Зависимые Типы для верификации Реалистичного Кода

Илья Сергей

ilyasergey.net

UCL

PLC-2017

Тип — это *множество* данных и *операций* над ними.

Как *описать* вычисления

- Формальная спецификация
- Суперкомпиляция
- Равенство
- Взаимозаменяемость
- Завершаемость
- Non-interference

Как *реализовать* вычисления

- Указатели
- **while (true)** {…}
- Input/Output
- **goto/break/continue**
- Exceptions
- Многопоточность
- Распределенные вычисления

λ calculus
A. Church (1930s)

Simply typed λ calculus
A. Church (1940)

Turing Machine
A. Turing (1936)

Типы как Множества

```
Datatype unit := tt.
```

$$\text{unit} \stackrel{\text{def}}{=} \{ \text{tt} \}$$

$$\text{tt} \in \text{unit}$$

```
Datatype unit := tt.
```

$$\text{unit} \stackrel{\text{def}}{=} \{\ \text{tt}\ \}$$

$$\text{tt} : \text{unit}$$

```
Datatype bool := true | false.
```

$$\text{bool} \stackrel{\text{def}}{=} \{\ \text{true}, \text{false}\ \}$$

true : bool

false : bool

```
Datatype nat := 0 | .+1 of nat.
```

$$\text{nat} \stackrel{\text{def}}{=} \{\, 0,\ \underbrace{(0.\!+\!1)}_{1},\ \underbrace{(0.\!+\!1.\!+\!1)}_{2},\ \ldots \,\}$$

$$0 : \text{nat} \qquad\qquad \frac{n : \text{nat}}{n.\!+\!1 : \text{nat}}$$

```
Function negate : bool -> bool :=
  fun b => match b with
    | true  => false
    | false => true
  end.
```

$$\frac{x : A \ \vdash \ e : B}{\mathbf{fun}\ x \Rightarrow e : A \to B}$$

$$\frac{f : A \to B \qquad x : A}{f(x) : B}$$

```
Datatype Record2 A B := {a : A; b : B}

Datatype Record3 A B C :=
                {a : A; b : B; c : C}
```

$$\frac{}{\texttt{Record3}\ A\ B\ C \ <:\ \texttt{Record2}\ A\ B}$$

$$\frac{P <: P' \qquad e : P' \rightarrow Q' \qquad Q' <: Q}{e : P \rightarrow Q}$$

$$\frac{x : A,\ \ f : A \rightarrow B \ \vdash\ e : B}{(\mathrm{Rec}\ f : A \rightarrow B := \mathrm{fun}\ x \Rightarrow e) : A \rightarrow B}$$

```
Rec even : nat -> bool :=
  fun n => match n with
    | n'.+1 => negate (even n')
    | 0     => true
  end.
```

```
Rec even : nat -> bool :=
  fun n => match n with
    | n'.+1 => negate (even n')
    | 0     => 0
  end.
```

Wrong type: bool expected, but nat found.

$$\Pi(x: A).\ B(x)$$

```
F = Π(b: bool). if b then nat else unit
```

```
Function foo : F :=
  fun b => match b with
             | true  => 0
             | false => tt
  end
```

Checkpoint 1
Типы как Множества

- Программы — значения, элементы множеств;
- *Well-typed program don’t go wrong* (R. Milner);
- Типы — спецификация программ;
- Проверка типов — верификация программ;
- Модульность по принципу подстановки: *тип программы независим от контекста ее применения*.

λ calculus
A. Church (1930s)

Simply typed λ calculus
A. Church (1940)

ML
R. Milner (1973)

Haskell
S. Peyton-Jones et al. (1990)

Turing Machine
A. Turing (1936)

Fortran
J. Backus (1957)

C
D. Richie (1972)

C++
B. Stroustrup (1983)

Java
J. Gosling (1995)

λ calculus
A. Church (1930s)

Simply typed λ calculus
A. Church (1940)

Turing Machine
A. Turing (1936)

Fortran
J. Backus (1957)

C
D. Richie (1972)

ML
R. Milner (1973)

C++
B. Stroustrup (1983)

Haskell
S. Peyton-Jones et al. (1990)

Java
J. Gosling (1995)

```
x := 0;

Function x_non_neg : bool -> bool :=
  fun b => match b with
    | true  => x >= 0;
    | false => x < 0;
  end.

x := 1;
x_non_neg(true); // true

x := -1;
x_non_neg(true); // false
```

```
x := 0;

Function x_non_neg: Πb. if b then nat else unit :=
   fun b => match b with
     | true  => if x >= 0 then 0 else tt;
     | false => tt;
   end.

x := 1;
x_non_neg(true); // 0 : nat

x := -1;
x_non_neg(true); // tt : unit
```

λ calculus
A. Church (1930s)

Simply typed λ calculus
A. Church (1940)

Turing Machine
A. Turing (1936)

Fortran
J. Backus (1957)

Program Logics
R.W.Floyd, C.A.R. Hoare (1969)

C
D. Richie (1972)

ML
A. Milner (1973)

C++
B. Stroustrup (1983)

Haskell
S. Peyton-Jones et al. (1990)

Java
J. Gosling (1995)

$$\{ P \}\ c\ \{ Q \}$$

предусловие постусловие

$$\underbrace{s_0 : P(s_0) \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n}_{c} \Rightarrow Q(s_n)$$

$$\{\ \text{True}\ \} \quad x := 3 \quad \{\ x = 3\ \}$$

$$\{\, Q[e/x] \,\}\ x := e\ \{\, Q \,\} \qquad \text{(Assign)}$$

$$\{\, 3 = 3 \,\}\ x := 3\ \{\, x = 3 \,\}$$

$$\frac{\{\ P\ \}\ c_1\ \{\ Q\ \} \qquad \{\ Q\ \}\ c_2\ \{\ R\ \}}{\{\ P\ \}\ c_1;\ c_2\ \{\ R\ \}}\ (\text{Seq})$$

$\{???\}$ x := 3; y := x $\{x = 3 \wedge y = 3\}$

$$\frac{\{\ P\ \}\ c_1\ \{\ Q\ \} \qquad \{\ Q\ \}\ c_2\ \{\ R\ \}}{\{\ P\ \}\ c_1;\ c_2\ \{\ R\ \}}\ (\mathrm{Seq})$$

$\boxed{\{3 = 3 \wedge 3 = 3\}}$

x := 3;

$\{x = 3 \wedge x = 3\}$ (Assign)

y := x

$\{x = 3 \wedge y = 3\}$ (Assign)

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}\ (\text{Conseq})$$

$$\{\ \textbf{True}\ \} \quad \Rightarrow \quad \{3 = 3 \wedge 3 = 3\}$$

$$x := 3;\ y := x$$

$$\{x = 3 \wedge y = 3\}$$

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}\ \text{(Conseq)}$$

$$\{\, \textbf{True} \,\} \quad x := 3;\ y := x \ \ \{x = 3 \land y = 3\}$$

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}\ \text{(Conseq)}$$

$$\frac{P <: P' \qquad e : P' \rightarrow Q' \qquad Q' <: Q}{e : P \rightarrow Q}$$

$$\forall\, a, b,$$

$$\Big(\{x = a \wedge y = b\}\ t := x;\ \ x := y;\ \ y := t\ \{x = b \wedge y = a\} \Big)$$

```
Π(b: bool). if b then nat else unit
```

$$\frac{\{\boxed{Inv} \wedge b\}\ c\ \{\boxed{Inv}\}}{\{\boxed{Inv}\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{\boxed{Inv} \wedge \neg b\}}\ (\text{While})$$

$$\frac{x : A,\ \boxed{f : A \to B} \vdash e : B}{(\textbf{Rec}\ f : A \to B := \textbf{fun}\ x \Rightarrow e) : \boxed{A \to B}}$$

Почему Hoare Logic не работает

```
int ival = 3;
int *x = …;
int *y = …;
```

$\{ x \mapsto - \wedge y \mapsto b \}$ `*x = &ival;` $\{ x \mapsto 3 \wedge y \mapsto b \}$

Что делать, если `x` и `y` указывают на одно и то же значение?

```
int ival = 3;
int *x = …;
int *y = …;
```

$\{\ x \mapsto -\ \wedge\ y \mapsto b\ \}$

```
x = &ival;
```

$\{\ x \mapsto \texttt{ival}\ \wedge$

$(x \neq y \wedge y \mapsto b) \vee (x = y \wedge y \mapsto \texttt{ival})\}$

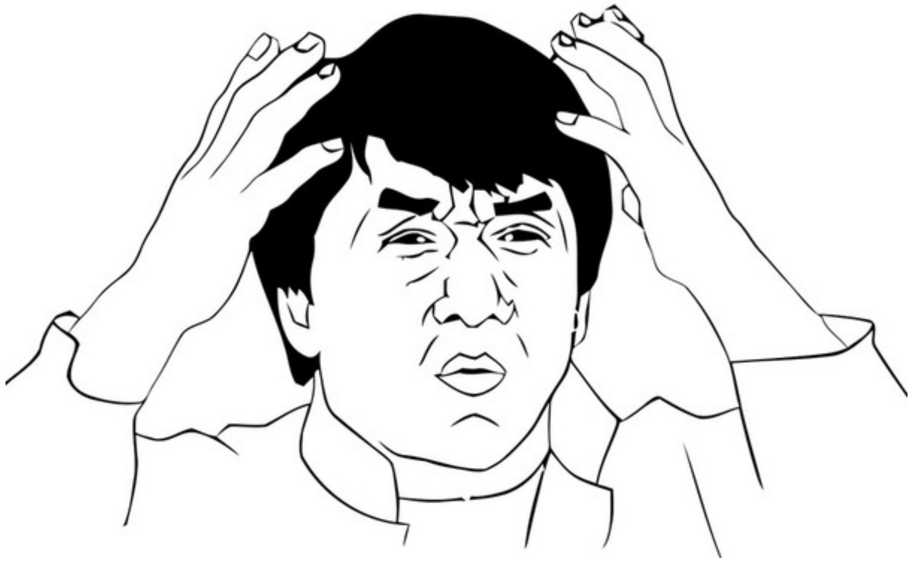
```
int ival = 3;
int *x = …;
int *y = …;
int *z = …;
```

$\{\ x \mapsto - \wedge y \mapsto b\ \}$

```
x = &ival;
```

$\{\ x \mapsto \texttt{ival} \wedge$

$(x \neq y \wedge y \mapsto b) \vee (x = y \wedge y \mapsto \texttt{ival})\}$



λ calculus
A. Church (1930s)

Turing Machine
A. Turing (1936)

Simply typed λ calculus
A. Church (1940)

Fortran
J. Backus (1957)

Program Logics
R.W.Floyd, C.A.R. Hoare (1969)

C
D. Richie (1972)

ML
A. Milner (1973)

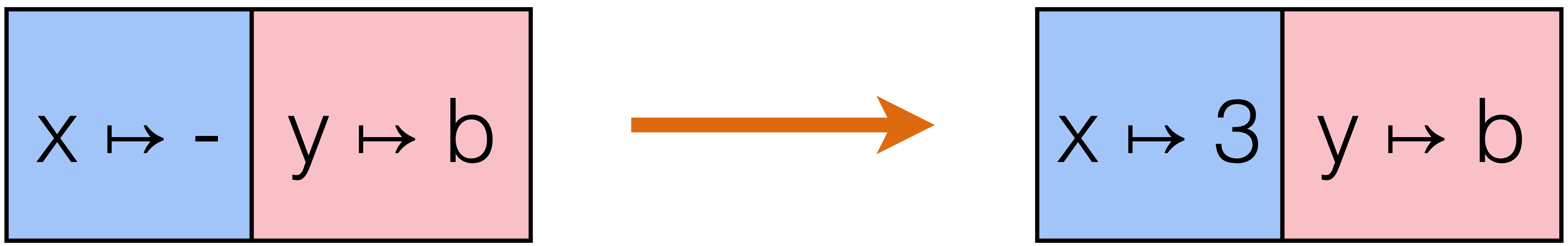
C++
B. Stroustrup (1983)

Java
J. Gosling (1995)

Haskell
S. Peyton-Jones et al. (1990)

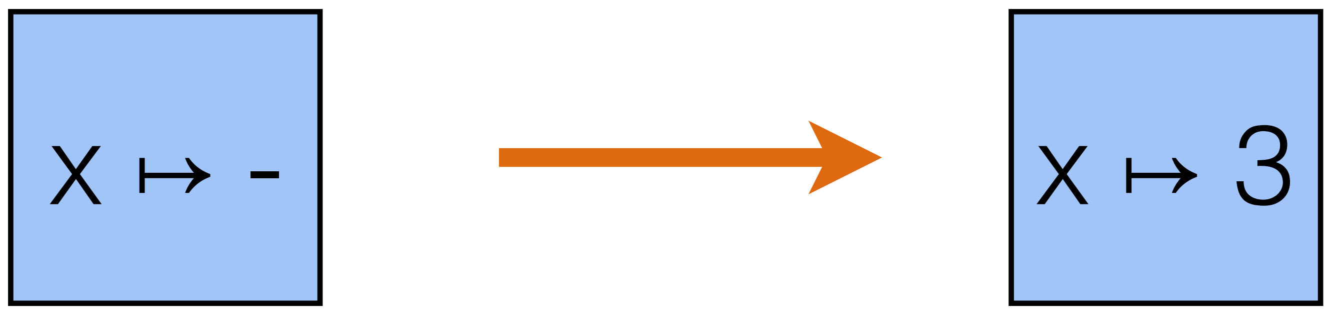
Separation Logic
J. Reynolds (2002)

$\{\, x \mapsto - \wedge y \mapsto b \,\}$ `*x = 3;` $\{\, x \mapsto 3 \wedge y \mapsto b \,\}$



$\{\ x \mapsto - \uplus y \mapsto b\ \}$ `*x = 3;` $\{\ x \mapsto 3 \uplus y \mapsto b\ \}$

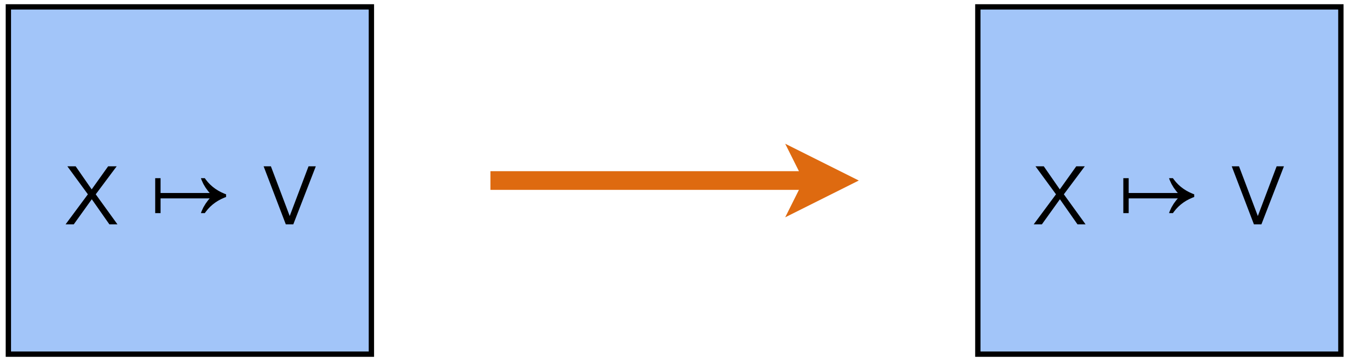
несвязное объединение регионов памяти



$\{\ x \mapsto -\ \}$ `*x = 3;` $\{\ x \mapsto 3\ \}$

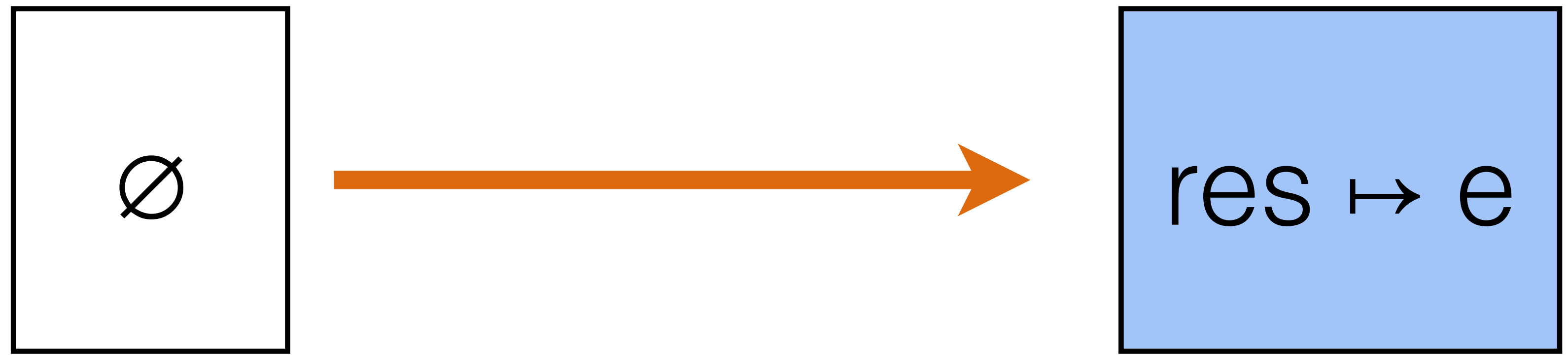
$\{h \mid h = x \mapsto -\}$ $*x = e;$ $\{res, h \mid h = x \mapsto e \wedge res = tt\}$

(Write)



$\{h \mid h = x \mapsto v\}$ *x $\{res, h \mid h = x \mapsto v \wedge res = v\}$

(Read)

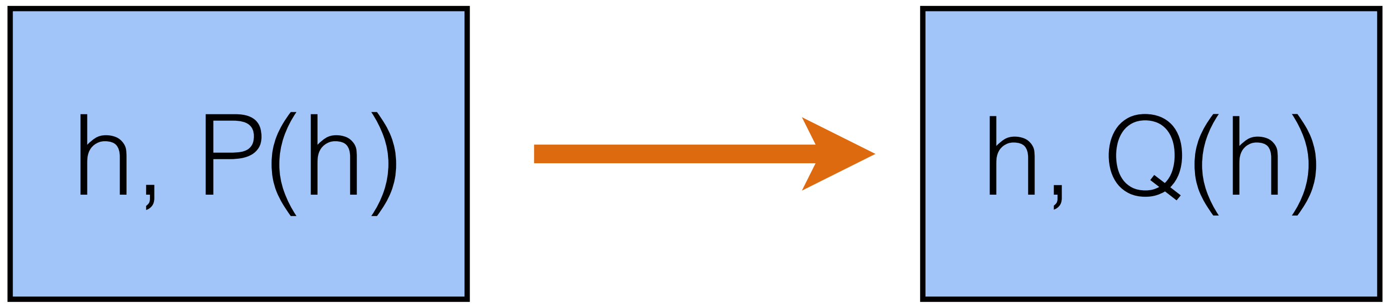


$\{h \mid h = \emptyset\}$ alloc(e); $\{res, h \mid h = res \mapsto e\}$

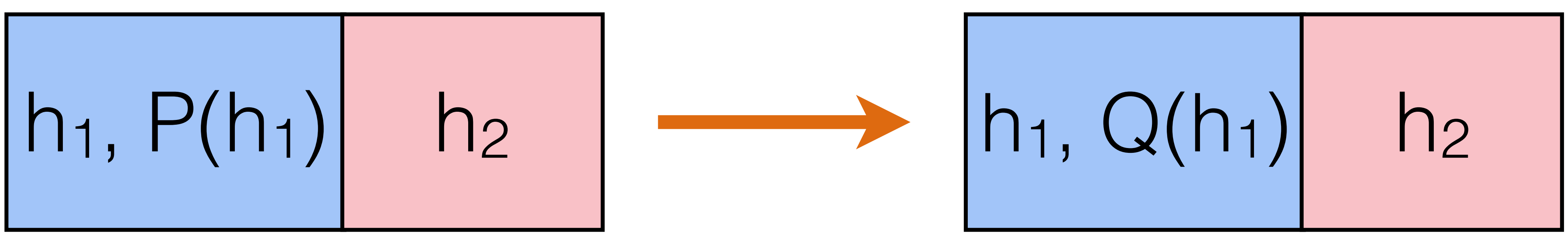
(Alloc)

$\{h \mid h = x \mapsto -\}$ free(x); $\{res, h \mid h = \emptyset \wedge res = tt\}$

(Free)

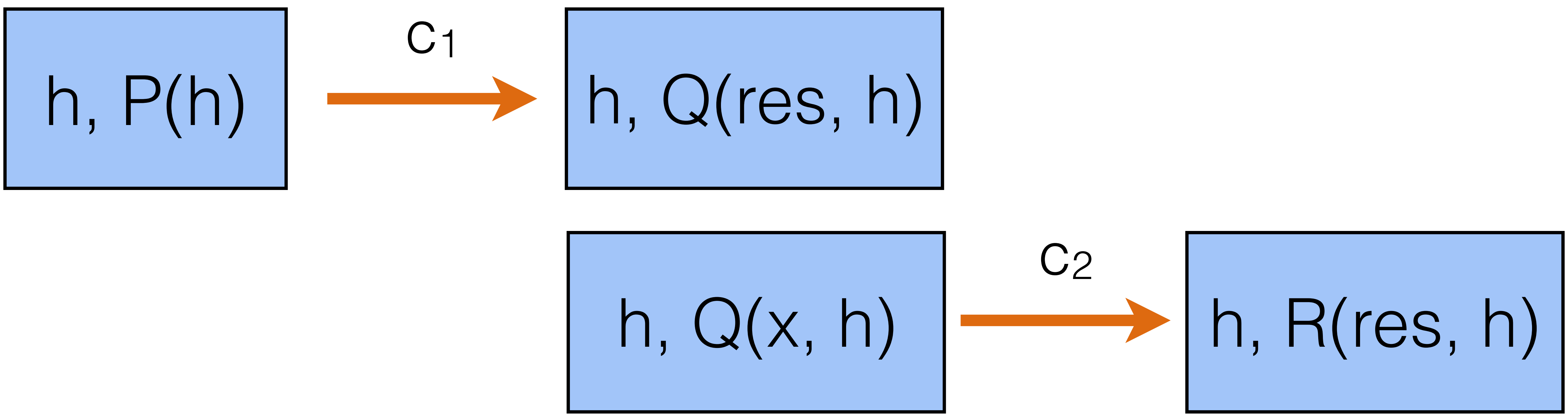


$$\frac{\{h \mid \mathbf{P}(h)\}\ c\ \{res, h \mid \mathbf{Q}(h)\}}{\{h \mid \exists h_1,\ \ h = h_1 \uplus h_2 \wedge\ \mathbf{P}(h_1)\}\ c\ \{res, h \mid \exists h_1,\ \ h = h_1 \uplus h_2 \wedge\ \mathbf{Q}(h_1)\}}\ \text{(Frame)}$$

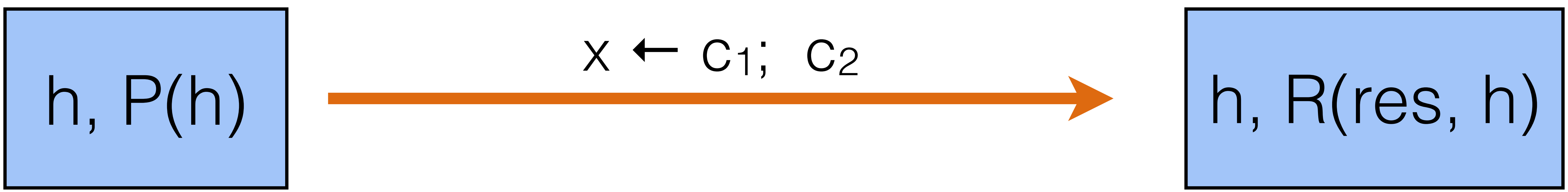


$\{h \mid \mathbf{P}(h)\}\ \ \mathbf{ret}\ e;\ \{res, h \mid \mathbf{P}(h) \land res = e\}$ (Return)

$$\frac{\begin{array}{c}\{h \mid \mathbf{P}(h)\}\ c_1\ \{res, h \mid \mathbf{Q}(res, h)\} \\ \{h \mid \mathbf{Q}(x, h)\}\ c_2\ \{res, h \mid \mathbf{R}(res, h)\}\end{array}}{\{h \mid \mathbf{P}(h)\}\ \ x \leftarrow c_1;\ \ c_2\ \{res, h \mid \mathbf{R}(res, h)\}}\ \text{(Bind)}$$



$$\dfrac{\begin{array}{c}\{h \mid \mathbf{P}(h)\}\ c_1\ \{res, h \mid \mathbf{Q}(res, h)\} \\ \{h \mid \mathbf{Q}(x, h)\}\ c_2\ \{res, h \mid \mathbf{R}(res, h)\}\end{array}}{\{h \mid \mathbf{P}(h)\}\ \ x \leftarrow c_1;\ \ c_2\ \{res, h \mid \mathbf{R}(res, h)\}}\ \text{(Bind)}$$



λ calculus
A. Church (1930s)

Turing Machine
A. Turing (1936)

Simply typed λ calculus
A. Church (1940)

Fortran
J. Backus (1957)

Program Logics
R.W.Floyd, C.A.R. Hoare (1969)

C
D. Richie (1972)

ML
A. Milner (1973)

C++
B. Stroustrup (1983)

Java
J. Gosling (1995)

Haskell
S. Peyton-Jones et al. (1990)

Separation Logic
J. Reynolds (2002)

Facebook Infer
P.O'Hearn et al. (2013)

SL Спецификации как Типы

- Программы — комбинации операций с памятью;
- “Типы” — пред-/постусловия;
- Принцип подстановки (Frame rule);
- *Well-typed programs don’t go wrong.*

Hoare/Separation Logic
+
Dependent Types

Hoare Types

λ calculus
A. Church (1930s)

Turing Machine
A. Turing (1936)

Simply typed λ calculus
A. Church (1940)

Fortran
J. Backus (1957)

Program Logics
R.W. Floyd, C.A.R. Hoare (1969)

ML
A. Milner (1973)

C
D. Richie (1972)

C++
B. Stroustrup (1983)

Haskell
S. Peyton-Jones et al. (1990)

Java
J. Gosling (1995)

Separation Logic
J. Reynolds (2002)

Hoare Type Theory
A. Nanevski (2006)

Facebook Infer
P.O'Hearn et al. (2013)

$$\forall\, x_1, x_2, \ldots \; \{\, h \mid P(h, x_i) \,\} \; c \; \{ res, h \mid Q(res, h, x_i) \,\}$$

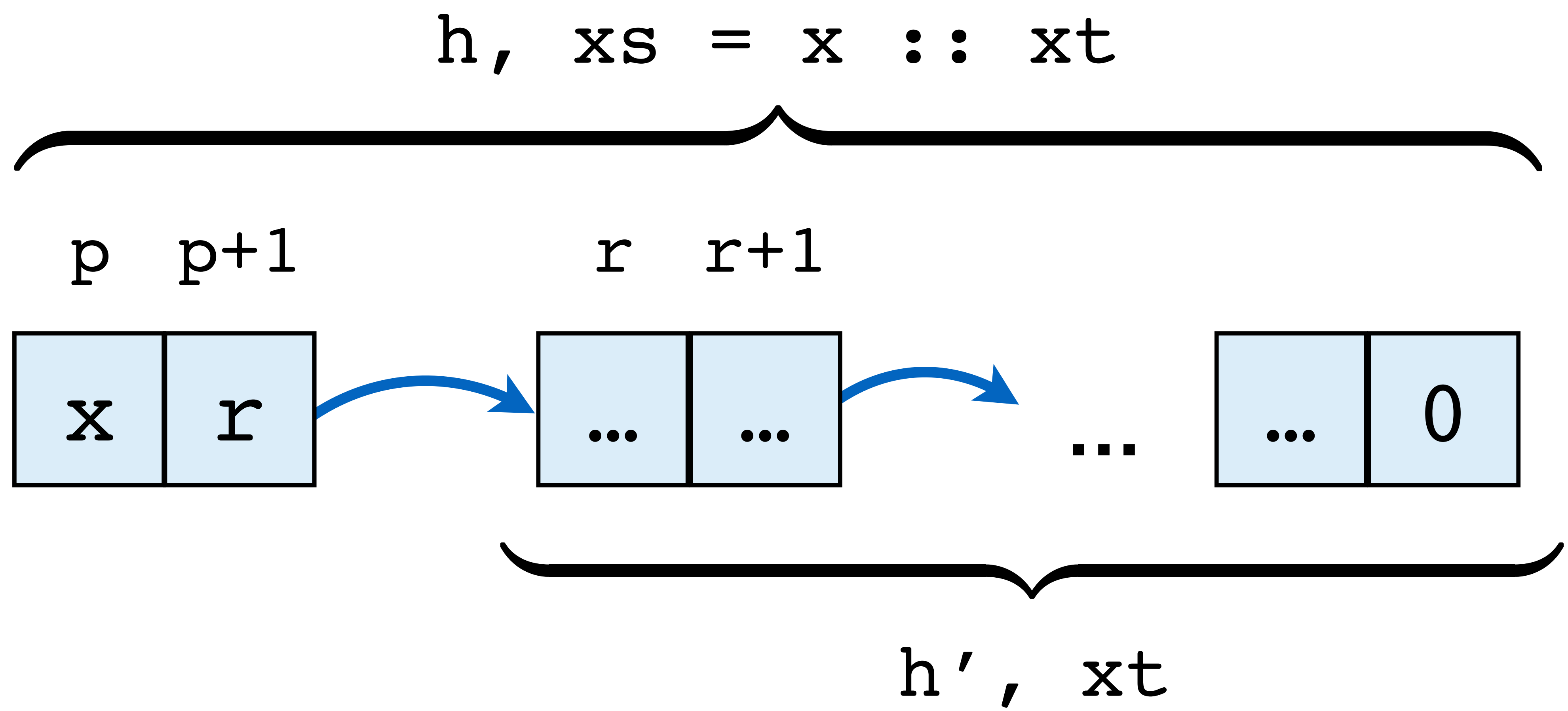
$$c : \{x_1, x_2, \ldots\}\ HT\ (P, Q)$$

```
alloc : Π (A : Type) (v : A),
          HT (fun h ⇒ h = ∅,
              fun res h ⇒ h = res ↦ v)
```

$\boxed{\varnothing}$

$\boxed{res \mapsto v}$

Манипуляции со связными списками




```
Fixpoint llist (p : ptr)(xs : list T) (h: heap) :=
  if xs is x :: xt then
    ∃ r h',
        h = p ↦ x ⊎ (p+1 ↦ r) ⊎ h' ∧ llist r xt h'
  else p = null ∧ h = ∅ .
```

```
Program Definition llist_map f
:= Fix (fun lmap p =>
     Do (if p == null
         then return tt
         else t ← *p;
              *p = f(t);
              nxt ← *(p + 1);
              lmap nxt)).
```

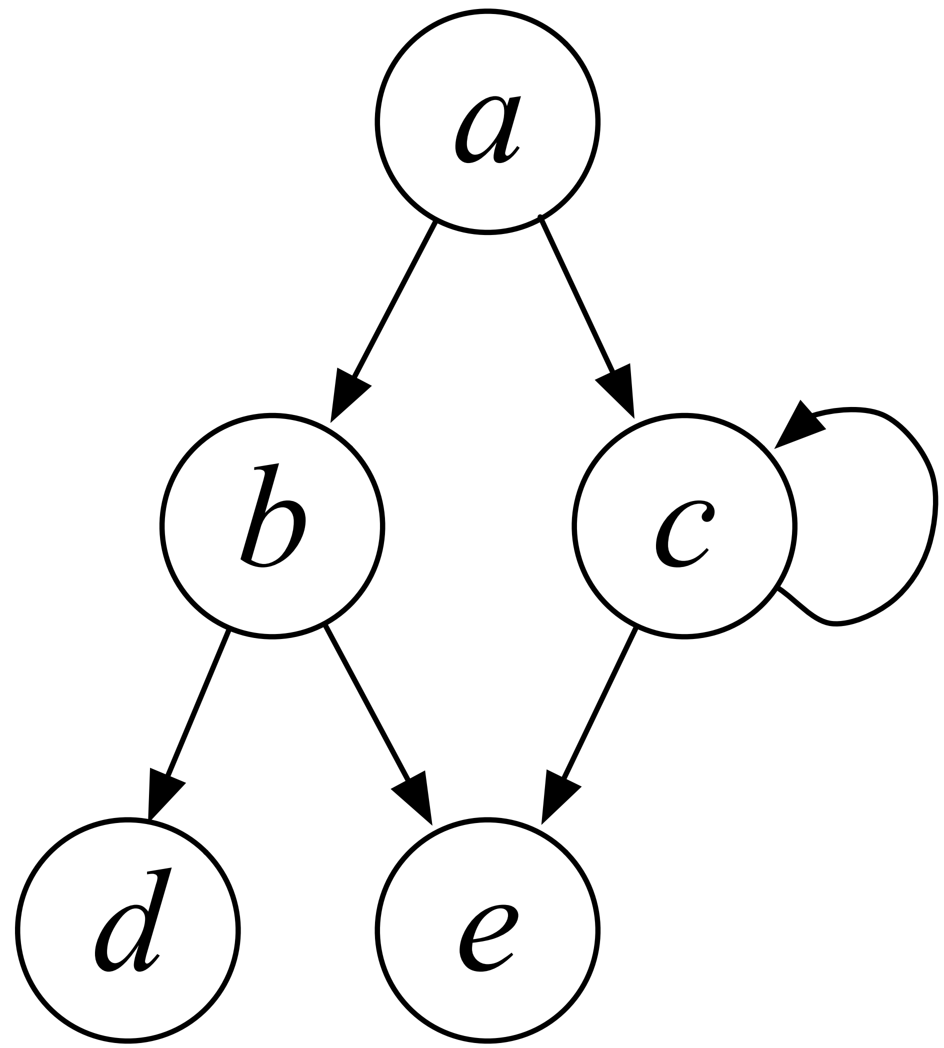
```
Definition llist_map_type (f : T -> S) :=
  forall (p : ptr),
    {xs : list T},
    HT (fun h => llist p xs h,
        fun (_ : unit) h =>
          llist p (map f xs) h).


Program Definition llist_map f : llist_map_type f
:= Fix (fun lmap p =>
    Do (if p == null
        then return tt
        else t ← *p;
             *p = f(t);
             nxt ← *(p + 1);
             lmap nxt)).
```

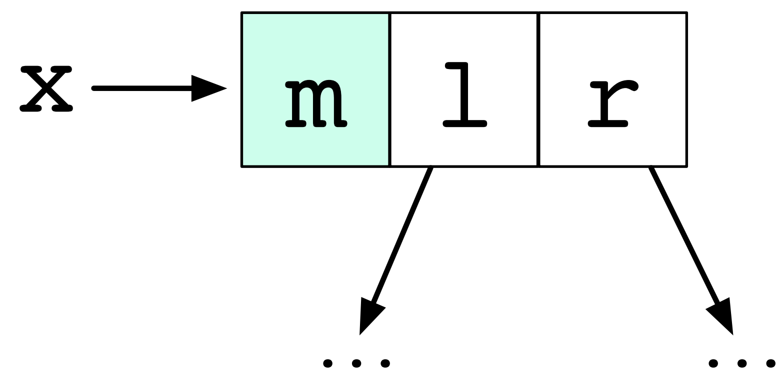
Proof. (6 LOC) **Qed.**

Checkpoint 2: Зависимые типы для программ указателями

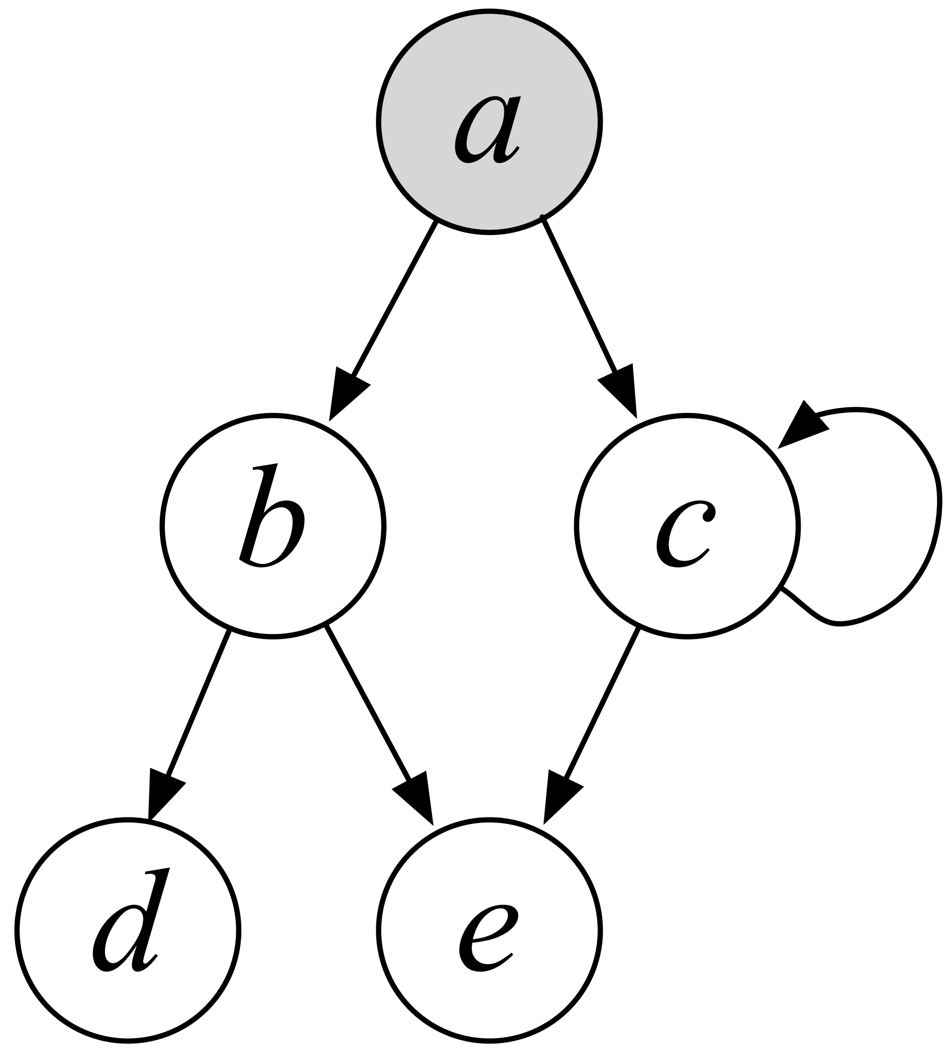
- Зависимые типы описывают *состояние памяти*;
- *Правила вывода* Hoare Types соответствуют *доказательствам* в Separation Logic;
- Проверка типов — верификация программ;
- *Well-typed programs don't go wrong* (again…).

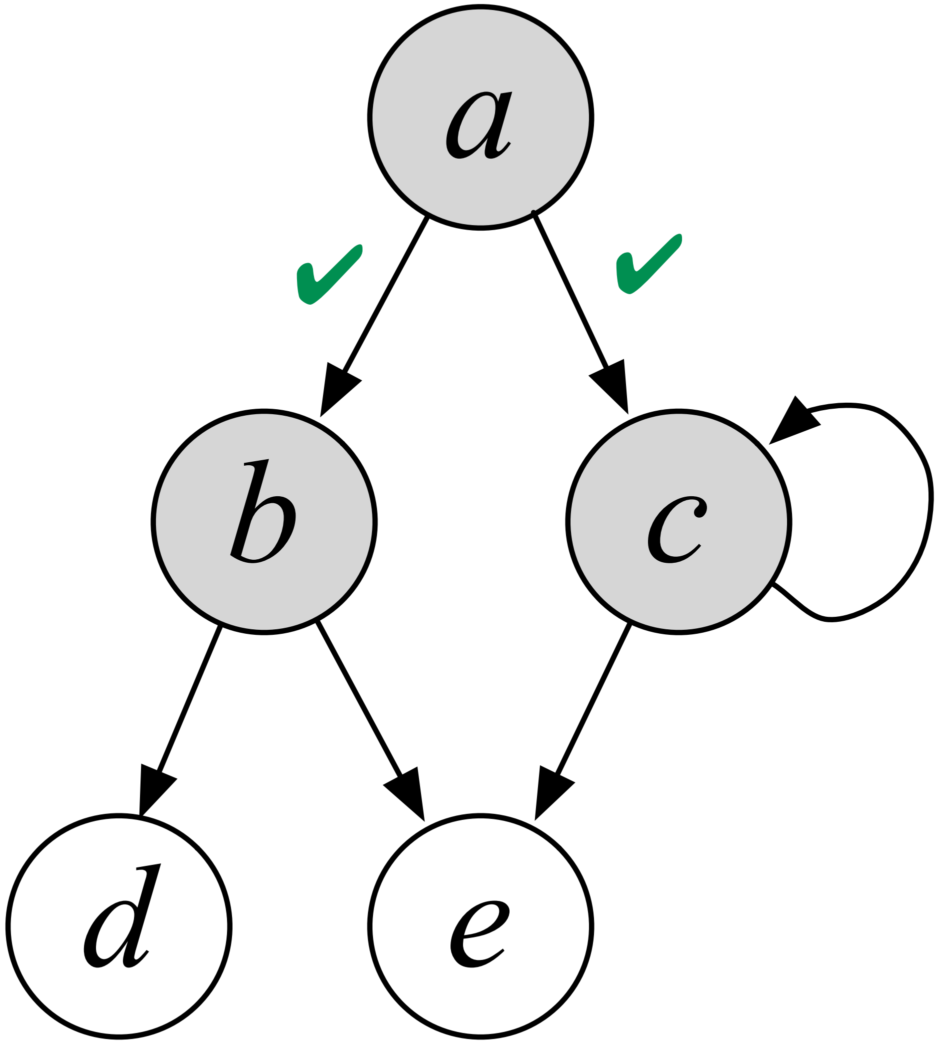
Многопоточность

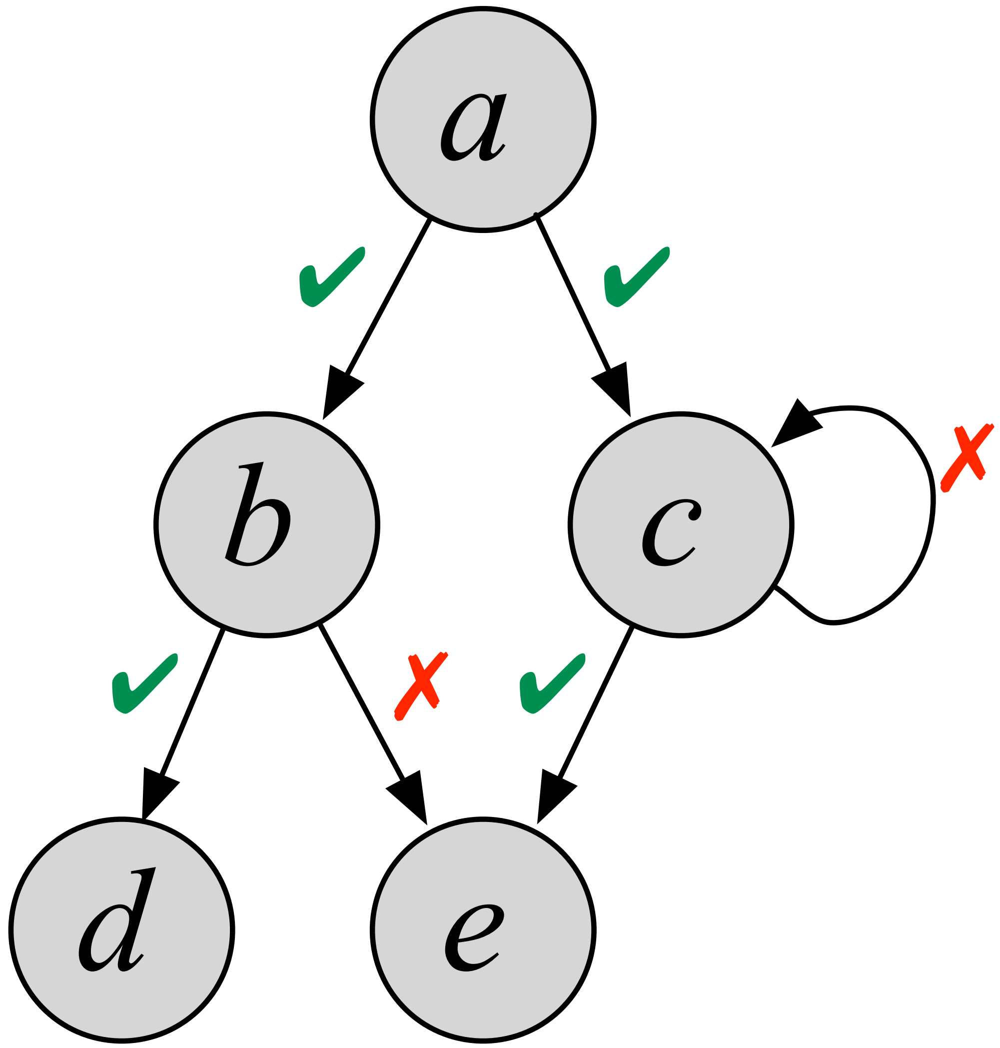

a
b
c
d
e

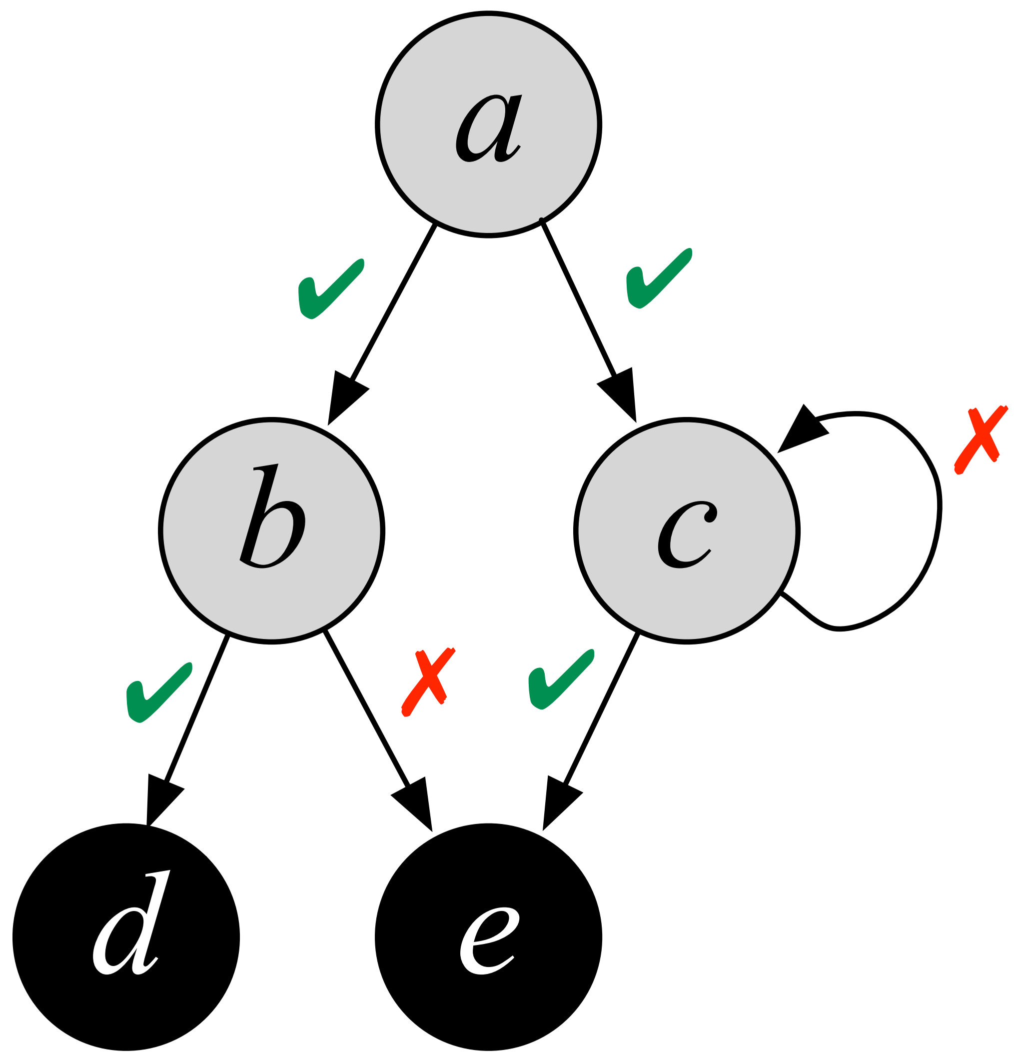


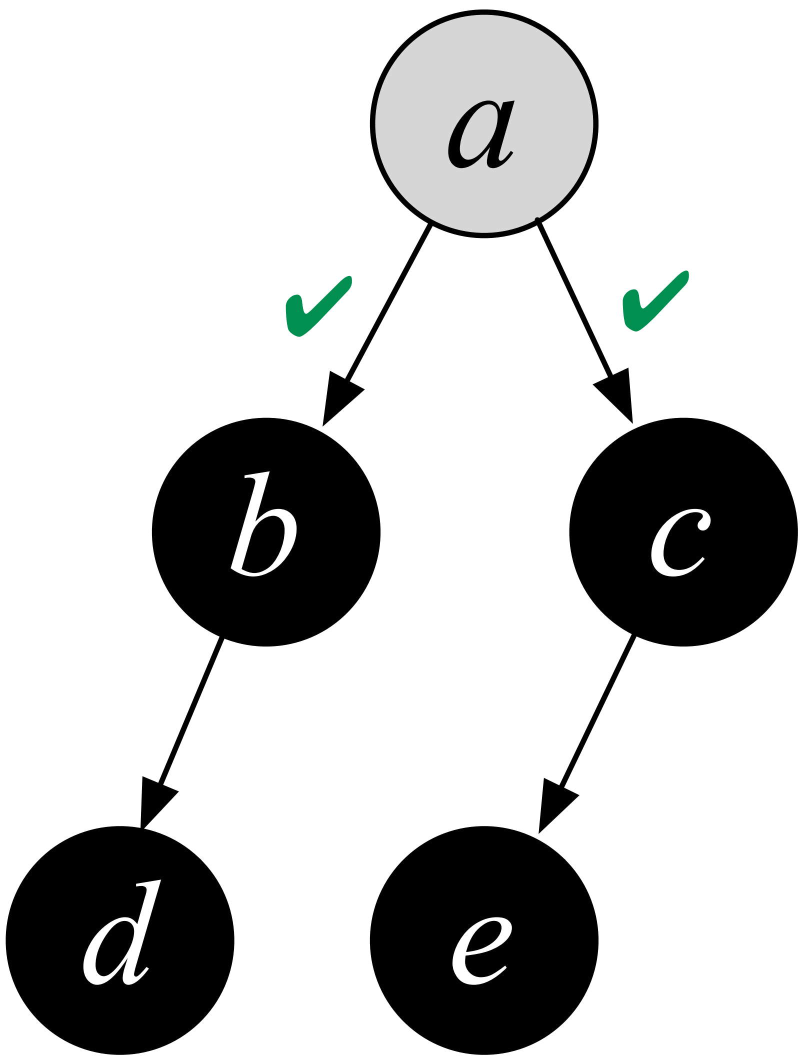

```
Program Definition span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (rₗ,rᵣ) ← (span(x->l) || span(x->r));
      if ¬rₗ then x->l := null;
      if ¬rᵣ then x->r := null;
      return true;
    else return false;
}
```

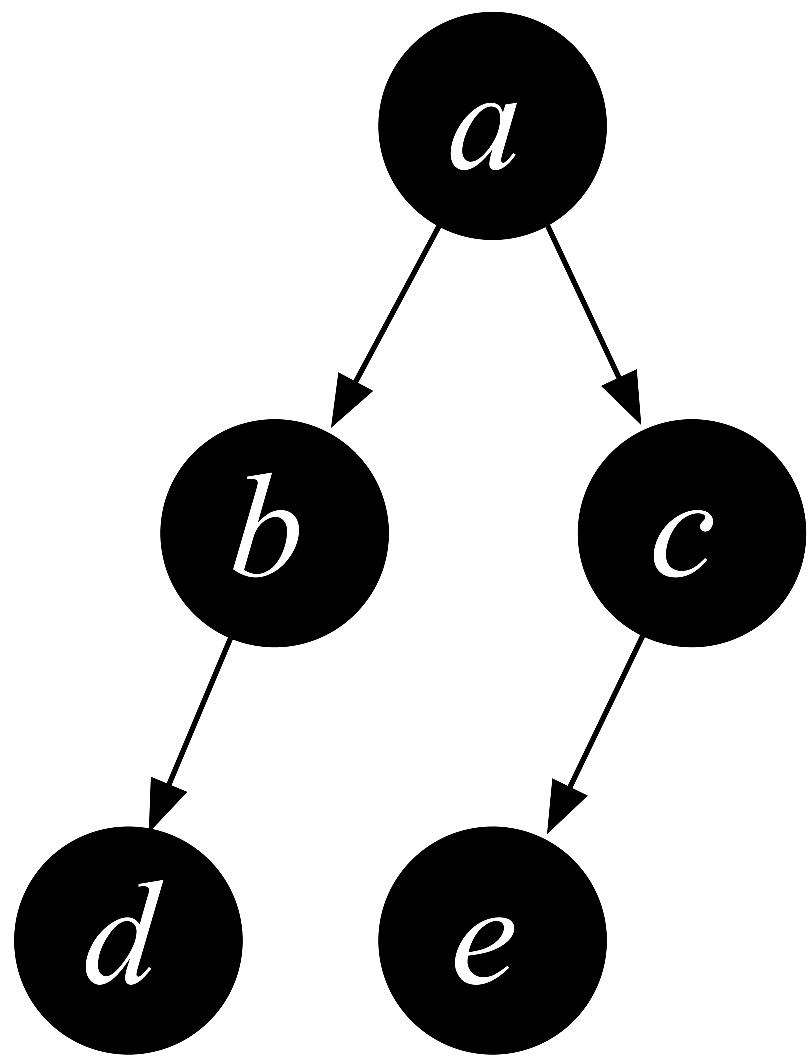

a
b
c
d
e


a
b
c
d
e


a
b
c
d
e


a
b
c
d
e
✔
✔
✔
✗
✔
✗


a
b
c
d
e

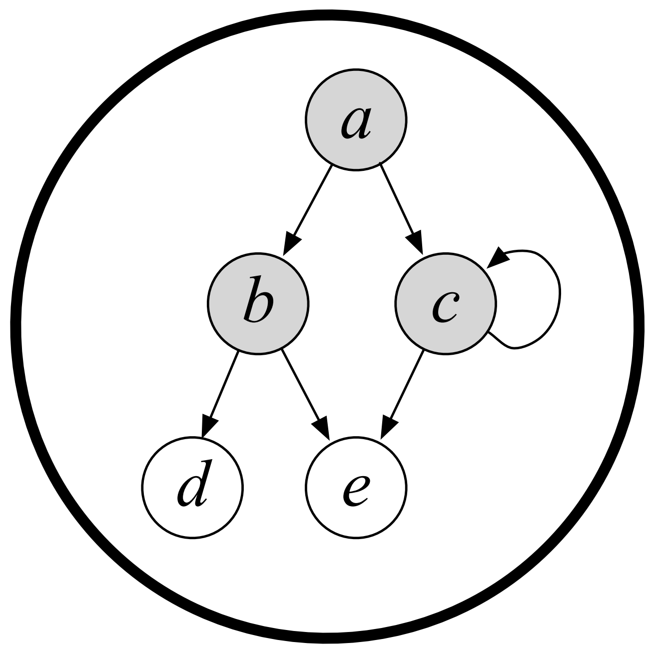

a
b
c
d
e

Верификация span

```
Program Definition span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (rₗ,rᵣ) ← (span(x->l) || span(x->r));
      if ¬rₗ then x->l := null;
      if ¬rᵣ then x->r := null;
      return true;
    else return false;
}
```

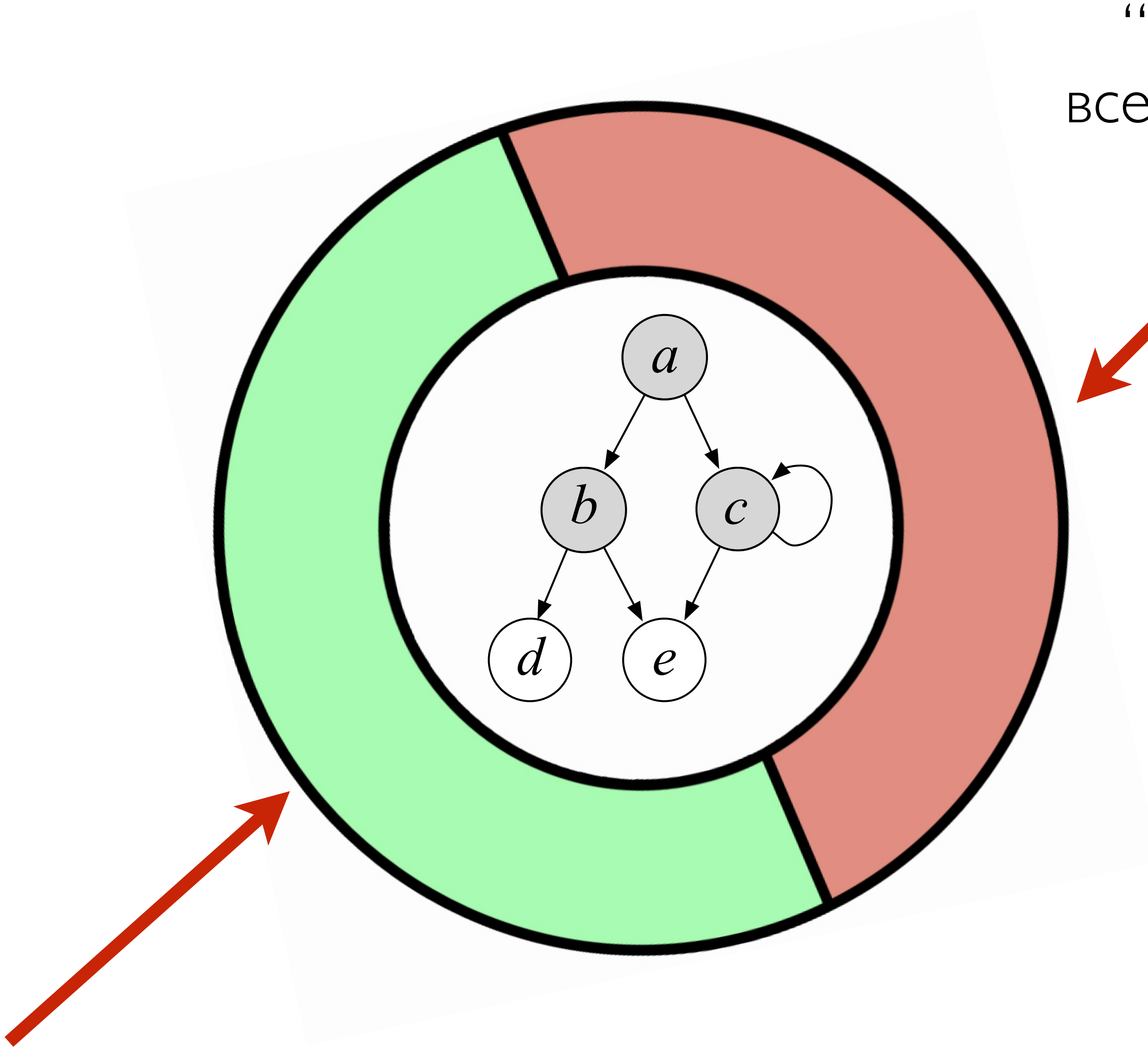
- Все достижимые вершины графа в итоге помечены
- Никакие “сторонние” процессы не изменяют граф
- Вызов из корневой вершины сделан одним “изначальным” потоком.

Модель разделяемой памяти



Модель разделяемой памяти

“Фиктивная” память
всех остальных потоков

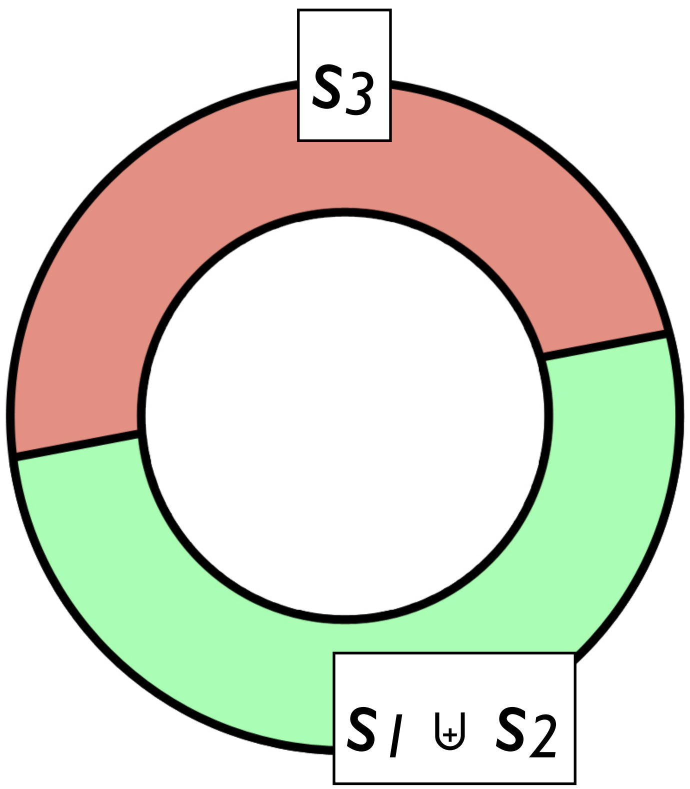


“Фиктивная” память
конкретного потока

Разделение фиктивной памяти

```
span(x)

span(x->l) || span(x->r)
```

Разделение фиктивной памяти

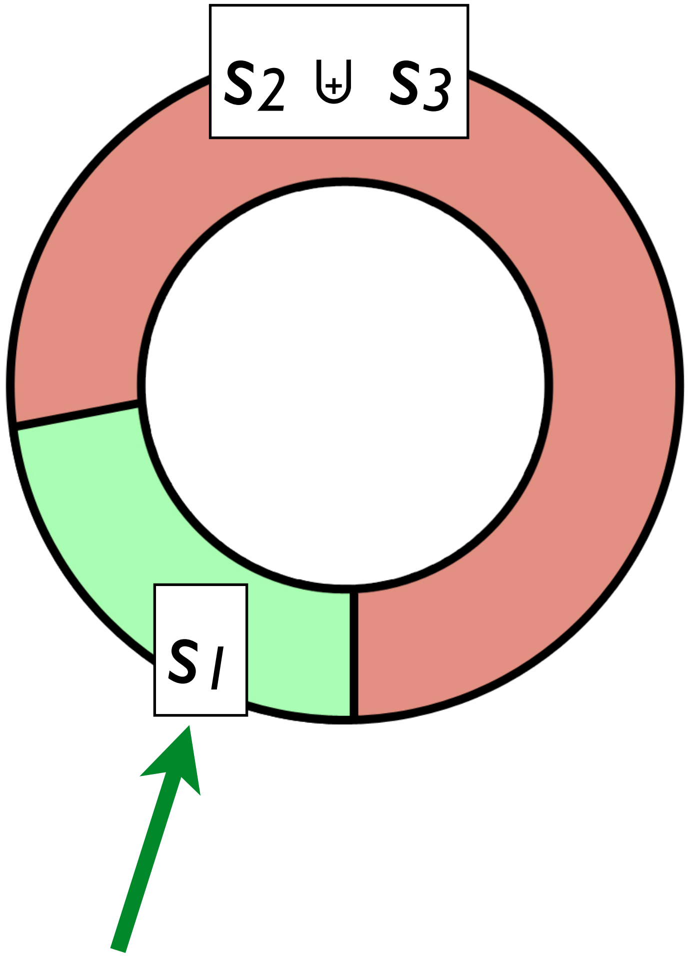


`span(x)`

$\{ s_1 \uplus s_2 \}$

`span(x->l)` $\Vert$ `span(x->r)`

Разделение фиктивной памяти



Вершины, "отданные" `span(x->l)`

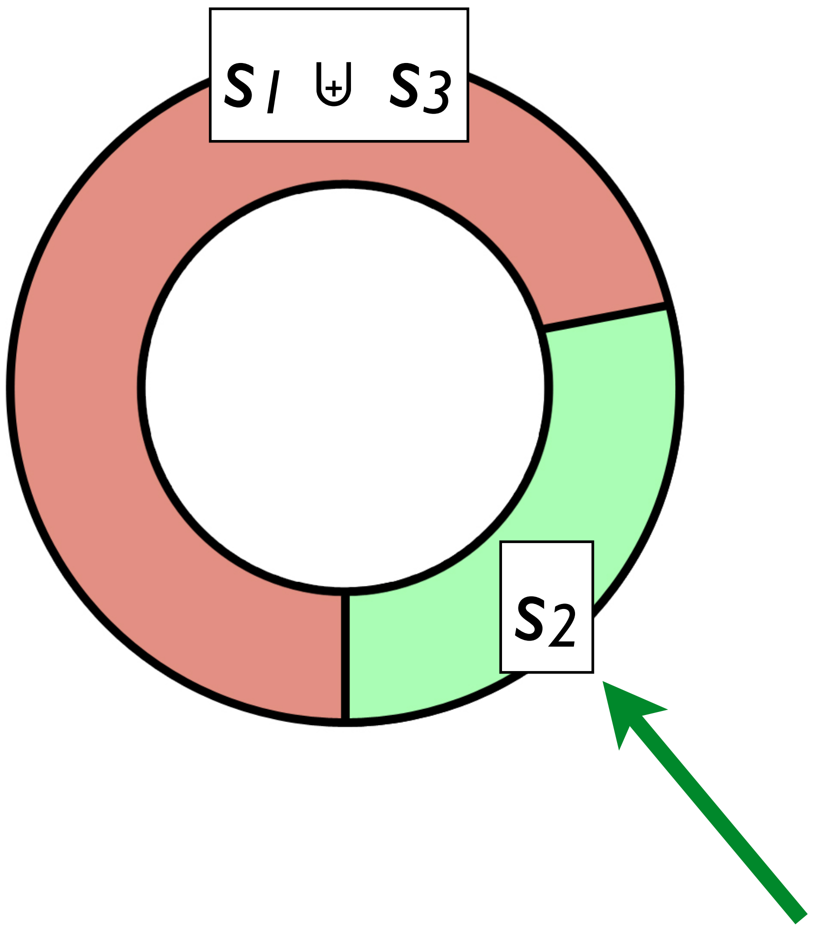
`span(x)`

$\{\ s_l \uplus s_2\ \}$

$\{\ s_l\ \}$

`span(x->l)` || `span(x->r)`

Разделение фиктивной памяти



`span(x)`

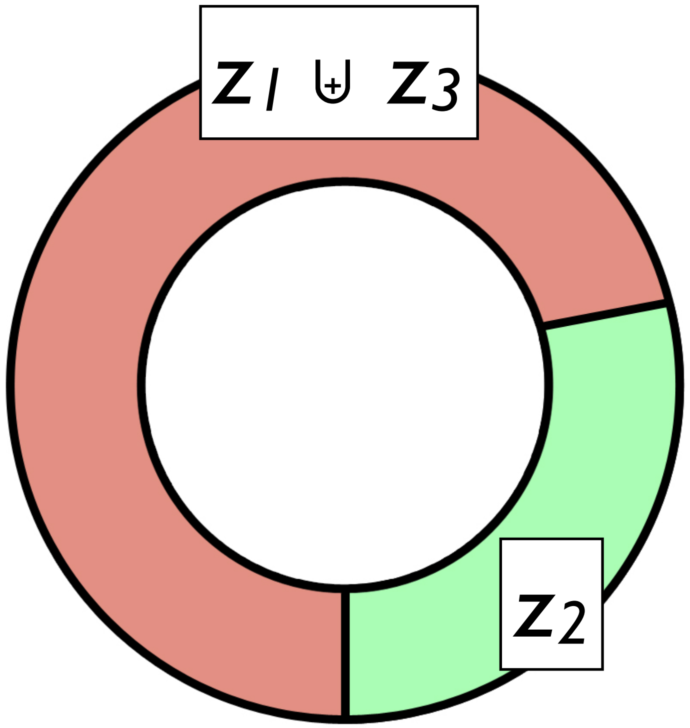
$\{\ s_1 \uplus s_2\ \}$

$\{\ s_1\ \}$ $\qquad\qquad$ $\{\ s_2\ \}$

`span(x->l)` || `span(x->r)`

Вершины, "отданные" `span(x->r)`

Разделение фиктивной памяти



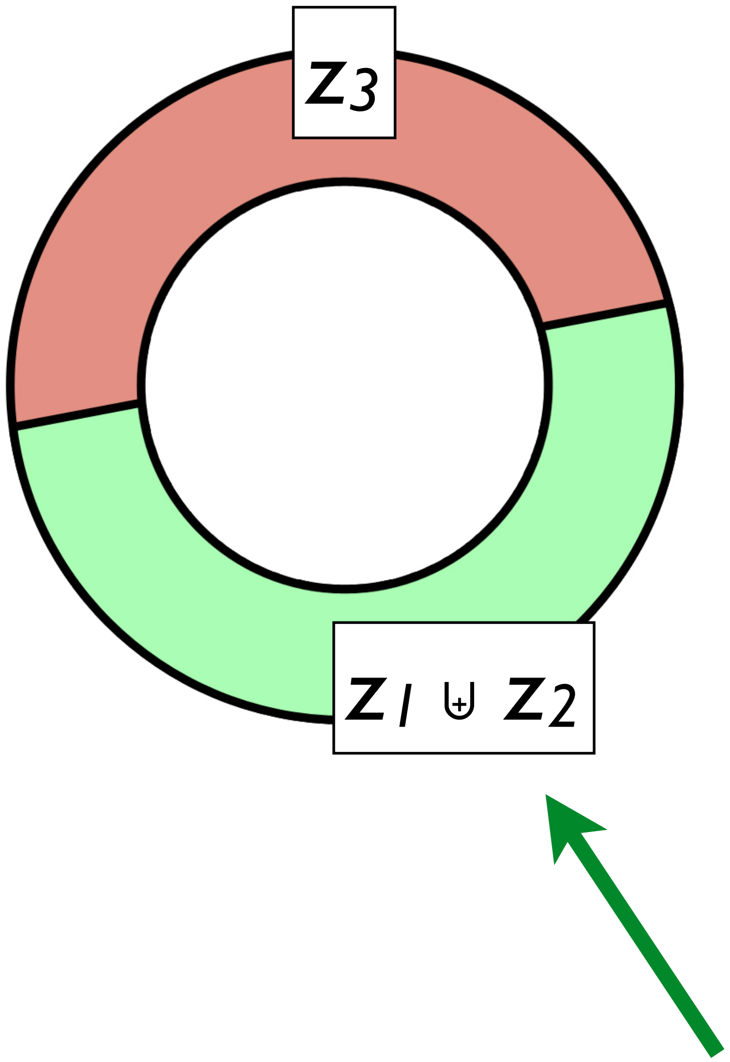
`span(x)`

$\{ s_1 \uplus s_2 \}$

$$\frac{\{ s_1 \}\ \texttt{span(x->l)}\ \{ z_1 \} \qquad \{ s_2 \}\ \texttt{span(x->r)}\ \{ z_2 \}}{} $$

$\{ s_1 \}$		$\{ s_2 \}$
`span(x->l)`	$\Vert$	`span(x->r)`
$\{ z_1 \}$		$\{ z_2 \}$

Разделение фиктивной памяти



Вершины, помеченные `span(x)`

`span(x)`

$\{\, s_1 \uplus s_2 \,\}$

$\{\, s_1 \,\}$ $\qquad$ $\{\, s_2 \,\}$

`span(x->l)` $\parallel$ `span(x->r)`

$\{\, z_1 \,\}$ $\qquad$ $\{\, z_2 \,\}$

$\{\, z_1 \uplus z_2 \,\}$

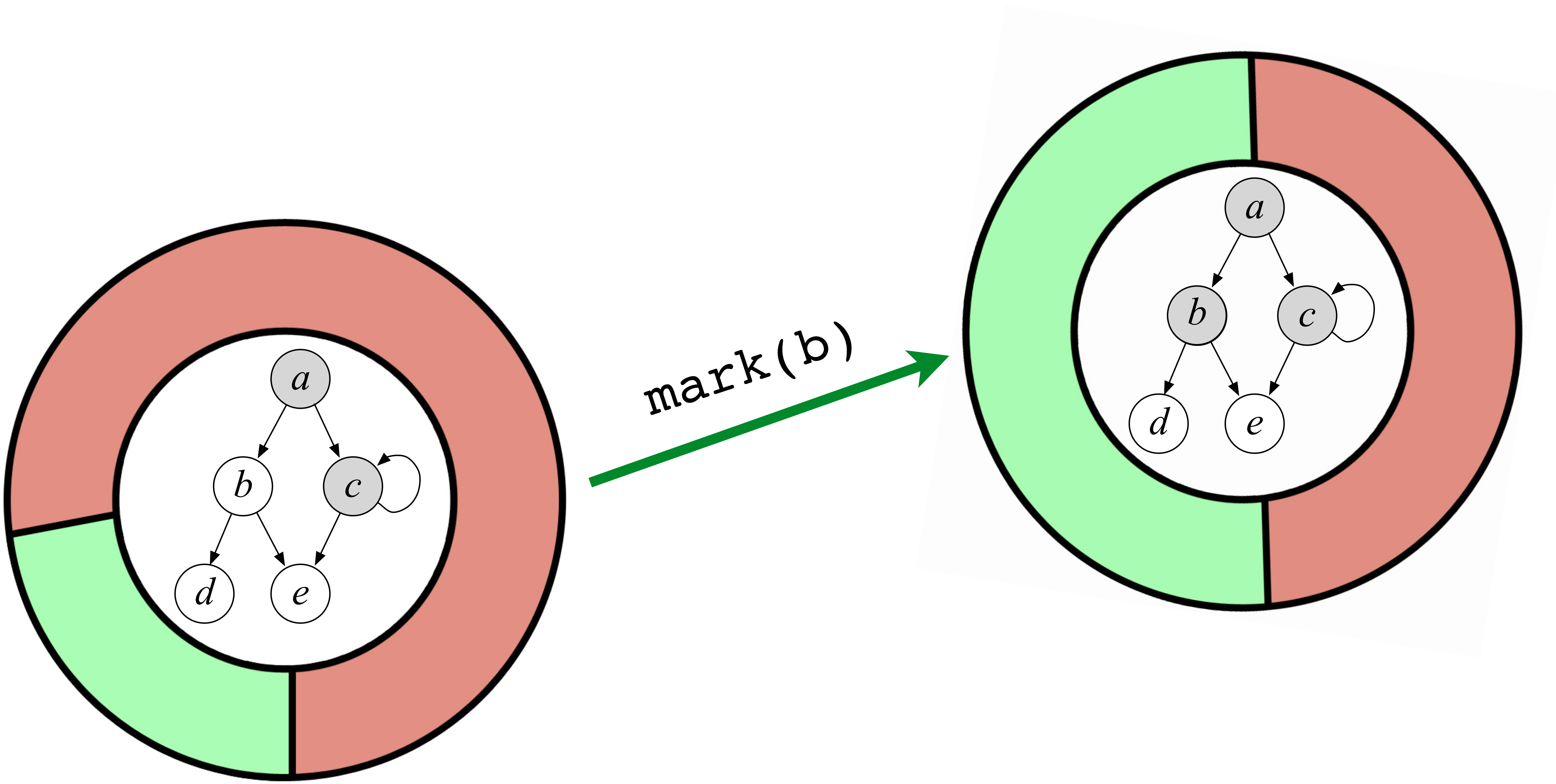
`span(x)`

Верификация span

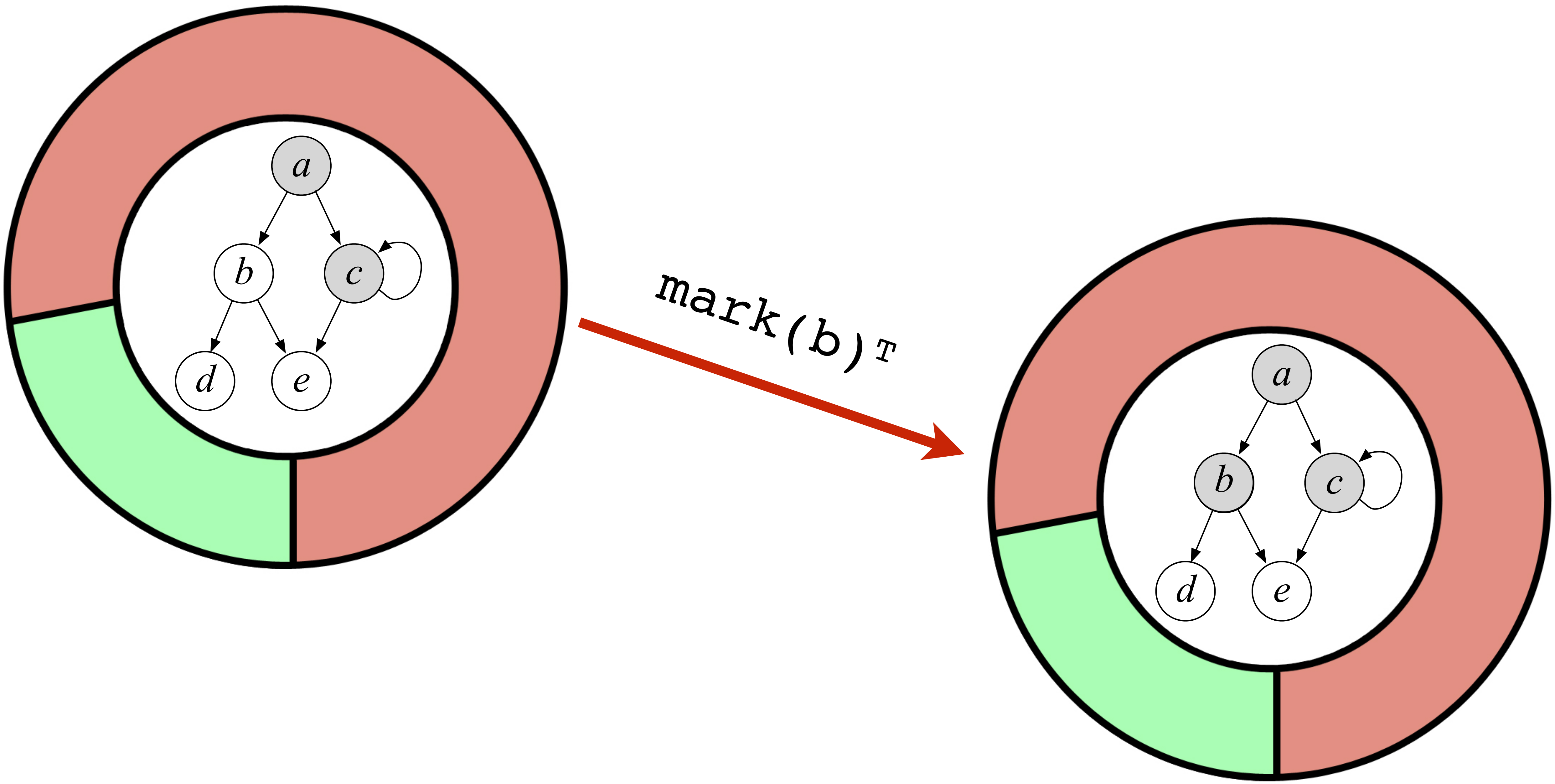
```
Program Definition span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (r_l,r_r) ← (span(x->l) || span(x->r));
      if ¬r_l then x->l := null;
      if ¬r_r then x->r := null;
      return true;
    else return false;
}
```

- Все достижимые вершины графа в итоге помечены
- Никакие “сторонние” процессы не изменяют граф
- Вызов из корневой вершины сделан одним “изначальным” потоком.

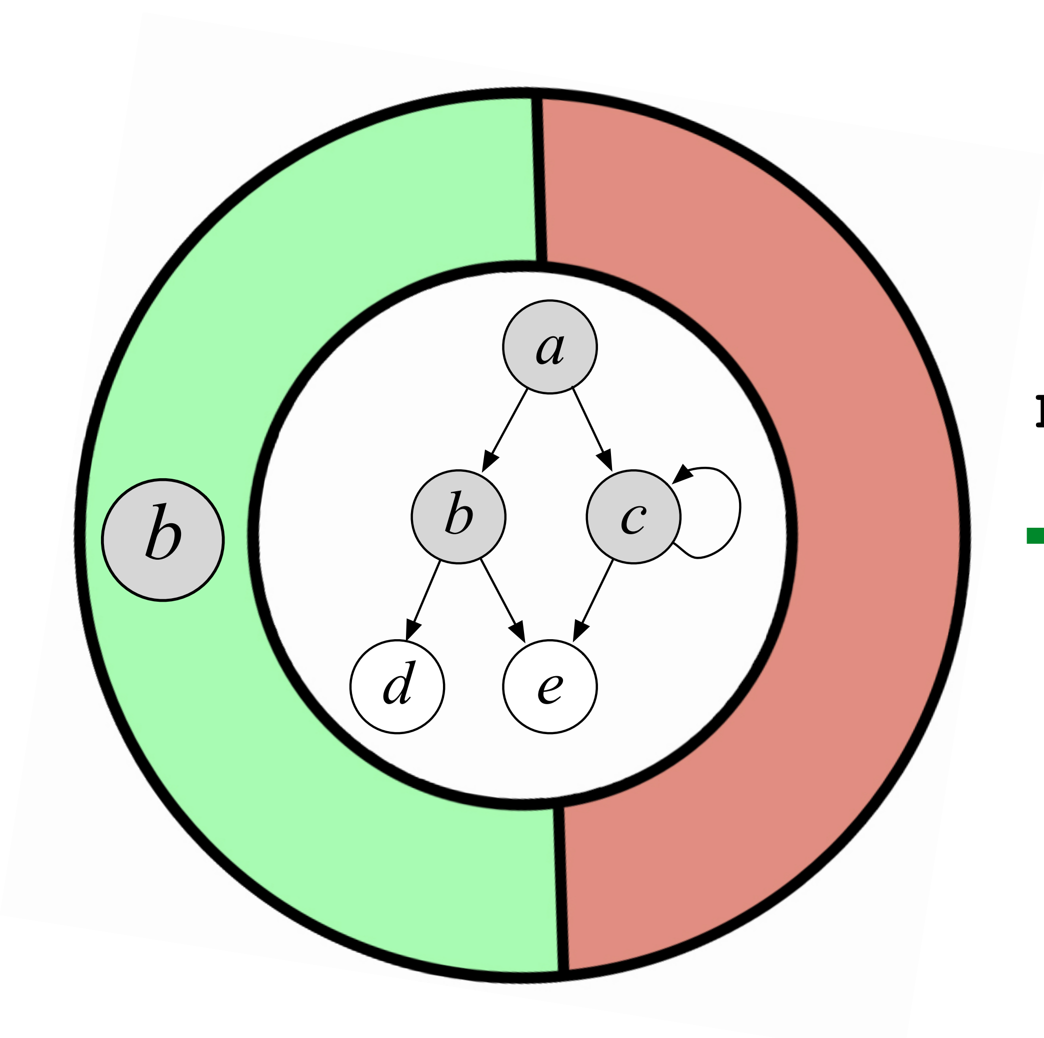
Соглашение 1: пометка вершины



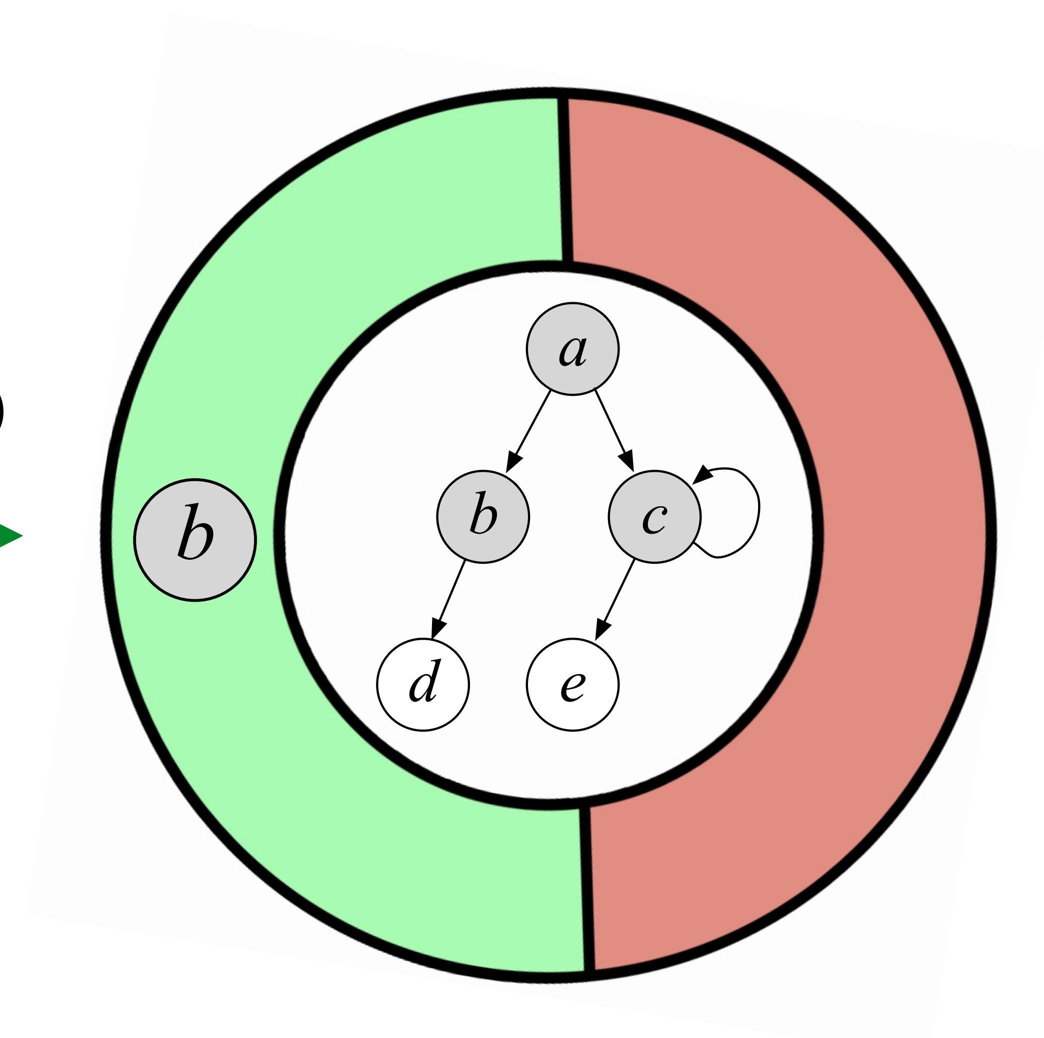
Соглашение 1: пометка вершины



Соглашение 2: удаление ребра



`nullify(b->r)`



```
Program Definition span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (r_l,r_r) ← (span(x->l) || span(x->r));
      if ¬r_l then x->l := null;
      if ¬r_r then x->r := null;
      return true;
    else return false;
}
```

```
Program Definition span : span_tp :=
 ffix (fun (loop : span_tp) (x : ptr) =>
   Do (if x == null then ret false else
       b <-- trymark x;
       if b then
         xl <-- read_child x Left;
         xr <-- read_child x Right;
         rs <-- par (loop xl) (loop xr);
         (if ~~rs.1 then nullify x Left else ret tt);;
         (if ~~rs.2 then nullify x Right else ret tt);;
         ret true
       else ret false)).
```

Зависимые типы
для многопоточности

λ calculus
A. Church (1930s)

Turing Machine
A. Turing (1936)

Simply typed λ calculus
A. Church (1940)

Fortran
J. Backus (1957)

Program Logics
R.W. Floyd, C.A.R. Hoare (1983)

ML
A. Milner (1973)

C
D. Richie (1972)

C++
B. Stroustrup (1983)

Haskell
S. Peyton-Jones et al. (1990)

Java
J. Gosling (1995)

Separation Logic
J. Reynolds (2002)

Hoare Type Theory
A. Nanevski (2006)

Facebook Infer
P.O'Hearn et al. (2013)

Concurrent Hoare Type Theory
A. Nanevski, I. Sergey (2013-15)

$$\{\ P\ \}\ c\ \{\ Q\ \}\ @R$$

$$\{\ P\ \}\ \texttt{span(x)}\ \{\ Q\ \}@R_{SpanTree}$$

$$\texttt{span(x) : span\_tp}\ (x, R_{SpanTree}, P\ , Q)$$

Тип/Спецификация

```
Program Definition span : span_tp :=
 ffix (fun (loop : span_tp) (x : ptr) =>
   Do (if x == null then ret false else
      b <-- trymark x;
      if b then
        xl <-- read_child x Left;
        xr <-- read_child x Right;
        rs <-- par (loop xl) (loop xr);
        (if ~~rs.1 then nullify x Left else ret tt);;
        (if ~~rs.2 then nullify x Right else ret tt);;
        ret true
      else ret false)).
```

Proof. (около 200 LOC) **Qed.**

стартовая вершина

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                exists (t : set ptr),
                  self s2 = self i ⊎ t ∧
                  tree g2 x t ∧
                  maximal g2 t ∧
                  front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

протокол взаимодействия

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

предусловие

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                exists (t : set ptr),
                  self s2 = self i ⊎ t ∧
                  tree g2 x t ∧
                  maximal g2 t ∧
                  front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

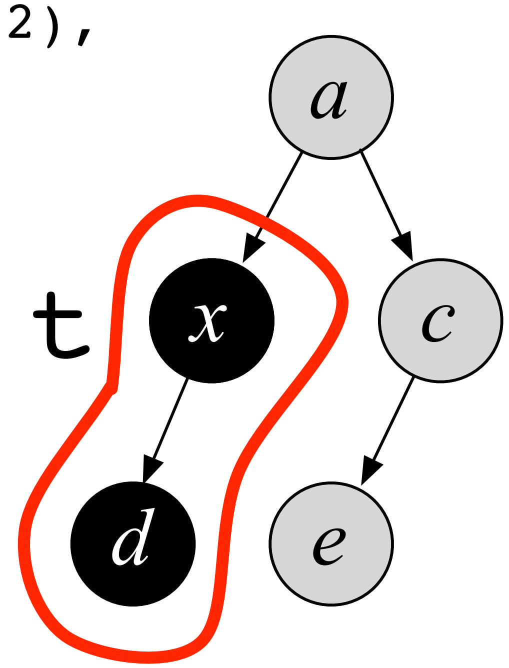
постусловие

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

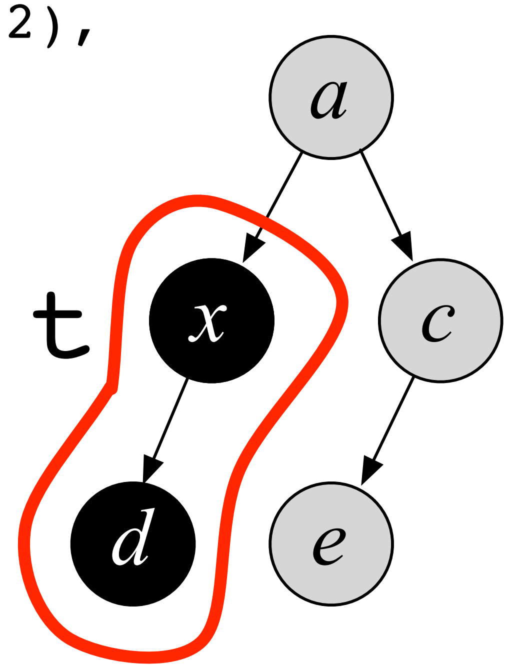
```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

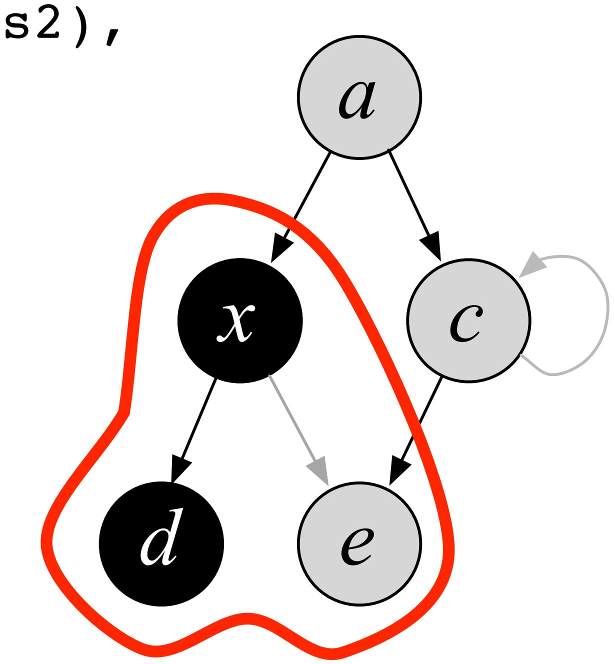
```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



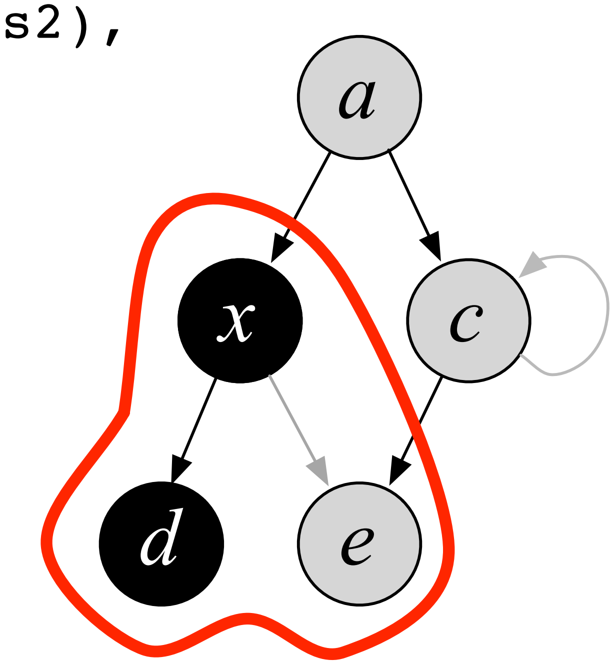
```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



```
Program Definition span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (rₗ,rᵣ) ← (span(x->l) || span(x->r));
      if ¬rₗ then x->l := null;
      if ¬rᵣ then x->r := null;
      return true;
    else return false;
}
```

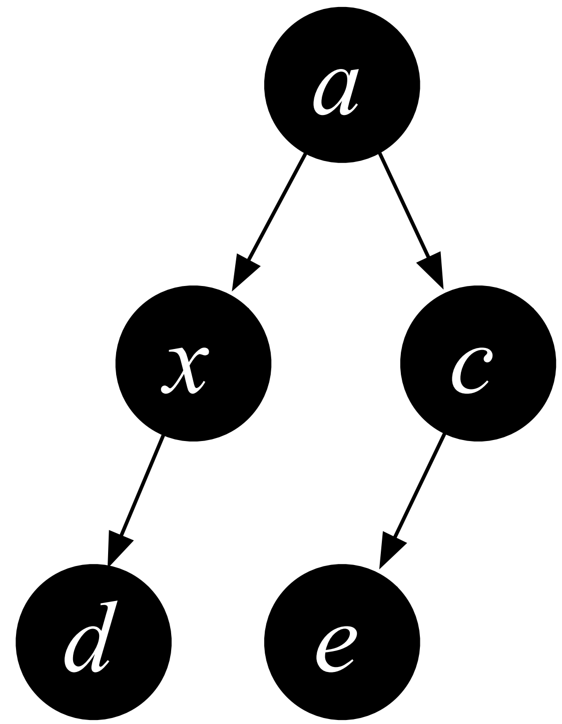
- Все достижимые вершины графа в итоге помечены
- Никакие "сторонние" процессы не изменяют граф
- Вызов из корневой вершины сделан одним "изначальным" потоком.

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊎ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊎ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



Предполагает эффект параллельных потоков

Изначально только 1 поток



следствия постусловия и связности графа

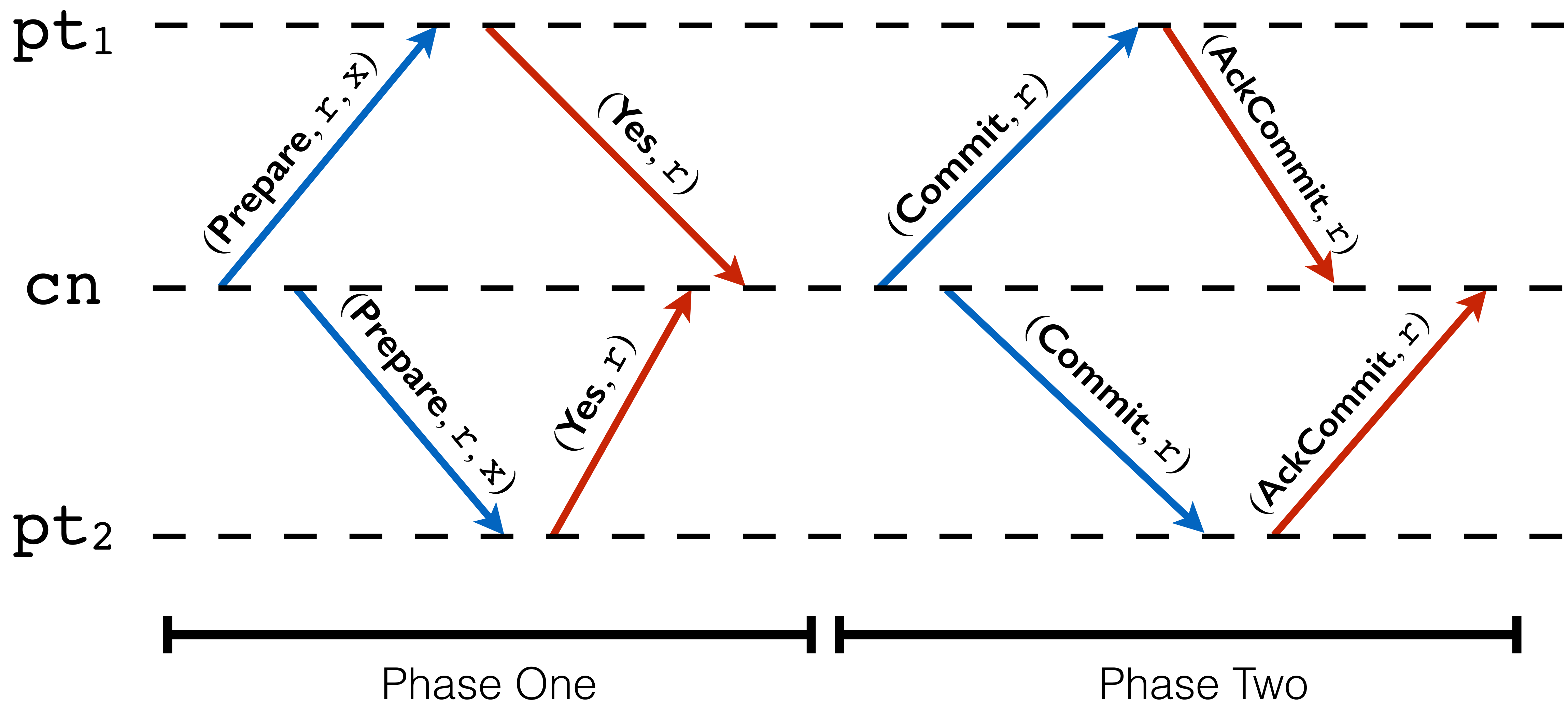
```
tree g2 a t            ∧ maximal g2 t ∧
front g1 t (self s2) ∧ t = self s2  ∧
is_root a g1           ∧ subgraph g1 g2
⇒ spanning t g1
```

Checkpoint 3: Зависимые типы для многопоточных программ

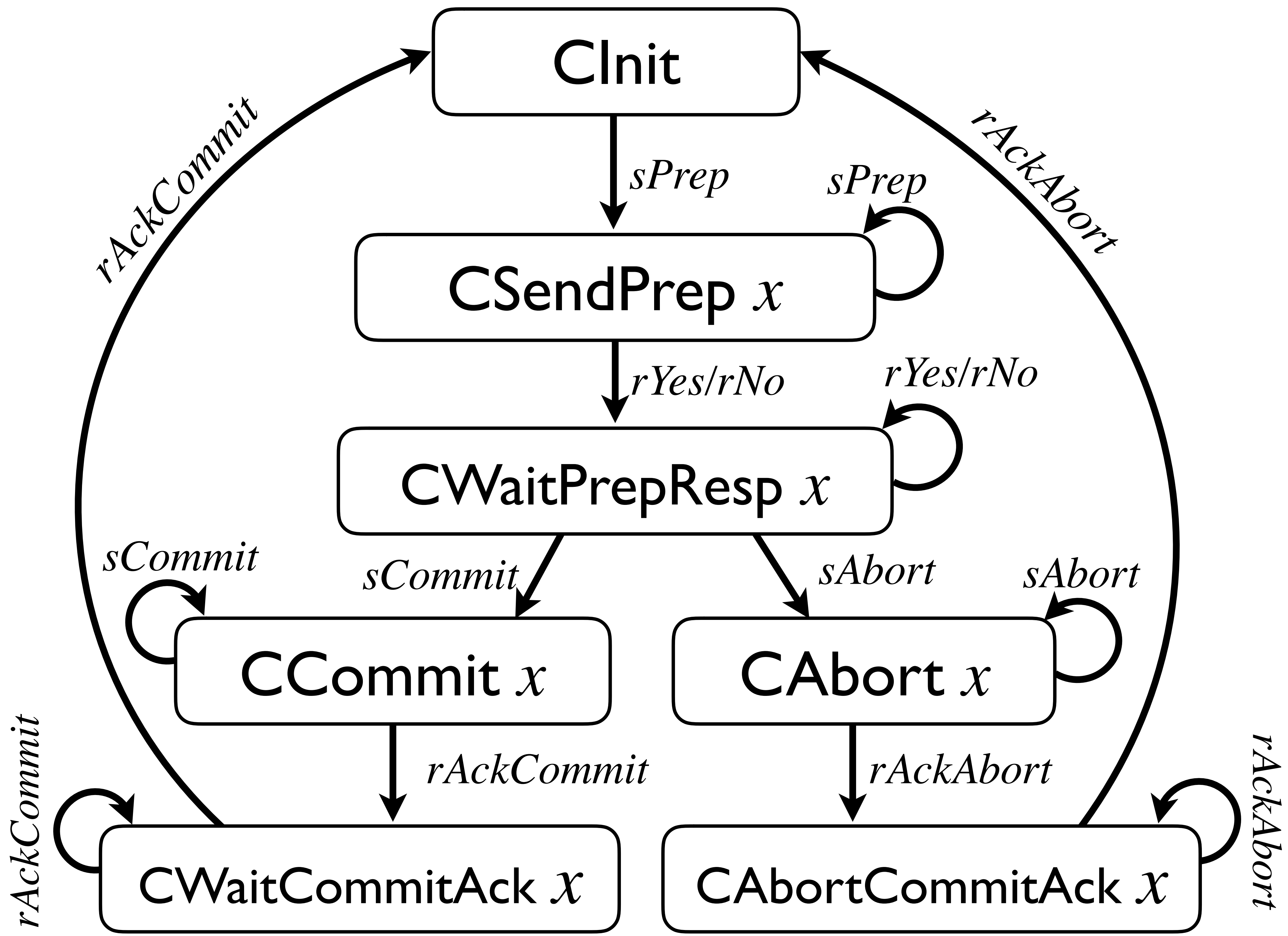
- Типы описывают *состояние разделенной памяти*;
- Типы также описывают *протокол* взаимодействия;
- Проверка типов (верификация) гарантирует, что *все потоки соблюдают протокол* и корректно делят “фиктивную” память.

Зависимые типы
для распределенных вычислений

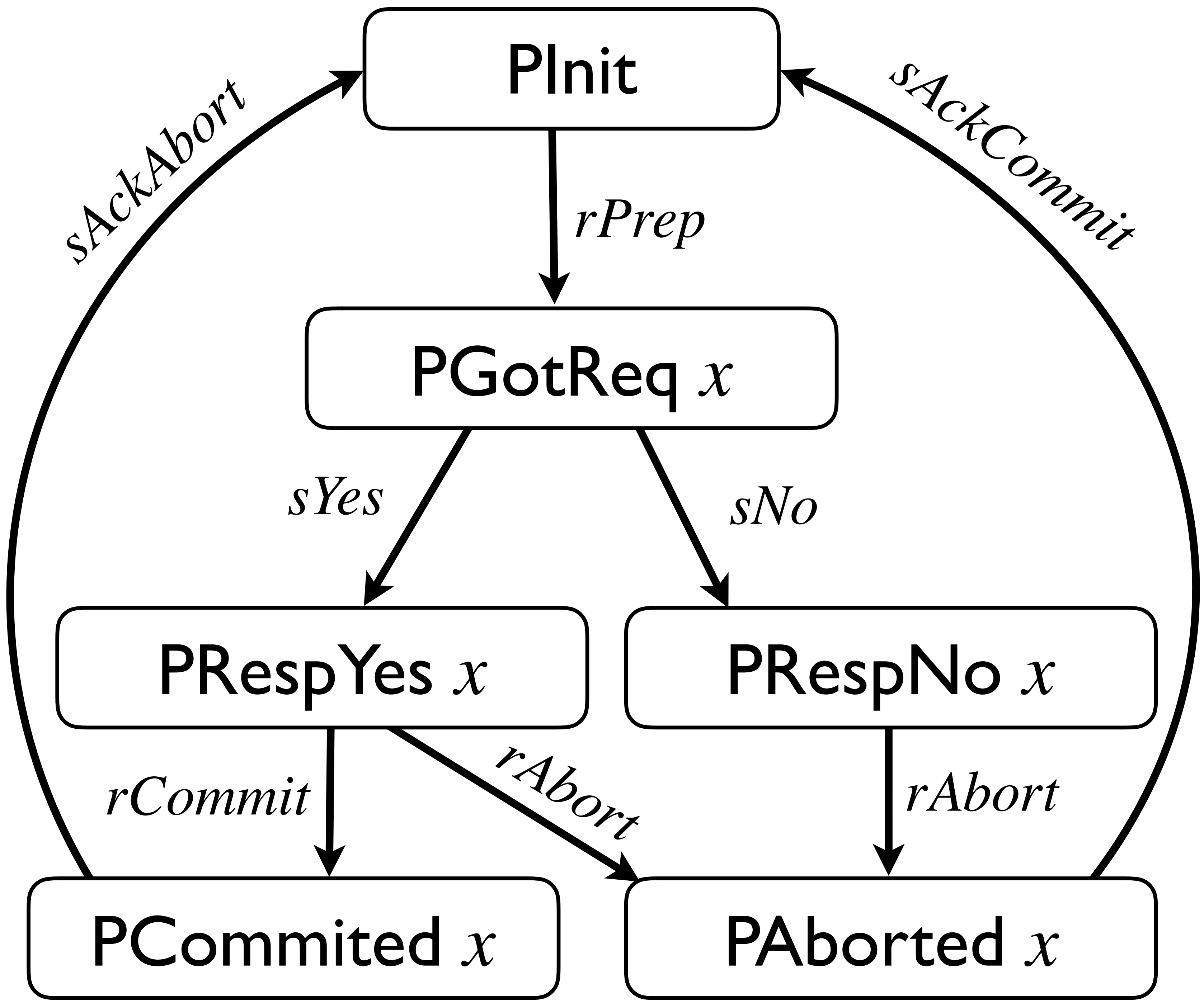
Two-Phase Commit Protocol



Состояния “Координатора”



Состояния “Участника"



Изменения в состоянии участника

Отправка сообщения

```
Definition p_send_step (r: round) (ps: PState) (log: Log)
           (commit: bool): round * PState * Log :=
  | PGotReq x ⇒ if commit then (r, PRespYes x, log)
                          else (r, PRespNo x, log)
  (* ... more cases depending on ps ... *)
  end.
```

Получение сообщения

```
Definition p_recv_step (r: round) (ps: PState)
           (log: Log) (tag: nat) (mbody: seq nat) :=
  | PRespYes x ⇒ if tag == Commit
      then (r + 1, PCommitted x, log ++ [(true, x)])
      else (r + 1, PAborted x, log ++ [(false, x)])
  (* ... more cases depending on ps, tag, mbody ... *)
  end.
```

λ calculus
A. Church (1930s)

Turing Machine
A. Turing (1936)

Simply typed λ calculus
A. Church (1940)

Fortran
J. Backus (1957)

Program Logics
R.W. Floyd, C.A.R. Hoare (1969)

ML
A. Milner (1973)

C
D. Richie (1972)

C++
B. Stroustrup (1983)

Haskell
S. Peyton-Jones et al. (1990)

Java
J. Gosling (1995)

Separation Logic
J. Reynolds (2002)

Hoare Type Theory
A. Nanevski (2006)

Facebook Infer
P.O'Hearn et al. (2013)

Concurrent Hoare Type Theory
A. Nanevski, I. Sergey (2013-15)

Distributed Hoare Types
I. Sergey (2017)

Реализация участника

```
Program Definition participant_round (commit: bool) :
    {(r: round) (log: Log)}, DHT [pt, W]

      (fun s ⇒  loc pt s = st ↦ (r, PInit) ⊎

                           lg ↦ log,

       fun _ s’ ⇒ ∃ (b: bool) (x: data),

                loc pt s’ = st ↦ (r+1, PInit) ⊎
                            lg ↦ (log ++ [(b, x)]))
 :=
    Do (r ← read_round;
        receive_prepare_req r;
        respond_to_req r commit;
        e ← receive_commit_or_abort r;
        send_ack e).
```

Инвариант системы 2PC

```
Lemma pt_log_agreement d r log pt :
  coh d → pt ∈ pts → Inv d →
  loc pt d = st ↦ (r, PInit) ⊎ lg ↦ log →
  ∀ pt' ps' log', pt' ∈ pts →
    loc pt' d = st ↦ (r, ps') ⊎ lg ↦ log' → log' = log.
```

Между раундами "истории" участников *совпадают*.

Участник как библиотека

```
Definition run_participant (choices: seq bool): DHT [pt, _]
      (fun i ⇒ i = init_state,
       fun _ m ⇒ ∃ r (results : seq bool) (stream: seq data),
           let log := zip results stream in
           loc pt m = st ↦ (r, PInit) ⊎
                      lg ↦ log ∧
            ∀ pt’ ps’ lg’, pt’ ∈ pts →
               loc pt’ m = st ↦ (r, ps’) ⊎
                           lg ↦ log’ → log = log’)
:=
   participant choices.
```

Checkpoint 4: Зависимые типы для распределённых приложений

- Типы описывают *протокол взаимодействия*, представленный системой переходов;
- Типы могут описывать состояние *многих* реплик;
- Проверка типов (верификация) гарантирует, что все участники соблюдают протокол, равно как и *сохранение инвариантов* системы.

В заключение

λ calculus
A. Church (1930s)

Turing Machine
A. Turing (1936)

Simply typed λ calculus
A. Church (1940)

Fortran
J. Backus (1957)

Program Logics
R.W. Floyd, C.A.R. Hoare (1969)

ML
A. Milner (1973)

C
D. Richie (1972)

C++
B. Stroustrup (1983)

Haskell
S. Peyton-Jones et al. (1990)

Java
J. Gosling (1995)

Separation Logic
J. Reynolds (2002)

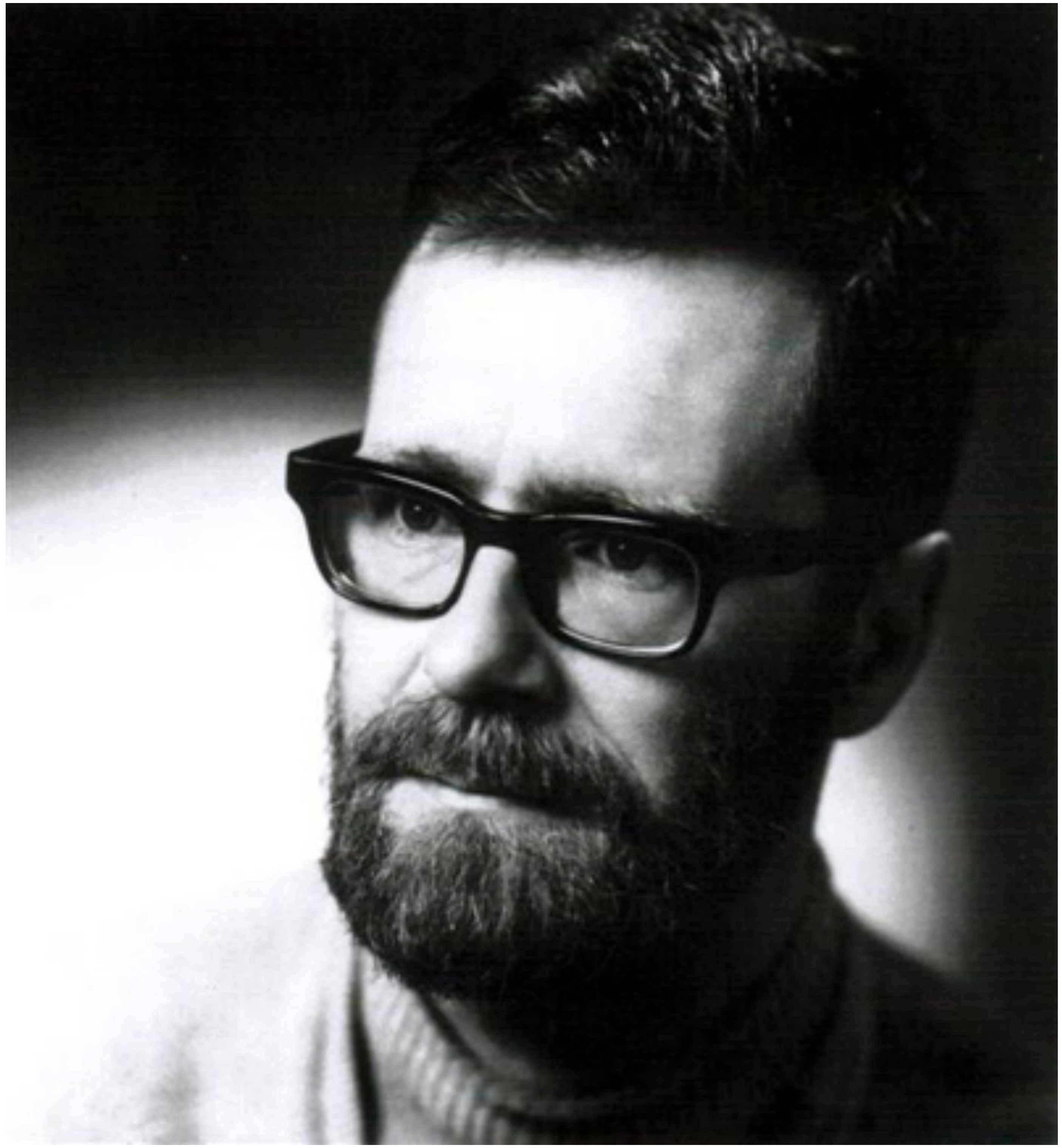
Hoare Type Theory
A. Nanevski (2006)

Facebook Infer
P.O'Hearn et al. (2013)

Concurrent Hoare Type Theory
A. Nanevski, I. Sergey (2013-15)

Distributed Hoare Types
I. Sergey (2017)

- Типизированные программы *не содержат ошибок*;
- Зависимые типы — отсутствие *нетривиальных* ошибок;
- Логики программ (**Program Logics**) — спецификация и верификация *императивного* кода;
- **Separation Logic + Dependent Types = Hoare Types** — типы для программ с *управлением памятью*;
- *Многопоточные* и *распределенные* вычисления: Hoare Types + расширенная модель состояния.



Edsger W. Dijkstra

On the cruelty of really teaching computing science

Edsger W. Dijkstra

We stress that the programmer's task is ***not just to write*** down a program, but that his main task is to ***give a formal proof*** that the program he proposes meets the ***formal functional specification***. […]

The rules of proof manipulation are ***so few and simple*** that very soon thereafter he makes the ***exciting discovery*** that he is beginning to master the use of ***a tool that, in all its simplicity, gives us a power*** that far surpasses his wildest dreams.

Спасибо за внимание!