Program Verification using Separation Logic

Lecture 2:
Simple Automatic Verifier

Hongseok Yang (Queen Mary, Univ. of London)

What's missing to build an automatic verifier?

```
$\{\mathsf{ls}\ (x,0)\}$
$y = 0;$
INV :  $\mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)$
while $(x \neq 0)$ {
  $\{x{\neq}0 \wedge \mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)\}$
  $\{\exists x'.\, \mathsf{ls}\ (y,0) * (x \mapsto x') * \mathsf{ls}\ (x',0)\}$
  $t = x;$
  $\{\exists x'.\, t{=}x \wedge \mathsf{ls}\ (y,0) * (x \mapsto x') * \mathsf{ls}\ (x',0)\}$
  $\{\exists x'.\, \mathsf{ls}\ (y,0) * (t \mapsto x') * \mathsf{ls}\ (x',0)\}$
  $x = *t;$
  $\{\exists x'.\, x{=}x' \wedge \mathsf{ls}\ (y,0) * (t \mapsto x') * \mathsf{ls}\ (x',0)\}$
  $\{\mathsf{ls}\ (y,0) * (t \mapsto x) * \mathsf{ls}\ (x,0)\}$
  $*t = y;$
  $\{\mathsf{ls}\ (y,0) * (t \mapsto y) * \mathsf{ls}\ (x,0)\}$
  $y = t$
  $\{\exists y'.\, y{=}t \wedge \mathsf{ls}\ (y',0) * (t \mapsto y') * \mathsf{ls}\ (x,0)\}$
  $\{\mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)\}$
}
$\{x{=}0 \wedge \mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)\}$
$\{\mathsf{ls}\ (y,0)\}$
```

What's missing to build an automatic verifier?

```
$\{\mathsf{ls}\,(x,0)\}$
 $y = 0;$
 $\mathsf{INV}:\ \mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)$
 while $(x \neq 0)$ {
   $\{x{\neq}0 \wedge \mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)\}$
   $\{\exists x'.\,\mathsf{ls}\,(y,0) * (x \mapsto x') * \mathsf{ls}\,(x',0)\}$
    $t = x;$
   $\{\exists x'.\,t{=}x \wedge \mathsf{ls}\,(y,0) * (x \mapsto x') * \mathsf{ls}\,(x',0)\}$
   $\{\exists x'.\,\mathsf{ls}\,(y,0) * (t \mapsto x') * \mathsf{ls}\,(x',0)\}$
    $x = *t;$
   $\{\exists x'.\,x{=}x' \wedge \mathsf{ls}\,(y,0) * (t \mapsto x') * \mathsf{ls}\,(x',0)\}$
   $\{\mathsf{ls}\,(y,0) * (t \mapsto x) * \mathsf{ls}\,(x,0)\}$
    $*t = y;$
   $\{\mathsf{ls}\,(y,0) * (t \mapsto y) * \mathsf{ls}\,(x,0)\}$
    $y = t$
   $\{\exists y'.\,y{=}t \wedge \mathsf{ls}\,(y',0) * (t \mapsto y') * \mathsf{ls}\,(x,0)\}$
   $\{\mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)\}$
 }
$\{x{=}0 \wedge \mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)\}$
$\{\mathsf{ls}\,(y,0)\}$
```

1. Infer a loop invariant.

What's missing to build an automatic verifier?

```
$\{\mathsf{ls}\ (x, 0)\}$
 $y = 0;$
 $\mathsf{INV}:\ \ \mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)$
 while $(x \neq 0)$ {
   $\{x{\neq}0 \wedge \mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)\}$
   $\{\exists x'.\, \mathsf{ls}\ (y, 0) * (x \mapsto x') * \mathsf{ls}\ (x', 0)\}$
    $t = x;$
   $\{\exists x'.\, t{=}x \wedge \mathsf{ls}\ (y, 0) * (x \mapsto x') * \mathsf{ls}\ (x', 0)\}$
   $\{\exists x'.\, \mathsf{ls}\ (y, 0) * (t \mapsto x') * \mathsf{ls}\ (x', 0)\}$
    $x = *t;$
   $\{\exists x'.\, x{=}x' \wedge \mathsf{ls}\ (y, 0) * (t \mapsto x') * \mathsf{ls}\ (x', 0)\}$
   $\{\mathsf{ls}\ (y, 0) * (t \mapsto x) * \mathsf{ls}\ (x, 0)\}$
    $*t = y;$
   $\{\mathsf{ls}\ (y, 0) * (t \mapsto y) * \mathsf{ls}\ (x, 0)\}$
    $y = t$
   $\{\exists y'.\, y{=}t \wedge \mathsf{ls}\ (y', 0) * (t \mapsto y') * \mathsf{ls}\ (x, 0)\}$
   $\{\mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)\}$
 }
$\{x{=}0 \wedge \mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)\}$
$\{\mathsf{ls}\ (y, 0)\}$
```

1. Infer a loop invariant.
2. Massage assertions using Consequence.

   a) exposing pointsto.

What's missing to build an automatic verifier?

```
$\{\mathsf{ls}\ (x, 0)\}$
 $y = 0;$
 $\mathsf{INV}:\ \ \mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)$
 $\mathtt{while}\ (x \neq 0)\ \{$
   $\{x{\neq}0 \wedge \mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)\}$
   $\{\exists x'.\, \mathsf{ls}\ (y, 0) * (x \mapsto x') * \mathsf{ls}\ (x', 0)\}$
    $t = x;$
   $\{\exists x'.\, t{=}x \wedge \mathsf{ls}\ (y, 0) * (x \mapsto x') * \mathsf{ls}\ (x', 0)\}$
   $\{\exists x'.\, \mathsf{ls}\ (y, 0) * (t \mapsto x') * \mathsf{ls}\ (x', 0)\}$
    $x = *t;$
   $\{\exists x'.\, x{=}x' \wedge \mathsf{ls}\ (y, 0) * (t \mapsto x') * \mathsf{ls}\ (x', 0)\}$
   $\{\mathsf{ls}\ (y, 0) * (t \mapsto x) * \mathsf{ls}\ (x, 0)\}$
    $*t = y;$
   $\{\mathsf{ls}\ (y, 0) * (t \mapsto y) * \mathsf{ls}\ (x, 0)\}$
    $y = t$
   $\{\exists y'.\, y{=}t \wedge \mathsf{ls}\ (y', 0) * (t \mapsto y') * \mathsf{ls}\ (x, 0)\}$
   $\{\mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)\}$
 $\}$
$\{x{=}0 \wedge \mathsf{ls}\ (y, 0) * \mathsf{ls}\ (x, 0)\}$
$\{\mathsf{ls}\ (y, 0)\}$
```

1. Infer a loop invariant.

2. Massage assertions using Consequence.

a) exposing pointsto.

b) removing equality.

```
$\{\mathsf{ls}\,(x,0)\}$
 $y = 0;$
 INV : $\mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)$
 while $(x \neq 0)$ {
   $\{x{\neq}0 \wedge \mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)\}$
   $\{\exists x'.\, \mathsf{ls}\,(y,0) * (x \mapsto x') * \mathsf{ls}\,(x',0)\}$
    $t = x;$
   $\{\exists x'.\, t{=}x \wedge \mathsf{ls}\,(y,0) * (x \mapsto x') * \mathsf{ls}\,(x',0)\}$
   $\{\exists x'.\, \mathsf{ls}\,(y,0) * (t \mapsto x') * \mathsf{ls}\,(x',0)\}$
    $x = *t;$
   $\{\exists x'.\, x{=}x' \wedge \mathsf{ls}\,(y,0) * (t \mapsto x') * \mathsf{ls}\,(x',0)\}$
   $\{\mathsf{ls}\,(y,0) * (t \mapsto x) * \mathsf{ls}\,(x,0)\}$
    $*t = y;$
   $\{\mathsf{ls}\,(y,0) * (t \mapsto y) * \mathsf{ls}\,(x,0)\}$
    $y = t$
   $\{\exists y'.\, y{=}t \wedge \mathsf{ls}\,(y',0) * (t \mapsto y') * \mathsf{ls}\,(x,0)\}$
   $\{\mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)\}$
 }
$\{x{=}0 \wedge \mathsf{ls}\,(y,0) * \mathsf{ls}\,(x,0)\}$
$\{\mathsf{ls}\,(y,0)\}$
```

What's missing to build an automatic verifier?

1. Infer a loop invariant.
2. Massage assertions using Consequence.

a) exposing pointsto.

b) removing equality.

c) removing emp.

What's missing to build an automatic verifier?

$\{\mathsf{ls}\ (x,0)\}$

$y = 0;$

$\mathrm{INV}:\ \mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)$

$\texttt{while}\ (x \neq 0)\ \{$

$\quad \{x{\neq}0 \wedge \mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)\}$

$\quad \{\exists x'.\, \mathsf{ls}\ (y,0) * (x \mapsto x') * \mathsf{ls}\ (x',0)\}$

$\quad t = x;$

$\quad \{\exists x'.\, t{=}x$ $)\}$

$\quad \{\exists x'.\, \mathsf{ls}\ (y$

$\quad x = *t;$

$\quad \{\exists x'.\, x{=}x$ $0)\}$

$\quad \{\mathsf{ls}\ (y,0) *$

$\quad *t = y;$

$\quad \{\mathsf{ls}\ (y,0) * (t \mapsto y) * \mathsf{ls}\ (x,0)\}$

$\quad y = t$

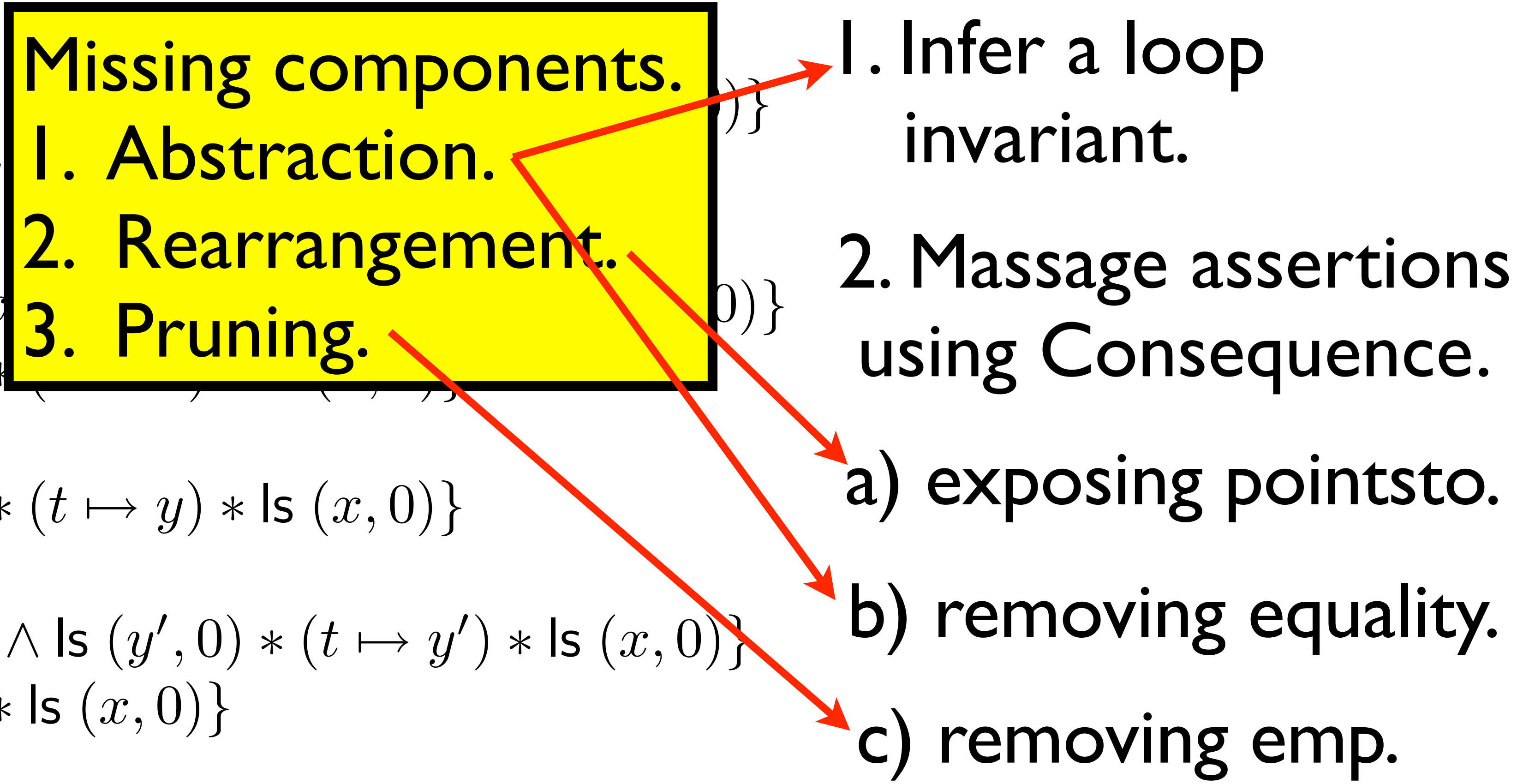
$\quad \{\exists y'.\, y{=}t \wedge \mathsf{ls}\ (y',0) * (t \mapsto y') * \mathsf{ls}\ (x,0)\}$

$\quad \{\mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)\}$

$\}$

$\{x{=}0 \wedge \mathsf{ls}\ (y,0) * \mathsf{ls}\ (x,0)\}$

$\{\mathsf{ls}\ (y,0)\}$



Missing components.

1. Abstraction.
2. Rearrangement.
3. Pruning.

1. Infer a loop invariant.

2. Massage assertions using Consequence.

a) exposing pointsto.

b) removing equality.

c) removing emp.

Today’s Lecture

- Understand the abstraction and rearrangement.
- Should be able to build a basic program verifier.
- Will use the term “static analysis”, instead of “automatic program verifier”.

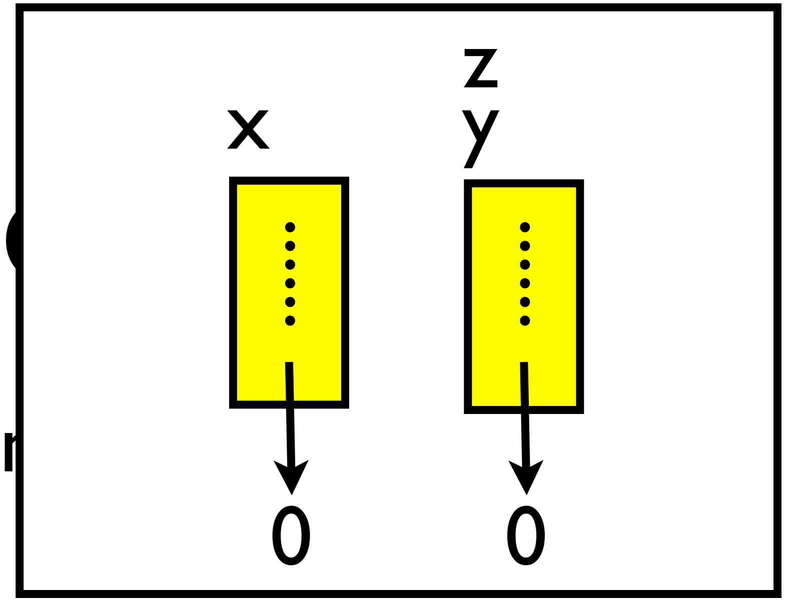
Symbolic Heaps

- Assertions of the particular form:

$$
\begin{array}{rcl}
E, F & ::= & x \mid 0 \\
\Pi & ::= & E = F \mid E \neq F \mid \mathsf{true} \mid \Pi \wedge \Pi' \\
\Sigma & ::= & \mathsf{emp} \mid (E \mapsto F) \mid \mathsf{ls}\,(E, F) \mid \mathsf{true} \mid \Sigma * \Sigma' \\
P, Q & ::= & \exists \vec{x}.\ \Pi \wedge \Sigma
\end{array}
$$

- Restricted. No sep. implication and no univ. quan.
- Built-in list segment predicate without alpha.
- E.g. $y = z \wedge \mathsf{ls}\,(x, 0) * \mathsf{ls}\,(y, 0)$
  $\exists v' w'.\, \mathsf{ls}\,(x, v') * y \mapsto v' * v' \mapsto w'$

Why Symbolic Heaps?
1. Easy to understand visually.
2. Easy to design heuristics for abstraction.



- Assertions of the particular for

$$
\begin{array}{rcl}
E, F & ::= & x \mid 0 \\
\Pi & ::= & E = F \mid E \neq F \mid \mathsf{true} \mid \Pi \wedge \Pi' \\
\Sigma & ::= & \mathsf{emp} \mid (E \mapsto F) \mid \mathsf{ls}\,(E, F) \mid \mathsf{true} \mid \Sigma * \Sigma' \\
P, Q & ::= & \exists \vec{x}.\, \Pi \wedge \Sigma
\end{array}
$$

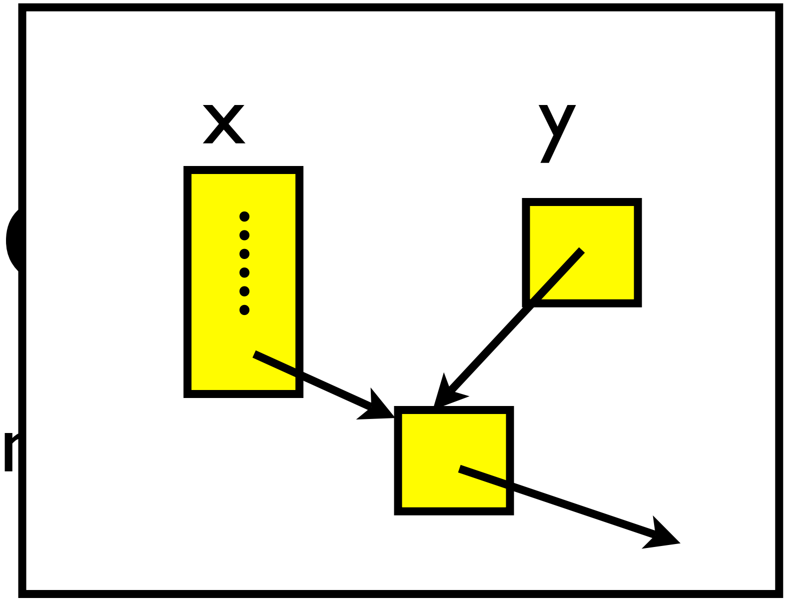
z

- Restricted. No sep. implication and no univ. quan.
- Built-in list segment predicate without alpha.
- E.g. $\boxed{y = z \wedge \mathsf{ls}\,(x, 0) * \mathsf{ls}\,(y, 0)}$

  $\exists v' w'.\, \mathsf{ls}\,(x, v') * y \mapsto v' * v' \mapsto w'$

Why Symbolic Heaps?

1. Easy to understand visually.
2. Easy to design heuristics for abstraction.



- Assertions of the particular for

$$
\begin{array}{rcl}
E, F & ::= & x \mid 0 \\
\mathit{\Pi} & ::= & E = F \mid E \neq F \mid \mathsf{true} \mid \mathit{\Pi} \wedge \mathit{\Pi}' \\
\mathit{\Sigma} & ::= & \mathsf{emp} \mid (E \mapsto F) \mid \mathsf{ls}\,(E, F) \mid \mathsf{true} \mid \mathit{\Sigma} * \mathit{\Sigma}' \\
P, Q & ::= & \exists \vec{x}.\, \mathit{\Pi} \wedge \mathit{\Sigma}
\end{array}
$$

z

- Restricted. No sep. implication and no univ. quan.
- Built-in list segment predicate without alpha.
- E.g. $y = z \wedge \mathsf{ls}\,(x, 0) * \mathsf{ls}\,(y, 0)$

$$\boxed{\exists v' w'.\, \mathsf{ls}\,(x, v') * y \mapsto v' * v' \mapsto w'}$$

Analysis Algorithm

- Input: a set (i.e., disjunction) of sym. heaps, and a program.
- Output: a set of sym. heaps at each program point.
- Finds a proof sketch in separation logic.
- Algorithm:
  - Abstractly run a program with sym. heaps.
  - Accumulate all the obtained sym. heaps.
  - Repeat until no changes.

Analysis Algorithm

- Input: a set (i.e., disjunction) of sym. heaps, and a program.
- Output: a set of sym. heaps at each program point
- Finds a proof sketch in s
- Algorithm:
  - Abstractly run a program with sym. heaps.
  - Accumulate all the obtained sym. heaps.
  - Repeat until no changes.

Rearrangement +
Proof rule in sep. logic +
Abstraction

create

emp

```
h = 0;


while (nondet)
{

    t = new(1);


    *t=h;


    h=t;


    t=0;

}
```

create

emp

```
h = 0;
```

h=0 ∧ emp

```
while (nondet)
{

   t = new(1);


   *t=h;


   h=t;


   t=0;

}
```

create

```
            emp
h = 0;
              h=0 ∧ emp
while (nondet)
{
              h=0 ∧ emp
   t = new(1);


   *t=h;


   h=t;


   t=0;

}
```

create

```
        emp
h = 0;
            h=0 ∧ emp
while (nondet)
{
            h=0 ∧ emp
   t = new(1);
        ∃t’. h=0 ∧ t↦t’
   *t=h;


   h=t;


   t=0;

}
```

create

```
              emp
h = 0;
              h=0 ∧ emp
while (nondet)
{
              h=0 ∧ emp
   t = new(1);
          ∃t'. h=0 ∧ t↦t'
   *t=h;
          ∃t'. h=0 ∧ t↦h
   h=t;


   t=0;

}
```

create

```
                emp
h = 0;
                h=0 ∧ emp
while (nondet)
{
                h=0 ∧ emp
   t = new(1);
            ∃t’. h=0 ∧ t↦t’
   *t=h;
            h=0 ∧ t↦0
   h=t;


   t=0;

}
```

create

```
emp
h = 0;
        h=0 ∧ emp
while (nondet)
{
        h=0 ∧ emp
   t = new(1);
     ∃t’. h=0 ∧ t↦t’
   *t=h;
     h=0 ∧ t↦0
   h=t;
     ∃h’. h=t ∧ h’=0 ∧ t↦h’
   t=0;

}
```

create

```
              emp
h = 0;
              h=0 ∧ emp
while (nondet)
{
              h=0 ∧ emp
   t = new(1);
              ∃t’. h=0 ∧ t↦t’
   *t=h;
              h=0 ∧ t↦0
   h=t;
              h=t ∧ t↦0
   t=0;

}
```

create

```
             emp
h = 0;
             h=0 ∧ emp
while (nondet)
{
             h=0 ∧ emp
   t = new(1);
         ∃t’. h=0 ∧ t↦t’
   *t=h;
         h=0 ∧ t↦0
   h=t;
         h=t ∧ t↦0
   t=0;
         ∃t’. t=0 ∧ h=t’ ∧ t’↦0
}
```

create

```
        emp
h = 0;
          h=0 ∧ emp
while (nondet)
{
          h=0 ∧ emp
  t = new(1);
      ∃t’. h=0 ∧ t↦t’
  *t=h;
      h=0 ∧ t↦0
  h=t;
          h=t ∧ t↦0
  t=0;
          t=0 ∧ h↦0
}
```

create

```
emp
h = 0;
        h=0 ∧ emp
while (nondet)
{
        h=0 ∧ emp
    t = new(1);
      ∃t’. h=0 ∧ t↦t’
    *t=h;
        h=0 ∧ t↦0
    h=t;
            h=t ∧ t↦0
    t=0;
           t=0 ∧ h↦0
}
```

t=0 ∧ h↦0

create

emp

`h = 0;`

$h=0 \wedge \mathrm{emp}$ $t=0 \wedge h \mapsto 0$

`while (nondet)`

`{`

$h=0 \wedge \mathrm{emp}$ $t=0 \wedge h \mapsto 0$

`t = new(1);`

$\exists t'.\ h=0 \wedge t \mapsto t'$

`*t=h;`

$h=0 \wedge t \mapsto 0$

`h=t;`

$h=t \wedge t \mapsto 0$

`t=0;`

$t=0 \wedge h \mapsto 0$

`}`

create

```
            emp
h = 0;
            h=0 ∧ emp              t=0 ∧ h↦0
while (nondet)
{
            h=0 ∧ emp              t=0 ∧ h↦0
    t = new(1);
          ∃t’. h=0 ∧ t↦t’     ∃t’t”. t’=0 ∧ t↦t” * h↦0
    *t=h;
          h=0 ∧ t↦0
    h=t;
          h=t ∧ t↦0
    t=0;
          t=0 ∧ h↦0
}
```

create

```
              emp
h = 0;
              h=0 ∧ emp            t=0 ∧ h↦0
while (nondet)
{             h=0 ∧ emp            t=0 ∧ h↦0
    t = new(1);
          ∃t’. h=0 ∧ t↦t’       ∃t”. t↦t” * h↦0
    *t=h;
          h=0 ∧ t↦0
    h=t;
          h=t ∧ t↦0
    t=0;
          t=0 ∧ h↦0
}
```

create

emp

h = 0;

$h=0 \land emp$ $t=0 \land h \mapsto 0$

while (nondet)
{

$h=0 \land emp$ $t=0 \land h \mapsto 0$

t = new(1);

$\exists t'.\ h=0 \land t \mapsto t'$ $\exists t''.\ t \mapsto t'' * h \mapsto 0$

*t=h;

$h=0 \land t \mapsto 0$ $\exists t''.\ t \mapsto h * h \mapsto 0$

h=t;

$h=t \land t \mapsto 0$

t=0;

$t=0 \land h \mapsto 0$

}

create

```
emp
h = 0;
        h=0 ∧ emp              t=0 ∧ h↦0
while (nondet)
{
        h=0 ∧ emp              t=0 ∧ h↦0
   t = new(1);
      ∃t’. h=0 ∧ t↦t’       ∃t”. t↦t” * h↦0
   *t=h;
      h=0 ∧ t↦0               t↦h * h↦0
   h=t;
      h=t ∧ t↦0
   t=0;
      t=0 ∧ h↦0
}
```

create

```
            emp
h = 0;
            h=0 ∧ emp            t=0 ∧ h↦0
while (nondet)
{
            h=0 ∧ emp            t=0 ∧ h↦0
   t = new(1);
        ∃t’. h=0 ∧ t↦t’          ∃t”. t↦t” * h↦0
   *t=h;
        h=0 ∧ t↦0                t↦h * h↦0
   h=t;
        h=t ∧ t↦0                ∃h’. h=t ∧ t↦h’ * h’↦0
   t=0;
        t=0 ∧ h↦0
}
```

create

```
        emp
h = 0;
        h=0 ∧ emp              t=0 ∧ h↦0
while (nondet)
{
        h=0 ∧ emp              t=0 ∧ h↦0
  t = new(1);
      ∃t’. h=0 ∧ t↦t’          ∃t”. t↦t” * h↦0
  *t=h;
      h=0 ∧ t↦0                t↦h * h↦0

  h=t;
      h=t ∧ t↦0                ∃h’. h=t ∧ ls (t,0)
  t=0;
      t=0 ∧ h↦0
}
```

create

```
        emp
h = 0;
        h=0 ∧ emp            t=0 ∧ h↦0
while (nondet)
{
        h=0 ∧ emp            t=0 ∧ h↦0
   t = new(1);
     ∃t’. h=0 ∧ t↦t’         ∃t”. t↦t” * h↦0
   *t=h;
     h=0 ∧ t↦0               t↦h * h↦0
   h=t;
     h=t ∧ t↦0               h=t ∧ ls(t,0)
   t=0;
     t=0 ∧ h↦0
}
```

create

```
              emp
h = 0;
              h=0 ∧ emp                  t=0 ∧ h↦0
while (nondet)
{
              h=0 ∧ emp                  t=0 ∧ h↦0
   t = new(1);
          ∃t’. h=0 ∧ t↦t’            ∃t”. t↦t” * h↦0
   *t=h;
           h=0 ∧ t↦0                   t↦h * h↦0
   h=t;
              h=t ∧ t↦0                  h=t ∧ ls(t,0)
   t=0;
             t=0 ∧ h↦0              ∃t’. t=0 ∧ h=t’ ∧ ls(t’,0)
}
```

create

```
emp
h = 0;
        h=0 ∧ emp              t=0 ∧ h↦0
while (nondet)
{
        h=0 ∧ emp              t=0 ∧ h↦0
    t = new(1);
      ∃t’. h=0 ∧ t↦t’          ∃t”. t↦t” * h↦0
    *t=h;
      h=0 ∧ t↦0                t↦h * h↦0
    h=t;
      h=t ∧ t↦0                h=t ∧ ls(t,0)
    t=0;
      t=0 ∧ h↦0                t=0 ∧ ls(h,0)
}
```

create

emp

h = 0;

h=0 ∧ emp

t=0 ∧ h↦0

t=0 ∧ ls(h,0)

while (nondet)

{

h=0 ∧ emp

t=0 ∧ h↦0

t = new(1);

∃t’. h=0 ∧ t↦t’

∃t”. t↦t” * h↦0

*t=h;

h=0 ∧ t↦0

t↦h * h↦0

h=t;

h=t ∧ t↦0

h=t ∧ ls(t,0)

t=0;

t=0 ∧ h↦0

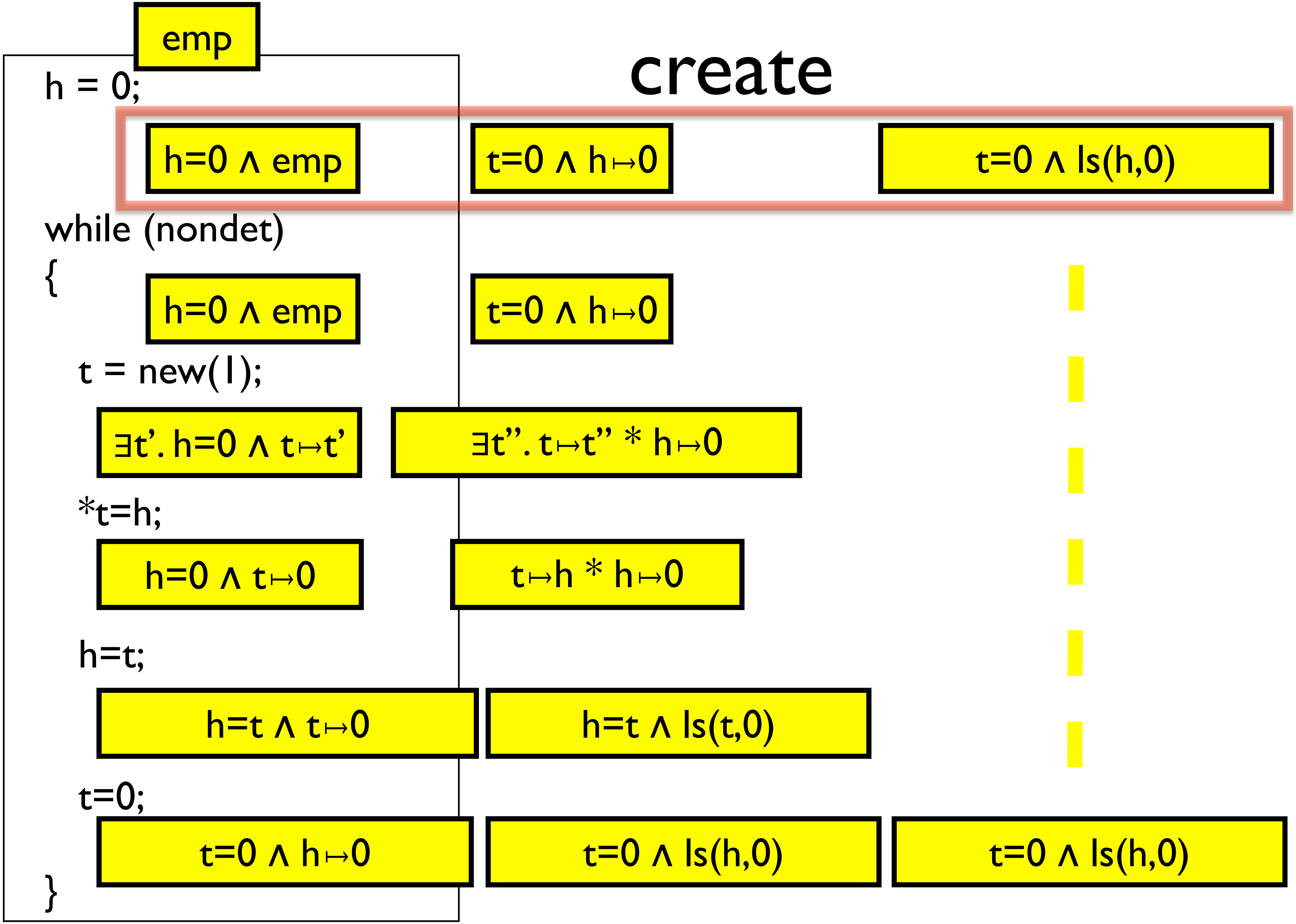
t=0 ∧ ls(h,0)

}

create

```
         emp
h = 0;
         h=0 ∧ emp            t=0 ∧ h↦0                t=0 ∧ ls(h,0)
while (nondet)
{
         h=0 ∧ emp            t=0 ∧ h↦0
   t = new(1);
      ∃t’. h=0 ∧ t↦t’     ∃t”. t↦t” * h↦0
   *t=h;
      h=0 ∧ t↦0             t↦h * h↦0
   h=t;
      h=t ∧ t↦0             h=t ∧ ls(t,0)
   t=0;
      t=0 ∧ h↦0             t=0 ∧ ls(h,0)            t=0 ∧ ls(h,0)
}
```

create



emp

h = 0;

h=0 ∧ emp | t=0 ∧ h↦0 | t=0 ∧ ls(h,0)

while (nondet)

{

h=0 ∧ emp | t=0 ∧ h↦0

t = new(1);

∃t’. h=0 ∧ t↦t’ | ∃t”. t↦t” * h↦0

*t=h;

h=0 ∧ t↦0 | t↦h * h↦0

h=t;

h=t ∧ t↦0 | h=t ∧ ls(t,0)

t=0;

t=0 ∧ h↦0 | t=0 ∧ ls(h,0) | t=0 ∧ ls(h,0)

}

create

```
emp
h = 0;
      h=0 ∧ emp            t=0 ∧ h↦0                    t=0 ∧ ls(h,0)
while (nondet)
{
      h=0 ∧ emp            t=0 ∧ h↦0
   t = new(1);
   ∃t’. h=0 ∧ t↦t’     ∃t”. t↦t” * h↦0
   *t=h;
   h=0 ∧ t↦0              t↦h * h↦0
   h=t;
   h=t ∧ t↦0              h=t ∧ ls(t,0)
   t=0;
   t=0 ∧ h↦0              t=0 ∧ ls(h,0)               t=0 ∧ ls(h,0)
}
     h=0 ∧ emp            t=0 ∧ h↦0                    t=0 ∧ ls(h,0)
```

create

```
emp
h = 0;
      h=0 ∧ emp            t=0 ∧ h↦0            t=0 ∧ ls(h,0)
while (nondet)
{
      h=0 ∧ emp            t=0 ∧ h↦0
  t = new(1);

  h=t;
    ∃h’. h=t ∧ h’=0 ∧ t↦h’    ∃h’. h=t ∧ t↦h’ * h’↦0
  t=0;
    ∃t’. t=0 ∧ h=t’ ∧ t’↦0    ∃t’. t=0 ∧ h=t’ ∧ ls(t’,0)    t=0 ∧ ls(h,0)
}
      h=0 ∧ emp            t=0 ∧ h↦0            t=0 ∧ ls(h,0)
```

The output is a proof sketch of

{ emp }create{ (h=0∧emp) ∨ (t=0∧h↦0) ∨ (t=0∧ls(h,0)) }

Abstract Semantics of Atomic Command A

$$
\begin{aligned}
(\!|A|\!) &\quad : \quad \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\
(\!|A|\!) &\quad = \quad \mathsf{Abs} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A
\end{aligned}
$$

Abstract Semantics of Atomic Command A

$$\begin{array}{lcl} (\!|A|\!) & : & \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) & = & \mathsf{Abs} \circ \mathsf{RuleApply}_A \circ \boxed{\mathsf{Rearr}_A} \end{array}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

$$\frac{}{\{\exists \vec{x'}.\, \mathit{\Pi} \wedge x{\mapsto}x' * \mathit{\Sigma}\} {*x{=}t} \{\exists \vec{x'}.\, \mathit{\Pi} \wedge x{\mapsto}t * \mathit{\Sigma}\}}$$

$$\cfrac{\cfrac{\overline{\{x{\mapsto}x' * (\Pi \wedge \Sigma)\}{*x{=}t}\{x{\mapsto}t * (\Pi \wedge \Sigma)\}}}{\{\ \Pi \wedge x{\mapsto}x' * \Sigma\}{*x{=}t}\{\exists\vec{x'}.\ \Pi \wedge x{\mapsto}t * \Sigma\}}\ \text{Conseq.}}{\{\exists\vec{x'}.\ \Pi \wedge x{\mapsto}x' * \Sigma\}{*x{=}t}\{\exists\vec{x'}.\ \Pi \wedge x{\mapsto}t * \Sigma\}}\ \text{Exist.}$$

omic Command A

SymHeap) ∪ {err}

$$(\!|A|\!) \quad = \quad \mathsf{Abs} \circ \mathsf{RuleApply}_A \circ \boxed{\mathsf{Rearr}_A}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

$$\overline{\{\exists\vec{x'}.\ \Pi \wedge x{\mapsto}x' * \Sigma\}{*x{=}t}\{\exists\vec{x'}.\ \Pi \wedge x{\mapsto}t * \Sigma\}}$$

omic Command A

$$\dfrac{\dfrac{\overline{\{x\mapsto x' * (\Pi \wedge \Sigma)\}{*x}{=}t\{x\mapsto t * (\Pi \wedge \Sigma)\}}}{\{\ \Pi \wedge x\mapsto x' * \Sigma\}{*x}{=}t\{\exists \vec{x}'.\ \Pi \wedge x\mapsto t * \Sigma\}}\ \text{Conseq.}}{\{\exists \vec{x}'.\ \Pi \wedge x\mapsto x' * \Sigma\}{*x}{=}t\{\exists \vec{x}'.\ \Pi \wedge x\mapsto t * \Sigma\}}\ \text{Exist.}$$

SymHeap) ∪ {err}

$$(\!|A|\!) \quad = \quad \text{Abs} \circ \text{RuleApply}_A \circ \boxed{\text{Rearr}_A}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

$$\overline{\{\exists \vec{x}'.\ \Pi \wedge x\mapsto x' * \Sigma\}{*x}{=}t\{\exists \vec{x}'.\ \Pi \wedge x\mapsto t * \Sigma\}}$$

$\exists z'.\ \text{ls}\ (x, z') * (z\mapsto z') * (z'\mapsto 0)$

$$\dfrac{\dfrac{\overline{\{x\mapsto x' * (\mathit{\Pi} \wedge \mathit{\Sigma})\} * x{=}t \{x\mapsto t * (\mathit{\Pi}\wedge\mathit{\Sigma})\}}}{\{\ \mathit{\Pi} \wedge x\mapsto x' * \mathit{\Sigma}\} * x{=}t\{\exists \vec{x'}.\ \mathit{\Pi}\wedge x\mapsto t * \mathit{\Sigma}\}}\ \text{Conseq.}}{\{\exists\vec{x'}.\ \mathit{\Pi}\wedge x\mapsto x' *\mathit{\Sigma}\} * x{=}t \{\exists\vec{x'}.\ \mathit{\Pi}\wedge x\mapsto t*\mathit{\Sigma}\}}\ \text{Exist.}$$

omic Command A

SymHeap) ∪ {err}

$$(\!|A|\!) \quad = \quad \mathsf{Abs} \circ \mathsf{RuleApply}_A \circ \boxed{\mathsf{Rearr}_A}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

$$\overline{\{\exists\vec{x'}.\ \mathit{\Pi}\wedge x\mapsto x' *\mathit{\Sigma}\} * x{=}t \{\exists\vec{x'}.\ \mathit{\Pi}\wedge x\mapsto t*\mathit{\Sigma}\}}$$

$$\exists z'.\ \mathsf{ls}\ (x, z') * (z\mapsto z') * (z'\mapsto 0)$$

$$\{\ \exists z'.\ (x{=}z' \wedge \mathsf{emp}) * (z\mapsto z') * (z'\mapsto 0),\quad \exists x'.(x\mapsto x') * \mathsf{ls}\ (x', z') * (z\mapsto z') * (z'\mapsto 0)\ \ \}$$

$$\cfrac{\cfrac{\overline{\{x\mapsto x' * (\mathit{\Pi} \wedge \mathit{\Sigma})\}{*x{=}t}\{x\mapsto t * (\mathit{\Pi} \wedge \mathit{\Sigma})\}}}{\{\ \mathit{\Pi} \wedge x\mapsto x' * \mathit{\Sigma}\}{*x{=}t}\{\exists \vec{x'}.\ \mathit{\Pi} \wedge x\mapsto t * \mathit{\Sigma}\}}\ \text{Conseq.}}{\{\exists \vec{x'}.\ \mathit{\Pi} \wedge x\mapsto x' * \mathit{\Sigma}\}{*x{=}t}\{\exists \vec{x'}.\ \mathit{\Pi} \wedge x\mapsto t * \mathit{\Sigma}\}}\ \text{Exist.}$$

omic Command A

SymHeap) $\cup$ {err}

$$(\!|A|\!) \quad = \quad \mathsf{Abs} \circ \mathsf{RuleApply}_A \circ \boxed{\mathsf{Rearr}_A}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

$$\overline{\{\exists \vec{x'}.\ \mathit{\Pi} \wedge x\mapsto x' * \mathit{\Sigma}\}{*x{=}t}\{\exists \vec{x'}.\ \mathit{\Pi} \wedge x\mapsto t * \mathit{\Sigma}\}}$$

$$\exists z'.\ \mathsf{ls}\ (x, z') * (z\mapsto z') * (z'\mapsto 0)$$

$$\Downarrow$$

$$\{\quad \exists z'.\ (x{=}z') \wedge (z\mapsto x) * (x\mapsto 0), \qquad \exists x'.(x\mapsto x') * \mathsf{ls}\ (x', z') * (z\mapsto z') * (z'\mapsto 0)\ \ \}$$

Abstract Semantics of Atomic Command A

$$\begin{array}{lcl} (\!|A|\!) & : & \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) & = & \mathsf{Abs} \circ \boxed{\mathsf{RuleApply}_A} \circ \mathsf{Rearr}_A \end{array}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

$$\frac{}{\{\exists \vec{x'}.\ \mathit{\Pi} \wedge x {\mapsto} x' * \mathit{\Sigma}\} {*x{=}t} \{\exists \vec{x'}.\ \mathit{\Pi} \wedge x {\mapsto} t * \mathit{\Sigma}\}}$$

$$\exists z'.\ \mathsf{ls}\ (x, z') * (z {\mapsto} z') * (z' {\mapsto} 0)$$

$$\Downarrow$$

$$\{ \quad \exists z'.\ (x{=}z') \wedge (z {\mapsto} x) * (x {\mapsto} 0), \qquad \exists x'.(x {\mapsto} x') * \mathsf{ls}\ (x', z') * (z {\mapsto} z') * (z' {\mapsto} 0) \quad \}$$

Abstract Semantics of Atomic Command A

$$\begin{aligned} (\!|A|\!) \quad &: \quad \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) \quad &= \quad \mathsf{Abs} \circ \boxed{\mathsf{RuleApply}_A} \circ \mathsf{Rearr}_A \end{aligned}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

$$\frac{}{\{\exists \vec{x'}.\ \Pi \wedge x \mapsto x' * \Sigma\} {*x{=}t} \{\exists \vec{x'}.\ \Pi \wedge x \mapsto t * \Sigma\}}$$

$$\exists z'.\, \mathsf{ls}\ (x, z') * (z \mapsto z') * (z' \mapsto 0)$$

$$\Downarrow$$

$$\{ \quad \exists z'.\, (x{=}z') \wedge (z \mapsto x) * (x \mapsto 0), \qquad \exists x'.(x \mapsto x') * \mathsf{ls}\ (x', z') * (z \mapsto z') * (z' \mapsto 0) \quad \}$$

$$\Downarrow$$

$$\{ \ \exists z'.\, (x{=}z') \wedge (z \mapsto x) * (x \mapsto t), \quad \exists x'.(x \mapsto t) * \mathsf{ls}\ (x', z') * (z \mapsto z') * (z' \mapsto 0) \ \}$$

Abstract Semantics of Atomic Command A

$$(\!|A|\!) \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$(\!|A|\!) \quad = \quad \boxed{\mathsf{Abs}} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}x' * \Sigma\}*x{=}t\{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}t * \Sigma\}}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z{\mapsto}z') * (z'{\mapsto}0)$$

$$\Downarrow$$

$$\{ \quad \exists z'.\, (x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}0), \qquad \exists x'.(x{\mapsto}x') * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \quad \}$$

$$\Downarrow$$

$$\{ \; \exists z'.\, (x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}t), \quad \exists x'.(x{\mapsto}t) * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \; \}$$

Abstract Semantics of Atomic Command A

$$\begin{aligned} (\!|A|\!) &: \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) &= \boxed{\mathsf{Abs}} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A \end{aligned}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}x' * \Sigma\} {*}x{=}t \{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}t * \Sigma\}}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z{\mapsto}z') * (z'{\mapsto}0)$$

$$\Downarrow$$

$$\{\quad \exists z'.\,(x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}0), \quad \exists x'.(x{\mapsto}x') * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \quad\}$$

$$\Downarrow$$

$$\{\ \exists z'.\,(x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}t),\ \ \exists x'.(x{\mapsto}t) * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\ \}$$

$$\Downarrow$$

$$\{\ \exists z'.\,(x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}t),\ \ \exists x'.(x{\mapsto}t) * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\ \}$$

Abstract Semantics of Atomic Command A

$$\begin{aligned} (\!|A|\!) &\quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) &\quad = \quad \boxed{\mathsf{Abs}} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A \end{aligned}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}x' * \Sigma\} *x{=}t \{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}t * \Sigma\}}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z{\mapsto}z') * (z'{\mapsto}0)$$

⇓

$$\{\quad \exists z'.\, (x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}0), \qquad \exists x'.(x{\mapsto}x') * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \quad\}$$

⇓

$$\{\ \exists z'.\, (x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}t), \quad \exists x'.(x{\mapsto}t) * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \ \}$$

⇓

$$\{ \qquad (z{\mapsto}x) * (x{\mapsto}t), \quad \exists x'.(x{\mapsto}t) * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \ \}$$

Abstract Semantics of Atomic Command A

$$\begin{aligned} (\!|A|\!) \quad &: \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) \quad &= \quad \boxed{\mathsf{Abs}} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A \end{aligned}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\, \Pi \wedge x \mapsto x' * \Sigma\} * x = t \{\exists \vec{x'}.\, \Pi \wedge x \mapsto t * \Sigma\}}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z \mapsto z') * (z' \mapsto 0)$$

⇓

$$\{ \quad \exists z'.\, (x = z') \wedge (z \mapsto x) * (x \mapsto 0), \qquad \exists x'.(x \mapsto x') * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0) \quad \}$$

⇓

$$\{ \; \exists z'.\, (x = z') \wedge (z \mapsto x) * (x \mapsto t), \quad \exists x'.(x \mapsto t) * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0) \; \}$$

⇓

$$\{ \qquad (z \mapsto x) * (x \mapsto t), \qquad \exists x'.(x \mapsto t) * \mathsf{true} * (z \mapsto z') * (z' \mapsto 0) \qquad \}$$

Abstract Semantics of Atomic Command A

$$(\!|A|\!) \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$(\!|A|\!) \quad = \quad \boxed{\mathsf{Abs}} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\, \mathit{\Pi} \wedge x{\mapsto}x' * \mathit{\Sigma}\}{*}x{=}t\{\exists \vec{x'}.\, \mathit{\Pi} \wedge x{\mapsto}t * \mathit{\Sigma}\}}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z{\mapsto}z') * (z'{\mapsto}0)$$

$$\Downarrow$$

$$\{\quad \exists z'.\,(x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}0), \qquad \exists x'.(x{\mapsto}x') * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\quad\}$$

$$\Downarrow$$

$$\{\ \exists z'.\,(x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}t), \quad \exists x'.(x{\mapsto}t) * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\ \}$$

$$\Downarrow$$

$$\{\qquad (z{\mapsto}x) * (x{\mapsto}t), \qquad \exists x'.(x{\mapsto}t) * \mathsf{true} * (\mathsf{ls}\,(z, 0)) \qquad\}$$

Abstract Semantics of Atomic Command A

$$\begin{aligned} (\!|A|\!) &: \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \\ (\!|A|\!) &= \boxed{\mathsf{Abs}} \circ \mathsf{RuleApply}_A \circ \mathsf{Rearr}_A \end{aligned}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}x' * \Sigma\} {*x{=}t} \{\exists \vec{x'}.\, \Pi \wedge x{\mapsto}t * \Sigma\}}$$

$$\exists z'.\, \mathsf{ls}\ (x, z') * (z{\mapsto}z') * (z'{\mapsto}0)$$

⇓

$$\{\quad \exists z'.\, (x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}0), \qquad \exists x'.(x{\mapsto}x') * \mathsf{ls}\ (x', z') * (z{\mapsto}z') * (z'{\mapsto}0) \quad\}$$

⇓

$$\{\ \exists z'.\, (x{=}z') \wedge (z{\mapsto}x) * (x{\mapsto}t), \quad \exists x'.(x{\mapsto}t) * \mathsf{ls}\ (x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\ \}$$

⇓

$$\{\qquad (z{\mapsto}x) * (x{\mapsto}t), \qquad (x{\mapsto}t) * \mathsf{true} * (\mathsf{ls}\ (z, 0)) \qquad\}$$

Abstract Semantics of Atomic Command A

$$(\!|A|\!) \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$(\!|A|\!) \quad = \quad \mathsf{Abs} \circ \mathsf{RuleApply}_A \circ \boxed{\mathsf{Rearr}_A}$$

[Step1] Rearrange a symbolic heap, so that it pattern-matches the precond. of an adjusted proof rule.

[Step2] Apply the rule.

[Step3] Abstract symbolic heaps.

$$\frac{}{\{\exists \vec{x'}.\ \Pi \wedge x \mapsto x' * \Sigma\} *x{=}t \{\exists \vec{x'}.\ \Pi \wedge x \mapsto t * \Sigma\}}$$

ls $(x, 0)$

⇩

err

⇩

err

⇩

err

Adjusting a Proof Rule

$$\dfrac{\dfrac{\dfrac{}{\{E\mapsto E_0 * (\Pi \wedge \Sigma)\}{*}E{=}F\{E\mapsto F * (\Pi \wedge \Sigma)\}}\ \text{Update}}{\{\Pi \wedge (E\mapsto E_0 * \Sigma)\}{*}E{=}F\{\Pi \wedge (E\mapsto F * \Sigma)\}}\ \text{Conseq.}}{\{\exists \vec{x'}.\ \Pi \wedge (E\mapsto E_0 * \Sigma)\}{*}E{=}F\{\exists \vec{x'}.\ \Pi \wedge (E\mapsto F * \Sigma)\}}\ \text{Exist.}$$

- Make a rule work for symbolic heaps.
  - The precondition becomes a symbolic heap with the accessed cell exposed.
- Use Consequence and the below equivalence:

$$(E{=}F \wedge \Sigma_0 * \Sigma_1) \iff \Sigma_0 * (\Sigma_1 \wedge E{=}F)$$

$$(E{\neq}F \wedge \Sigma_0 * \Sigma_1) \iff \Sigma_0 * (\Sigma_1 \wedge E{\neq}F)$$

- Use the existential elimination rule.

Adjusting a Proof Rule

$$\dfrac{\dfrac{\dfrac{}{\{E\mapsto F * (\mathit{\Pi} \wedge \mathit{\Sigma})\}x{=}{*}E\{\exists x'.\, x{=}E[x'/x] \wedge (E\mapsto F * (\mathit{\Pi} \wedge \mathit{\Sigma}))[x'/x]\}}\ \text{Update}}{\{\mathit{\Pi} \wedge E\mapsto F * \mathit{\Sigma}\}x{=}{*}E\{\exists x'.\, x{=}E[x'/x] \wedge (\mathit{\Pi} \wedge E\mapsto F * \mathit{\Sigma}))[x'/x]\}}\ \text{Conseq.}}{\{\exists \vec{y'}.\mathit{\Pi} \wedge E\mapsto F * \mathit{\Sigma}\}x{=}{*}E\{\exists \vec{y'}x'.\, x{=}E[x'/x] \wedge (\mathit{\Pi} \wedge E\mapsto F * \mathit{\Sigma}))[x'/x]\}}\ \text{Exist.}$$

- Make a rule work for symbolic heaps.
  - The precondition becomes a symbolic heap with the accessed cell exposed.
- Use Consequence and the below equivalence:

$$(E{=}F \wedge \mathit{\Sigma}_0 * \mathit{\Sigma}_1) \iff \mathit{\Sigma}_0 * (\mathit{\Sigma}_1 \wedge E{=}F)$$

$$(E{\neq}F \wedge \mathit{\Sigma}_0 * \mathit{\Sigma}_1) \iff \mathit{\Sigma}_0 * (\mathit{\Sigma}_1 \wedge E{\neq}F)$$

- Use the existential elimination rule.

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\frac{}{\{\exists \vec{x'}.\ \Pi \wedge (E \mapsto E_0 * \Sigma)\} {*E{=}F} \{\exists \vec{x'}.\ \Pi \wedge (E \mapsto F * \Sigma)\}}$$

- Transforms a sym. heap so that it pattern-matches the precondition of the rule.
- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists \vec{x'} y'.\ \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\ \Pi \wedge E' \mapsto F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E \mapsto F' * \Sigma\ \}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z \mapsto z') * (z' \mapsto 0)$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\, \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\, \exists \vec{x'}.\, \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists \vec{x'} y'.\, \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma \,\}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\, \Pi \wedge E' \mapsto F' * \Sigma \quad \rightsquigarrow_E \quad \{\, \exists \vec{x'}.\, \Pi \wedge E \mapsto F' * \Sigma \,\}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\exists z'.\, \underline{\mathsf{ls}\,(x, z')} * (z \mapsto z') * (z' \mapsto 0)$$

$$\Downarrow x$$

$$\{\ \exists z'.\, x = z' \wedge (z \mapsto z') * (z' \mapsto 0),\ \ \exists z' x'.(x \mapsto x') * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0)\ \}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\, \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \ \ \rightsquigarrow_E \ \ \{\ \exists \vec{x'}.\, \Pi \wedge E' = F' \wedge \Sigma,\ \ \exists \vec{x'} y'.\, \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma\ \} \qquad (\text{when } \Pi \Rightarrow E = E')$$

$$\exists \vec{x'}.\, \Pi \wedge E' \mapsto F' * \Sigma \ \ \rightsquigarrow_E \ \ \{\ \exists \vec{x'}.\, \Pi \wedge E \mapsto F' * \Sigma\ \} \qquad (\text{when } \Pi \Rightarrow E = E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\boxed{\begin{array}{l}
\qquad \exists z'.\, \mathsf{ls}\,(x, z') * (z \mapsto z') * (z' \mapsto 0) \\
\qquad\qquad\qquad\qquad \Downarrow x \\
\{\ \exists z'.\, x{=}z' \wedge (z \mapsto z') * \underline{(z' \mapsto 0)},\ \ \exists z' x'.(x \mapsto x') * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0)\ \ \} \\
\qquad\qquad\qquad\qquad \Downarrow x \\
\{\ \exists z'.\, x{=}z' \wedge (z \mapsto z') * (x \mapsto 0),\ \ \exists z' x'.(x \mapsto x') * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0)\ \ \}
\end{array}}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\, \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \leadsto_E \quad \{\, \exists \vec{x'}.\, \Pi \wedge E'{=}F' \wedge \Sigma,\quad \exists \vec{x'} y'.\, \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma \,\}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\boxed{\begin{array}{l}
\exists \vec{x'}.\, \Pi \wedge E' \mapsto F' * \Sigma \quad \leadsto_E \quad \{\, \exists \vec{x'}.\, \Pi \wedge E \mapsto F' * \Sigma \,\} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{when } \Pi \Rightarrow E{=}E')
\end{array}}$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\exists z'.\, \mathsf{ls}\,(x, z') * (z{\mapsto}z') * (z'{\mapsto}0)$$

$$\Downarrow_{\mathsf{x}}$$

$$\{\ \exists z'.\, x{=}z' \wedge (z{\mapsto}z') * (z'{\mapsto}0),\ \ \exists z'x'.(x{\mapsto}x') * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\ \}$$

$$\Downarrow_{\mathsf{x}}$$

$$\{\ \exists z'.\, x{=}z' \wedge (z{\mapsto}z') * \underline{(x{\mapsto}0),}\ \ \exists z'x'\underline{.(x{\mapsto}x')} * \mathsf{ls}\,(x', z') * (z{\mapsto}z') * (z'{\mapsto}0)\ \}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and $\boxed{\text{an allocatedness check.}}$

$$\exists \vec{x'}.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E', F') * \mathit{\Sigma} \quad \rightsquigarrow_E \quad \{\, \exists \vec{x'}.\, \mathit{\Pi} \wedge E'{=}F' \wedge \mathit{\Sigma},\quad \exists \vec{x'}y'.\, \mathit{\Pi} \wedge E{\mapsto}y' * \mathsf{ls}\,(y', F') * \mathit{\Sigma}\,\}$$

$$(\text{when } \mathit{\Pi} \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\, \mathit{\Pi} \wedge E'{\mapsto}F' * \mathit{\Sigma} \quad \rightsquigarrow_E \quad \{\, \exists \vec{x'}.\, \mathit{\Pi} \wedge E{\mapsto}F' * \mathit{\Sigma}\,\}$$

$$(\text{when } \mathit{\Pi} \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\exists z'.\,\mathsf{ls}\,(x, z') * (z \mapsto z') * (z' \mapsto 0)$$

$$\Downarrow_{\mathsf{x}}$$

$$\{\ \exists z'.\, x = z' \wedge (z \mapsto z') * (z' \mapsto 0),\ \ \exists z' x'.(x \mapsto x') * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0)\ \}$$

return

$$\Downarrow_{\mathsf{x}}$$

$$\boxed{\{\ \exists z'.\, x = z' \wedge (z \mapsto z') * (x \mapsto 0),\ \ \exists z' x'.(x \mapsto x') * \mathsf{ls}\,(x', z') * (z \mapsto z') * (z' \mapsto 0)\ \}}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E' = F' \wedge \Sigma,\quad \exists \vec{x'} y'.\ \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E = E')$$

$$\exists \vec{x'}.\ \Pi \wedge E' \mapsto F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E \mapsto F' * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E = E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\mathsf{ls}(x, 0)$$

$$\Downarrow_{\mathsf{x}}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists \vec{x'} y'.\ \Pi \wedge E {\mapsto} y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\ \Pi \wedge E' {\mapsto} F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E {\mapsto} F' * \Sigma\ \}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\mathsf{ls}(x, 0)$$

$$\Downarrow \mathsf{x}$$

$$\{\ \ x{=}0 \wedge \mathsf{emp},\ \ \exists x'.\,(x{\mapsto}x') * \mathsf{ls}\,(x', 0)\ \ \}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E'{=}F' \wedge \Sigma,\quad \exists \vec{x'} y'.\ \Pi \wedge E{\mapsto}y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\ \Pi \wedge E'{\mapsto}F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E{\mapsto}F' * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\mathsf{ls}(x, 0)$$

⇩x

$$\{ \ \boxed{x{=}0 \wedge \mathsf{emp},} \ \exists x'.\,(x {\mapsto} x') * \mathsf{ls}\,(x', 0) \ \}$$

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists \vec{x'} y'.\ \Pi \wedge E {\mapsto} y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\ \Pi \wedge E' {\mapsto} F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E {\mapsto} F' * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\mathsf{ls}(x, 0)$$

$$\Downarrow \mathsf{x}$$

$$\{ \;\; x{=}0 \wedge \mathsf{emp}, \quad \exists x'.\, (x{\mapsto}x') * \mathsf{ls}\,(x', 0) \;\; \}$$

return err

- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\, \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \leadsto_E \quad \{\, \exists \vec{x'}.\, \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists \vec{x'} y'.\, \Pi \wedge E{\mapsto}y' * \mathsf{ls}\,(y', F') * \Sigma \,\}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\, \Pi \wedge E'{\mapsto}F' * \Sigma \quad \leadsto_E \quad \{\, \exists \vec{x'}.\, \Pi \wedge E{\mapsto}F' * \Sigma \,\}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\frac{}{\{\exists\vec{x'}.\ \Pi \wedge (E \mapsto E_0 * \Sigma)\} * E{=}F \{\exists\vec{x'}.\ \Pi \wedge (E \mapsto F * \Sigma)\}}$$

- Transforms a sym. heap so that it pattern-matches the preco
- Unro s) to expos
- Defined by rewriting rules and an allocatedness check.

Allocatedness check:
1. Filter out inconsistent sym. heaps.
2. Checks the existence of E↦F.

$$\exists\vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists\vec{x'}.\ \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists\vec{x'}y'.\ \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists\vec{x'}.\ \Pi \wedge E' \mapsto F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists\vec{x'}.\ \Pi \wedge E \mapsto F' * \Sigma\ \}$$
$$(\text{when } \Pi \Rightarrow E{=}E')$$

Rearr

$$\mathsf{Rearr}_A \quad : \quad \mathsf{SymHeap} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

$$\frac{}{\{\exists \vec{x'}.\ \Pi \wedge (E \mapsto E_0 * \Sigma)\} {*E}{=}F \{\exists \vec{x'}.\ \Pi \wedge (E \mapsto F * \Sigma)\}}$$

- Transforms a sym. heap so that it pattern-matches the precondition of the rule.
- Unroll inductively defined predicates (e.g., ls) to expose (E↦F) about accessed cell E.
- Defined by rewriting rules and an allocatedness check.

$$\exists \vec{x'}.\ \Pi \wedge \mathsf{ls}\,(E', F') * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E'{=}F' \wedge \Sigma, \quad \exists \vec{x'} y'.\ \Pi \wedge E \mapsto y' * \mathsf{ls}\,(y', F') * \Sigma\ \}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

$$\exists \vec{x'}.\ \Pi \wedge E' \mapsto F' * \Sigma \quad \rightsquigarrow_E \quad \{\ \exists \vec{x'}.\ \Pi \wedge E \mapsto F' * \Sigma\ \}$$

$$(\text{when } \Pi \Rightarrow E{=}E')$$

Abs

$$\mathsf{Abs} \quad : \quad \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \to \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

- Forgets the length of linked lists, and simplify quantifiers.
- Maps err to err.
- Defined by rewriting rules (true implications in sep. logic)

$$(\exists \vec{y'} x'.\, \Pi \wedge E \mapsto x' * x' \mapsto F * \Sigma) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma)$$
$$(\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma, E, F))$$

$$(\exists \vec{y'} x'.\, \Pi \wedge \mathsf{ls}\,(E, x') * \mathsf{ls}\,(x', F) * \Sigma) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma)$$
$$(\text{when } y' \notin \mathsf{FV}(\Pi, \Sigma, E, F))$$

$$(\exists \vec{y'} x'.\, \Pi \wedge x' \mapsto F * \Sigma) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{true} * \Sigma)$$
$$(\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma))$$

$$(\exists \vec{y'} x'.\, x' = E \wedge \Pi \wedge \Sigma) \;\rightsquigarrow\; (\exists \vec{y'}.\, (\Pi \wedge \Sigma)[E/x'])$$

Abs

$$\exists x' x'' y'.\ x \mapsto x' * x' \mapsto x'' * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$\rightsquigarrow$$

$$\begin{array}{rcl} (\exists \vec{y'} x'.\, \Pi \wedge E \mapsto x' * x' \mapsto F * \Sigma) & \rightsquigarrow & (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma) \\ & & (\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma, E, F)) \end{array}$$

$$\begin{array}{rcl} (\exists \vec{y'} x'.\, \Pi \wedge \mathsf{ls}\,(E, x') * \mathsf{ls}\,(x', F) * \Sigma) & \rightsquigarrow & (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma) \\ & & (\text{when } y' \notin \mathsf{FV}(\Pi, \Sigma, E, F)) \end{array}$$

$$\begin{array}{rcl} (\exists \vec{y'} x'.\, \Pi \wedge x' \mapsto F * \Sigma) & \rightsquigarrow & (\exists \vec{y'}.\, \Pi \wedge \mathsf{true} * \Sigma) \\ & & (\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma)) \end{array}$$

$$(\exists \vec{y'} x'.\, x' {=} E \wedge \Pi \wedge \Sigma) \ \rightsquigarrow\ (\exists \vec{y'}.\, (\Pi \wedge \Sigma)[E/x'])$$

Abs

$$\exists x' x'' y'.\ x \mapsto x' * x' \mapsto x'' * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$\rightsquigarrow$$

$$\exists x'' y'.\ \mathsf{ls}\,(x, x'') * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$(\exists \vec{y'} x'.\, \mathit{\Pi} \wedge E \mapsto x' * x' \mapsto F * \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E, F) * \mathit{\Sigma})$$
$$(\text{when } x' \notin \mathsf{FV}(\mathit{\Pi}, \mathit{\Sigma}, E, F))$$

$$(\exists \vec{y'} x'.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E, x') * \mathsf{ls}\,(x', F) * \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E, F) * \mathit{\Sigma})$$
$$(\text{when } y' \notin \mathsf{FV}(\mathit{\Pi}, \mathit{\Sigma}, E, F))$$

$$(\exists \vec{y'} x'.\, \mathit{\Pi} \wedge x' \mapsto F * \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \mathit{\Pi} \wedge \mathsf{true} * \mathit{\Sigma})$$
$$(\text{when } x' \notin \mathsf{FV}(\mathit{\Pi}, \mathit{\Sigma}))$$

$$(\exists \vec{y'} x'.\, x' = E \wedge \mathit{\Pi} \wedge \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, (\mathit{\Pi} \wedge \mathit{\Sigma})[E/x'])$$

Abs

$$\exists x' x'' y'.\ x \mapsto x' * x' \mapsto x'' * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$\leadsto$$

$$\exists x'' y'.\ \mathsf{ls}\,(x, x'') * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$\leadsto$$

$$\exists y'.\ \mathsf{ls}\,(x, 0) * y' \mapsto 0$$

.

c)

$$(\exists \vec{y'} x'.\, \Pi \wedge E \mapsto x' * x' \mapsto F * \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma)$$
$$(\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma, E, F))$$

$$(\exists \vec{y'} x'.\, \Pi \wedge \mathsf{ls}\,(E, x') * \mathsf{ls}\,(x', F) * \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma)$$
$$(\text{when } y' \notin \mathsf{FV}(\Pi, \Sigma, E, F))$$

$$(\exists \vec{y'} x'.\, \Pi \wedge x' \mapsto F * \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{true} * \Sigma)$$
$$(\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma))$$

$$(\exists \vec{y'} x'.\, x' = E \wedge \Pi \wedge \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, (\Pi \wedge \Sigma)[E/x'])$$

Abs

$$\exists x' x'' y'.\ x \mapsto x' * x' \mapsto x'' * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$\leadsto$$

$$\exists x'' y'.\ \mathsf{ls}\,(x, x'') * \mathsf{ls}\,(x'', 0) * y' \mapsto 0$$

$$\leadsto$$

$$\exists y'.\ \mathsf{ls}\,(x, 0) * y' \mapsto 0$$

$$\leadsto$$

$$\mathsf{ls}\,(x, 0) * \mathsf{true}$$

$$(\exists \vec{y'} x'.\, \Pi \wedge E \mapsto x' * x' \mapsto F * \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma)$$
$$(\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma, E, F))$$

$$(\exists \vec{y'} x'.\, \Pi \wedge \mathsf{ls}\,(E, x') * \mathsf{ls}\,(x', F) * \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{ls}\,(E, F) * \Sigma)$$
$$(\text{when } y' \notin \mathsf{FV}(\Pi, \Sigma, E, F))$$

$$(\exists \vec{y'} x'.\, \Pi \wedge x' \mapsto F * \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, \Pi \wedge \mathsf{true} * \Sigma)$$
$$(\text{when } x' \notin \mathsf{FV}(\Pi, \Sigma))$$

$$(\exists \vec{y'} x'.\, x' = E \wedge \Pi \wedge \Sigma) \quad \leadsto \quad (\exists \vec{y'}.\, (\Pi \wedge \Sigma)[E/x'])$$

Abs

$$\mathsf{Abs} \quad : \quad \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\} \rightarrow \mathcal{P}(\mathsf{SymHeap}) \cup \{\mathsf{err}\}$$

- Forgets the length of linked lists, and simplify quantifiers.
- Maps err to err.
- Defined by rewriting rules (true implications in sep. logic)

$$(\exists \vec{y'} x'.\, \mathit{\Pi} \wedge E \mapsto x' * x' \mapsto F * \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E, F) * \mathit{\Sigma})$$
$$(\text{when } x' \notin \mathsf{FV}(\mathit{\Pi}, \mathit{\Sigma}, E, F))$$

$$(\exists \vec{y'} x'.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E, x') * \mathsf{ls}\,(x', F) * \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \mathit{\Pi} \wedge \mathsf{ls}\,(E, F) * \mathit{\Sigma})$$
$$(\text{when } y' \notin \mathsf{FV}(\mathit{\Pi}, \mathit{\Sigma}, E, F))$$

$$(\exists \vec{y'} x'.\, \mathit{\Pi} \wedge x' \mapsto F * \mathit{\Sigma}) \quad \rightsquigarrow \quad (\exists \vec{y'}.\, \mathit{\Pi} \wedge \mathsf{true} * \mathit{\Sigma})$$
$$(\text{when } x' \notin \mathsf{FV}(\mathit{\Pi}, \mathit{\Sigma}))$$

$$(\exists \vec{y'} x'.\, x' = E \wedge \mathit{\Pi} \wedge \mathit{\Sigma}) \;\rightsquigarrow\; (\exists \vec{y'}.\, (\mathit{\Pi} \wedge \mathit{\Sigma})[E/x'])$$

HW

- Run the analyzer manually for freelist:
  - while(x!=0) { t=x; x=*t; free(t); t=0 }
  - Precondition: { ls(x,0) }.
- If necessary, add rewriting rules for abstraction and rearrangement.
- Run the analyzer for list reversal.

Reference

- Distefano et al's TACAS 2006 paper.
- Berdine et al's APLAS 2005 paper.