Type and Proof Structures for Concurrency

Aleksandar Nanevski
IMDEA Software Institute, Madrid

In collaboration with Ruy-Ley Wild, Ilya Sergey, Anindya Banerjee, German Delbianco, Ignacio Fabregas, Frantishek Farka, Joakim Ohman and Jesus Dominguez

Universidad Complutense de Madrid
April 19, 2022

1

Concurrent programs & their formal proofs

In programs	In formal proofs
Information hiding	Proof of component ***depends*** on state of another
Code abstraction	Proofs overwhelmingly ***detailed***
Code reuse	Must ***redo*** proofs for every new use context

2

2

Applying programming ideas to proofs

Most approaches: ***automate*** spurious proof obligations.

Our approach: ***avoid*** proof obligations by hiding, abstraction & reuse.

Curry-Howard isomorphism: ***proofs = programs***
- for purely-functional programs

Goal: new foundations for concurrent progs, specs & proofs
- Linguistic & math concepts that make proofs scale
- Do for proofs what structured programming did for programming

3

3

Outline

1. Subjective state
2. Specifying ADTs
3. State transition systems as types
4. Function types

4

Starting point: Owicki-Gries auxiliary (ghost) state

`{x = 0}`

$$\langle\ x := x + 1\ \rangle \parallel \langle\ x := x + 1\ \rangle$$

`{x = 2}`

Notation: < e > - lock; execute e; unlock.

Prove without enumerating all thread interleaving

5

5

Starting point: Owicki-Gries auxiliary (ghost) state

Resource invariant: $\boldsymbol{V} = \boldsymbol{x} \mapsto \boldsymbol{a}{+}\boldsymbol{b}$

$$\{x = 0\}$$

$$\left\langle \begin{array}{l} x := x + 1; \\ a := a + 1 \end{array} \right\rangle \parallel \left\langle \begin{array}{l} x := x + 1; \\ b := b + 1 \end{array} \right\rangle$$

$$\{x = 2\}$$

Type-theoretic move

6

6

Proofs depend on thread topology

Say we want to show that a 3-way increment adds 3 to *x*.

$$\left\langle \begin{array}{l} x := x + 1; \\ a := a + 1 \end{array} \right\rangle \parallel \left\langle \begin{array}{l} x := x + 1; \\ b := b + 1 \end{array} \right\rangle \parallel \left\langle \begin{array}{l} x := x + 1; \\ c := c + 1 \end{array} \right\rangle$$

Requires a new resource invariant: $\boldsymbol{V} = \boldsymbol{x} \mapsto \boldsymbol{a}{+}\boldsymbol{b}{+}c$.

Problem: The two-thread subproof can't be reused because it relies on $\boldsymbol{V} = \boldsymbol{x} \mapsto \boldsymbol{a}{+}\boldsymbol{b}$.

7

7

Proofs depend on thread topology

```
incr 0     = skip
incr (n+1) = <x:=x+1> || incr n
```

8

8

How to hide thread topology?

Idea: let's turn Hoare triples into types

- dependent monads
- not a mere syntactic change

$$e : [x_1, \ldots, x_n].\ ST\ \{P\}\{Q\} @ V$$

"logical" variables

What is Hoare type for increment?

$$\langle x := x+1;\ a := a+1 \rangle : \boldsymbol{ST}\ \{a=0\}\{a=1\}@(x \mapsto a+b)$$

$$\langle x := x+1;\ b := b+1 \rangle : \boldsymbol{ST}\ \{b=0\}\{b=1\}@(x \mapsto a+b)$$

What is Hoare type for increment?

$\lambda a.$ $\forall a.$

$$\langle x := x+1;\ a := a+1 \rangle : \boldsymbol{ST}\ \{a=0\}\{a=1\}@(x \mapsto a+b)$$

$$\langle x := x+1;\ b := b+1 \rangle : \boldsymbol{ST}\ \{b=0\}\{b=1\}@(x \mapsto a+b)$$

$\lambda b.$ $\forall b.$

What is Hoare type for increment?

$\lambda a\, b.$ $\forall a\, b.$

$$\langle x := x+1;\ a := a+1 \rangle : \boldsymbol{ST}\ \{a=0\}\{a=1\}@(x \mapsto a+b)$$

$$\langle x := x+1;\ b := b+1 \rangle : \boldsymbol{ST}\ \{b=0\}\{b=1\}@(x \mapsto a+b)$$

$\lambda b\, a.$ $\forall b\, a.$

Subjective ghost variables

Each thread and type should have ***two local*** variables.

- a_s - how much ***"we"*** added to x
- a_o - how much ***"others"*** added to x (novel kind of variable)

13

13

Relating to old ghosts

In 3-way increment:

	left thread	middle thread	right thread
a_s	a	b	c
a_o	$b + c$	$c + a$	$a + b$

Resource invariant $V = x \mapsto (a_s + a_o)$ is same in all threads

The variables a_s and a_o are local but not independent.

14

14

Remodeling parallel composition

parent

$\{ \ a_s = b_1 + b_2, \ a_o = c \ \}$

child$_1$ || child$_2$

parent$'$

15

15

Remodeling parallel composition

parent

$\{ \ a_s = b_1 + b_2, \ a_o = c \ \}$

$\{ \ a_s = b_1, \ a_o = b_2 + c \ \}$ || $\{ \ a_s = b_2, \ a_o = c + b_1 \ \}$

child$_1$ || child$_2$

parent$'$

Once forked, *child$_1$* is part of *child$_2$*'s environment, and vice-versa.

16

16

Remodeling parallel composition

parent

$\{\ a_s = b_1{+}b_2,\ a_o = c\ \}$

$\{\ a_s = b_1,\ a_o = b_2{+}c\ \}$ || $\{\ a_s = b_2,\ a_o = c{+}b_1\ \}$

child₁ child₂

$\{\ a_s = b_1{'},\ a_o = c_1{'}\ \}$ || $\{\ a_s = b_2{'},\ a_o = c_2{'}\ \}$

$\{\ a_s = b_1{'}{+}b_2{'},\ a_o = c_1{'}{-}b_2{'} = c_2{'}{-}b_1{'}\ \}$

parent′

Once forked, *child₁* is part of *child₂*'s environment, and vice-versa.

17

17

Subjective conjunction

$$\frac{e_1 : ST\ \{P_1\}\ \{Q_1\} \qquad e_2 : ST\ \{P_2\}\ \{Q_2\}}{e_1 \parallel e_2 : ST\ \{P_1 \circledast P_2\}\ \{Q_1 \circledast Q_2\}}$$

18

18

Subjective conjunction

$$\frac{e_1 : ST\ \{P_1\}\ \{Q_1\} \qquad e_2 : ST\ \{P_2\}\ \{Q_2\}}{e_1 \parallel e_2 : ST\ \{P_1 \circledast P_2\}\ \{Q_1 \circledast Q_2\}}$$

$(a_s,\ a_o) \vDash P_1 \circledast P_2$ iff

$\exists a_1\ a_2.\ \ a_s = a_1 + a_2$ and

$(a_1,\ a_2 + a_o) \vDash P_1$ and $(a_2,\ a_1 + a_o) \vDash P_2$

19

19

Subjective conjunction

$$\frac{e_1 : ST\ \{P_1\}\ \{Q_1\} \qquad e_2 : ST\ \{P_2\}\ \{Q_2\}}{e_1 \parallel e_2 : ST\ \{P_1 \circledast P_2\}\ \{Q_1 \circledast Q_2\}}$$

$(a_s,\ a_o) \vDash P_1 \circledast P_2$ iff

$\exists a_1\ a_2.\ \ a_s = a_1 + a_2$ and

$(a_1,\ a_2 + a_o) \vDash P_1$ and $(a_2,\ a_1 + a_o) \vDash P_2$

Works for every (partial) commutative, associative operation with unit (PCM)

20

20

Relationship to separation logic

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \parallel e_2\, \{Q_1 * Q_2\}}$$

$a_s \vDash P_1 * P_2$ iff

$\exists a_1\, a_2.\ \ a_s = a_1 \uplus a_2$ and

$a_1 \vDash P_1$ and $a_2 \vDash P_2$

Where a_s is a heap variable and $\uplus$ is disjoint heap union.

Framing in separation logic

if

$e : ST\ \{P\}\ \{Q\}$

then

$e : ST\ \{P * R\}\ \{Q * R\}$

Framing in our system

if

$e : ST\ \{a_s = a \wedge a_o = c\}\ \{a_s = b \wedge a_o = d\}$

then

$e : ST\ \{a_s = a{+}r \wedge a_o = c{-}r\}\ \{a_s = b{+}r \wedge a_o = d{-}r\}$

Fault avoidance

In separation logic:

Verified programs don't fault
if starting state satisfies precondition

In our setting:

Well-typed programs don't go wrong

Conclusion: separation logic = type theory of state

One program/ghost state/proof for all contexts

$$\{a_s = 0, a_o = -\}$$
$$\left\langle \begin{array}{c} x := x + 1; \\ a_s := a_s + 1 \end{array} \right\rangle$$
$$\{a_s = 1, a_o = -\}$$

Code/proof reuse

$$\begin{array}{ccc} & \{a_s = 0, a_o = -\} & \\ \{a_s = 0, a_o = -\} & & \{a_s = 0, a_o = -\} \\ \left\langle \begin{array}{c} x := x + 1; \\ a_s := a_s + 1 \end{array} \right\rangle & \parallel & \left\langle \begin{array}{c} x := x + 1; \\ a_s := a_s + 1 \end{array} \right\rangle \\ \{a_s = 1, a_o = -\} & & \{a_s = 1, a_o = -\} \\ & \{a_s = 2, a_o = -\} & \end{array}$$

Same code, ghost code, proof on both sides of ||.

Code/proof reuse

$$\mathsf{incr}\ 0 \quad = \quad \{a_s = 0, a_o = -\}\ \mathsf{skip}\ \{a_s = 0, a_o = -\}$$

$$\mathsf{incr}\ (n+1) \quad = \quad \begin{array}{ccc} & \{a_s = 0, a_o = -\} & \\ \{a_s = 0, a_o = -\} & & \{a_s = 0, a_o = -\} \\ \left\langle \begin{array}{c} x := x + 1; \\ a_s := a_s + 1 \end{array} \right\rangle & \parallel & \mathsf{incr}\ n \\ \{a_s = 1, a_o = -\} & & \{a_s = n, a_o = -\} \\ & \{a_s = n + 1, a_o = -\} & \end{array}$$

Same code/proof can be substituted into any context

Abstraction and information hiding

$$\{a_s = 0, a_o = -\}$$
$$\mathsf{incr}\ n$$
$$\{a_s = n, a_o = -\}$$

Abstraction and information hiding

$$\text{incr } n \quad : \quad ST \begin{array}{l}\{a_s = 0, a_o = -\} \\ \{a_s = n, a_o = -\}\end{array}$$

29

Abstraction and information hiding

$$\left\langle \begin{array}{c} x := x + n; \\ a_s := a_s + n \end{array} \right\rangle \quad : \quad ST \begin{array}{l}\{a_s = 0, a_o = -\} \\ \{a_s = n, a_o = -\}\end{array}$$

30

Outline

1. Subjective state
2. **Specifying ADTs**
3. State transition systems as types
4. Function types

How to specify stacks?

$$\texttt{push(x)}: [xs].\ ST\ \{a_s = xs\}\ \{a_s = x :: xs\}$$

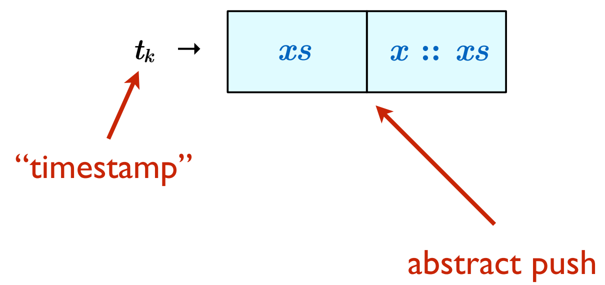
$$\begin{array}{l}\texttt{pop()}: [xs].\ ST\ \{a_s = xs\} \\ \quad \{res = \text{None} \wedge a_s = xs = \text{nil} \\ \quad \vee\ \exists x\ xs'.\ res = \text{Some}\ x\ \wedge \\ \quad\quad xs = x :: xs' \wedge a_s = xs'\}\end{array}$$

Suitable for sequential case, but useless in concurrency

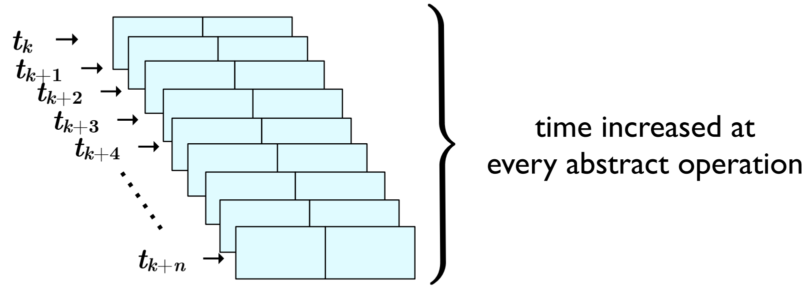
Need PCM for stack effects

32

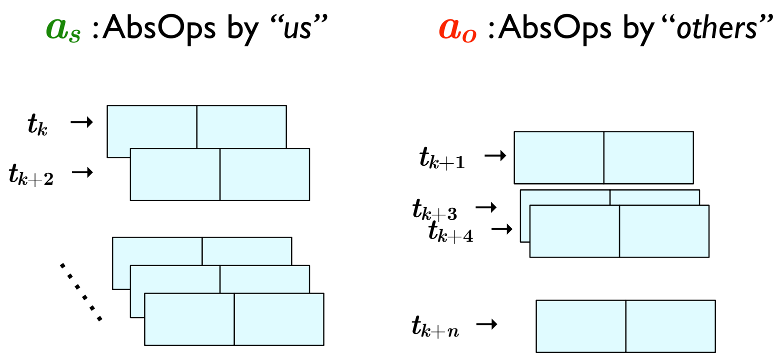
Histories of abstract ops



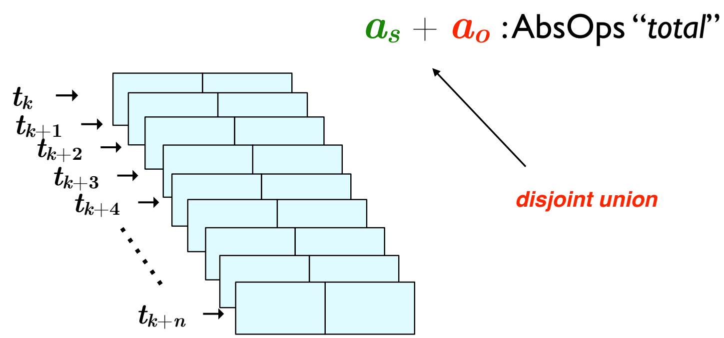
Timestamps capture real time



Subjectivity with histories



Subjectivity with histories



Histories = Heaps as PCM

$\mathbb{H}\text{ist} = (\text{timestamps} \rightharpoonup_{fin} \text{AbsOp}, +, \varnothing)$

time

$\mathbb{H}\text{eap} = (\text{pointers} \rightharpoonup_{fin} \text{Values}, +, \varnothing)$

space

Separation logic = type theory of time as well

Method specs

push(x) : $ST\ \{\, a_s = \varnothing \}$
$\{\, \exists t\ xs.\ a_s = t \mapsto (xs,\ x{::}xs) \,\}$

Method specs

push(x) : $ST\ \{\, a_s = \varnothing \wedge a_o = k \}$
$\{\, \exists t\ xs.\ a_s = t \mapsto (xs,\ x{::}xs) \wedge$
$t > last\ k \}$

Non-local condition

Similar to linearizability, but at user level

Method specs

pop : $[k].\ ST\ \{ a_s = \varnothing \wedge a_o = k \}$
$\{\text{if } res \text{ is } Some\ x \text{ then}$
$\exists t\ xs.\ a_s = t \mapsto (x{::}xs,\ xs) \wedge t > last\ k$
$\text{else } a_s = \varnothing \wedge \exists g.\ k \subseteq g \subseteq a_o \wedge empty\ g \}$

Recording unsuccessful pop is optional

- specifying histories at user level may be useful for relaxing linearizability and implementing other correctness conditions

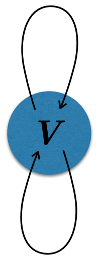
Outline

1. Subjective state
2. Specifying ADTs
3. **State transition systems as types**
4. Function types

How to specify lock-free programs?

Owicki-Gries = Resource Invariant (i.e., set of states)
- must lock ***whole*** stack before modification

$e : \{P\}\{Q\}$ @



For lock-free programs, add transitions:
- atomic moves allowed to the programs
- variant of Rely-Guarantee [Jones 83, Dinsdale-Young et al. 2010]
- only programs of equal resource type compose

Also relevant:
- Abadi+Lamport's refinement mappings
- Lamport's TLA

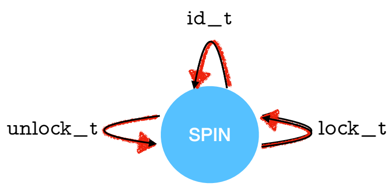
42

Example: spin locks

```
Program lock :=
  do
      x ⟵ CAS (r, U, L)
  while ¬x
```

```
Program unlock :=
  r := U
```


SPIN resource and ghost histories



State space (aka. invariant)

$$r = last\_op\ (a_s + a_o) \wedge alternate\ (a_s + a_o)$$

Transitions:

lock_tr: $\neg locked(a_s + a_o) \wedge a_s' = a_s + fresh(a_s + a_o) \mapsto L$

unlock_tr: $locked(a_s + a_o) \wedge a_s' = a_s + fresh(a_s + a_o) \mapsto U$

Ghost code chooses transitions

```
Program lock :=
  do
     x ⟵ CAS (r, U, L)
  while ¬x
```


Ghost code chooses transitions

```
Program lock :=
  do
     ⟨x ⟵ CAS (r, U, L);
      if x then lock_tr else id_tr⟩
  while ¬x
```

log successful locking to history

Ghost code chooses transitions

```
Program unlock :=
  r := U
```


Ghost code chooses transitions

```
Program unlock :=
  ⟨x ⟵ !r;
   r := U;
   if x = L then unlock_tr else id_tr⟩
```

If called when lock is free, no change to history

Specs for lock and unlock

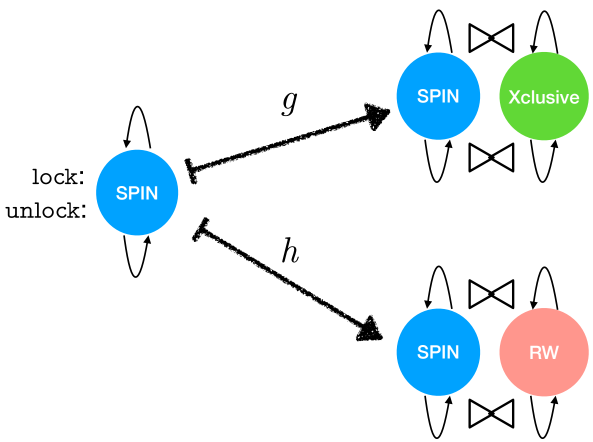
lock : $[k].\ ST\ \{a_s = \emptyset \wedge a_o = k\}$
$\{\exists t.\ a_s = t \mapsto L \wedge t > \mathit{last}\ k\}$@SPIN

unlock : $[k].\ ST\ \{a_s = \emptyset \wedge a_o = k\}$
$\{\exists t.\ a_s = t \mapsto U \wedge t > \mathit{last}\ k\ \vee$
$a_s = \emptyset \wedge \exists g.\ k \subseteq g \subseteq a_o \wedge \mathit{locked}\ g\}$@SPIN

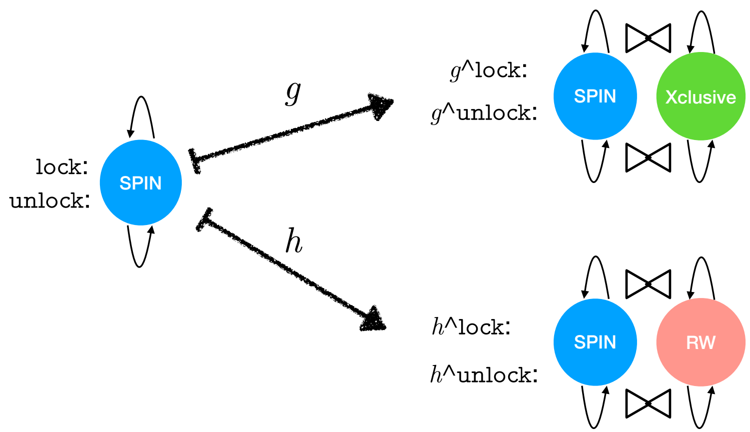
Outline

1. Subjective state
2. Specifying ADTs
3. State transition systems as types
4. **Function types**

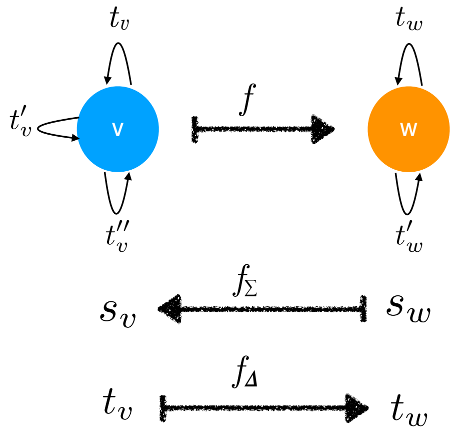
Extending SPIN with new ghost state/ code



Need functions to coerce programs between resources

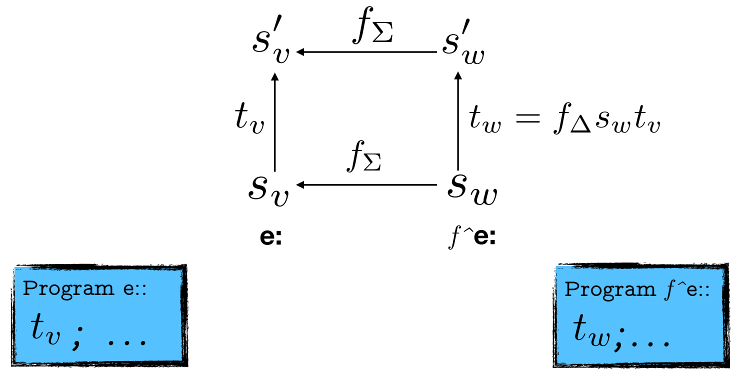


Resource morphism



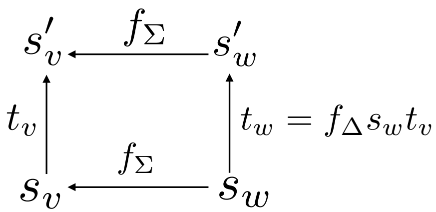
53

Action of morphism f on program e



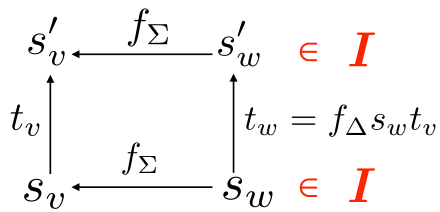
54

Need invariant for the morphing loop



55

Need invariant for the morphing loop

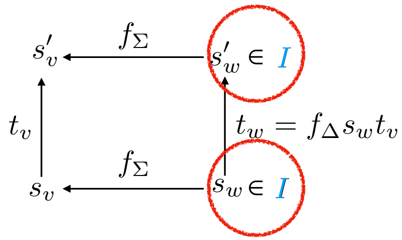


- I is a simulation.

56

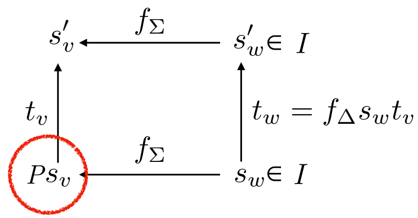
Inference Rule

$$\frac{\text{e: } \{P\}\ \{Q\} @ \text{V}}{f\text{\^{}e: } \{\ \}\ \{\ \} @ \text{W}}$$



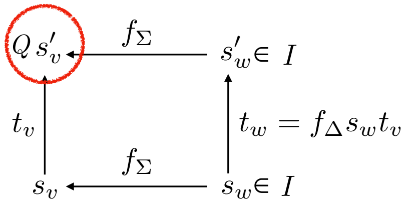
Inference Rule

$$\frac{\text{e: } \{P\}\ \{Q\} @ \text{V}}{f\text{\^{}e: } \{I \wedge \ldots\}\ \{I \wedge \ldots\} @ \text{W}}$$



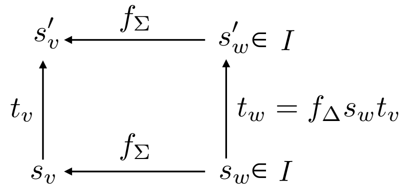
Inference Rule

$$\frac{\text{e: } \{P\}\ \{Q\} @ \text{V}}{f\text{\^{}e: } \{I \wedge f_\Sigma^{-1}\, P\}\ \{I \wedge \ldots\} @ \text{W}}$$



Inference Rule

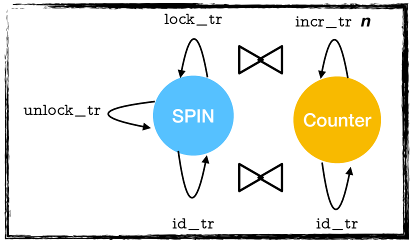
$$\frac{\text{e: } \{P\}\ \{Q\} @ \text{V}}{f\text{\^{}e: } \{I \wedge f_\Sigma^{-1}\, P\}\ \{I \wedge f_\Sigma^{-1}\, Q\} @ \text{W}}$$



Morphing example

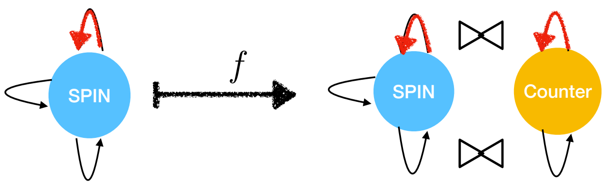
Attaching behaviours to spin locks

- Add *n* to a counter simultaneously with each locking.



CSPIN

Morphism definition

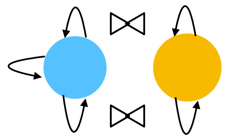


$a_s \xleftarrow{f_\Sigma} (a_s\ ,\ \kappa_s)$

lock_tr $\xrightarrow{f_\Delta}$ lock_tr ⋈ incr_tr n

unlock_tr $\xrightarrow{f_\Delta}$ unlock_tr ⋈ id_tr

Expected morphed spec

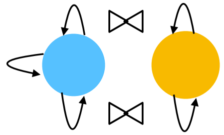


$f\,\hat{}\,\mathrm{lock} : \{\kappa_s = 0\}\ \{\kappa_s = n\}$ @ CSPIN

$f\,\hat{}\,\mathrm{lock} : \{I \wedge f_\Sigma^{-1} P\}\ \{I \wedge f_\Sigma^{-1}\ Q\}$ @CSPIN

$\mathrm{lock} : \{a_s = \varnothing \wedge a_o = h\}$
$\{\exists t.\ a_s = t \Mapsto \mathbf{L} \wedge t > \mathrm{last}\ h\}$
@ SPIN

Expected morphed spec



f^lock : $\{I \wedge f_{\Sigma}^{-1} P\}$ $\{I \wedge f_{\Sigma}^{-1} Q\}$ @CSPIN

lock : $\{ a_s = \varnothing \}$

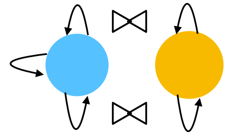
$\{\exists t.\ a_s = t \mapsto \mathbf{L}\ \}$

@ SPIN

$I \triangleq \kappa_s = n\ (\#_{\mathrm{L}}\ a_s)$

65

Expected morphed spec

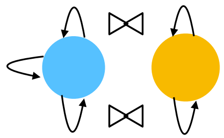


f^lock : $\{ \kappa_s = n\ (\#_{\mathrm{L}}\ a_s) \wedge f_{\Sigma}^{-1}(a_s = \varnothing)\}$

$\{ \kappa_s = n\ (\#_{\mathrm{L}}\ a_s) \wedge f_{\Sigma}^{-1}(\exists t.\ a_s = t \mapsto \mathbf{L}\ )\}$ @CSPIN

66

Expected morphed spec

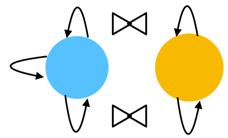


f^lock : $\{ \kappa_s = n\ (\#_{\mathrm{L}}\ a_s) \wedge a_s = \varnothing\}$

$\{ \kappa_s = n\ (\#_{\mathrm{L}}\ a_s) \wedge \exists t.\ a_s = t \mapsto \mathbf{L}\ \}$ @CSPIN

67

Expected morphed spec



f^lock : $\{ \kappa_s = \mathbf{0}\ \}$

$\{ \kappa_s = n\ \}$ @CSPIN

68

Conclusions

1. Type theory very suitable for modelling concurrency
2. New foundations for concurrent reasoning
   - new abstractions for type/code/specs, new rules for proofs
3. Many well-known concepts receive type-inspired modification
   - similar to how structured programming changed programming
4. Separation logic = dependent type theory
   - arises directly from Owicki-Gries approach via types
5. Hoare triples = dependent monads

Important technical ideas

1. Subjective variables (a_s and a_o)
   - local access to global state and global invariants
   - give rise to novel algebra of PCMs (POPL20)
2. Subjective histories
   - separation logic = temporal+spatial reasoning
   - user-level encoding of linearizability
3. Algebra of resources and morphisms
   - type-level ~ Abadi-Lamport refinements
   - novel reasoning rule for morphism application

Implementation

1. Implementation as minimalistic system
   - 9 Hoare-style rules + Coq (shallow embedding)
2. Verified number of benchmark programs
   - locks, stacks, snapshots, flat combiner, graph marking,…

Q&A slides

Differences with separation logic

In separation logic	In our system
$x \mapsto 3 * y \mapsto 42$	$a_s = x \mapsto 3 + y \mapsto 42$
$[x \mapsto 3]^{\text{heap}} * [1]^{\text{ghost}}$	$a_s = (x \mapsto 3,\ 1)$ $\text{fst}\ a_s = x \mapsto 3 \wedge \text{snd}\ a_s = 1$ (leads to theory of PCM functions and relations)
$\exists n.\ [x \mapsto n{+}2]^{\text{heap}} * [n]^{\text{ghost}}$	$\text{fst}\ a_s = x \mapsto (\text{snd}\ a_s + 2)$
$x \mapsto 3 * y \mapsto 42$ ✗	$a_s = x \mapsto 3 \wedge a_o = y \mapsto 42$

Rules

$$[\Gamma].\ \{P\}\ A\ \{Q\}@V = \{e : \mathsf{ST}\ V\ A \mid \forall \Gamma. \forall s \in \Sigma(V).\ P\ s \to \mathsf{vrf}\ e\ Q\ s\}$$

$$\begin{aligned}
\mathsf{fix} &: (T \to T) \to T \\
\mathsf{vrf_post} &: (\forall v\ s.\ J\ s \to Q_1\ v\ s \to Q_2\ v\ s) \to J\ s \to \mathsf{vrf}\ e\ Q_1\ s \to \mathsf{vrf}\ e\ Q_2\ s \\
\mathsf{vrf_ret} &: (Q\ v)^\bullet\ s \to \mathsf{vrf}\ (\mathsf{ret}\ v)\ Q\ s \\
\mathsf{vrf_bnd} &: \mathsf{vrf}\ e_1\ (\lambda x.\ \mathsf{vrf}\ (e_2\ x)\ Q)\ s \to \mathsf{vrf}\ (x \leftarrow e_1; (e_2\ x))\ Q\ s \\
\mathsf{vrf_par} &: ((\mathsf{vrf}\ e_1\ Q_1) * (\mathsf{vrf}\ e_2\ Q_2))\ s \to \mathsf{vrf}\ (e_1 \parallel e_2)\ (\lambda v{:}A_1{\times}A_2.\ (Q_1\ v.1) * (Q_2\ v.2))\ s \\
&\quad \text{where } (P * Q)\ s \mathrel{\widehat{=}} \exists s_1\ s_2.\ s = s_1 * s_2 \wedge P\ s_1 \wedge Q\ s_2 \\
\mathsf{vrf_frame} &: ((\mathsf{vrf}\ e\ Q_1) * Q_2^\bullet)\ s \to \mathsf{vrf}\ e\ (\lambda v.\ (Q_1\ v) * Q_2)\ s \\
\mathsf{vrf_act} &: (\lambda s'.\ \exists s''\ v.\ [a]\ s' = (s'', v) \wedge (Q\ v)^\bullet\ s'')^\bullet\ s \to \mathsf{vrf}\ \langle a \rangle\ Q\ s \\
\mathsf{vrf_morph} &: f\hat{}(\mathsf{vrf}\ e\ Q)\ s_w \to I\ s_w \to \mathsf{vrf}\ (\mathsf{morph}\ f\ e)\ (\lambda v\ s'_w.\ f\hat{}(Q\ v)\ s'_w \wedge I\ s'_w)\ s_w \\
&\quad \text{where } f\hat{}R\ s_w \mathrel{\widehat{=}} \exists s_v.\ s_v = f_\Sigma\ s_w \wedge R\ s_v
\end{aligned}$$

Definitions

Definition 3.9. A ***resource morphism*** $f : V \to W$ consists of two partial functions $f_\Sigma : \Sigma(W) \rightharpoonup \Sigma(V)$ (note the contravariance), and $f_\Delta : \Sigma(W) \rightharpoonup \Delta(V) \rightharpoonup \Delta(W)$, such that:

(1) *(locality of f_Σ)* there exists a function $\phi : M(W) \to M(V)$ such that if $f_\Sigma(s_w \triangleright p) = s_v$, then there exists s'_v such that $s_v = s'_v \triangleright \phi(p)$, and $f_\Sigma(s_w \triangleleft p) = s'_v \triangleleft \phi(p)$.
(2) *(locality of f_Δ)* if $f_\Delta(s_w \triangleright p)(t_v) = t_w$, then $f_\Delta(s_w \triangleleft p)(t_v) = t_w$.
(3) *(other-fixity)* if $a_o(s_w) = a_o(s'_w)$ and $f_\Sigma(s_w)$, $f_\Sigma(s'_w)$ exist, then $a_o(f_\Sigma(s_w)) = a_o(f_\Sigma(s'_w))$.

Definitions

Definition 3.11. Given a morphism $f : V \to W$, an ***f-simulation*** is a predicate I on W-states such that:

(1) if $I\ s_w$, and $s_v = f_\Sigma(s_w)$ exists, and $t_v\ s_v\ s'_v$, then there exist $t_w = f_\Delta\ s_w\ t_v$ and s'_w such that $I\ s'_w$ and $s'_v = f_\Sigma(s'_w)$, and $t_w\ s_w\ s'_w$.
(2) if $I\ s_w$, and $s_v = f_\Sigma(s_w)$ exists, and $s_w \xrightarrow[W]{}^* s'_w$, then $I\ s'_w$, and $s'_v = f_\Sigma(s'_w)$ exists, and $s_v \xrightarrow[V]{}^* s'_v$. Here, the relation $s \xrightarrow[W]{} s'$ denotes that s ***other-steps*** by W to s', i.e., that there exists a transition $t \in \Delta(W)$ such that $t\ s^\top\ s'^\top$. The ***transposition*** $s^\top = (a_o\ s, a_j\ s, a_s\ s)$ swaps the subjective components of s, to obtain the view of *other* threads. The relation $\xrightarrow[W]{}^*$ is the reflexive-transitive closure of $\xrightarrow[W]{}$, allowing for an arbitrary number of steps.

Definitions

Definition B.2. A ***PCM morphism*** $\phi : A \to B$ with a compatibility relation $\perp_\phi$ is a partial function from A to B such that:

(1) $\phi \, \mathbb{1}_A = \mathbb{1}_B$
(2) if $x \perp_\phi y$, then $\phi\, x$, $\phi\, y$ exist, and $\phi\, x \perp_B \phi\, y$, and $\phi\,(x \bullet y) = \phi\, x \bullet \phi\, y$

The morphism ϕ is *total* if $\perp_\phi$ equals $\perp_A$.

Definition B.3. PCM A is a ***sub-PCM*** of a PCM B if there exists a total PCM morphism $\iota : A \to B$ (injection) and a morphism $\rho : B \to A$ (retraction), such that:

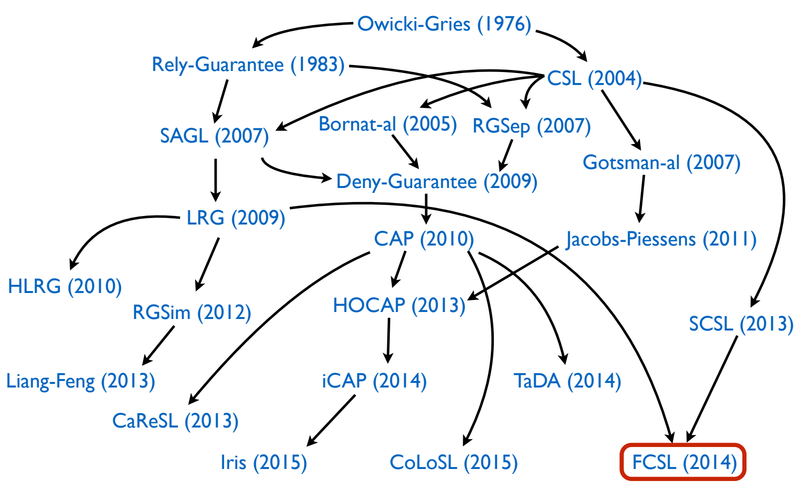
(1) $\rho\,(\iota\, a) = a$
(2) if $b \perp_\rho \mathbb{1}_B$ then $\iota\,(\rho\, b) = b$
(3) if $(\rho\, x) \perp_A (\rho\, y)$ then $x \perp_\rho y$

Definitions

Definition B.5. Let R be an invariant compatibility relation on $M\,(V)$. The ***sub-resource*** V/R is defined with the same type, transitions and erasure as V, but with the PCM and the state space defined as

(1) $M\,(V/R) = M\,(V)/R$
(2) $s \in \Sigma\,(V/R)$ iff $s \in \Sigma\,(V) \wedge (a_s\, s)\; R\; (a_o\, s)$

There is a generic resource morphism $\iota : V \to V/R$ that is inclusion on states and identity on transitions.



79