

# Checking Interference with Fractional Permissions

John Boyland



## Checking Interference with Fractional Permissions

---

- Non-interference: one piece of code doesn't affect the results of another piece of code.
- Two models for checking interference:
  - effects (compare read/write effects of two pieces of code)
  - permissions (check uses of state against permissions)
- Previous work:
  - Using effects analysis requires additionally an alias analysis;
  - Using permissions does not satisfactorily distinguish reads from writes.
- This work: read permission is fraction of write permission.

## Interference: What?

---

- Two computations interfere if the evaluation of one affects the evaluation of the other. (Symmetric definition)
- Example: (L || R)  
 $\{ y = 1; z = x+1; \} \parallel \{ w = y+1; z = w-x; \}$ 
  - L reads x, writes y and z.
  - R reads x and y, writes w and z.
- Kinds of interference:
  - Read/Write (e.g. y in the previous example)
  - Write/Write (e.g. z in the previous example)
  - Reads do not interfere with each other (e.g. x)

## Interference: Why?

---

- Non-interference, lack of a possibility of interference, enables:
  - cleaner formal reasoning;
  - parallelism without race conditions or contention;
  - code reordering.
- We are interested in “safe” algorithms, one which may overstate, but never understate interference.
- This paper gives a provably safe type system (and a sketch of an algorithm).

## Background: Checking Interference with Effects (1 of 3)

---

- Effects are inferred for statements in a bottom-up fashion

E.g.:

L: {  $y = 1; z = x+1; \}$

–  $FX(L) = \{ \text{reads } x, \text{ writes } y, \text{ writes } z \}$

R: {  $w = y+1; z = w-x; \}$

–  $FX(R) = \{ \text{reads } x, \text{ reads } y, \text{ reads } w, \text{ writes } z, \text{ writes } w \}$

- We check whether one set contains a write on a variable read or written by the other:

– writes  $y$  is in  $FX(L)$ , reads  $y$  is in  $FX(R)$ ;

– writes  $z$  is in  $FX(L)$ , writes  $z$  is in  $FX(R)$ .

- See Reynolds [1978], Greenhouse & Boyland [1999].

## Background: Checking Interference with Effects (2 of 3)

---

- Effects analysis gets much harder with pointers:  $L \parallel R$   
 $\{ *p = *q; p = s; \} \parallel \{ r = s; *r = 3 + *q; \}$ 
  - $FX(L) = \text{reads } \{p, q, s, *q\} \text{ writes } \{*p, p\}$
  - $FX(R) = \text{reads } \{s, r, q, *q\} \text{ writes } \{r, *r\}$
  - No apparent interference, but aliasing may cause it.
- Effects analysis must be augmented with an alias analysis [Boyland & Greenhouse 1999]:
  - $\text{MayEqual}(a, b)$ :  $a$  and  $b$  may have the same pointer values;
  - $\text{MayEqual}$  is similar to  $\text{may-alias}$  and to  $\text{PointsTo}$ ;
  - $\text{MayEqual}$  takes into account where  $a$  and  $b$  are evaluated;
  - A  $\text{MayEqual}$  analysis may need to use an effects analysis.

## Background: Checking Interference with Effects (3 of 3)

---

- Some selected rules: (See Greenhouse [2003])

$$\frac{\Gamma \vdash e_1 ! \varphi_1 \quad \Gamma \vdash e_2 ! \varphi_2}{\Gamma \vdash (*e_1 = e_2) ! \varphi_1 \cup \varphi_2 \cup \{\text{writes } *e_1\}}$$

$$\frac{\Gamma \vdash s_1 ! \varphi_1 \quad \Gamma \vdash s_2 ! \varphi_2 \quad \varphi_1 \# \varphi_2}{\Gamma \vdash (s_1 \parallel s_2) ! \varphi_1 \cup \varphi_2}$$

- We need to record uses of expressions in context, to be used by the MayEqual analysis.

## Bckgrnd: Checking Interference with Permissions (1 of 4)

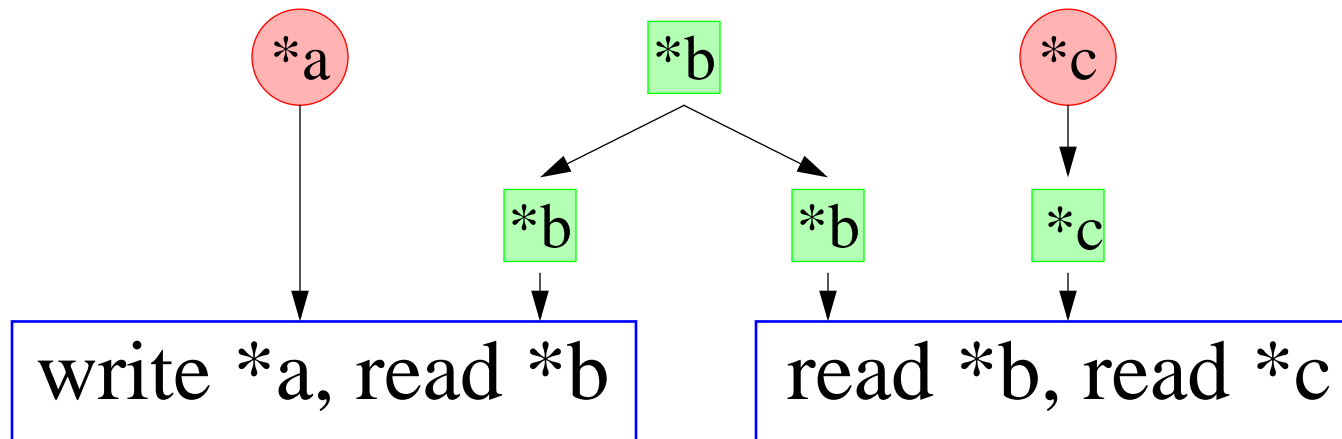
- The context indicates what state may be accessed.  
E.g.:  $\Pi = \{x, y, z\}$  means we may access  $x$ ,  $y$  or  $z$ .
- Pointers are handled using alias types [Smith et al. 2000]  
E.g.  $v: \text{ptr}(\rho)$  means  $v$  is a pointer with value  $\rho$ .  
E.g.  $\Pi = \{\rho\}$  means we may access the memory at  $\rho$ .  
Quantifiers ( $\forall \exists$ ) are necessary for normal pointer types.
- To check interference, partition the permissions  $\Pi$ :
  - $\Pi_1$  is used to check one part;
  - $\Pi_2$  is used to check the other part.Non-interfering by construction.

## Bckgrnd: Checking Interference with Permissions (2 of 4)

### ○ Related Work:

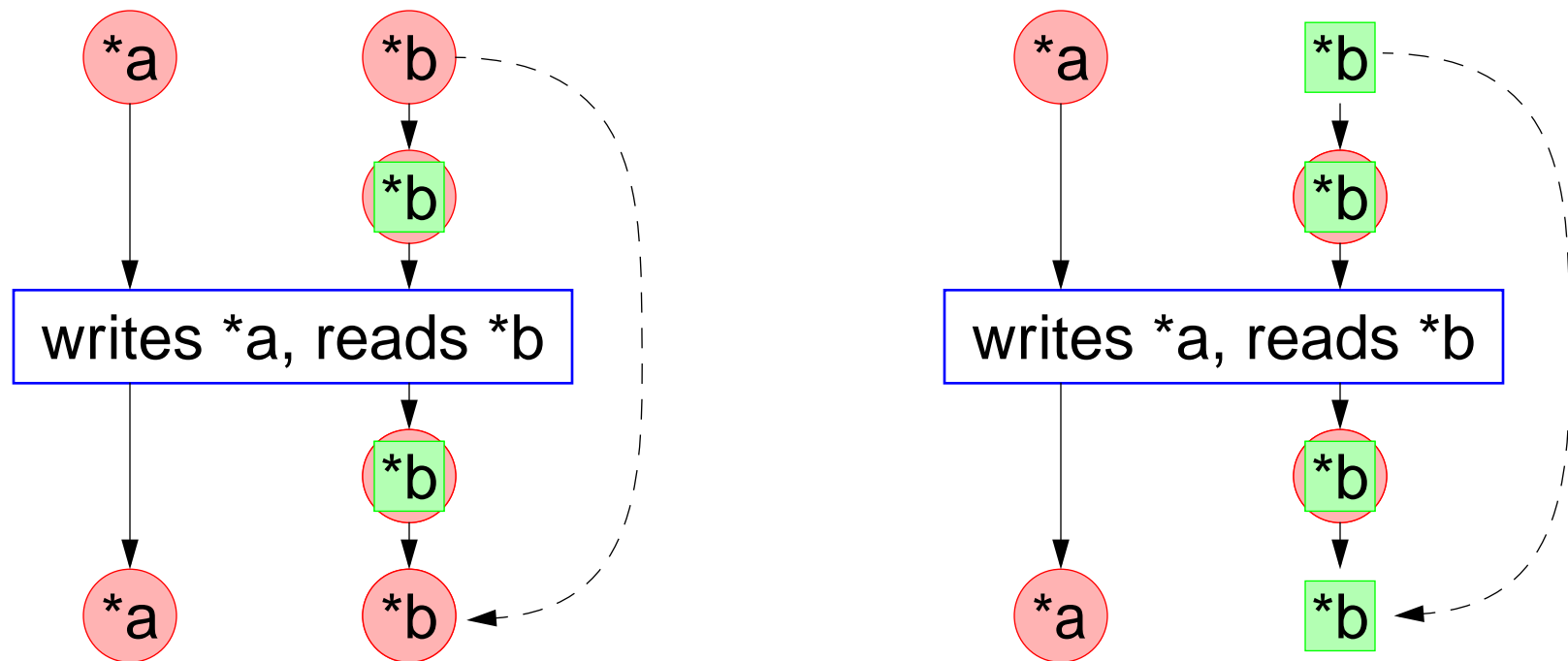
- Locks [Flanagan & Abadi 1999, Boyapati et al 2001/2002]
- Heap Logics [Ishtiaq & O'Hearn 2001, Reynolds 2000/2002]
- CL for checking regions [Walker et al 2000]

### ○ What if we want to distinguish reads from writes?



## Bckgrnd: Checking Interference with Permissions (3 of 4)

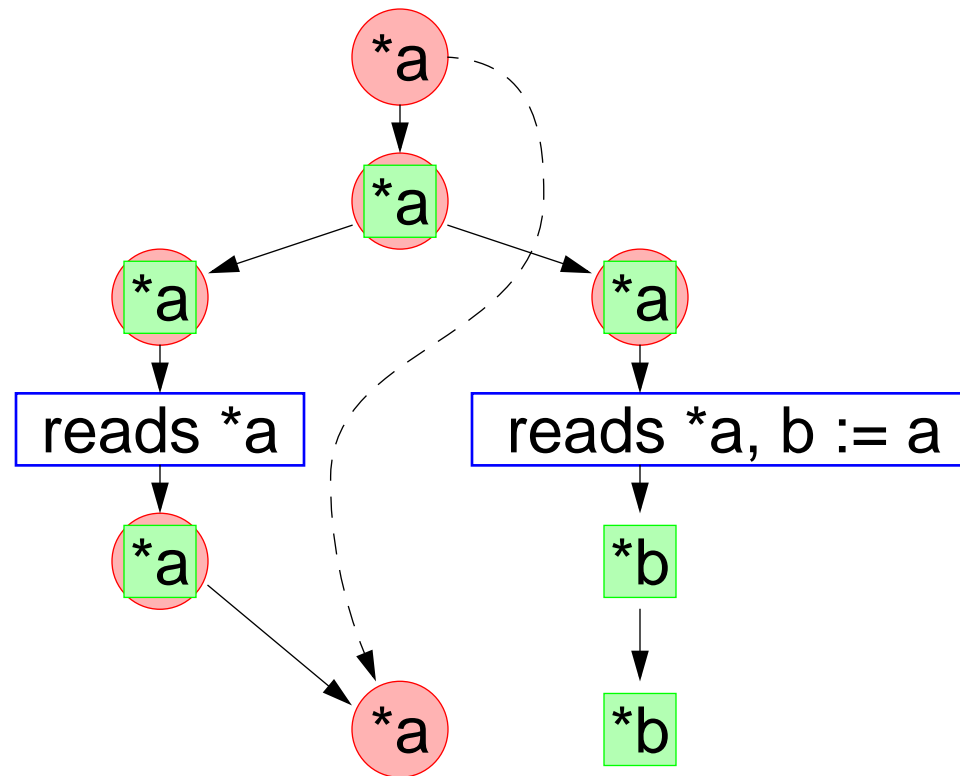
- Use bounded quantification to recover write permission:



(We need input/output model for allocation/de-allocation.)

## Bckgrnd: Checking Interference with Permissions (4 of 4)

- Problem: Duplication of bounded variables is unsound:

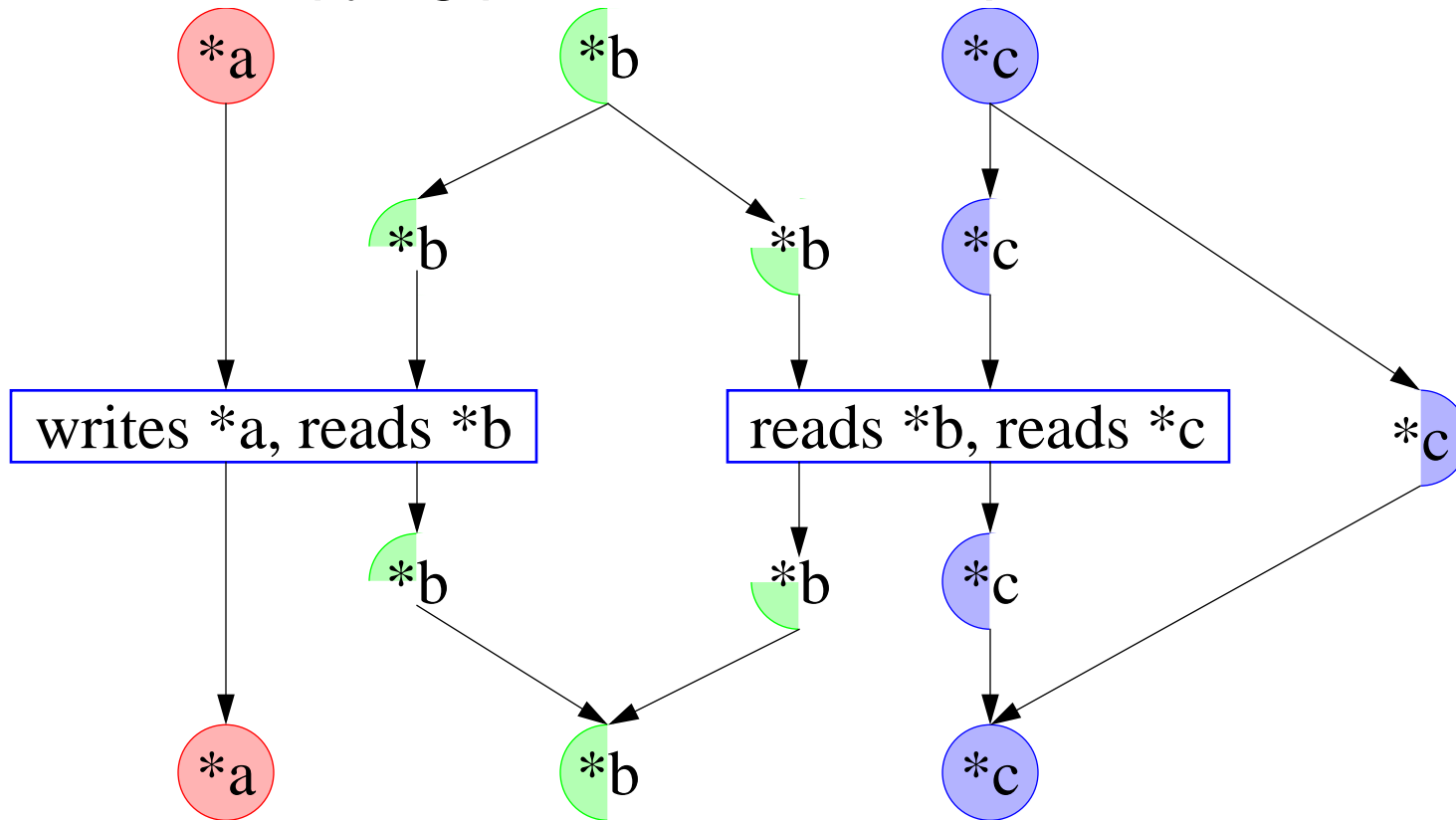


It is not possible to recover write permission after sharing.

## Contribution: Fractional Permissions (1 of 3)

---

- Instead of copying permissions, we *split* them.

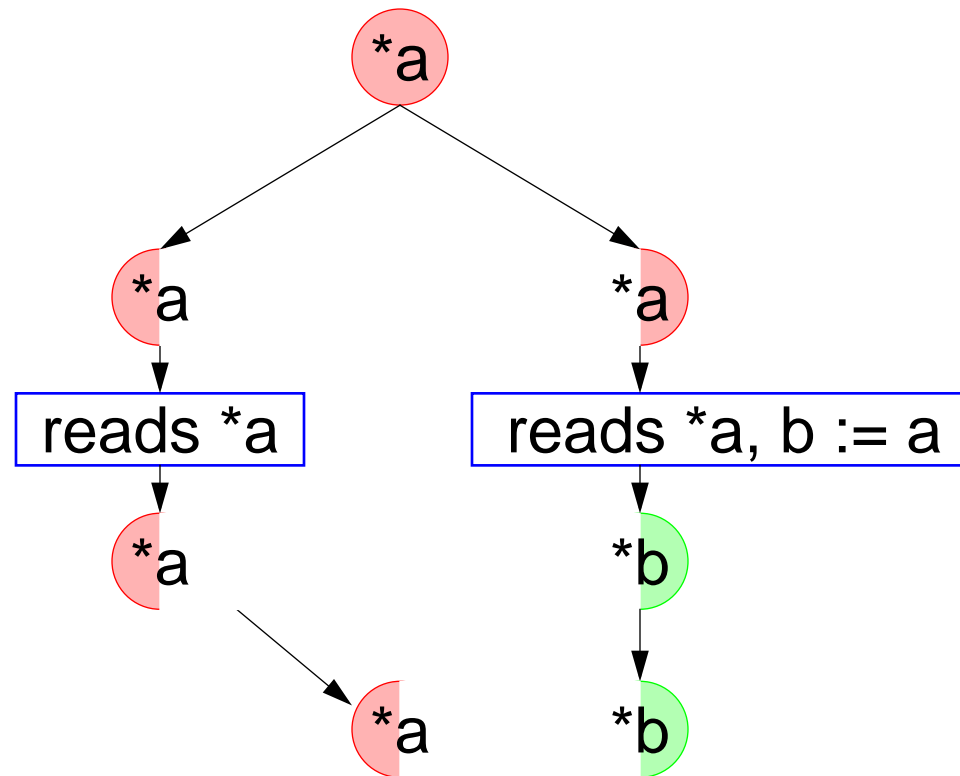


All permissions are treated linearly.

## Contribution: Fractional Permissions (2 of 3)

---

- Linearity preserves soundness:



## Contribution: Fractional Permissions (3 of 3)

---

- We need two kinds of polymorphism:
  - location polymorphism (as in Alias Types);
  - fraction polymorphism (new)
    - $\forall z: z$  stands for a fraction between 0 and 1:  $(0,1)$
- In the paper, we assume fractions are real numbers  $(0,1]$ 
  - Other possibilities are rationals or power-of-two rationals.
  - In general, a monoid (associative multiplication with identity)
    - without any inverses (except  $1 \cdot 1 = 1$ );
    - and a unary operation  $1 - z \neq 1$  for  $z \neq 1$  where  $1 - (1 - z) = z$ .

## Syntax

---

$$s ::= v := \text{new} \mid v := v' \mid *v := e \mid \text{skip}$$
$$\mid s ; s' \mid s \mid \mid s' \mid \text{if } b \text{ then } s \text{ else } s' \mid \text{call } p$$
$$e ::= n \mid e + e \mid *v$$
$$b ::= \text{true} \mid \text{false} \mid v == v' \mid e != 0$$
$$g ::= \{p \rightarrow s, \dots\}$$
$$\mu ::= \{v \rightarrow l, \dots, l \rightarrow n, \dots\}$$

where  $v$  (source variables),  $l$  (locations),  $p$  (procedures) and  $n$  (numbers) are atomic.

## Evaluation: Selected Rules

---

$$\frac{l \notin \text{Dom } \mu_L}{\langle \mu, v := \text{new} \rangle \rightarrow_g \langle \mu[v \mapsto l, l \mapsto 0], \text{skip} \rangle}$$

$$\frac{\langle \mu, e \rangle \rightarrow \langle \mu, e' \rangle}{\langle \mu, *v := e \rangle \rightarrow_g \langle \mu, *v := e' \rangle} \quad \langle \mu, *v := i \rangle \rightarrow_g \langle \mu[\mu v \mapsto i], \text{skip} \rangle$$

$$\frac{\langle \mu, s_1 \rangle \rightarrow_g \langle \mu', s'_1 \rangle}{\langle \mu, s_1 \mid \mid s_2 \rangle \rightarrow_g \langle \mu', s'_1 \mid \mid s_2 \rangle} \quad \frac{\langle \mu, s_2 \rangle \rightarrow_g \langle \mu', s'_2 \rangle}{\langle \mu, s_1 \mid \mid s_2 \rangle \rightarrow_g \langle \mu', s_1 \mid \mid s'_2 \rangle}$$

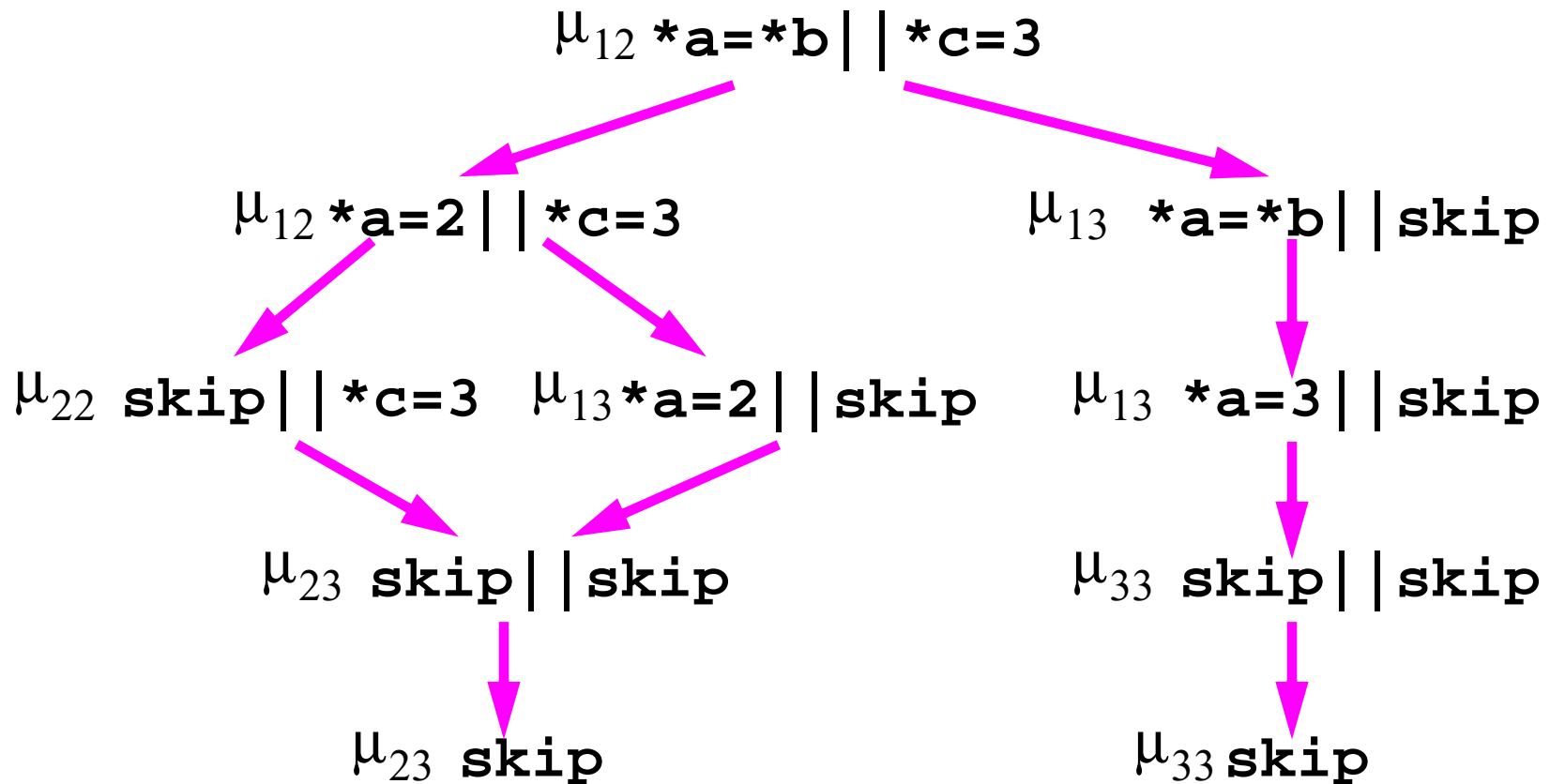
$$\langle \mu, \text{skip} \mid \mid \text{skip} \rangle \rightarrow_g \langle \mu, \text{skip} \rangle \quad \langle \mu, \text{call } p \rangle \rightarrow_g \langle \mu, gp \rangle$$

$$\langle \mu, *v \rangle \rightarrow \langle \mu, \mu(\mu v) \rangle \quad \langle \mu, v == v' \rangle \rightarrow \langle \mu, \mu v = \mu v' \rangle$$

## Example: Nondeterministic Evaluation

---

$$\mu_{xy} = \{ \mathbf{a} \rightarrow l, \mathbf{b} \rightarrow l', \mathbf{c} \rightarrow l', l \rightarrow x, l' \rightarrow y \}$$



## Type Domains

---

$$E ::= \Delta; \Pi$$

$$\Delta ::= \cdot \mid \rho \mid z \mid \Delta, \Delta$$

$$\Pi ::= \cdot \mid \pi \mid \Pi, \Pi$$

$$\pi ::= \xi \beta$$

$$\xi ::= 1 \mid \varepsilon$$

$$\varepsilon ::= z \mid 1 - \varepsilon \mid \varepsilon \varepsilon$$

$$\beta ::= v : \text{ptr}(\rho) \mid \rho$$

where  $z$  and  $\rho$  are fraction and location variables.

## Substructural Rules:

---

$$\cdot, \Pi \equiv \Pi$$

$$\Pi_1, \Pi_2 \equiv \Pi_2, \Pi_1$$

$$\Pi_1, (\Pi_2, \Pi_3) \equiv (\Pi_1, \Pi_2), \Pi_3$$

$$\Delta \vdash \varepsilon \text{ frac}$$

---

$$\Delta; \varepsilon \pi, (1 - \varepsilon) \pi, \Pi \equiv \Delta; \pi, \Pi$$

$$\varepsilon(\varepsilon' \varepsilon'') \equiv (\varepsilon \varepsilon') \varepsilon''$$

$$(1 - (1 - \varepsilon)) \equiv \varepsilon$$

## Permission Types (Selected Rules)

$$E \vdash_{\omega} s \Rightarrow E'$$

---

$$\frac{\rho \text{ fresh}}{\Delta; 1v : ?, \Pi \vdash_{\omega} v := \text{new} \Rightarrow \rho, \Delta; 1\rho, 1v : \text{ptr}(\rho), \Pi}^{\text{NEW}}$$

$$\frac{E = (\Delta; \xi v : \text{ptr}(\rho), 1\rho, \Pi') \quad E \vdash e : \text{Int}}{E \vdash_{\omega} *v := e \Rightarrow E}^{\text{UPDATE}}$$

$$\frac{E \vdash_{\omega} s_1 \Rightarrow E' \quad E' \vdash_{\omega} s_2 \Rightarrow E''}{E \vdash_{\omega} s_1 ; s_2 \Rightarrow E''}^{\text{SEQ}}$$

$$\frac{\Delta; \Pi_1 \vdash_{\omega} s_1 \Rightarrow \Delta'_1; \Pi'_1 \quad \Delta; \Pi_2 \vdash_{\omega} s_2 \Rightarrow \Delta'_2; \Pi'_2}{\Delta; \Pi_1, \Pi_2 \vdash_{\omega} s_1 \parallel s_2 \Rightarrow \Delta'_1 \cup \Delta'_2; \Pi'_1, \Pi'_2}^{\text{PAR}}$$

## Consistency Between Memory and Types (1 of 3)

---

- Permission types
  - indicate aliasing using location variables ( $\rho$ );
  - give permission fractions using fraction variables ( $z$ ).
- We connect types with a memory with a mapping  $\psi$ 
  - maps location variables to locations;
  - maps fraction variables to real numbers (fractions).
- Consistency requires:
  - aliased variables are indeed aliased.
  - sum of permissions for a cell  $\leq 1$ ;

## Consistency Between Memory and Types (2 of 3)

---

$$\psi ::= \{\rho \rightarrow l, \dots, z \rightarrow r, \dots\}$$

$$\psi 1 = 1$$

$$\psi(\varepsilon \varepsilon') = (\psi \varepsilon)(\psi \varepsilon')$$

$$\psi(1 - \varepsilon) = 1 - \psi \varepsilon$$

$$\psi(\xi v : \text{ptr}(\rho)) = [v \mapsto \psi \xi]$$

$$\psi(\xi \rho) = [\psi \rho \mapsto \psi \xi]$$

$$\psi(\Pi_1, \Pi_2) = (\psi \Pi_1) + (\psi \Pi_2)$$

## Consistency Between Memory and Types (3 of 3)

---

$$\frac{\text{Dom } \psi = \Delta \quad \text{Rng } (\psi \Pi) \subseteq [0, 1] \quad \psi; \mu \vdash \Pi \text{ consistent}}{\Delta; \Pi \vdash \mu \text{ ok}}$$

$$\frac{\psi; \mu \vdash \Pi_1 \text{ consistent} \quad \psi; \mu \vdash \Pi_2 \text{ consistent}}{\psi; \mu \vdash \Pi_1, \Pi_2 \text{ consistent}}$$

$$\frac{\psi(\rho) = \mu(v)}{\psi; \mu \vdash \xi v : \text{ptr}(\rho) \text{ consistent}}$$

$$\frac{\psi \rho \in \text{Dom } \mu}{\psi; \mu \vdash \xi \rho \text{ consistent}}$$

## Determinism Theorem

---

- For a statement  $s$ , environment  $E$ , and memory  $\mu$  where
  - $s$  type checks in  $E$ , and
  - $s$  has a terminating evaluation in  $\mu$ , and
  - $\mu$  is consistent with  $E$ .

Then:

- all evaluations of  $s$  terminate in the same number of steps,
  - all evaluations end up with the “same” memory (up to isomorphism).
- In particular: no race conditions in parallel code.

## Permission Checking Algorithm

---

- The type system is not algorithmic: main difficulty is PAR
    - It requires us to “guess” how to split permissions;
    - Substructural rules can be applied at any time.
  - Intuition: For PAR, put all permissions in a “bag.”
    - While checking first half, take out permissions as needed:
      - if only read permission is needed, split off a piece;
      - if write permission is needed, bring the whole permission out.
    - Then check second part using permissions still in “bag.”
- Further work: formally define the algorithm and implement.

## Extensions and Further Work

---

- Records and Recursive Types
  - Permissions for field access;
  - Existential types.
- Memory management:
  - handling delete: forbid the use of dangling pointers.
  - handling garbage collection:
    - a new memory isomorphism;
    - collect permissions too.
- Connecting with Adoption/Ownership/Uniqueness

## Conclusions

---

- Permission types express in a single formalism
  - Aliasing;
  - Effects.
- Permission types enable interference checking.
- Fractional permissions:
  - permit read/read parallelism,
  - without loss of linearity.

## Procedure Types

---

$$\forall \Delta. (\Pi \rightarrow \exists \Delta'. \Pi')$$

Permissions that are not required by the procedure are passed through unchanged.

Example: a procedure  $p$  such that  $gp = x := y$  has (among other types) the type

$$\forall \rho, \rho', z. \left( \begin{array}{l} 1x : \text{ptr}(\rho), zY : \text{ptr}(\rho') \rightarrow \\ \exists \rho''. 1x : \text{ptr}(\rho''), zY : \text{ptr}(\rho') \end{array} \right)$$