1; \ldots; M_h) & \\ \mathrm{CV}[\![t_1 \,/\,]\!]_{V;M}; \ldots; \mathrm{CV}[\![t_h \,/\,]\!]_{V;M}; M & t = \texttt{fix}\ (f_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \\ & \text{where } M = f_1 \mapsto v_1; \ldots; f_h \mapsto v_h \end{cases}$$

where v, v_i, v_{ij} are fresh C variables

Note: We consider Gallina variables unique.

Figure 20. C variable allocation.

$$\frac{}{E[\Gamma] \vdash_{\mathrm{b}} x} \qquad \frac{}{E[\Gamma] \vdash_{\mathrm{b}} c} \qquad \frac{}{E[\Gamma] \vdash_{\mathrm{b}} C}$$

$$\frac{E[\Gamma] \vdash_{\mathrm{b}} t \quad E[\Gamma] \vdash x : T \quad T \text{ is a non-dependent inductive type}}{E[\Gamma] \vdash_{\mathrm{b}} t\ x}$$

$$\frac{E[\Gamma] \vdash_{\mathrm{b}} t \quad E[\Gamma :: (x := t : T)] \vdash_{\mathrm{b}} u \quad T \text{ is a non-dependent inductive type}}{E[\Gamma] \vdash_{\mathrm{b}} \texttt{let}\ x := t : T\ \texttt{in}\ u}$$

$$\frac{\begin{array}{c} E[\Gamma] \vdash x : T \quad T \text{ is a non-dependent inductive type with } p \text{ parameters: } I\ u_1 \ldots u_p \\ E[] \vdash C_i\ u_1 \ldots u_p : T_{i1} \to \cdots \to T_{i\,\mathrm{NM}_{C_i}} \to T \quad T_{ij} \text{ are non-dependent inductive types} \\ E[\Gamma :: (y_{i1} : T_{i1}) :: \cdots :: (y_{i\,\mathrm{NM}_{C_i}} : T_{i\,\mathrm{NM}_{C_i}})] \vdash_{\mathrm{b}} t_i \end{array}}{E[\Gamma] \vdash_{\mathrm{b}} \texttt{match}\ x\ \texttt{with}\ (C_i \Rightarrow \lambda y_{i1} \ldots \lambda y_{i\,\mathrm{NM}_{C_i}}.\, t_i)_{i=1\ldots h}\ \texttt{end}}$$

$$\frac{E[\Gamma :: (x : T)] \vdash_{\mathrm{b}} t \quad T \text{ is a non-dependent inductive type}}{E[\Gamma] \vdash_{\mathrm{b}} \lambda x : T.\, t}$$

$$\frac{E[\Gamma] \vdash_{\mathrm{f}} \texttt{fix}\ (f_i : T_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j}{E[\Gamma] \vdash_{\mathrm{b}} \texttt{fix}\ (f_i : T_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j}$$

$$\frac{E[\Gamma] \vdash t : T \quad T \text{ is a non-dependent inductive type} \quad E[\Gamma] \vdash_{\mathrm{b}} t}{E[\Gamma] \vdash_{\mathrm{f}} t}$$

$$\frac{E[\Gamma :: (x : T)] \vdash_{\mathrm{f}} t \quad T \text{ is a non-dependent inductive type}}{E[\Gamma] \vdash_{\mathrm{f}} \lambda x : T.\, t}$$

$$\frac{E[\Gamma :: (f_1 : T_1) :: \cdots :: (f_h : T_h)] \vdash_{\mathrm{f}} t_i}{E[\Gamma] \vdash_{\mathrm{f}} \texttt{fix}\ (f_i : T_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j}$$

Figure 21. The Gallina subset for C code generation.

GENFUN$^{\mathrm{M}}[\![c]\!]$ translates the function (constant) c with one or more auxiliary functions. We assume c is defined as `Definition` $c := t$. The auxiliary functions $f_1 \ldots f_n$ are fixpoint bounded functions in t which are invoked as functions. We assume the types of them:

$$c : T_{01} \to \cdots \to T_{0m_0} \to T_{00}$$
$$f_i : T_{i1} \to \cdots \to T_{im_i} \to T_{i0} \qquad i = 1 \ldots n$$

where T_{i0} are inductive types ($i = 0 \ldots n$)

The formal arguments of c are $x_{01} \ldots x_{0m_0} = \mathrm{fvars}[\![t]\!]$ and the formal arguments of f_i are $x_{i1} \ldots x_{im_i} = \mathrm{fvars}'[\![f_i]\!]$. f_i invocation in C needs extra arguments, $y_{i1} : U_{i1} \ldots y_{io_i} : U_{io_i}$, addition to the actual arguments in Gallina application because the free variables of the fixpoint should also be passed. If the free variables contain a function bounded by an outer fixpoint, the function itself is not passed but the free variables of the outer fixpoint are also passed. We iterate it until no fixpoint functions.

GENFUN$^{\mathrm{M}}[\![c]\!]$ = "enum_entries$[\![c]\!]$ arg_structdefs$[\![c]\!]$ forward_decl$[\![c]\!]$ entry_functions$[\![c]\!]$ body_function$[\![c]\!]$"

enum_entries$[\![c]\!]$ = "`enum` `enum_func_`c `{func_`c`,func_`f_1`,` $\ldots$ `,func_`f_n`};`"

arg_structdefs$[\![c]\!]$ = "main_structdef$[\![c]\!]$ aux_structdef$[\![c]\!]_1 \ldots$ aux_structdef$[\![c]\!]_n$"

main_structdef$[\![c]\!]$ = "`struct` `arg_`c `{` T_{01} `arg1;` $\ldots$ T_{0m_0} `arg`m_0`;};`"

aux_structdef$[\![c]\!]_i$ = "`struct` `arg_`f_i `{` U_{i1} `outer1;` $\ldots$ U_{io_i} `outer`o_i`;` T_{i1} `arg1;` $\ldots$ T_{im_i} `arg`m_i`;};`"

forward_decl$[\![c]\!]$ = "`static void body_function_`c`(enum enum_func_`c `g,void *arg,void *ret);`"

entry_functions$[\![c]\!]$ = "main_function$[\![c]\!]$ aux_function$[\![c]\!]_1 \ldots$ aux_function$[\![c]\!]_n$"

main_function$[\![c]\!]$ = "`static` T_{00} `c(`T_{01} x_{01}`,` $\ldots$ `,`T_{0m_0} x_{0m_0}`) {`
`struct arg_c arg = {`x_{01}`,` $\ldots$ `,`x_{0m_0}`};` T_{00} `ret;`
`body_function_c(func_c,&arg,&ret);return ret;`
`}`"

aux_function$[\![c]\!]_i$ = "`static` T_{i0} f_i`(`U_{i1} y_{i1}`,` $\ldots$ `,`U_{io_i} y_{io_i}`,`T_{i1} x_{i1}`,` $\ldots$ `,`T_{im_i} x_{im_i}`) {`
`struct arg_`f_i `arg = {`y_{i1}`,` $\ldots$ `,`y_{io_i}`,`x_{i1}`,` $\ldots$ `,`x_{im_i}`};` T_{i0} `ret;`
`body_function_c(func_`f_i`,&arg,&ret); return ret;`
`}`"

body_function$[\![c]\!]$ = "`static void body_function_c(enum enum_func_c g,void *arg,void *ret) {`
decls
`switch (g) {` aux_case$[\![c]\!]_1 \ldots$ aux_case$[\![c]\!]_n$ main_case$[\![c]\!]$ `}`
GENBODY$^{\mathrm{B}}_K[\![t]\!]$
`}`"

aux_case$[\![c]\!]_i$ = "`case func_`f_i`:`
y_{i1} `= ((struct arg_`f_i `*)arg)->outer1;` $\ldots$ `;` y_{io_i} `= ((struct arg_`f_i `*)arg)->outer`o_i`;`
x_{i1} `= ((struct arg_`f_i `*)arg)->arg1;` $\ldots$ `;` x_{im_i} `= ((struct arg_`f_i `*)arg)->arg`m_i`;`
`goto entry_`f_i`;`"

main_case$[\![c]\!]$ = "`default:;`
x_{01} `= ((struct arg_c *)arg)->arg1;` $\ldots$ `;` x_{0m_0} `= ((struct arg_c *)arg)->arg`m_0`;`"

where *decls* is local variable declarations for variables used in GENBODY$^{\mathrm{B}}_K[\![t]\!]$.

$K(e)$ = "`*(`T_{00}`*)ret =` e`;return;`"

Figure 22. Translation for a top-level function which is translated to multiple C functions.

$\text{GENFUN}^{\text{S}}[\![c]\!]$ translates the function (constant) c to a single C function.

$$\text{GENFUN}^{\text{S}}[\![c]\!] = \text{“}\texttt{static}\ T_0\ c(\ \text{fargs}'[\![t]\!])\ \{\ \mathit{decls}\ \text{GENBODY}^{\text{B}}_K[\![t]\!]\ \}\text{”}$$

where c is defined as $\texttt{Definition}\ c : T_1 \to \cdots \to T_n \to T_0 := t.$

T_0 is an inductive type

decls is local variable declarations for variables used in $\text{GENBODY}^{\text{B}}_K[\![t]\!]$ excluding $\text{fargs}[\![t]\!]$.

$K(e) = \text{“}\texttt{return}\ e\,;\text{”}$

$$\text{fargs}[\![t]\!] = \begin{cases} \text{“}T\ x,\ \text{fargs}[\![u]\!]\text{”} & t = \lambda x{:}T.\ u \\ \text{fargs}[\![t_j]\!] & t = \texttt{fix}\ (f_i := t_i)_{i=1\ldots h}\ \texttt{for}\ f_j \\ \text{“”} & \text{otherwise} \end{cases}$$

$\text{fargs}'[\![t]\!] = \text{fargs}[\![t]\!]$ without the trailing comma

Figure 23. Translation for a top-level function which is translated to a single C function.

$$\text{GENFUN}[\![c]\!] = \begin{cases} \text{GENFUN}^{\text{M}}[\![c]\!] & t \text{ needs multiple functions} \\ \text{GENFUN}^{\text{S}}[\![c]\!] & \text{otherwise} \end{cases}$$

where c is defined as $\texttt{Definition}\ c := t.$

Figure 24. Translation for top-level function.

```
#include <stdlib.h>
#include <stdbool.h> /* bool, true, false */

struct list_bool_tag {
  bool head;
  struct list_bool_tag *tail;
};
typedef struct list_bool_tag *list_bool;

#define lb_nil NULL
static list_bool lb_cons(bool h, list_bool t) {
  list_bool c;
  if ((c = malloc(sizeof(*c))) == NULL)
    abort();
  c→head = h;
  c→tail = t;
  return c;
}

#define lb_is_nil(l) ((l) == lb_nil)
#define lb_head(l) ((l)→head)
#define lb_tail(l) ((l)→tail)
```

Figure 25. `list bool` implementation in C.

```
typedef bool unit; static const unit tt = true;
typedef struct {
  unsigned char *mem; size_t len; size_t max;
} *buffer;

buffer make_buffer(nat max) {
  buffer buf;
  if ((buf = malloc(sizeof(*buf))) == NULL) abort();
  if (max == 0) max = 16; /* avoid malloc(0) */
  if ((buf→mem = malloc(max)) == NULL) abort();
  buf→len = 0; buf→max = max;
  return buf;
}
buffer buf_addch(buffer buf, unsigned char c) {
  if (buf→max <= buf→len) {
    unsigned char *mem; size_t max = buf→max * 2;
    if ((mem = realloc(buf→mem, max)) == NULL)
      abort();
    buf→mem = mem; buf→max = max;
  }
  buf→mem[buf→len++] = c;
  return buf;
}
unit free_buffer(buffer buf) {
  free(buf→mem); free(buf);
  return tt;
}
```

Figure 26. `buffer` implementation in C.

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*.

[2] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 171–183. https://doi.org/10.1145/237721.237771

[3] Olivier Danvy and Lasse R Nielsen. 2005. CPS transformation of beta-redexes. *Inform. Process. Lett.* 94, 5 (2005), 217–224. https://doi.org/10.1016/j.ipl.2005.02.002

[4] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 237–247. https://doi.org/10.1145/155090.155113

[5] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[6] Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002 (Lecture Notes in Computer Science, Vol. 2646)*, Herman Geuvers and Freek Wiedijk (Eds.). Springer-Verlag. https://doi.org/10.1007/3-540-39185-1_12

[7] Eric Mullen, Stuart Pernsteiner, James R Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 172–185. https://doi.org/10.1145/3167089

[8] Gonzalo Navarro. 2016. *Compact data structures: A practical approach*. Cambridge University Press. https://doi.org/10.1017/CBO9781316588284

[9] Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. 2018. Safe low-level code generation in Coq using monomorphization and monadification. *Journal of Information Processing* 26 (2018), 54–72. https://doi.org/10.2197/ipsjjip.26.54

[10] The Coq Development Team. 2020. The Coq Proof Assistant. https://coq.inria.fr/.

[11] The Coq Development Team. 2020. The Coq reference manual: Release 8.12.0. (2020).