Towards less painful verification of the full correctness for C

Keiko Nakata

25 October 2008

The content of the talk

A report on my experience in combining an automatic decision procedure (Ergo) and interactive reasoning (Coq) to prove both functional correctness and memory safety for a subset of C programs, or C without goto, based on a separation logic framework.

What is Coq? (1)
Background

Coq is a proof assistant, where the programmer interactively constructs a proof term which witnesses that the stated proposition is true.

```
Lemma inj_prj: ∀p1 p2:nat * nat,
  fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2.
Proof.
  destruct p1; destruct p2; simpl in |- *; intros Hfst Hsnd.
  rewrite Hfst; rewrite Hsnd; reflexivity.
Qed.
```

```
inj_prj = fun p1 :  nat * nat =>
let (n, n0) as p
  return (∀p2 :  nat * nat, fst p = fst p2 -> snd p = snd p2 -> p = p2) := p1 in
fun p2 :  nat * nat =>
let (n1, n2) as p
  return (fst (n, n0) = fst p -> snd (n, n0) = snd p -> (n, n0) = p) := p2 in
fun (Hfst :  n = n1) (Hsnd :  n0 = n2) =>
eq_ind_r (fun n3 :  nat => (n3, n0) = (n1, n2))
  (eq_ind_r (fun n3 :  nat => (n1, n3) = (n1, n2)) (refl_equal (n1, n2)) Hsnd) Hfst
:  ∀p1 p2 :  nat * nat, fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2
```

What is Coq? (1)
Background

Coq is a proof assistant, where the programmer interactively constructs a proof term which witnesses that the stated proposition is true.

```
Lemma inj_prj: ∀p1 p2:nat * nat,
  fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2.
Proof.
  destruct p1; destruct p2; simpl in |- *; intros Hfst Hsnd.
  rewrite Hfst; rewrite Hsnd; reflexivity.
Qed.
```

```
inj_prj = fun p1 :  nat * nat =>
let (n, n0) as p
  return (∀p2 :  nat * nat, fst p = fst p2 -> snd p = snd p2 -> p = p2) := p1 in
fun p2 :  nat * nat =>
let (n1, n2) as p
  return (fst (n, n0) = fst p -> snd (n, n0) = snd p -> (n, n0) = p) := p2 in
fun (Hfst :  n = n1) (Hsnd :  n0 = n2) =>
eq_ind_r (fun n3 :  nat => (n3, n0) = (n1, n2))
  (eq_ind_r (fun n3 :  nat => (n1, n3) = (n1, n2)) (refl_equal (n1, n2)) Hsnd) Hfst
: ∀p1 p2 :  nat * nat, fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2
```

What is Coq? (1)
Background

Coq is a proof assistant, where the programmer interactively constructs a proof term which witnesses that the stated proposition is true.

```
Lemma inj_prj: ∀p1 p2:nat * nat,
  fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2.
Proof.
  destruct p1; destruct p2; simpl in |- *; intros Hfst Hsnd.
  rewrite Hfst; rewrite Hsnd; reflexivity.
Qed.
```

```
inj_prj = fun p1 :  nat * nat =>
let (n, n0) as p
  return (∀p2 :  nat * nat, fst p = fst p2 -> snd p = snd p2 -> p = p2) := p1 in
fun p2 :  nat * nat =>
let (n1, n2) as p
  return (fst (n, n0) = fst p -> snd (n, n0) = snd p -> (n, n0) = p) := p2 in
fun (Hfst :  n = n1) (Hsnd :  n0 = n2) =>
eq_ind_r (fun n3 :  nat => (n3, n0) = (n1, n2))
  (eq_ind_r (fun n3 :  nat => (n1, n3) = (n1, n2)) (refl_equal (n1, n2)) Hsnd) Hfst
:  ∀p1 p2 :  nat * nat, fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2
```

What is Coq? (2)
Background

Why Coq is useful?

Coq type checks that the constructed proof term inhabits the stated proposition seen as a type.
Thus the correctness of the proof is machine-verified.

What is Coq? (2)
Background

Why Coq is useful?

Coq type checks that the constructed proof term inhabits the stated proposition seen as a type.
Thus the correctness of the proof is machine-verified.

What is Coq? (2)

Background

Why Coq is useful?

Coq type checks that the constructed proof term inhabits the stated proposition seen as a type.
Thus the correctness of the proof is machine-verified.

What is Coq? (3)
Background

There are some practicality issues.

- The programmer has to construct a complete proof term.

No “obvious”, “similar to above cases”, as you might write in paper proof. (Some tactics are provided such as `omega` for automating arithmetic, and `auto` for a Prolog-like resolution procedure, etc)

- The type checker must be sound and is supposed to be terminating for any input.

There are some limitation on how to construct a proof term.

Both engineering efforts and theoretical study are ongoing to address those issues.

What is Coq? (3)
Background

There are some practicality issues.

- The programmer has to construct a complete proof term.

No “obvious”, “similar to above cases”, as you might write in paper proof. (Some tactics are provided such as `omega` for automating arithmetic, and `auto` for a Prolog-like resolution procedure, etc)

- The type checker must be sound and is supposed to be terminating for any input.

There are some limitation on how to construct a proof term.

Both engineering efforts and theoretical study are ongoing to address those issues.

What is Coq? (3)

Background

There are some practicality issues.

- The programmer has to construct a complete proof term.

No “obvious”, “similar to above cases”, as you might write in paper proof. (Some tactics are provided such as `omega` for automating arithmetic, and `auto` for a Prolog-like resolution procedure, etc)

- The type checker must be sound and is supposed to be terminating for any input.

There are some limitation on how to construct a proof term.

Both engineering efforts and theoretical study are ongoing to address those issues.

What is Coq? (3)

Background

There are some practicality issues.

- The programmer has to construct a complete proof term.

No “obvious”, “similar to above cases”, as you might write in paper proof. (Some tactics are provided such as `omega` for automating arithmetic, and `auto` for a Prolog-like resolution procedure, etc)

- The type checker must be sound and is supposed to be terminating for any input.

There are some limitation on how to construct a proof term.

Both engineering efforts and theoretical study are ongoing to address those issues.

What is Separation logic? (1)

Background

Separation logic is a variant of Hoare logic, which facilitates reasoning about imperative programs that explicitly operates on memory.

```
Variable mem :Set
Definition assert := mem → Prop.
Variables p, q : assert.
```

{p}s{q} means “in a memory state where p holds, the program s can be executed without unsafe memory access (safety), and if the execution terminates then q holds at the final memory state (correctness).”

What is Separation logic? (2)

Background

The infix operator ** expresses disjoint union.

```
Variable mem :Set
Definition assert := mem → Prop.
Variables p, q : assert.

Definition p ** q := fun m → ∃ m1, ∃ m2,
 disjunion m m1 m2 /\ p1 m1 /\p2 m2
```

Frame rule

$$\frac{\{p1\}s\{p2\}}{\{p1 * q\}s\{p2 * q\}}$$

What is Separation logic? (2)

Background

The infix operator ** expresses disjoint union.

```
Variable mem :Set
Definition assert := mem → Prop.
Variables p, q : assert.

Definition p ** q := fun m → ∃ m1, ∃ m2,
 disjunion m m1 m2 /\ p1 m1 /\p2 m2
```

Frame rule

$$\frac{\{p1\}s\{p2\}}{\{p1 * * q\}s\{p2 * * q\}}$$

What is Separation logic? (3)

Background

assignment

$$\{\texttt{ex i}, \texttt{x} \mapsto \texttt{i}\}\ \texttt{x := j}\ \{\texttt{x} \mapsto \texttt{j}\}$$

sequence

$$\frac{\{\texttt{p1}\}\texttt{s1}\{\texttt{p3}\} \quad \{\texttt{p3}\}\texttt{s2}\{\texttt{p2}\}}{\{\texttt{p1}\}\texttt{s1;s2}\{\texttt{p2}\}}$$

$\{\texttt{x} \mapsto 3 \ast\ast \texttt{y} \mapsto 2\}\ \texttt{x := 5; y := 7}\ \{\texttt{x} \mapsto 5 \ast\ast \texttt{y} \mapsto 7\}$
is derived from:

$$\frac{\{\texttt{x} \mapsto 3\}\texttt{x} := 5\{\texttt{x} \mapsto 5\}}{\{\texttt{x} \mapsto 3 \ast\ast \texttt{y} \mapsto 2\}\texttt{x} := 5\{\texttt{x} \mapsto 5 \ast\ast \texttt{y} \mapsto 2\}}$$

$$\frac{\{\texttt{y} \mapsto 2\}\texttt{y} := 7\{\texttt{y} \mapsto 7\}}{\{\texttt{y} \mapsto 2 \ast\ast \texttt{x} \mapsto 5\}\texttt{y} := 7\{\texttt{y} \mapsto 7 \ast\ast \texttt{x} \mapsto 5\}}$$

What is Separation logic? (3)

Background

assignment

$$\{\texttt{ex i}, \texttt{x} \mapsto \texttt{i}\}\ \texttt{x := j}\ \{\texttt{x} \mapsto \texttt{j}\}$$

sequence

$$\frac{\{\texttt{p1}\}\texttt{s1}\{\texttt{p3}\} \quad \{\texttt{p3}\}\texttt{s2}\{\texttt{p2}\}}{\{\texttt{p1}\}\texttt{s1;s2}\{\texttt{p2}\}}$$

$$\{\texttt{x} \mapsto 3 \ \texttt{**}\ \texttt{y} \mapsto 2\}\ \texttt{x := 5; y := 7}\ \{\texttt{x} \mapsto 5 \ \texttt{**}\ \texttt{y} \mapsto 7\}$$

is derived from:

$$\frac{\{\texttt{x} \mapsto 3\}\texttt{x} := 5\{\texttt{x} \mapsto 5\}}{\{\texttt{x} \mapsto 3 * \texttt{y} \mapsto 2\}\texttt{x} := 5\{\texttt{x} \mapsto 5 * \texttt{y} \mapsto 2\}}$$

$$\frac{\{\texttt{y} \mapsto 2\}\texttt{y} := 7\{\texttt{y} \mapsto 7\}}{\{\texttt{y} \mapsto 2 * \texttt{x} \mapsto 5\}\texttt{y} := 7\{\texttt{y} \mapsto 7 * \texttt{x} \mapsto 5\}}$$

What is Separation logic? (3)

Background

assignment

$$\{\texttt{ex i}, \texttt{x} \mapsto \texttt{i}\}\ \texttt{x := j}\ \{\texttt{x} \mapsto \texttt{j}\}$$

sequence

$$\frac{\{\texttt{p1}\}\texttt{s1}\{\texttt{p3}\} \quad \{\texttt{p3}\}\texttt{s2}\{\texttt{p2}\}}{\{\texttt{p1}\}\texttt{s1;s2}\{\texttt{p2}\}}$$

$$\{\texttt{x} \mapsto 3 \ \texttt{**}\ \texttt{y} \mapsto 2\}\ \texttt{x := 5; y := 7}\ \{\texttt{x} \mapsto 5 \ \texttt{**}\ \texttt{y} \mapsto 7\}$$

is derived from:

$$\frac{\{\texttt{x} \mapsto 3\}\texttt{x} := 5\{\texttt{x} \mapsto 5\}}{\{\texttt{x} \mapsto 3 * \texttt{y} \mapsto 2\}\texttt{x} := 5\{\texttt{x} \mapsto 5 * \texttt{y} \mapsto 2\}}$$

$$\frac{\{\texttt{y} \mapsto 2\}\texttt{y} := 7\{\texttt{y} \mapsto 7\}}{\{\texttt{y} \mapsto 2 * \texttt{x} \mapsto 5\}\texttt{y} := 7\{\texttt{y} \mapsto 7 * \texttt{x} \mapsto 5\}}$$

What is Separation logic? (3)

Background

assignment

$$\{\texttt{ex i}, \texttt{x} \mapsto \texttt{i}\}\ \texttt{x := j}\ \{\texttt{x} \mapsto \texttt{j}\}$$

sequence

$$\frac{\{\texttt{p1}\}\texttt{s1}\{\texttt{p3}\} \quad \{\texttt{p3}\}\texttt{s2}\{\texttt{p2}\}}{\{\texttt{p1}\}\texttt{s1;s2}\{\texttt{p2}\}}$$

$$\{\texttt{x} \mapsto 3 \ \texttt{**}\ \texttt{y} \mapsto 2\}\ \texttt{x := 5; y := 7}\ \{\texttt{x} \mapsto 5 \ \texttt{**}\ \texttt{y} \mapsto 7\}$$

is derived from:

$$\frac{\{\texttt{x} \mapsto 3\}\texttt{x} := 5\{\texttt{x} \mapsto 5\}}{\{\texttt{x} \mapsto 3 * * \texttt{y} \mapsto 2\}\texttt{x} := 5\{\texttt{x} \mapsto 5 * * \texttt{y} \mapsto 2\}}$$

$$\frac{\{\texttt{y} \mapsto 2\}\texttt{y} := 7\{\texttt{y} \mapsto 7\}}{\{\texttt{y} \mapsto 2 * * \texttt{x} \mapsto 5\}\texttt{y} := 7\{\texttt{y} \mapsto 7 * * \texttt{x} \mapsto 5\}}$$

Separation logic for Clight

Background

Our target language Clight, is C without goto and is the front-end langauge of the Compcert certified compiler.

We have formalized in Coq a separation logic for Clight, which is proved sound w.r.t. the operational semantics.

If the programmer proves $\{p\}s\{q\}$ is derivable within the logic, then the executable compiled by the Compcert certified compiler is safe and correct.

But is the logic usable?
Is the proof for $\{p\}s\{q\}$ is doable without undue verification overhead?

Separation logic for Clight

Background

Our target language Clight, is C without goto and is the front-end langauge of the Compcert certified compiler.

We have formalized in Coq a separation logic for Clight, which is proved sound w.r.t. the operational semantics.

If the programmer proves $\{p\}s\{q\}$ is derivable within the logic, then the executable compiled by the Compcert certified compiler is safe and correct.

But is the logic usable?
Is the proof for $\{p\}s\{q\}$ is doable without undue verification overhead?

Separation logic for Clight

Background

Our target language Clight, is C without goto and is the front-end langauge of the Compcert certified compiler.

We have formalized in Coq a separation logic for Clight, which is proved sound w.r.t. the operational semantics.

If the programmer proves $\{p\}s\{q\}$ is derivable within the logic, then the executable compiled by the Compcert certified compiler is safe and correct.

But is the logic usable?
Is the proof for $\{p\}s\{q\}$ is doable without undue verification overhead?

Separation logic for Clight

Background

Our target language Clight, is C without goto and is the front-end langauge of the Compcert certified compiler.

We have formalized in Coq a separation logic for Clight, which is proved sound w.r.t. the operational semantics.

If the programmer proves $\{p\}s\{q\}$ is derivable within the logic, then the executable compiled by the Compcert certified compiler is safe and correct.

But is the logic usable?
Is the proof for $\{p\}s\{q\}$ is doable without undue verification overhead?

Separation logic for Clight

Background

Our target language Clight, is C without goto and is the front-end langauge of the Compcert certified compiler.

We have formalized in Coq a separation logic for Clight, which is proved sound w.r.t. the operational semantics.

If the programmer proves $\{\mathtt{p}\}\mathtt{s}\{\mathtt{q}\}$ is derivable within the logic, then the executable compiled by the Compcert certified compiler is safe and correct.

But is the logic usable?
Is the proof for $\{\mathtt{p}\}\mathtt{s}\{\mathtt{q}\}$ is doable without undue verification overhead?

Machine-validated program verification

Two opposite approaches to machine-validated program verification have been developed.

Towards full automation
A machine-validated VCG generates proof obligations from annotated programs. The obligations are (supposed to be) discharged by decision procedures.

Interactive reasoning (we are on this side)
The programmer manually proves the safety and correctness, using machine-validate deductive systems.

Full automation approach

Machine-validated program verification

Pos. Extremely easy to use when proof obligations are automatically discharged.

Cons. When decision procedures fail to prove the obligations, their manual proof can be highly painful. Notably,

- full functional correctness
- modulo arithmetic
- pointer cast

are difficult or impossible to be dealt with automatically.

Interactive reasoning approach

Machine-validated program verification

Pos. The programmer can reason about any properties, as long as the properties can be expressed in the logic of the proof assistant.

Cons. Manual proof can require undue verification overhead.

What is a happy medium of the two approaches?

Programmer-navigated semi-automation

We are experimenting the combination of

- Programmer's interaction
  - for navigating the proof search
  - and for performing non-trivial reasoning
- External decision procedure (Ergo & Dp)
  - for easing first-order reasoning
- Home made tactic library
  - for easing separation logic related reasoning

Programmer-navigated semi-automation

We are experimenting the combination of

- Programmer's interaction
  for navigating the proof search
  and for performing non-trivial reasoning
- External decision procedure (Ergo & Dp)
  for easing first-order reasoning
- Home made tactic library
  for easing separation logic related reasoning

Programmer-navigated semi-automation

We are experimenting the combination of

- Programmer's interaction
  for navigating the proof search
  and for performing non-trivial reasoning
- External decision procedure (Ergo & Dp)
  for easing first-order reasoning
- Home made tactic library
  for easing separation logic related reasoning

Programmer-navigated semi-automation

We are experimenting the combination of

- Programmer's interaction
  for navigating the proof search
  and for performing non-trivial reasoning
- External decision procedure (Ergo & Dp)
  for easing first-order reasoning
- Home made tactic library
  for easing separation logic related reasoning

Programmer-navigated semi-automation

We are experimenting the combination of

- Programmer's interaction
  for navigating the proof search
  and for performing non-trivial reasoning
- External decision procedure (Ergo & Dp)
  for easing first-order reasoning
- Home made tactic library
  for easing separation logic related reasoning

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

What we do not attempt to automate

We want to automate “trivial” steps.

In particular, the programmer is responsible to reasoning about

- mathematically difficult properties, which are beyond the ability of (existing) decision procedures
- instantiation of existential variables
- unfolding of inductive predicates
- modulo arithmetic, when integer overflow might happen
- cast, when the underlying representation of the casted value might change

Above are not automated, but are supported.

Comparison w.r.t. lexicographic ordering

An example program to be verified

```
extern int malloc$i(void *);

int str_cmp
(int len, unsigned char *str1, unsigned char *str2){
  int i = 0;
  while (i < len) {
    if ((int)*(str1 + i) > (int)*(str2 + i)) { return 1; }
    if ((int)*(str1 + i) < (int)*(str2 + i)) { return -1; }
    else { i = i + 1; }
  }
  return 0;
}
```

The spec says that `str_cmp` compares the given strings pointed to by `str1` and `str2` of length `len` w.r.t. lexicographic ordering and does not get stuck by accessing invalid memory.

Code snippet for verifying `str_cmp` (1)

```
Lemma safe: semax pf post FF FF pre code_str_cmp FF.
Proof.
Step_assign.
Step_while_auto invr.
EEx.Exists 0.
EProp.Split_nth 1. ergo.
EProp.Split_nth 1. ergo.
Permutation.
EEx.Intro i_curr. by Prove_eval_safe.
Permutation.
AEx.Intro i_curr.
AProp.Destruct_nth 1. move => i_curr_inbound.
AProp.Destruct_nth 1. move => cont_eq_sofar.
let p := AMisc.Pre in
match p with | ?p1 et ?p2 => Deduce_pre p2 end.
move => curr_lt_len.
```

Code snippet for verifying `str_cmp` (2)

```
have: 0 <= i_curr <len. ergo. move => O_le_curr_lt_len.
have := cont1_welldef_inbound O_le_curr_lt_len. move => cont1_l8unsigned.
have := cont2_welldef_inbound O_le_curr_lt_len. move => cont2_l8unsigned.
have := array_destruct_item cont1 l_arr1 O_le_curr_lt_len. move => destr_curr1.
have := array_destruct_item cont2 l_arr2 O_le_curr_lt_len. move => destr_curr2.
AEt.Et_weakening_R.
let p := AMisc.Pre in
let next := constr:(p et (TT ** prop ( (cont1 i_curr > cont2 i_curr)))) in
apply (step_sequence next).
apply semax_Sifthenelse.
ESubst.Prepare_for_eval destr_curr1. move => G1. Assoc_H G1.
ESubst.Prepare_for_eval destr_curr2. move => G2. Assoc_H G2.
Prove_eval_safe.
SSubst.Prepare_for_eval destr_curr1. move => G1. Assoc_H G1.
SSubst.Prepare_for_eval destr_curr2. move => G2. Assoc_H G2.
let p := SMisc.Pre in
match p with | ?p1 et ?p2 => Deduce_pre p2 end.
move => cont_gt.
...
```

Homemade tactic library

Preparation

```
Variable mem: Set.
Definition assert := mem → Prop.
Variables p, q : assert.

Definition entail (p q: assert): Prop := ∀ m, p m → q m.
Definition p ** q := fun m → ∃ m1, ∃ m2,
 disjunion m m1 m2 /\ p1 m1 /\p2 m2
```

Homemade tactic library

Our tactic library consists of two main utilities:

- Symbolic evaluation of program expressions.

```
entail
  {x ↦ 3 ** y ↦ 2 ** z ↦ 5}
  (eval_expr (Eq (Add x y) z) 1)
```

Machine arithmetic is discussed later.

- Rearrangement of assertions.

```
entail
  {x ↦ 3 ** y ↦ 2}
  {(ex i:Z, y ↦ i) ** x ↦ 3}
```

is proved in 2 steps with tactics for rearrangement:

```
by Exists 2; Permutation.
```

Homemade tactic library

Our tactic library consists of two main utilities:

- Symbolic evaluation of program expressions.

```
entail
  {x ↦ 3 ** y ↦ 2 ** z ↦ 5}
  (eval_expr (Eq (Add x y) z) 1)
```

Machine arithmetic is discussed later.

- Rearrangement of assertions.

```
entail
  {x ↦ 3 ** y ↦ 2}
  {(ex i:Z, y ↦ i) ** x ↦ 3}
```

is proved in 2 steps with tactics for rearrangement:

```
by Exists 2; Permutation.
```

Homemade tactic library

Our tactic library consists of two main utilities:

- Symbolic evaluation of program expressions.

```
entail
  {x ↦ 3 ** y ↦ 2 ** z ↦ 5}
  (eval_expr (Eq (Add x y) z) 1)
```

Machine arithmetic is discussed later.

- Rearrangement of assertions.

```
entail
  {x ↦ 3 ** y ↦ 2}
  {(ex i:Z, y ↦ i) ** x ↦ 3}
```

is proved in 2 steps with tactics for rearrangement:

```
by Exists 2; Permutation.
```

Outline of the tactics for symbolic evaluation (1)

The proof search of the tactics for symbolic evaluation is based on an axiomatic semantics of Clight. The semantics is not complete but is sound w.r.t the operational semantics.

```
Axiom eval_expr_Ebinop:∀ty, ∀p op d1 ty1 d2 ty2 v1 v2,
entail p (eval_expr (Expr d1 ty1) v1) →
entail p (eval_expr (Expr d2 ty2) v2) →
∀v, (sem_add v1 ty1 v2 ty2 v) →
entail p
  (eval_expr
    (Expr (Eadd (Expr d1 ty1) (Expr d2 ty2)) ty) v).
```

∀-variables are placed to benefit from ssreflect goodies.

Those axioms are the spec of the tactics.

Outline of the tactics for symbolic evaluation (2)

```
Axiom load_TintI8Signed:∀p d l ofs,
entail p (eval_lvalue (Expr d (Tint I8 Signed)) l ofs) →
∀n1, signed ofs = n1 →
∀n3, n1 = n3 →
∀q n, entail p (q ** mapsto l n3 S1 (Vint n)) →
∀n2, cast8signed n = n2 →
entail p (eval_expr (Expr d (Tint I8 Signed)) (Vint n2)).
```

The aximatization uses assertions of the separation logic to specify assumptions about the memory.

The lemmas for the aximatization are shaped to interleave calls to decision procedures.
I.e. to let decision procedures discharge arithmetic.

Outline of the tactics for symbolic evaluation (3)

Tactics immediately fail when they cannot prove an assumption.

This is a strength in that we can identify the assumption the tactics failed to prove via error messages.

Admittedly idtac is not very nice for that purpose.

In case of failure, the programmer can augment the proof context by manually proving the failed assumption.

Then the next run of the tactics may succeed. At least, they advance one step further.

Outline of the tactics for symbolic evaluation (3)

Tactics immediately fail when they cannot prove an assumption.

This is a strength in that we can identify the assumption the tactics failed to prove via error messages.

Admittedly idtac is not very nice for that purpose.

In case of failure, the programmer can augment the proof context by manually proving the failed assumption.

Then the next run of the tactics may succeed. At least, they advance one step further.

Outline of the tactics for symbolic evaluation (3)

Tactics immediately fail when they cannot prove an assumption.

This is a strength in that we can identify the assumption the tactics failed to prove via error messages.

Admittedly idtac is not very nice for that purpose.

In case of failure, the programmer can augment the proof context by manually proving the failed assumption.

Then the next run of the tactics may succeed. At least, they advance one step further.

Outline of the tactics for symbolic evaluation (3)

Tactics immediately fail when they cannot prove an assumption.

This is a strength in that we can identify the assumption the tactics failed to prove via error messages.

Admittedly idtac is not very nice for that purpose.

In case of failure, the programmer can augment the proof context by manually proving the failed assumption.

Then the next run of the tactics may succeed. At least, they advance one step further.

Modular arithmetic (1)

Tactics for symbolic evaluation

Symbolic evaluation intensively involves integer arithmetic.

We want to automate arithmetic, without compelling the programmer to give up machine arithmetic, i.e. 32-bit arithmetic.

Arithmetic is handled by the tactics by internally calling decision procedures, only when that integer overflow does not happen is provable from the proof context.

Modular arithmetic (1)

Tactics for symbolic evaluation

Symbolic evaluation intensively involves integer arithmetic.

We want to automate arithmetic, without compelling the programmer to give up machine arithmetic, i.e. 32-bit arithmetic.

Arithmetic is handled by the tactics by internally calling decision procedures, only when that integer overflow does not happen is provable from the proof context.

Modular arithmetic (1)

Tactics for symbolic evaluation

Symbolic evaluation intensively involves integer arithmetic.

We want to automate arithmetic, without compelling the programmer to give up machine arithmetic, i.e. 32-bit arithmetic.

Arithmetic is handled by the tactics by internally calling decision procedures, only when that integer overflow does not happen is provable from the proof context.

Modular arithmetic (2)

Tactics for symbolic evaluation

We use two functions to come and go between the world of mathematical integers and the world of modular integers:

```
(* Representation of bounded integer *)
Record int: Set :=
  mkint { intval:Z; intrange:0 <= intval < 2^32}.
(* to convert Clight integer to Coq integer *)
Definition Z_of_int (x:int):Z := intval x.
(* to convert Coq integer to Clight integer *)
Definition int_of_Z (x:Z):int :=
  mkint (Zmod x 2^32) (mod_in_range x).

Lemma inbound_id:
  ∀i, 0 <= i < 2^32  → Z_of_int (int_of_Z i) = i
```

Modular arithmetic (2)

Tactics for symbolic evaluation

We use two functions to come and go between the world of mathematical integers and the world of modular integers:

```
(* Representation of bounded integer *)
Record int: Set :=
  mkint { intval:Z; intrange:0 <= intval < 2^32}.
(* to convert Clight integer to Coq integer *)
Definition Z_of_int (x:int):Z := intval x.
(* to convert Coq integer to Clight integer *)
Definition int_of_Z (x:Z):int :=
  mkint (Zmod x 2^32) (mod_in_range x).

Lemma inbound_id:
  ∀i, 0 <= i < 2^32  → Z_of_int (int_of_Z i) = i
```

Modular arithmetic (2)

Tactics for symbolic evaluation

We use two functions to come and go between the world of mathematical integers and the world of modular integers:

```
(* Representation of bounded integer *)
Record int: Set :=
  mkint { intval:Z; intrange:0 <= intval < 2^32}.
(* to convert Clight integer to Coq integer *)
Definition Z_of_int (x:int):Z := intval x.
(* to convert Coq integer to Clight integer *)
Definition int_of_Z (x:Z):int :=
  mkint (Zmod x 2^32) (mod_in_range x).

Lemma inbound_id:
  ∀i, 0 <= i < 2^32  → Z_of_int (int_of_Z i) = i
```

Modular arithmetic (3)

Tactics for symbolic evaluation

Consider a less-than comparison on 32-bit unsigned integers:

```
Definition ltu (n1:int) (n2:int): Prop :=
  Zlt (intval n1) (intval n2)
```

We provide axiomatic views of the comparison.

```
Lemma deduce_from_ltu:∀ i j,
0 <= i < 2^32 →
0 <= j < 2^32 →
ltu (int_of_Z i) (int_of_Z j) →
i < j
```

The idea is inspired by Caduceus.

Modular arithmetic (3)

Tactics for symbolic evaluation

Consider a less-than comparison on 32-bit unsigned integers:

```
Definition ltu (n1:int) (n2:int): Prop :=
  Zlt (intval n1) (intval n2)
```

We provide axiomatic views of the comparison.

```
Lemma deduce_from_ltu:∀ i j,
0 <= i < 2^32 →
0 <= j < 2^32 →
ltu (int_of_Z i) (int_of_Z j) →
i < j
```

The idea is inspired by Caduceus.

Modular arithmetic (4)

Tactics for symbolic evaluation

In the case of comparison on 32-bit signed integer, Clight integer is converted to Coq integer in the signed way:

```
Definition signed (n:int):Z :=
if zlt (intval n) 2^32/2 then (intval n) else (intval n - 2^32).
```

A less-than comparison on 32-bit signed integers:

```
Definition lt (n1 n2: int): Prop := Zlt (signed n1) (signed n2)
```

We provide several axiomatic views:

```
Lemma deduce_from_lt_1:∀ i j,
- 2^32/2 <= i < 2^32/2 → -2^32/2 <= j < 2^32/2 →
lt (int_of_Z i) (int_of_Z j) → i < j
Lemma deduce_from_moins2:∀ i j,
-2^32/2 <= i < 2^32/2 → lt (int_of_Z i) (int_of_Z j) →
i < (signed (int_of_Z j))
And so on.
```

This is not exactly how the tactics are implemented.

Modular arithmetic (4)

Tactics for symbolic evaluation

In the case of comparison on 32-bit signed integer, Clight integer is converted to Coq integer in the signed way:

```
Definition signed (n:int):Z :=
if zlt (intval n) 2^32/2 then (intval n) else (intval n - 2^32).
```

A less-than comparison on 32-bit signed integers:

```
Definition lt (n1 n2: int): Prop := Zlt (signed n1) (signed n2)
```

We provide several axiomatic views:

```
Lemma deduce_from_lt_1:∀ i j,
- 2^32/2 <= i < 2^32/2 → -2^32/2 <= j < 2^32/2 →
lt (int_of_Z i) (int_of_Z j) → i < j
Lemma deduce_from_moins2:∀ i j,
-2^32/2 <= i < 2^32/2 → lt (int_of_Z i) (int_of_Z j) →
i < (signed (int_of_Z j))
And so on.
```

This is not exactly how the tactics are implemented.

Modular arithmetic (4)

Tactics for symbolic evaluation

In the case of comparison on 32-bit signed integer, Clight integer is converted to Coq integer in the signed way:

```
Definition signed (n:int):Z :=
if zlt (intval n) 2^32/2 then (intval n) else (intval n - 2^32).
```

A less-than comparison on 32-bit signed integers:

```
Definition lt (n1 n2: int): Prop := Zlt (signed n1) (signed n2)
```

We provide several axiomatic views:

```
Lemma deduce_from_lt_1:∀ i j,
- 2^32/2 <= i < 2^32/2 → -2^32/2 <= j < 2^32/2 →
lt (int_of_Z i) (int_of_Z j) → i < j
Lemma deduce_from_moins2:∀ i j,
-2^32/2 <= i < 2^32/2 → lt (int_of_Z i) (int_of_Z j) →
i < (signed (int_of_Z j))
And so on.
```

This is not exactly how the tactics are implemented.

Modular arithmetic (5)

Tactics for symbolic evaluation

What can the programmer do when arithmetic is not satisfactory automated?

There is no magic, but the programmer can collaborate with tactics interactively.

```
Variable i:Z.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < (signed (int_of_Z i))

Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < i
```

Modular arithmetic (5)

Tactics for symbolic evaluation

What can the programmer do when arithmetic is not satisfactory automated?
There is no magic, but the programmer can collaborate with tactics interactively.

```
Variable i:Z.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < (signed (int_of_Z i))

Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < i
```

Modular arithmetic (5)

Tactics for symbolic evaluation

What can the programmer do when arithmetic is not satisfactory automated?
There is no magic, but the programmer can collaborate with tactics interactively.

```
Variable i:Z.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < (signed (int_of_Z i))

Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < i
```

Modular arithmetic (5)

Tactics for symbolic evaluation

What can the programmer do when arithmetic is not satisfactory automated?
There is no magic, but the programmer can collaborate with tactics interactively.

```
Variable i:Z.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < (signed (int_of_Z i))

Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < i
```

Modular arithmetic (5)

Tactics for symbolic evaluation

What can the programmer do when arithmetic is not satisfactory automated?
There is no magic, but the programmer can collaborate with tactics interactively.

```
Variable i:Z.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < (signed (int_of_Z i))

Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 50) (int_of_Z i)).
⇒ 50 < i
```

Modular arithmetic (6)

Tactics for symbolic evaluation

Cast can be dealt with interactively.

```
Variable i:Z.
Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 2^32/2) (int_of_Z i)).
⇒ (signed (int_of_Z 2^32/2)) < i.

Variable cast: int_of_Z 2^32/2 = int_of_Z (-2^32/2).
```

Then rewrite,

```
lt (int_of_Z 2^32/2) (int_of_Z i)
```

into

```
lt (int_of_Z (-2^32/2)) (int_of_Z i)
```

Then rerun the tactic.

```
Deduce (lt (int_of_Z (-2^32/2)) (int_of_Z i)).
⇒ -2^32/2 < i
```

Modular arithmetic (6)

Tactics for symbolic evaluation

Cast can be dealt with interactively.

```
Variable i:Z.
Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 2^32/2) (int_of_Z i)).
⇒ (signed (int_of_Z 2^32/2)) < i.

Variable cast: int_of_Z 2^32/2 = int_of_Z (-2^32/2).
```

Then rewrite,

```
lt (int_of_Z 2^32/2) (int_of_Z i)
```

into

```
lt (int_of_Z (-2^32/2)) (int_of_Z i)
```

Then rerun the tactic.

```
Deduce (lt (int_of_Z (-2^32/2)) (int_of_Z i)).
⇒ -2^32/2 < i
```

Modular arithmetic (6)

Tactics for symbolic evaluation

Cast can be dealt with interactively.

```
Variable i:Z.
Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 2^32/2) (int_of_Z i)).
⇒ (signed (int_of_Z 2^32/2)) < i.

Variable cast: int_of_Z 2^32/2 = int_of_Z (-2^32/2).
```

Then rewrite,

```
lt (int_of_Z 2^32/2) (int_of_Z i)
```

into

```
lt (int_of_Z (-2^32/2)) (int_of_Z i)
```

Then rerun the tactic.

```
Deduce (lt (int_of_Z (-2^32/2)) (int_of_Z i)).
⇒ -2^32/2 < i
```

Modular arithmetic (6)

Tactics for symbolic evaluation

Cast can be dealt with interactively.

```
Variable i:Z.
Variable i_inbound: -2^32/2 <= i < 2^32/2.
Deduce (lt (int_of_Z 2^32/2) (int_of_Z i)).
⇒ (signed (int_of_Z 2^32/2)) < i.

Variable cast: int_of_Z 2^32/2 = int_of_Z (-2^32/2).
```

Then rewrite,

```
lt (int_of_Z 2^32/2) (int_of_Z i)
```

into

```
lt (int_of_Z (-2^32/2)) (int_of_Z i)
```

Then rerun the tactic.

```
Deduce (lt (int_of_Z (-2^32/2)) (int_of_Z i)).
⇒ -2^32/2 < i
```

Calling Ergo from Coq (1)

Many of our tactics internally call Ergo, as well as omega, to automate arithmetic during the symbolic evaluation.

Ergo is an automatic theorem prover for the polymorphic first-order logic.

Ergo's ability of instantiating lemmas is interesting to complement Coq's tactic omega, which does not instantiate lemmas.

Calling Ergo from Coq (1)

Many of our tactics internally call Ergo, as well as omega, to automate arithmetic during the symbolic evaluation.

Ergo is an automatic theorem prover for the polymorphic first-order logic.

Ergo's ability of instantiating lemmas is interesting to complement Coq's tactic omega, which does not instantiate lemmas.

Calling Ergo from Coq (1)

Many of our tactics internally call Ergo, as well as omega, to automate arithmetic during the symbolic evaluation.

Ergo is an automatic theorem prover for the polymorphic first-order logic.

Ergo's ability of instantiating lemmas is interesting to complement Coq's tactic omega, which does not instantiate lemmas.

Calling Ergo from Coq (2)

Below the proof context ensures all elements of the arrays are within the bounds of signed 32-bit integers.

```
Variables arr1, arr2:Z → Z.
Variables len: Z.
Variable arr1_elm_inbound:∀i,
0 <= i < len → -2^32/2 <= arr1 i < 2^32/2.
Variable arr2_elm_inbound:∀i,
0 <= i < len → -2^32/2 <= arr2 i < 2^32/2.
Variable index: Z.
Variable index_inrange: 0 <= index < len

Deduce (lt (int_of_Z (arr1 index)) (int_of_Z (arr2 index)))
⇒ arr1 index < arr2 index
```

Above the success of the tactic owes Ergo.

Calling Ergo from Coq (3)

We rely on Dp to bridge the gap between the logic of Coq (CIC) and the logic of Ergo (PFOL).

Dp selectively and soundly translates terms of Coq to terms of Ergo. E.g. higher-order terms are ignored.

In this way, the programmer can call Ergo interactively within Coq and we can call Ergo from our tactics.

- Caveat: the combination of our tactics and Ergo will be available from the next official release of Coq.

The translation by Dp can be interleaved with manual translation. This is indispensable to smoothly combine Ergo and our tactics. E.g., we suppress proof contexts that the tactics library have to do with when calling Ergo.

Calling Ergo from Coq (3)

We rely on Dp to bridge the gap between the logic of Coq (CIC) and the logic of Ergo (PFOL).

Dp selectively and soundly translates terms of Coq to terms of Ergo. E.g. higher-order terms are ignored.

In this way, the programmer can call Ergo interactively within Coq and we can call Ergo from our tactics.

Caveat: the combination of our tactics and Ergo will be available from the next official release of Coq.

The translation by Dp can be interleaved with manual translation. This is indispensable to smoothly combine Ergo and our tactics. E.g., we suppress proof contexts that the tactics library have to do with when calling Ergo.

Calling Ergo from Coq (3)

We rely on Dp to bridge the gap between the logic of Coq (CIC) and the logic of Ergo (PFOL).

Dp selectively and soundly translates terms of Coq to terms of Ergo. E.g. higher-order terms are ignored.

In this way, the programmer can call Ergo interactively within Coq and we can call Ergo from our tactics.

Caveat: the combination of our tactics and Ergo will be available from the next official release of Coq.

The translation by Dp can be interleaved with manual translation. This is indispensable to smoothly combine Ergo and our tactics. E.g., we suppress proof contexts that the tactics library have to do with when calling Ergo.

Calling Ergo from Coq (3)

We rely on Dp to bridge the gap between the logic of Coq (CIC) and the logic of Ergo (PFOL).

Dp selectively and soundly translates terms of Coq to terms of Ergo. E.g. higher-order terms are ignored.

In this way, the programmer can call Ergo interactively within Coq and we can call Ergo from our tactics.

Caveat: the combination of our tactics and Ergo will be available from the next official release of Coq.

The translation by Dp can be interleaved with manual translation. This is indispensable to smoothly combine Ergo and our tactics. E.g., we suppress proof contexts that the tactics library have to do with when calling Ergo.

Tactics for rearranging assertions

Many tactics involve heavy rewriting of assertion terms.

```
entail {x ↦ 3 ** y ↦ 2} {(ex i:Z, y ↦ i) ** x ↦ 1}
Exists 2; Permutation.
```

Internally works as follows:

Scope extension of the existential (rewriting).

```
entail {x ↦ 3 ** y ↦ 2} {ex i:Z, (y ↦ i ** x ↦ 3)}
```

Application of a basic lemma with `x` instantiated to `2`:

```
Axiom ex_destruct:∀p A (q:A → assert) x,
entail p (q x) → entail p (ex A q).
```

results in

```
entail {x ↦ 3 ** y ↦ 2} {y ↦ 2 ** x ↦ 3}
```

The last step by `Permutation` is trivial rewriting.

Tactics for rearranging assertions

Many tactics involve heavy rewriting of assertion terms.

```
entail {x ↦ 3 ** y ↦ 2} {(ex i:Z, y ↦ i) ** x ↦ 1}
Exists 2; Permutation.
```

Internally works as follows:

Scope extension of the existential (rewriting).

```
entail {x ↦ 3 ** y ↦ 2} {ex i:Z, (y ↦ i ** x ↦ 3)}
```

Application of a basic lemma with `x` instantiated to `2`:

```
Axiom ex_destruct:∀p A (q:A → assert) x,
entail p (q x) → entail p (ex A q).
```

results in

```
entail {x ↦ 3 ** y ↦ 2} {y ↦ 2 ** x ↦ 3}
```

The last step by `Permutation` is trivial rewriting.

Tactics for rearranging assertions

Many tactics involve heavy rewriting of assertion terms.

```
entail {x ↦ 3 ** y ↦ 2} {(ex i:Z, y ↦ i) ** x ↦ 1}
Exists 2; Permutation.
```

Internally works as follows:

Scope extension of the existential (rewriting).

```
entail {x ↦ 3 ** y ↦ 2} {ex i:Z, (y ↦ i ** x ↦ 3)}
```

Application of a basic lemma with `x` instantiated to `2`:

```
Axiom ex_destruct:∀p A (q:A → assert) x,
entail p (q x) → entail p (ex A q).
```

results in

```
entail {x ↦ 3 ** y ↦ 2} {y ↦ 2 ** x ↦ 3}
```

The last step by `Permutation` is trivial rewriting.

Tactics for rearranging assertions

Many tactics involve heavy rewriting of assertion terms.

```
entail {x ↦ 3 ** y ↦ 2} {(ex i:Z, y ↦ i) ** x ↦ 1}
Exists 2; Permutation.
```

Internally works as follows:

Scope extension of the existential (rewriting).

```
entail {x ↦ 3 ** y ↦ 2} {ex i:Z, (y ↦ i ** x ↦ 3)}
```

Application of a basic lemma with `x` instantiated to `2`:

```
Axiom ex_destruct:∀p A (q:A → assert) x,
entail p (q x) → entail p (ex A q).
```

results in

```
entail {x ↦ 3 ** y ↦ 2} {y ↦ 2 ** x ↦ 3}
```

The last step by `Permutation` is trivial rewriting.

Tactics for rearranging assertions

Many tactics involve heavy rewriting of assertion terms.

```
entail {x ↦ 3 ** y ↦ 2} {(ex i:Z, y ↦ i) ** x ↦ 1}
Exists 2; Permutation.
```

Internally works as follows:

Scope extension of the existential (rewriting).

```
entail {x ↦ 3 ** y ↦ 2} {ex i:Z, (y ↦ i ** x ↦ 3)}
```

Application of a basic lemma with `x` instantiated to `2`:

```
Axiom ex_destruct:∀p A (q:A → assert) x,
entail p (q x) → entail p (ex A q).
```

results in

```
entail {x ↦ 3 ** y ↦ 2} {y ↦ 2 ** x ↦ 3}
```

The last step by `Permutation` is trivial rewriting.

Tactics for rearranging assertions

Many tactics involve heavy rewriting of assertion terms.

```
entail {x ↦ 3 ** y ↦ 2} {(ex i:Z, y ↦ i) ** x ↦ 1}
Exists 2; Permutation.
```

Internally works as follows:

Scope extension of the existential (rewriting).

```
entail {x ↦ 3 ** y ↦ 2} {ex i:Z, (y ↦ i ** x ↦ 3)}
```

Application of a basic lemma with `x` instantiated to `2`:

```
Axiom ex_destruct:∀p A (q:A → assert) x,
entail p (q x) → entail p (ex A q).
```

results in

```
entail {x ↦ 3 ** y ↦ 2} {y ↦ 2 ** x ↦ 3}
```

The last step by `Permutation` is trivial rewriting.

More efforts are required

Tactics for rearranging assertions

Occurrence selection was one of the most unpleasant efforts.
I.e. to be careful enough not to rewrite irrelevant terms which happen to have the same shape as the target term to be rewritten.
The current implementation is not robust and needs to be improved.

The heavy rewriting blows up the proof term and QED fails due to out of memory.

I believe these difficulties can be solved by engineering efforts and good programming practice; but I am suffering them now.

More efforts are required

Tactics for rearranging assertions

Occurrence selection was one of the most unpleasant efforts.
I.e. to be careful enough not to rewrite irrelevant terms which happen to have the same shape as the target term to be rewritten.
The current implementation is not robust and needs to be improved.

The heavy rewriting blows up the proof term and `QED` fails due to out of memory.

I believe these difficulties can be solved by engineering efforts and good programming practice; but I am suffering them now.

More efforts are required

Tactics for rearranging assertions

Occurrence selection was one of the most unpleasant efforts.
I.e. to be careful enough not to rewrite irrelevant terms which happen to have the same shape as the target term to be rewritten.
The current implementation is not robust and needs to be improved.

The heavy rewriting blows up the proof term and `QED` fails due to out of memory.

I believe these difficulties can be solved by engineering efforts and good programming practice; but I am suffering them now.

Summary(0): Still, it's up to the programmer

How to write specifications matters

How to write the specification of the program to be verified has an great impact on the verification overhead.

- Separate concerns about the functional correctness and the memory separation.

- Write the specification about the functional correctness in the first-order logic, as mush as possible.

Summary(0): Still, it's up to the programmer

How to write specifications matters

How to write the specification of the program to be verified has an great impact on the verification overhead.

- Separate concerns about the functional correctness and the memory separation.

- Write the specification about the functional correctness in the first-order logic, as mush as possible.

Summary (1): A benefit of interactive reasoning

Program verification involves at the same time both verifying the program code and debugging the specification and the code.

When I encountered an unprovable goal, I did not know which of the program code, the specification, or the proof plan is wrong.

That I can identify how facts in the proof context have been introduced and how I have reached the current goal by redoing the script so far is helpful to recover from the failure.

Summary (1): A benefit of interactive reasoning

Program verification involves at the same time both verifying the program code and debugging the specification and the code.

When I encountered an unprovable goal, I did not know which of the program code, the specification, or the proof plan is wrong.

That I can identify how facts in the proof context have been introduced and how I have reached the current goal by redoing the script so far is helpful to recover from the failure.

Summary (1): A benefit of interactive reasoning

Program verification involves at the same time both verifying the program code and debugging the specification and the code.

When I encountered an unprovable goal, I did not know which of the program code, the specification, or the proof plan is wrong.

That I can identify how facts in the proof context have been introduced and how I have reached the current goal by redoing the script so far is helpful to recover from the failure.

Summary(2): A benefit of using separation logic

Separation logic, together with the pureness of Clight expressions, was helpful to keep clearer the interface between the concerns about the functional correctness and the memory separation. (We use CIL as a preprocessor.)

- Memory separation is critical only when reasoning about assignment. Hence the tactics for symbolic evaluation need not take the disjointness into account.

- The disjointness is syntactically visible via the $**$-construct. Thus we could develop tactics for rearranging assertions simply using pattern matching on terms.

Summary(2): A benefit of using separation logic

Separation logic, together with the pureness of Clight expressions, was helpful to keep clearer the interface between the concerns about the functional correctness and the memory separation. (We use CIL as a preprocessor.)

- Memory separation is critical only when reasoning about assignment. Hence the tactics for symbolic evaluation need not take the disjointness into account.

- The disjointness is syntactically visible via the $**$-construct. Thus we could develop tactics for rearranging assertions simply using pattern matching on terms.

Summary(2): A benefit of using separation logic

Separation logic, together with the pureness of Clight expressions, was helpful to keep clearer the interface between the concerns about the functional correctness and the memory separation. (We use CIL as a preprocessor.)

- Memory separation is critical only when reasoning about assignment. Hence the tactics for symbolic evaluation need not take the disjointness into account.

- The disjointness is syntactically visible via the $**$-construct. Thus we could develop tactics for rearranging assertions simply using pattern matching on terms.

Closing

I believe there is still a lot of room to ease the pains in interactive program verification with more engineering efforts and by more aggressively incorporating ideas and tools for automatic verification.

Many thanks to Jean-Christophe Filliâtre and Xavier Leroy.

When Ergo was useful?
Digression

Reasoning about arithmetic is pervasive. Although their proofs may be easy in the paper, the manual proofs in Coq can be painful; from the programmer's viewpoint, it should be nice that Ergo and omega complement each other.

Ergo's ability to instantiate lemmas can prove the following goal.

```
Variable i_curr :Z.
Variables cont1 cont2 :Z -> Z.
Variable cont_eq_sofar:
  ∀i :Z, 0 <= i < i_curr -> cont1 i = cont2 i.
Variable not_cont_gt :cont1 i_curr ⩾ cont2 i_curr.
Variable not_cont_lt :cont2 i_curr ⩾ cont1 i_curr.

Goal ∀i :Z, 0 <= i < i_curr + 1 -> cont1 i = cont2 i.
Proof.  ergo.  Qed.
```

When Ergo was useful?
Digression

Reasoning about arithmetic is pervasive. Although their proofs may be easy in the paper, the manual proofs in Coq can be painful; from the programmer's viewpoint, it should be nice that Ergo and omega complement each other.

Ergo's ability to instantiate lemmas can prove the following goal.

```
Variable i_curr :Z.
Variables cont1 cont2 :Z -> Z.
Variable cont_eq_sofar:
  ∀i :Z, 0 <= i < i_curr -> cont1 i = cont2 i.
Variable not_cont_gt :cont1 i_curr ⊁ cont2 i_curr.
Variable not_cont_lt :cont2 i_curr ⊁ cont1 i_curr.

Goal ∀i :Z, 0 <= i < i_curr + 1 -> cont1 i = cont2 i.
Proof.  ergo.  Qed.
```