Applicative invariants for Lustre

Amos Robinson, international man of leisure

Chestnut-backed chickadee

Synchronous languages at an autonomous vehicle startup

Ghost gum

Happy days

- Implemented controllers, drivers, state machines in Lustre
- Proved some reassuring properties
- Ran them in the car
- Worked first time (more or less)

Rainbow lorikeet

Success with Kind2

Kind2 is generally great:

- Solid for small programs
- Actively developed
- Friendly, helpful developers

Rainbow lorikeet

Sadness with Kind2

But as our programs got bigger…

- Hard to predict whether true properties can be proved
- Modifications difficult, could require serious restructuring to keep proofs
- Spurious failures in CI make us look bad

White-winged chough

What we couldn't prove reliably

Willie wagtail vs grey butcherbird

```
node lastn(const n: int; pred: bool)
returns (out: bool)
let
  count = if pred then (0 -> pre count) + 1 else 0;
  out   = count >= n;
tel
```

```
node lastn(const n: int; pred: bool)
returns (out: bool)
let
  count = if pred then (0 -> pre count) + 1 else 0;
  out   = count >= n;
tel

function delta_valid(input1: int, input2: int)
returns (ok: bool)
let
  ok = abs(input1 - input2) <= DELTA_MAX
tel
```

```
node signal_valid(input: int)
returns (ok: bool)
(*@contract
  guarantee
    not lastn(10, delta_valid(input, 0 -> pre input)) =>
    not ok;
  ...
*)
```

```
node signal_valid(input: int)
returns (ok: bool)
(*@contract
  guarantee
    not lastn(10, delta_valid(input, 0 -> pre input)) =>
    not ok;
  ...
*)

type SAMPLE       = { adc: int; ... }
const SAMPLE_ZERO = { adc  = 0; ... }

node main(sample: SAMPLE)
returns (engaged: bool; ...)
(*@contract
  guarantee
    not lastn(10, delta_valid(sample.adc, (SAMPLE_ZERO -> pre sample).adc)) =>
    not engaged;
*)
let
  engaged = signal_valid(sample.adc);
  ...
tel
```

Pen-and-paper proof

Jacky winter

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };

---------------------------------------------------------------------

show main.guarantee:
  not lastn(10, delta_valid(sample.adc, (SAMPLE_ZERO -> pre sample).adc)) =>
  not engaged;
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };

--------------------------------------------------------------------

show main.guarantee:
  not lastn(10, delta_valid(sample.adc, (SAMPLE_ZERO -> pre sample).adc)) =>
  not engaged;


rewrite prim_distributes_arrow: forall stream e e', pure function f.
  f (e -> e') = (f e) -> (f e')
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };

---------------------------------------------------------------------

show main.guarantee:
  not lastn(10, delta_valid(sample.adc, SAMPLE_ZERO.adc -> (pre sample).adc)) =>
  not engaged;


rewrite prim_distributes_arrow: forall stream e e', pure function f.
  f (e -> e') = (f e) -> (f e')
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };

---------------------------------------------------------------------

show main.guarantee:
  not lastn(10, delta_valid(sample.adc, SAMPLE_ZERO.adc -> (pre sample).adc)) =>
  not engaged;


rewrite prim_distributes_pre: forall stream e, pure function f.
  f (pre e) = pre (f e)
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };

---------------------------------------------------------------------

show main.guarantee:
  not lastn(10, delta_valid(sample.adc, SAMPLE_ZERO.adc -> pre sample.adc)) =>
  not engaged;


rewrite prim_distributes_pre: forall stream e, pure function f.
  f (pre e) = pre (f e)
```

```
assume
  signal_valid.guarantee[input := sample.adc, ok := engaged]:
    not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
    not engaged;

  engaged = signal_valid(sample.adc);

  SAMPLE_ZERO = { adc = 0; ... };

----------------------------------------------------------------------

show main.guarantee:
  not lastn(10, delta_valid(sample.adc, 0 -> pre sample.adc)) =>
  not engaged;


unfold SAMPLE_ZERO, constant propagation, etc
```

Transition system proof

Australasian grebes

```
node delta_valid_sample(sample: SAMPLE)
returns (delta_main: bool, delta_signal: bool)
let
  delta_main   = delta_valid(sample.adc, (SAMPLE_ZERO -> pre sample).adc);
  delta_signal = delta_valid(sample.adc, 0 -> pre sample.adc);
tel
```

```
node delta_valid_sample(sample: SAMPLE)
returns (delta_main: bool, delta_signal: bool)
let
  delta_main   = delta_valid(sample.adc, (SAMPLE_ZERO -> pre sample).adc);
  delta_signal = delta_valid(sample.adc, 0 -> pre sample.adc);
tel

==>

lts delta_valid_sample:
  type state = { init:         bool;
                 delay_main:   SAMPLE;
                 delay_signal: int; };

  init(s: state): bool =
    s.init == true;

  step(s: state, sample: SAMPLE): (state * bool * bool) =
    let
      delay_main                        = if s.init then SAMPLE_ZERO else s.delay_main
      delta_main                        = delta_valid(sample.adc, delay_main.adc)

      delay_signal                      = if s.init then 0 else s.delay_signal
      delta_signal                      = delta_valid(sample.adc, delay_signal)

    in
      ({ init           = false;
         delay_main     = sample;
         delay_signal   = sample.adc },
         delta_main, delta_signal)
```

```
node delta_valid_sample(sample: SAMPLE)
returns (delta_main: bool, delta_signal: bool)
let
  delta_main   = delta_valid(sample.adc, (SAMPLE_ZERO -> pre sample).adc);
  delta_signal = delta_valid(sample.adc, 0 -> pre sample.adc);
tel
```

==>

```
lts delta_valid_sample:
  type state = { init:         bool;
                 delay_main:   SAMPLE;
                 delay_signal: int; };

  init(s: state): bool =
    s.init == true;

  step(s: state, sample: SAMPLE): (state * bool * bool) =
    let
      delay_main                         = if s.init then SAMPLE_ZERO else s.delay_main
      delta_main                         = delta_valid(sample.adc, delay_main.adc)

      delay_signal                       = if s.init then 0 else s.delay_signal
      delta_signal                       = delta_valid(sample.adc, delay_signal)

    in
      ({ init            = false;
         delay_main      = sample;
         delay_signal    = sample.adc },
         delta_main, delta_signal)
```

```
node delta_valid_last2(delta_ok: bool)
returns (last_main: bool, last_signal: bool)
let
  last_main   = lastn(10, delta_ok);
  last_signal = lastn(10, delta_ok);

tel
```

```
node delta_valid_last2(delta_ok: bool)
returns (last_main: bool, last_signal: bool)
let
  last_main   = lastn(10, delta_ok);
  last_signal = lastn(10, delta_ok);

tel

==>

lts delta_valid_last2:
  type state = { last_main:   lastn.state;
                 last_signal: lastn.state; };

  init(s: state): bool =
    lastn.init(s.last_main) and lastn.init(s.last_signal);

  step(s: state, delta_ok: bool): (state * bool * bool) =
    let
      (state_last_main',   last_main)   = lastn.step(s.last_main,   10, delta_ok)
      (state_last_signal', last_signal) = lastn.step(s.last_signal, 10, delta_ok)
    in
      ({ init        = false;
         last_main   = state_last_main';
         last_signal = state_last_signal' }, last_main, last_signal)
```

```
node delta_valid_last2(delta_ok: bool)
returns (last_main: bool, last_signal: bool)
let
  last_main   = lastn(10, delta_ok);
  last_signal = lastn(10, delta_ok);

tel

==>

lts delta_valid_last2:
  type state = { last_main:   lastn.state;
                 last_signal: lastn.state; };

  init(s: state): bool =
    lastn.init(s.last_main) and lastn.init(s.last_signal);

  step(s: state, delta_ok: bool): (state * bool * bool) =
    let
      (state_last_main',   last_main)   = lastn.step(s.last_main,   10, delta_ok)
      (state_last_signal', last_signal) = lastn.step(s.last_signal, 10, delta_ok)
    in
      ({ init        = false;
         last_main   = state_last_main';
         last_signal = state_last_signal' }, last_main, last_signal)
      --%PROPERTY state_last_main' == state_last_signal'
```

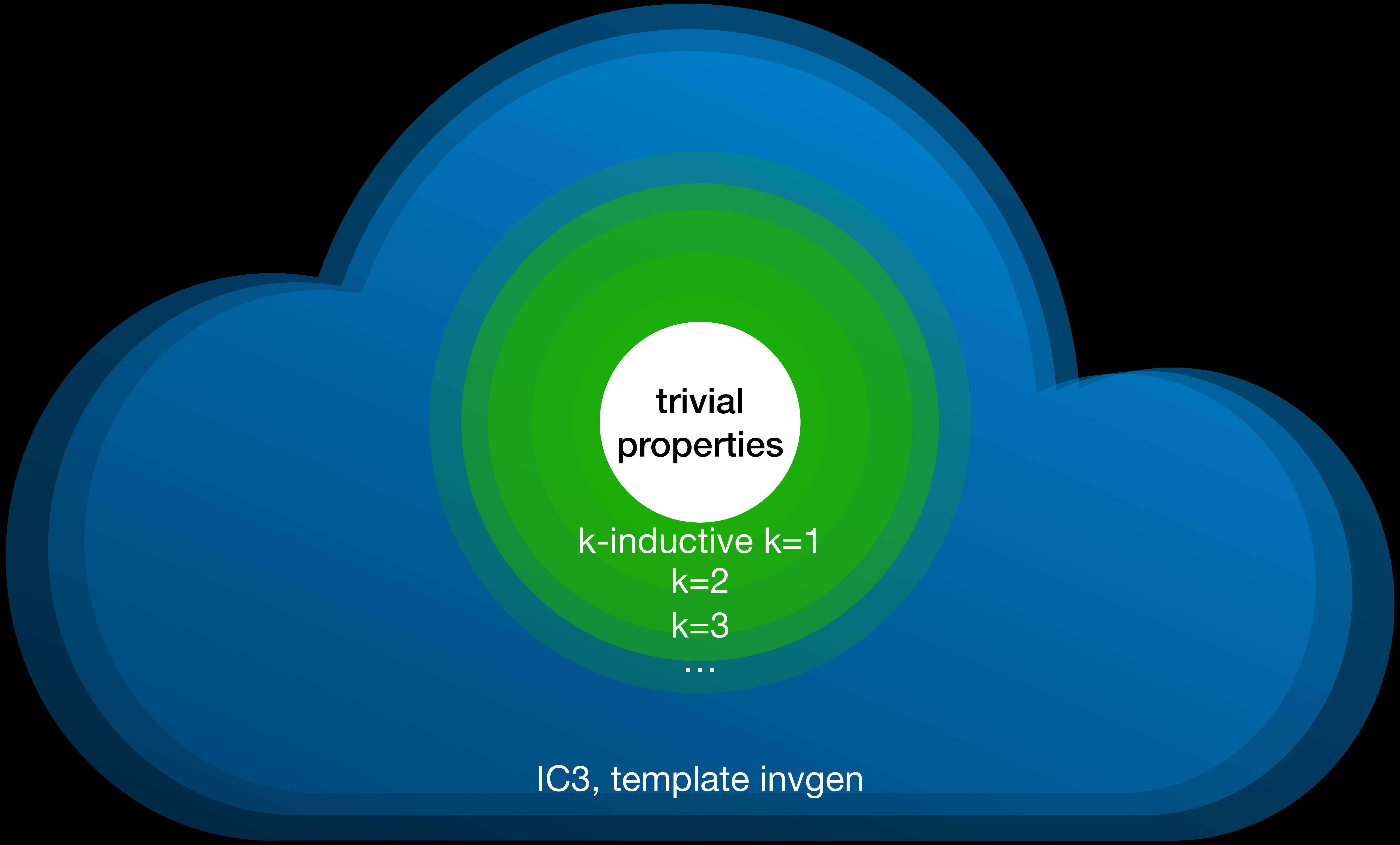
```
node delta_valid_last2(delta_ok: bool)
returns (last_main: bool, last_signal: bool)
let
  last_main   = lastn(10, delta_ok);
  last_signal = lastn(10, delta_ok);
  --%PROPERTY ??? = ???;
tel

==>

lts delta_valid_last2:
  type state = { last_main:   lastn.state;
                 last_signal: lastn.state; };

  init(s: state): bool =
    lastn.init(s.last_main) and lastn.init(s.last_signal);

  step(s: state, delta_ok: bool): (state * bool * bool) =
    let
      (state_last_main',   last_main)   = lastn.step(s.last_main,   10, delta_ok)
      (state_last_signal', last_signal) = lastn.step(s.last_signal, 10, delta_ok)
    in
      ({ init        = false;
         last_main   = state_last_main';
         last_signal = state_last_signal' }, last_main, last_signal)
      --%PROPERTY state_last_main' == state_last_signal'
```

Two problems

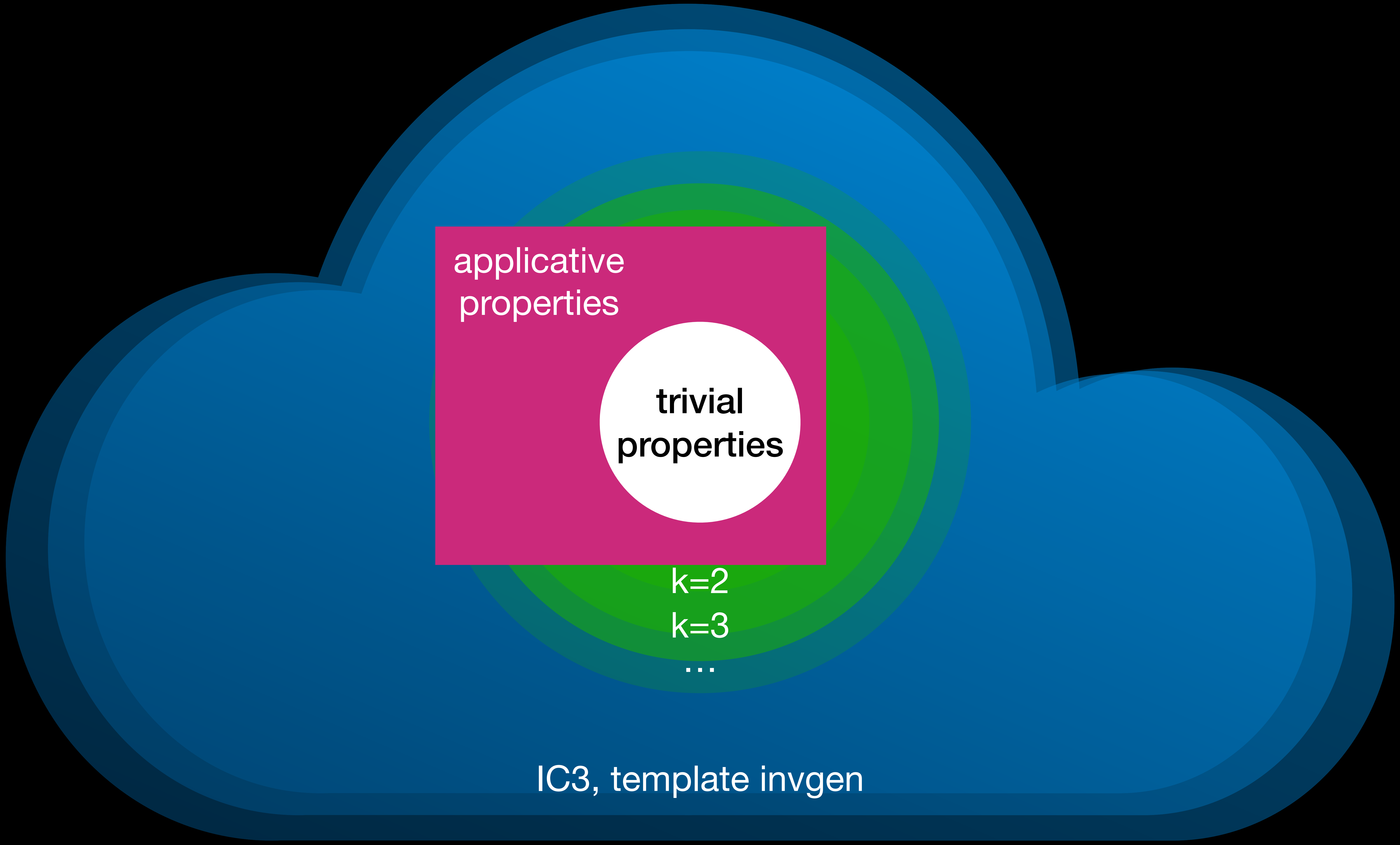
- "obvious" invariants aren't obvious on the translated system
- we can't state the invariant, even though we know it!

Emus

What we can prove



What we can prove - proposed class



Invariant extraction with e-graphs

Idea:

- convert node to equivalence graph, instantiating any subnodes
- apply the applicative rewrites
- extract the equalities we learn as invariants

Chestnut-backed chickadee

```
node signal_valid(input: int)
(*@contract guarantee not last.out => not ok; *)
  ok            : bool;
  pre_input      = pre input;
  del_input      = 0 -> pre_input;
  subnode last   = lastn(10, delta_valid(input, del_input))

node main(sample: SAMPLE)
(*@contract guarantee not last.out => not engaged; *)
  sample_adc     = sample.adc;
  pre_sample     = pre sample;
  del_sample     = SAMPLE_ZERO -> pre_sample;
  del_sample_adc = del_sample.adc;
  subnode valid  = signal_valid(sample_adc);
  subnode last   = lastn(10, delta_valid(sample_adc, del_sample_adc))
  engaged        = valid.ok;
```

```
-- subnode valid = signal_valid(sample.adc)
-- guarantee not valid.last.out => not valid.ok;
  valid.pre_input = pre sample_adc;
  valid.del_input = 0 -> valid.pre_input;
  valid.last.out  = lastn(10, delta_valid(sample_adc, valid.del_input));

-- node main(sample)
-- guarantee not last.out => not engaged;
  sample_adc      = sample.adc;
  pre_sample      = pre sample;


  del_sample      = SAMPLE_ZERO -> pre_sample;
  del_sample_adc  = del_sample.adc;


  last.out        = lastn(10, delta_valid(sample_adc, del_sample_adc));
  engaged         = valid.ok;
```

```
-- subnode valid = signal_valid(sample.adc)
-- guarantee not valid.last.out => not valid.ok;
  valid.pre_input = pre sample_adc;
  valid.del_input = 0 -> valid.pre_input;
  valid.last.out  = lastn(10, delta_valid(sample_adc, valid.del_input));

-- node main(sample)
-- guarantee not last.out => not engaged;
  sample_adc      = sample.adc;
  pre_sample      = pre sample;


  del_sample      = SAMPLE_ZERO -> pre_sample;
  del_sample_adc  = del_sample.adc;


  last.out        = lastn(10, delta_valid(sample_adc, del_sample_adc));
  engaged         = valid.ok;

---------------------------------------------------------------------

rewrite prim_distributes_arrow: forall stream e e', pure function f.
  f (e -> e') = (f e) -> (f e')
```

```
-- subnode valid = signal_valid(sample.adc)
-- guarantee not valid.last.out => not valid.ok;
  valid.pre_input = pre sample_adc;
  valid.del_input = 0 -> valid.pre_input;
  valid.last.out  = lastn(10, delta_valid(sample_adc, valid.del_input));

-- node main(sample)
-- guarantee not last.out => not engaged;
  sample_adc      = sample.adc;
  pre_sample      = pre sample;


  del_sample      = SAMPLE_ZERO -> pre_sample;
  del_sample_adc  = del_sample.adc;
  del_sample_adc  = SAMPLE_ZERO.adc -> pre_sample.adc;

  last.out        = lastn(10, delta_valid(sample_adc, del_sample_adc));
  engaged         = valid.ok;

---------------------------------------------------------------------

rewrite prim_distributes_arrow: forall stream e e', pure function f.
  f (e -> e') = (f e) -> (f e')
with f := _.adc, e := SAMPLE_ZERO, e' := pre sample:
  (SAMPLE_ZERO -> pre sample).adc = SAMPLE_ZERO.adc -> (pre sample).adc
```

```
-- subnode valid = signal_valid(sample.adc)
-- guarantee not valid.last.out => not valid.ok;
  valid.pre_input = pre sample_adc;
  valid.del_input = 0 -> valid.pre_input;
  valid.last.out  = lastn(10, delta_valid(sample_adc, valid.del_input));

-- node main(sample)
-- guarantee not last.out => not engaged;
  sample_adc      = sample.adc;
  pre_sample      = pre sample;


  del_sample      = SAMPLE_ZERO -> pre_sample;
  del_sample_adc  = del_sample.adc;
  del_sample_adc  = SAMPLE_ZERO.adc -> pre_sample.adc;

  last.out        = lastn(10, delta_valid(sample_adc, del_sample_adc));
  engaged         = valid.ok;

---------------------------------------------------------------------

rewrite prim_distributes_pre: forall stream e, pure function f.
  f (pre e) = pre (f e)
```

```
-- subnode valid = signal_valid(sample.adc)
-- guarantee not valid.last.out => not valid.ok;
  valid.pre_input = pre sample_adc;
  valid.del_input = 0 -> valid.pre_input;
  valid.last.out  = lastn(10, delta_valid(sample_adc, valid.del_input));

-- node main(sample)
-- guarantee not last.out => not engaged;
  sample_adc      = sample.adc;
  pre_sample      = pre sample;
  pre_sample_adc  = pre sample_adc;
  pre_sample_adc  = (pre sample).adc;
  del_sample      = SAMPLE_ZERO -> pre_sample;
  del_sample_adc  = del_sample.adc;
  del_sample_adc  = SAMPLE_ZERO.adc -> pre_sample_adc;

  last.out        = lastn(10, delta_valid(sample_adc, del_sample_adc));
  engaged         = valid.ok;

---------------------------------------------------------------------

rewrite prim_distributes_pre: forall stream e, pure function f.
  f (pre e) = pre (f e)
with f := _.adc, e := sample
  (pre sample).adc = pre sample.adc
```

```
-- subnode valid = signal_valid(sample.adc)
-- guarantee not valid.last.out => not valid.ok;
  valid.pre_input = pre_sample_adc;
  valid.del_input = del_sample_adc;
  valid.last.out  = lastn(10, delta_valid(sample_adc, del_sample_adc));

-- node main(sample)
-- guarantee not last.out => not engaged;
  sample_adc      = sample.adc;
  pre_sample      = pre sample;
  pre_sample_adc  = pre sample_adc;
  pre_sample_adc  = (pre sample).adc;
  del_sample      = SAMPLE_ZERO -> pre_sample;
  del_sample_adc  = del_sample.adc;
  del_sample_adc  = SAMPLE_ZERO.adc -> pre_sample_adc;
  del_sample_adc  = 0 -> pre_sample_adc;
  last.out        = lastn(10, delta_valid(sample_adc, del_sample_adc));
  engaged         = valid.ok;
```

```
node lastn(const n: int; pred: bool)
  pre_count     = pre count;
  arr_count     = 0 -> pre_count;
  count         = if pred then arr_count + 1 else 0;
  out           = count >= n;
```

```
-- subnode last = lastn(10, del_sample_adc)
  last.pre_count       = pre last.count;
  last.arr_count       = 0 -> last.pre_count;
  last.count           = if del_sample_adc
                         then last.arr_count + 1
                         else 0;
  last.out             = last.count >= 10;

-- subnode valid.last  = lastn(10, del_sample_adc)
  valid.last.pre_count = pre valid.last.count;
  valid.last.arr_count = 0 -> valid.last.pre_count;
  valid.last.count     = if del_sample_adc
                         then valid.last.arr_count + 1
                         else 0;
  valid.last.out       = valid.last.count >= 10;
```

```
  last.pre_count        = pre last.count;
  last.arr_count        = 0 -> last.pre_count;
  last.count            = if del_sample_adc
                          then last.arr_count + 1
                          else 0;
```

```
==(rewrite recursive binding)=>
```

```
  last.count            = fix (λcount.
                            if del_sample_adc
                            then (0 -> pre count) + 1
                            else 0);
```

```
-- subnode last = lastn(10, delta_valid(...))
  last.pre_count       = valid.last.pre_count
                       = pre last.count;
  last.arr_count       = valid.last.arr_count
                       = 0 -> last.pre_count;
  last.count           = valid.last.count
                       = if del_sample_adc
                         then last.arr_count + 1
                         else 0;
                       = if del_sample_adc
                         then valid.last.arr_count + 1
                         else 0;
                       = fix (λcount.
                           if del_sample_adc
                           then (0 -> pre count) + 1
                           else 0);
  last.out             = valid.last.out
                       = last.count >= 10;
```

```
-- subnode last = lastn(10, delta_valid(...))
  last.pre_count        = valid.last.pre_count
                        = pre last.count;
  last.arr_count        = valid.last.arr_count
                        = 0 -> last.pre_count;
  last.count            = valid.last.count
                        = if del_sample_adc
                          then last.arr_count + 1
                          else 0;
                        = if del_sample_adc
                          then valid.last.arr_count + 1
                          else 0;
                        = fix (λcount.
                            if del_sample_adc
                            then (0 -> pre count) + 1
                            else 0);
  last.out              = valid.last.out
                        = last.count >= 10;

invariant:
  true -> last.pre_count = valid.last.pre_count
```

Relationship to k-induction

Flannel flower

Equivalences sometimes stronger

```
node SoFar(pred: bool)
returns (out: bool)
let
  out = pred and (true -> pre out);
tel

---------------------------------------------------------------------
assume
  SoFar(X) => P

show
  SoFar(X) => P
```

- k-induction alone cannot find the invariant that SoFar(X)=SoFar(X)

K-induction sometimes stronger

```
node countmod4()
returns (mod4: int)
let
  mod4 = 0 -> if pre mod4 = 3 then 0 else pre mod4 + 1;
  --%PROPERTY mod4 <= 6;
tel
```

- bound <= 6 not tight enough to be 1-inductive, but 3-induction can get it

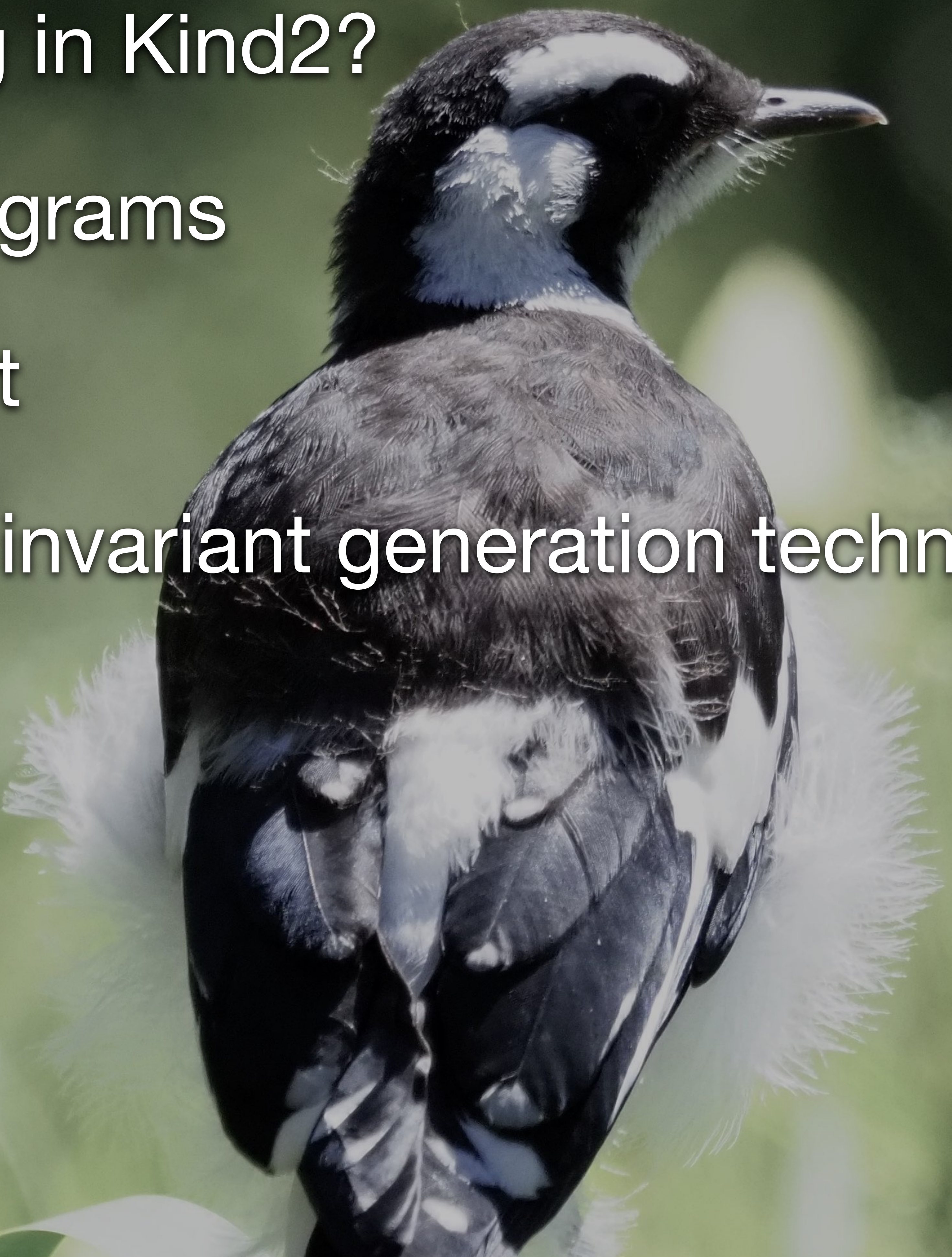
Modularity

- equivalences can extract invariants from subnodes separately
- could skip equality-saturation on nodes that are too big and still benefit from subnode analyses

- k-induction is monolithic: k parameter controls unfolding for whole system

Future work

- current implementation in experimental Scala EDSL for Lustre
  - worth implementing in Kind2?
- evaluate on larger programs
- improve clock support
- integration with other invariant generation techniques



Magpie-lark