

A Relational Model for Confined Separation Logic

J.N. Oliveira¹

(joint work with Shuling Wang² and Luís Barbosa¹)

¹FAST Group, U. Minho, Braga, Portugal

²Peking Univ., Beijing, China

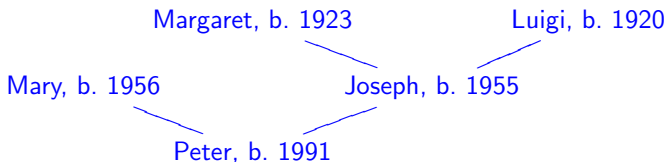
CIC'07 Meeting
October 2007
CWI, Amsterdam

Motivation

Consider Haskell datatype

```
data PTree = Node {  
    name    :: String ,  
    birth   :: Int      ,  
    mother  :: Maybe PTree,  
    father  :: Maybe PTree  
}
```

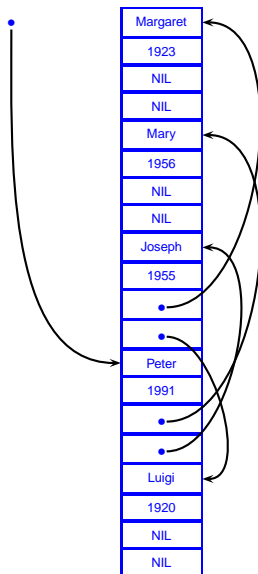
able to model family trees such as eg.



What if the same model is to be built in C/C++ ?

Motivation

The model becomes “more **concrete**” as we go down to such programming level;



Trees get converted to **pointer** structures stored in dynamic **heaps**.

A glimpse at the heap/pointer level

Still in Haskell:

- Heaps *shaped* for PTrees:

```
data Heap a k = Heap [(k,(a,Maybe k, Maybe k))] k
```

- Function which represents PTrees in terms of such heaps:

```
r (Node n b m f) = let x = fmap r m  
                    y = fmap r f  
                    in merge (n,b) x y
```

- This is a *fold* over PTrees which builds the heap for a tree by joining the heaps of the subtrees, where ...

A glimpse at the heap/pointer level

... merge performs **separated union** of heaps

```
merge a Nothing Nothing =
  Heap ([ 1 |-> (a, Nothing, Nothing) ]) 1
merge a (Just x) (Just y) =
  Heap ([ 1 |-> (a, Just k1, Just k2) ] ++ h1 ++ h2) 1
    where (Heap h1 k1) = bmap id even_ x
          (Heap h2 k2) = bmap id odd_ y

....
....

even_ k = 2*k
odd_  k = 2*k+1
```

Note how *even_* and *odd_* ensure that heaps joined have disjoint domains. (More details in [4].)

Data “heapification”

Source

```
t= Node {name = "Peter", birth = 1991,  
        mother = Just (Node {  
            name = "Mary", birth = 1956,  
            mother = Nothing,  
            father = Just (Node {name = "Jules",  
                                birth = 1917, mother = N  
            ..... }}}
```

“heapifies” into:

```
r t = Heap [(1,(("Peter",1991),Just 2,Just 3)),  
            (2,(("Mary",1956),Nothing,Just 6)),  
            (6,(("Jules",1917),Nothing,Nothing)),  
            (3,(("Joseph",1955),Just 5,Just 7)),  
            (5,(("Margaret",1923),Nothing,Nothing)),  
            (7,(("Luigi",1920),Nothing,Nothing))]
```

What about the way back?

- The way back (abstraction) is a **partial** unfold

```
f (Heap h k) = let Just (a,x,y) = lookup k h
               in  Node (fst a)(snd a)
                   (fmap (f . Heap h) x)
                   (fmap (f . Heap h) y)
```

because of pointer **dereferencing** is not a total operation.

- More about this in my GTTSE'07 tutorial [4]
- Use of **separated union** in heap/pointer-level PTree example suggests **separation logic** developed by Peter O'Hearn, John Reynolds [5] and others
- Interest in **separation logic** spiced up by visit of Shuling Wang earlier this year

Aims

We decided to

- Study the application of separation logic to pointer/heap data **refinement**,

which entailed

- Studying the **semantics** of separation logic (in particular of the **confined** variant proposed by Wang Shuling and Qiu Zongyan [7])

which entailed

- Applying the **PF-transform** to confined separation logic

Terminology

Mac **Aa** dictionary:

- **reference** — “the action of mentioning or alluding to something”
- **referent** — “the thing that a word or phrase denotes or stands for”

Thus

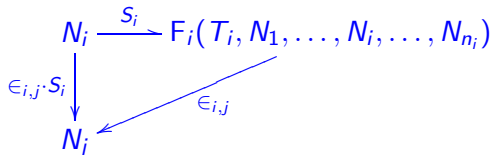
- references are **names** and referents are **things** (aka **objects**).

Problems:

- **aliasing** — “Eric Blair, alias George Orwell”: two names for the same thing
- **referential integrity** — “Eric Blair : unknown author, sorry”

Name spaces

In a diagram:



where

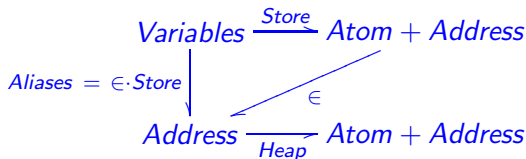
- S_i : relation between **names** and **things** (reference \mapsto referent) in **name** space of type i
(F_i describes the structure of i -**things** and T_i embodies other attributes of such **things**)
- $\in_{i,j}$: relation which spots **names** of type j in **things** of type i
- $\in_{i,j} \cdot S_i$: **name-to-name** relation (*dependence graph*) between types i and j .

Name space ubiquity

Name spaces are everywhere:

- **Databases** (foreign/primary keys, entities)
- **Grammars** (nonterminals, productions)
- **Objects** (identities, classes)
- Caches and **heaps** (memory cells, pointers)

Name spaces in separation logic:



that is, a state is a *Store* (as in Hoare logic) paired with a *Heap*.

Separated union

It is a partial function of type

$$\text{Heap} \xleftarrow{*} \text{Heap} \times \text{Heap}$$

which joins two heaps

$$H * (H_1, H_2) \stackrel{\text{def}}{=} (H_1 \parallel H_2) \wedge (H = H_1 \cup H_2) \quad (1)$$

in case they are disjoint:

$$H_1 \parallel H_2 \stackrel{\text{def}}{=} \neg \langle \exists b, a, k :: b H_1 k \wedge a H_2 k \rangle$$

NB: $t H k$ means “thing t is the referent of k in heap H ”

Let's spruce up notation

Thanks to the PF (“point free”) transform $:-$):

$$\begin{aligned}
 & \neg \langle \exists b, a, k :: b H_1 k \wedge a H_2 k \rangle \\
 \equiv & \quad \{ \exists\text{-nesting (Eindhoven quantifier calculus)} \} \\
 & \neg \langle \exists b, a :: \langle \exists k :: b H_1 k \wedge a H_2 k \rangle \rangle \\
 \equiv & \quad \{ \text{relational converse: } b R^\circ a \text{ the same as } a R b \} \\
 & \neg \langle \exists b, a :: \langle \exists k :: b H_1 k \wedge k H_2^\circ a \rangle \rangle \\
 \equiv & \quad \{ \text{introduce relational composition} \} \\
 & \neg \langle \exists b, a :: b (H_1 \cdot H_2^\circ) a \rangle \\
 \equiv & \quad \{ \text{de Morgan ; negation} \} \\
 & \langle \forall b, a :: b (H_1 \cdot H_2^\circ) a \Rightarrow \text{FALSE} \rangle
 \end{aligned}$$

Let's spruce up notation

Thanks to the PF (“point free”) transform $:-$:

$$\begin{aligned}
 & \neg \langle \exists b, a, k :: b H_1 k \wedge a H_2 k \rangle \\
 \equiv & \quad \{ \exists\text{-nesting (Eindhoven quantifier calculus)} \} \\
 & \neg \langle \exists b, a :: \langle \exists k :: b H_1 k \wedge a H_2 k \rangle \rangle \\
 \equiv & \quad \{ \text{relational converse: } b R^\circ a \text{ the same as } a R b \} \\
 & \neg \langle \exists b, a :: \langle \exists k :: b H_1 k \wedge k H_2^\circ a \rangle \rangle \\
 \equiv & \quad \{ \text{introduce relational composition} \} \\
 & \neg \langle \exists b, a :: b (H_1 \cdot H_2^\circ) a \rangle \\
 \equiv & \quad \{ \text{de Morgan ; negation} \} \\
 & \langle \forall b, a :: b (H_1 \cdot H_2^\circ) a \Rightarrow \text{FALSE} \rangle
 \end{aligned}$$

Let's spruce up notation

$$\equiv \quad \{ \text{ empty relation: } b \perp a \text{ is always false} \}$$

$$\langle \forall b, a :: b(H_1 \cdot H_2^\circ)a \Rightarrow b \perp a \rangle$$

$$\equiv \quad \{ \text{ drop points } a, b \}$$

$$H_1 \cdot H_2^\circ \subseteq \perp$$

So we can redefine

$$H_1 \parallel H_2 \stackrel{\text{def}}{=} H_1 \cdot H_2^\circ \subseteq \perp \quad (2)$$

cf diagram:

$$\begin{array}{ccc}
 K & \xrightarrow{H_1} & F(A, K) \\
 \uparrow id & & \uparrow \perp \\
 K & \xleftarrow{H_2^\circ} & F(A, K)
 \end{array}$$

Let's spruce up notation

$$\equiv \quad \{ \text{empty relation: } b \perp a \text{ is always false} \}$$

$$\langle \forall b, a :: b(H_1 \cdot H_2^\circ)a \Rightarrow b \perp a \rangle$$

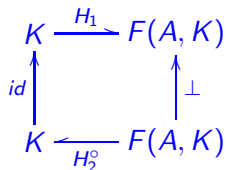
$$\equiv \quad \{ \text{drop points } a, b \}$$

$$H_1 \cdot H_2^\circ \subseteq \perp$$

So we can redefine

$$H_1 \parallel H_2 \stackrel{\text{def}}{=} H_1 \cdot H_2^\circ \subseteq \perp \quad (2)$$

cf diagram:



Summary of PF-transform

ϕ	$PF\ \phi$
$\langle \exists a :: b R a \wedge a S c \rangle$	$b(R \cdot S)c$
$\langle \forall a, b :: b R a \Rightarrow b S a \rangle$	$R \subseteq S$
$\langle \forall a :: a R a \rangle$	$id \subseteq R$
$\langle \forall x :: x R b \Rightarrow x S a \rangle$	$b(R \setminus S)a$
$\langle \forall c :: b R c \Rightarrow a S c \rangle$	$a(S / R)b$
$b R a \wedge c S a$	$(b, c)\langle R, S \rangle a$
$b R a \wedge d S c$	$(b, d)(R \times S)(a, c)$
$b R a \wedge b S a$	$b(R \cap S)a$
$b R a \vee b S a$	$b(R \cup S)a$
$(f\ b) R (g\ a)$	$b(f^\circ \cdot R \cdot g)a$
TRUE	$b \top a$
FALSE	$b \perp a$

(3)

where R , S , id are binary relations.

Binary Relations

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation to B (target) from A (source).

Points

$b R a$ — “ R relates b to a ”, that is, $(b, a) \in R$.

Identity of composition

id such that $R \cdot id = id \cdot R = R$

Converse

Converse of R — R° such that $a(R^\circ)b$ iff $b R a$.

Ordering

$R \subseteq S$ — the obvious “ R is at most S ” inclusion ordering.

Binary Relations

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation to B (target) from A (source).

Points

$b R a$ — “ R relates b to a ”, that is, $(b, a) \in R$.

Identity of composition

id such that $R \cdot id = id \cdot R = R$

Converse

Converse of R — R° such that $a(R^\circ)b$ iff $b R a$.

Ordering

$R \subseteq S$ — the obvious “ R is at most S ” inclusion ordering.

Binary Relations

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation to B (target) from A (source).

Points

$b R a$ — “ R relates b to a ”, that is, $(b, a) \in R$.

Identity of composition

id such that $R \cdot id = id \cdot R = R$

Converse

Converse of R — R° such that $a(R^\circ)b$ iff $b R a$.

Ordering

$R \subseteq S$ — the obvious “ R is at most S ” inclusion ordering.

Binary Relations

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation to B (target) from A (source).

Points

$b R a$ — “ R relates b to a ”, that is, $(b, a) \in R$.

Identity of composition

id such that $R \cdot id = id \cdot R = R$

Converse

Converse of R — R° such that $a(R^\circ)b$ iff $b R a$.

Ordering

$R \subseteq S$ — the obvious “ R is at most S ” inclusion ordering.

Binary Relations

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation to B (target) from A (source).

Points

$b R a$ — “ R relates b to a ”, that is, $(b, a) \in R$.

Identity of composition

id such that $R \cdot id = id \cdot R = R$

Converse

Converse of R — R° such that $a(R^\circ)b$ iff $b R a$.

Ordering

$R \subseteq S$ — the obvious “ R is at most S ” inclusion ordering.

Standard separation logic

Syntax:

$$\begin{array}{lcl} p & ::= & \dots \\ & | & \mathbf{emp} \quad /* \text{ heap is empty } */ \\ & | & e \mapsto e \quad /* \text{ singleton heap } */ \\ & | & p * p \quad /* \text{ separating conjunction } */ \\ & | & p \multimap p \quad /* \text{ separating implication } */ \end{array}$$

Semantics:

$$\llbracket e \rrbracket : \text{Store} \rightarrow \text{Atom} + \text{Address}$$
$$\llbracket p \rrbracket : (\text{Heap} \times \text{Store}) \rightarrow \mathbb{B}$$

Semantics of separating connectives

Separating conjunction:

$$\begin{aligned} \llbracket p * q \rrbracket(H, S) &\stackrel{\text{def}}{=} \\ &\langle \exists H_0, H_1 :: H * (H_0, H_1) \wedge \llbracket p \rrbracket(H_0, S) \wedge \llbracket q \rrbracket(H_1, S) \rangle \end{aligned}$$

Separating implication:

$$\begin{aligned} \llbracket p \multimap q \rrbracket(H, S) &\stackrel{\text{def}}{=} \\ &\langle \forall H_0 : H_0 \parallel H : \llbracket p \rrbracket(H_0, S) \Rightarrow \llbracket q \rrbracket(H_0 \cup H, S) \rangle \end{aligned}$$

Emptiness:

$$\llbracket \text{emp} \rrbracket(H, S) \stackrel{\text{def}}{=} H = \perp$$

etc.

Standard inference rules

- Our attention was driven to

[There are] two further rules capturing the adjunctive relationship between separating conjunction and separating implication:

$$\frac{p_1 * p_2 \Rightarrow p_3}{p_1 \Rightarrow (p_2 \multimap p_3)} \qquad \frac{p_1 \Rightarrow (p_2 \multimap p_3)}{p_1 * p_2 \Rightarrow p_3}$$

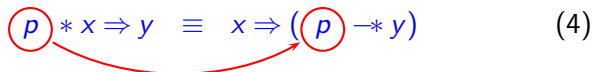
(quoting [5])

- These rules are (everywhere!) stated without proof wrt. the given semantics.

Checking inference rules

Steps in checking these rules:

- Put them together thus making Galois connection apparent:

$$(p) * x \Rightarrow y \equiv x \Rightarrow ((p) -* y) \quad (4)$$


(We like this kind of approach because it reminds us of the “al-djabr” rules

$$x - (z) \leq y \equiv x \leq y + (z)$$


familiar from school algebra.)

- Define semantics at PF-level so as to take advantage of relational calculus

PF-relational semantics for separation logic

We define

- assertion semantics as a **relation** between stores and heaps,

$$\text{Store} \xleftarrow{[[p]]} \text{Heap}$$

a natural decision since every binary predicate is nothing but a relation :-)

- the **preorder** on assertions induced by these semantics

$$p \rightarrow q \stackrel{\text{def}}{=} [[p]] \subseteq [[q]] \quad (5)$$

so that it can be distinguished from standard logic implication \Rightarrow .

PF-relational semantics for separation logic

PF-transform of Reynolds original definition of separating conjunction follows:

$$\llbracket p * q \rrbracket(H, S) \stackrel{\text{def}}{=} \langle \exists H_0, H_1 :: H * (H_0, H_1) \wedge \llbracket p \rrbracket(H_0, S) \wedge \llbracket q \rrbracket(H_1, S) \rangle$$

becomes

$$\llbracket p * q \rrbracket \stackrel{\text{def}}{=} (*) \cdot \langle \llbracket p \rrbracket, \llbracket q \rrbracket \rangle \quad (6)$$

just by recalling two rules of the PF-transform (3): composition

$$b(R \cdot S)c \equiv \langle \exists a :: bRa \wedge aSc \rangle \quad (7)$$

and *splitting*

$$(a, b) \langle R, S \rangle c \equiv a R c \wedge b S c \quad (8)$$

PF-relational semantics for separation logic

PF-transform of Reynolds original definition of separating conjunction follows:

$$\llbracket p * q \rrbracket(H, S) \stackrel{\text{def}}{=} \langle \exists H_0, H_1 :: H * (H_0, H_1) \wedge \llbracket p \rrbracket(H_0, S) \wedge \llbracket q \rrbracket(H_1, S) \rangle$$

becomes

$$\llbracket p * q \rrbracket \stackrel{\text{def}}{=} (*) \cdot \langle \llbracket p \rrbracket, \llbracket q \rrbracket \rangle \quad (6)$$

just by recalling two rules of the PF-transform (3): composition

$$b(R \cdot S)c \equiv \langle \exists a :: bRa \wedge aSc \rangle \quad (7)$$

and *splitting*

$$(a, b) \langle R, S \rangle c \equiv a R c \wedge b S c \quad (8)$$

Calculation of \multimap

Then we re-write (4) into what we should have written in the first place

$$(p * x) \rightarrow y \equiv x \rightarrow (p \multimap y) \quad (9)$$

which

- we regard as an **equation**
- where we know everything apart from \multimap (the **unknown**, the “*cousa*”), which we want to calculate:

$$\begin{aligned}
 & (p * x) \rightarrow y \\
 \equiv & \quad \{ \text{semantic preorder (5)} \} \\
 & \llbracket p * x \rrbracket \subseteq \llbracket y \rrbracket \\
 \equiv & \quad \{ \text{PF-definition (6)} \} \\
 & (*) \cdot \langle \llbracket p \rrbracket, \llbracket x \rrbracket \rangle \subseteq \llbracket y \rrbracket
 \end{aligned}$$

Stop and think

GCs are like *mushrooms*, the stereotype of rapid growth:

- never ignore the ones you know already, eg.

$$R \cdot X \subseteq S \equiv X \subseteq R \setminus S \quad (10)$$

where

$$b(R \setminus Y)a \equiv \langle \forall c : c R b : c Y a \rangle \quad (11)$$

- nor the ones you can derive yourself, eg.

$$\langle R, S \rangle \subseteq X \equiv S \subseteq R \triangleright X \quad (12)$$

where

$$b(R \triangleright S)a \equiv \langle \forall c : c R a : (c, b) S a \rangle \quad (13)$$

(a “kind of implication”).

Calculation of \multimap (cntd)

We proceed:

$$\begin{aligned}
 & (*) \cdot \langle \llbracket p \rrbracket, \llbracket x \rrbracket \rangle \subseteq \llbracket y \rrbracket \\
 \equiv & \quad \{ \text{the two GCs above in a row} \} \\
 & \llbracket x \rrbracket \subseteq \llbracket p \rrbracket \triangleright ((*) \setminus \llbracket y \rrbracket) \\
 \equiv & \quad \{ \text{introduce } p \multimap y \text{ such that } \llbracket p \multimap y \rrbracket = \llbracket p \rrbracket \triangleright ((*) \setminus \llbracket y \rrbracket) \} \\
 & \llbracket x \rrbracket \subseteq \llbracket p \multimap y \rrbracket \\
 \equiv & \quad \{ \text{semantic preorder (5)} \} \\
 & x \rightarrow (p \multimap y)
 \end{aligned}$$

We are left with the meaning of $p \triangleright ((*) \setminus \llbracket y \rrbracket)$, see next slides

Calculation of \multimap (cntd)

$$\begin{aligned}
 & H\llbracket p \multimap y \rrbracket S \\
 \equiv & \quad \{ \text{above} \} \\
 & H(\llbracket p \rrbracket \triangleright ((*) \setminus \llbracket y \rrbracket)) S \\
 \equiv & \quad \{ \triangleright \text{pointwise (13)} \} \\
 & \langle \forall H_0 : H_0\llbracket p \rrbracket S : (H_0, H)((*) \setminus \llbracket y \rrbracket) S \rangle \\
 \equiv & \quad \{ \text{left division (11) pointwise} \} \\
 & \langle \forall H_0 : H_0\llbracket p \rrbracket S : \langle \forall H_1 : H_1 * (H_0, H) : H_1\llbracket y \rrbracket S \rangle \rangle \\
 \equiv & \quad \{ \text{nesting: (4.21) of [1]} \}
 \end{aligned}$$

Calculation of \multimap (cntd)

$$\begin{aligned}
 & \langle \forall H_0, H_1 : H_0 \llbracket p \rrbracket S \wedge H_1 * (H_0, H) : H_1 \llbracket y \rrbracket S \rangle \\
 \equiv & \quad \{ \text{separated union (1)} \} \\
 & \langle \forall H_0, H_1 : H_0 \llbracket p \rrbracket S \wedge H_0 \parallel H \wedge H_1 = H_0 \cup H : H_1 \llbracket y \rrbracket S \rangle \\
 \equiv & \quad \{ \text{one-point: (4.24) of [1]} \} \\
 & \langle \forall H_0 : H_0 \llbracket p \rrbracket S \wedge H_0 \parallel H : (H_0 \cup H) \llbracket y \rrbracket S \rangle \\
 \equiv & \quad \{ \text{trading: (4.28) of [1]} \} \\
 & \langle \forall H_0 : H_0 \parallel H : H_0 \llbracket p \rrbracket S \Rightarrow (H_0 \cup H) \llbracket y \rrbracket S \rangle
 \end{aligned}$$

As expected, we reach the definition **postulated** by J. Reynolds [5]

Benefits of $((*) , -*)$ connection

The following are immediate consequences of the connection, where \leftrightarrow denotes the antisymmetric closure of \rightarrow :

$$p * (x_1 \vee x_2) \leftrightarrow (p * x_1) \vee (p * x_2) \quad (14)$$

$$(x_1 \vee x_2) * p \leftrightarrow (x_1 * p) \vee (x_2 * p) \quad (15)$$

$$p -* (x_1 \wedge x_2) \leftrightarrow (p -* x_1) \wedge (p -* x_2) \quad (16)$$

plus monotonicity, cancellations,

$$x \rightarrow (p -* (p * x)) \quad (17)$$

$$p * (p -* y) \rightarrow y \quad (18)$$

etc. and some others, usually not mentioned in the literature

$$\mathbf{emp} \rightarrow p -* p \quad (19)$$

$$p * x \leftrightarrow p * (p -* (p * x)) \quad (20)$$

$$p -* x \leftrightarrow p -* (p * (p -* x)) \quad (21)$$

Moving on to the main objective

A problem

Aliasing — *In object-oriented programming it is difficult to control the spread and sharing of object references. This pervasive aliasing makes it nearly impossible to know accurately who owns a given object, that is to say, which other objects have references to it. [2]*

A proposal

Confinement — *A type is said to be confined in a domain if and only if all references to instances of that type originate from objects of the domain. [2]*

A question

- how do we incorporate *confinement* into separation logic?

Moving on to the main objective

A problem

Aliasing — *In object-oriented programming it is difficult to control the spread and sharing of object references. This pervasive aliasing makes it nearly impossible to know accurately who owns a given object, that is to say, which other objects have references to it. [2]*

A proposal

Confinement — *A type is said to be confined in a domain if and only if all references to instances of that type originate from objects of the domain. [2]*

A question

- how do we incorporate *confinement* into separation logic?

Enriching separation logic

The essence of separation logic being “separation” itself, Wang and Qiu [7] propose that the notion of heap disjointness be sophisticated in three directions:

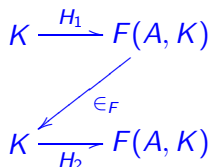
- **notIn** variant — heaps disjoint and such that no references of the first point to the other
- **In** variant — heaps disjoint and such that all references in the first do point into the other
- **inBoth** variant — heaps disjoint and such that all references in the first are confined to both.

Confined disjointness — **notIn**

No outgoing reference in heap H_1 goes into separate H_2 :

$$\text{notIn}(H_1, H_2) \stackrel{\text{def}}{=} H_1 \parallel H_2 \wedge H_2 \cdot \epsilon_F \cdot H_1 \subseteq \perp$$

In a diagram: path



is empty, that is (back to points)

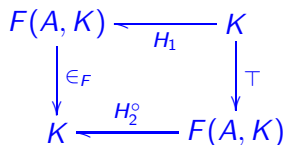
$$\neg \langle \exists k, k' : k \in \delta H_1 \wedge k' \in \delta H_2 : k' \in_F (H_1 k) \rangle$$

Confined disjointness — In

All outgoing references in H_1 dangle because they all go into separated H_2 :

$$\text{In}(H_1, H_2) \stackrel{\text{def}}{=} H_1 \parallel H_2 \wedge \in_F \cdot H_1 \subseteq H_2^\circ \cdot \top$$

In a diagram: dependency graph $\in_F \cdot H_1$



can only lead to references in the domain of H_2 (\top denotes the everywhere true predicate)

Confined disjointness — **inBoth**

H_1 and H_2 are disjoint and all outgoing references in H_1 are confined to either H_2 or itself:

$$inBoth(H_1, H_2) \stackrel{\text{def}}{=} H_1 \parallel H_2 \wedge \underbrace{\in_F \cdot H_1 \subseteq (H_1 \cup H_2)^\circ \cdot \top}_{\alpha}$$

Comments:

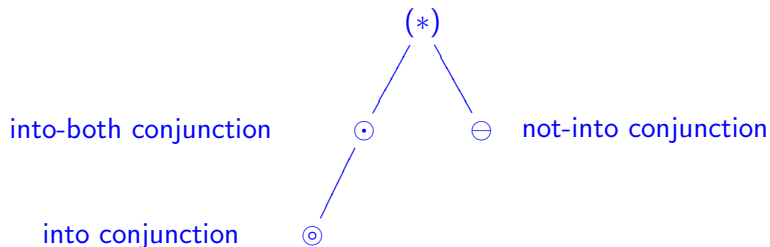
- Note how clumsy α becomes once mapped back to point-level:

$$\langle \forall k : \langle \exists k' : k' \in \delta H_1 : k \in_F (H_1 \ k') \rangle : k \in \delta H_1 \vee k \in \delta H_2 \rangle$$

- Clearly, $in \Rightarrow inBoth$

Confined separation logic

Three new variants of separating conjunction:



able to express confinement subtleties.

Confined separation logic

- *Left-not-into-right* conjunction:

$$\llbracket p \ominus q \rrbracket \stackrel{\text{def}}{=} (*) \cdot \Phi_{\text{notIn}} \cdot \langle \llbracket p \rrbracket, \llbracket q \rrbracket \rangle \quad (22)$$

- *Left-into-right* conjunction:

$$\llbracket p \odot q \rrbracket \stackrel{\text{def}}{=} (*) \cdot \Phi_{\text{In}} \cdot \langle \llbracket p \rrbracket, \llbracket q \rrbracket \rangle \quad (23)$$

- *Left-into-both* conjunction:

$$\llbracket p \odot q \rrbracket \stackrel{\text{def}}{=} (*) \cdot \Phi_{\text{inBoth}} \cdot \langle \llbracket p \rrbracket, \llbracket q \rrbracket \rangle \quad (24)$$

NB: relation Φ_p denotes the PF-transform of unary predicate p ,
see next slide

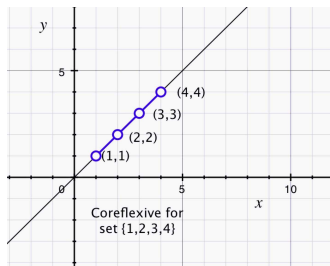
Background: PF-transforms of unary predicates

- There are several ways to encode **unary** predicates as binary relations in the PF-transform.
- A popular one is to use fragments of *id* (coreflexives) :

$$R = \Phi_p \equiv (y R x \equiv (p x) \wedge x = y)$$

eg. (in the natural numbers)

$$\llbracket 1 \leq x \leq 4 \rrbracket =$$



What about confined implication(s)?

Very easy:

- Just stick the relevant coreflexive (eg. Φ_{In}) to separate union $(*)$ and move this lot around as before
- Once points are back into formulæ, you get separate implication for each case, for instance:

$$H \llbracket p \multimap y \rrbracket S \stackrel{\text{def}}{=} \langle \forall H_0 : H_0 \parallel H \wedge In(H_0, H) : H_0 \llbracket p \rrbracket S \Rightarrow (H_0 \cup H) \llbracket y \rrbracket S \rangle$$

together with all the properties implicit

What about confined implication(s)?

Very easy:

- Just stick the relevant coreflexive (eg. Φ_{In}) to separate union $(*)$ and move this lot around as before
- Once points are back into formulæ, you get separate implication for each case, for instance:

$$H \llbracket p \multimap y \rrbracket S \stackrel{\text{def}}{=} \langle \forall H_0 : H_0 \parallel H \wedge In(H_0, H) : H_0 \llbracket p \rrbracket S \Rightarrow (H_0 \cup H) \llbracket y \rrbracket S \rangle$$

together with all the properties implicit

Confinement extension properties

- Semantics of confinement can be checked against eg. what happens to standard property

$$\mathbf{emp} * p \leftrightarrow p \leftrightarrow p * \mathbf{emp}$$

arising from two facts

$$\begin{aligned} H[\mathbf{emp}]S &\equiv H = \perp \\ H * (H', \perp) &\equiv H = H' \end{aligned}$$

- In the confined variants, semantics rules eventually lead us eg.

$$H[p]S \wedge \Phi_\alpha(H, \perp) \equiv H[p]S$$

or

$$H[p]S \wedge \Phi_\alpha(\perp, H) \equiv H[p]S$$

where α ranges over the three given variants.

Confinement extension properties

- When we check $\Phi_\alpha(\perp, H)$ and $\Phi_\alpha(H, \perp)$ for $\alpha := \text{In}$, for instance, calculations easily lead to:

$$\mathbf{emp} \odot p \leftrightarrow p$$

and

$$p \odot \mathbf{emp} \leftrightarrow p \Leftarrow p \rightarrow \mathbf{emp}$$

recalling

$$\text{In}(H_1, H_2) \stackrel{\text{def}}{=} H_1 \parallel H_2 \wedge \epsilon_F \cdot H_1 \subseteq H_2^\circ \cdot \top$$

- The two other variants trivially preserve the standard rule.

Discussion

- Is confined separation logic *enough* for reasoning about confinement in object-oriented programs? Shuling will tell from her current experiments
- If not, we anyway have a quite flexible framework for further extending the logic, if necessary
- Framework which is **parametric** on the *shapes* of both heap and store (this is relevant in OO, because every object is itself a “little store”, cf. instance variables)
- Each shape has its own membership

Background: PF-membership

A very powerful device:

$$\in_K \stackrel{\text{def}}{=} \perp \quad (25)$$

$$\in_{\text{Id}} \stackrel{\text{def}}{=} \textit{id} \quad (26)$$

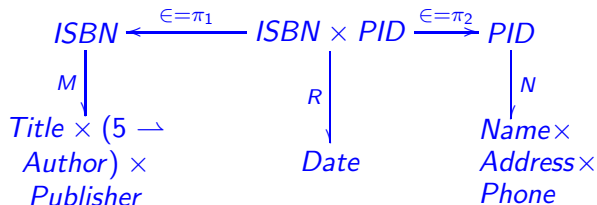
$$\in_{F \times G} \stackrel{\text{def}}{=} (\in_F \cdot \pi_1) \cup (\in_G \cdot \pi_2) \quad (27)$$

$$\in_{F+G} \stackrel{\text{def}}{=} \in_F \quad (28)$$

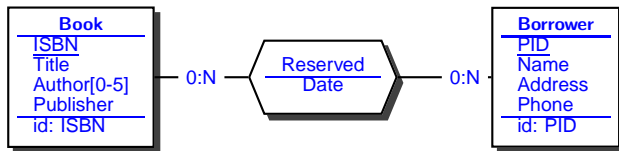
$$\in_{F \cdot G} \stackrel{\text{def}}{=} \in_G \cdot \in_F \quad (29)$$

Closing

- Synergy?



cf.



Closing

- More about this work in our paper [6]
- Last but not least — calculation superior to *invention + verification*:

(Bear in mind the following was written circa 300 years ago:)

I feel that controversies can never be finished . . . unless we give up complicated reasonings in favour of simple calculations, words of vague and uncertain meaning in favour of fixed symbols . . . every argument is nothing but an error of calculation. [With symbols] when controversies arise, there will be no more necessity for disputation between two philosophers than between two accountants. Nothing will be needed but that they should take pen and paper, sit down with their calculators, and say 'Let us calculate'.

Gottfried Wilhelm Leibniz (1646-1716), quoted in [3]

Related work

- Currently studying the upper adjoint of *split* in

$$\langle R, S \rangle \subseteq X \quad \equiv \quad S \subseteq R \triangleright X$$

recall

$$b(R \triangleright S)a \equiv \langle \forall c : c R a : (c, b) S a \rangle$$

in particular instantiated to functions

$$b(f \triangleright g)a \equiv (f\ a, b) = g\ a \quad (30)$$

satisfying properties such as eg.

$$b(f \triangleright \langle g, h \rangle)a \equiv f\ a = g\ a \wedge b = h\ a$$



R.C. Backhouse.

Mathematics of Program Construction.

Univ. of Nottingham, 2004.

Draft of book in preparation. 608 pages.



B. Bokowski and J. Vitek.

Confined types.

In *Proceedings of OOPSLA'99*, pages 82–96. ACM Press, New York, NY, USA, 1999.



C.B. Jones.

Systematic Software Development Using VDM.

Series in Computer Science. Prentice-Hall International, 1986.



J.N. Oliveira.

Transforming Data by Calculation.

In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE 2007 Proceedings*, pages 139–198, July 2007.



J.C. Reynolds.

Separation logic: a logic for shared mutable data structures, 2002.

Invited Paper, LICS'02.



Wang Shuling, L.S. Barbosa, and J.N. Oliveira.

A relational model for confined separation logic, Sep. 2007.
Submitted.



Wang Shuling and Qiu Zongyan.

Towards a semantic model of confinement with confined separation logic.

Technical report, School of Math., Peking University, 2007.