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

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
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.

Ccodegen eliminates tail recursion using goto to address the last gap. Ccodegen can generate a C function that contains multiple loops and nested loops from Gallina terms which contain fixpoints at non-tail positions and nested fix-points. Ccodegen 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.

---

```
t = x
| c
| C
| T
| Ax:T, t
| t u
| let x := t : T in u
| match t0 with (Ci ⇒ ti)i=1...n end
| fix (fj :ki : Ti := tj)i=1...n for fj
```

Note:
- u represents a term as t.
- y, z, and f represent a variable as x.
- U represents a type as T.
- We write (···(t u1) u2)··· un as t u1 ... un.
- k is an integer.
- fj for fixpoint specify the decreasing argument for fj.
- If it is unambiguous, we omit type annotations for the sake of simplicity. We also omit ki 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 ti 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.
We implement the Gallina to C translation with the following way. The latter translates Gallina functions to C functions.

### 3 Translation from Gallina to C

Transformations do not change the result of the function. This allows us to verify with Coq that these 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 translations 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 0 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)

### 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\(^3\):

\begin{align*}
\text{Definition} \ p\_\text{pow3} \ x & := \text{pow} \ x \ 3. (* \text{before trans.} *) \\
\text{Definition} \ s\_\text{pow3} \ x & := \ldots (* \text{after trans.} *)
\end{align*}

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.

\(^2\)Invalid characters for a C identifier are replaced by _

\(^3\)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 monomorphism is still possible.
We define S-reductions in Figure 5 to simplify V-normal form. S-reductions are the reduction rules for the partial evaluation without computation duplication.

The zeta-reduction. It applies the zeta-reduction when a zeta-redex are variables because of V-normal form. It does not duplicate computation because the argument does not contain a match-expression.

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.

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 to 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 y1...yq (0 < q), and the bounded expression, t x1...xp (0 ≤ 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, x1...xp, has no computation because they are just variables. t x1...xp is a partial application because it is the function of f y1...yq and 0 < q. The evaluation of t x1...xp 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 t1, t2, t0. 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.
**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 subsumed 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 is 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 $\mathtt{pow}$, beta expansion is also used before the delta-global expansion: when the normal form of three is $3$, $\mathtt{pow\ z\ three}$ is transformed to $\mathtt{(fun\ x\ \Rightarrow\ \mathtt{pow\ x\ 3})\ z}$ and $(\mathtt{fun\ x\ \Rightarrow\ \mathtt{pow\ x\ 3})\ z}$ is replaced with the constant $p_\mathtt{pow3}$. 

---

**Figure 5.** S-reductions.
when an evaluation of \( t \) is evaluated with \( \mathbf{match} \ x \ \mathbf{with} \ tt \Rightarrow \mathbf{fun} \ y \Rightarrow y \ \mathbf{end} \ z \). We formalized it in Appendix C to generates a variable mapping from Gallina to C.

Codecgen 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

Codecgen generates C code from the result of the convertible transformations. However Codecgen gives up code generation if the result still contains type computations6 or closure generation. We describe the subset of Gallina that can be translated to C in Appendix D7.

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. Codecgen can generate C code for them as far as arguments are given lexically outside of them. For example, Codecgen can generate C code for \( \mathbf{match} \ x \ \mathbf{with} \ tt \Rightarrow \mathbf{fun} \ y \Rightarrow y \ \mathbf{end} \ z \). Since

---

6The temporary variables for parallel assignments are the exception.

7This is the intermediate representation for Codecgen, not the definition of the source Gallina terms for Codecgen. We do not define the source Gallina terms for Codecgen because the convertible transformations may remove arbitrary complex terms. For example, Codecgen can handle

\[
\mathbf{match} \ u \ \mathbf{with} \ \mathbf{true} \Rightarrow 0 \ | \ \mathbf{false} \Rightarrow t \ \mathbf{end}
\]

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:

\[
\text{Definition } s_{\text{pow}}_{\text{2x}}\_3 \ v_1\_x := \\
\text{let } v_2\_n := v_1\_x + v_1\_x \text{ in} \\
\text{let } v_3\_n := 0 \text{ in} \\
\text{let } v_4\_n := v_1\_x \times v_3\_n \text{ in} \\
\text{let } v_5\_n := v_1\_x \times v_4\_n \text{ in} \\
\text{let } v_6\_n := v_2\_n \times v_5\_n.
\]

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.

\[
\text{Definition } f \ x \ y := \text{pow} \ x \ 3 + \text{pow} \ y \ 3.
\]

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

\[
\text{Definition } p_{\text{pow}}\_3 \ x := \text{pow} \ x \ 3. \\
\text{Definition } s_{\text{f}} \ v_1\_x \ v_2\_y := \\
\text{let } v_3\_n := p_{\text{pow}}\_3 \ v_1\_x \text{ in} \\
\text{let } v_4\_n := p_{\text{pow}}\_3 \ v_2\_y \text{ in} \\
\text{v}_3\_n + v_4\_n.
\]

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

\[
\text{Definition } s_{\text{pow}}\_3 \ v_1\_x := \\
\text{let } v_2\_n := 0 \text{ in} \\
\text{let } v_3\_n := S \ v_2\_n \text{ in} \\
\text{let } v_4\_n := v_1\_x * v_3\_n \text{ in} \\
\text{let } v_5\_n := v_1\_x * v_4\_n \text{ in} \\
v_1\_x * v_5\_n
\]

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 pow3 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.\(^9\)

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, 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.)

Coq has a list reversal function \(\text{rev}\) in the standard library. It is defined as follows.\(^10\)

\[
\text{Definition } \text{rev} := \text{fun} (A : \text{Type}) \Rightarrow \\
\text{fix } \text{rev} \ (l : \text{list} \ A) : \text{list} \ A := \\
\text{match } l \text{ with} \\
| \text{nil} \Rightarrow \text{nil} \\
| x :: l' \Rightarrow \text{rev} \ l' ++ x :: \text{nil} \text{ end.}
\]

\(^9\)The standard library uses Section to define this term.

\(^10\)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.
$A_K[t/x_1 \ldots x_n]$ generates C code for $t$ in a non-tail position. The result expression is passed to $K$.

$K(e) = \langle v = e; \rangle$ in simple situations.

- $A_K[x/\ ] = K(\"x\")$
- $A_K[x/x_1 \ldots x_n] = \text{pass} (\text{fvars}[x], x_1 \ldots x_n) \text{ goto entry\_x; }$ $n > 0 \land x$ is bounded by a fixpoint $\land x \in TR$
- $A_K[x/x_1 \ldots x_n] = K(\"x(y_1, \ldots , y_0, x_1, \ldots , x_n)\")$ $n > 0 \land x$ is bounded by a fixpoint $\land x \notin TR$
- $A_K[c/x_1 \ldots x_n] = K(\"c(x_1, \ldots , x_n)\")$ $n \geq 0$
- $A_K[C/x_1 \ldots x_n] = K(\"C(x_1, \ldots , x_n)\")$ $n \geq 0$
- $A_K[t/x_0/x_1 \ldots x_n] = A_K[t/x_0/x_1 \ldots x_n]$
- $A_K[\lambda x. t/x_1 x_2 \ldots x_n] = A_K[t/x_2 \ldots x_n]$ (x and $x_1$ are mapped to the same C variable)
- $A_K[\text{match } x \text{ with } (C_i \Rightarrow y_{i1} \ldots y_{in_{MC_i}}; t_i)_{i=1\ldots h} \text{ end} / x_1 \ldots x_n] = \text{switch (swfunc}_T(x)) \{
\}
\text{caselabel}\_C_i: ;
\text{break; }
\}
\text{pass} (\text{fvars}[t_j], x_1 \ldots x_n)$ $f_j \in TR$
- $\text{GENBODY}_{AN}[\text{fix } (f_i := t_i)_{i=1\ldots h} \text{ for } f_j / x_1 \ldots x_n] = \text{GENBODY}_{AN}[\text{fix } (f_i := t_i)_{i=1\ldots h} \text{ for } f_j / x_1 \ldots x_n]$
- $K(f_j(y_{i1}, \ldots , y_{in}, x_1, \ldots , x_n))$
- $\text{goto } \text{skip}_f_j;$
- $\text{GENBODY}_{AN}[\text{fix } (f_i := t_i)_{i=1\ldots h} \text{ for } f_j / x_1 \ldots x_n]$
- $\text{skip}_f_j;$

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.
- $TR = \text{TR}[t_n]$ where the translating function is defined as $\text{Definition } c := t$ and $t$ is an $n$-arguments function.
- $\text{NM}_C$ is the number of the members of the constructor $C$ (the number of arguments without the parameters for the inductive type): $\text{NM}_C = m$ if $C : T_1 \rightarrow \cdots \rightarrow T_0$ is inductive type which has $p$ parameters.
- $\text{swfunc}_T, \text{caselabel}_C$, and $\text{get\_member}_{C,j}$ are defined by a user to translate match-expressions for the inductive type $T$.
- passassign$(y_{i1} \ldots y_{in}, x_1, \ldots , x_n)$ is a parallel assignment. It is translated to a sequence of assignments to assign $x_1 \ldots x_n$ into $y_{i1} \ldots y_{in}$. It may require temporary variables.
- $y_1, \ldots , y_n$ are the outer variables of the fixpoint.
- We do not define $A_K[\lambda x. t/\ ]$ because we do not support closures yet.
- Actual Codegen generates $\text{GENBODY}_{AN}[\ ]$ in a different position to avoid the label $\text{skip}_f_j$ and $\text{goto } \text{skip}_f_j;$. 

Figure 7. Translation to C for a non-tail position.
\(B_K[t/x_1\ldots x_n]\) generates C code for \(t x_1\ldots x_n\) in a tail position. The result expression is passed to \(K\).

\(K(e) = \textit{return } e;\) in simple situations.

\[
\begin{align*}
B_K[x/f] &= K("x") \\
B_K[x/x_1\ldots x_n] &= \text{"passign(fvars'[x], x_1\ldots x_n) goto entry_x;" } & n > 0 \land x \text{ is bounded by a fixpoint} \\
B_K[c/x_1\ldots x_n] &= K("c(x_1,\ldots ,x_n)") \\
B_K[C/x_1\ldots x_n] &= K("C(x_1,\ldots ,x_n)") \\
B_K[t/x_0/x_1\ldots x_n] &= B_K[t/x_0 x_1\ldots x_n] \\
B_K[\text{let } x := t_1 \text{ in } t_2 / x_1\ldots x_n] &= \text{"A}_K[\text{let } t_1 / \text{let } t_2 / x_1\ldots x_n]" \quad \text{where } K'(e) = \text{"x = e;"} \\
B_K[\lambda x. t / x_1 x_2\ldots x_n] &= B_K[t / x_2\ldots x_n] \\
\text{fvars}[t] &= \text{fvars'}[t_1] \\
\text{fvars'}[t_1] &= \text{fvars}[t_1] \quad \text{for functions bounded by } \text{fix } (f_i := t_1)i=1\ldots h \text{ for } f_j \\
\text{GENBODY}_K^T[t] &= \begin{cases} 
\text{"entry}_f[i]: \text{GENBODY}_K^T[t_i]\" & t = \text{fix } (f_i := t_1)i=1\ldots h \text{ for } f_j \\
\text{GENBODY}_K^T[t_i] & \text{for } i = j, 1, \ldots , (j-1), (j+1), \ldots , h \\
\text{GENBODY}_K[\text{fix } (f_i := t_1)i=1\ldots h / f_j] & \text{otherwise} \\
\end{cases} \\
\text{GENBODY}_K^A[t] &= \begin{cases} 
\text{"entry}_f[i]: \text{GENBODY}_K^A[t_i]\" & t = \text{fix } (f_i := t_1)i=1\ldots h \text{ for } f_j \\
\text{GENBODY}_K^A[t_i] & \text{for } i = 1, \ldots , h \\
\text{GENBODY}_K[\text{fix } (f_i := t_1)i=1\ldots h / f_j] & \text{otherwise where } t : T \text{ } K(e) = \text{"*(T*)ret } e; \text{ return;"} \\
\end{cases} \\
\text{GENBODY}_K^B[t] &= \begin{cases} 
\text{"entry}_f[i]: \text{GENBODY}_K^B[t_i]\" & t = \text{fix } (f_i := t_1)i=1\ldots h \text{ for } f_j \\
\text{GENBODY}_K^B[t_i] & \text{for } i = j, 1, \ldots , (j-1), (j+1), \ldots , h \\
\text{GENBODY}_K[\text{fix } (f_i := t_1)i=1\ldots h / f_j] & \text{otherwise} \\
\end{cases}
\end{align*}
\]

Note:

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

\textbf{Figure 9.} Auxiliary functions for translation to C.
TR[t]_n is the first element of RNT[t]_n.
(R, N, T) = 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.

TR[t]_n = R where (R, N, T) = RNT[t]_n

RNT[t]_n =

\[
(\varnothing, \varnothing, \{x\}) \quad t = x \\
(\varnothing, \varnothing, \varnothing) \quad t = c \lor t = C \\
(R, N \cup \{x\}, T) \quad t = u \times \text{where} (R, N, T) = RNT[u]_{n+1} \\
(R_1 \cup R_2, N_1 \cup T_1 \cup N_2 - \{x\}, T_2 - \{x\}) \quad t = \text{let } x := t_1 \text{ in } t_2 \\
\left(\bigcup_{i=1}^{h} R_i, \bigcup_{i=1}^{h} N_i, \bigcup_{i=1}^{h} T_i\right) \quad t = \text{match } x \text{ with } (C_i \Rightarrow t_i)_{i=1...h} \text{ end} \text{ where } (R_i, N_i, T_i) = RNT[t_i]_{NM_{C_i} + n} \\
(R, N - \{x\}, T - \{x\}) \quad t = \lambda x. \mu n > 0 \text{ where } (R, N, T) = RNT[u]_{n-1} \\
(R, (N \cup T) - \{x\}, \varnothing) \quad t = \lambda x. u \land n = 0 \text{ where } (R, N, T) = RNT[u]_{NA_u} \text{ (closure)} \\
\left(\bigcup_{i=1}^{h} R_i, \bigcup_{i=1}^{h} N_i, \bigcup_{i=1}^{h} T_i\right) \quad t = \text{fix } (f_i := t_i)_{i=1...h} \text{ for } f_j \land \\
\left(\bigcup_{i=1}^{h} N_i - \{f_i, \ldots, f_h\}, \bigcup_{i=1}^{h} T_i - \{f_i, \ldots, f_h\}\right) \quad n = NA_t \land (\bigcup_{i=1}^{h} N_i \cap \{f_i, \ldots, f_h\} = \varnothing) \text{ where} (R_i, N_i, T_i) = RNT[t_i]_{NA_{N_i}} \\
\left(\bigcup_{i=1}^{h} R_i, \bigcup_{i=1}^{h} N_i, \bigcup_{i=1}^{h} T_i\right) \quad t = \text{fix } (f_i := t_i)_{i=1...h} \text{ for } f_j \land \text{ (real function, maybe closure)} \\
\left(\bigcup_{i=1}^{h} N_i \cup T_i - \{f_i, \ldots, f_h\}, \varnothing\right) \quad \neg(n = NA_t \land (\bigcup_{i=1}^{h} N_i \cap \{f_i, \ldots, f_h\} = \varnothing)) \text{ where} (R_i, N_i, T_i) = RNT[t_i]_{NA_{N_i}}
\]

Note:
- NA_t is the number of arguments of t: NA_t = m if t : T_1 \rightarrow \cdots \rightarrow T_m \rightarrow 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.

```c
static nat pow3(nat v1_x) {
    nat v2_n; nat v3_n; nat v4_n; nat v5_n;
    v2_n = 0();
    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.

```c
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 \( E[\Gamma] \vdash t \). Figure 21 defines \( E[\Gamma] \vdash t \) and \( E[\Gamma] \vdash t \). \( E[\Gamma] \vdash t \) determines that \( t \) is valid function for our C code generator. \( E[\Gamma] \vdash t \) 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 \( \text{Definition } c := t \), 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 => 0 | S m => uphalf m end
  with uphalf n :=
    match n with O => 0 | 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);
  }
}
```
• \(F[r]\) 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.

• \(BR[t]_{m,q}\) transforms a branch of a match expression into a nested abstraction expression that takes constructor members. Fixpoint expressions are not allowed.

• \(E[t / x_1 \ldots x_p]_q\) is a term convertible with \(t x_1 \ldots x_p\) that does not contain a partial application. \(E[t / x_1 \ldots x_p]_q\) 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.

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

where \(t : T_1 \rightarrow \cdots \rightarrow T_m \rightarrow T_0\)

- \(T_0\) is an inductive type
- \(x_1 \ldots x_m\) are fresh variables

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

where \(x_1 \ldots x_m\) are fresh variables

\[
E[t / x_1 \ldots x_p]_q = \begin{cases} 
x & t = x \land p = q = 0 \\
x \times_1 \ldots x_p & t = x \land \neg(p = q = 0) \land r = 0 \\
F[x / x_1 \ldots x_p] & t = x \land (p = 0 \land q > 0) \\
t \times_1 \ldots x_p & t = x \land (p = 0 \land q > 0) \\
F[t / x_1 \ldots x_p] & t = x \land (p = 0 \land q > 0) \\
F[r] & t = x \land p = q = 0 \\
E[u / x_1 \ldots x_p]_q & t = \lambda x. u \land (p = 0 \land q > 0) \lor r > 0 \\
\lambda x. E[u / p+q-1] & t = \lambda x. u \land (p = 0 \land q > 0) \lor r > 0 \\
E[u / x_1 \ldots x_p]_q & t = u \times_0 \\
\text{let } x := E[t_1 / x_1 \ldots x_p]_q \text{ in } E[t_2 / x_1 \ldots x_p]_q & t = \text{let } x := t_1 \text{ in } t_2 \text{ (zeta-app)} \\
\text{match } x \text{ with } (C_i \Rightarrow BR[t_i]_{NM(\text{lin},p+q)}))_{i=1 \ldots h} \text{ end } x_1 \ldots x_p & t = \text{match } x \text{ with } (C_i \Rightarrow t_i)_{i=1 \ldots h} \text{ end } x_1 \ldots x_p \\
(fix (f_i := F[t_i])_{i=1 \ldots h} \text{ for } f_j) & t = \text{fix } (f_i := t_i)_{i=1 \ldots h} \text{ for } f_j \\
\end{cases}
\]

where \(t : T_1 \rightarrow \cdots \rightarrow T_p \rightarrow T_{p+1} \rightarrow \cdots \rightarrow T_{p+q} \rightarrow T_{p+q+1} \rightarrow \cdots \rightarrow T_{p+q+r} \rightarrow T_0\)

- \(T_0\) is an inductive type

Note: This transformation assumes \(t\) is not dependently typed: no type terms and no dependent \texttt{match}-expressions.

**Figure 18.** Argument completion.

\[
x : \text{Gallina variable}
\]
\[
v : C \text{ variable}
\]
\[
V = \text{empty} \mid x \mapsto v \mid V; V
\]

**Figure 19.** Variable mapping from Gallina to C.

**F list bool Implementation**

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

**G buffer Implementation**

\texttt{buffer} type can be implemented in C as Figure 26. \texttt{unit} type is also implemented.

\texttt{make_buffer} allocates a buffer, \texttt{buf_addch} adds a character to the buffer destructively, and \texttt{free_buffer} deallocates the buffer.

\texttt{buf_addch} and \texttt{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.
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.

\[
\begin{align*}
CV[t / x_1 \ldots x_n]_V = \\
\begin{cases}
\text{empty} & t = x \lor t = c \lor t = C \lor t = T \\
CV[u / \forall: M; M] & t = \lambda x. u \land n = 0 \quad \text{where } M = x \mapsto v \\
CV[u / x_2 \ldots x_n]_V & t = \lambda x. u \land n > 0 \quad \text{where } M = x \mapsto V(x_1) \\
CV[u / x_0 x_1 \ldots x_n]_V & t = u x_0 \\
CV[t_1 / \forall: V; CV[t_2 / x_1 \ldots x_n]_V; M] & t = \text{let } x := t_1 \text{ in } t_2 \quad \text{where } M = x \mapsto v \\
(CV[t_1 / x_1 \ldots x_n]_V; \ldots; ) & t = \text{match } x \text{ with } (C_i \Rightarrow \lambda y_{i_1} \ldots \lambda y_{i_{NM_C_i}} \cdot t_i)_{i=1\ldots h} \text{ end} \\
CV[t_h / x_1 \ldots x_n]_V; M_h; & \quad \text{where } M_i = y_{i_{1}} \mapsto v_{i_{1}}; \ldots; y_{i_{NM_C_i}} \mapsto v_{i_{NM_C_i}} \\
\text{\textcolor{red}{M} \text{\textcolor{red}{t}}} & t = \text{fix } (f_i := t_i)_{i=1\ldots h} \text{ for } f_j \\
\text{\textcolor{red}{M} \text{\textcolor{red}{t}}} & \quad \text{where } M = f_i \mapsto v_{i_{1}}; \ldots; f_h \mapsto v_h \\
\end{cases}
\end{align*}
\]

where \( v, v_{ij}, v_{ij} \) are fresh C variables

Note: We consider Gallina variables unique.

**Figure 20.** C variable allocation.

\[
E[\Gamma] \vdash_b x \quad E[\Gamma] \vdash_b c \\
E[\Gamma] \vdash_b t \quad E[\Gamma] \vdash x : T 
T \text{ is a non-dependent inductive type} \\
E[\Gamma] \vdash_b t \quad E[\Gamma : (x := t : T)] \vdash_b u 
T \text{ is a non-dependent inductive type} \\
E[\Gamma] \vdash x : T 
T \text{ is a non-dependent inductive type with } p \text{ parameters: } u_1 \ldots u_p \\
E[] \vdash C_i u_1 \ldots u_p : T_i \rightarrow \cdots \rightarrow T_{i_{NM_C_i}} \rightarrow T 
T_{ij} \text{ are non-dependent inductive types} \\
E[\Gamma : (y_{i_{1}}; T_{i_{1}}); \cdots; (y_{i_{NM_C_i}}; T_{i_{NM_C_i}})] \vdash_b t_i \\
E[\Gamma] \vdash_b \text{match } x \text{ with } (C_i \Rightarrow \lambda y_{i_{1}} \ldots \lambda y_{i_{NM_C_i}} \cdot t_i)_{i=1\ldots h} \text{ end} \\
\]

**Figure 21.** The Gallina subset for C code generation.
GENFUN[^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 \( f_i \) … \( f_n \) are fixpoint bounded functions in t which are invoked as functions. We assume the types of them:

\[
\begin{align*}
c & : T_01 \to \cdots \to T_{0}\_m \to T_{0} \\
f_i & : T_{i1} \to \cdots \to T_{i}\_m \to T_{i0} & \text{ for } i = 1 \ldots n
\end{align*}
\]

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

The formal arguments of c are x\( _{01} \ldots x_{0}\_m = \text{fvars}[t] \) and the formal arguments of \( f_i \) are x\( _{1i} \ldots x_{i}\_m = \text{fvars}[f_i] \).

\( f_i \) invocation in C needs extra arguments, \( y_{1i} : U_{i1} \ldots y_{i}\_o : U_{io} \), 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.

\[
\text{GENFUN[^M][c]} = \text{"enum\_entries}[c]\ \text{arg\_structdefs}[c]\ \text{forward\_decl}[c]\ \text{entry\_functions}[c]\ \text{body\_function}[c]"
\]

\[
\begin{align*}
\text{enum\_entries}[c] & = \text{"enum\_enum\_func\_c\{func\_c,func\_f1,\ldots,func\_fn\};"} \\
\text{arg\_structdefs}[c] & = \"main\_structdef[c] aux\_structdef[c]_1 \ldots aux\_structdef[c]_n\" \\
\text{main\_structdef[c]} & = \"struct arg_c\{T_{0} arg1;\ldots T_{0}\_m argm0;\};\" \\
\text{aux\_structdef[c]}_i & = \"struct arg\_f_i\{U_{i1} outer1;\ldots U_{i}\_o outero_i; T_{i1} arg1;\ldots T_{i}\_m argm_i;\};\" \\
\text{forward\_decl}[c] & = \"static\ void body\_function\_c(enum\ enum\\_func\_c g, void *arg, void *ret);\" \\
\text{entry\_functions}[c] & = \"main\_function[c] aux\_function[c]_1 \ldots aux\_function[c]_n\" \\
\text{main\_function[c]} & = \"static\ T_{0}0 c(T_{01} x_{01}, \ldots, T_{0}\_m x_{0}\_m) \{ \\
& \quad \text{struct arg\_c arg} = \{x_{01}, \ldots, x_{0}\_m\}; T_{0} ret; \\
& \quad \text{body\_function\_c(func\_c, &arg, &ret); return ret; \} \" \\
\text{aux\_function[c]}_i & = \"static\ T_{i}0 \ f_i(U_{i1} y_{i1}, \ldots, U_{i}\_o y_{i}\_o, T_{i1} x_{i1}, \ldots, T_{i}\_m x_{i}\_m) \{ \\
& \quad \text{struct arg\_f}_i arg = \{y_{i1}, \ldots, y_{i}\_o, x_{i1}, \ldots, x_{i}\_m\}; T_{i} ret; \\
& \quad \text{body\_function\_c(func\_f}_i, &arg, &ret); return ret; \} \" \\
\text{body\_function[c]} & = \"static\ void body\_function\_c(enum\ enum\\_func\_c g, void *arg, void *ret)\{ \\
& \quad \text{decls} \\
& \quad \text{switch (g) \{ aux\_case[c]_1 \ldots aux\_case[c]_n main\_case[c] } \\
& \quad \text{GENBODY^K_B}[t]\} \" \\
\text{aux\_case[c]}_i & = \"\text{case func\_f}_i:\ \\
& \quad y_{i1} = ((\text{struct arg\_f}_i*arg)->outer1; \ldots; y_{i}\_o = ((\text{struct arg\_f}_i*arg)->outero_i; \\
& \quad x_{i1} = ((\text{struct arg\_f}_i*arg)->arg1; \ldots; x_{i}\_m = ((\text{struct arg\_f}_i*arg)->argm_i; \\
& \quad \text{goto entry\_f}_i;\" \\
\text{main\_case[c]} & = \"\text{default;:} \\
& \quad x_{01} = ((\text{struct arg\_c*arg})->arg1; \ldots; x_{0}\_m = ((\text{struct arg\_c*arg})->argm_0;\" \\
\end{align*}
\]

where \( \text{decls} \) is local variable declarations for variables used in GENBODY^K_B[t].

\( K(e) = \ast(T_{0}\_0*ret) = e; \text{return; }\"

**Figure 22.** Translation for a top-level function which is translated to multiple C functions.
GENFUN\textsuperscript{S}[c] translates the function (constant) c to a single C function.

\[
\text{GENFUN}\textsuperscript{S}[c] = \begin{cases} 
\text{"static } T_0 \ c(\ fargs'[t] ) \ ( \ \text{decls} \ \text{GENBODY}\textsuperscript{B}[t]) \" & \text{if } t \text{ needs multiple functions} \\
\text{GENFUN}\textsuperscript{S}[c] & \text{otherwise}
\end{cases}
\]

where \(c\) is defined as

\[
\text{Definition } c : T_1 \to \cdots \to T_n \to T_0 := t.
\]

\(T_0\) is an inductive type

\(\text{decls}\) is local variable declarations for variables used in \(\text{GENBODY}\textsuperscript{B}[t]\) excluding \(\text{fargs}[t]\).

\(K(e) = \text{"return } e;"\)

\(\text{fargs}[t] = \begin{cases} 
\text{"T x, fargs'[u]"} & t = \lambda x: T. u \\
\text{"fargs'[t_j]"} & t = \text{fix } (f_i := t_i)_{i=1...k} \text{ for } f_j \\
\text{""} & \text{otherwise}
\end{cases}
\)

\(\text{fargs}'[t] = \text{fargs}[t]\) without the trailing comma

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

**Figure 24.** Translation for top-level function.

```c
#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.

```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


