Smart & ProvenTools

Proof Techniques That Scale

Stéphane Lescuyer

Prove & Run

Entropy 2019, Stockholm, 16/06/2019

PROVE & RUN

ProvenCore

- ProvenCore is very large verification project
  - → 17000 lines of actual code
  - → 380000 lines of specs and lemmas across 720+ modules and 4 refinement levels
  - → 180000 *hints* to prove 29000 VCs

ProvenCore

- ProvenCore is very large verification project
  - → 17000 lines of actual code
  - → 380000 lines of specs and lemmas across 720+ modules and 4 refinement levels
  - → 180000 *hints* to prove 29000 VCs
- in an interactive proof system, with limited manpower

PROVE & RUN

ProvenCore

- ProvenCore is very large verification project
  - → 17000 lines of actual code
  - → 380000 lines of specs and lemmas across 720+ modules and 4 refinement levels
  - → 180000 *hints* to prove 29000 VCs
- in an interactive proof system, with limited manpower
- how do we achieve and maintain such a large-scale effort?

PROVE & RUN


London


WINTER AIN'T COMING YET...
MUST FINISH PROOF FIRST...
imgflip.com

Scalable approach

- proof by refinement allows *parallel* work

Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools

Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  → Smart is a unique language for code & specs

PROVE & RUN

Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  - → Smart is a unique language for code & specs
  - → C generator supports ghost code, and the linear displicine it enforces is light and natural

Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  → Smart is a unique language for code & specs
  → C generator supports ghost code, and the linear displicine it enforces is light and natural
  → automated and assisted maintenance of proofs

PROVE & RUN

Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  - → Smart is a unique language for code & specs
  - → C generator supports ghost code, and the linear displicine it enforces is light and natural
  - → automated and assisted maintenance of proofs
  - → static analyses for the *framing problem*
    O. F. Andreescu, T. Jensen, S. Lescuyer, B. Montagu. *Inferring frame conditions with static correlation analysis*. PACMPL 3(POPL): 47:1-47:29 (2019).

PROVE & RUN

Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  - → Smart is a unique language for code & specs
  - → C generator supports ghost code, and the linear displicine it enforces is light and natural
  - → automated and assisted maintenance of proofs
  - → static analyses for the *framing problem*
    O. F. Andreescu, T. Jensen, S. Lescuyer, B. Montagu. *Inferring frame conditions with static correlation analysis*. PACMPL 3(POPL): 47:1-47:29 (2019).
  - → makes strict separation of code and specs/proofs possible

PROVE & RUN

Obfuscating code with specs (ADA/Spark2014)

```ada
package body PrefixSum is

  procedure Upsweep (A : in out Input; Output_Space : out Positive) is
    Space : Positive := 1;
    Left  : Natural;
    Right : Natural;
  begin
    while Space < A'Length loop
      pragma Loop_Invariant
        (All_Elements_In (A, Space * Maximum)
          and then
        (Space = 1 or Space = 2 or Space = 4)
          and then
        (for all K in A'Range =>
          (if (K + 1) mod 8 = 0
             and then Space = 8
           then
             A (K) = A'Loop_Entry (0) + A'Loop_Entry (1) +
                     A'Loop_Entry (2) + A'Loop_Entry (3) +
                     A'Loop_Entry (4) + A'Loop_Entry (5) +
                     A'Loop_Entry (6) + A'Loop_Entry (7)
           elsif (K + 1) mod 4 = 0
             and then Space >= 4
           then
             A (K) = A'Loop_Entry (K) + A'Loop_Entry (K-1) +
                     A'Loop_Entry (K-2) + A'Loop_Entry (K-3)
           elsif (K + 1) mod 2 = 0
             and then Space >= 2
           then
             A (K) = A'Loop_Entry (K) + A'Loop_Entry (K-1)
           else
             A (K) = A'Loop_Entry (K))));
      pragma Loop_Variant (Increases => Space);

      Left := Space - 1;

      while Left < A'Length loop
        pragma Loop_Invariant (
          (Left + 1) mod Space = 0
            and then
          All_Left_Elements_In (A, Left, Space * 2 * Maximum)
            and then
          All_Right_Elements_In (A, Left - 1, Space * Maximum)
            and then
          (Left + 1) mod (Space * 2) = Space
            and then
          (if Left >= A'Length then Left = 8 or Left = 9)
            and then
          (for all K in A'Range =>
            (if K in A'First .. Left - Space
               and then (K + 1) mod (2 * Space) = 0
             then
                A (K) = A'Loop_Entry (K) + A'Loop_Entry (K - Space)
             else
                A (K) = A'Loop_Entry (K))));
        pragma Loop_Variant (Increases => Left);

        Right      := Left + Space;
        A (Right) := A (Left) + A (Right);
        Left       := Left + Space * 2;
      end loop;
      Space := Space * 2;
    end loop;
    Output_Space := Space;
  end Upsweep;
```

```ada
  procedure Downsweep
    (Ghost : Input; A : in out Input; Input_Space : in Positive)
  is
    Space : Natural := Input_Space;
    Left  : Natural;
    Right : Natural;
    Temp  : Integer;
  begin
    A (A'Last) := 0;
    Space      := Space / 2;

    while Space > 0 loop
      pragma Loop_Invariant
        ((Space = 4 or Space = 2 or Space = 1)
           and then
         All_Elements_In (A, (4 / Space) * 8 * Maximum)
           and then
         (for all K in A'Range =>
              (if Space = 4 then
                 A (K) = A'Loop_Entry (K)
              elsif Space = 2 and then (K+1) mod 8 = 0 then
                 A (K) = A'Loop_Entry (K) + A'Loop_Entry (K - 2*Space)
              elsif Space = 2 and then (K+1) mod 4 = 0 then
                 A (K) = A'Loop_Entry (K + 2*Space)
              elsif Space = 2 then
                 A (K) = A'Loop_Entry (K)
              elsif Space = 1 and then (K+1) mod 2 = 0 then
                 A (1) = A'Loop_Entry (7) and
                 A (3) = A'Loop_Entry (1) + A'Loop_Entry (7) and
                 A (5) = A'Loop_Entry (7) + A'Loop_Entry (3) and
                 A (7) = A'Loop_Entry (5) + A'Loop_Entry (7)
                     + A'Loop_Entry (3)
              else
                 A (K) = A'Loop_Entry (K))));
      pragma Loop_Variant (Decreases => Space);

      Right := Space * 2 - 1;
      while Right < A'Length loop
        pragma Loop_Invariant (
          (for all K in A'Range =>
            (if K in A'First .. Right - Space * 2 then
              (if (K + 1) mod (2 * Space) = 0 then
                  A (K) = A'Loop_Entry (K) + A'Loop_Entry (K - Space)
               elsif (K + 1) mod Space = 0 then
                  A (K) = A'Loop_Entry (K + Space)
               else
               A (K) = A'Loop_Entry (K))
             else
               A (K) = A'Loop_Entry (K)))
             and then
          (Right + 1) mod (Space * 2) = 0
             and then
          (if Right >= A'Length then
             Right = 9 or Right = 11 or Right = 15));
        pragma Loop_Variant (Increases => Right);

        Left      := Right - Space;
        Temp      := A (Right);
        A (Right) := A (Left) + A (Right);
        A (Left)  := Temp;
        Right     := Right + Space * 2;
      end loop;
      Space := Space / 2;
    end loop;
  end Downsweep;

end PrefixSum;
```

PROVE & RUN

Obfuscating code with specs (Java/VeriFast)

```
/*@

inductive bintree = leaf(int) | node(bintree, bintree);

predicate array_tree0(int[] a, int left, int right, bintree values) =
    right > left + 1 ?
        array_tree0(a, left - (right - left) / 2, left, ?lvalues) &*&
        array_tree0(a, right - (right - left) / 2, right, ?rvalues) &*&
        values == node(lvalues, rvalues) &*&
        right - left == (right - left) / 2 * 2
    :
        array_element(a, left, ?l) &*& array_element(a, right, ?r)
        &*& values == node(leaf(l), leaf(r)) &*&
        right == left + 1;

fixpoint int treesum(bintree t) {
    switch (t) {
        case leaf(n): return n;
        case node(n1, n2): return treesum(n1) + treesum(n2);
    }
}

fixpoint bintree getleft(bintree t) {
    switch (t) {
        case leaf(n): return t;
        case node(t1, t2): return t1;
    }
}

fixpoint bintree getright(bintree t) {
    switch (t) {
        case leaf(n): return t;
        case node(t1, t2): return t2;
    }
}

fixpoint boolean isleaf(bintree t) {
    switch (t) {
        case leaf(n): return true;
        case node(t1, t2): return false;
    }
}

predicate array_tree1(int[] a, int left, int right, bintree values) =
    right > left + 1 ?
        array_tree1(a, left - (right - left) / 2, left, ?lvalues) &*& array_element(a, left, treesum(lvalues)) &*&
        array_tree1(a, right - (right - left) / 2, right, ?rvalues) &*&
        values == node(lvalues, rvalues) &*&
        right - left == (right - left) / 2 * 2
    :
        array_element(a, left, ?l) &*& getleft(values) == leaf(l) &*& isleaf(getright(values)) == true
        &*& !isleaf(values) &*&
        right == left + 1;

predicate array_tree2(int[] a, int left, int right, int leftSum, bintree values) =
    right > left + 1 ?
        array_tree2(a, left - (right - left) / 2, left, leftSum, ?lvalues) &*&
        array_tree2(a, right - (right - left) / 2, right, leftSum + treesum(lvalues), ?rvalues) &*&
        values == node(lvalues, rvalues) &*&
        right - left == (right - left) / 2 * 2
    :
        array_element(a, left, leftSum) &*& array_element(a, right, ?r) &*& getleft(values) == leaf(r - leftSum)
        &*& !isleaf(values) &*& isleaf(getright(values)) == true &*&
        right == left + 1;

fixpoint list<int> prefixSums(int leftSum, bintree b) {
    switch (b) {
        case leaf(n): return cons(leftSum, nil);
        case node(t1, t2): return append(prefixSums(leftSum, t1), prefixSums(leftSum + treesum(t1), t2));
    }
}
```

```
lemma void array_tree2_prefixsums()
    requires array_tree2(?a, ?left, ?right, ?leftSum, ?values);
    ensures array_slice(a, left + 1 - (right - left), right + 1, prefixSums(leftSum, values));
{
    open array_tree2(_, _, _, _, _);
    if (right > left + 1) {
        array_tree2_prefixsums();
        array_tree2_prefixsums();
    } else {
        assert right == left + 1;
        switch (values) {
            case leaf(n): assert true;
            case node(t1, t2):
                switch (t2) {
                    case leaf (n): assert true;
                    case node(t21, t22): assert true;
                }
        }
    }
}

@*/

class PrefixSumRec {

    private int[] a;

    PrefixSumRec(int [] a)
        //@ requires true;
        //@ ensures this.a |-> a;
    {
        this.a = a;
    }


    public void upsweep(int left, int right)
        //@ requires a |-> ?a_ &*& array_tree0(a_, left, right, ?values);
        //@ ensures a |-> a_ &*& array_tree1(a_, left, right, values) &*& array_element(a_, right, treesum(values));
    {
        //@ open array_tree0(_, _, _, _);
        if (right > left+1) {
            int space = right - left;
            upsweep(left-space/2,left);
            upsweep(right-space/2,right);
        }
        a[right] = a[left]+a[right];
        //@ close array_tree1(a_, left, right, values);
    }


    public void downsweep(int left, int right)
        //@ requires a |-> ?a_ &*& array_tree1(a_, left, right, ?values) &*& array_element(a_, right, ?leftSum);
        //@ ensures a |-> a_ &*& array_tree2(a_, left, right, leftSum, values);
    {
        //@ open array_tree1(_, _, _, _);
        int tmp = a[right];
        a[right] = a[right] + a[left];
        a[left] = tmp;
        if (right > left+1) {
            int space = right - left;
            downsweep(left-space/2,left);
            downsweep(right-space/2,right);
        }
        //@ close array_tree2(a_, left, right, leftSum, values);
    }
}
```

Obfuscating code with specs (Why3)

```
module PrefixSumRec
   use int.Int
   use int.ComputerDivision
   use int.Power

   use map.Map
   use array.Array
   use array.ArraySum

(** {2 Preliminary lemmas on division by 2 and power of 2} *)

 (** Needed for the proof of phase1_frame and phase1_frame2 *)
 lemma Div_mod_2:
   forall x:int. x >= 0 -> x >= 2 * div x 2 >= x-1

 (** The step must be a power of 2 *)
 predicate is_power_of_2 (x:int) = exists k:int. (k >= 0 /\ x = power 2 k)

 (* needed *)
 lemma is_power_of_2_1:
   forall x:int. is_power_of_2 x -> x > 1 -> 2 * div x 2 = x

(**
{2 Modeling the "upsweep" phase }
*)

 (** Shortcuts *)
 function go_left (left right:int) : int =
   let space = right - left in left - div space 2

 function go_right (left right:int) : int =
   let space = right - left in right - div space 2

(**
  Description in a purely logic way the effect of the first phase
    "upsweep" of the algorithm.  The second phase "downsweep" then
    traverses the array in the same way than the first phase. Hence,
    the inductive nature of this definition is not an issue. *)

 inductive phase1 (left:int) (right:int) (a0: array int) (a: array int) =
   | Leaf : forall left right:int, a0 a : array int.
      right = left + 1 ->
      a[left] = a0[left] ->
      phase1 left right a0 a
   | Node : forall left right:int, a0 a : array int.
      right > left + 1 ->
      phase1 (go_left left right) left a0 a ->
      phase1 (go_right left right) right a0 a ->
      a[left] = sum a0 (left-(right-left)+1) (left+1) ->
      phase1 left right a0 a

 (** Frame properties of the "phase1" inductive *)

 (** frame lemma for "phase1" on fourth argument.
    needed to prove both upsweep, downsweep and compute_sums
  *)
 let rec lemma phase1_frame (left right:int) (a0 a a' : array int) : unit
   requires { forall i:int. left-(right-left) < i < right ->
              a[i] = a'[i]}
   requires { phase1 left right a0 a }
   variant { right-left }
   ensures { phase1 left right a0 a' } =
   if right > left + 1 then begin
     phase1_frame (go_left left right) left a0 a a';
     phase1_frame (go_right left right) right a0 a a'
   end

 (** frame lemma for "phase1" on third argument.
    needed to prove upsweep and compute_sums
  *)
 let rec lemma phase1_frame2 (left right:int) (a0 a0' a : array int) : unit
   requires { forall i:int. left-(right-left) < i < right ->
              a0[i] = a0'[i]}
   requires { phase1 left right a0 a }
   variant { right-left }
   ensures { phase1 left right a0' a } =
   if right > left + 1 then begin
     phase1_frame2 (go_left left right) left a0 a0' a;
     phase1_frame2 (go_right left right) right a0 a0' a
   end
```

```
(** {2 The upsweep phase}

  First function: modify partially the table and compute some
      intermediate partial sums

*)

  let rec upsweep (left right: int) (a: array int)
    requires { 0 <= left < right < a.length }
    requires { -1 <= left - (right - left) }
    requires { is_power_of_2 (right - left) }
    variant { right - left }
    ensures { phase1 left right (old a) a }
    ensures { let space = right - left in
      a[right] = sum (old a) (left-space+1) (right+1) /\
      (forall i: int. i <= left-space -> a[i] = (old a)[i]) /\
      (forall i: int. i > right -> a[i] = (old a)[i]) }
  = let space = right - left in
    if right > left+1 then begin
      upsweep (left - div space 2) left a;
      upsweep (right - div space 2) right a;
      assert { phase1 (left - div space 2) left (old a) a };
      assert { phase1 (right - div space 2) right (old a) a };
      assert { a[left] = sum (old a) (left-(right-left)+1) (left+1) };
      assert { a[right] = sum (old a) (left+1) (right+1) }
    end;
    a[right] <- a[left] + a[right];
    assert {
      right > left+1 -> phase1 (left - div space 2) left (old a) a };
    assert {
      right > left+1 -> phase1 (right - div space 2) right (old a) a }

(** {2 The downsweep phase} *)

 (** The property we ultimately want to prove *)
 predicate partial_sum (left:int) (right:int) (a0 a : array int) =
   forall i : int. (left-(right-left)) < i <= right -> a[i] = sum a0 0 i

 (** Second function: complete the partial using the remaining intial
     value and the partial sum already computed *)
 let rec downsweep (left right: int) (ghost a0 : array int) (a : array int)
   requires { 0 <= left < right < a.length }
   requires { -1 <= left - (right - left) }
   requires { is_power_of_2 (right - left) }
   requires { a[right] = sum a0 0 (left-(right-left) + 1) }
   requires { phase1 left right a0 a }
   variant  { right - left }
   ensures { partial_sum left right a0 a }
   ensures { forall i: int. i <= left-(right-left) -> a[i] = (old a)[i] }
   ensures { forall i: int. i > right -> a[i] = (old a)[i] }
 = let tmp = a[right] in
   assert { a[right] = sum a0 0 (left-(right-left) + 1) };
   assert { a[left] = sum a0 (left-(right-left)+1) (left+1) };
   a[right] <- a[right] + a[left];
   a[left] <- tmp;
   assert { a[right] = sum a0 0 (left + 1) };
   if right > left+1 then
   let space = right - left in
   assert { phase1 (go_left left right) left a0 (old a) };
   assert { phase1 (go_right left right) right a0 (old a) };
   assert { phase1 (go_right left right) right a0 a };
   downsweep (left - div space 2) left a0 a;
   assert { phase1 (go_right left right) right a0 a };
   downsweep (right - div space 2) right a0 a;
   assert { partial_sum (left - div space 2) left a0 a };
   assert { partial_sum (right - div space 2) right a0 a }

(** {2 The main procedure} *)

 let compute_sums a
   requires { a.length >= 2 }
   requires { is_power_of_2 a.length }
   ensures { forall i : int. 0 <= i < a.length -> a[i] = sum (old a) 0 i }
 = let a0 = ghost (copy a) in
   let l = a.length in
   let left = div l 2 - 1 in
   let right = l - 1 in
   upsweep left right a;
   (* needed for the precondition of downsweep *)
   assert { phase1 left right a0 a };
   a[right] <- 0;
   downsweep left right a0 a;
   (* needed to prove the post-condition *)
   assert { forall i : int.
     left-(right-left) < i <= right -> a[i] = sum a0 0 i }

end
```

PROVE & RUN

Separation of code and specifications

→ better readability

→ simpler dependencies (important for CC evaluation)

→ separation of concerns

PROVE & RUN

Separation of code and specifications

→ better readability

→ simpler dependencies (important for CC evaluation)

→ separation of concerns

How to achieve separation?

→ do not use Hoare-style contracts

$$\{P\}\ f\ \{Q\}$$

becomes a single separate lemma

$$P \rightarrow f \rightarrow Q$$

Separation of code and specifications

- $\rightarrow$ better readability
- $\rightarrow$ simpler dependencies (important for CC evaluation)
- $\rightarrow$ separation of concerns

How to achieve separation?

- $\rightarrow$ do not use Hoare-style contracts

$$\{P1 \wedge P2\}\, f\, \{Q1 \wedge Q2\}$$

becomes a single separate lemma

$$P1 \rightarrow P2 \rightarrow f \rightarrow Q1 \wedge Q2$$

Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?

→ do not use Hoare-style contracts

$$\{P1 \land P2\}\, f\, \{Q1 \land Q2\}$$

becomes two separate lemmas

$$P1 \rightarrow P2 \rightarrow f \rightarrow Q1$$
$$P1 \rightarrow P2 \rightarrow f \rightarrow Q2$$

Separation of code and specifications

- $\rightarrow$ better readability
- $\rightarrow$ simpler dependencies (important for CC evaluation)
- $\rightarrow$ separation of concerns

How to achieve separation?

- $\rightarrow$ do not use Hoare-style contracts

$$\{P1 \wedge P2\}\, f\, \{Q1 \wedge Q2\}$$

becomes two separate lemmas

$$P1 \rightarrow f \rightarrow Q1$$
$$P2 \rightarrow f \rightarrow Q2$$

Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?

→ do not use Hoare-style contracts

$$\{P1 \wedge P2\}\, f\, \{Q1 \wedge Q2\}$$

becomes two separate lemmas

$$P1 \rightarrow f \rightarrow Q1$$
$$P2 \rightarrow f \rightarrow Q2$$

→ how to get rid of loop invariants? *(without getting rid of loops)*

Inductive loop invariants

- loop invariants hold at every iteration
- inductive loop properties are preserved by the loop

→ reasoning about a loop means finding *inductive loop invariants*

PROVE & RUN

Inductive loop invariants

- loop invariants hold at every iteration
- inductive loop properties are preserved by the loop

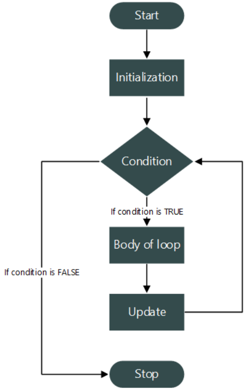
$\rightarrow$ reasoning about a loop means finding *inductive loop invariants*

Let $\mathcal{I}$ be the set of inductive loop invariants

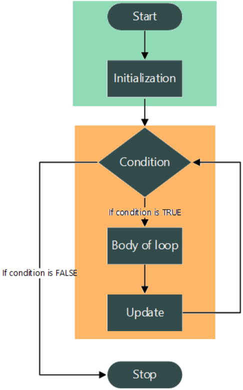
- the conjunction of two inductive invariants is an inductive invariant
- $(\mathcal{I}, \supseteq)$ form a lattice, its join operation is the conjunction operator $\wedge$
- its bottom element is *True*, and its maximum element $\bigwedge_{I \in \mathcal{I}} I$ is what we call the most general inductive invariant (*MGI*)

PROVE & RUN

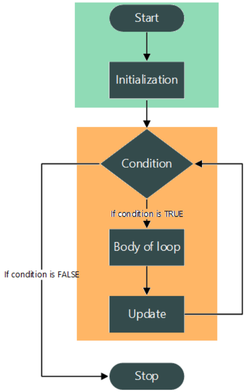
Generating the MGI

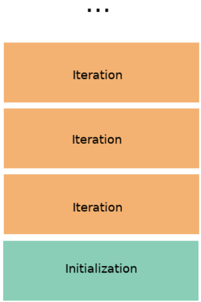


Generating the MGI

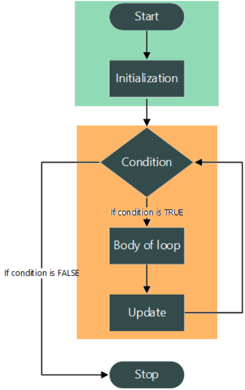


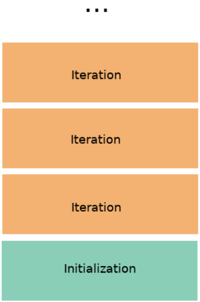
Generating the MGI





Generating the MGI





→ MGI can be defined as the inductive closure of the relation which contains the loop initialization and which is closed by applying an iteration of the loop body

MGIs in practice

- this works with loops in sequence or even *nested loops*

MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track

    → tracking less variables means the MGI is not so general anymore, but applies to more loops

MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track
  - $\rightarrow$ tracking less variables means the MGI is not so general anymore, but applies to more loops
- MGIs allow sharing proofs between “similar” loops
  - $\rightarrow$ if the MGI of some loop $\mathcal{L}$ is an invariant of some loop $\mathcal{L}'$, all invariants of $\mathcal{L}$ are invariants of $\mathcal{L}'$

PROVE & RUN

MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track
  - $\rightarrow$ tracking less variables means the MGI is not so general anymore, but applies to more loops
- MGIs allow sharing proofs between “similar” loops
  - $\rightarrow$ if the MGI of some loop $\mathcal{L}$ is an invariant of some loop $\mathcal{L}'$, all invariants of $\mathcal{L}$ are invariants of $\mathcal{L}'$
- MGI generation need not be trusted

PROVE & RUN

MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track
  - → tracking less variables means the MGI is not so general anymore, but applies to more loops
- MGIs allow sharing proofs between "similar" loops
  - → if the MGI of some loop $\mathcal{L}$ is an invariant of some loop $\mathcal{L}'$, all invariants of $\mathcal{L}$ are invariants of $\mathcal{L}'$
- MGI generation need not be trusted
- we use a similar trick to delay *termination proofs* for recursive predicates or internal loops

PROVE & RUN

Conclusion

- good tooling is key to large verification project like ProvenCore
- ProvenTools is designed to meet our ends and make the project manageable
- we value separation of code and specs
  → original way of dealing with loop reasoning

Conclusion

- good tooling is key to large verification project like ProvenCore
- ProvenTools is designed to meet our ends and make the project manageable
- we value separation of code and specs

  → original way of dealing with loop reasoning

```
type ints<S> = array<S, nat>;

public upsweep(ints<S> ints0, ints<S> ints+, nat spc+) -> [true, fail]
/// Up-Sweep phase of the prefix sum algorithm
program {{ nat n }} {
    ints := ints0;
    succ(ZERO, spc+);
    length(ints0, n+);
    while
    guarded guard(ints, spc, n)
    inductive inv(old@ints, ints, spc, n)
    {{ nat left }} {
        [false : exit] lt(spc, n);
        [ZERO : fail] pred(spc, left+);
        while
        guarded guard2(ints, spc, left, n) else fail
        inductive inv2(old@ints, ints, spc, left, n)
        {{ nat right, nat l, nat r, nat d }}
        [out_of_bounds : fail] {
            [false : exit] lt(left, n);
            plus(left, spc, right+);
            get(ints, left, l+);
            get(ints, right, r+);
            plus(l, r, r+);
            set(ints, right, r, ints+);
            plus(spc, spc, d+);
            plus(left, d, left+);
        }
        plus(spc, spc, spc+);
    }
}
```

```
public downsweep(ints<S> ints0, nat spc0, ints<S> ints+) -> [true, fail]
/// Down-Sweep phase of the prefix sum algorithm
program {{ nat spc, nat two, nat n, nat last }} {
    succ(ZERO, two+); succ(two, two+);
    [div_by_zero : fail] div_eucl(spc0, two, spc+);
    length(ints0, n+);
    [ZERO : fail] pred(n, last+);
    [out_of_bounds : fail] set(ints0, last, ZERO, ints+);

    while
    guarded guard(ints, spc, n)
    inductive inv(old@ints, ints, spc, n)
    {{ nat right }} {
        [false : exit] lt(ZERO, spc);
        plus(spc, spc, right+);
        [ZERO : fail] pred(right, right+);

        while
        guarded guard2(ints, spc, right, n) else fail
        inductive inv2(old@ints, ints, spc, right, n)
        {{ nat left, nat tmp, nat l, nat r, nat d }}
        [out_of_bounds : fail] {
            [false : exit] lt(right, n);
            minus(right, spc, left+);
            get(ints, right, tmp+);
            get(ints, left, l+);
            plus(l, tmp, r+);
            set(ints, right, r, ints+);
            set(ints, left, tmp, ints+);
            plus(spc, spc, d+);
            plus(right, d, right+);
        }
        [div_by_zero : fail] div_eucl(spc, two, spc+);
    }
}

public main(ints<S> ints0, ints<S> ints+) -> [true, fail]
/// Computes the prefix-sum of the given array, in place
program {{ nat spc }} {
    upsweep(ints0, ints+, spc+);
    downsweep(ints, spc, ints+);
}
```

PROVE & RUN