Concurrent separation logic

Stephen Brookes

Lugano 2007
Summer School

Practical session

Using and extending the logic

- Examples
- Recursive specifications
- Recursive procedures

... some advantages of this approach

logic

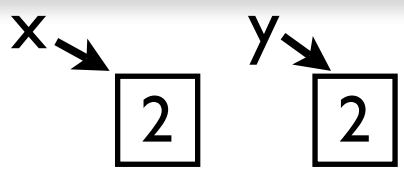
Formulas describe store + heap

Brief recap:

- emp the heap is *empty*
- $e \mapsto e'$ *singleton* heap
- p*q p and q hold in *separate* sub-heaps
- $p \wedge q$ p and q hold in the *same* heap

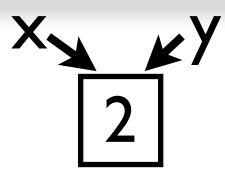
examples

- $x \mapsto 2 * y \mapsto 2$



heap consists of distinct cells denoted by x, y and containing 2

- $x \mapsto 2 \wedge y \mapsto 2$



heap consists of single cell denoted by x, y and containing 2

properties

$$\mathsf{emp} * p = p$$

$$p * q = q * p$$

$$(p * q) * r = p * (q * r)$$

$$(p_1 \vee p_2) * q \;=\; (p_1 * q) \vee (p_2 * q)$$

$$(p_1 \wedge p_2) * q \;\Rightarrow\; (p_1 * q) \wedge (p_2 * q)$$

$$\exists x.(p * q) \;=\; (\exists x.p) * q$$

if x *not free in* q

properties

- When p is *pure* *(independent of heap)*

$$(p \wedge \mathsf{emp}) * q = p \wedge q$$

- When p and q are *pure*

$$p * q = p \wedge q$$

- When q is *precise* *(holds in unique sub-heap)*

$$(p_1 \wedge p_2) * q = (p_1 * q) \wedge (p_2 * q)$$

Recall: *expressions e, b are pure*
resource invariants are precise

exercises

- Verify the properties from the previous slide
  - For each equation of the form $p_1 = p_2$ show that, for all stores s and heaps h,

$$(s,h) \models p_1 \quad \text{if and only if} \quad (s,h) \models p_2$$

inference rules

parallel

$$\frac{\Gamma \vdash \{p_1\}c_1\{q_1\} \quad \Gamma \vdash \{p_2\}c_2\{q_2\}}{\Gamma \vdash \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}}$$

if $\text{free}(c_1) \cap \text{writes}(c_2) \subseteq \text{owned}(\Gamma)$
& $\text{free}(c_2) \cap \text{writes}(c_1) \subseteq \text{owned}(\Gamma)$
& $\text{free}(p_1, q_1) \cap \text{writes}(c_2) = \{\}$
& $\text{free}(p_2, q_2) \cap \text{writes}(c_1) = \{\}$

region

$$\frac{\Gamma \vdash \{(p * R) \wedge b\}c\{q * R\}}{\Gamma, r(X):R \vdash \{p\}\textbf{with } r \textbf{ when } b \textbf{ do } c\{q\}}$$

resource

$$\frac{\Gamma, r(X):R \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * R\}\textbf{resource } r \textbf{ in } c\{q * R\}}$$

inference rules

update

$$\frac{}{\Gamma \vdash \{\, e \mapsto \_ \,\}[e] := e' \{\, e \mapsto e' \,\}}$$

when free(e,e′)∩owned(Γ) = { }

dispose

$$\frac{}{\Gamma \vdash \{\, e \mapsto \_ \,\}\text{dispose } e \{\, \text{emp} \,\}}$$

when free(e)∩owned(Γ) = { }

frame

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * R\}c\{q * R\}}$$

when free(R)∩writes(c) = { }

and free(R)∩owned(Γ) = { }

distributed counting

```
resource r in
    COUNT(M;x,z) || COUNT(N;y,z)
```

... shared variable z tracks x+y

COUNT(M;x,z) ::

```
local k=0 in
    while k<M do
        (with r do (x:=x+1; z:=z+1); k:=k+1)
```

COUNT(N;y,z) ::

```
local k=0 in
    while k<N do
        (with r do (y:=y+1; z:=z+1); k:=k+1)
```

exercise

Using the inference rules, prove:

$\vdash \{ x=y=z=0 \wedge M \geq 0 \wedge N \geq 0 \}$

resource r **in**

COUNT(M;x,z) || COUNT(N;y,z)

$\{ x=M \wedge y=N \wedge z=M+N \}$

proof outline

$\vdash$ { x=y=z=0 ∧ M≥0 ∧ N≥0 }

{ (x=0∧M≥0) ∗ (y=0∧N≥0) ∗ (z=x+y∧emp) }

resource r **in**

{ (x=0∧M≥0) ∗ (y=0∧N≥0) }

COUNT(M;x,z) || COUNT(N;y,z)

{x=M ∗ y=N}

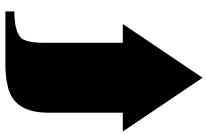
{ x=M ∗ y=N ∗ (z=x+y∧emp)}

{ x=M ∧ y=N ∧ z=M+N }

ingredients

$r(z): z=x+y \wedge emp \vdash$
$\{x=0 \wedge M \geq 0\}$ COUNT(M;x,z) $\{x=M\}$

$r(z): z=x+y \wedge emp \vdash$
$\{y=0 \wedge N \geq 0\}$ COUNT(N;y,z) $\{y=N\}$



parallel rule

$r(z): z=x+y \wedge emp \vdash$
$\{(x=0 \wedge M \geq 0) * (y=0 \wedge N \geq 0)\}$
COUNT(M;x,z) || COUNT(N;y,z)
$\{x=M * y=N\}$

put

$R :: (full \wedge c \mapsto \_) \vee (\neg full \wedge emp)$

buf(c,full) : $R \vdash \{ x \mapsto \_ \}$

with buf **when** ¬full **do**

$\{ (x \mapsto \_ * R) \wedge \neg full \}$

(c:=x; full:=true)

$\{ emp * R \}$

$\{ emp \}$

ownership transfer

get

$R :: (\text{full} \wedge c \mapsto \_) \vee (\neg\text{full} \wedge \text{emp})$

buf(c,full) : $R \vdash$ { emp }

with buf **when** full **do**

{ (emp * R)∧full }

(y:=c; full:=false)

{ $y \mapsto \_ * R$ }

{ $y \mapsto \_$ }

ownership transfer

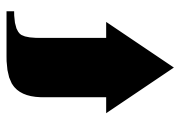
put || get

ownership transfer

$R :: (full \wedge c \mapsto \_) \vee (\neg full \wedge emp)$

$buf(c,full) : R \vdash \{ x \mapsto \_ \}$ PUT $\{ emp \}$

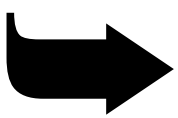
$buf(c,full) : R \vdash \{ emp \}$ GET $\{ y \mapsto \_ \}$



$buf(c,full) : R \vdash \{ x \mapsto \_ * emp \}$

PUT || GET

$\{ emp * y \mapsto \_ \}$



$buf(c,full) : R \vdash \{ x \mapsto \_ * emp \}$

PUT || (GET; dispose y)

$\{ emp * emp \}$

put

$R' :: (\text{full} \wedge \text{emp}) \vee (\neg\text{full} \wedge \text{emp})$

$\text{buf}(c,\text{full}) : R' \vdash \{ x \mapsto \_ \}$

with buf **when** ¬full **do**

$\{ (x \mapsto \_ * R') \wedge \neg\text{full} \}$

(c:=x; full:=true)

$\{ x \mapsto \_ * R' \}$

$\{ x \mapsto \_ \}$

no ownership transfer

get

$R' :: (\text{full} \wedge \text{emp}) \vee (\neg\text{full} \wedge \text{emp})$

buf(c,full) : $R' \vdash$ { emp }

with buf **when** full **do**

{ (emp * R')∧full }

(y:=c; full:=false)

{ emp * R' }

{ emp }

no ownership transfer

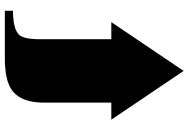
put || get

no ownership transfer

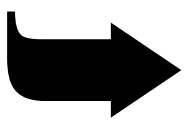
$R' :: (\text{full} \wedge \text{emp}) \vee (\neg\text{full} \wedge \text{emp})$

$\text{buf}(c,\text{full}) : R' \vdash \{ x \mapsto \_ \}\ \text{PUT}\ \{ x \mapsto \_ \}$

$\text{buf}(c,\text{full}) : R' \vdash \{ \text{emp} \}\ \text{GET}\ \{ \text{emp} \}$



$\text{buf}(c,\text{full}) : R' \vdash \{ x \mapsto \_ * \text{emp} \}$
$\text{PUT} \parallel \text{GET}$
$\{ x \mapsto \_ * \text{emp} \}$



$\text{buf}(c,\text{full}) : R' \vdash \{ x \mapsto \_ * \text{emp} \}$
$(\text{PUT};\text{dispose } x) \parallel \text{GET}$
$\{ \text{emp} * \text{emp} \}$

...hence

$\vdash$ { ¬full $\wedge$ x$\mapsto$ _ }

resource r **in**
PUT || (GET; dispose y)

{ emp }

$\vdash$ { ¬full $\wedge$ x$\mapsto$ _ }

resource r **in**
(PUT; dispose x) || GET

{ emp }

provable

so

race-free

racy version

(PUT; dispose x) || (GET; dispose y)

- Racy from any initial state such that $x \mapsto \_$
- There are no Γ , q such that

$$\Gamma \vdash \{ x \mapsto \_ \}$$
(PUT; dispose x) || (GET; dispose y)
$$\{ q \}$$

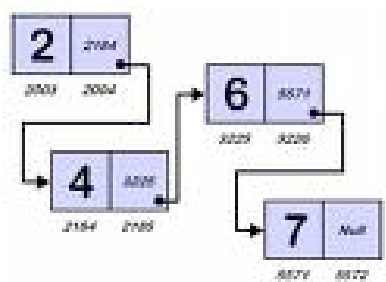
is provable

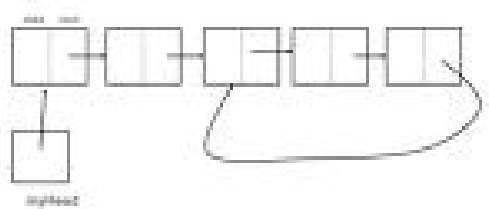
provable programs are race-free

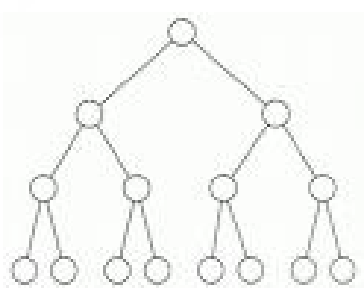
Recursive specs

... an extension to the logic

- Very convenient for describing recursive data structures







singly-linked lists

doubly-linked lists

acyclic lists

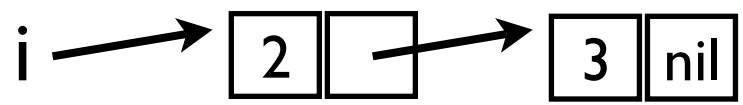
binary trees

..... *Reynolds, O'Hearn, etc*

Singly-linked lists

List(i,L) = "i *represents list* L"

- List(i, [2,3])



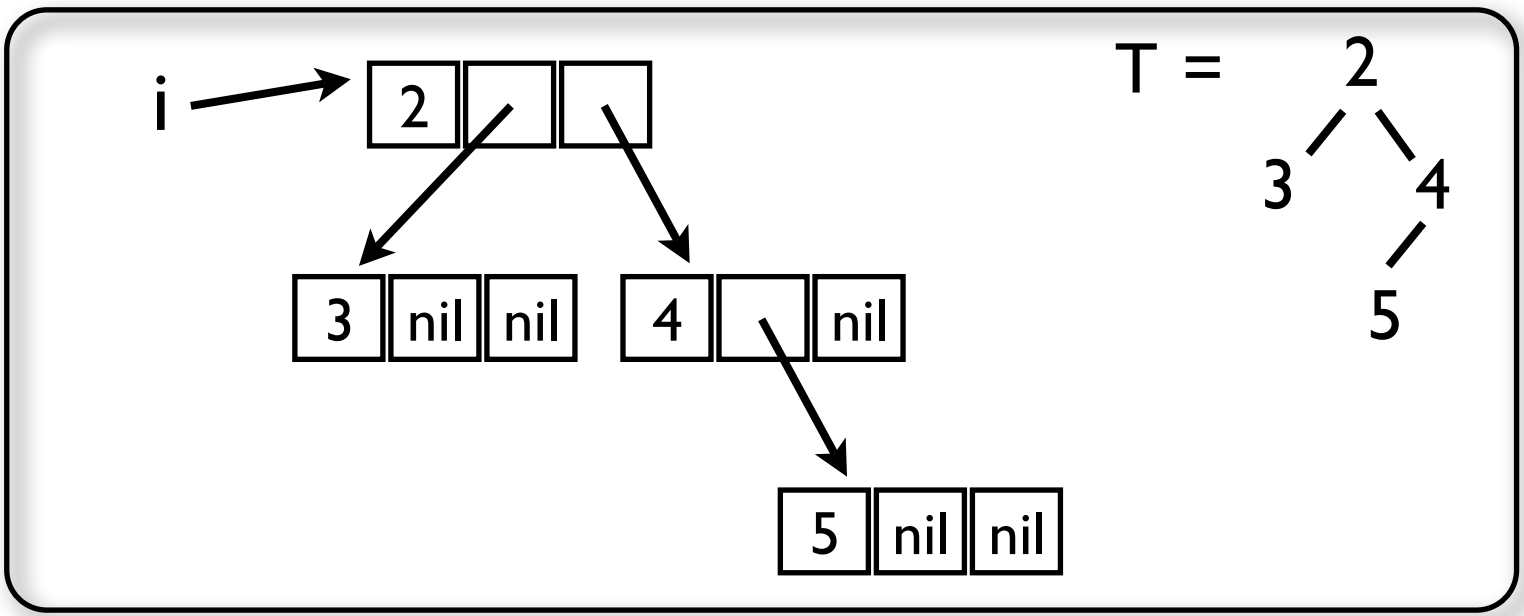
$\text{List}(i, L) =_{\text{def}} (L = [\,] \wedge \mathbf{emp} \wedge i{=}\text{nil})$

$\vee \ (\exists x, i', L'.\ L{=}x::L' \wedge i{\mapsto}x * i{+}1{\mapsto}i' * \text{List}(i', L'))$

Binary trees

Tree(i,T) = "i *represents tree* T"

T ::= Leaf | Node(e,T_1,T_2)



Tree(x, Leaf) $=_{def}$ x=nil $\wedge$ emp

Tree(x, Node(e,T_1,T_2)) $=_{def}$

$\exists x_1,x_2.\ x \mapsto e * x+1 \mapsto x_1 * x+2 \mapsto x_2$

$* \text{Tree}(x_1,T_1) * \text{Tree}(x_2,T_2)$

exercise

- Prove or disprove the following:

  (i) For all lists L,

  $$\vdash \{\mathrm{List}(i,L)\}\ x{:=}\mathrm{cons}(e,i)\ \{\mathrm{List}(x, e::L)\}$$

  (ii) For all lists L_1, L_2

  $$\mathrm{List}(i,L_1@L_2) \Rightarrow \exists j.\ \mathrm{List}(i, L_1) * \mathrm{List}(j,L_2)$$

procedures

... an extension to the programming language

- Procedure *declarations* of form h(xs; ys) = c
- Commands can contain *calls* h(es; zs)

h	*procedure name*
xs	list of variables only read in c
ys	list of variables written in c *... no global variables!*
es	list of integer expressions
zs	list of variables *... no aliasing!*

Examples *(non-recursive)*

```
swap(a, i, j) = local x, y in
                  ( x:=[a+i]; y:=[a+j];
                    [a+i]:=y; [a+j]:=x )
```

```
count(M;x,z) = local k=1 in
                 while k≤M do
                  (with r do (x:=x+1; z:=z+1); k:=k+1)
```

```
reverse(; i, j) = local k in
                    ( j:= nil;
                      while i≠nil do
                       ( k:=[i+1]; [i+1]:=j; j:=i; i:=k) )
```

Inference rules

for non-recursive procedures

- In scope of declaration h(xs; ys)=c

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p\}h(xs; ys)\{q\}}$$

non-recursion rule

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\pi\Gamma \vdash \{\pi p\}\pi c\{\pi q\}}$$

(π *a non-aliasing substitution*)

call rule

Substitutions

- π: Ide $\rightarrow$ Exp
- For i $\in$ writes(c)$\cup$owned(Γ)
  and j $\in$ free(p,c,q,Γ), πi $\in$ Ide - free(πj)

  ➥ maps a *write variable* to a *variable*

  ➥ *avoids aliasing*

- π: Exp $\rightarrow$ Exp, π: Com $\rightarrow$ Com, and $\pi\Gamma$
  defined in the obvious way,
  replacing each free occurrence of i by πi

examples

$\vdash \{a+i \mapsto x * a+j \mapsto y\}$ swap(a,i,j) $\{a+i \mapsto y * a+j \mapsto x\}$

r(z): $z=x+y \wedge emp \vdash$
$\{x=0 \wedge M \geq 0\}$ count(M;x,z) $\{x=M\}$

$\vdash$ {List(i,L)} reverse(;i, j) {List(j, *rev* L)}
where
rev [ ] = [ ]
rev (x::L′) = (*rev* L′) @ [x]

all are valid and provable

exercises

- Using Substitution and Consequence, derive

$r(z)\colon z=x+y \wedge \mathrm{emp} \vdash \{y=0 \wedge N\geq 0\}\ \mathrm{count}(N;y,z)\ \{y=N\}$

from

$r(z)\colon z=x+y \wedge \mathrm{emp} \vdash \{x=0 \wedge M\geq 0\}\ \mathrm{count}(M;x,z)\ \{x=M\}$

- Show that for all lists L,

$\vdash \{\mathrm{List}(i,L)\}\ \mathrm{reverse}(;i,j)\ \{\mathrm{List}(j, \mathit{rev}\ L)\}$

is provable

inference rule

for recursive procedures

- In scope of declaration h(xs; ys)=c

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p\}h(xs;\, ys)\{q\}}$$

recursion rule

tree disposal

(a recursive procedure)

... dispose heap cells representing a tree

```
chop x =
  if x=nil then skip else
    local x1, x2 in
     (
       x1:=[x+1]; x2:=[x+2];
       chop x1 || chop x2 ||
       dispose x || dispose(x+1) || dispose(x+2)
     )
```

$\vdash$ {Tree(x,T)} chop x {emp}

exercise

- Show by structural induction on T that for all trees T,

$$\vdash \{\mathrm{Tree}(x,T)\}\ \mathrm{chop}\ x\ \{\mathrm{emp}\}$$

  is provable

Trees are inductively given by

$T ::= \mathrm{Leaf} \mid \mathrm{Node}(e, T_1, T_2)$

$\mathrm{Tree}(x, \mathrm{Leaf}) =_{\mathrm{def}} x{=}\mathrm{nil} \wedge \mathrm{emp}$

$\mathrm{Tree}(x, \mathrm{Node}(e,T_1,T_2)) =_{\mathrm{def}}$

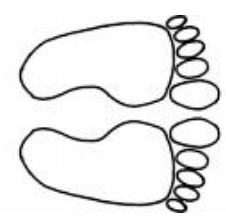
$\exists x_1,x_2.\ x \mapsto e * x{+}1 \mapsto x_1 * x{+}2 \mapsto x_2$

$* \mathrm{Tree}(x_1,T_1) * \mathrm{Tree}(x_2,T_2)$

```
chop x =
  if x=nil then skip else
     local x1, x2 in
( x1:=[x+1]; x2:=[x+2];
 chop x1 || chop x2 ||
 dispose x || dispose(x+1) || dispose(x+2)
)
```

advantages

- The *chop* procedure works because there's ***no sharing*** of heap between the sub-trees
- The use of * in the proof makes this clear
- No need to include complex assumptions
  - "*nobody else touches my tree*"
  - "*I don't touch anything else*"
- Specifications are *tight*
  - describe the *footprint* of program



Array

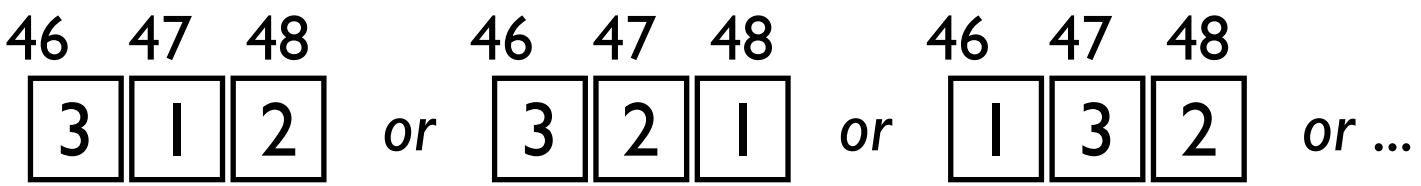
$$\text{Array}(a, i, j, L) = (i>j \wedge L=[\,] \wedge \text{emp}) \vee (i \leq j \wedge \exists x \in L.\ a+i \mapsto x * \text{Array}(a, i+1, j, L \backslash x))$$

Sorted

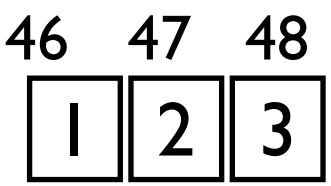
$$\text{Sorted}(a, i, j, L) = (i>j \wedge L=[\,] \wedge \text{emp}) \vee (i \leq j \wedge \exists x\,.\, x=\min(L).\ a+i \mapsto x * \text{Sorted}(a, i+1, j, L \backslash x))$$

examples

Array(42, 4, 6, [3,1,2])



Sorted(42, 4, 6, [3,1,2])



partition

- Using swap, write a procedure

partition(a,i,j; k)

satisfying the specification

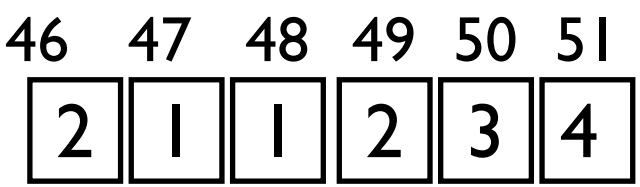
$$\vdash \{\, \mathsf{Array}(a,i,j,L) \wedge i<j \,\}$$
$$\mathsf{partition}(a,i,j;\, k)$$
$$\{\, i \le k \le j \wedge \exists L_1, L_2, m.\ L \approx m::L_1 @ L_2 \wedge L_1 < m \wedge L_2 \ge m \wedge \mathsf{Array}(a,i,k-1,L_1) * \mathsf{Array}(a,k+1,j,L_2) * a+k \mapsto m \,\}$$

Parted(a,i,j,k,L)

intuition

Parted(a,i,j,k,L) = “a[i..k-1] < a[k],
a[k+1,...j] ≥ a[k],
a[i..j] ≈ L”

example



Parted(42,4,9,8,[4,1,2,3,2,1])

parallel quicksort

- Using partition, write a recursive procedure

  quicksort(a,i,j)

  satisfying the specification

$$\vdash \{ \text{Array}(a,i,j,L) \}\ \text{quicksort}(a,i,j)\ \{ \text{Sorted}(a,i,j,L) \}$$

hints

For all lists L, when $i \le k \le j$,

Array(a,i,j,L) *implies*

$\exists\ L_1, L_2, m.\quad L \approx m::L_1 @ L_2\ \wedge$

$\text{Array}(a,i,k-1,L_1) * \text{Array}(a,k+1,j,L_2) * a+k \mapsto m$

$\text{Sorted}(a,i,k-1,L_1) * \text{Sorted}(a,k+1,j,L_2) * a+k \mapsto m$

$\wedge\ \ L_1 < m\ \wedge\ \ L_2 \ge m$

implies $\text{Sorted}(a,i,j,m::L_1 @ L_2)$

parallel quicksort

```
quicksort(a,i,j) =
        if i<j then
                local k in
                    (
                      partition(a,i,j; k);
                      quicksort(a,i,k-1) || quicksort(a,k+1,j)
                    )
                else skip
```

$\vdash$ {Array(a,i,j,L)} quicksort(a,i,j){Sorted(a,i,j,L)}

outline

⊢ {Array(i,j,L)}

if i<j **then**

{Array(i,j,L)∧i<j}

local k **in**

({Array(i,j,L)∧i<j}

partition(a,i,j; k);

{Parted(a,i,j,k,L)}

quicksort(a,i,k-1) || quicksort(a,k+1,j)

{Sorted(a,i,j,L)})

else

{Array(a,i,j,L)∧i≥j}

skip

{Sorted(a,i,j,L)}

{Sorted(a,i,j,L)}

sketch

Let Q be $a+k \mapsto m \wedge L_1 < m \wedge L_2 \geq m$

$\vdash$ {Array(a,i,k-1,L_1) * Array(a,k-1,j,L_2) * Q}
quicksort(a,i,k-1) || quicksort(a,k+1,j)
{Sorted(a,i,k-1,L_1) * Sorted(a,k-1,j,L_2) * Q}

using

- parallel rule
- recursion rule
- substitution rule
- frame rule (using Q)
- rule of consequence

conclusions

- Concurrent separation logic has advantages
  - provable ⇒ race-free
  - local reasoning ⇒ *succinct* specs
  - pointers + concurrency
  - recursion and procedures
- Improvement over traditional methods
  - wider applicability
  - less complex
  - better scalability