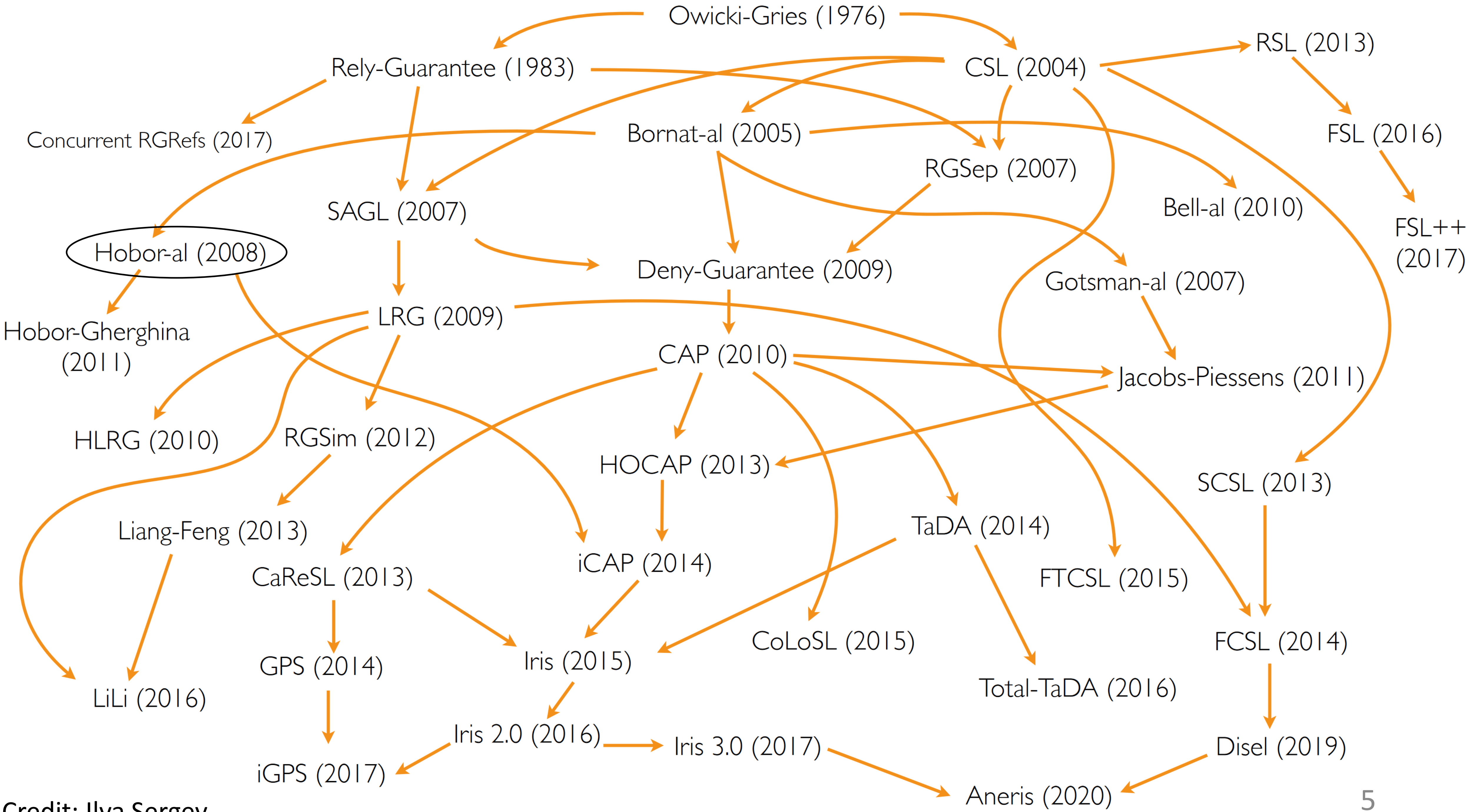
Integrating Iris into the Verified Software Toolchain, and vice versa

William Mansky, University of Illinois Chicago

Iris Workshop, May 23rd, 2023

The Verified Software Toolchain (VST)

- Concurrent separation logic verifier in Coq
- Higher-order, step-indexed, symbolic execution + entailment solving
- Specialized to C, connected to CompCert
  - End-to-end soundness theorem



Owicki-Gries (1976)

Rely-Guarantee (1983)

CSL (2004)

RSL (2013)

Concurrent RGRefs (2017)

Bornat-al (2005)

FSL (2016)

RGSep (2007)

SAGL (2007)

Bell-al (2010)

FSL++ (2017)

Hobor-al (2008)

Deny-Guarantee (2009)

Gotsman-al (2007)

LRG (2009)

Hobor-Gherghina (2011)

CAP (2010)

Jacobs-Piessens (2011)

HLRG (2010)

RGSim (2012)

HOCAP (2013)

SCSL (2013)

Liang-Feng (2013)

TaDA (2014)

CaReSL (2013)

iCAP (2014)

FTCSL (2015)

CoLoSL (2015)

FCSL (2014)

GPS (2014)

Iris (2015)

Total-TaDA (2016)

LiLi (2016)

Iris 2.0 (2016)

Iris 3.0 (2017)

Disel (2019)

iGPS (2017)

Aneris (2020)

Credit: Ilya Sergey

Iris and VST

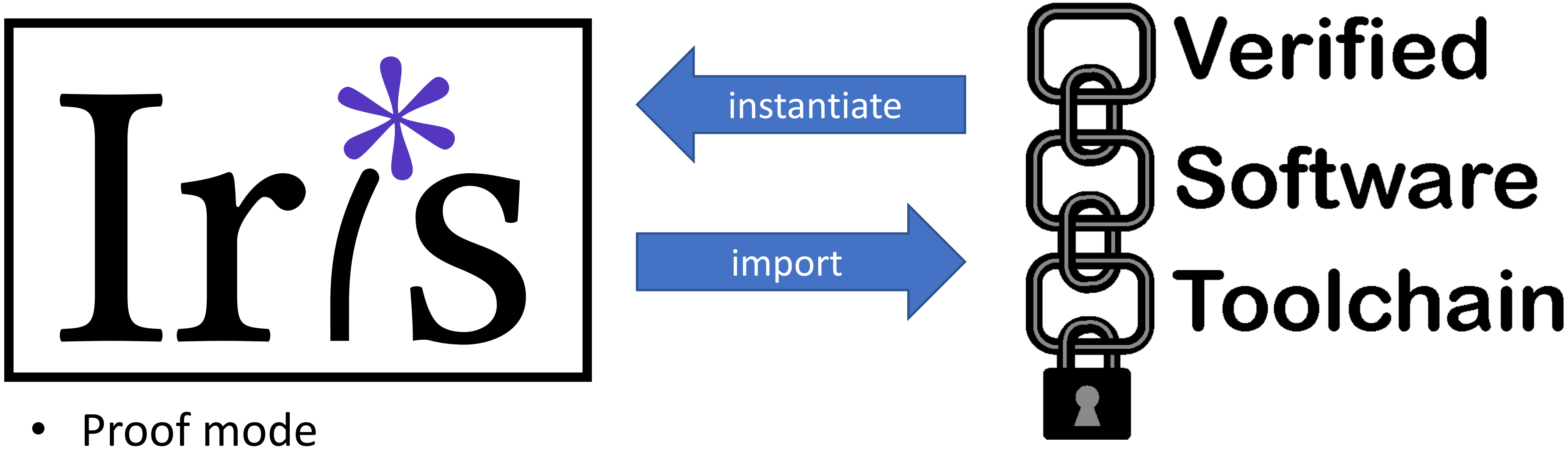
Iris

- Proof mode
- Custom ghost state
- Invariants
- Logical atomicity
- …

Verified
Software
Toolchain

- C isn’t garbage-collected, so logic shouldn’t be affine
- Ownership can’t be “just ghost state”: it’s translated to CompCert permissions and used for adequacy

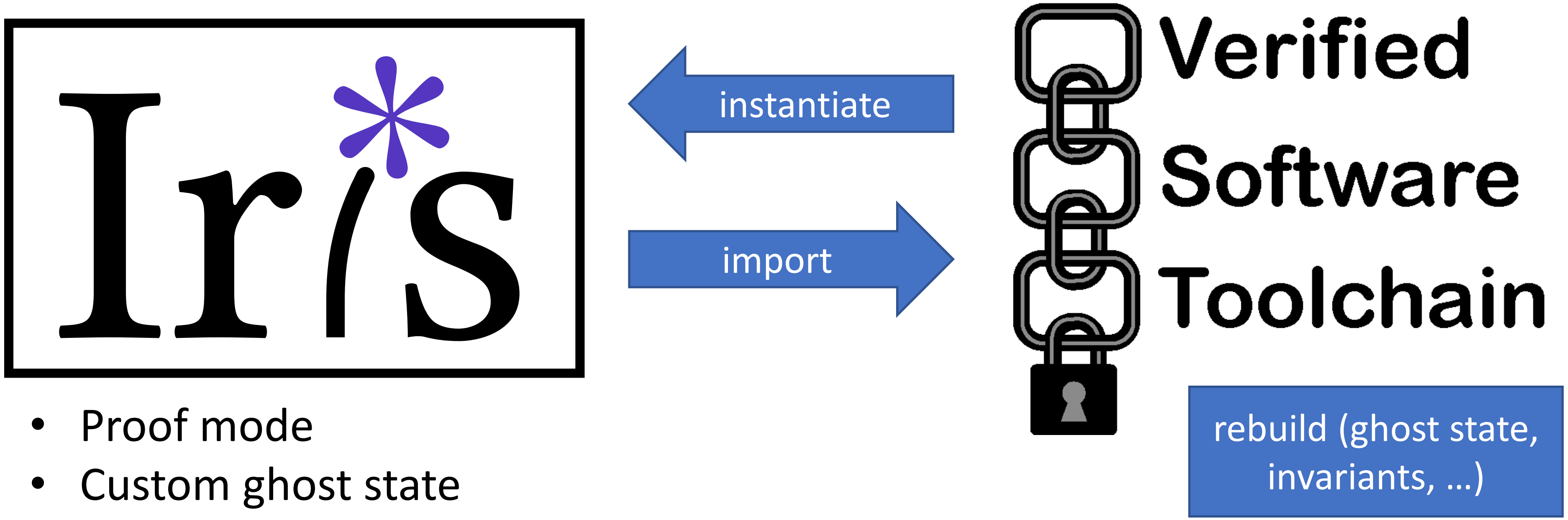
Iris in VST



- Proof mode
- Custom ghost state
- Invariants
- Logical atomicity
- …

- Keep VST’s foundations the same, import or reconstruct the features we want

Iris in VST



instantiate

import

rebuild (ghost state, invariants, …)

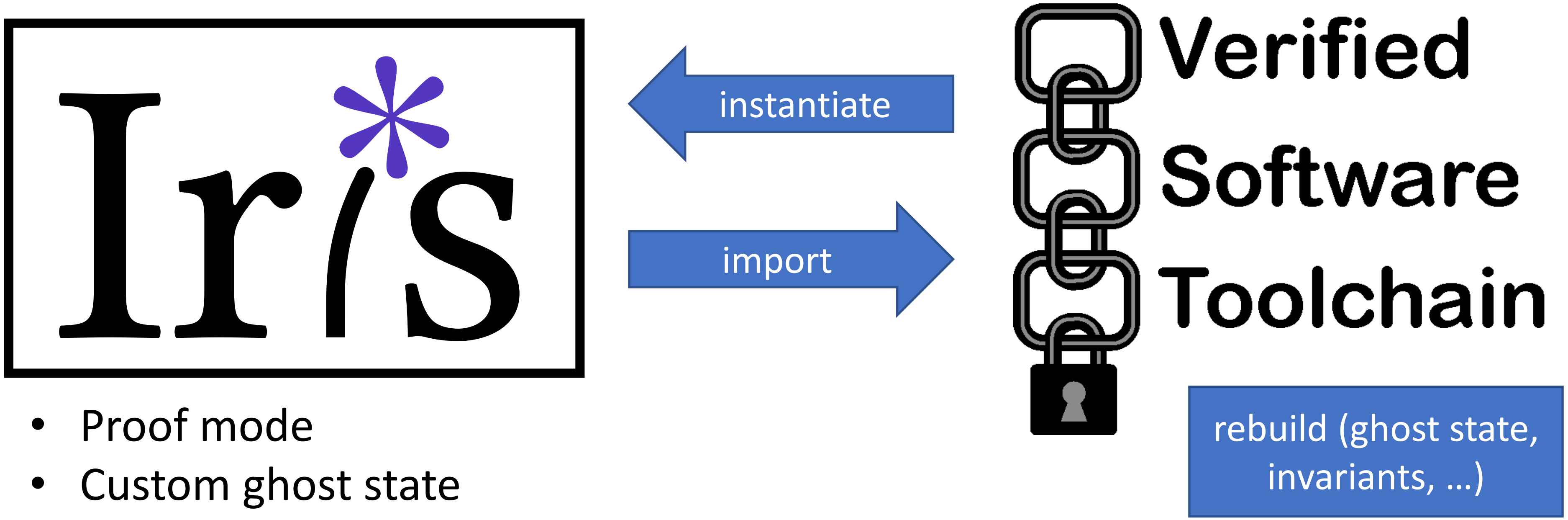
- Proof mode
- Custom ghost state
- Invariants
- Logical atomicity
- …

- Keep ~~VST's foundations the same~~ the core of the model and soundness proof, import or reconstruct the features we want

VST on Iris

- Replace VST’s foundations with Iris, rebuild the rest of VST on top, get Iris features for free

Iris in VST



rebuild (ghost state, invariants, …)

- Proof mode
- Custom ghost state
- Invariants
- Logical atomicity
- …

- Keep ~~VST's foundations the same~~ the core of the model and soundness proof, import or reconstruct the features we want

VST + MoSeL

Get IPM by instantiating the BI interface with VST's logic

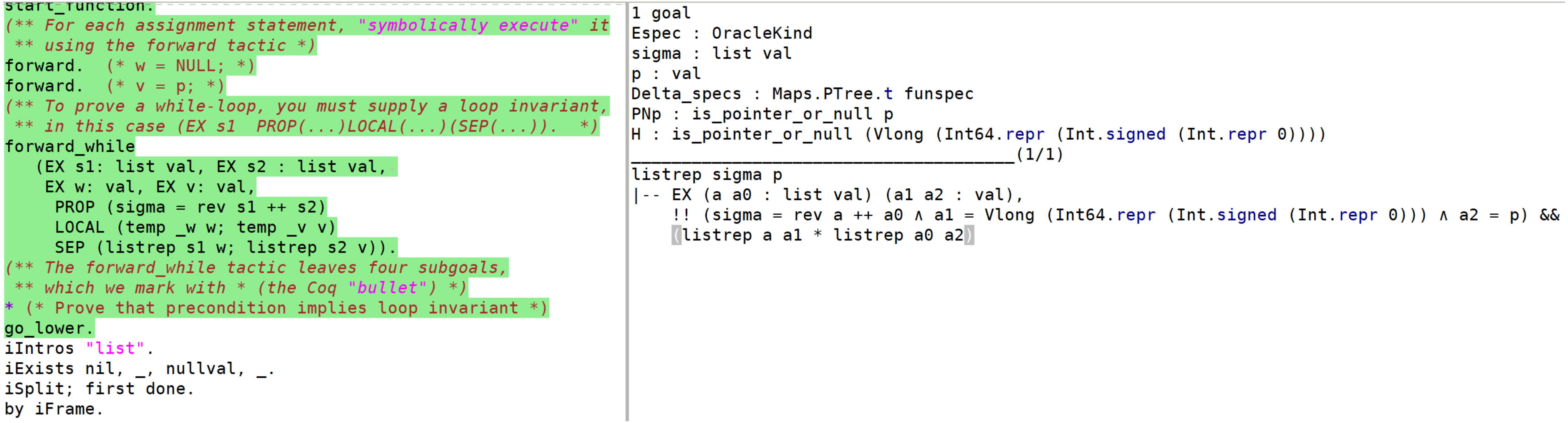
Non-obvious parts of BI:

- Step-indexing and $\triangleright$ — but they're exactly the same in VST as in Iris
- Persistence ($\boxdot$) modality
  - Default definition: $(\boxdot P)(x)$ when $(P)(\mathrm{core}\ x)$
  - VST has core too, but core is always emp!
  - Simple instantiation: $(\boxdot P)(x)$ when $(P \wedge \mathrm{emp})(x)$

MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic, Krebbers et al., ICFP 2018

VST + MoSeL

Immediately get MoSeL tactics for VST’s separation logic

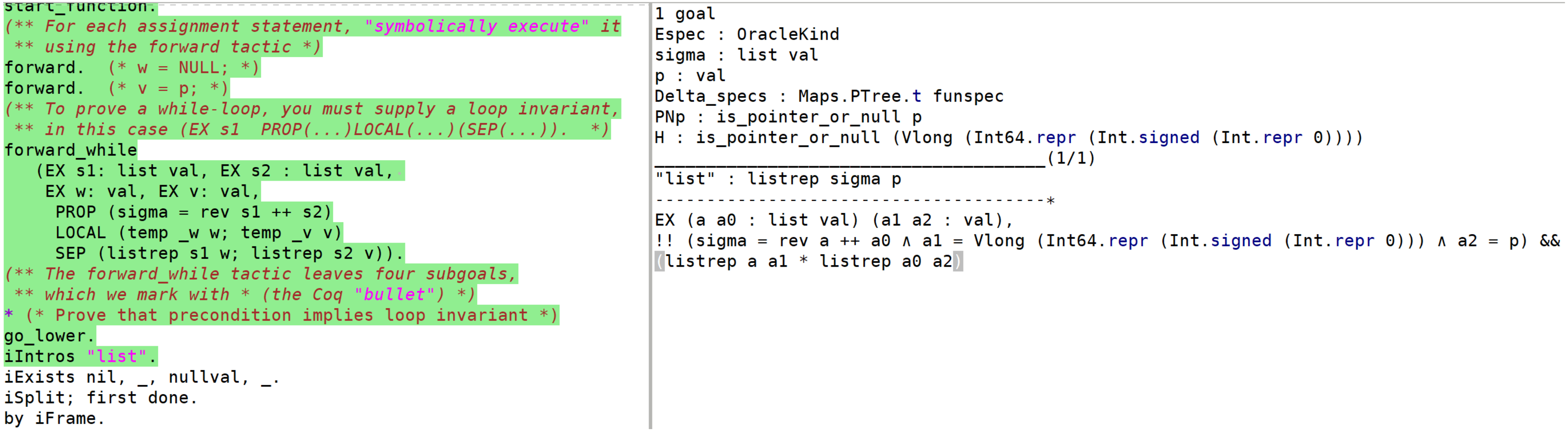


```
start_function.
(** For each assignment statement, "symbolically execute" it
 ** using the forward tactic *)
forward.  (* w = NULL; *)
forward.  (* v = p; *)
(** To prove a while-loop, you must supply a loop invariant,
 ** in this case (EX s1  PROP(...)LOCAL(...)(SEP(...)).  *)
forward_while
   (EX s1: list val, EX s2 : list val,
    EX w: val, EX v: val,
     PROP (sigma = rev s1 ++ s2)
     LOCAL (temp _w w; temp _v v)
     SEP (listrep s1 w; listrep s2 v)).
(** The forward_while tactic leaves four subgoals,
 ** which we mark with * (the Coq "bullet") *)
* (* Prove that precondition implies loop invariant *)
go_lower.
iIntros "list".
iExists nil, _, nullval, _.
iSplit; first done.
by iFrame.
```

```
1 goal
Espec : OracleKind
sigma : list val
p : val
Delta_specs : Maps.PTree.t funspec
PNp : is_pointer_or_null p
H : is_pointer_or_null (Vlong (Int64.repr (Int.signed (Int.repr 0))))
______________________________________(1/1)
listrep sigma p
|-- EX (a a0 : list val) (a1 a2 : val),
    !! (sigma = rev a ++ a0 ∧ a1 = Vlong (Int64.repr (Int.signed (Int.repr 0))) ∧ a2 = p) &&
    (listrep a a1 * listrep a0 a2)
```

VST + MoSeL

Immediately get MoSeL tactics for VST's separation logic



```
start_function.
(** For each assignment statement, "symbolically execute" it
 ** using the forward tactic *)
forward.  (* w = NULL; *)
forward.  (* v = p; *)
(** To prove a while-loop, you must supply a loop invariant,
 ** in this case (EX s1  PROP(...)LOCAL(...)(SEP(...)).  *)
forward_while
   (EX s1: list val, EX s2 : list val,
    EX w: val, EX v: val,
     PROP (sigma = rev s1 ++ s2)
     LOCAL (temp _w w; temp _v v)
     SEP (listrep s1 w; listrep s2 v)).
(** The forward_while tactic leaves four subgoals,
 ** which we mark with * (the Coq "bullet") *)
* (* Prove that precondition implies loop invariant *)
go_lower.
iIntros "list".
iExists nil, _, nullval, _.
iSplit; first done.
by iFrame.
```

```
1 goal
Espec : OracleKind
sigma : list val
p : val
Delta_specs : Maps.PTree.t funspec
PNp : is_pointer_or_null p
H : is_pointer_or_null (Vlong (Int64.repr (Int.signed (Int.repr 0))))
______________________________________(1/1)
"list" : listrep sigma p
--------------------------------------*
EX (a a0 : list val) (a1 a2 : val),
!! (sigma = rev a ++ a0 ∧ a1 = Vlong (Int64.repr (Int.signed (Int.repr 0))) ∧ a2 = p) &&
(listrep a a1 * listrep a0 a2)
```

VST + MoSeL

Immediately get MoSeL tactics for VST's separation logic

- Don't get `wp` tactics, but VST has its own (`forward`)
- Don't yet get tactics for invariants, updates, atomic updates, etc. – those have their own classes to instantiate
- But first, we need those things to exist in VST!

MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic, Krebbers et al., ICFP 2018

Ghost State in VST

- Model of Iris: $M \triangleq \prod_{i \in I} \mathbb{N} \rightharpoonup M_i$,
  where cameras M_i may include predicates

- Model of VST ("rmap"): $R \triangleq \mathrm{loc} \rightharpoonup \mathrm{res}$,
  where res may include predicates ("predicates in the heap")

- New model of VST: $R * M$
  - In practice, we don't get HO ghost state, just agreement

Ghost State Updates in VST

- New model of VST: $R * M$ (approximately)
- Now we can define $\Rrightarrow$, and add updates between steps in our semantics
  - And instantiate BUpd class, use `iMod` and `iModIntro`
- Hoare rules are unchanged, since program ops ignore ghost state
- Adequacy proof basically unchanged

- Used for simple double-increment example, external state reasoning

Verifying an HTTP Key-Value Server with Interaction Trees and VST, Zhang et al., ITP 2021

Invariants in VST

Verified
Software
Toolchain

- Invariants can be built out of ghost state
- In Iris: $W \triangleq \bullet(\blacktriangleright I) * \mathop{\Huge *}_{i \in \mathrm{dom}(I)} (\triangleright I(i) * \mathrm{dis}(i)) \vee \mathrm{en}(i)$ where I is a map from names to assertions
- This is exactly what we can't do with our restricted HO ghost state!
- Refactored construction:

$$W \triangleq \bullet\, G * \mathop{\Huge *}_{i \in \mathrm{dom}(G)} \mathrm{ag}_{G(i)} I(i) * ((\triangleright I(i) * \mathrm{dis}(i)) \vee \mathrm{en}(i))$$

- Satisfies all the same proof rules, and we can build namespaces, instantiate proofmode classes for invariants, etc. on top of it

Fancy Updates in VST

Verified
Software
Toolchain

WP-VUP

$$\mathrel{\vert\!\!\Rrightarrow}_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e \left\{v.\, \mathrel{\vert\!\!\Rrightarrow}_{\mathcal{E}} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

WP-ATOMIC

$$\frac{\boxed{\text{atomic}}(e)}{{}^{\mathcal{E}_1}\!\mathrel{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e \left\{v.\, {}^{\mathcal{E}_2}\!\mathrel{\vert\!\!\Rrightarrow}^{\mathcal{E}_1} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

$$\{\boxed{x \mapsto 3 \vee x \mapsto 4}\}$$
$$x \leftarrow 3 \mid\mid x \leftarrow 4$$
$$\{\boxed{x \mapsto 3 \vee x \mapsto 4}\}$$

- In C, this is undefined behavior!

Fancy Updates in VST

Verified
Software
Toolchain

WP-VUP

$$\mid\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\; e \left\{v.\ \mid\!\Rrightarrow_{\mathcal{E}} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}$$

WP-ATOMIC

$$\frac{\boxed{\text{atomic}}(\mathrm{e})}{{}^{\mathcal{E}_1}\!\mid\!\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\; e \left\{v.\ {}^{\mathcal{E}_2}\!\mid\!\Rrightarrow^{\mathcal{E}_1} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\; e\; \{\Phi\}}$$

$$\{\boxed{\mathrm{x} \mapsto 3 \vee \mathrm{x} \mapsto 4}\}$$
$$\mathrm{x} \leftarrow 3 \;||\; \mathrm{x} \leftarrow 4$$
$$\{\boxed{\mathrm{x} \mapsto 3 \vee \mathrm{x} \mapsto 4}\}$$

- In C, this is undefined behavior!
- We set atomic to mean concurrency-atomic: lock acquire/release, atomic_load/store, etc., and nothing else

Fancy Updates in VST

Iris

Verified
Software
Toolchain

WP-VUP

$$\models\!\!\!\Rightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\; e \left\{v.\; \models\!\!\!\Rightarrow_{\mathcal{E}} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}$$

WP-ATOMIC

$$\frac{\mathsf{atomic(e)}}{{}^{\mathcal{E}_1}\!\models\!\!\!\Rightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\; e \left\{v.\; {}^{\mathcal{E}_2}\!\models\!\!\!\Rightarrow^{\mathcal{E}_1} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\; e\; \{\Phi\}}$$

- We set atomic to mean concurrency-atomic: lock acquire/release, atomic_load/store, etc.

- Unlike basic updates, this changes the semantics: “real” resources can change hands between steps
- For concurrent soundness, have to prove race-freedom, which seems true but not obvious

Persistence in VST

- In Iris: invariants are *persistent*, can freely be automatically duplicated and passed between threads
- In VST: we defined $\boxdot P$ to only hold on emp!
- Invariants really need to be affine too, but in VST nothing is affine!

- Step 1: weaken the core axiom
- Step 2: make the logic *semi-linear*

Persistence in VST

Step 1: weaken the core axiom

VST: $a \preccurlyeq b \rightarrow \mathrm{core}(b) = \mathrm{core}(a)$

Iris: $a \preccurlyeq b \rightarrow \mathrm{core}(b) \preccurlyeq \mathrm{core}(a)$

- In Iris, the core of $(\mathbb{N}, +)$ can tell us “the value is at least n”; in VST, it can only tell us “the value is at least 0”
- Simple solution: weaken VST’s core axiom
  - Heap resources still have trivial cores, but ghost state doesn’t have to
  - Now we can define useful persistence
  - And all the existing proofs still work

Persistence in VST

Iris

Verified
Software
Toolchain

Step 2: make the logic *semi-linear*

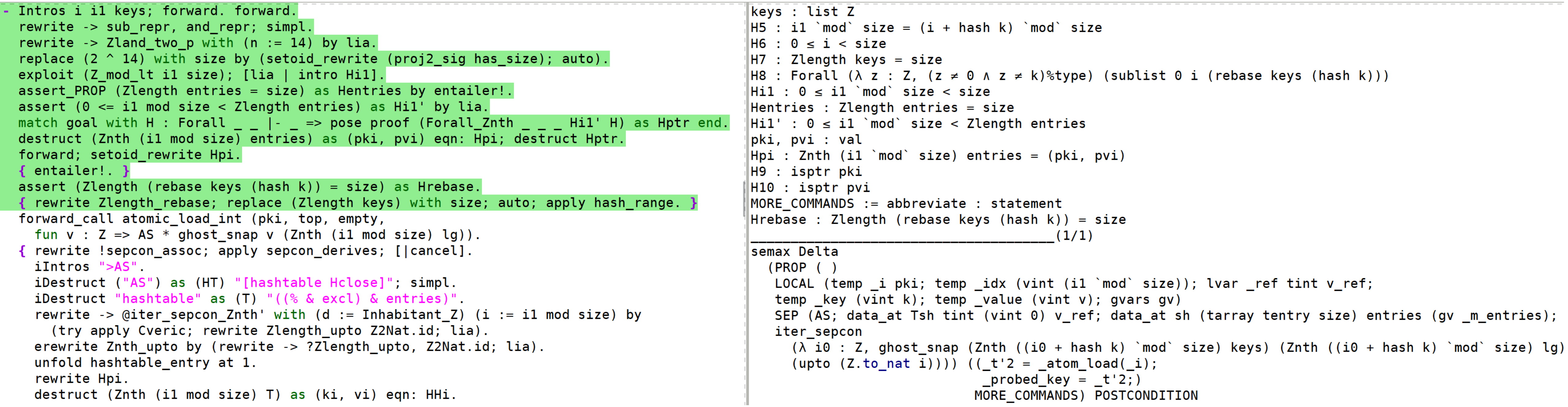
- ORA idea from MoSeL: equip algebras with an *extension order* $\sqsubseteq$ describing which resources can be thrown away
- Define predicates to be closed under $\sqsubseteq$
- Surprisingly, VST's model also has a slot for this order! Included in 2009, never mentioned in a paper or instantiated nontrivially
- We choose: $(r, m) \sqsubseteq (r', m') \triangleq r' = r \land m \preccurlyeq m'$
- Now all ghost state is affine, and all ghost state cores (including invariants) are intuitionistic!

Using Iris in VST

Iris

Verified
Software
Toolchain

- We now have custom ghost state, invariants, updates, and all the relevant Iris tactics in VST
- Can import definitions like logical atomicity directly

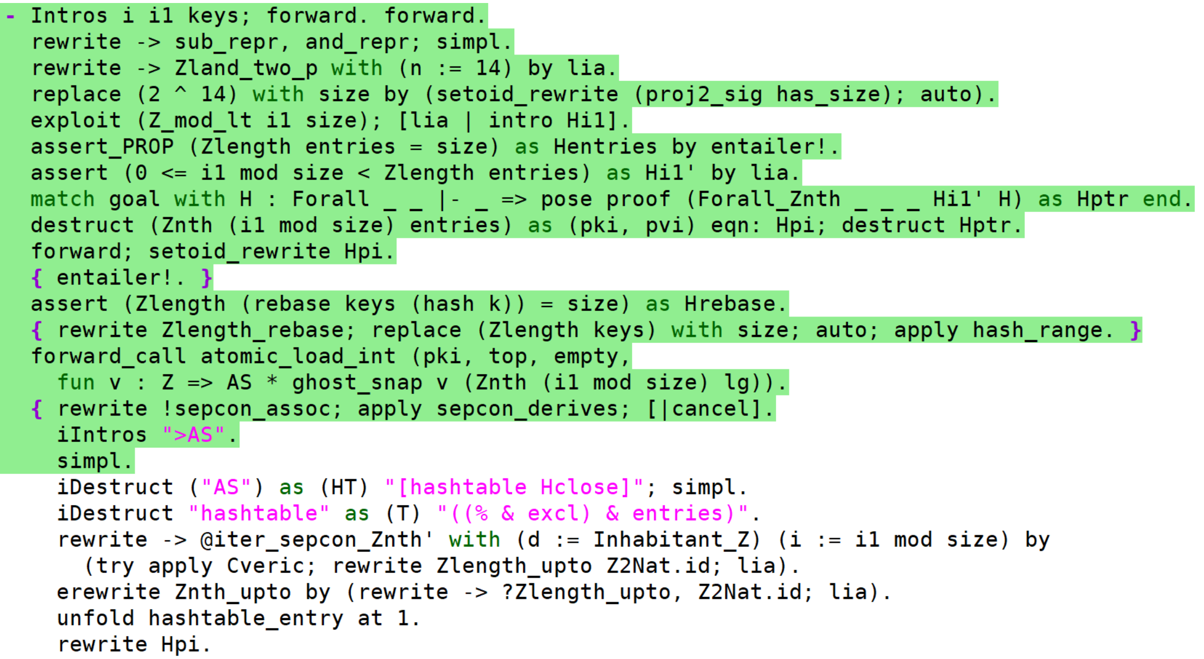


```
- Intros i i1 keys; forward. forward.
  rewrite -> sub_repr, and_repr; simpl.
  rewrite -> Zland_two_p with (n := 14) by lia.
  replace (2 ^ 14) with size by (setoid_rewrite (proj2_sig has_size); auto).
  exploit (Z_mod_lt i1 size); [lia | intro Hi1].
  assert_PROP (Zlength entries = size) as Hentries by entailer!.
  assert (0 <= i1 mod size < Zlength entries) as Hi1' by lia.
  match goal with H : Forall _ _ |- _ => pose proof (Forall_Znth _ _ _ Hi1' H) as Hptr end.
  destruct (Znth (i1 mod size) entries) as (pki, pvi) eqn: Hpi; destruct Hptr.
  forward; setoid_rewrite Hpi.
  { entailer!. }
  assert (Zlength (rebase keys (hash k)) = size) as Hrebase.
  { rewrite Zlength_rebase; replace (Zlength keys) with size; auto; apply hash_range. }
  forward_call atomic_load_int (pki, top, empty,
    fun v : Z => AS * ghost_snap v (Znth (i1 mod size) lg)).
  { rewrite !sepcon_assoc; apply sepcon_derives; [|cancel].
    iIntros ">AS".
    iDestruct ("AS") as (HT) "[hashtable Hclose]"; simpl.
    iDestruct "hashtable" as (T) "((% & excl) & entries)".
    rewrite -> @iter_sepcon_Znth' with (d := Inhabitant_Z) (i := i1 mod size) by
      (try apply Cveric; rewrite Zlength_upto Z2Nat.id; lia).
    erewrite Znth_upto by (rewrite -> ?Zlength_upto, Z2Nat.id; lia).
    unfold hashtable_entry at 1.
    rewrite Hpi.
    destruct (Znth (i1 mod size) T) as (ki, vi) eqn: HHi.
```

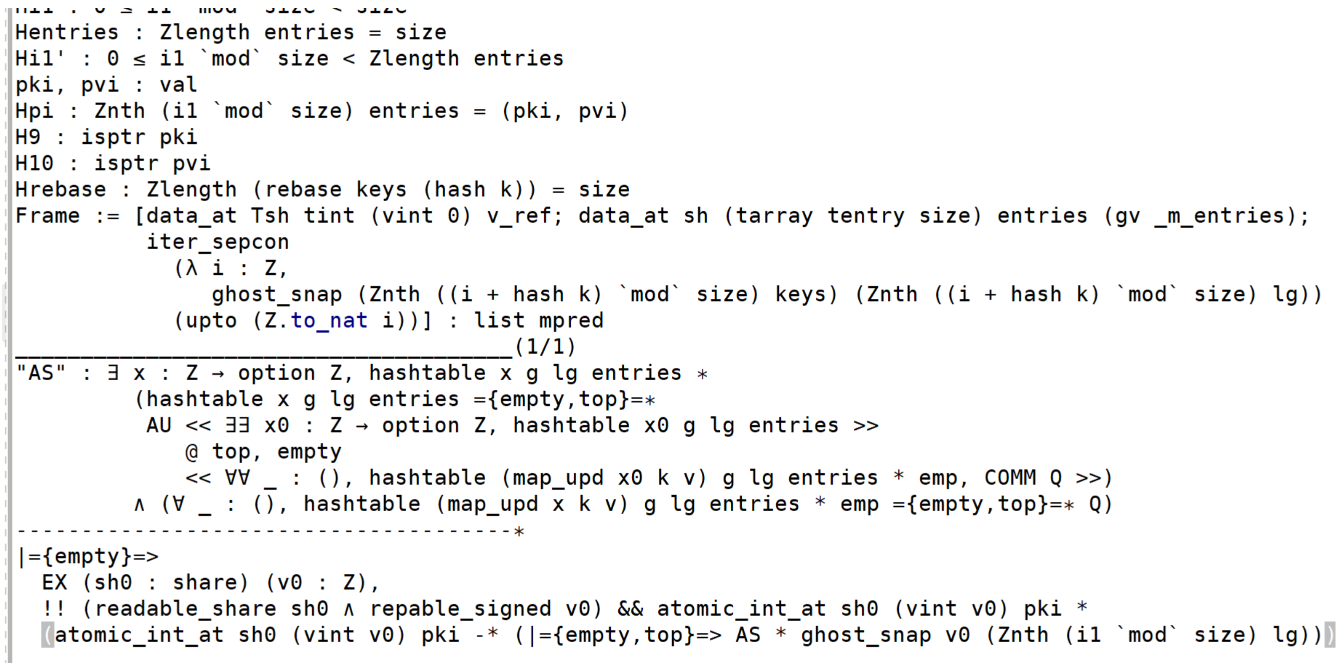
```
keys : list Z
H5 : i1 `mod` size = (i + hash k) `mod` size
H6 : 0 ≤ i < size
H7 : Zlength keys = size
H8 : Forall (λ z : Z, (z ≠ 0 ∧ z ≠ k)%type) (sublist 0 i (rebase keys (hash k)))
Hi1 : 0 ≤ i1 `mod` size < size
Hentries : Zlength entries = size
Hi1' : 0 ≤ i1 `mod` size < Zlength entries
pki, pvi : val
Hpi : Znth (i1 `mod` size) entries = (pki, pvi)
H9 : isptr pki
H10 : isptr pvi
MORE_COMMANDS := abbreviate : statement
Hrebase : Zlength (rebase keys (hash k)) = size
______________________________________(1/1)
semax Delta
  (PROP ( )
   LOCAL (temp _i pki; temp _idx (vint (i1 `mod` size)); lvar _ref tint v_ref;
   temp _key (vint k); temp _value (vint v); gvars gv)
   SEP (AS; data_at Tsh tint (vint 0) v_ref; data_at sh (tarray tentry size) entries (gv _m_entries);
   iter_sepcon
     (λ i0 : Z, ghost_snap (Znth ((i0 + hash k) `mod` size) keys) (Znth ((i0 + hash k) `mod` size) lg)
     (upto (Z.to_nat i)))) ((_t'2 = _atom_load(_i);
                            _probed_key = _t'2;)
                           MORE_COMMANDS) POSTCONDITION
```

Using Iris in VST

- We now have custom ghost state, invariants, updates, and all the relevant Iris tactics in VST
- Can import definitions like logical atomicity directly



```
- Intros i i1 keys; forward. forward.
  rewrite -> sub_repr, and_repr; simpl.
  rewrite -> Zland_two_p with (n := 14) by lia.
  replace (2 ^ 14) with size by (setoid_rewrite (proj2_sig has_size); auto).
  exploit (Z_mod_lt i1 size); [lia | intro Hi1].
  assert_PROP (Zlength entries = size) as Hentries by entailer!.
  assert (0 <= i1 mod size < Zlength entries) as Hi1' by lia.
  match goal with H : Forall _ _ |- _ => pose proof (Forall_Znth _ _ _ Hi1' H) as Hptr end.
  destruct (Znth (i1 mod size) entries) as (pki, pvi) eqn: Hpi; destruct Hptr.
  forward; setoid_rewrite Hpi.
  { entailer!. }
  assert (Zlength (rebase keys (hash k)) = size) as Hrebase.
  { rewrite Zlength_rebase; replace (Zlength keys) with size; auto; apply hash_range. }
  forward_call atomic_load_int (pki, top, empty,
    fun v : Z => AS * ghost_snap v (Znth (i1 mod size) lg)).
  { rewrite !sepcon_assoc; apply sepcon_derives; [|cancel].
    iIntros ">AS".
    simpl.
    iDestruct ("AS") as (HT) "[hashtable Hclose]"; simpl.
    iDestruct "hashtable" as (T) "((% & excl) & entries)".
    rewrite -> @iter_sepcon_Znth' with (d := Inhabitant_Z) (i := i1 mod size) by
      (try apply Cveric; rewrite Zlength_upto Z2Nat.id; lia).
    erewrite Znth_upto by (rewrite -> ?Zlength_upto, Z2Nat.id; lia).
    unfold hashtable_entry at 1.
    rewrite Hpi.
```



```
Hentries : Zlength entries = size
Hi1' : 0 ≤ i1 `mod` size < Zlength entries
pki, pvi : val
Hpi : Znth (i1 `mod` size) entries = (pki, pvi)
H9 : isptr pki
H10 : isptr pvi
Hrebase : Zlength (rebase keys (hash k)) = size
Frame := [data_at Tsh tint (vint 0) v_ref; data_at sh (tarray tentry size) entries (gv _m_entries);
          iter_sepcon
            (λ i : Z,
               ghost_snap (Znth ((i + hash k) `mod` size) keys) (Znth ((i + hash k) `mod` size) lg))
            (upto (Z.to_nat i))] : list mpred
_____________________________________(1/1)
"AS" : ∃ x : Z → option Z, hashtable x g lg entries ∗
         (hashtable x g lg entries ={empty,top}=∗
          AU << ∃∃ x0 : Z → option Z, hashtable x0 g lg entries >>
             @ top, empty
             << ∀∀ _ : (), hashtable (map_upd x0 k v) g lg entries * emp, COMM Q >>)
         ∧ (∀ _ : (), hashtable (map_upd x k v) g lg entries * emp ={empty,top}=∗ Q)
--------------------------------------∗
|={empty}=>
  EX (sh0 : share) (v0 : Z),
  !! (readable_share sh0 ∧ repable_signed v0) && atomic_int_at sh0 (vint v0) pki *
  (atomic_int_at sh0 (vint v0) pki -* (|={empty,top}=> AS * ghost_snap v0 (Znth (i1 `mod` size) lg)))
```

Iris in VST: Summary

- All the concurrency features of Iris, in VST
- Foundational changes: ghost state in the model, weaker core axiom, extension order for affine ghost state, fancy updates in the semantics
- Now we can prove atomic specs for concurrent C programs, using VST for C code and switching to Iris tactics for concurrency reasoning
- Could be useful for other non-Iris verifiers that want Iris features
- Can now reconstruct ghost-state-based reasoning in VST, e.g. ReLoC

- Paper on arXiv, opam package `coq-vst-iris`

Iris in VST: Summary

- Now we can prove atomic specs for concurrent C programs, using VST for C code and switching to Iris tactics for concurrency

But:

- Concurrent soundness is still complicated
- We’re reconstructing Iris features, and there’s always more we might want to reconstruct (transfinite step-indexes, later credits, …)
- We’re working in parallel to RefinedC and the whole Iris ecosystem

- What if VST was built on Iris instead?

Iris and VST

Iris

- Proof mode
- Custom ghost state
- Invariants
- Logical atomicity
- …

Verified
Software
Toolchain

- C isn’t garbage-collected, so logic shouldn’t be affine

We can use ORAs!

- Ownership can’t be “just ghost state”: it’s translated to CompCert permissions and used for adequacy

Need a fancier relationship between physical state and mapsto assertions

VST on Iris

- Replace VST’s foundations with Iris, rebuild the rest of VST on top, get Iris features for free

VST on Iris: “juicy” view

- State interpretation: $\bullet\, \sigma$ where σ is a map from locations to values
- Maps-to: $l \mapsto v$ is defined as $\circ\, \{[l := v]\}$

$$\bullet\, \sigma \ * \ l \mapsto v \ \vdash \sigma(l) = v$$

- In VST, these don’t coincide!
  - Physical memory (CompCert) maps locations to values + permissions (readable, writable, etc.)
  - Logical memory maps locations to rmap resources + shares
  - Semantics defined in terms of a “juicy mem” that includes both CompCert mem and rmap, plus proof that they are **coherent**

VST on Iris: “juicy” view

- General *views*: parameterized by a relation R, give:
  - $a \ast \circ b \vdash R\ a\ b$
- In VST, we can choose $R \triangleq \mathbf{coherent}$, and get:
  - $m \ast l \mapsto_{\pi} v \vdash \mathbf{coherent}\ m\ l\ \pi\ v$

VST on Iris: “juicy” view

- General *views*: parameterized by a relation R, give:

$$\bullet\, a \ast \circ\, b \vdash R\ a\ b$$

- In VST, we can choose $R \triangleq \mathbf{coherent}$, and get:

$$\bullet\, m \ast l \mapsto_{\pi} v \vdash \mathbf{coherent}\ m\ l\ \pi\ v$$

Old VST: $\forall j.\, (l \mapsto v)\ (\mathrm{rmap\_of}\ j) \to \mathrm{valid\_pointer}\ l\ (\mathrm{mem\_of}\ j)$

VST on Iris: $\bullet\, m \ast l \mapsto v \vdash \ulcorner\ \mathrm{valid\_pointer}\ l\ m\ \urcorner$

VST on Iris: semantics

- Iris: $\mathrm{wp}\ e\ \{\Phi\}$ when either e is terminated in a state satisfying Φ, or $S(\sigma) \Rrightarrow (e,\sigma) \to (e',\sigma') \Rrightarrow S(\sigma') * \mathrm{wp}\ e'\ \{\Phi\}$
- VST defines *safety* similarly, except that there are two kinds of steps:
  - *Core steps* are steps by the Clight semantics
  - *External calls* call arbitrary external functions with provided pre- and postconditions
- Safety was originally defined as a relation on juicy mems, but we can rephrase it inside the logic analogously to wp

VST on Iris: program logic

- Proved exactly the same triples for C statements (mod. Iris notation)
- Proofs are about ½ the size of old versions
  - #1 reduction: reasoning at the logic level instead of unfolding to the model
  - #2 reduction: proof mode tactics

VST on Iris: adequacy

- Still in progress: should be the same paper proof, but in Iris terms
- Aim to prove as much as possible (probably everything!) in the logic instead of unfolding to the model

- VST has complicated armature for lifting CompCert’s soundness to concurrency; it should be easier with Iris, but basically the same
- We’re long overdue for a better approach to compiler correctness for concurrency! Happy to talk if you have ideas.

VST on Iris: user interface

- Still need to rebuild symbolic execution tactics and automation

- Interaction mode 1: VST + Iris
  - Can do anything we did before in VST in exactly the same way
  - Drop into Iris proof mode as desired for invariants, atomics, etc.

- Interaction mode 2: Iris style
  - Turn Hoare triples into WP format, stay in IPM the whole time
  - Will require retooling VST's automation (`forward`, etc.) to work on IPM goals
  - More comfortable for Iris people, could adapt Diaframe

Conclusion

Verified
Software
Toolchain

Iris

- Iris in VST: mostly done
  - Can prove logically atomic specs for C programs using Iris logic and tactics
  - Takes cues from Iris, reuses some of it, rebuilds a lot more

- VST on Iris: looks like it'll work!
  - More expressive ghost state
  - Can incorporate new Iris ideas: transfinite step-indexing, later credits, …
  - Integrate with other tools? Diaframe, RefinedC, …
  - What would you do with a CompCert C mode for Iris?

VST on Iris: ownership

- Iris mapsto is simple: $l \mapsto_q v$, where q is a positive fraction
  - Any q is enough to read, 1 is required to write
- VST uses tree shares, with 4 distinct permission levels (corresponding to CompCert permission levels): nonempty, readable, writable, freeable
- Nonempty ownership gives knowledge of the location, but not its value!

```
Inductive shared :=
| YES (dq : dfrac) (rsh : readable_dfrac dq) (v : agree V)
| NO (sh : share0) (rsh : ¬readable_share' sh).
```

- $l \mapsto_q v$ is $\{[l := \mathrm{YES}\ q\ \_\ v]\}$
- Also have $l \mapsto_q \bot$, which is $\{[l := \mathrm{NO}\ q\ \_]\}$

VST on Iris: resources

- Model of VST: $R \triangleq \mathrm{loc} \rightharpoonup \mathrm{res}$, where res may include predicates ("predicates in the heap")
- $\mathrm{res} \triangleq \mathrm{VAL}\ v \mid \mathrm{LK}\ R \mid \mathrm{FUN}\ A\ P\ Q$

- Last two use "predicates in the heap"
  - But in Iris they don't need to! We'll come back to this

VST on Iris: predicates in the heap

- Model of VST: $R \triangleq \mathrm{loc} \rightharpoonup \mathrm{res}$, where res may include predicates ("predicates in the heap")
- $\mathrm{res} \triangleq \mathrm{VAL}\ v \mid \mathrm{LK}\ R \mid \mathrm{FUN}\ A\ P\ Q$

- We can take the predicates out of the heap, and use ghost state/invariants for them instead

VST on Iris: predicates in the heap

- Model of VST: $R \triangleq \mathrm{loc} \rightharpoonup \mathrm{res}$, where res may include predicates ("predicates in the heap")
- $\mathrm{res} \triangleq \mathrm{VAL}\ v \mid \mathrm{LK} \mid \mathrm{FUN}\ A\ P\ Q$

- We can take the predicates out of the heap, and use ghost state/invariants for them instead
- $\mathrm{isLK}\ l\ R \triangleq l \mapsto \mathrm{LK} * \mathrm{inv}\ R$

VST on Iris: predicates in the heap

- $\mathrm{res} \triangleq \mathrm{VAL}\ v \mid \mathrm{LK} \mid \mathrm{FUN}\ A\ P\ Q$

```
Inductive funspec := mk_funspec: typesig -> calling_convention ->
  forall A (P: A -> mpred) (Q: A -> mpred), funspec.
```

- $l \mapsto \mathrm{FUN}\ A\ P\ Q$ asserts that l is a function pointer w/ spec $\forall a\colon A, \{P\ a\}\ l\ \{Q\ a\}$

- We build an OFE for funspec (roughly isomorphic to $\{A\ \&\ (A \rightarrow \mathrm{mpred}) * (A \rightarrow \mathrm{mpred})\}$)

VST on Iris: predicates in the heap

- $\text{res} \triangleq \text{VAL}\ v \mid \text{LK} \mid \text{FUN}$

```
Inductive funspec := mk_funspec: typesig -> calling_convention ->
  forall A (P: A -> mpred) (Q: A -> mpred), funspec.
```

- $l \mapsto \text{FUN}\ A\ P\ Q$ asserts that l is a function pointer w/ spec $\forall a{:}\,A, \{P\ a\}\ l\ \{Q\ a\}$

- We build an OFE for funspec (roughly isomorphic to $\{A\ \&\ (A \to \text{mpred}) * (A \to \text{mpred})\}$)
- Define isFUN $l\ f \triangleq l \mapsto \text{FUN} * \circ\{[l := \triangleright f]\}$
  - Analogous to invariant construction