leveraging rust types for program synthesis

Nadia Polikarpova

with Jonáš Fiala, Peter Müller, Shachar Itzhaky, and Ilya Sergey

WG2.8 2023

program synthesis with guarantees

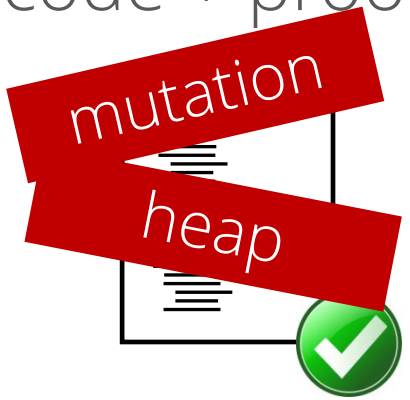
formal specification

code + proof

program synthesis for imperative code

formal specification

code + proof



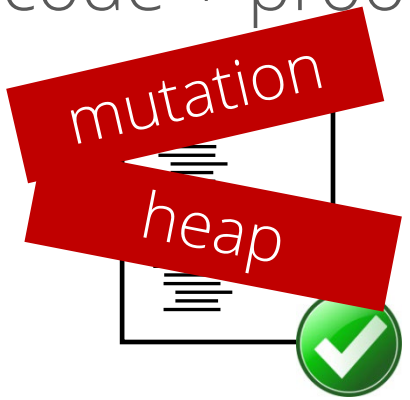
program synthesis for imperative code

separation logic

SuSLik

[Polikarpova & Sergey'19]

code + proof



but who wants to write separation logic?

program synthesis for ~~imperative code~~ rust

rust type

\+

functional spec

RusSOL

code + proof

this talk

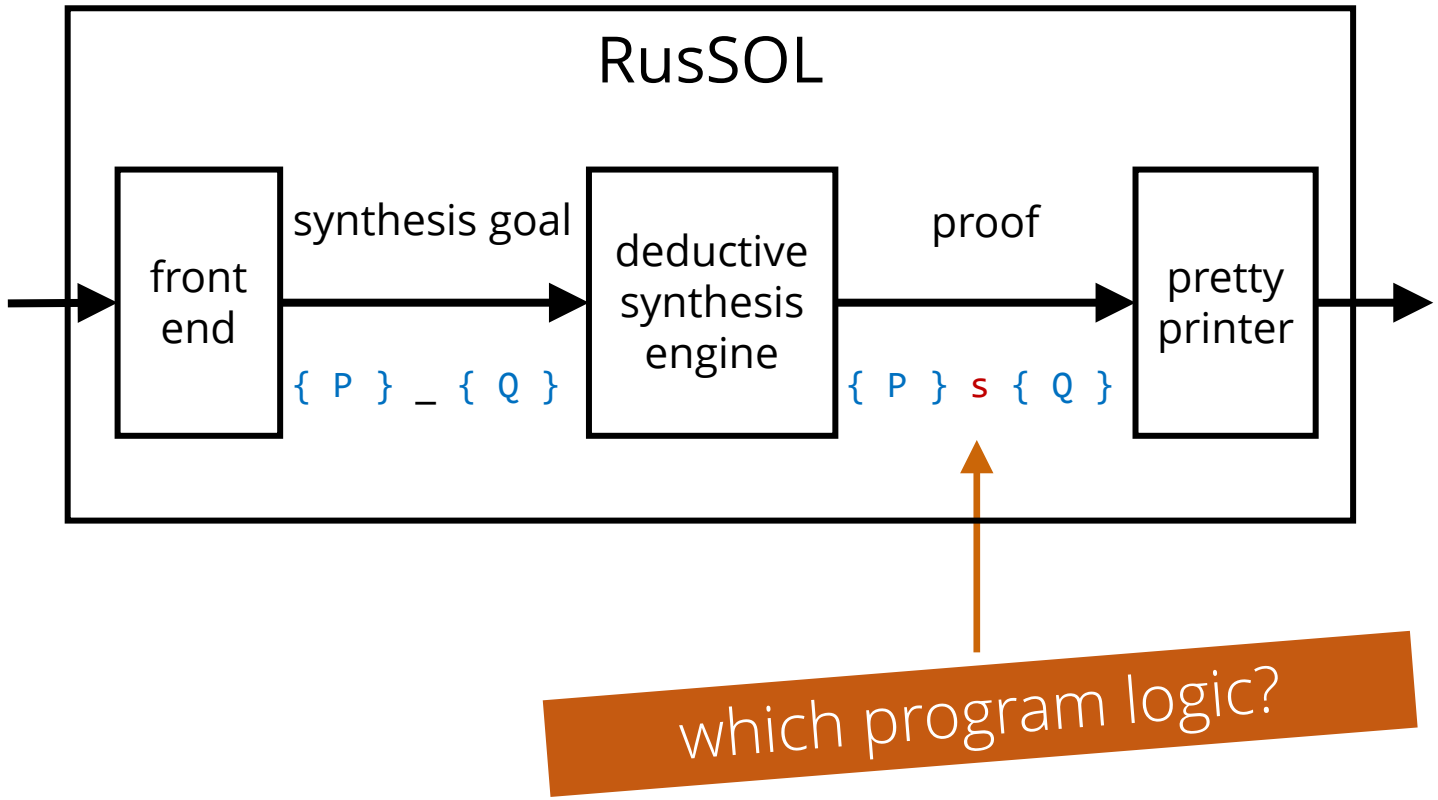
1. a taste of RusSOL
2. synthetic ownership logic (SOL)
3. references and borrowing

demo

RusSOL workflow

rust type +
functional spec

```
[#requires …]
[#ensures …]
fn target(x: T1…) -> T

[#pure]
fn f(x: T1…) -> T
…
```



rust program

```
fn target(x: T1…)
    -> T
{ s }
```

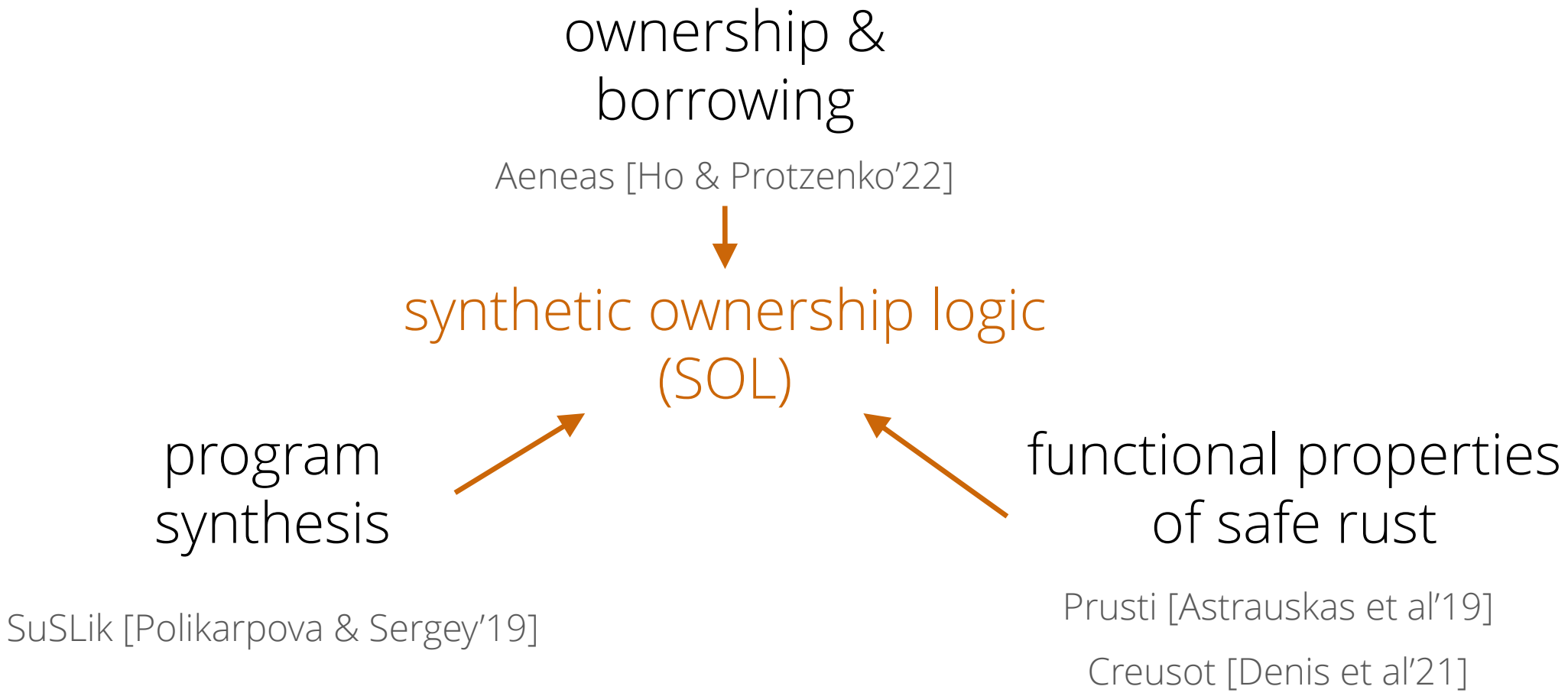
this talk

1. a taste of RusSOL

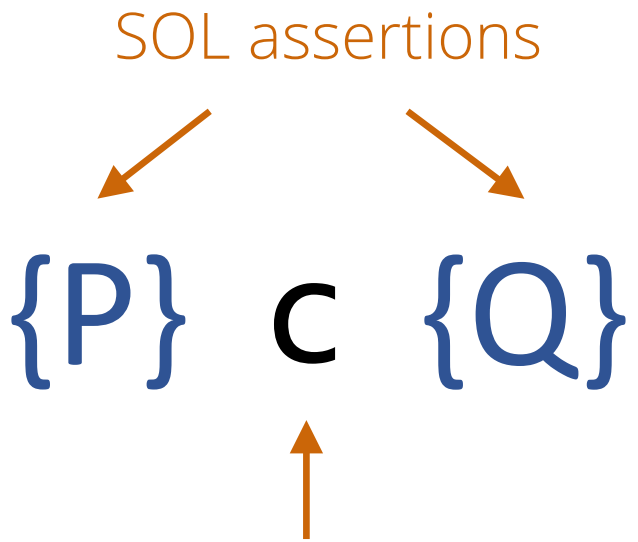
2. synthetic ownership logic (SOL)

3. references and borrowing

requirements



SOL triples



starting in a state that satisfies P
program **c** will execute safely*
and terminate in state that satisfies Q

SOL assertions

empty heap { emp }

SOL assertions

empty heap	{ emp }
variable binding	{ x : List }

SOL assertions

empty heap	{ emp }
variable binding	{ x : List }
separating conjunction	{ x : List * y : Box }

SOL rules

DROP

$$\frac{}{\{x : T\}\ \texttt{drop!}(x)\ \{emp\}}$$

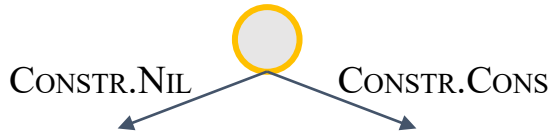
CONSTR.NIL

$$\frac{}{\{emp\}\ \textbf{let}\ x = \texttt{List::Nil}\ \{x : \texttt{List}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}


    todo!();


    // {result: List}
    result
 }
}
```

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```



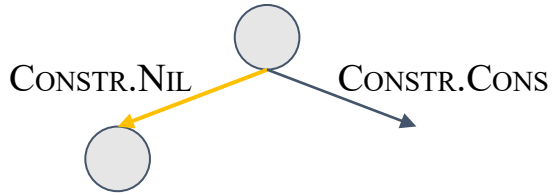
CONSTR.NIL

$$\frac{}{\{\mathsf{emp}\}\ \mathsf{let}\ \mathtt{x} = \mathtt{List::Nil}\ \{\mathtt{x}\colon \mathtt{List}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}

    todo!();

    // {}
    let result = List::Nil; // Const.Nil
    // {result: List}
    result
 }
}
```

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

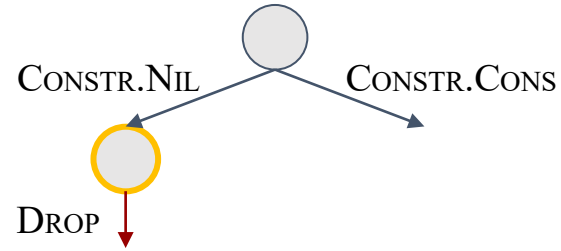


$$\frac{}{\{\mathsf{emp}\}\ \mathsf{let}\ \mathtt{x} = \mathtt{List::Nil}\ \{\mathtt{x}: \mathtt{List}\}}\ \textsc{Constr.Nil}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}

    todo!();

    // {}
    let result = List::Nil; // Const.Nil
    // {result: List}
    result
 }
}
```

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```



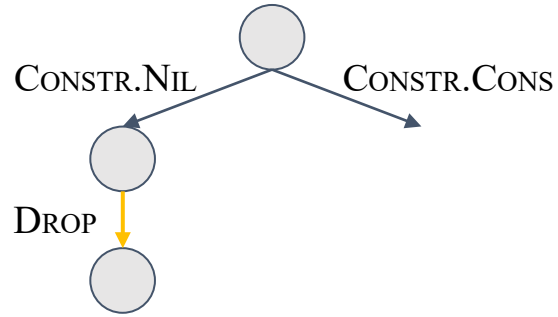
DROP

$$\frac{}{\{x : T\}\ \texttt{drop!}(x)\ \{\texttt{emp}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}
    drop(elem);                 // Drop
    // {}
    todo!();
    // {}
    let result = List::Nil; // Const.Nil
    // {result: List}
    result
 }
}
```

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

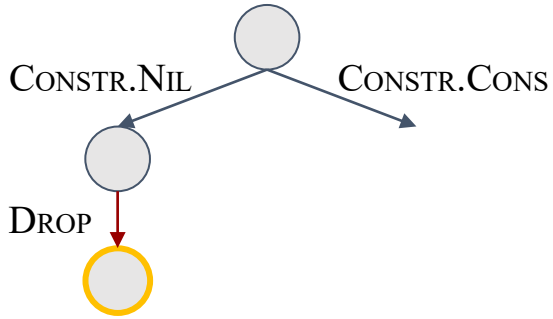


$$\frac{}{\{x : T\}\ \mathsf{drop!}(x)\ \{\mathsf{emp}\}}\ \textsc{Drop}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}
    drop(elem);              // Drop
    // {}
    todo!();
    // {}
    let result = List::Nil; // Const.Nil
    // {result: List}
    result
 }
}
```

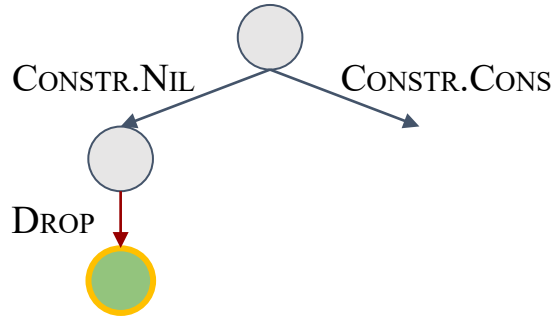
```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}
    drop(elem);              // Drop
    // {}

    // {}
    let result = List::Nil; // Const.Nil
    // {result: List}
    result
 }
}
```

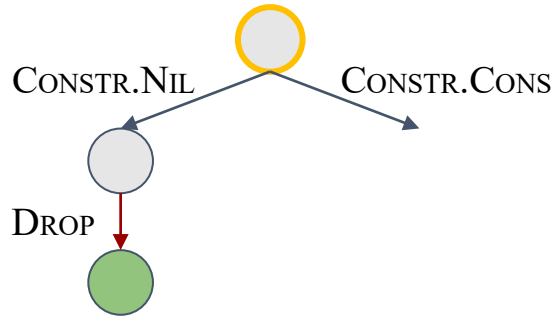
```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```



example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}



    todo!();



    // {result: List}
    result
 }
}
```

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```



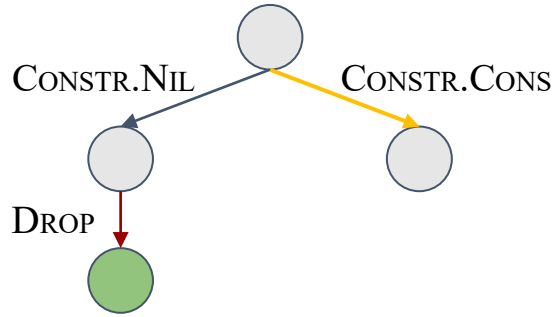
CONSTR.CONS

$$\frac{}{\left\{\begin{array}{l}\texttt{e: T} * \\ \texttt{n: Box}\end{array}\right\}\ \textbf{let}\ \texttt{x} = \texttt{List::Cons}\ \{\ \texttt{e}, \texttt{n}\ \}\ \{\texttt{x: List}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}


    todo!();


    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



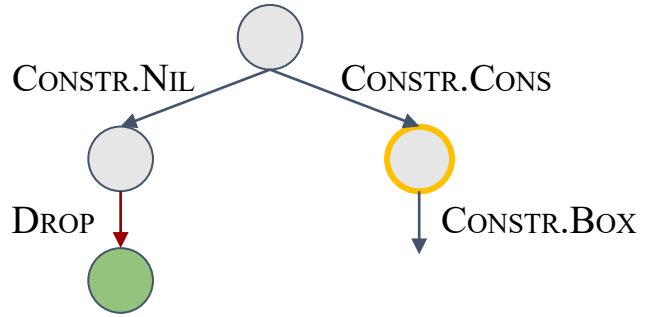
CONSTR.CONS

$$\frac{}{\left\{\begin{array}{l}\texttt{e: T *}\\ \texttt{n: Box}\end{array}\right\}\ \textbf{let}\ \texttt{x} = \texttt{List::Cons}\ \{\ \texttt{e, n}\ \}\ \{\texttt{x: List}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}


    todo!();


    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



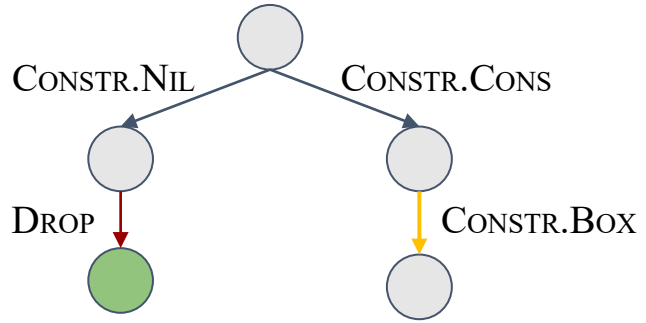
CONSTR.BOX

$$\frac{}{\{l : \mathtt{List}\}\ \mathtt{let}\ x = \mathtt{Box::new}(l)\ \{x : \mathtt{Box}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}

    todo!();

    // {elem: T * list: List}
    let next = Box::new(list);            // Const.Box
    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



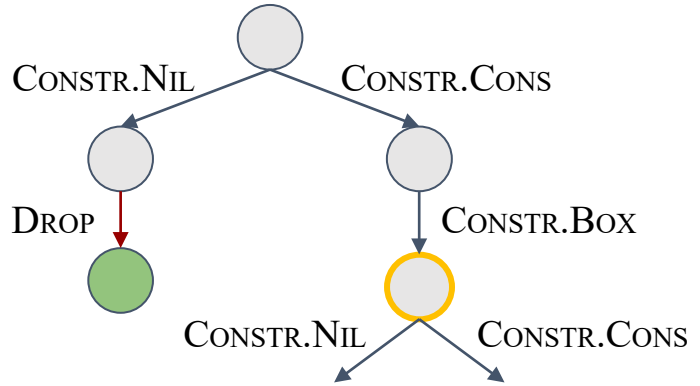
Constr.Box

$$\frac{}{\{l : \texttt{List}\}\ \texttt{let x = Box::new(l)}\ \{x : \texttt{Box}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}

    todo!();

    // {elem: T * list: List}
    let next = Box::new(list);           // Const.Box
    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```

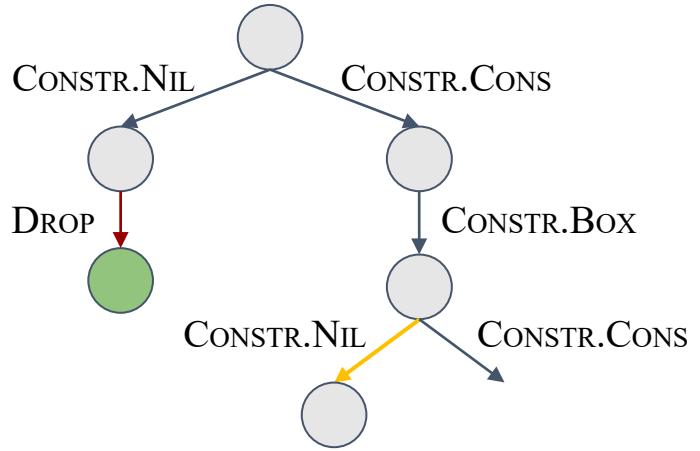


$$\frac{}{\{\mathsf{emp}\}\ \mathbf{let}\ \mathtt{x} = \mathtt{List::Nil}\ \{\mathtt{x}\colon \mathtt{List}\}}\ \textsc{Constr.Nil}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}
    todo!();
    // {elem: T}
    let list = List::Nil;                    // Const.Nil
    // {elem: T * list: List}
    let next = Box::new(list);               // Const.Box
    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



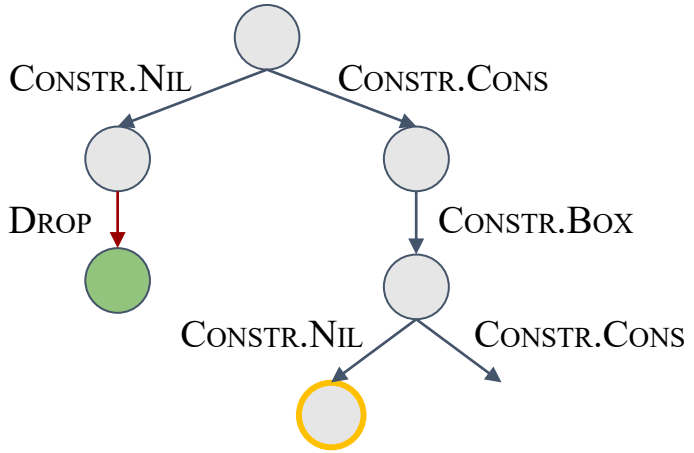
Constr.Nil

$$\frac{}{\{\text{emp}\}\ \texttt{let x = List::Nil}\ \{\texttt{x: List}\}}$$

example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}
    todo!();
    // {elem: T}
    let list = List::Nil;                    // Const.Nil
    // {elem: T * list: List}
    let next = Box::new(list);               // Const.Box
    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

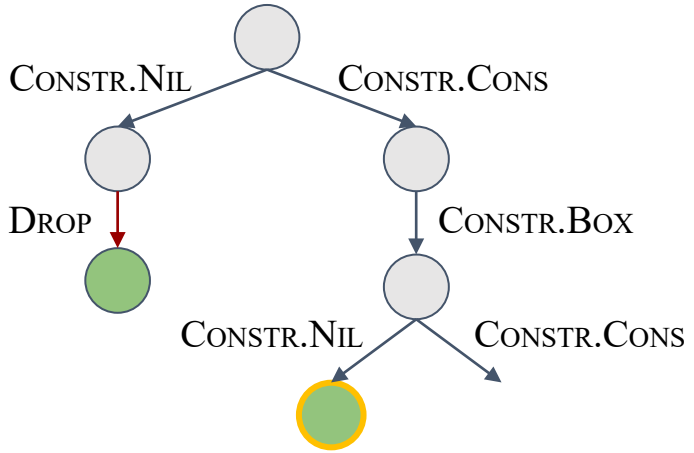
```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



example: singleton

```
impl<T> List<T> {
 pub fn singleton(elem: T) -> Self {
    // {elem: T}

    // {elem: T}
    let list = List::Nil;                  // Const.Nil
    // {elem: T * list: List}
    let next = Box::new(list);             // Const.Box
    // {elem: T * next: Box}
    let result = List::Cons{elem, next}; // Const.Cons
    // {result: List}
    result
 }
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```



SOL assertions: snapshots

empty heap	{ emp }
variable binding	{ x : List(*s*) }
separating conjunction	{ x : List(*s*) ∗ y : Box(*b*) }

SOL assertions: snapshots

empty heap	{ emp }
variable binding	{ x : List(s) }
separating conjunction	{ x : List(s) $*$ y : Box(b) }
pure formulas	{ $s.\delta$ = Cons $\wedge$ s.elem = 4 ; x : List(s) }

example: head

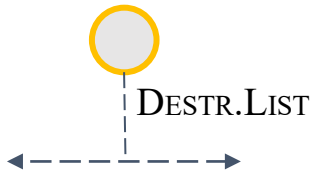
```
#[requires(self.len() > 0)]
pub fn head(self) -> T {
  // {s.len > 0 | self: List(s)}



  todo!();




  // {result: T(r)}
  result
}
```

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```

```
pub fn len(&self) -> usize {
  match self {
    List::Nil => 0,
    List::Cons { next, .. } =>
      1 + next.len(),
  }
}
```

example: head

```
pub fn head(self) -> T {
  // {s.len > 0 | self: List(s)}




  todo!();




  // {result: T}
  result
}
```



$$\textsc{Destr.List}\ \frac{\{l = \{\delta\colon \mathtt{Nil}, \mathtt{len}\colon 0\} \mid P\}\ c_0\ \{Q\} \qquad \{l = \ldots \mid \mathtt{e}\colon \mathtt{T}(v) * \mathtt{n}\colon \mathtt{Box}(w) * P\}\ c_1\ \{Q\}}{\{\mathtt{x}\colon \mathtt{List}(l) * P\}\ \mathbf{match}\ \mathtt{x}\ \left\{\begin{array}{c}\mathtt{Nil} \Rightarrow c_0 \\ \mathtt{Cons}\{\mathtt{e}, \mathtt{n}\} \Rightarrow c_1\end{array}\right\}\ \{Q\}}$$

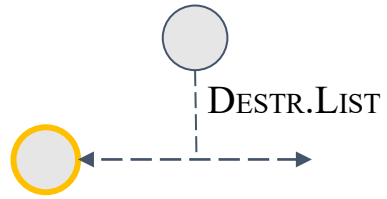
example: head

```
pub fn head(self) -> T {
  // {s.len > 0 | self: List(s)}
  match self {            // Destr.List
    List::Nil => ...



    List::Cons { elem, next } => ...



 }// {result: T}
  result
}
```

$$\textsc{Destr.List}\ \frac{\{l = \{\delta\colon \mathtt{Nil}, \mathtt{len}\colon 0\} \mid P\}\ c_0\ \{Q\} \qquad \{l = \ldots \mid \mathtt{e}\colon \mathsf{T}(v) * \mathtt{n}\colon \mathsf{Box}(w) * P\}\ c_1\ \{Q\}}{\{\mathtt{x}\colon \mathtt{List}(l) * P\}\ \textbf{match}\ \mathtt{x}\ \left\{\begin{array}{c}\mathtt{Nil} \Rightarrow c_0 \\ \mathtt{Cons}\{\mathtt{e},\mathtt{n}\} \Rightarrow c_1\end{array}\right\}\ \{Q\}}$$

example: head

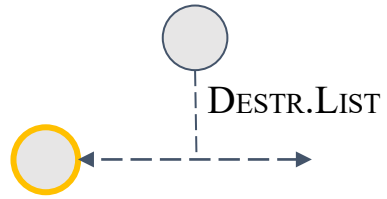
```
pub fn head(self) -> T {
  // {s.len > 0 | self: List(s)}
  match self {            // Destr.List
    List::Nil => {
      // {s.len > 0 ∧ s = {δ:Nil, len:0} | emp}

      todo!();
      // {result: T}
    }

    List::Cons { elem, next } => ...
  }// {result: T}
  result
}
```



$$\textsc{Destr.List}\ \frac{\{l = \{\delta\colon \texttt{Nil}, \texttt{len}\colon 0\} \mid P\}\ c_0\ \{Q\} \qquad \{l = \ldots \mid \texttt{e}\colon \texttt{T}(v) * \texttt{n}\colon \texttt{Box}(w) * P\}\ c_1\ \{Q\}}{\{\texttt{x}\colon \texttt{List}(l) * P\}\ \textbf{match}\ \texttt{x}\ \left\{\begin{array}{c}\texttt{Nil} \Rightarrow c_0 \\ \texttt{Cons}\{\texttt{e}, \texttt{n}\} \Rightarrow c_1\end{array}\right\}\ \{Q\}}$$

example: head

```
pub fn head(self) -> T {
  // {s.len > 0 | self: List(s)}
  match self {              // Destr.List
    List::Nil => {
      // {s.len > 0 ∧ s = {δ:Nil, len:0} | emp}
⇒     // {false | emp}
      todo!();
      // {result: T}
    }

    List::Cons { elem, next } => ...
 }// {result: T}
  result
}
```

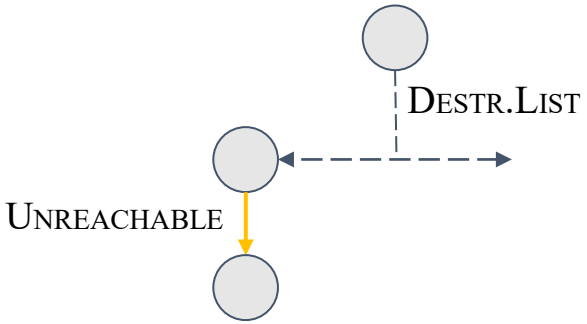


UNREACHABLE

$$\frac{}{\{\text{false} \mid P\}\ \texttt{unreachable!}()\ \{Q\}}$$

example: head

```
pub fn head(self) -> T {
  // {s.len > 0 | self: List(s)}
  match self {              // Destr.List
    List::Nil => {
      // {s.len > 0 ∧ s = {δ:Nil, len:0} | emp}
      // {false | emp}
      unreachable!()      // Unreachable
      // {result: T}
    }

    List::Cons { elem, next } => ...
  }// {result: T}
  result
}
```



UNREACHABLE

$$\frac{}{\{\text{false} \mid P\}\ \texttt{unreachable!}()\ \{Q\}}$$

this talk

1. a taste of RusSOL
2. synthetic ownership logic (SOL)
3. references and borrowing

SOL assertions: references

empty heap	{ emp }
variable binding	{ x : List }
separating conjunction	{ x : List ∗ y : Box }
reference	{ x $\overset{\text{`}\alpha}{\mapsto}$ List }

SOL assertions: references

empty heap — $\{ \text{emp} \}$

variable binding — $\{ x^{\theta} : \text{List} \}$

separating conjunction — $\{ x^{\theta} : \text{List} * y^{\theta} : \text{Box} \}$

reference — $\{ x^{\theta} \overset{\text{`}\alpha}{\mapsto} \text{List} \}$

blocking set

$\{ x^{\{\}} : \text{List} \}$

```
let y = &mut x;
```

$\{ x^{\{\text{`}\alpha\}} : \text{List} * y^{\{\}} \overset{\text{`}\alpha}{\mapsto} \text{List} \}$

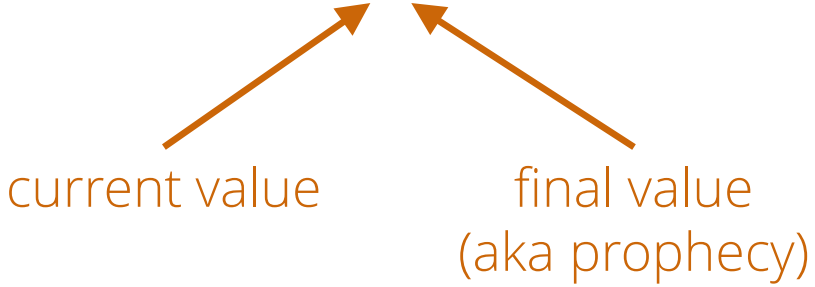
```
let z = &mut y;
```

$\{ x^{\{\text{`}\alpha\}} : \text{List} * y^{\{\text{`}\beta\}} \overset{\text{`}\alpha}{\mapsto} \text{List} * z^{\{\}} \overset{\text{`}\beta}{\mapsto} \text{List} \}$

"magic wand" in SL speak

SOL assertions: prophecies

empty heap	$\{ \mathsf{emp} \}$
variable binding	$\{ x^{\theta} : \mathsf{List} \}$
separating conjunction	$\{ x^{\theta} : \mathsf{List} * y^{\theta} : \mathsf{Box} \}$
reference	$\{ x^{\theta} \overset{`\alpha}{\mapsto} \mathsf{List}(s,\hat{s}) \}$



SOL rules for references

WRITE

$$\frac{}{\{\mathsf{x} \mapsto \mathsf{T}(v, \hat{x}) * \mathsf{y} : \mathsf{T}(c)\}\ *\mathsf{x} = \mathsf{y}\ \{\mathsf{x} \mapsto \mathsf{T}(c, \hat{x})\}}$$

DROPREF

$$\frac{}{\{\phi \mid \mathsf{x}^{\theta} \mapsto \mathsf{T}(c, \hat{x})\}\ \mathsf{drop!}(\mathsf{x})\ \{c = \hat{x} \wedge \phi \mid \mathsf{emp}\}}$$

drops the variable, not the referent

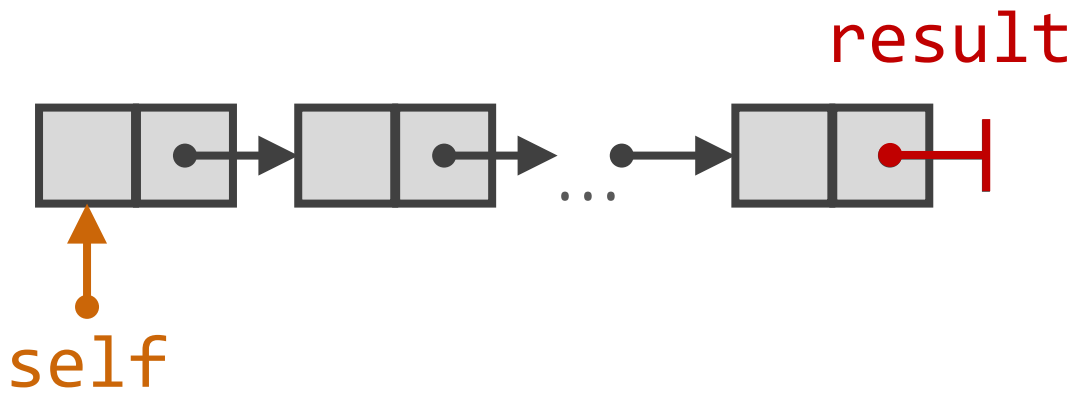
REBORROW

$$\frac{}{\{\mathsf{x} \mapsto \mathsf{T}(c, \hat{x})\}\ \mathbf{let}\ \mathsf{y} = \&\mathsf{mut}\ *\mathsf{x}\ \left\{ \begin{array}{l} \mathsf{y} \overset{'a}{\longmapsto} \mathsf{T}(c, \hat{y}) * \\ \mathsf{x}^{\{'a\}} \mapsto \mathsf{T}(\hat{y}, \hat{x}) \end{array} \right\}}$$

example: re-borrowing

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

```
pub fn last_mut(&mut self) -> & mut List<T>
```



example: re-borrowing

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

```
#[requires((^self).len() ==
                (*self).len() + (^result).len())]
pub fn last_mut(&mut self) -> & mut List<T> {
  // {self → List(s, ŝ)}



  todo!();



  // {ŝ.len == s.len + ř.len | result → List(_, ř)}
}
```

example: re-borrowing

```
enum List<T> {
   Nil,
   Cons { elem: T, next: Box<List<T>> },
}
```

```
pub fn last_mut(&mut self) -> & mut List<T> {
  match self {
    List::Nil => ...
      // {s.len = 0 | self → List(s, ŝ)}

      todo!();



      // {ŝ.len == s.len + ř.len | result → List(_, ř)}
    List::Cons { elem, next } => ...
  }
}
```

example: re-borrowing

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

```
pub fn last_mut(&mut self) -> &mut List<T> {
  match self {
    List::Nil => ...
      // {s.len = 0 | self → List(s, ŝ)}
      let result = &mut self;           // Reborrow
      // {s.len = 0 | result → List(s, ř) * self{'α} → List(ř, ŝ)}
      todo!();

      // {ŝ.len == s.len + ř.len | result → List(_, ř)}
    List::Cons { elem, next } => ...
  }
}
```

ReBorrow

$$\frac{}{\{x \mapsto T(c, \hat{x})\}\ \texttt{let y = \&mut *x}\ \left\{\begin{array}{l} y \overset{'a}{\longmapsto} T(c, \hat{y}) * \\ x^{\{'a\}} \mapsto T(\hat{y}, \hat{x}) \end{array}\right\}$$

example: re-borrowing

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

```
pub fn last_mut(&mut self) -> & mut List<T> {
  match self {
    List::Nil => ...
      // {s.len = 0 | self → List(s, ŝ)}
      let result = &mut self;              // Reborrow
      // {s.len = 0 | result → List(s, ř) * self{'α} → List(ř, ŝ)}
      drop!(self);                         // DropRef
      // {s.len = 0 ∧ ŝ == ř     | result → List(s, ř)}
      // {ŝ.len == s.len + ř.len | result → List(_, ř)}
    List::Cons { elem, next } => ...
  }
}
```

⇒

DROPREF

$$\frac{}{\left\{\phi \mid x^{\theta} \mapsto \mathsf{T}(c, \hat{x})\right\}\ \texttt{drop!}(x)\ \{c = \hat{x} \wedge \phi \mid \mathsf{emp}\}}$$

program synthesis for rust

rust type

\+

functional spec

RusSOL

code + proof