A Gallina Subset for C Extraction of Non-structural Recursion

Akira Tanaka
National Institute of Advanced Industrial Science and Technology (AIST)
2019-09-08

Outline

- Our C code generator and its problem
- Verification of AST including NSR (Non-Structural Recursion)
- Example of translation of NSR and GA (GA: our intermediate language)
- Technical details of GA
- Conclusion

Our Idea

We want a simple C code generator for Coq

- Gallina and C have similar constructs
  - Both have variables
  - "let" = "variable initialization"
  - "function application" = "function call"
  - "match expression" = "switch statement"
- It should be possible to translate from a first-order subset of Gallina to a subset of C

More Practical than It Seems at First

- We implemented the CODEGEN plugin for Coq
  https://github.com/akr/codegen
- It can generate practical code
  - HTML escape method usable from Ruby
    Ruby Extension Library Verified using Coq Proof-assistant, RubyKaigi 2017
  - rank algorithm for succinct data structures
    Safe Low-level Code Generation in Coq using Monomorphization and Monadification, IPSJ-SIGPRO, 2017-06-09
- They are as fast as hand-written C code

Wanted Features for Practical Code Generation

- Monomorphization to support polymorphism
- Customizable inductive type implementation
  - (Monomorphized) Gallina types directly mapped to C types
    - Native C types can be used: 64 bit integer, 128 bit SSE type, etc.
  - Constructors are implemented in C
  - switch statements can be customized for each inductive type
- Gallina functions are directly mapped to C functions
- Primitive functions are implemented in C;
  a "primitive function" can be macro which can use
  - Binary/unary operators
  - Compiler builtin such as SSE intrinsics
- Loop without stack consumption
  - Tail recursions are translated to goto
- Destructive update can be used with linearity checker

Problems of Current CODEGEN

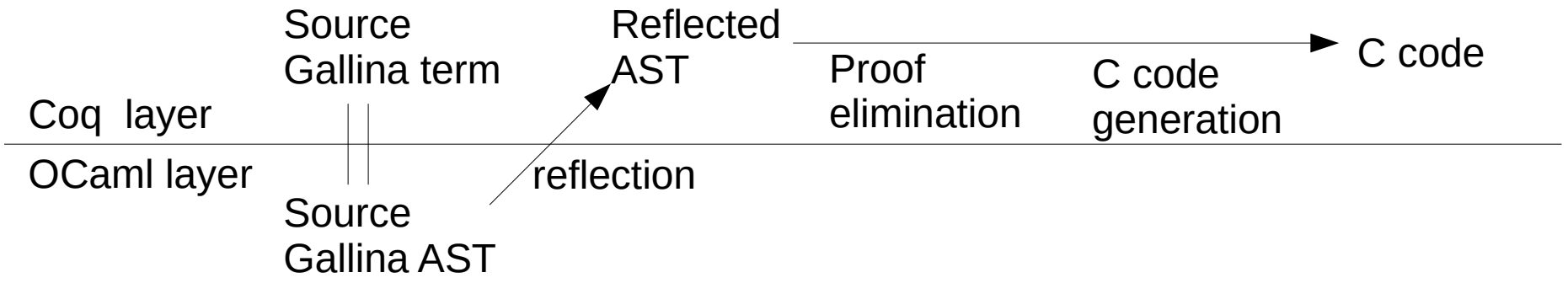
- Non-structural recursion (NSR) is not supported because no proof elimination
  - Some algorithms, such as breadth-first search, needs NSR
- CODEGEN itself is not verified
  - It is implemented in OCaml

Our Plan for the Future CODEGEN2

- Support NSR
  This requires limited support for proof elimination
- Rewrite most of code generation in Gallina;
  This makes possible to verify CODEGEN itself

Today's topic:
Reflecting Gallina terms with NSR

Structure of Future CODEGEN2



- Most of the translation is implemented in Gallina;
  It is verifiable in Coq
- Source Gallina term reflection still needs OCaml
  (out of scope of verification)

Outline

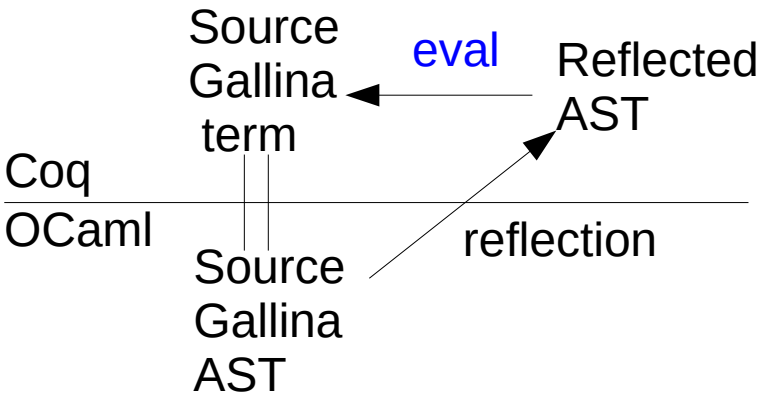
- Our C code generator and its problem
- Verification of AST including NSR
- Example translation of NSR and GA
- Technical details of GA
- Conclusion

Verification of the Reflection

We verify the result of the reflection

verification: term == eval AST

This verification doesn't need to verify OCaml code
cf. Œuf [Mullen2018]



Question:
Is it possible to represent a non-structural recursive term as an AST and verify it?

Answer:
Possible, for a Gallina subset we define

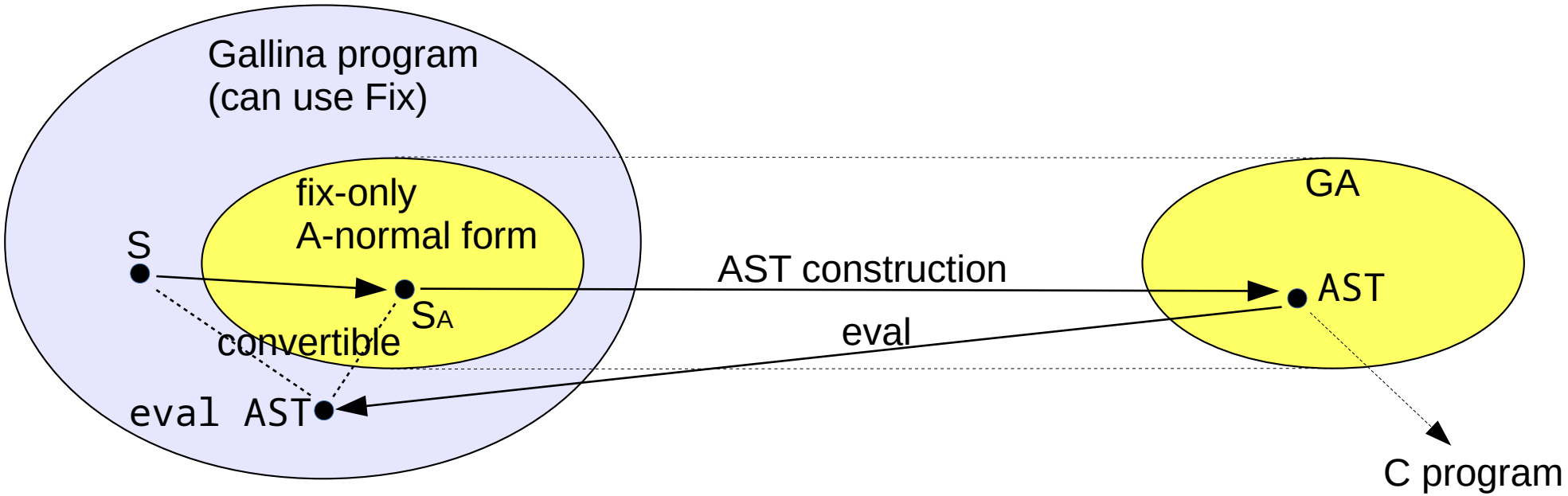
NSR using Coq.Init.Wf.Fix

- Fix constructs fix-term with a decreasing argument
- Fix is useful to prove NSR programs, assuming the axiom of functional extensionality
- Fix supports only one argument
- Multiple arguments version of Fix (Fix2, Fix3, ...) can be defined
  (it needs a Gallina extension, primitive projections, for reasoning)

Our Strategy for NSR

- User writes a source NSR program using Fix, Fix2, ...
- Automatic convertible translation
  - Expands Fix, Fix2, ... to fix (delta-reduction)
  - Simplify (beta-reduction)
  - Insert let for A-normal form (zeta-expansion)
- Construct an AST from an A-normal term
  (A-normal form: function arguments are local variables)
  - We define an intermediate language, GA, for this AST
- Verify the AST by checking that evaluation of the AST is convertible to the source program

AST Verification Structure



- S: Source program
- S_A: A-normal form of Fix-expanded S
- S, S_A and `eval AST` are convertible
- GA is implemented as inductive types
- Verification of "S_A is convertible with `eval AST`" means "`AST` represents S correctly"

Outline

- Our C code generator and its problem
- Verification of AST including NSR
- Example translation of NSR and GA
- Technical details of GA
- Conclusion

Example with NSR in C (Generated Code)

We want to generate C code like:

```
L:
switch (i < N) {
  case false:
    return true;
  default:
    i = i + 1;
    goto L;
}
```

```
same as:
for (; i < N; i++) {}
return true;
```

This loop increases i to N which needs NSR in Gallina

Example in Gallina (Source Code)

We can represent the increasing loop in Gallina using Fix as:

```
Definition upto_F (i : nat) (f : forall (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.
```

Coq.Init.Wf.Fix

- N is a parameter
- R x y is the relation: N - x < N - y
- Rwf is a proof of well_founded R

Use of Fix is appropriate for proof but not C-friendly because the recursion structure is embedded in Fix

C-Friendly Form of the Example (Intermediate Representation)

- upto (previous slide) and upto_noFix (this slide) are convertible

```
Definition upto_body (upto_rec : forall (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.
Definition upto_noFix :=
  fun x => (fix upto_rec i a := upto_body upto_rec i a) x (Rwf x)
```

- Uses fix instead of Fix: the recursion structure is explicit
- A-normal form
- proof part is separated as upto_lemma
- Limited use of higher order function

GA: Gallina A-normal Form

- We define GA to represent C-friendly Gallina term simply
  GA defines a subset of Gallina
- GA provides only "variable", "application", "match" and "let" which can be translated to C easily
- GA distinguishes non-dependent types and dependent types to implement syntactic proof elimination
- GA is an intermediate language. Users don't see it

Example in GA (Intermediate Representation)

The body of upto_body described in GA:

```
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
```

GA supports non-structural recursion by equality proof constructing "match" and "let"

GA Describes Function Body

C-friendly Gallina program:

```
Definition upto_body
  (upto_rec : forall (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.
```

The function body described in GA:

```
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
```

Syntax of GA

f : global function name
r : recursive function name
h : lemma name

v : non-dependent type local variable
p : proof variable (dependent type)
C : constructor

app = f v_1 ... v_n global function application
rapp = r v_1 ... v_n p_1 ... p_m recursive function application
papp = h v_1 ... v_n p_1 ... p_m lemma application

exp = v
| app
| rapp
| nmatch v_0 with (| C_i v_{i1} ... v_{im_i} => exp)$_{i=1...n}$ end
| dmatch v_0 with (| C_i v_{i1} ... v_{im_i} p_i => exp)$_{i=1...n}$ end $p_i : C_i\ v_{i1} \ldots v_{im_i} = v_0$
| leta v p := app in exp $p : v = app$
| letr v := rapp in exp
| letp p := papp in exp
| letn v := match v_0 with (| C_i v_{i1} ... v_{im_i} => exp)$_{i=1...n}$ in exp
| letd v := match v_0 with (| C_i v_{i1} ... v_{im_i} p_i => exp)$_{i=1...n}$ in exp $p_i : C_i\ v_{i1} \ldots v_{im_i} = v_0$

Correspondence of GA and Gallina

- GA distinguishes non-dependent types and dependent types to implement syntactic proof elimination
- Most constructs correspond directly:
  - Gallina: application GA: app, rapp, papp
  - Gallina: variable GA: v, p, r
  - Gallina: constant GA: f, h
  - Gallina: constructor GA: C
  - Gallina: match GA: nmatch, dmatch
  - Gallina: let GA: letr, letp, leta
  - Gallina: let & match GA: letn, letd
- dmatch and leta bind equality proof
  - Gallina: match b as b' return b' = b -> T with
    | true => fun H1 => E1 | false => fun H2 => E2 end erefl (convoy pattern)
    GA: dmatch b with | true H1 => E1 | false H2 => E2 end
  - Gallina: let b := ltn i n in let Hb : b = ltn i n := erefl in E
    GA: leta b Hb := ltn i n in E

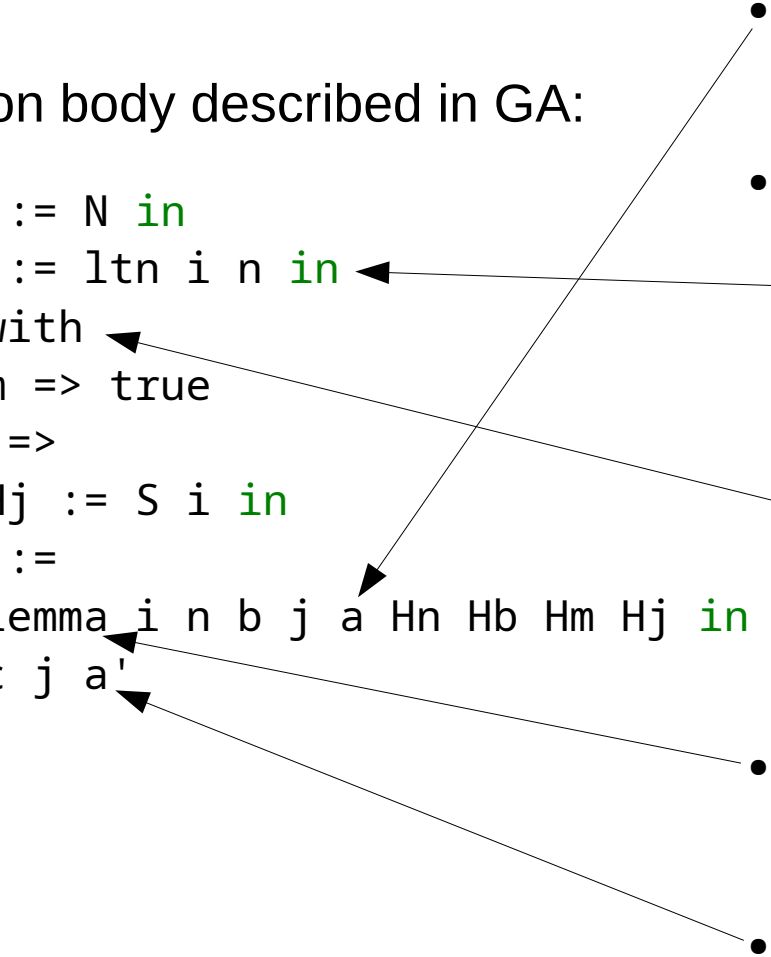
Design of GA

- Decreasing argument construction for NSR
- Syntactic proof elimination

Decreasing Argument Construction

The function body described in GA:

```
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
```



- A decreasing argument is given as a proof variable: a
- Construct proofs for context
  - "leta v p := f args in exp" constructs a proof of "v = f args"
  - "dmatch v_0 with ... end" constructs a proof of "C_i args = v_0" for each i
- Invoke a lemma to construct a smaller decreasing argument
- Use the smaller decreasing argument for a recursive call

Syntactic Proof Elimination of GA

f : global function name
r : recursive function name
h : lemma name

v : non-dependent type local variable
p : proof variable (dependent type)
C : constructor

app = f v_1 ... v_n — global function application
rapp = r v_1 ... v_n p_1 ... p_m — recursive function application
papp = h v_1 ... v_n p_1 ... p_m — lemma application

exp = v
| app
| rapp
| nmatch v_0 with (| C_i v_{i1} … v_{im_i} => exp)$_{i=1...n}$ end
| dmatch v_0 with (| C_i v_{i1} … v_{im_i} p_i => exp)$_{i=1...n}$ end — $p_i : C_i\ v_{i1} \ldots v_{im_i} = v_0$
| leta v p := app in exp — p : v = app
| letr v := rapp in exp
| letp p := papp in exp
| letn v := match v_0 with (| C_i v_{i1} … v_{im_i} => exp)$_{i=1...n}$ in exp
| letd v := match v_0 with (| C_i v_{i1} … v_{im_i} p_i => exp)$_{i=1...n}$ in exp — $p_i : C_i\ v_{i1} \ldots v_{im_i} = v_0$

Syntactic proof elimination is possible
because proofs are distinguished by syntax

Syntactic Proof Elimination of GA

f : global function name
r : recursive function name
h : lemma name

v : non-dependent type local variable
p : proof variable (dependent type)
C : constructor

app = f v_1 ... v_n — global function application
rapp = r v_1 ... v_n p_1 ... p_m — recursive function application
papp = h v_1 ... v_n p_1 ... p_m — lemma application

exp = v
| app
| rapp
| nmatch v_0 with (| C_i v_{i1} … v_{im_i} => exp)$_{i=1...n}$ end
| dmatch v_0 with (| C_i v_{i1} … v_{im_i} p_i => exp)$_{i=1...n}$ end — p_i : C_i v_{i1} … v_{im_i} = v_0
| leta v p := app in exp — p : v = app
| letr v := rapp in exp
| letp p := papp in exp
| letn v := match v_0 with (| C_i v_{i1} … v_{im_i} => exp)$_{i=1...n}$ in exp
| letd v := match v_0 with (| C_i v_{i1} … v_{im_i} p_i => exp)$_{i=1...n}$ in exp — p_i : C_i v_{i1} … v_{im_i} = v_0

Syntactic proof elimination is possible
because proofs are distinguished by syntax

Syntactic Proof Elimination Example

The function body described in GA:

```
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
```

GA AST Example

- We implement GA as inductive types in Coq
- The body of upto_body can be represented as an AST

```
...
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                       *)
```

Inductive Types for GA

```
Section Exp_Global_Parameter.
Variables (gT : genvtype)
  (lT : lenvtype gT) (rT : renvtype gT).

Inductive papp_exp (nT : nenvtype)
    (pT : penvtype gT nT)
    (P : pty gT nT) : Type :=
| papp_expC : forall (name : string)
    (nargs pargs : seq nat)
    (nT' := map (ntnth nT) nargs)
    (pT' : penvtype gT nT')
    (P' : pty gT nT')
    (H1 : ltyC gT nT' pT' P' =
            ltlookup gT lT name)
    (H2 : transplant_pt gT nT nargs pT' =
            map (ptnth gT nT pT) pargs)
    (H3 : transplant_pty gT nT nargs P' = P),
    papp_exp nT pT P.

Inductive rapp_exp : Type :=
| rapp_expC : forall (name : string)
    (nargs : seq nat) (pargs : seq nat)
    (nT' := map (ntnth nT) nargs)
    (pT' : penvtype gT nT')
    (H1 : rtyC gT nT' pT' Tr =
            rtlookup gT rT name)
    (H2 : transplant_pt gT nT nargs pT' =
            map (ptnth gT nT pT) pargs),
    rapp_exp.
```

```
Inductive exp (nT : nenvtype) (pT : penvtype gT nT) : Set ->
Type :=
| var : forall (i : nat), exp nT pT (ntnth nT i)
| app : forall (Tr : Set) (name : string) (nargs : seq nat)
    (nT' := map (ntnth nT) nargs)
    (H : gtyC nT' Tr = gtlookup gT name),
    exp nT pT Tr
| leta : forall (Tv Tr : Set) (name : string)
    (nargs : seq nat)
    (nT' := map (ntnth nT) nargs)
    (H : gtyC nT' Tv = gtlookup gT name)
    (Peq : pty gT (Tv :: nT) :=
      leta_eq nT Tv name nargs H),
    exp (Tv :: nT) (Peq :: pt_shift0 gT Tv nT pT) Tr ->
    exp nT pT Tr
| rappC : forall (Tr : Set), rapp_exp nT pT Tr ->
    exp nT pT Tr
| letrC : forall (Tv Tr : Set), rapp_exp nT pT Tv ->
    exp (Tv :: nT) (pt_shift0 gT Tv nT pT) Tr ->
    exp nT pT Tr
| letpC : forall (Tr : Set) (P : pty gT nT),
    papp_exp nT pT P ->
    exp nT (P :: pT) Tr -> exp nT pT Tr
| nmatch : forall (Tr : Set) (i : nat) (Tv := ntnth nT i),
    nmatch_exp nT pT Tv Tr [::] -> exp nT pT Tr
| letn : forall (Tv2 Tr : Set) (i : nat) (Tv1 := ntnth nT i),
    nmatch_exp nT pT Tv1 Tv2 [::] ->
    exp (Tv2 :: nT) (pt_shift0 gT Tv2 nT pT) Tr ->
    exp nT pT Tr
| dmatch : forall (Tr : Set) (i : nat),
    dmatch_exp nT pT i Tr [::] -> exp nT pT Tr
| letd : forall (Tv2 Tr : Set) (i : nat),
    dmatch_exp nT pT i Tv2 [::] ->
    exp (Tv2 :: nT) (pt_shift0 gT Tv2 nT pT) Tr ->
    exp nT pT Tr
```

```
with nmatch_exp (nT : nenvtype)
    (pT : penvtype gT nT) :
    forall (Tv : Set) (Tr : Set)
    (Cts : seq (cstrT Tv)), Type :=
| nmatch_nil : forall (Tv Tr : Set)
    (Cts : seq (cstrT Tv)),
    Matcher Tv Cts -> nmatch_exp nT pT Tv Tr Cts
| nmatch_cons : forall (Tv Tr : Set) (Ct : cstrT Tv)
    (Cts : seq (cstrT Tv))
    (Ts : seq Set := Tms4Ct Tv Ct),
    let nT' := nt_shift0s Ts nT in
    let pT' := pt_shift0s gT Ts nT pT in
    exp nT' pT' Tr ->
    nmatch_exp nT pT Tv Tr (Ct :: Cts) ->
    nmatch_exp nT pT Tv Tr Cts
with dmatch_exp (nT : nenvtype)
    (pT : penvtype gT nT) :
    forall (i:nat) (Tv := ntnth nT i) (Tr : Set)
    (Cts : seq (cstrT Tv)), Type :=
| dmatch_nil : forall (i : nat) (Tv := ntnth nT i)
    (Tr : Set) (Cts : seq (cstrT Tv)),
    Matcher Tv Cts -> dmatch_exp nT pT i Tr Cts
| dmatch_cons : forall (i : nat) (Tv := ntnth nT i)
    (Tr : Set) (Ct : cstrT Tv) (Cts : seq (cstrT Tv))
    (Ts : seq Set := Tms4Ct Tv Ct)
    (c : arrow_ntS Ts Tv := cstr4Ct Tv Ct),
    let nT' := nt_shift0s Ts nT in
    let pT' := (dmatch_pty gT nT i Ts c) ::
                  (pt_shift0s gT Ts nT pT) in
    exp nT' pT' Tr ->
    dmatch_exp nT pT i Tr (Ct :: Cts) ->
    dmatch_exp nT pT i Tr Cts.
```

Outline

- Our C code generator and its problem
- Verification of AST including NSR
- Example translation of NSR and GA
- Technical details of GA
- Conclusion

Technical Details of GA AST

- Environments
- Representation of dependent types
- Representation of match-expression
- AST design for evaluator
  - Evaluator cannot interpret fix-term
  - A-normal form
- Verification of AST by roundtrip
- Translation of recursive functions

Technical Details of GA AST

- Environments
- Representation of dependent types
- Representation of match-expression
- AST design for evaluator
  - Evaluator cannot interpret fix-term
  - A-normal form
- Verification of AST by roundtrip
- Translation of recursive functions

Environments

Purpose of these many environments:

- Non-dependent environments and dependent environments for
  - Syntactic proof elimination
  - Dependent types representation
- Global environments for functions and local environments for values

AST has 5 environment parameters:

- genv: global non-dependent environment
- nenv: local non-dependent environment
- lenv: global dependent (lemma) environment
- penv: local dependent (proof) environment
- renv: recursive function environment
  (arguments can be dependently typed but return type is non-dependent)

Representation of Dependent Types

A dependent type is represented as a function from genv and nenv to a proposition

- Gallina: `Acc R i`
  GA AST:
```
fun (_ : genviron GT5) (nenv : nenviron nT2) =>
                    Acc R (*i*)nenv.2.1
```

- Gallina: `n = N`
  GA AST:
```
fun (genv : genviron GT5) (nenv : nenviron nT2) =>
                  (*n*)nenv.1 = glookup GT5 genv "N" tt
```

GA AST uses de Bruijn index for local variables
An environment is nested pairs: nenv is `(n,(i,tt))`

Outline

- Our C code generator and its problem
- Verification of AST including NSR
- Example translation of NSR and GA
- Technical details of GA
- Conclusion

Conclusion

- We defined GA to represent a Gallina subset including non-structural recursion
  GA can be defined as inductive types in Coq
- We defined an evaluator for GA
  The evaluator is usable to verify AST
- We can translate Gallina term with non-structural recursion into GA AST in a trustworthy way
- Future Work
  - Automatic generation of GA AST
  - Proof elimination and C code generation

Extra Slides

Representation of match-expression

Problem: the evaluation of an AST must be convertible to the original term but evaluator can not implement match-expression in type-generic fashion

- We define a matcher for each inductive type
- AST embeds a matcher
- Evaluator invokes a matcher for match-expressions in AST

```
Definition bool_nmatcher
    (Tr : Set) (v : bool)
    (branch_true : Tr) (branch_false : Tr) : Tr :=
  match v with
  | true => branch_true
  | false => branch_false
  end.
```

```
Definition bool_dmatcher
    (Tr : Set) (v : bool)
    (branch_true : true = v -> Tr)
    (branch_false : false = v -> Tr) : Tr :=
  match v as v' return v' = v -> Tr with
  | true => branch_true
  | false => branch_false
  end erefl.
```

```
Definition bool_matcher := mkMatcher bool [:: bool_true; bool_false]
  bool_nmatcher bool_dmatcher.
```

AST Design for Evaluator

- Evaluator for fix-term is not termination checkable
- AST must be A-normal form to avoid a dependent type problem in evaluator

Evaluator for fix-term

- Problem: evaluator for fix-term is not termination checkable
  - fix-term: fix f arg := body
  - Conceptual evaluator implementation:
    eval [fix f arg := body] =
      fix f arg := eval body
  - There is no information about body when defining the eval function
    So, termination is not checkable
- Our solution
  GA AST represents only a body of a function
  fix-term is constructed outside of AST

A-normal Form

- Reasons for A-normal fom
  - Make evaluation order explicit
  - Solve dependent type problem in evaluator
- Gallina has no specific evaluation order
  A-normal form AST makes it possible to prove evaluation order dependent properties such as complexity
- A-normal form avoids dependent typing problem
  - Assume
    f : forall (x:A), B x
    v : A
    lookup [f] = f
    lookup [v] = v
    lookup_type [f] = forall (x:A), B x
    eval_type [f] = lookup_type [f] = forall (x:A), B a
    eval_type [v] = A
    eval : forall (e:Exp), eval_type e
  - A-normal form: eval [f v] = (lookup [f]) (lookup [v]) : (lookup_type [f]) (lookup [v])
  - non A-normal form: eval [f v] = (eval [f]) (eval [v]) : (eval_type [f]) (eval [v])
  - eval itself appear in the later type
  - When type-checking the recursive function eval for non-A-normal form,
    - eval is not reducible
    - eval [v] is not convertible with v
    - The type of eval [f v] is not convertible with the type of (f v)

eval in type

Verification of AST by Roundtrip

- We can verify the correctness of AST
  It needs a bit lengthy set up of environments
- Definition upto_body' (upto_rec : forall (i : nat), Acc R i -> bool)
    (i : nat) (acc : Acc R i) : bool :=
  let gT := GT5 in let genv := GENV5 in
  let lT := LT5 in let lenv := LENV5 in
  let nT := [::(*i*)nat] in
  let pT := [:: (fun (genv : genviron gT) (nenv : nenviron nT) => Acc R (*i*)nenv.1)] in
  let rT : renvtype gT := ("upto_rec", rtyC gT nT pT bool) :: RT5 in
  let renv : renviron gT rT genv :=
    ((uncurryR gT nT pT bool genv upto_rec), RENV5) in
  let nenv : nenviron nT := [nenv: i] in
  let u := uncurry_pty gT nT in
  let pT : penvtype gT nT := [:: (u (fun genv i => Acc R i))] in
  let penv : penviron gT nT pT genv nenv := (acc,I) in
  let Tr := bool in
  eval gT lT rT nT pT Tr genv lenv renv nenv penv upto_body_AST.
  Lemma upto_body_ok : upto_body = upto_body'. Proof. reflexivity. Qed.
- So, upto_body_AST represents upto_body correctly

Translation of Recursive Functions

- We translate the inside and outside of fix-term separatedly:
- Separated source definition:
  Definition upto_body upto_rec i a := ...
  Fixpoint upto_rec i a := upto_body upto_rec i a
  Definition upto i := upto_rec i (Rwf i)
- Translation steps
  1. Generate AST of upto_body and verify it
  2. Import upto_rec into renv (because "a" is dependently typed)
  3. Generate AST of upto and verify it
  4. Import upto into genv
- Now, we have verified AST of upto function

We Import the Original Funciton, Instead of AST-based One

- Problem: AST-based recursive function is not termination checkable
- upto_body and upto_body' are convertible
  Lemma upto_body_ok : upto_body = upto_body'. Proof. reflexivity. Qed.
- upto_rec using upto_body has no problem
  Fixpoint upto_rec (i : nat) (a : Acc R i) {struct a} : bool :=
    upto_body upto_rec i a.
- But AST-based one, upto_body', is too complex for termination checker
  Fail Fixpoint upto_rec'' (i : nat) (a : Acc R i) { struct a } :=
     upto_body' upto_rec'' i a.
  (* Recursive definition of upto_rec'' is ill-formed. *)
- The normal form of upto_body' has no problem
  Definition upto_body'' := Eval cbv in upto_body'.
  Fixpoint upto_rec'' i a := upto_body'' upto_rec'' i a.
- We import upto_rec instead of upto_rec'' because they should be convertible and the normal form can be very big term

Œuf

- Gallina frontend for CompCert
- Heavy weight compiler
- Recursor for pattern matching and recursion
- No NSR
- Verify AST by term = eval AST

CertiCoq

- Heavy weight compiler
- Uses Template-Coq for reflection
  - Template-Coq is not typed reflection
  - Need to re-implement Coq type system in Coq

Translation from Gallina to GA

1. Expand specified definitions such as "Fix"
2. Simplify by beta-reduction
   (zeta-expansion to avoid duplication of computation, if required)
3. Monomorphization by specializing functions with respect to type arguments
4. Insert "let" by zeta-expantion to make a term A-normal form
5. Construct AST

Steps 1 to 4 are convertible

We give up if the term cannot correspond to GA

We implement this translation in CODEGEN (future work)

Function of Recdef

- Function generates a term which uses sig type as a return type
- It conflicts with syntactic proof elimination