Intrinsically Typed Reflection of a Gallina Subset Supporting Dependent Types for Non-structural Recursion of Coq

Akira Tanaka

National Institute of Advanced Industrial Science and Technology (AIST)
2018-11-21

The 14th Theorem Proving and Provers meeting (TPP 2018)

Purpose

We want to generate practical C programs from Coq

- Write a program in Gallina
- Verify the program using Coq
- Translate the program to C
- Execute the program efficiently

We are developing codegen plugin for Coq

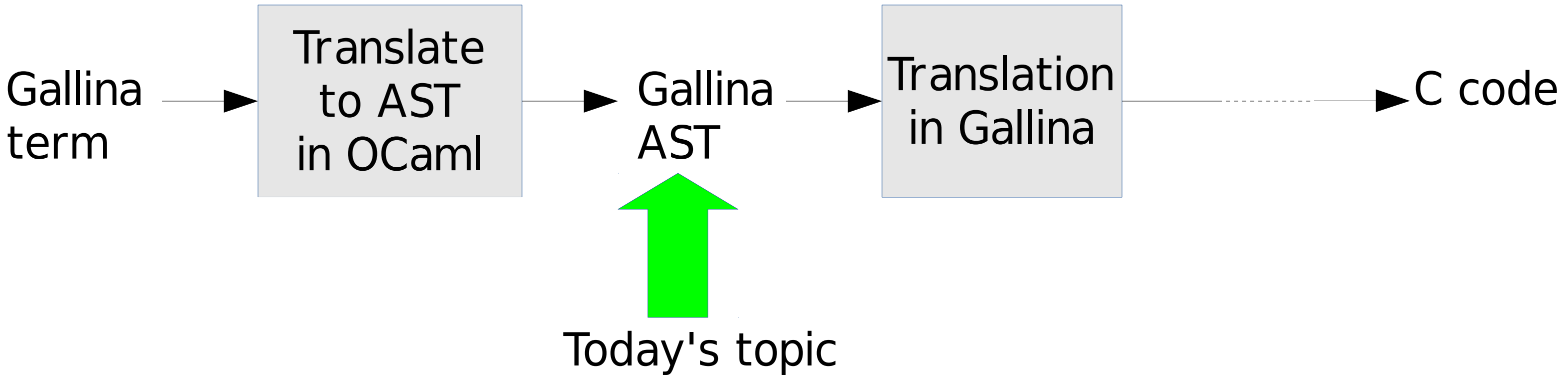
- Redesign codegen to reduce trusted computing base

Short Story

- We want to generate practical C program including increasing loop, BFS (breadth first search), etc.
  - Non-structural recursion (dependent type) is required to represent them in Gallina
  - Proof elimination is required
- We want to verify code generation itself
  - Currently code generator is implemented in OCaml entirely
  - Rewriting a part of code generator in Gallina makes verification easier
  - We need to represet Gallina term as AST for code generator
  - We designed AST which can represent non-structural recursion

Translating Gallina to C

- Accessing Gallina term needs OCaml
- AST translation would be possible in Gallina
  It makes verifying translation easier (than OCaml)
- (Writing a C file needs OCaml)



Gallina term

Translate to AST in OCaml

Gallina AST

Translation in Gallina

C code

Today's topic

Expected Translation

-
  ```
  Definition f a b c :=
    a + g b + c
  ```

-
  ```
  nat f(nat a,
        nat b, nat c) {
    nat t1 = g(b);
    nat t2 = addn(a,t1);
    nat t3 = addn(t2,c);
    return t3;
  }
  ```

Efficient C program generation

- Inductive types is fully customizable in C
  - nat type in C is implemented by hand

    ```
    typedef uint64_t nat;
    #definie addn(a,b) ((a)+(b))
    ```
- Polymorphic function is monomorphized
- Tail-recursion is translated to goto

Efficient (Normal) C Program

- Strict evaluation (Not lazy evaluation)
- Primitives types
  Ex. uint64_t (No tag bit)
- Primitive operators
  Ex. +, __builtin_popcount
  (No function call overhead for them)
- Normal calling convention supported by CPU and standard ABI
  (No trick for general tail call without stack consumption)
- Loop if possible (Not recursion by default)
- Avoid heap allocation if possible (Not heap by default)

Our Main Idea

- There is a Gallina subset close to a C subset powerful enough
  → Direct mapping from Gallina to C without overhead is possible
- Several translation phases are convertible (A-normal form and monomorphization)
  → Convertible phases can be verified easily (single reflexivity tactic proof)

Expected Translation

- Definition f a b c :=
  a + g b + c

A-normal form

- Definition _f a b c :=
  let t1 := g b in
  let t2 := addn a t1 in
  let t3 := addn t2 c in
  t3

f and _f is convertible

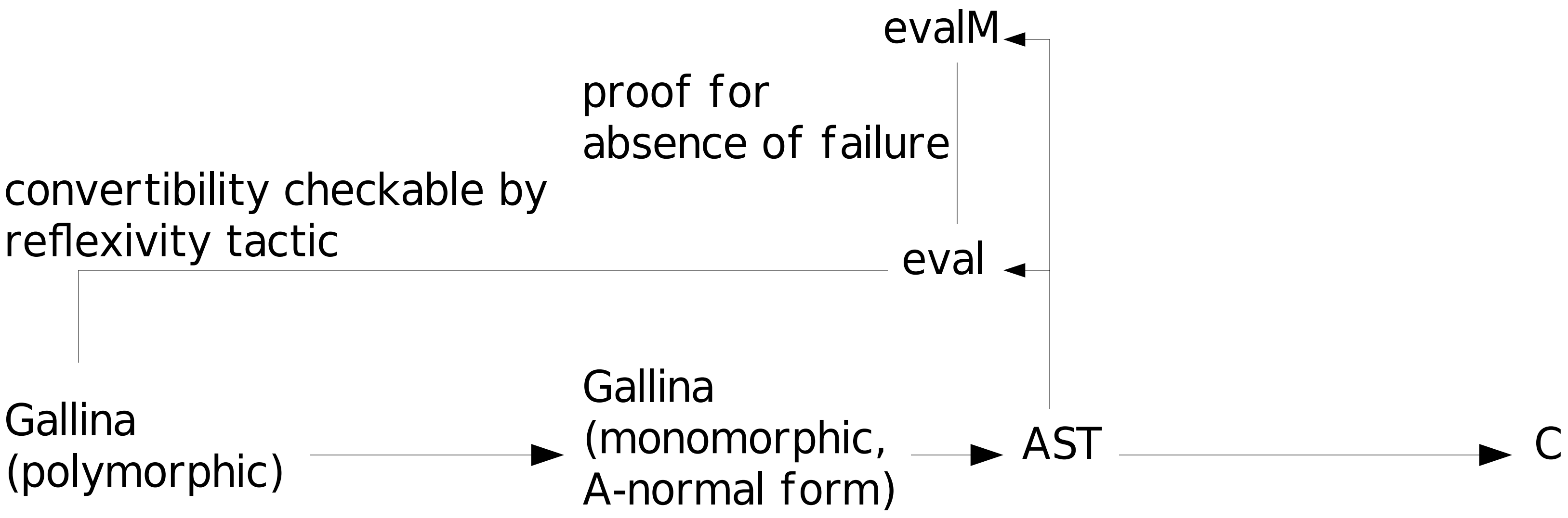
C code generation

- nat f(nat a,
        nat b, nat c) {
    nat t1 = g(b);
    nat t2 = addn(a,t1);
    nat t3 = addn(t2,c);
    return t3;
  }

Requirement for Gallina subset AST

- AST can represent a Gallina subset correspond to practical C program including increasing loop, BFS, etc
- Evaluation returns original term
  eval term_AST = term
- Monadic evaluation function implementable
  This is required for verification of uint64_t implementation of nat
  evalM term_AST = Some term
- Proof elimination implementable
- C code generation implementable

Translation Structure



evalM

proof for
absence of failure

convertibility checkable by
reflexivity tactic

eval

Gallina
(polymorphic)

Gallina
(monomorphic,
A-normal form)

AST

C

Convertible
Translation
- monomorphization
- A-normal form
- AST generation

Non-convertible
Translation
- Proof elimination
- C code generation

Structural Recursion is not Enough

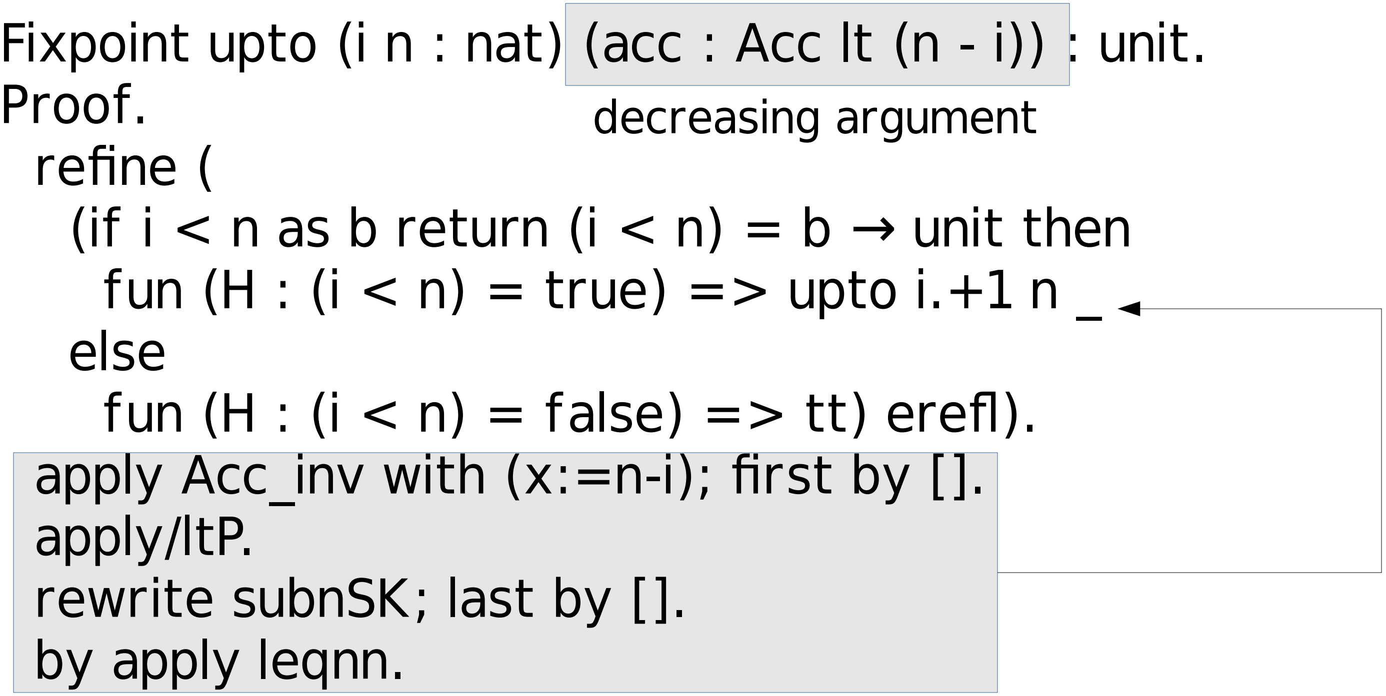
- Decreasing loop can be implemented in Coq
  ```
  for (i = n; 0 < i; i--) {}
  Fixpoint f n := match n with 0 => ... | n'.+1 => f n' end
  ```
- Increasing loop is not possible
  ```
  for (i = 0; i < n; i++) {}
  Fixpoint f i n := if i < n then f i.+1 n else ...
  (* Error: Cannot guess decreasing argument of fix. *)
  ```
- There are practical C code which doesn't correspond to structurally recursive Coq function:
  - increasing loop
  - queue of breadth first search (BFS) can shrink and grow
- Non-structural recursion is required

Non-structural Recursion in Coq

- Recursive function in Coq must be structural recursion
- Non-structural recursion is emulated by additional argument which is structurally decreasing and it's sort is Prop
- Typically, Acc type is used for the argument
- The additional argument is removed at extraction because it is Prop
- The additional argument depends on prior arguments i.e. **dependent type** is required
- We need to represent dependent type in AST

Example of Non-structural Recursion: Increasing Loop

- Argument i is increasing to n.
- Fixpoint upto (i n : nat) (acc : Acc lt (n - i)) : unit.



```
Fixpoint upto (i n : nat) (acc : Acc lt (n - i)) : unit.
Proof.
  refine (
    (if i < n as b return (i < n) = b → unit then
      fun (H : (i < n) = true) => upto i.+1 n _
    else
      fun (H : (i < n) = false) => tt) erefl).
  apply Acc_inv with (x:=n-i); first by [].
  apply/ltP.
  rewrite subnSK; last by [].
  by apply leqnn.
Defined.
```

decreasing argument

AST Supporting Non-structural Recursion

Terminals of AST Syntax

f : global function name

r : recursive function name

B : recursive function body name

h : proof function name

C : constructor

v : normal (non-dependent) variable

p : proof variable (can depend on normal variables)

D : decreasing proof argument initializer
(typically "lt_wf v" for "Acc lt v")

AST of Expression

```
exp = v
      | app
      | letapp v p := app in exp
      | rapp
      | letrapp v := rapp in exp
      | letproof p := proof in exp
      | nmatch v with | C v* => exp | … end
      | dmatch v with | C v* p => exp | … end
      | letnmatch v := v with | C v* => exp | … end
      | letdmatch v := v with | C v* p => exp | … end
```

variable

application
with/without let

match
with/without let

```
app = f v*             (* global function application *)
rapp = r v* [p]        (* recursive function application. *)
proof = h v* p*        (* lemma application *)
```

AST of Program (not implemented yet)

```
program = def*


def= f v* := exp                      <-- non-recursive function
     | f v* := fix r (r := B)+ [D]    <-- recursive function
     | B r+ v* := exp                  }
     | B r+ v* p := exp                } recursive function body
```

Semantics of Expression AST

Semantics of AST is defined as Gallina expression

- E[v] = v
- E[f v1...] = f v1...
- E[letapp v0 p = f v1 ... in e] =
  let v0 := f v1... in let p : (v0 = f v1...) := erefl v0 in E[e]
- E[rapp v1... [p]] = r v1... [p]
- E[letrapp v := r v1... [p] in e] = let v := r v1... [p] in E[e]
- E[letproof p := h v1... p1... in e] = let p := h v1... p1... in E[e]
- E[nmatch v with | C1 v11... => e | ... end] =
  match v with | C1 v11... => E[e] | ... end
- E[dmatch v with | C1 v11... p1 => e1 | ... end] =
  match v as v' return v = v' → T with
  | C1 v11... => fun (p : (v = C1 v11..)) => E[e1] | ...
  end (erefl v)
- E[letnmatch v1 := v2 with ... end in e] =
  let v1 := E[nmatch v2 with ... end] in E[e]
- E[letdmatch v1 := v2 with ... end in e] =
  let v1 := E[dmatch v2 with ... end] in E[e]

Trivial Except letapp and dmatch

Semantics of AST is trivial except letapp and dmatch:

- E[v] = v
- E[f v1...] = f v1...
- E[letapp v0 p = f v1 ... in e] =
  let v0 := f v1... in E[e]
- E[rapp v1... [p]] = r v1... [p]
- E[letrapp v := r v1... [p] in e] = let v := r v1... [p] in E[e]
- E[letproof p := h v1... p1... in e] = let p := h v1... p1... in E[e]
- E[nmatch v with | C1 v11... => e1 | ... end] =
  match v with | C1 v11... => E[e1] | ... end
- E[dmatch v with | C1 v11... p1 => e1 | ... end] =
  match v with
  | C1 v11... => E[e1] | ...
  end
- E[letnmatch v1 := v2 with ... end in e] =
  let v1 := E[nmatch v2 with ... end] in E[e]
- E[letdmatch v1 := v2 with ... end in e] =
  let v1 := E[dmatch v2 with ... end] in E[e]

Technical Challenges for AST

- The index of GADT-style AST
  → Five environments
- No type-generic match expression in Gallina
  → matcher function
- Proof needs zeta-reduction
  $E[\Gamma] \vdash \text{let } x := u \text{ in } t \triangleright t\{x/u\}$
  → letapp binds equality proof
  proof can use the equality instead of zeta-reduction

GADT-style AST

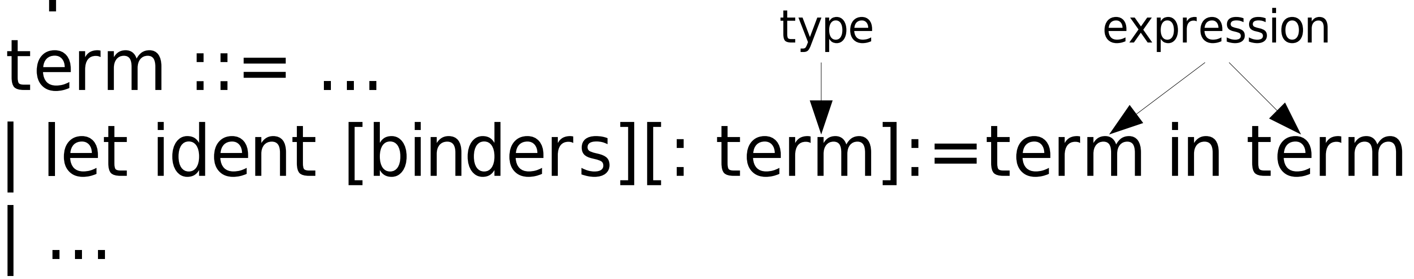
- Usual GADT interpreter (typically explained in Haskell) uses expression type indexed by return type
  I.e. Inductive exp : Set → Type
- Variables needs another index for variable types (cf. CPDT)
  I.e. Inductive exp : seq Set → Set → Type
- Our AST uses five indexes for variables
  - global environment
  - lemma environment
  - recursive function environment
  - normal (non-dependent) environment
  - proof environment

AST Indexed by Actual Dependent Type doesn't Work

- Naive GADT-style AST would be:
  Inductive exp := **Type** → Type := ...
  index is actual expression type
- Dependent type makes AST traversal impossible
- Consider a function with dependent type
  fun (x y : nat) (H : x < y) => H
- The actual type of H is available for actual value of x and y which are not known until the function is called
- H_AST : exp (x < y)
- We need actual value of x and y before obtaining H_AST
- It is not possible for proof elimination

AST Indexed by Type AST doesn't Work

- Gallina syntax doesn't distinguish type and expression:



type

expression

term ::= ...
| let ident [binders][: term]:=term in term
| ...

- GADT-style AST would be:
  Inductive term := **term** → Type := ...
- Of course, "term" is not usable as index because it is not defined yet

Split Non-dependent Variables and Dependent Variables

- Inductive exp (nT : nenvtype)
  (pT : penvtype nT) : Set → Type
- nT is index for normal variables (non-dependent type)
- pT is index for proof variables (dependent type)
- proof type is a function from normal environment to Prop
- Actually, we need more indexes for variables

No Type-generic match in Gallina

- Number of constructors is fixed in match expression:
  match v with
  | **C1** v11... => e1 | ... | **Cn** vn1... => en
  end
- Evaluation needs type-generic match
  E[nmatch v with | C1 v11... => e1 | ... end] =
  match v with | C1 v11... => E[e1] | ... end
- How we implement eval for nmatch?

(Normal) Matcher Function

- We use matcher functions
  - Matcher function is a function similar to recursor (such as nat_rect) but only dispatch, doesn't recurse
  - Embed matcher function in AST
  - Evaluation function invokes matcher function
- matcher function is defined for each inductive type:

```
Definition nat_nmatcher (Tr : Set) (v : nat)
  (branch_O : Tr) (branch_S : nat -> Tr) : Tr :=
match v with
| O => branch_O
| S n => branch_S n
end
```

Dependent Matcher Function

- Proof needs information about selected branch at match expression
- dmatch provides this information

-
  ```
  Definition nat_dmatcher (Tr : Set) (v : nat)
    (branch_O : v = O → Tr)
    (branch_S : forall n, v = S n → Tr) : Tr :=
  match v as v' return v = v' → Tr with
  | O => branch_O
  | S n => branch_S n
  end (erefl v).
  ```

Provide Equality Proof to zeta-reduction

- In AST, proof is build by lemma invocation
- exp = letproof p := proof in exp | …
  proof = h v* p*   (* lemma application *)
- zeta-reduction of argument is not possible in a lemma

zeta-reduction Usable in Proof

- Argument i is increasing to n.
- Fixpoint upto (i n : nat) (acc : Acc lt (n - i)) : unit.

```
Proof.
  refine (
    (if i < n as b return (i < n) = b → unit then
      fun (H : (i < n) = true) => let j := i.+1 in upto j n _
    else
      fun (H : (i < n) = false) => tt) erefl).
  apply Acc_inv with (x:=n-i); first by [].
  apply/ltP.
  rewrite subnSK; last by [].
  by apply leqnn.
Defined.
```

Bind j for A-normal form

j and i.+1 is convertible here

zeta-reduction Unusable in Lemma

- Lemma upto_lemma (i n : nat) (b : bool)
  (j : nat) (acc : Acc lt (n – i))
  (Hb : b = (i < n)) (Hm : b = true)
  (Hj : j = i.+1) : Acc lt (n – j).
  Proof.
  Defined.

j and i.+1 is not convertible
We need Hj instead

- letapp provides equality proof
  E[letapp v0 p = f v1 … in e] =
  let v0 := f v1… in
  let p : (v0 = f v1…) := erefl v0 in E[e]

Proof Usage in AST

- Proof bindings
  - recursive function definition binds an (optional) proof argument
  - letapp binds p : v0 = f v1…
  - dmatch binds p : v = Ci vi1…
  - letproof binds p := h v1… p1…
- Proof occurrences
  - letproof uses p1…
  - rapp and letrapp uses an (optional) proof argument for recursive function application
- The decreasing proof argument of recursive application is built by lemma invocation (letproof) with given decreasing argument and proofs for variable definitions by letapp and dmatch

Example

Example of Non-structural Recursion: Increasing Loop

- Argument i is increasing to n.
- Fixpoint upto (i n : nat) (acc : Acc lt (n - i)) : unit.

  ```
  Proof.
    refine (
      (if i < n as b return (i < n) = b → unit then
        fun H => upto i.+1 n _
      else
        fun H => tt) erefl).
    apply Acc_inv with (x:=n-i); first by [].
    apply/ltP.
    rewrite subnSK; last by [].
    by apply leqnn.
  Defined.
  ```

Split Definition

-
  ```
  Lemma upto_lemma (i n : nat) (b : bool) (j : nat) (acc : Acc lt (n – i))
    (Hb : b = (i < n)) (Hm : b = true) (Hj : j = i.+1) : Acc lt (n – j).
  Proof. … Defined.
  ```

-
  ```
  Definition upto_body (upto : forall (i n : nat) (acc : Acc lt (n - i)), unit)
      (i n : nat) (acc : Acc lt (n - i)) : unit :=
    let b := i < n in let Hb : b = (i < n) := erefl in
    (if b as b' return b = b' -> unit then
      fun Hm =>
        let j := i.+1 in let Hj : j = i.+1 := erefl in
        let acc' := upto_lemma i n b j acc Hb Hm Hj in
        upto j n acc'
    else
      fun Hm => tt) erefl.
  ```

-
  ```
  Fixpoint upto (i n : nat) (acc : Acc lt (n - i)) {struct acc} : unit :=
    upto_body upto i n acc.
  ```
- This definition is convertible to previous upto

AST for upto_body

-
  ```
  letapp b Hb := ltn i n in
  dmatch b with
  | true Hm =>
      letapp j Hj := S i in
      letproof acc' :=
        upto_lemma i n b j acc Hb Hm Hj in
      upto j n acc'
  | false Hm =>
      tt
  end
  ```
- Evaluation of this AST is convertible with previous upto_body

Details of AST

Five Environments

- global environment
- lemma environment
- recursive function environment
- normal (non-dependent) environment
- proof environment

Global and local variables

- We have two kind of normal (non-dependent) variables
  - global constant, referenced by name (string)
  - local variable, referenced using de Bruijn Index
- Inductive exp (gT : seq (string * gty))
  (nT : seq Set) : Set → Type
- gty represents a type of global constant
- global constant can be a function
  Definition gty : Type := seq Set * Set.

Dependent-typed AST

- We need dependent type for proof variables
- So, we need another index for proof variables which depends on normal variables
- Inductive exp gT nT (pT : seq (pty gT nT)) : Set → Type
- (pty gT nT) represent a proof variable type
- pty gT nT is a function type which takes global environment and normal environment and returns a Prop type

Proof Type Depends on Global Environment

- E[letapp v0 p = f v1 ... in e] =
  let v0 := f v1... in
  let p : (v0 = f v1...) := erefl
  in E[e]
- "f" refers a global constant
- (v0 = f v1...) actually uses global environment as: v0 = glookup genv "f" v1...
- So, proof type must depends on global environment type

Lemma Environment

- We need another environment for lemmas
- E[letproof p := h v1... p1... in e] =
  let p := h v1... p1... in E[e]
- "h" refers a lemma
- We cannot embed lemma in AST
  - Actual value of global constant is unknown in AST
    For example, "S" can refer any function of nat → nat
  - So, lemma environment must be defined against for a specific global environment

Recursive Function Environment

- Recursive function is described as:
  - def= ...
    | f v* := fix r (r := B)+ [D]
    | B r+ v* := exp
    | B r+ v* p := exp
  - exp = ... | rapp | letrapp v := rapp in exp
  - rapp = r v* [p]
- "r" needs yet another environment
  It is similar to global function but it can take a proof argument
- Global function cannot take proof arguments to avoid mutual reference between gT and pT

Limitation of letrapp, letnmatch and letdmatch

- letrapp, letnmatch and letdmatch doesn't bind equality proof as letapp
- Equality at letrapp depends on the recursive function
  pty doesn't take recursive environment to avoid mutual reference between pty and rT
  So, letrapp doesn't bind equality proof
- Equality at letnmatch and letdmatch depends on eval because nmatch and dmatch has subexpression
  But pty doesn't take eval function
  So, letnmatch and letdmatch doesn't bind equality proof
- I think this limitation is not a big problem for non-structural recursion
- Function of FunInd has a similar limitation that function is pure pattern-matching tree
  Pure pattern-matching tree doesn't need letnmatch and letdmatch

Expression Types

- Definition nenvtype : Type := seq Set.
- Definition gty : Type := nenvtype * Set.
- Definition genvtype : Type := seq (string * gty).
- Definition pty (gT : genvtype) (nT : nenvtype) : Type :=
  genviron gT -> nenviron nT -> Prop.
- Definition penvtype (gT : genvtype) (nT : nenvtype) : Type := seq (pty gT nT).
- Definition lty gT : Type := {nT:nenvtype & (penvtype gT nT * pty gT nT)%type}.
- Definition lenvtype (gT : genvtype) : Type := seq (string * lty gT).
- Definition rty gT : Type := {nT:nenvtype & option (pty gT nT)} * Set.
- Definition renvtype (gT : genvtype) : Type := seq (string * rty gT).
- Inductive exp (gT : genvtype)
  (lT : lenvtype gT) (rT : renvtype gT)
  (nT : nenvtype) (pT : penvtype gT nT) :
  Set -> Type := …

Environment is Heterogenious List

-
  ```
  Fixpoint nenviron (nT : nenvtype) : Set :=
    match nT with
    | [::] => unit
    | T :: nT' => prod T (nenviron nT')
    end.
  ```
- nenviron [:: nat; bool] is (nat * (bool * tt))
- genviron, penviron, lenviron, renviron is similar

Verify AST and Import a Function into Global Environment

Note: This process will be automated with codegen plugin

- Definition GT1 := (* global constant types *) **(*Base global env*)**
- Definition GENV1 : genviron GT1 := (* global constant environment *)
- Lemma upto_lemma … **(*Split Definition*)**
- Definition upto_body …
- Fixpoint upto …
- Definition LT2 : lenvtype GT1 := ("upto_lemma", …) :: … **(*Extend lemma env*)**
- Definition LENV2 : lenviron GT1 LT2 GENV1 := … :: …
- Definition upto_body_AST := … **(*Define AST*)**
- Definition upto_body' … := … (eval … upto_body_AST). **(*Verify AST*)**
- Lemma upto_body_ok : upto_body = upto_body'. reflexivity. Qed.
- Definition upto_without_acc (i n : nat) := upto i n (lt_wf (n - i)).
- Definition GT2 := ("upto", …) :: GT1. **(*Extend global env*)**
- Definition GENV2 : genviron GT2 := (upto_without_acc, GENV1).

Recursive Function Definition and Termination Checker

- We cannot define a recursive function using AST due to limitation of Coq termination checker
  - eval is too complex for Coq termination checker

    ```
    Fail Fixpoint upto'' (i n : nat) (acc : Acc lt (n - i)) {struct acc} : unit :=
      upto_body' upto'' i n acc.
    (* Recursive call to upto'' has not enough arguments. *)
    ```
- So, we imported the original function, not AST-based one

  -
    ```
    Definition upto_without_acc (i n : nat) := upto i n (lt_wf (n – i)).
    Definition GT2 := ("upto", ...) :: GT1.
    Definition GENV2 : genviron GT2 := (upto_without_acc, GENV1).
    ```
- It is possible to import AST-based one if we reduce function body

  -
    ```
    Definition upto_body'' (upto : forall (i n : nat), Acc lt (n - i) -> unit)
        (i n : nat) (acc : Acc lt (n - i)) : unit := Eval cbv in upto_body' upto i n acc.
    Fixpoint upto'' (i n : nat) (acc : Acc lt (n - i)) {struct acc} : unit :=
      upto_body'' upto'' i n acc.
    Goal upto = upto''. reflexivity. Qed.
    ```
- But it seems not so worth to do it

Future Work

- Automatic AST generation
- Implement monadic eval and try to verify failable primitives
- Implement and verify proof elimination
  Proof eliminated AST should return same value
  Needs a relation between nmatcher and dmatcher
- C code generation
  Needs type names. One more environment?
- Support a dmatch variant for v' = v instead of v = v'?
  Coq's Program command (Russel) uses v' = v
- Support higher order function?
  Verification of proof elimination would be difficult
  (It seems relation between before/after proof elimination
  violates positivity condition)

Summary

- Dependent-typed AST is defined
- This AST represent a Gallina subset which can support non-structural recursion
- Evaluation of this AST is convertible with original term
- This AST is designed as an intermediate representation for codegen