The WhyRel Prototype for Relational Verification

Ramana Nagasamudram[1], Anindya Banerjee[2], David A. Naumann[1]

[1]Stevens Institute of Technology, Hoboken, USA
[2]IMDEA Software Institute, Madrid, Spain

[1]Partially supported by NSF Award 1718713

MAJORIZATION

For $n \geq 4$, $n! > 2^n$

$C_0 \triangleq$ $i := n$;
$z := 24$;
while $(i > 4)$ do
$z := z \times i$;
$i := i - 1$;

$C_1 \triangleq$ $i := n$;
$z := 16$;
while $(i > 4)$ do
$z := z \times 2$;
$i := i - 1$;

MAJORIZATION

For $n \geqslant 4$, $n! > 2^n$

$c_0 \mathrel{\hat{=}}$ `i := n;`
`z := 24;`
`while (i > 4) do`
`z := z x i;`
`i := i-1;`

$c_1 \mathrel{\hat{=}}$ `i := n;`
`z := 16;`
`while (i > 4) do`
`z := z x 2;`
`i := i-1;`

Relational judgement: $c_0 | c_1 : n \geqslant 4 \wedge n \ddot{=} n \Rightarrow z \ddot{>} z$

prerelation: same values for n on "both" sides + $\geqslant 4$

postrelation: z on left > z on right

MAJORIZATION

For $n \geqslant 4$, $n! > 2^n$

$c_0 \mathrel{\hat{=}}$ $i := n;$
$z := 24;$
while $(i > 4)$ do
$z := z \times i;$
$i := i - 1;$

$c_1 \mathrel{\hat{=}}$ $i := n;$
$z := 16;$
while $(i > 4)$ do
$z := z \times 2;$
$i := i - 1;$

Relational judgement: $c_0 | c_1 : n \geqslant 4 \wedge n \ddot{=} n \approx z \ddot{>} z$

To prove: show c_0 computes $n!$, c_1 computes 2^n

But there is a simpler way

MAJORIZATION — USING ALIGNMENT

For $n \geqslant 4$, $n! > 2^n$

$c_0 \mathrel{\hat{=}}$ `i := n;`
`z := 24;`
`while (i > 4) do`
`z := z x i;`
`i := i-1;`

$c_1 \mathrel{\hat{=}}$ `i := n;`
`z := 16;`
`while (i > 4) do`
`z := z x 2;`
`i := i-1;`

same # of iters. for $n \ddot{=} n$

Relational inv:

$i \ddot{=} i \wedge z \ddot{>} z$

Now in lin. arith

Relational judgement: $c_0 | c_1 : n \geqslant 4 \wedge n \ddot{=} n \approx z \ddot{>} z$

MAJORIZATION — EXPRESSING ALIGNMENT

For $n \geq 4$, $n! > 2^n$

Biprogram capturing a lockstep alignment of c_0, c_1:

$(i := n \mid i := n)$;

$(z := 24 \mid z := 16)$;

while $(i > 4 \mid i > 4)$ do inv $\{i \stackrel{=}{=} i \wedge z \stackrel{>}{>} z\}$

$(z := z \times i \mid z := z \times 2)$;

$(i := i - 1 \mid i := i - 1)$;

■ left prog
■ right prog

Relational judgement: $c_0 | c_1 : n \geq 4 \wedge n \stackrel{=}{=} n \Rightarrow z \stackrel{>}{>} z$

MAJORIZATION — EXPRESSING ALIGNMENT

For $n \geq 4$, $n! > 2^n$

Biprogram capturing a lockstep alignment of c_0, c_1:

```
(i := n | i := n);          ← aligned assignments to i
(z := 24 | z := 16);
while (i>4 | i>4) do        inv {i ≡ i ∧ z ≥̈ z}
    (z := z x i | z := z x2);
    (i := i - 1 | i := i - 1);
```

← lockstep aligned loop

↑ relational invariant

"|" indicates pairing

Relational judgement: $c_0 | c_1 : n \geq 4 \wedge n \ddot{\equiv} n \Rightarrow z \ddot{\geq} z$

The WhyRel Prototype

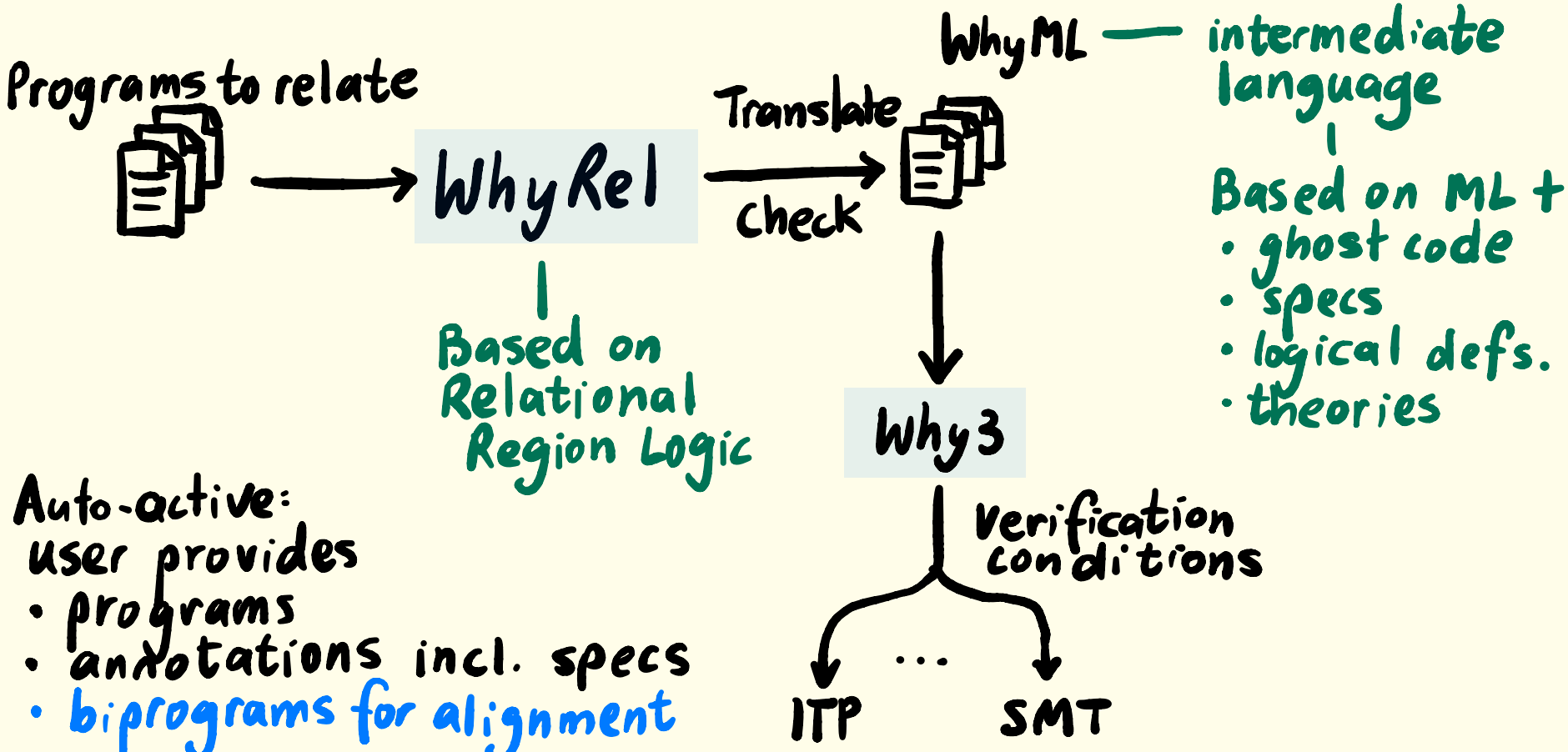
WhyRel is an auto-active tool for relational verification

Supports
- objects on the heap
- modular (relational) reasoning about procedures
- encapsulation
- data-dependent alignment (see paper)

Is backed by extensive theoretical development [TOPLAS '22]

Agenda
- Example introducing key features
- Walkthrough and encoding details
- Evaluation and prospects

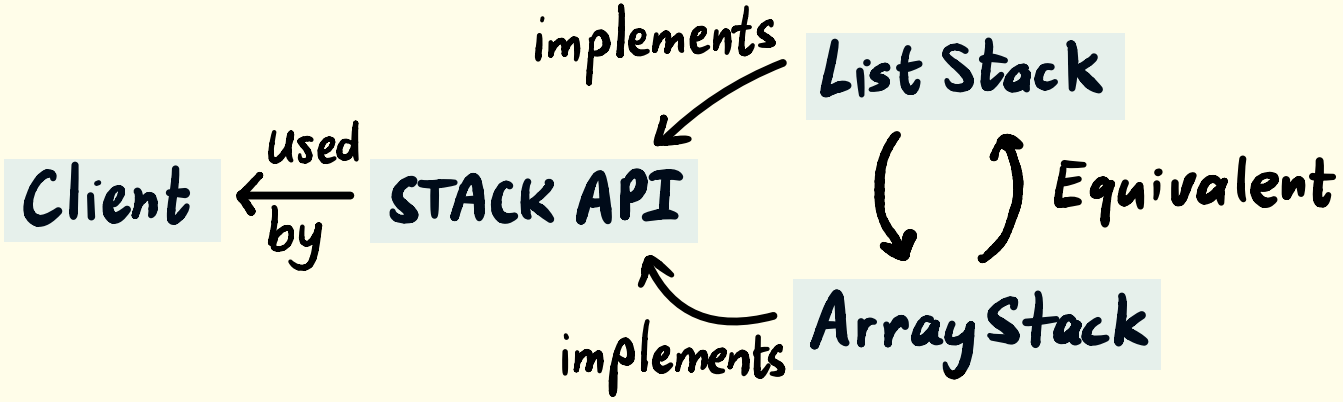
Overview of WhyRel



Auto-active:
user provides
- programs
- annotations incl. specs
- biprograms for alignment

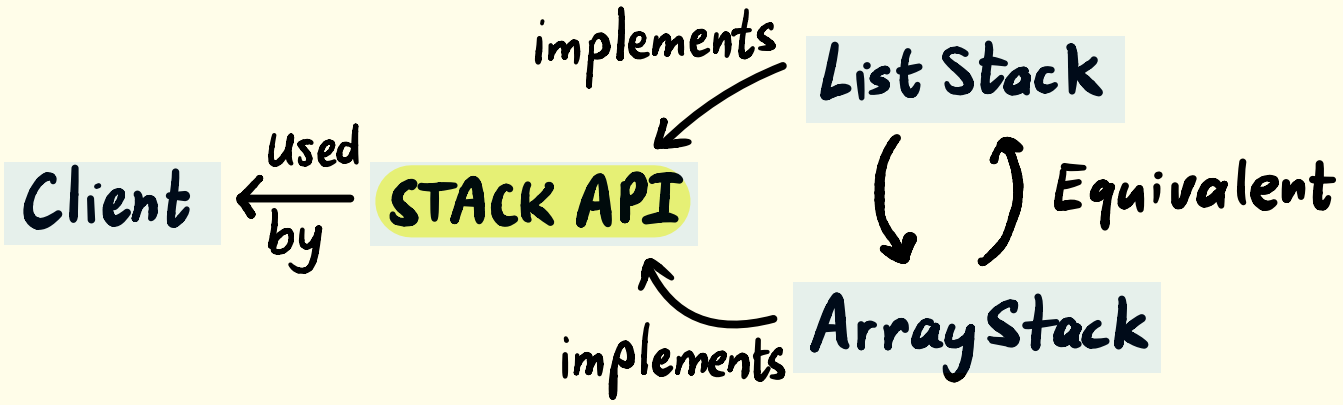
AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks



AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks

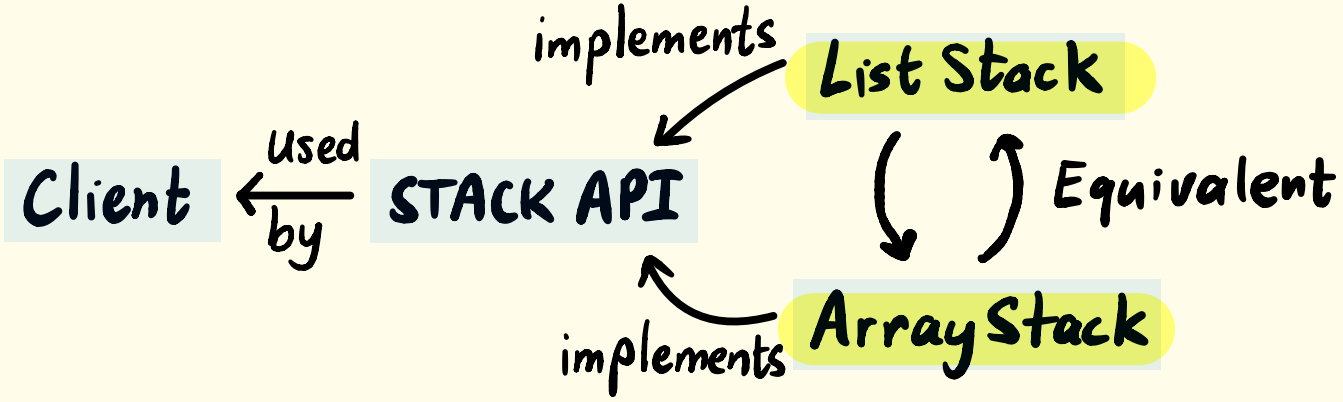


Stack interface exposes public invariants clients rely on

e.g. no aliasing among distinct stack objects

AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks

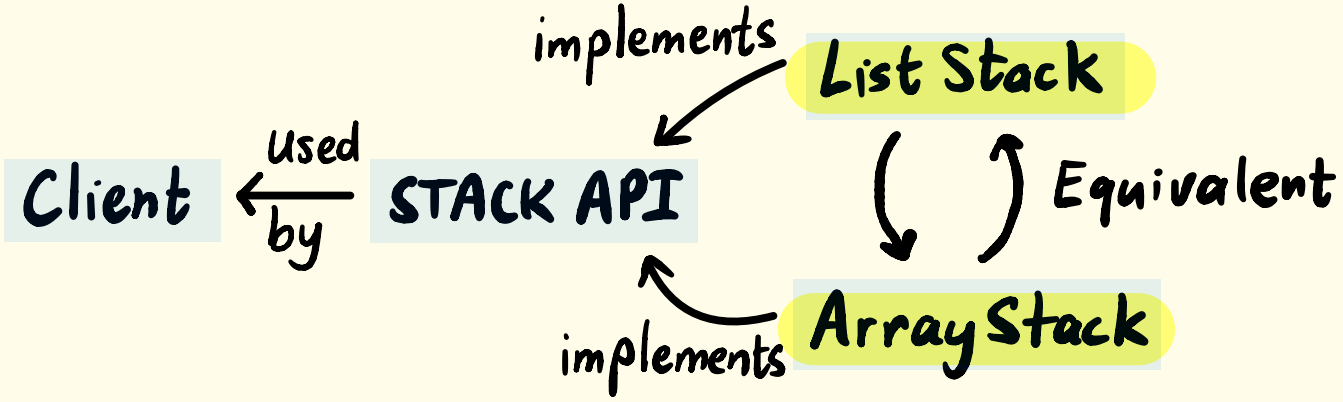


ListStack and ArrayStack rely on private invariants

e.g. in ArrayStack: stack objs. use a fixed size array

AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks



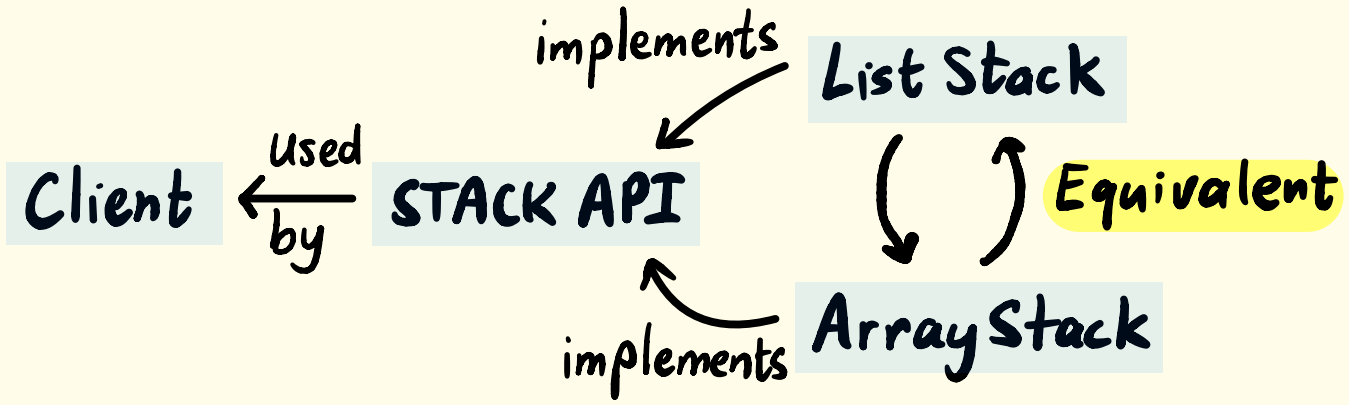
ListStack and ArrayStack rely on private invariants

e.g. in ArrayStack: stack objs. use a fixed size array

Provided these only depend on encapsulated state, clients are exempt from reasoning about them

AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks



Equivalent (rel. prop.):

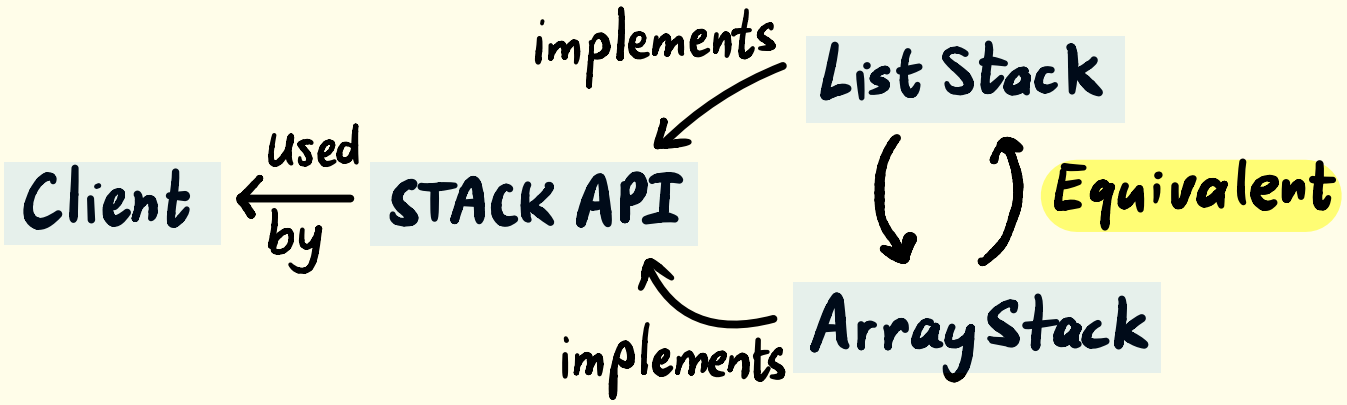
Beh. of client linked against ListStack

same as

Beh. of client linked against ArrayStack

AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks



Equivalent (rel. prop.):

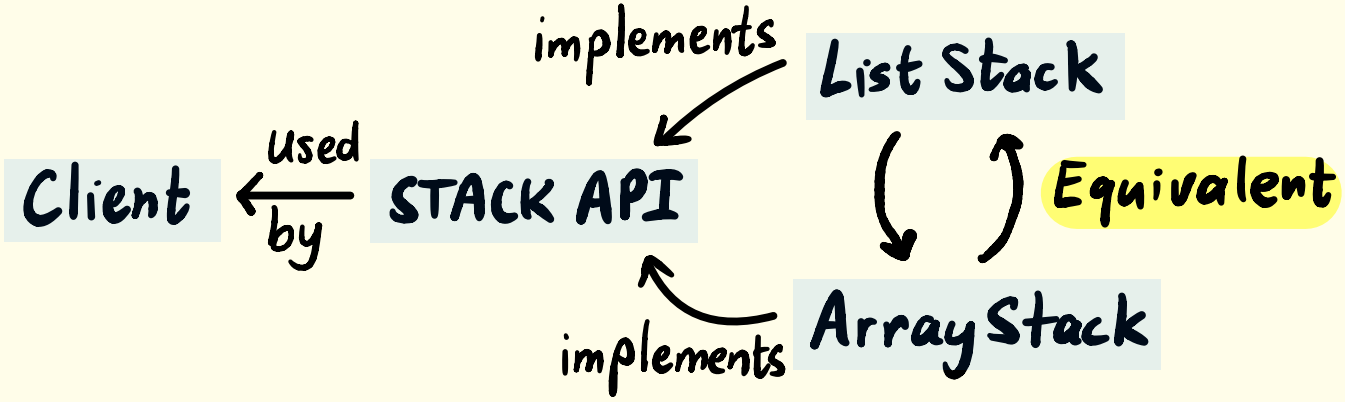
Requires API methods in both module are locally equivalent

"related" inputs → "related" outputs

Relation needs to be identity on client-visible state

AN EXAMPLE IN WHYREL

Equivalence of two modules implementing stacks



Equivalent (rel. prop.):

Representations are different: need a coupling relation on encap'd state relating objs. in both modules

INTERFACES AND SPECS IN WHYREL

```
interface STACK =

    public maxSize : int

    class Stack {
        size: int;
        ghost abs: intlist; }

    meth push (self: Stack, k:int)

        requires {self.size < maxSize}

        ensures {self.abs = cons(k, old(self.abs))}

        ensures {self.size = old(self.size) + 1}
```

REGIONS AND FRAME CONDITIONS

```
interface STACK =

    public maxSize : int

    class Stack {
        size: int;
        ghost abs: intlist;
        ghost rep: rgn; }

    meth push (self: Stack, k: int)

        requires {...} ensures {...}
```

A region (rgn) is a set of references

← "rep" holds refs to objects notionally owned by the stack

REGIONS AND FRAME CONDITIONS

```
interface STACK =

    public maxSize : int

    class Stack {
        size: int;
        ghost abs: intlist;
        ghost rep: rgn; }

    meth push (self: Stack, k: int)

        requires {...}  ensures {...}

        effects { rw {self}`any, {self}`rep`any

                  rd self, k, maxSize }
```

A region (rgn) is a set of references

← "rep" holds refs to objects notionally owned by the stack

↗ r/w any field of self

r/w any field of any obj in self.rep ↗

DYNAMIC BOUNDARIES AND ENCAPSULATION

```
interface STACK =
    public maxSize : int
    public pool : rgn
    boundary {pool, pool`any, pool`rep`any, maxSize}
    class Stack {
        size: int;
        ghost abs: intlist;
        ghost rep: rgn; }

    meth push (self: Stack, k:int)
        ...
```

DYNAMIC BOUNDARIES AND ENCAPSULATION

```
interface STACK =
    public maxSize : int
    public pool : rgn
    boundary {pool, pool`any, pool`rep`any, maxSize}
    class Stack {
        size: int;
        ghost abs: intlist;
        ghost rep: rgn; }

    meth push (self: Stack, k:int)
        ...
```

Heap encapsulation at the granularity of modules

← objs owned by the module

Client's can't directly r/w locs designated by the boundary

PATTERNS OF HEAP USAGE AND PUBLIC INVS

interface STACK =

public maxSize : int public pool : rgn

boundary {pool, pool`any, pool`rep`any , maxSize}

class Stack { ... }

meth push (self: Stack, k: int)

public invariant : $\forall s : Stack \in pool.$

$0 \leq s.size \leq maxSize$

$\wedge (\forall t : Stack \in pool.\ s \neq t \Rightarrow s.rep \cap t.rep \subseteq \{null\})$

Client reasoning relies on public invariants : for e.g.

modifying one stack does not modify another

IMPLEMENTATIONS AND UNARY VERIF

module ArrayStack : STACK =

class Stack {... ; arr: int array; top: int; }

private invariant: ∀s:Stack∈pool.

s.size = top+1 ∧ s.arr.length = maxSize ∧ ...

Obligations:

- Methods conform to specs including frame conditions and preserve pub/priv. invariants
- Private invariants framed by boundary

Private invariants not included in specs used by client

EQUIVALENCE OF STACK MODULES

```
module ArrayStack =
    class Stack {...
        top : int;
        arr : int array;}

    meth push(...)
```

```
module ListStack =
    class Node {... ; nxt : Node;}

    class Stack {... ; head : Node;}

    meth push(...)
```

$$\text{coupling} : \forall s : \text{Stack} \in \text{pool} \mid s' : \text{Stack} \in \text{pool}.$$
$$s \doteq s' \Rightarrow s.\text{abs} \doteq s'.\text{abs}$$

EQUIVALENCE OF STACK MODULES

```
module ArrayStack =
    class Stack {...
        top : int;
        arr : int array;}

    meth push (...)
```

```
module ListStack =
    class Node {... ; nxt : Node;}

    class Stack {... ; head : Node;}

    meth push (...)
```

$$\text{coupling} : \forall s : \text{Stack} \in \text{pool} \mid s' : \text{Stack} \in \text{pool}.$$
$$s \doteq s' \Rightarrow s.\text{abs} \doteq s'.\text{abs}$$

```
push (self: Stack, k: int) | push (self: Stack, k: int):
    requires {self ≐ self ∧ k ≐ k ∧ coupling ∧ Both (unary pre)}
    ensures {coupling ∧ Both (unary post)}
```

EQUIVALENCE OF STACK MODULES

module ArrayStack = ... module ListStack = ...

$$\text{coupling} : \forall s : \text{Stack} \in \text{pool} \mid s' : \text{Stack} \in \text{pool}.$$
$$s \doteq s' \Rightarrow s.\text{abs} \doteq s'.\text{abs}$$

User provides: biprograms + rel. equiv. specs + coupling

Obligations:

- Coupling framed by boundaries of underlying modules
- Biprograms conform to rel. specs + preserve coupling

EQUIVALENCE OF STACK MODULES

module ArrayStack = ... module ListStack = ...

coupling : $\forall s : \text{Stack} \in \text{pool} \mid s' : \text{Stack} \in \text{pool}.$

$s \doteq s' \Rightarrow s.\text{abs} \doteq s'.\text{abs}$

User provides: biprograms + rel. equiv. specs + coupling

Obligations:

- Coupling framed by boundaries of underlying modules
- Biprograms conform to rel. specs + preserve coupling

For any client C: (import ArrayStack; C) $\approx$ (import ListStack; C)

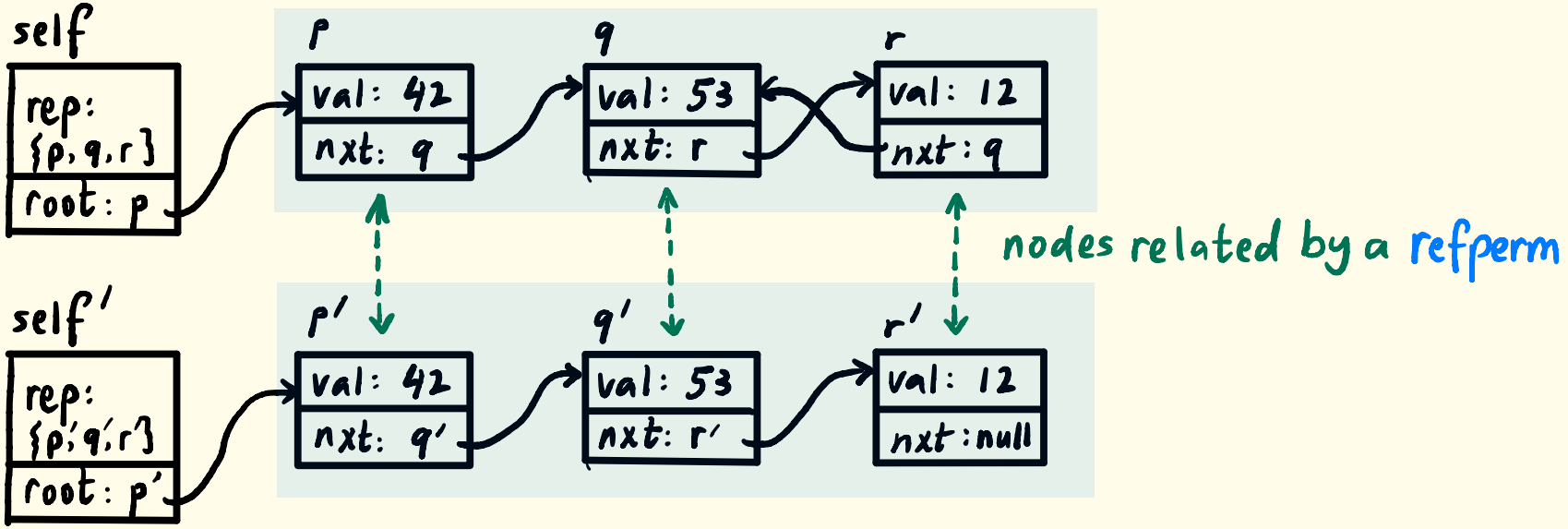
provided C respects encapsulation

AGREEMENT FORMULAS

For expressing agreement on unbounded pointer structures

private invariant : $\forall s \in List \in pool.$
$s.rep^{\bullet} nxt \subseteq s.rep \wedge s.root \in s.rep$

In some equivalence spec: $\mathbb{A}$ self.rep$^{\bullet}$ value



WhyRel Usage and Encoding Details

EVALUATION AND CASE STUDIES

To evaluate WhyRel, we've performed case studies

- Equivalence of two implementations of priority queues
  Dijkstra's SSSP algorithm as client
- Equivalence of two implementations of union find
  Kruskal's MST algorithm as client
- Correctness of optimizations such as loop-tiling
- Noninterference examples + other general rel. properties from recent lit.

diff. style couplings

Verification conditions discharged purely using SMT

Larger examples take ~30 min to verify; others ~5-10 min on an i5-6500 machine with 32 GB RAM

RELATED WORK

Many mature tools for relational verification with varying degrees of automation

- ReLoC : Prove refinements of concurrent programs; Iris
- SecCSL : Info flow for concurrent C
- REFINITY : Based on the mature KeY framework
  Similar: dynamic frames for heap reasoning
- Eilers et al. : Tool based on VIPER; encoding of products to facilitate procedure-modular reasoning
- Descartes : k-safety properties of Java programs
  Based on Cartesian Hoare Logic

Automated & infer alignments: Unno et al., Churchill et al., Shemer et al.

Many more! SymDiff for prog. diffs, LLRêve: regression verif...

Take Aways

- Auto-active rel. verif. of heap-manipulating programs is feasible and amenable to automation
- Auto-active = specs + invariants + biprograms for alignment
- Our work: a proof-of-principle prototype
  - based on a logic with encap and info hiding that has been proven sound
  - applications: noninterference, optimizations, representation independence

Stable release of tool: zenodo.org/record/7308342

Development version: github.com/dnaumann/RelRL

Future work

- Automation to support constructing biprograms
- Better use of abstraction and tactics in Why3
    - Expected to speed up verification;
      avoid superfluous proof obligations
- Machine checked proofs connecting tool with logic

SUPPLEMENTAL SLIDES

Contents

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List):int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List):int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

Satisfies a noninterference spec:

Does not depend on values of nonpublic nodes

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

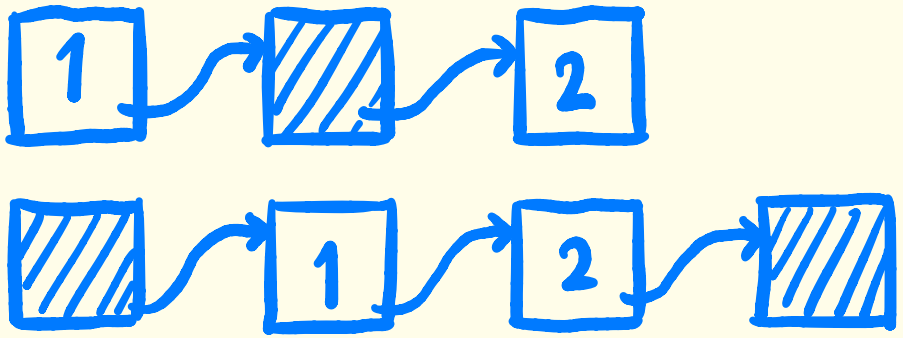
```
sum (self:List|self:List) : (int|int)
    requires {listpub(self) ≐ listpub(self)}
    ensures  {sum ≐ sum}
```

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

e.g.



pub. vals: 1,2

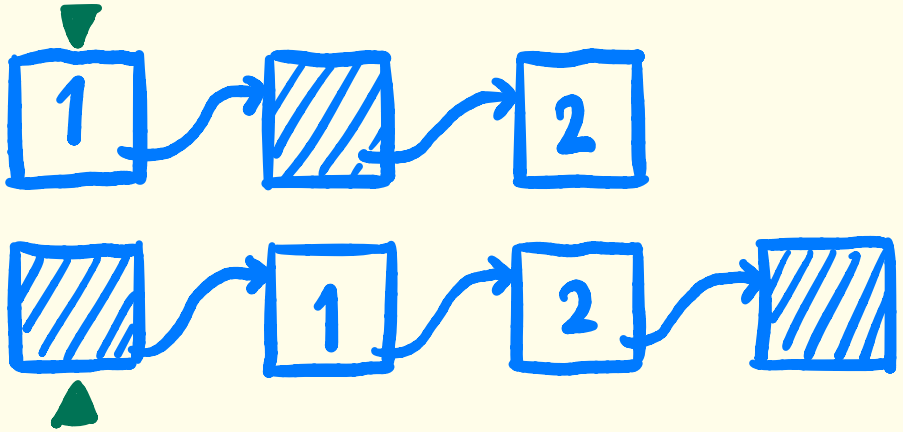
pub. vals: 1,2

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

Alignment

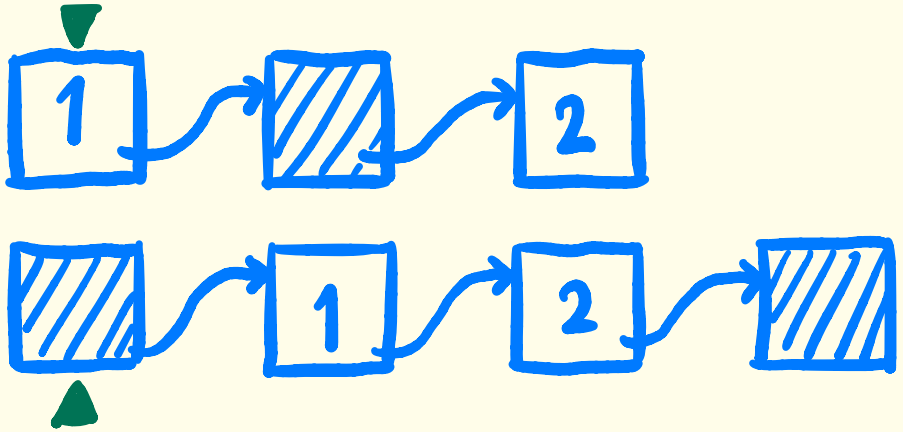


Data-dependent alignment

```
meth sum(self:List):int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List):int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

Alignment



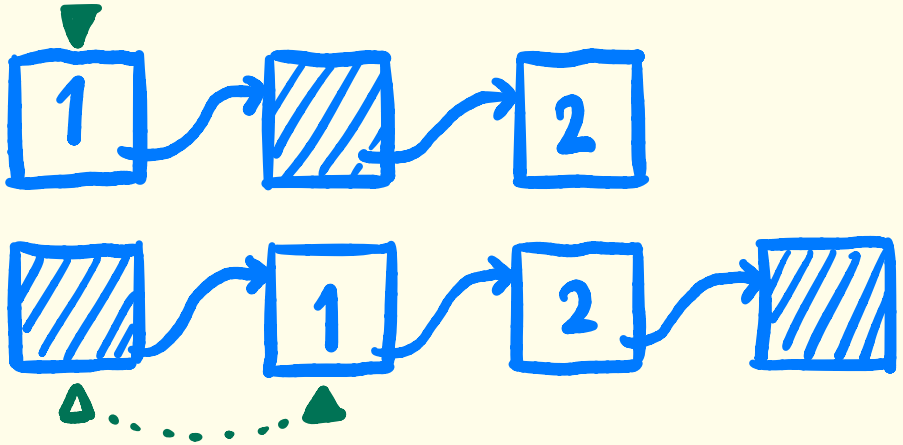
Node non-public
Right only iter.

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

Alignment



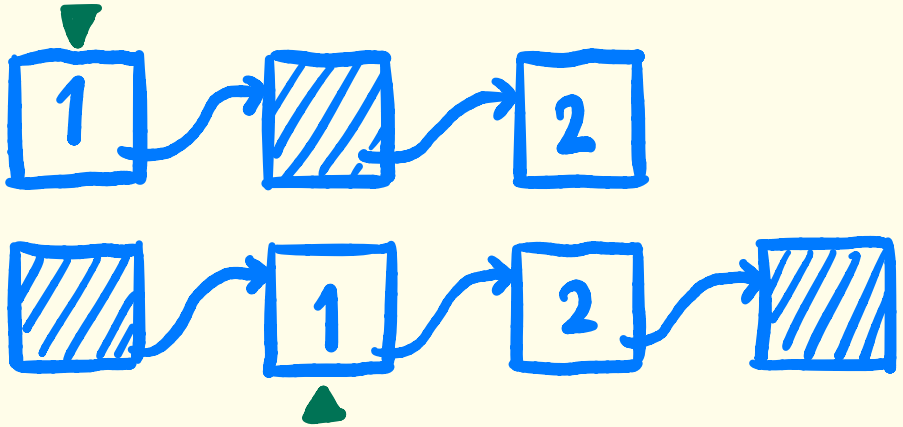
Node non-public
Right only iter.

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List):int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List):int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

Alignment



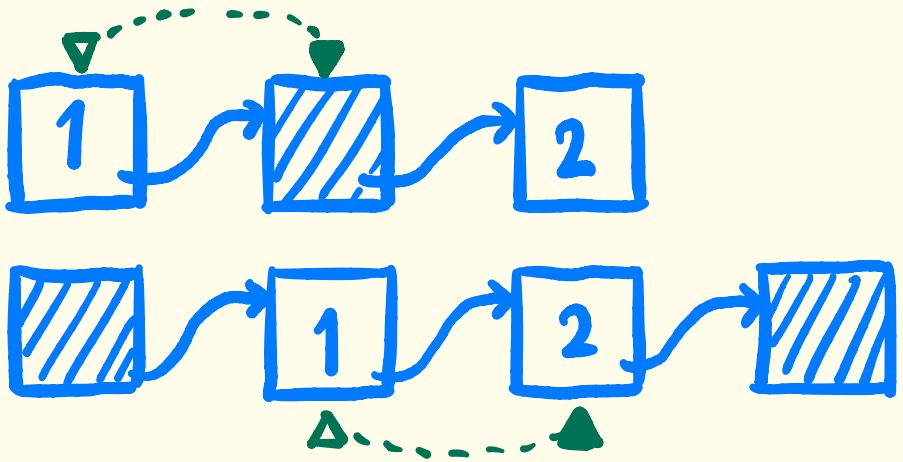
Both nodes public
Lockstep iter

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

Alignment



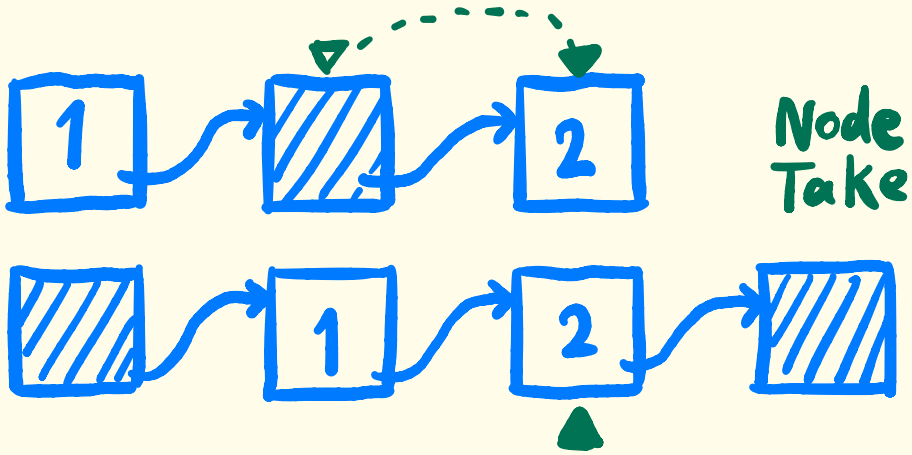
Preserve
sum ≐ sum

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

Alignment



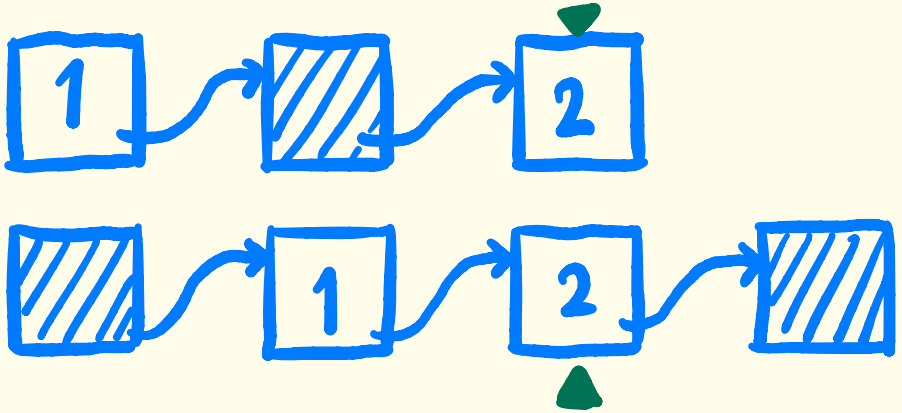
Node Take non-public left only step sum on left unchanged

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:= sum+p.value;
    p:=p.nxt;
```

Alignment



Both nodes public
Lockstep

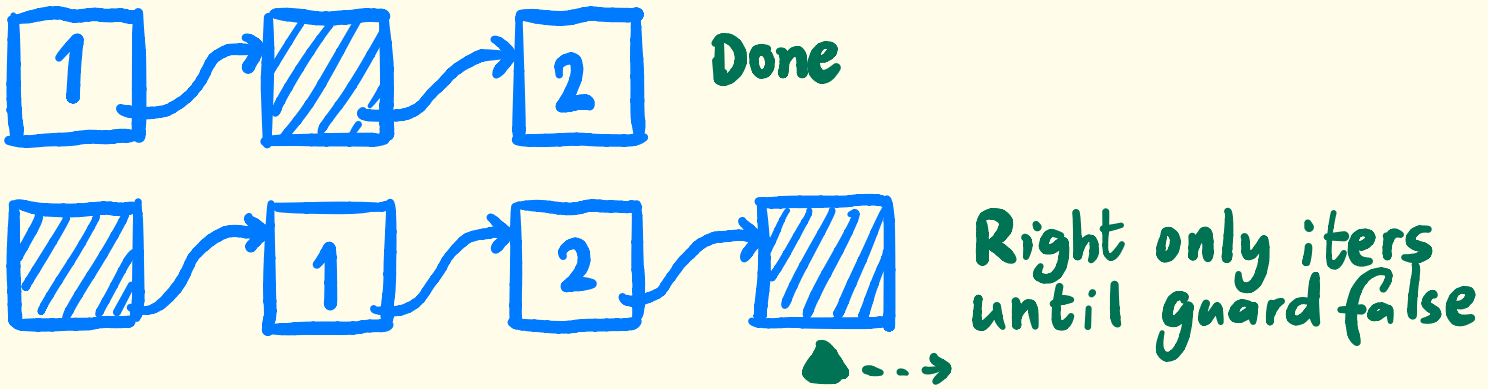
sum ≐ sum

DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

```
meth sum(self:List) :int =
  p:=self.head; sum:=0;
  while (p≠null) do
    if p.pub then
      sum:=sum+p.value;
    p:=p.nxt;
```

Alignment



DATA-DEPENDENT ALIGNMENT

```
meth sum(self:List | self:List)
  (p:=self.head | p:=self.head); (sum:=0 | sum:=0);
  while (p≠null | p≠null) do
    ⟨¬p.pub⟩ | ⟨¬p.pub⟩
    (if p.pub then
       sum:=sum+p.value
     |
     if p.pub then
       sum:=sum+p.value);
    (p:=p.nxt | p:=p.nxt);
```

Biprogram capturing alignment

left align guard | right align guard

When ⟨¬p.pub⟩ (resp. ⟨¬p.pub⟩), left (resp. right) only iter.

Lockstep iterations when both alignment guards false

MAJORIZATION IN WHYREL

```
interface I =
  meth f (n: int) : int
    effects  { rd n; rw result }
end
```

```
module Factorial : I =
  meth f (n: int) : int =
    var i: int in
    var z: int in
    i := n;
    z := 24;
    while (i > 4) do
      z := z * i;
      i := i - 1;
   done;
   result := z;
end
```

module Power2 is similar but replaces

$z := 24 \rightarrow z := 16$

$z := z * i \rightarrow z := z * 2$

```
bimodule B (Factorial | Power2) =
  meth f (n: int | n: int) : (int | int)
    requires { Both (n >= 4) }
    requires { n =:= n }
    ensures  { [< result <] > [> result >] }
  = Var i: int | i: int in
    Var z: int | z: int in

    ( i := n  | i := n  );
    ( z := 24 | z := 16 );

    While (i > 4) | (i > 4) . do
      invariant { i =:= i /\ [< z <] > [> z >]}
      invariant { [> z >] > [0] }
      invariant { Both (i >= 4) }

      ( z := z * i | z := z * 2 );
      ( i := i - 1 | i := i - 1 );
    done;

    ( result := z | result := z );
end
```

MAJORIZATION IN WHYREL

```
interface I =
  meth f (n: int) : int
    effects  { rd n; rw result }
end
```

```
module Factorial : I =
  meth f (n: int) : int =
    var i: int in
    var z: int in
    i := n;
    z := 24;
    while (i > 4) do
      z := z * i;
      i := i - 1;
   done;
   result := z;
end
```

module Power2 is similar but replaces

$z := 24 \rightarrow z := 16$

$z := z * i \rightarrow z := z * 2$

```
bimodule B (Factorial | Power2) =
  meth f (n: int | n: int) : (int | int)
    requires { Both (n >= 4) }
    requires { n =:= n }
    ensures  { [< result <] > [> result >] }
  = Var i: int | i: int in
    Var z: int | z: int in

    ( i := n  | i := n  );
    ( z := 24 | z := 16 );

    While (i > 4) | (i > 4) . do
      invariant { i =:= i /\ [< z <] > [> z >]}
      invariant { [> z >] > [0] }
      invariant { Both (i >= 4) }

      ( z := z * i | z := z * 2 );
      ( i := i - 1 | i := i - 1 );
    done;

    ( result := z | result := z );
end
```

value of result in left state

z on right > 0

unfortunate syntax [c] to use constants when comparing values in two states

MAJORIZATION IN WHY3

```
module Factorial
  use prelude.Prelude
  use I

  let f (n: int) : int diverges
  = let result = ref 0 in
    let n = ref n in
    let i = ref (!n) in
    let z = ref 24 in
    while !i > 4 do
      invariant { True }
      z := !z * !i;
      i := !i - 1;
    done;
    result := !z;
    !result
end
```

Biprogram for factorial and exponent

```
let f (l_n: int) (r_n: int) : (l_result: int, r_result: int)
  requires { l_n = r_n }
  requires { l_n >= 4 /\ r_n >= 4 }
  ensures  { l_result > r_result }
  diverges
= let l_result = ref 0 in let r_result = ref 0 in
  let l_n = ref l_n in let r_n = ref r_n in
  let l_i = ref (!l_n) in let r_i = ref (!r_n) in
  let l_z = ref 24 in let r_z = ref 16 in

  while !l_i > 4 do
    invariant { [@expl:lockstep adequacy]
      (!l_i > 4) <-> (!r_i > 4) }
    invariant { !l_i = !r_i }
    invariant { !l_z > !r_z }
    invariant { !r_z > 0 }
    invariant { !l_i >= 4 /\ !r_i >= 4 }

    l_z := !l_z * !l_i ; r_z := !r_z * 2 ;
    l_i := !l_i - 1 ; r_i := !r_i - 1 ;
  done;
  l_result := !l_z;
  r_result := !r_z;
  (!l_result, !r_result)
```

MAJORIZATION IN WHY3

```
module Factorial
  use prelude.Prelude
  use I

  let f (n: int) : int diverges
  = let result = ref 0 in
    let n = ref n in
    let i = ref (!n) in
    let z = ref 24 in
    while !i > 4 do
      invariant { True }
      z := !z * !i;
      i := !i - 1;
    done;
    result := !z;
    !result
end
```

state arg. not shown

to avoid VC's related to proving termination

locals modeled as WhyML refs

"result" always returned

Biprogram for factorial and exponent

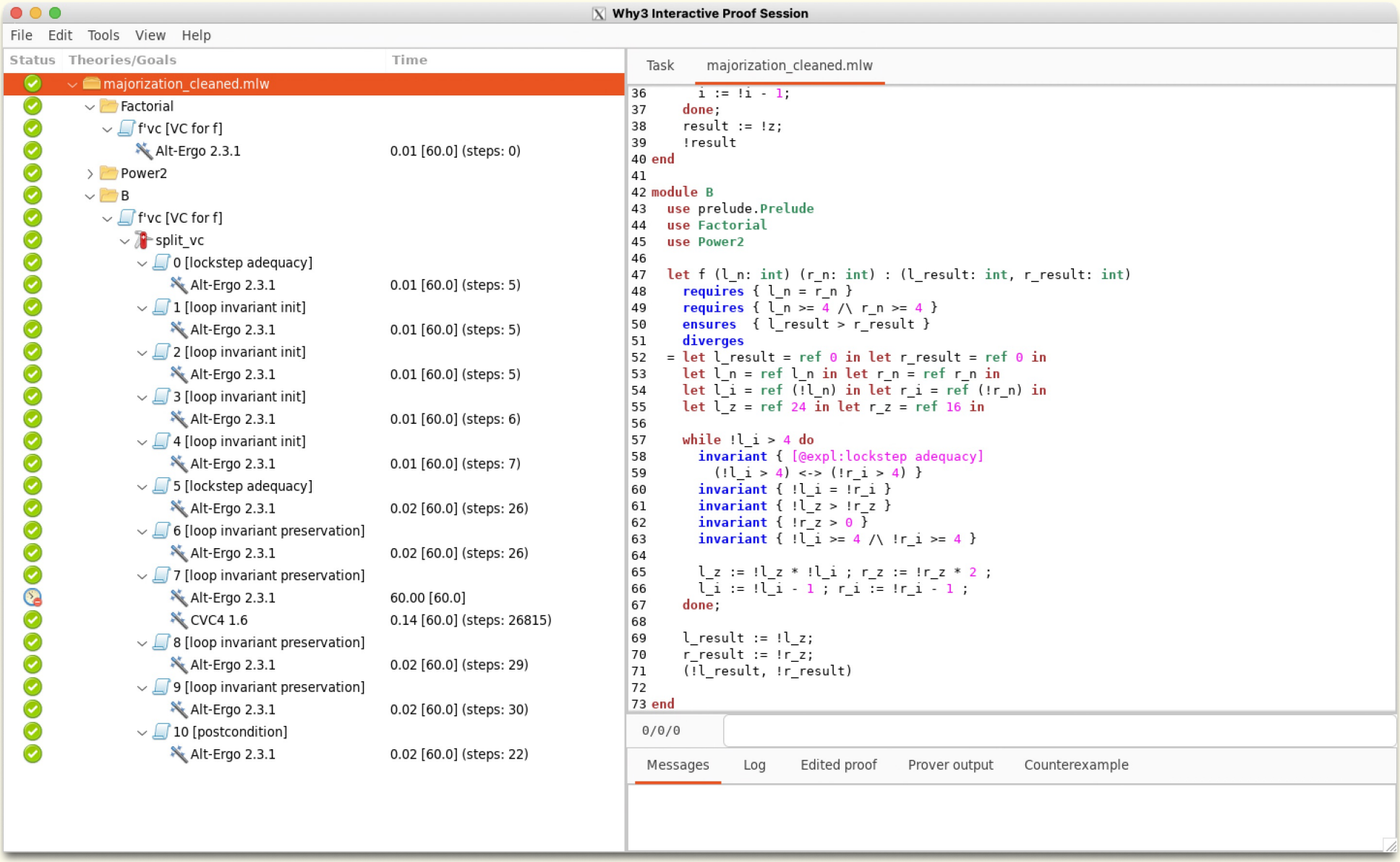
```
let f (l_n: int) (r_n: int) : (l_result: int, r_result: int)
  requires { l_n = r_n }
  requires { l_n >= 4 /\ r_n >= 4 }
  ensures  { l_result > r_result }
  diverges
= let l_result = ref 0 in let r_result = ref 0 in
  let l_n = ref l_n in let r_n = ref r_n in
  let l_i = ref (!l_n) in let r_i = ref (!r_n) in
  let l_z = ref 24 in let r_z = ref 16 in

  while !l_i > 4 do
    invariant { [@expl:lockstep adequacy]
      (!l_i > 4) <-> (!r_i > 4) }
    invariant { !l_i = !r_i }
    invariant { !l_z > !r_z }
    invariant { !r_z > 0 }
    invariant { !l_i >= 4 /\ !r_i >= 4 }

    l_z := !l_z * !l_i ; r_z := !r_z * 2 ;
    l_i := !l_i - 1 ; r_i := !r_i - 1 ;
  done;
  l_result := !l_z;
  r_result := !r_z;
  (!l_result, !r_result)
```

adequacy of lockstep aligned loop

MAJORIZATION IN WHY3



ENCODING DETAILS

```
interface NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc;
              rd i, n }

end

module NodeImpl : NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;

end
```

```
module State
  use prelude.Prelude

  type reference

  type reftype =
    | Node

  type heap = {
    mutable value : Map.t reference int;
    mutable next : Map.t reference reference
  }

  type state = {
    mutable heap: heap;
    mutable ghost alloct: Map.t reference reftype
  } invariant { not (Map.mem null alloct) }
    invariant {
      forall p: reference.
        alloct[p] = Node ->
        (Map.mem p heap.value /\ Map.mem p heap.next) /\
        (heap.next[p] = null
         \/ Map.mem heap.next[p] alloct
            /\ alloct[heap.next[p]] = Node) }
```

← abstract WhyML type
constants : null
operations : (=)

↑ class names

→ keeps track of alloc'd refs

↑ WhyML type invariant

generated WhyML file

ENCODING DETAILS

```
interface NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc;
              rd i, n }

end

module NodeImpl : NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;

end
```

```
  (* Axiomatizations of image expressions: r`f *)

  (* r`value = ø *)
  function img_value : state -> rgn -> rgn
  axiom img_value_ax : forall s: state, r: rgn. img_value s r
= emptyRgn

  (* r`next = o.next for all objects o of type Node in r *)
  function img_next : state -> rgn -> rgn
  axiom img_next_ax : forall s: state, r: rgn, p: reference.
    Rgn.mem p (img_next s r) <->
      exists q: reference.
        Map.mem q s.alloct /\
     s.alloct[q] = Node /\ Rgn.mem q r /\ p = s.heap.next[q]

  (* ... other definitions omitted ... *)
```

defs relevant to refperms
(not covered in talk)
defs related to framing, etc.

ENCODING DETAILS

```
interface NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc;
              rd i, n }

end

module NodeImpl : NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;

end
```

```
val cons (s: state) (i: int) (n: reference) : reference
  ensures  { result <> null /\ s.alloct[result] = Node }
  ensures  { s.heap.value[result] = i }
  ensures  { s.heap.next[result] = n }

  ensures  { wrs_to_values_framed_by
               (old s) s (Rgn.singleton result) }
  ensures  { wrs_to_next_framed_by
               (old s) s (Rgn.singleton result) }

  writes { s.heap.value, s.heap.next, s.alloct }
```

pub. spec. for cons

handling writes involving region expressions

can allocate

WhyML writes clause required

ENCODING DETAILS

```
interface NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc;
              rd i, n }

end

module NodeImpl : NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;

end
```

```
val cons (s: state) (i: int) (n: reference) : reference
  ensures  { result <> null /\ s.alloct[result] = Node }
  ensures  { s.heap.value[result] = i }
  ensures  { s.heap.next[result] = n }

  ensures  { wrs_to_values_framed_by
               (old s) s (Rgn.singleton result) }
  ensures  { wrs_to_next_framed_by
               (old s) s (Rgn.singleton result) }

  writes { s.heap.value, s.heap.next, s.alloct }
```

result ≠ null generated as post b/c of WhyRel's convention*:

"Node" — non-null Node ref.

"Node?" — possibly null Node ref.

More appropriate declaration for cons:

meth cons (i:int, n:Node?) : Node

*the decl. above also generates requires {n ≠ null}; omitted from WhyML spec.

ENCODING DETAILS

```
interface NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc;
              rd i, n }

end

module NodeImpl : NODE =
  class Node
    {value: int; next: Node;}

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;

end
```

```
  let cons (s: state) (i: int) (n: reference) : reference
    ensures  { result <> null /\ s.alloct[result] = Node }
    ensures  { s.heap.value[result] = i }
    ensures  { s.heap.next[result] = n }
    ensures  { wrs_to_values_framed_by
                 (old s) s (Rgn.singleton result) }
    ensures  { wrs_to_next_framed_by
                 (old s) s (Rgn.singleton result) }
    writes { s.heap.value, s.heap.next, s.alloct }
  = let result = ref null in
    let i = ref i in
    let n = ref n in
    let tmp = ref null in
    tmp := new_Node s;
    (* tmp.value := i *)
    s.heap.value <- M.add !tmp !i s.heap.value;
    (* tmp.next  := n *)
    s.heap.value <- M.add !tmp !n s.heap.next;
    result := !tmp;
    !result
```

Each write to s.heap or s generates type invariant VCs !!

Can potentially be avoided using Why3's abstraction facilities or Boogie style free requires/ensures. The state type invariant expresses conditions on typing and ideally should not incur extra VCs since WhyRel type checks programs.

ENCODING DETAILS

```
interface NODE =

  public pool : rgn

  class Node
    { value: int; next: Node;
      rep: rgn; }

  boundary { pool, pool`any, pool`rep`any }

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc; rd i, n }
end

module NodeImpl : NODE =

  private invariant nodePriv =
    forall n: Node in pool.
      let rep = n.rep in
      n in rep /\ rep`next subset rep

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;
end
```

Add to example by including:

- Global variable pool:rgn
- Field rep:rgn in Node
- Boundary
- Private invariant

ENCODING DETAILS

```
interface NODE =

  public pool : rgn

  class Node
    { value: int; next: Node;
      rep: rgn; }

  boundary { pool, pool`any, pool`rep`any }

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc; rd i, n }
end

module NodeImpl : NODE =

  private invariant nodePriv =
    forall n: Node in pool.
      let rep = n.rep in
      n in rep /\ rep`next subset rep

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;
end
```

Changes to state encoding

```
type heap = {
    mutable value : Map.t reference int;
    mutable next : Map.t reference reference;
    mutable ghost rep : Map.t reference rgn
  }

type state = {
    mutable heap: heap;
    mutable ghost alloct: Map.t reference reftype
    mutable ghost pool: rgn;
  } invariant { ... }
    invariant { forall q: reference.
      Rgn.mem q pool ->
      q = null \/ Map.mem q alloct }
```

Global variables are components of the state type

New type invariant for globals of type rgn — required since references are an abstract WhyML type

Note: rgn always ghost

ENCODING DETAILS

```
interface NODE =

  public pool : rgn

  class Node
    { value: int; next: Node;
      rep: rgn; }

  boundary { pool, pool`any, pool`rep`any }

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc; rd i, n }
end

module NodeImpl : NODE =

  private invariant nodePriv =
    forall n: Node in pool.
      let rep = n.rep in
      n in rep /\ rep`next subset rep

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;
end
```

```
  let cons (s: state) (i: int) (n: reference) :
reference
    requires { ... } ensures { ... }
    (* boundaries grow monotonically *)
    ensures  {
      let bsnap =
        old (Rgn.union s.pool
              (img_rep s s.pool)) in
      Rgn.subset bsnap
       (Rgn.union s.pool (img_rep s s.pool)) }
```

Specs now add special post to cater for a technical side condition of region logic: boundaries don't shrink

ENCODING DETAILS

```
interface NODE =

  public pool : rgn

  class Node
    { value: int; next: Node;
      rep: rgn; }

  boundary { pool, pool`any, pool`rep`any }

  meth cons (i: int, n: Node) : Node
    ensures { result.value = i }
    ensures { result.next = n }
    effects { rw {result}`any, alloc; rd i, n }
end

module NodeImpl : NODE =

  private invariant nodePriv =
    forall n: Node in pool.
      let rep = n.rep in
      n in rep /\ rep`next subset rep

  meth cons (i: int, n: Node) : Node
  = var tmp: Node in
    tmp := new Node;
    tmp.value := i;
    tmp.next := n;
    result := tmp;
end
```

```
lemma boundary_frames_invariant_Node =
  forall s: state, t: state, pi: Refperm.t.
    okRefperm s t pi ->
    Refperm.identity pi s.alloct t.alloct ->
    Refperm.idRgn pi s.pool t.pool ->
    agree_allfields s t pi
      (Rgn.union s.pool (img_rep s s.pool)) ->
    nodePriv s ->
    nodePriv t
```

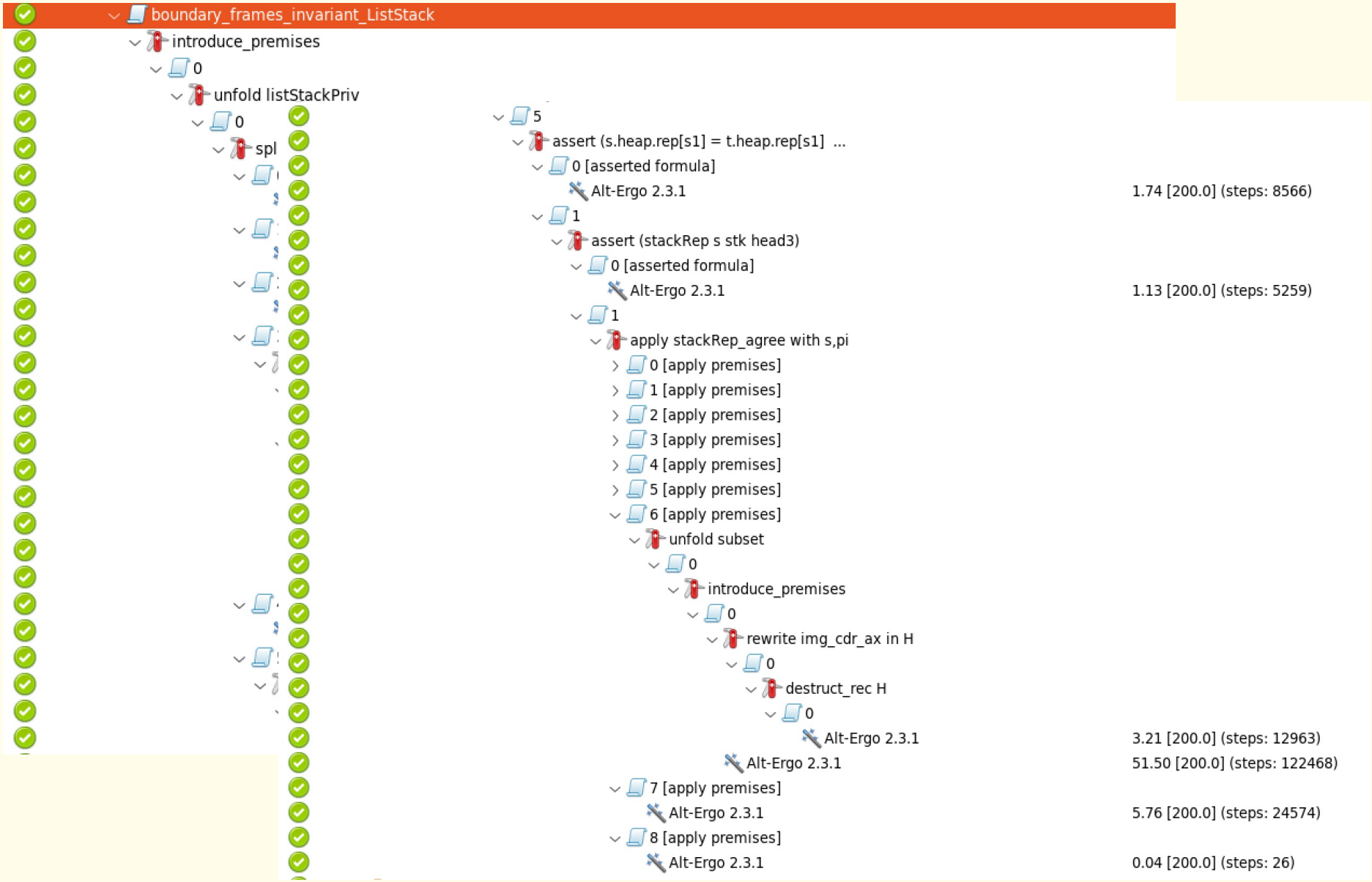
Lemma generated for user to prove: private invariant only depends on boundary

Similar lemma when it comes to coupling relations

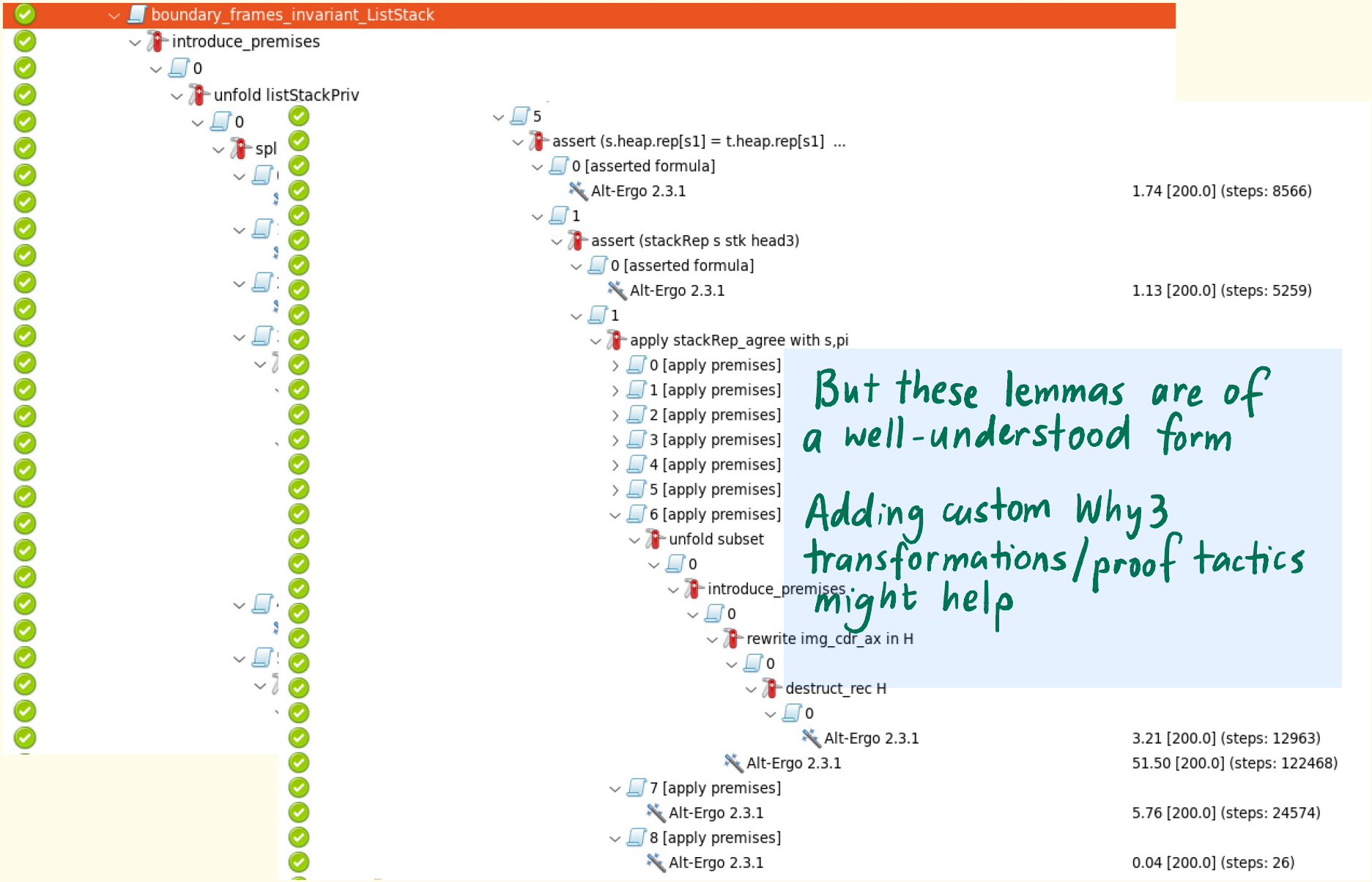
Proving this lemma can be challenging. e.g. ListStack ...

ENCODING DETAILS

ENCODING DETAILS



ENCODING DETAILS



But these lemmas are of a well-understood form

Adding custom Why3 transformations/proof tactics might help