A Gallina Subset for C Extraction of Non-structural Recursion

Akira Tanaka ⟨tanaka-akira aist.go.jp⟩
National Institute of Advanced Industrial Science and Technology (AIST)

Motivation and Approach: We want a Gallina-to-C translator that is trustworthy and that produces realistic C code which uses loop structures without stack consumption and doesn't use polymorphic types.

One particular technical problem is how to handle non-structural recursion. It is required for realistic algorithms such as breadth-first search. Previous work does not handle non-structural recursion so as to generate realistic C code. For example, Œuf [2] is another Gallina-to-C translator in Gallina. It uses recursion schemes, such as `nat_rect`, to describe recursion and pattern matching. This does not lead to realistic C code.

In this talk, we propose an approach to handle non-structural recursion. Our approach is similar to Œuf in that it uses a GADT-style typed AST to reflect Gallina terms. However, we do that in such a way as to handle non-structural recursion. Let us use a running example to explain how we handle non-structural recursion. This loop iterates `i` to `N` increasingly: `for (; i < N; i++) {} return true;`

In Gallina, the above program corresponds to a non-structural recursion. It requires dependent types such as `well_founded`. We assume a user writes our example as follows (`N` is a parameter; `R x y` is the relation: `N - x < N - y`; `Rwf` is a proof of `well_founded R`):

```
Definition upto_F (i : nat) (f : ∀ (i' : nat), R i' i → bool) : bool.  Proof.  refine (
    match i < N as b' return (b' = (i < N)) → bool with
    | false ⇒ fun (H : false = (i < N)) ⇒ true
    | true ⇒ fun (H : true = (i < N)) ⇒ f i.+1 _(* hole of type R i.+1 i *)
    end erefl).  ... proof for R i.+1 i snipped ...  Defined.
Definition upto := Fix Rwf (fun _ ⇒ bool) upto_F.
```

We can translate `upto` to a C-friendly form by reducing `Fix`. `Fix` of the standard library `Coq.Init.Wf` generates a recursive function: `upto` is a term convertible with `fun x ⇒ (fix upto_rec i a := upto_body upto_rec i a) x (Rwf x)`. `upto_body` represents the body of the `fix`-term which can be defined as follows (the proof part is separated as `upto_lemma`).

```
Definition upto_body (upto_rec : ∀ (i : nat), Acc R i → bool) (i : nat) (a : Acc R i) : bool :=
  let n := N in let Hn : n = N := erefl in let b := i < n in let Hb : b = (i < n) := erefl in
  match b as b' return b' = b → bool with
  | false ⇒ fun Hm : false = b ⇒ true
  | true ⇒ fun Hm : true = b ⇒ let j := i.+1 in let Hj : j = i.+1 := erefl in
      let a' := upto_lemma i n b j a Hn Hb Hm Hj in upto_rec j a'
  end erefl.
```

In the rest of this abstract, we explain the design of a GADT-style AST with an evaluator with which one can reflect non-structurally recursive Gallina programs so as to generate realistic C code. Using our AST, it will be possible to represent `upto_body` in an inductive type.

1 GA to Represent Non-structural Recursion

We define an intermediate language, GA (Gallina A-normal form). It corresponds to a Gallina subset which includes dependent type for non-structural recursion. In Coq, it is represented by an inductive type. Here, we explain only the syntax informally.

We use GA to represent the body of a `fix`-term because an AST for `fix`-term is not evaluable[1]. GA is A-normal form: arguments of an application must be local variables. This makes the evaluation order of a GA expression explicit[2]. The body of `upto_body` (i.e. `upto_body` without its arguments) is described in GA as follows. `upto_rec`, `i` and `a` are the arguments of `upto_body`.

```
leta n Hn := N in leta b Hb := ltn i n in
  dmatch b with
  | false Hm ⇒ true
  | true Hm ⇒ leta j Hj := S i in letp a' := upto_lemma i n b j a Hn Hb Hm Hj in upto_rec j a'
  end
```

[1] An evaluator for `fix`-term is not termination checkable.

[2] Another critical reason to use A-normal form is that the evaluator is not typable without A-normal form. The type checker cannot unfold the evaluator (recursive function) when type checking the evaluator. A-normal form avoids appearance of the evaluator in a dependent type.

Most constructs of GA correspond directly to Gallina but GA distinguishes non-dependent type and proof (dependent type) syntactically, hence the variants `leta`, `letp`, etc. Functions also have three variants: global functions (`ltn`, `S`, `true` and `N`) which are of non-dependent type; lemmas (`upto_lemma`) which are of dependent type; and recursive functions (`upto_rec`) whose arguments can be dependently typed but whose return type is non-dependent type.

`leta` and `dmatch` correspond to `let` and `match` but have extra binders for equality proofs. The equality for `leta` is an equality between the bound variable and its definition. The equality for `dmatch` is an equality between the constructor form and the match item. `leta b Hb := ltn i n in E` corresponds to `let b := ltn i n in let Hb : b = ltn i n := erefl in E` in Gallina. `Hb`, whose type is `b = ltn i n`, is used to rewrite `b` to its definition. `dmatch b with | true H1 ⇒ E1 | false H2 ⇒ E2 end`, corresponds to `match b as b' return b' = b → T with | true ⇒ fun H1 ⇒ E1 | false ⇒ fun H2 ⇒ E2 end erefl` which uses the convoy pattern, and where `T` is the type of the `dmatch` expression. `H1`, whose type is `true = b`, is used to rewrite `b` to its constructor form.

Translating Gallina to GA is easy because we require a source Gallina term convertible with a term in the Gallina subset defined by GA: polymorphic types are removed by specializing functions with respect to type arguments; ζ-expansion inserts `let` to make a term A-normal form.

2 Coq Implementation of GA and its Evaluator

We have implemented GA in the form of an inductive type. Let us explain this type using a concrete example: the body of `upto_body`.

```
...
Let nT2 := [:: (*n*)nat; (*i*)nat].
Let pT2 := [:: (*Hn*)fun (genv : genviron GT5) (nenv : nenviron nT2) ⇒ (*n*)nenv.1 = glookup GT5 genv "N" tt;
            (*a*)fun (_ : genviron GT5) (nenv : nenviron nT2) ⇒ Acc R (*i*)nenv.2.1].
...
Definition upto_body_AST : exp GT5 LT5 rT nT1 pT1 bool :=
 leta GT5 LT5 rT nT1 pT1 nat bool (* n Hn := *) "N" [::] erefl                      (* leta n Hn := N in        *)
  (leta GT5 LT5 rT nT2 pT2 bool bool (* b Hb := *) "ltn" [:: (*i*)1; (*n*)0] erefl  (* leta b Hb := ltn i n in  *)
   (dmatch GT5 LT5 rT nT3 pT3 bool (*b*)0                                           (* dmatch b with            *)
    (dmatch_cons GT5 LT5 rT nT3 pT3 (*b*)0 bool (* | false Hm *) bool_false [::]    (* | false Hm ⇒             *)
     (app GT5 LT5 rT nT3 pT4 bool "true" [::] erefl)                                (*   true                   *)
     (dmatch_cons GT5 LT5 rT nT3 pT3 (*b*)0 bool (* | true Hm *) bool_true [:: bool_false]  (* | true Hm ⇒      *)
      (leta GT5 LT5 rT nT3 pT5 nat bool (* j Hj := *) "S" [:: (*i*)2] erefl         (*   leta j Hj := S i in    *)
       (letp GT5 LT5 rT nT4 pT6 bool (* a' := *) "upto_lemma" upto_lemma_P          (*   letp a' := upto_lemma  *)
        [:: (*j*)0; (*b*)1; (*n*)2; (*i*)3] [:: (*Hj*)0; (*Hm*)1; (*Hb*)2; (*Hn*)3; (*a*)4]  (*     i n b j a Hn Hb Hm Hj in *)
        upto_lemma_pt upto_lemma_P erefl erefl erefl
        (rapp GT5 LT5 rT nT4 pT7 bool "upto_rec" [:: (*j*)0] [:: (*a'*)0] rtyF erefl erefl)))  (*   upto_rec j a'  *)
      (dmatch_nil GT5 LT5 rT nT3 pT3 (*b*)0 bool [:: bool_true; bool_false] bool_matcher))))).  (* end          *)
```

The inductive type for GA AST has five environment type parameters and an expression type as an index. There are many environments because we want to distinguish variables to perform proof elimination syntactically. Dependently typed variables are in `pT`n and `LT`n, non-dependently typed variables are in `nT`n and `GT`n, `rT` is for recursive functions (they require special treatment). Moreover, we have both local and global variables. The local variables use de Bruijn indices: they will be generated automatically inside the translator and invisible to the user.

The main technical features of this type are: (1) We represent a **dependent type** as a function from global and local non-dependent environment to a proposition. For example, `pT2` contains `n = N` and `Acc R i`. (2) We use a **matcher** to represent a pattern matching because `match`-expressions in Gallina are not type-generic. Therefore, we define a matcher for each inductive type, such as `bool_matcher` for `bool`. It contains a function containing just a `match`-expression. `eval` invokes the function for pattern matching. (3) We verify the AST as `upto_body` = `eval upto_body_AST`. Since this evaluation of GA AST is convertible with the original Gallina term, `reflexivity` is enough to prove it.

After the verification, we import the recursive function into the global environment. Repeating this translation, we can reflect the whole program into the GA AST[3].

The translation from GA to C should be easy. Proof elimination can be implemented syntactically. After proof elimination, GA constructs should be translated to C directly: application to function call, `dmatch` to `switch`, `let` to variable initialization. One non-trivial translation is tail-recursion using `goto` to avoid stack consumption. It should be compiled similar as a usual loop structure by a SSA-based optimizing compiler.

Conclusion: GA is usable to reflect a Gallina term to a GADT-style AST and its verification. We plan to rewrite codegen [1] to use GA. We will tackle BFS and its various applications including succinct data structures.

Acknowledgements: This work is supported by JSPS-CNRS Bilateral Program, "FoRmal tools for IoT sEcurity".

References

[1] codegen plugin for Coq, ⟨`https://github.com/akr/codegen/`⟩.

[2] Mullen, E., et al. Œuf: minimizing the Coq extraction TCB. Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. (2018).

[3] Full version of this abstract including the syntax of GA, ⟨`https://staff.aist.go.jp/tanaka-akira/succinct/`⟩.

[3] We import the original recursive function because the AST-based one (`fix (eval AST)`) is too complex for the termination checker.