Static Enforcement of Security in Runtime Systems

Mathias V. Pedersen Aslan Askarov

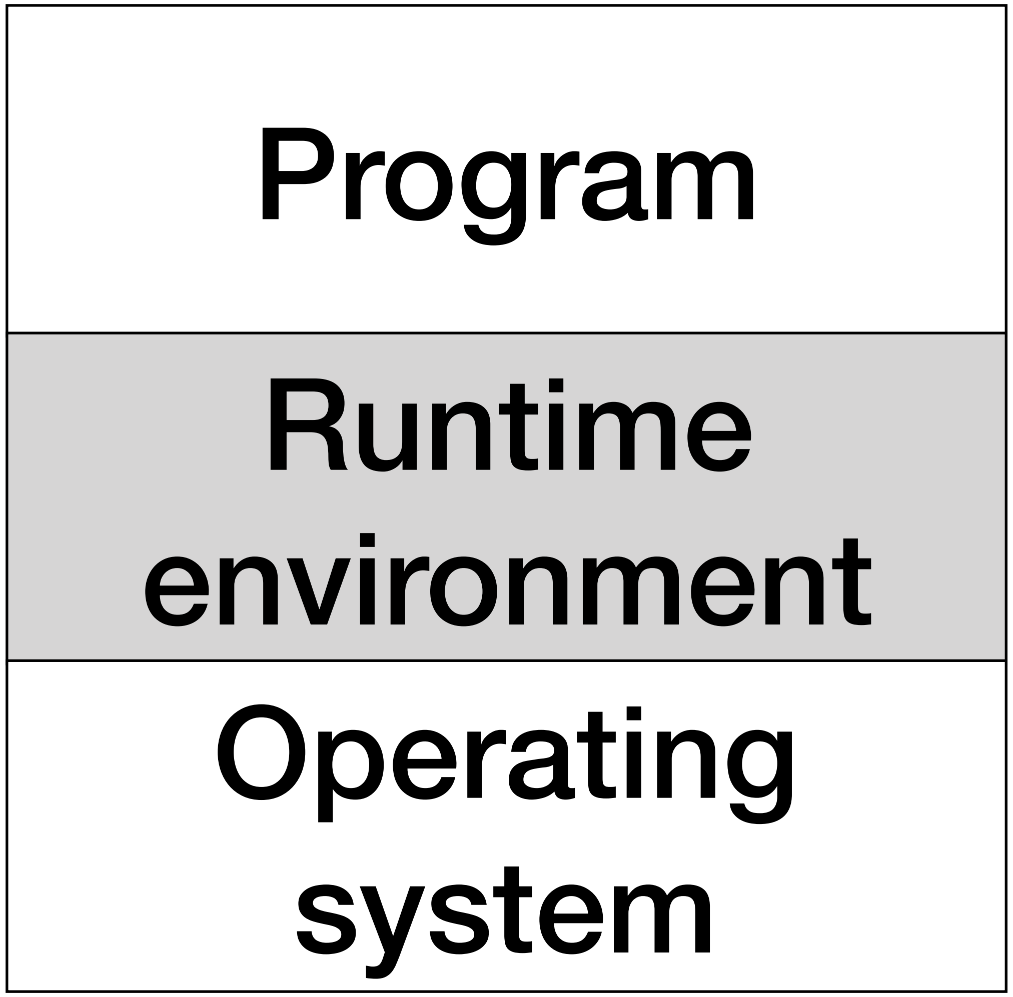
AARHUS UNIVERSITY

Motivation

Focus of this work:

Covert channels in programming language runtimes

Motivation



Program

Runtime environment

Operating system

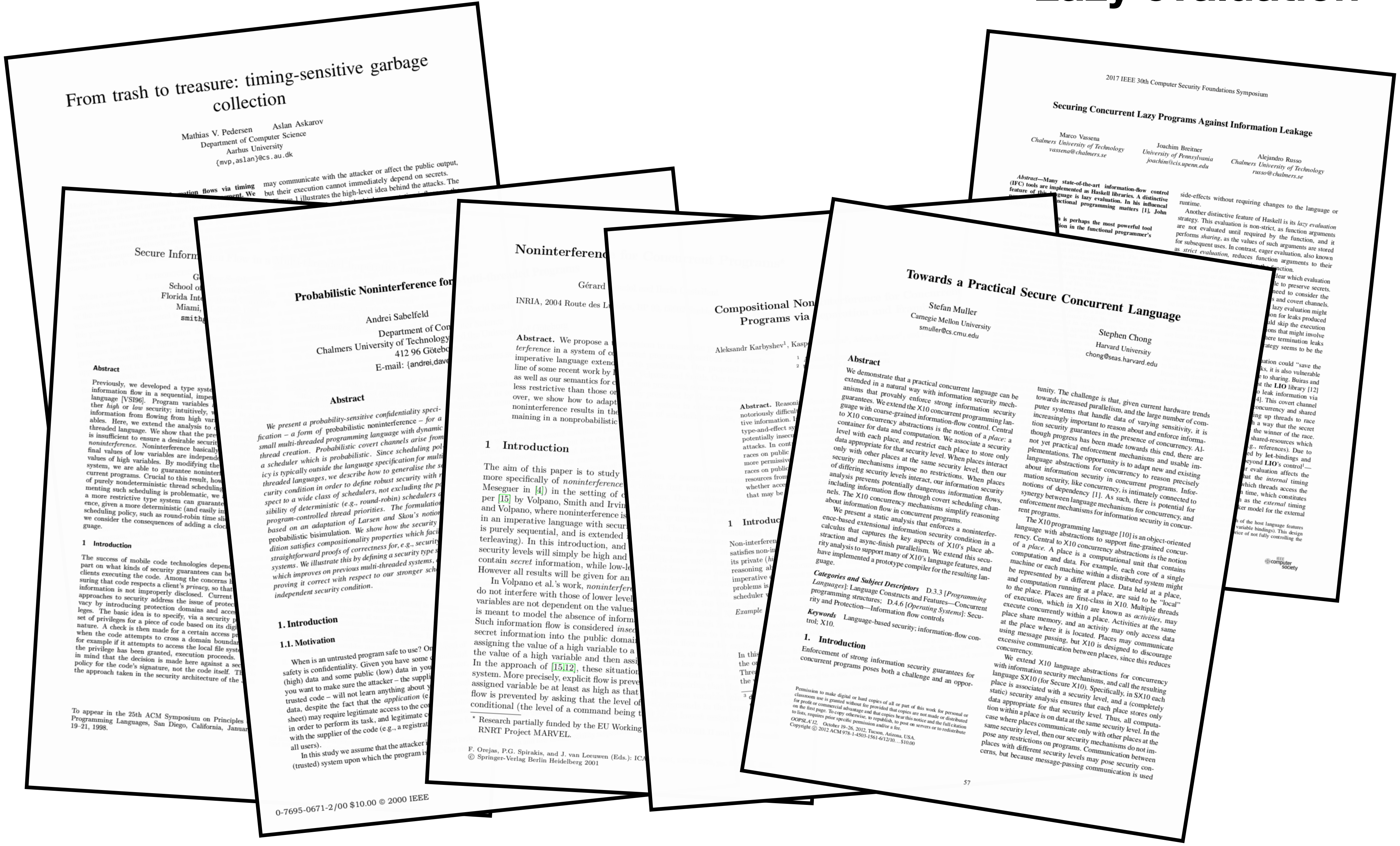
Contribution

A language to express and reason about information-flow control in implementations of runtime-related tasks (e.g., scheduling, garbage collection, sharing).

Information leaks via runtime

Garbage collection

Lazy evaluation



From trash to treasure: timing-sensitive garbage collection

Mathias V. Pedersen Aslan Askarov
Department of Computer Science
Aarhus University
{mvp,aslan}@cs.au.dk

Secure Inform

Probabilistic Noninterference for

Andrei Sabelfeld
Department of Com
Chalmers University of Technology
412 96 Götebo
E-mail: {andrei,dave

Noninterferenc

Gérard
INRIA, 2004 Route des L

Compositional Non
Programs via

Aleksandr Karbyshev[1], Kasp

Towards a Practical Secure Concurrent Language

Stefan Muller
Carnegie Mellon University
smuller@cs.cmu.edu

Stephen Chong
Harvard University
chong@seas.harvard.edu

2017 IEEE 30th Computer Security Foundations Symposium

Securing Concurrent Lazy Programs Against Information Leakage

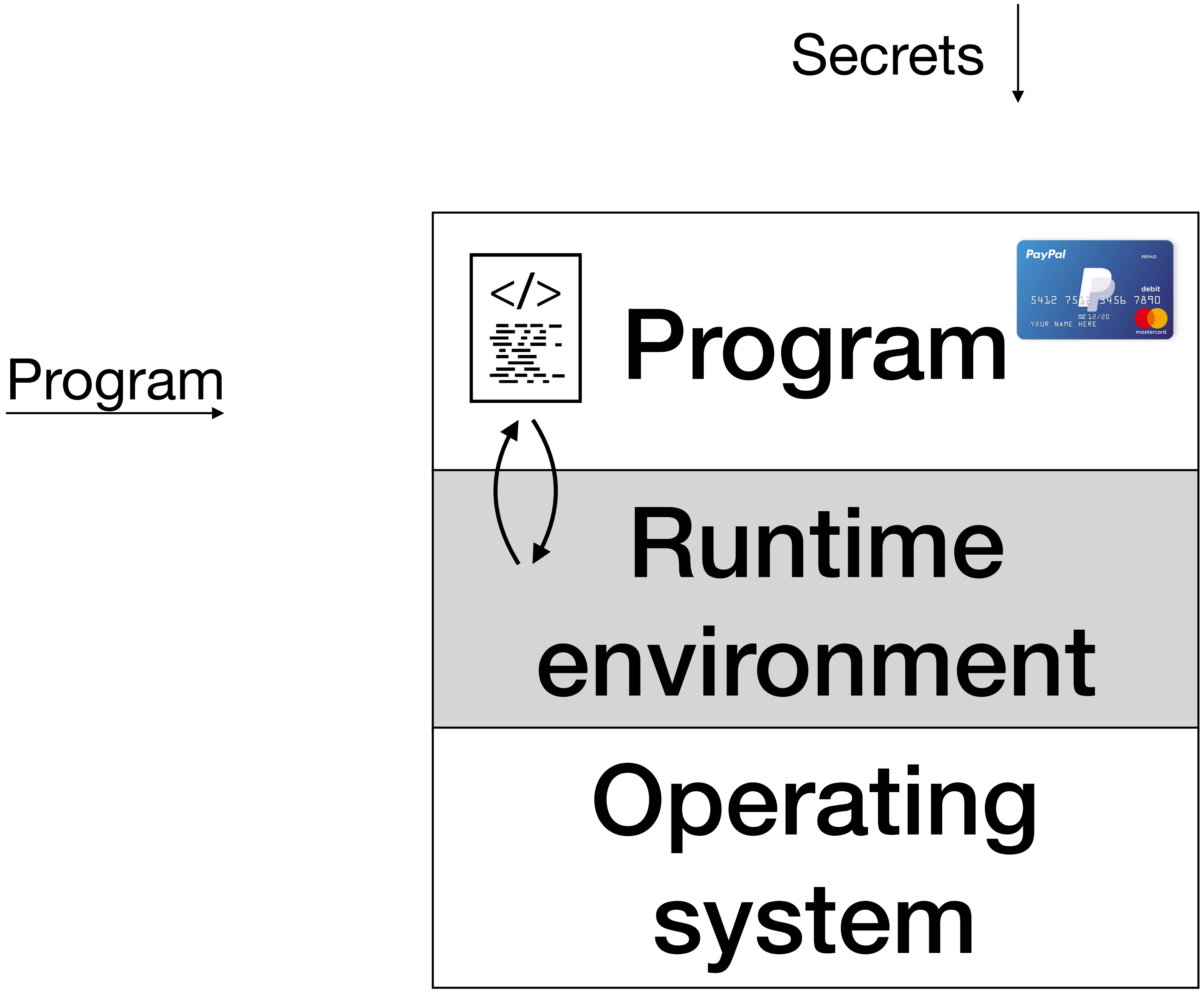
Marco Vassena
Chalmers University of Technology
vassena@chalmers.se

Joachim Breitner
University of Pennsylvania
joachim@cis.upenn.edu

Alejandro Russo
Chalmers University of Technology
russo@chalmers.se

Concurrency and scheduling

The setup



Secrets

Program

Program

Runtime environment

Operating system

Requirements

Wanted: A programming language for implementing runtimes

Must have: 1. Higher-order functions

2. Runtime type analysis

3. Heterogeneous arrays

And: Formal security guarantees

Task	Feature
Closures + thread scheduling	Higher-order functions
Object descriptors for GCs	Runtime type analysis
Modeling the stack	Heterogeneous arrays

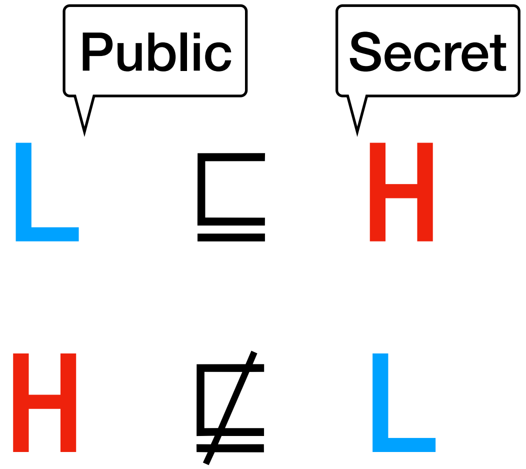
Necessary evils!

The rest of this talk …

- Example
- Typing the call stack
- Language
- Security guarantee
- Implementation and case studies

Example: Summing heterogeneous array

```
alloc()   def alloc() =
init()   → v := pack (int L, 0)
sum()              as ∃ t : type . t in
            p := malloc(5, v)
```



Example: Summing heterogeneous array

```
alloc()
init()
sum()
```

```
def alloc() =
  v := pack (int L,↓0)
          as ∃ t : type . t in
  p := malloc(5, v)
```

Public Secret

$L \sqsubseteq H$

$H \not\sqsubseteq L$

Example: Summing heterogeneous array

```
alloc()
init()
sum()
```

```
def alloc() =   ↓
  v := pack (int L, 0)
          as ∃ t : type . t in
  p := malloc(5, v)
```

Public

Secret

$$L \sqsubseteq H$$

$$H \not\sqsubseteq L$$

Example: Summing heterogeneous array

```
alloc()
init()
sum()
```

```
def alloc() =                ↓
  v := pack (int L, 0)
            as ∃ t : type . t in
  p := malloc(5, v)
```

Public Secret

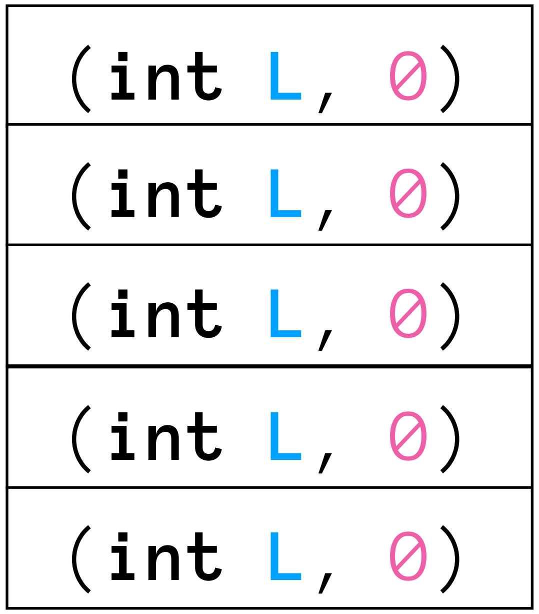
$$L \sqsubseteq H$$

$$H \not\sqsubseteq L$$

Example: Summing heterogeneous array

```
alloc()
init()
sum()
```

```
 def alloc() =
     v := pack (int L, 0)
              as ∃ t : type . t in
  →  p := malloc(5, v)
```



(int L, 0)
(int L, 0)
(int L, 0)
(int L, 0)
(int L, 0)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
→ ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

(int L, 0)
(int L, 0)
(int L, 0)
(int L, 0)
(int L, 0)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

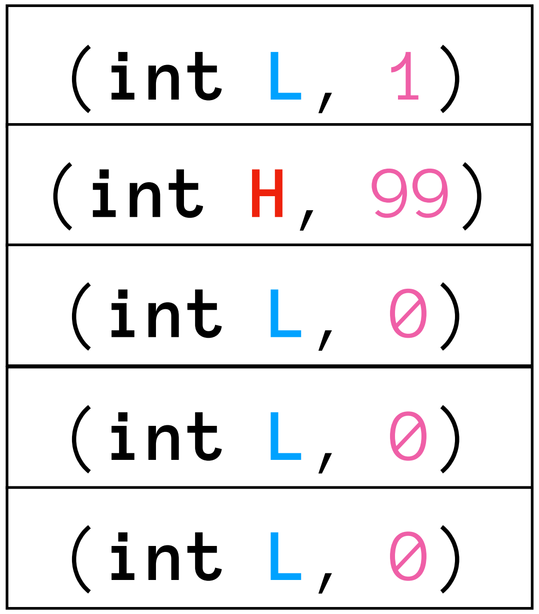
```
def init() =
  ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

(int L, 0)
(int L, 0)
(int L, 0)
(int L, 0)
(int L, 0)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
   ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
→  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
   ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
   ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
   ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```



(int L, 1)

(int H, 99)

(int L, 0)

(int L, 0)

(int L, 0)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
          as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  *(p + 0) := pack (int L, 1) as (∃ t : type . t)
  *(p + 1) := pack (int H, 99) as (∃ t : type . t)
  *(p + 2) := pack (int L, 3) as (∃ t : type . t)
  *(p + 3) := pack (int H, 101) as (∃ t : type . t)
  *(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
→ sumLows(&sum)
  print sum
```

```
def sumLows(psum: [int L] L) =
  sum : int L := 0 in
  i : int L := 0 in
  while i < length p do
    (t, v) := unpack *(p + i) in
      match t with
        int L → sum := sum + v
      | _ → skip
    i := i + 1
  *psum := sum
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
          as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
  def sum() =
    sum : int L := 0 in
    sumLows(&sum)
 → print sum
```

```
  def sumLows(psum: [int L] L) =
    sum : int L := 0 in
    i : int L := 0 in
    while i < length p do
      (t, v) := unpack ∗(p + i) in
        match t with
          int L → sum := sum + v
        | _ → skip
      i := i + 1
    ∗psum := sum
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
  sumLows(&sum)
  print sum
                    ↓
def sumLows(psum: [int L] L) =
  sum : int L := 0 in
  i : int L := 0 in
  while i < length p do
    (t, v) := unpack ∗(p + i) in
      match t with
        int L → sum := sum + v
      | _ → skip
    i := i + 1
  ∗psum := sum
```

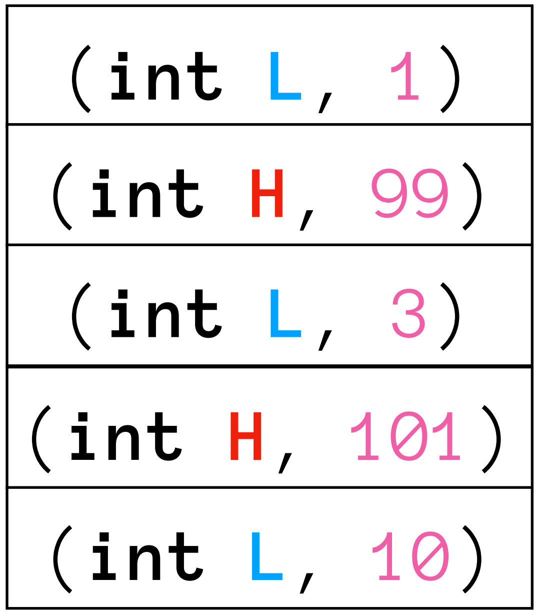
(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  *(p + 0) := pack (int L, 1) as (∃ t : type . t)
  *(p + 1) := pack (int H, 99) as (∃ t : type . t)
  *(p + 2) := pack (int L, 3) as (∃ t : type . t)
  *(p + 3) := pack (int H, 101) as (∃ t : type . t)
  *(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
  sumLows(&sum)
  print sum

def sumLows(psum: [int L] L) =
  sum : int L := 0 in
  i : int L := 0 in
→ while i < length p do
    (t, v) := unpack *(p + i) in
      match t with
        int L → sum := sum + v
      | _ → skip
    i := i + 1
  *psum := sum
```



(int L, 1)

(int H, 99)

(int L, 3)

(int H, 101)

(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
  sumLows(&sum)
  print sum
```

```
p : [(∃ t : type . t) L] L

∗(p + i) : (∃ t : type . t) L
```

```
    def sumLows(psum: [int L] L) =
      sum : int L := 0 in
      i : int L := 0 in
      while i < length p do
v : t →   (t, v) := unpack ∗(p + i) in
            match t with
              int L → sum := sum + v
            | _ → skip
          i := i + 1
      ∗psum := sum
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  *(p + 0) := pack (int L, 1) as (∃ t : type . t)
  *(p + 1) := pack (int H, 99) as (∃ t : type . t)
  *(p + 2) := pack (int L, 3) as (∃ t : type . t)
  *(p + 3) := pack (int H, 101) as (∃ t : type . t)
  *(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
  sumLows(&sum)
  print sum
```

```
def sumLows(psum: [int L] L) =
  sum : int L := 0 in
  i : int L := 0 in
  while i < length p do
    (t, v) := unpack *(p + i) in
   →  match t with
        int L → sum := sum + v
      | _ → skip
    i := i + 1
  *psum := sum
```

```
p : [(∃ t : type . t) L] L

*(p + i) : (∃ t : type . t) L
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  *(p + 0) := pack (int L, 1) as (∃ t : type . t)
  *(p + 1) := pack (int H, 99) as (∃ t : type . t)
  *(p + 2) := pack (int L, 3) as (∃ t : type . t)
  *(p + 3) := pack (int H, 101) as (∃ t : type . t)
  *(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
  sumLows(&sum)
  print sum
```

```
p : [(∃ t : type . t) L] L

*(p + i) : (∃ t : type . t) L
```

```
      def sumLows(psum: [int L] L) =
        sum : int L := 0 in
        i : int L := 0 in
        while i < length p do
          (t, v) := unpack *(p + i) in
            match t with
v : int L →   int L → sum := sum + v
            | _ → skip
          i := i + 1
        *psum := sum
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
def sum() =
  sum : int L := 0 in
  sumLows(&sum)
  print sum
```

```
p : [(∃ t : type . t) L] L

∗(p + i) : (∃ t : type . t) L
```

```
def sumLows(psum: [int L] L) =
  sum : int L := 0 in
  i : int L := 0 in
  while i < length p do
    (t, v) := unpack ∗(p + i) in
      match t with
        int L → sum := sum + v
      | _ → skip
    i := i + 1
→ ∗psum := sum
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

Example: Summing heterogeneous array

```
def alloc() =
  v := pack (int L, 0)
         as ∃ t : type . t in
  p := malloc(5, v)
```

```
def init() =
  ∗(p + 0) := pack (int L, 1) as (∃ t : type . t)
  ∗(p + 1) := pack (int H, 99) as (∃ t : type . t)
  ∗(p + 2) := pack (int L, 3) as (∃ t : type . t)
  ∗(p + 3) := pack (int H, 101) as (∃ t : type . t)
  ∗(p + 4) := pack (int L, 10) as (∃ t : type . t)
```

```
  def sum() =
    sum : int L := 0 in
    sumLows(&sum)
  → print sum
```

14

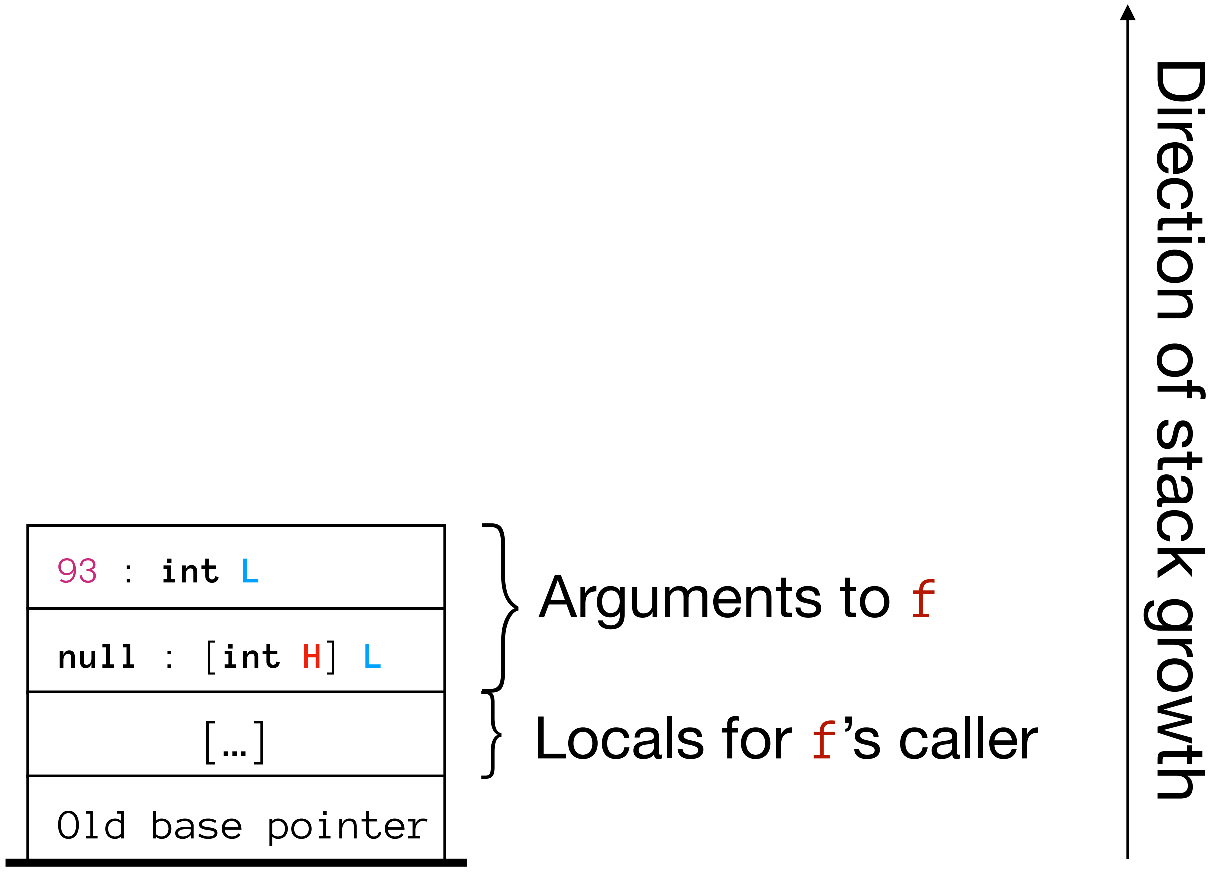
```
p : [(∃ t : type . t) L] L

∗(p + i) : (∃ t : type . t) L
```

```
def sumLows(psum: [int L] L) =
  sum : int L := 0 in
  i : int L := 0 in
  while i < length p do
    (t, v) := unpack ∗(p + i) in
      match t with
        int L → sum := sum + v
      | _ → skip
    i := i + 1
  ∗psum := sum
```

(int L, 1)
(int H, 99)
(int L, 3)
(int H, 101)
(int L, 10)

The call stack as a heterogeneous array

```
def g(q : int L, r : [[int H] L] L) =
  s : int H := 42 in
  ...

def f(x : int L, y : [int H] L) =
  z : int H := x in
  w : [[int H] L] L := &y in
  g(99, w)


→ f(93, null)
```



93 : int L

null : [int H] L

[…]

Old base pointer

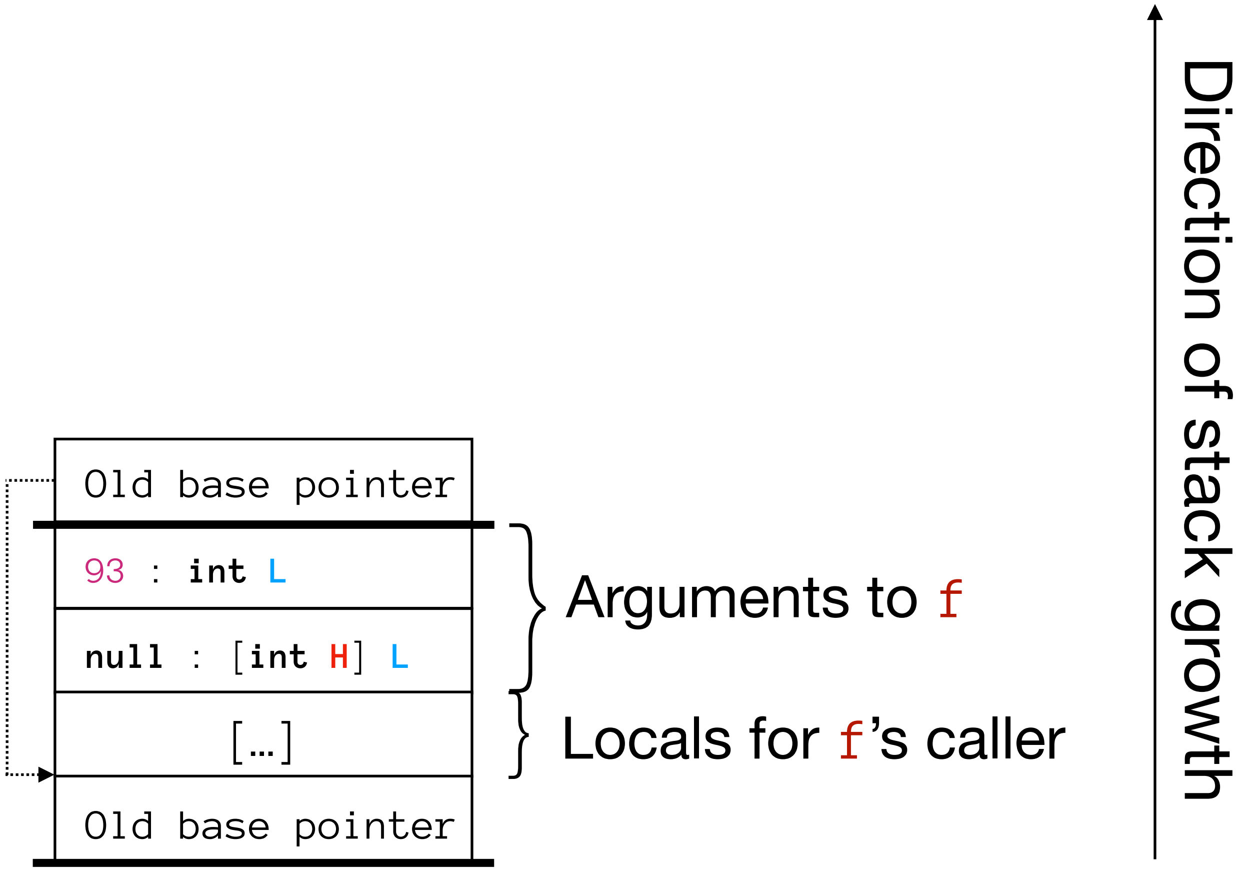
Arguments to f

Locals for f's caller

Direction of stack growth

The call stack as a heterogeneous array

```
  def g(q : int L, r : [[int H] L] L) =
    s : int H := 42 in
    ...

→ def f(x : int L, y : [int H] L) =
    z : int H := x in
    w : [[int H] L] L := &y in
    g(99, w)


  f(93, null)
```



```
Old base pointer
93 : int L
null : [int H] L
[…]
Old base pointer
```

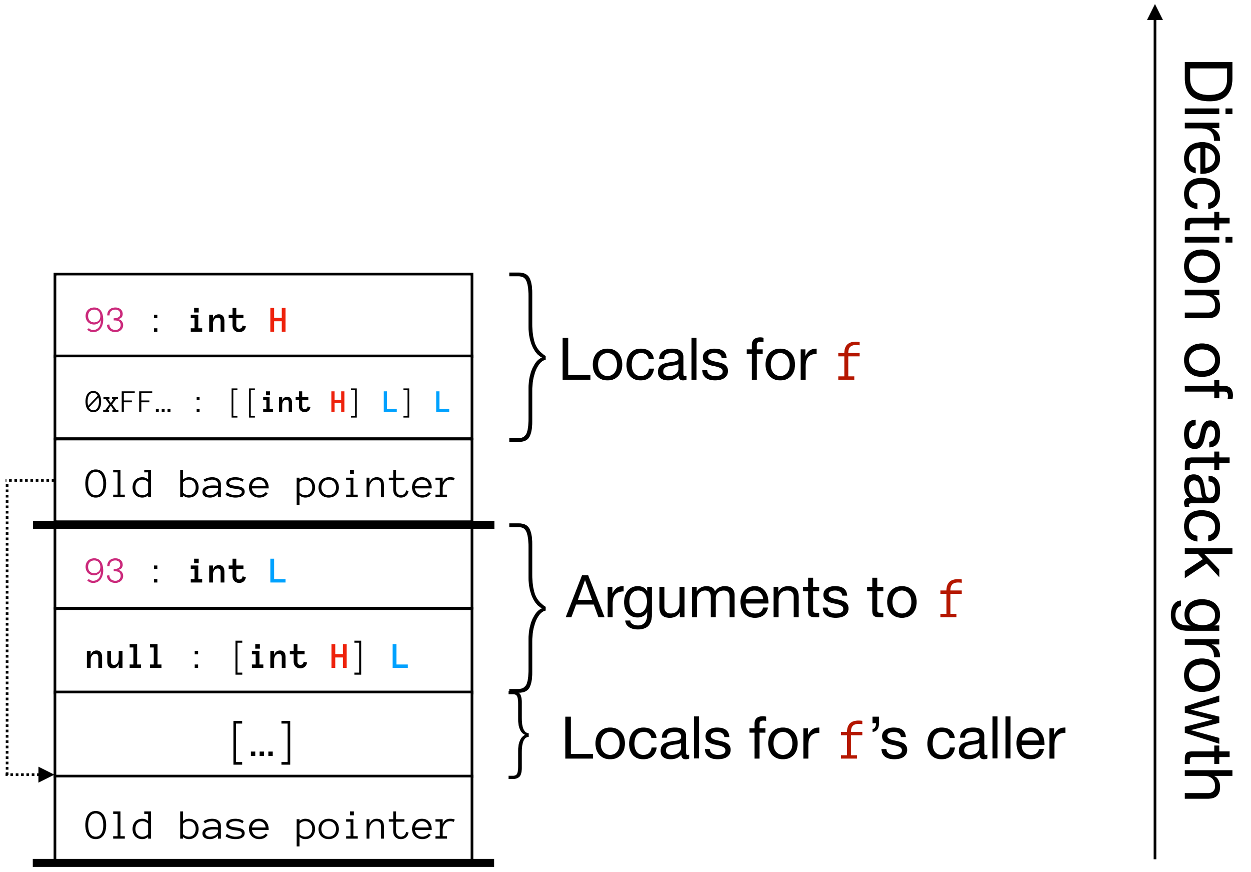
Arguments to f

Locals for f's caller

Direction of stack growth

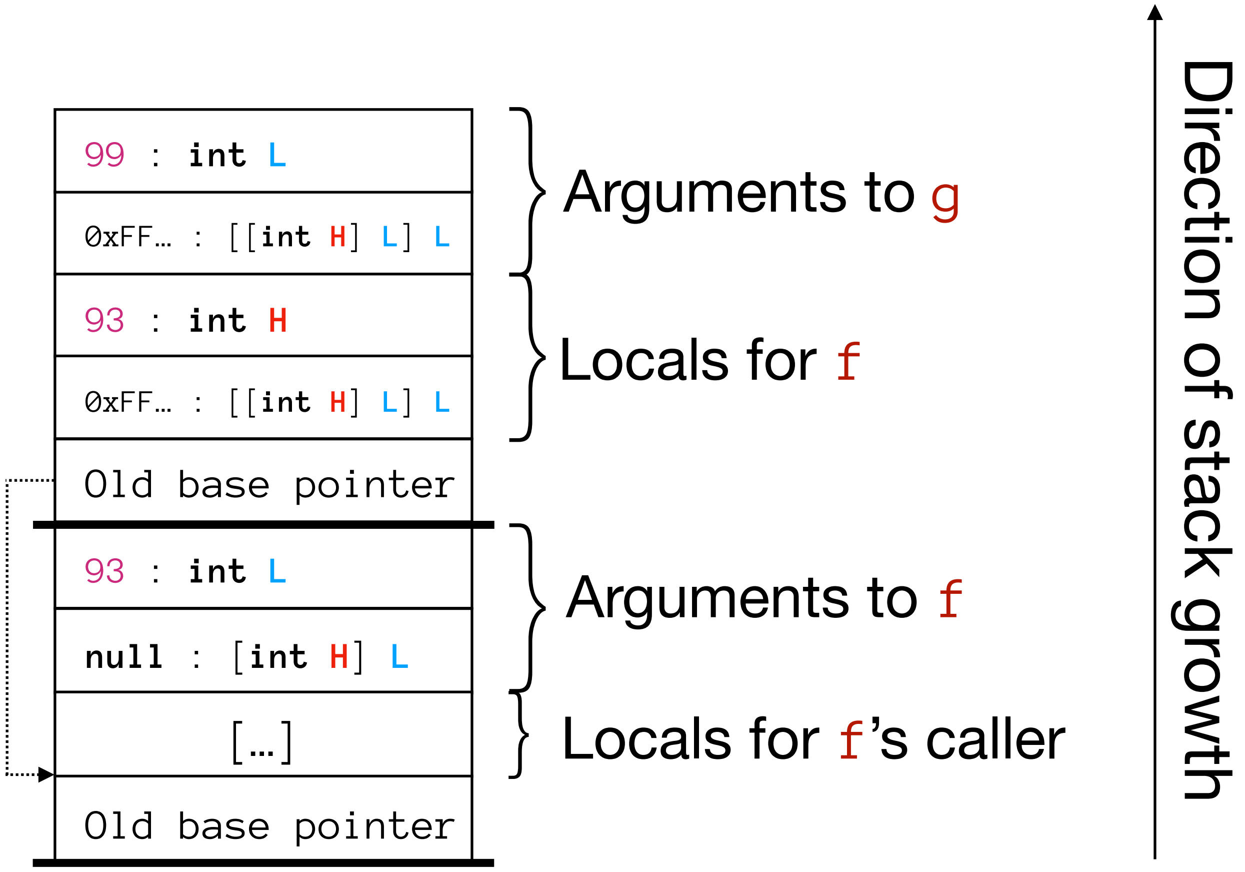
The call stack as a heterogeneous array

```
  def g(q : int L, r : [[int H] L] L) =
    s : int H := 42 in
    ...

  def f(x : int L, y : [int H] L) =
    z : int H := x in
    w : [[int H] L] L := &y in
→   g(99, w)


  f(93, null)
```



The call stack as a heterogeneous array

```
  def g(q : int L, r : [[int H] L] L) =
    s : int H := 42 in
    ...

  def f(x : int L, y : [int H] L) =
    z : int H := x in
    w : [[int H] L] L := &y in
→   g(99, w)


  f(93, null)
```



99 : int L

0xFF… : [[int H] L] L

93 : int H

0xFF… : [[int H] L] L

Old base pointer

93 : int L

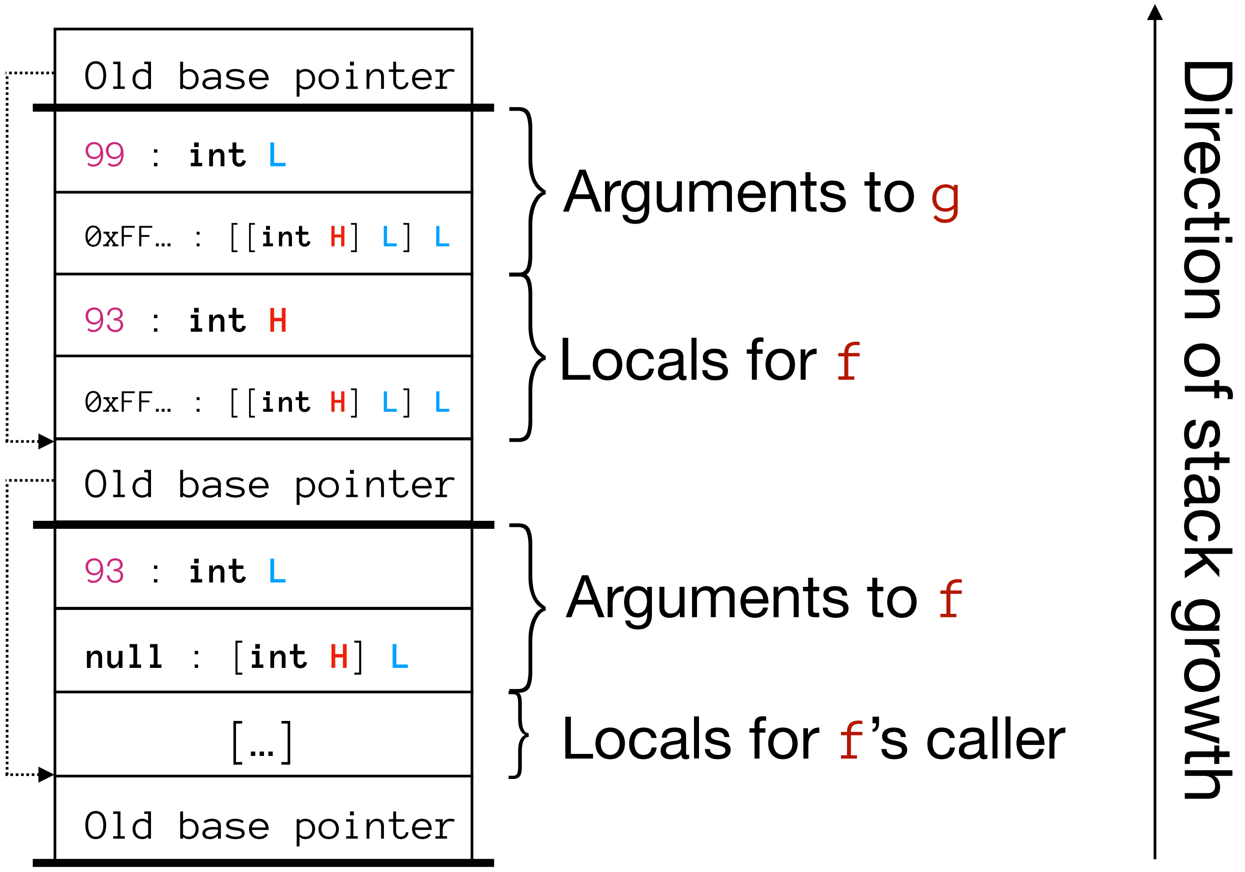
null : [int H] L

[…]

Old base pointer

The call stack as a heterogeneous array

```
→ def g(q : int L, r : [[int H] L] L) =
    s : int H := 42 in
    ...

  def f(x : int L, y : [int H] L) =
    z : int H := x in
    w : [[int H] L] L := &y in
    g(99, w)


  f(93, null)
```



Old base pointer

99 : int L

0xFF… : [[int H] L] L

93 : int H

0xFF… : [[int H] L] L

Old base pointer

93 : int L

null : [int H] L

[…]

Old base pointer

Arguments to g

Locals for f

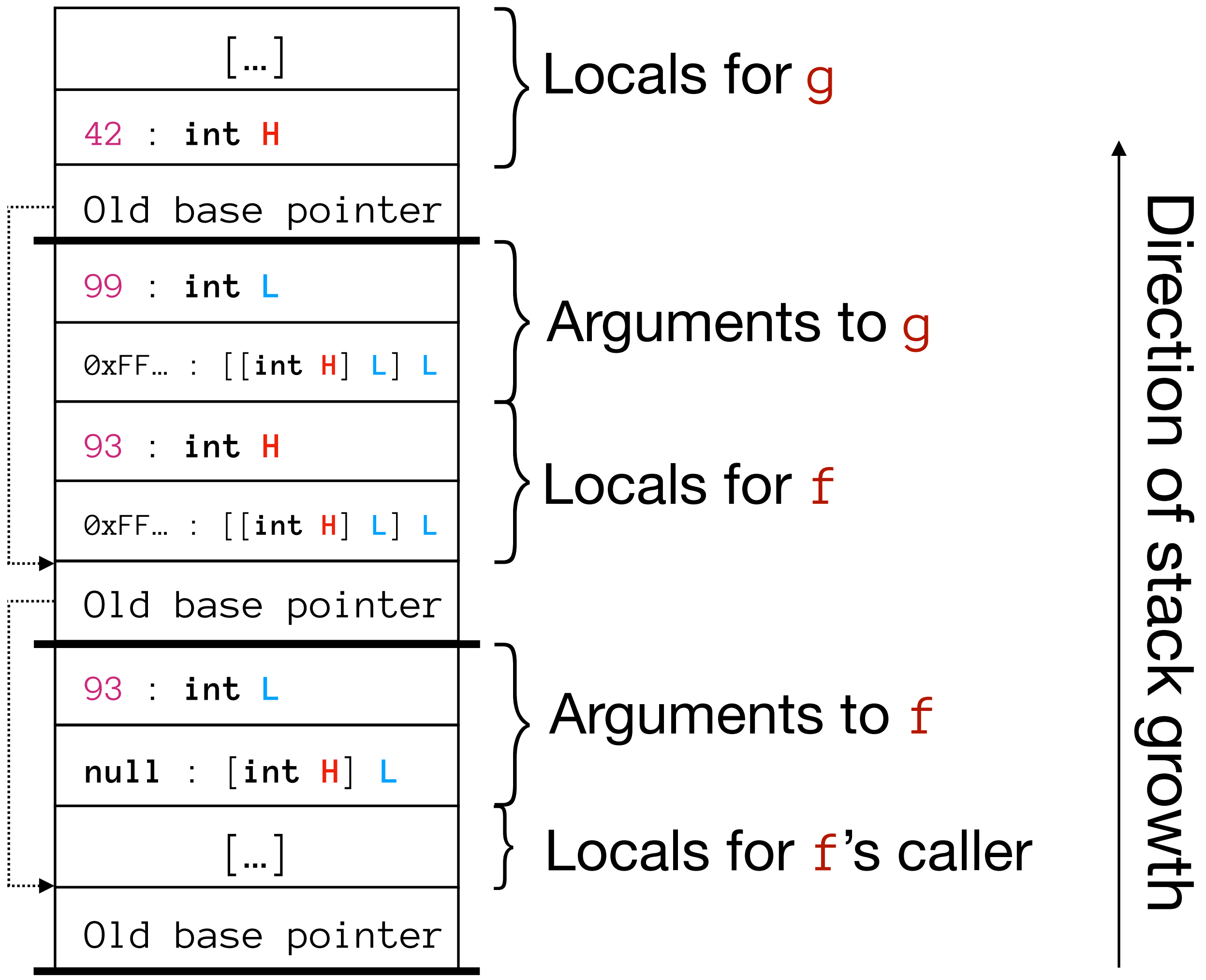
Arguments to f

Locals for f's caller

Direction of stack growth

The call stack as a heterogeneous array

```
  def g(q : int L, r : [[int H] L] L) =
    s : int H := 42 in
→   ...

  def f(x : int L, y : [int H] L) =
    z : int H := x in
    w : [[int H] L] L := &y in
    g(99, w)


  f(93, null)
```



[…]

42 : int H

Old base pointer

99 : int L

0xFF… : [[int H] L] L

93 : int H

0xFF… : [[int H] L] L

Old base pointer

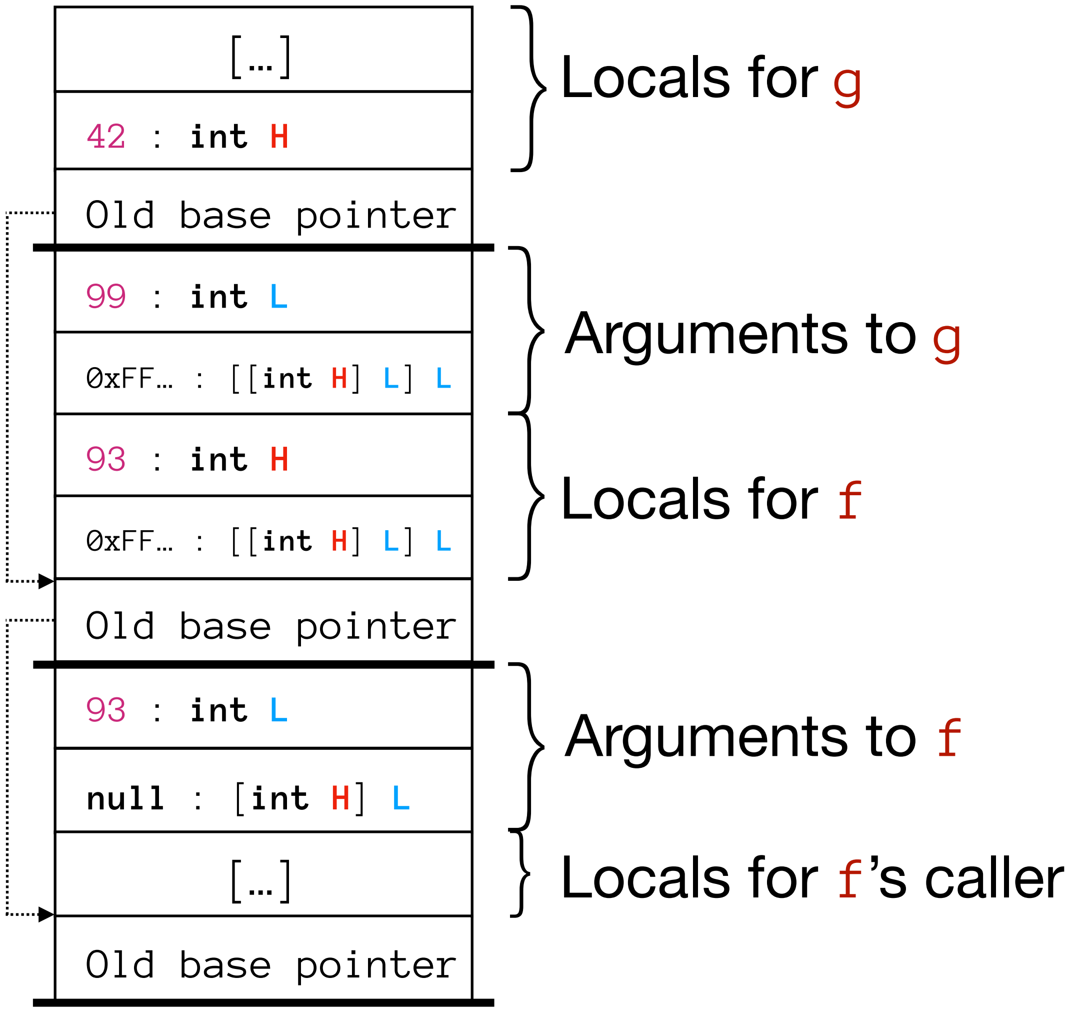
93 : int L

null : [int H] L

[…]

Old base pointer

The call stack as a heterogeneous array



[…]

42 : int H

Old base pointer

99 : int L

0xFF… : [[int H] L] L

93 : int H

0xFF… : [[int H] L] L

Old base pointer

93 : int L

null : [int H] L

[…]

Old base pointer

Locals for g

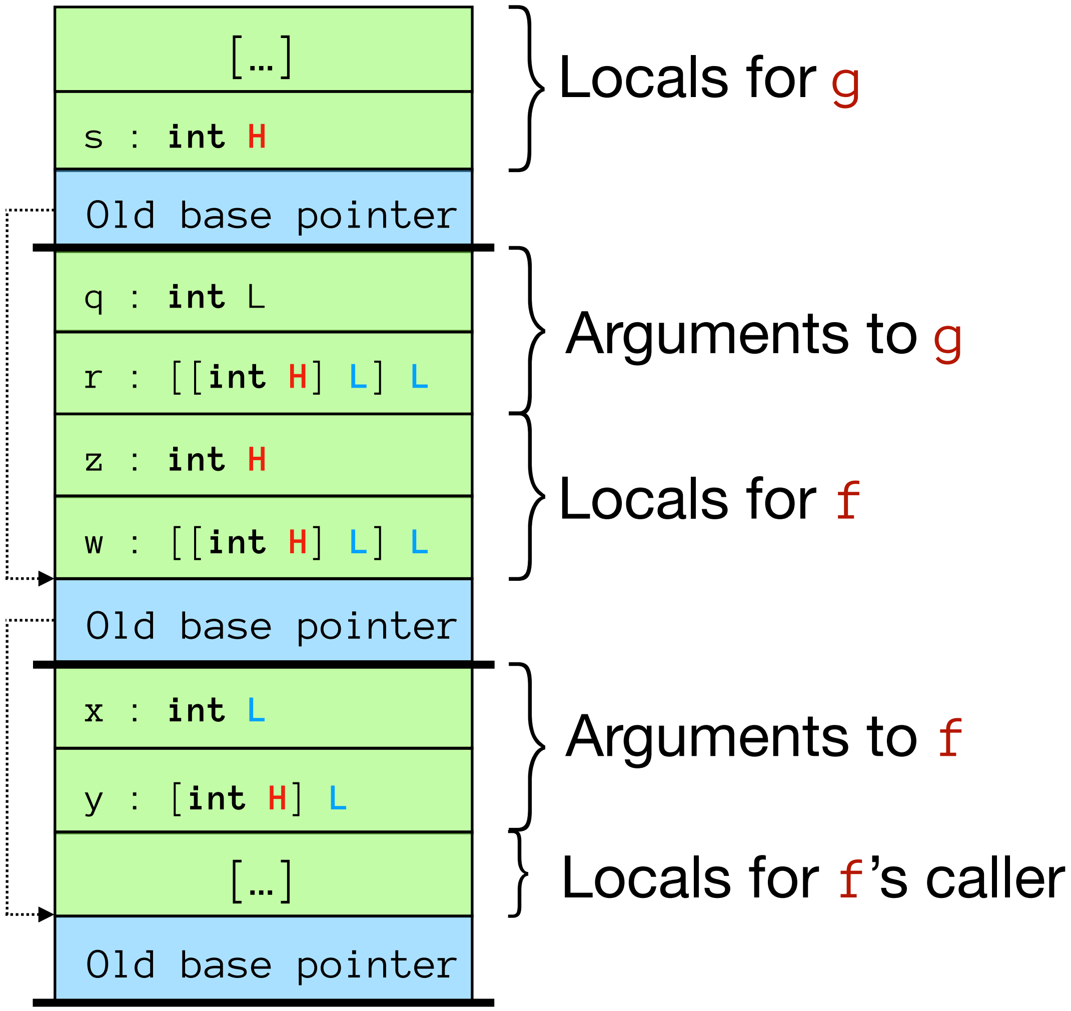
Arguments to g

Locals for f

Arguments to f

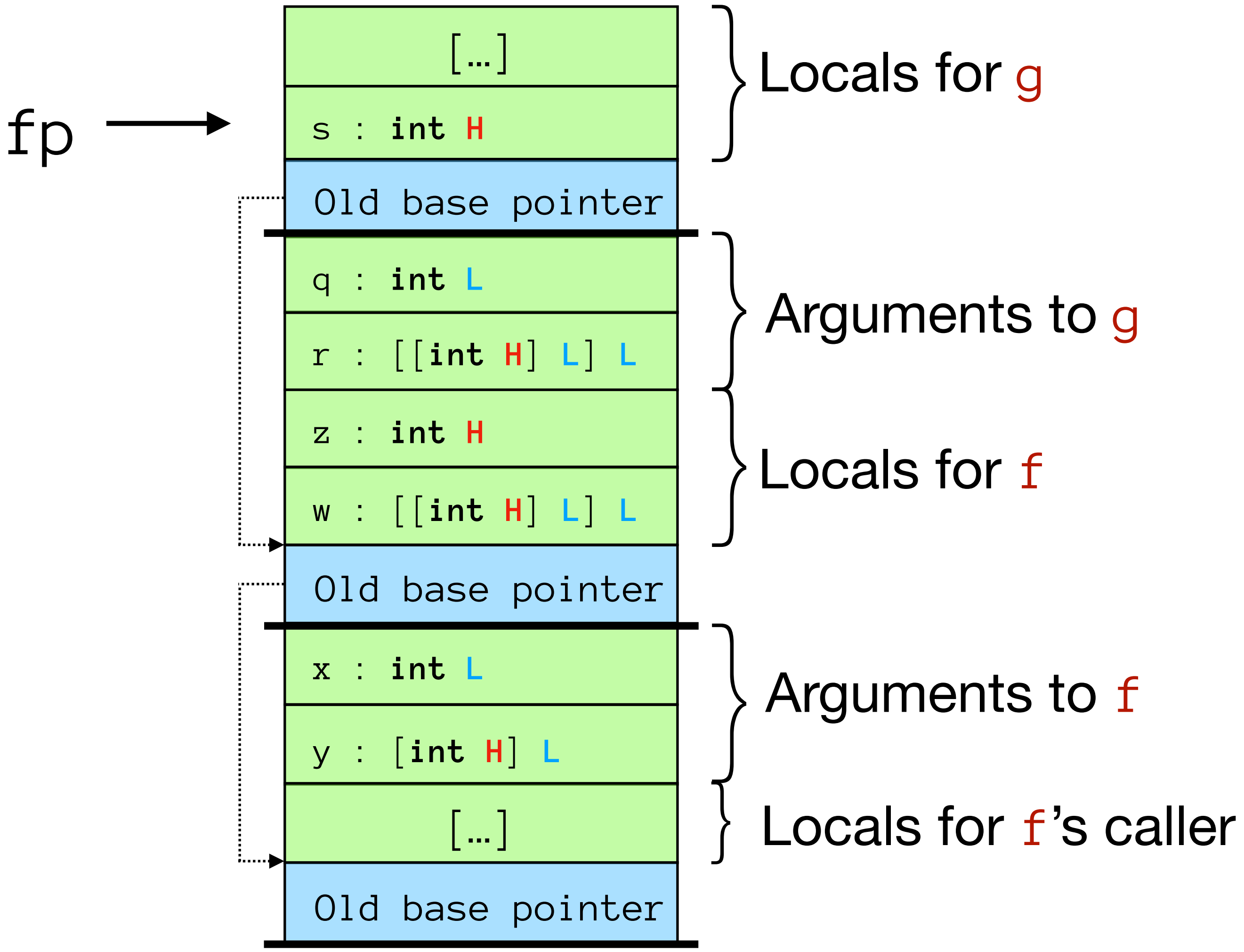
Locals for f’s caller

Typing the stack pointer



```
μ a : type . ∃ b : type . [a ∗ b]
```

Typing the stack pointer



fp

[…]

s : int H

Old base pointer

q : int L

r : [[int H] L] L

z : int H

w : [[int H] L] L

Old base pointer

x : int L

y : [int H] L

[…]

Old base pointer

Locals for g

Arguments to g

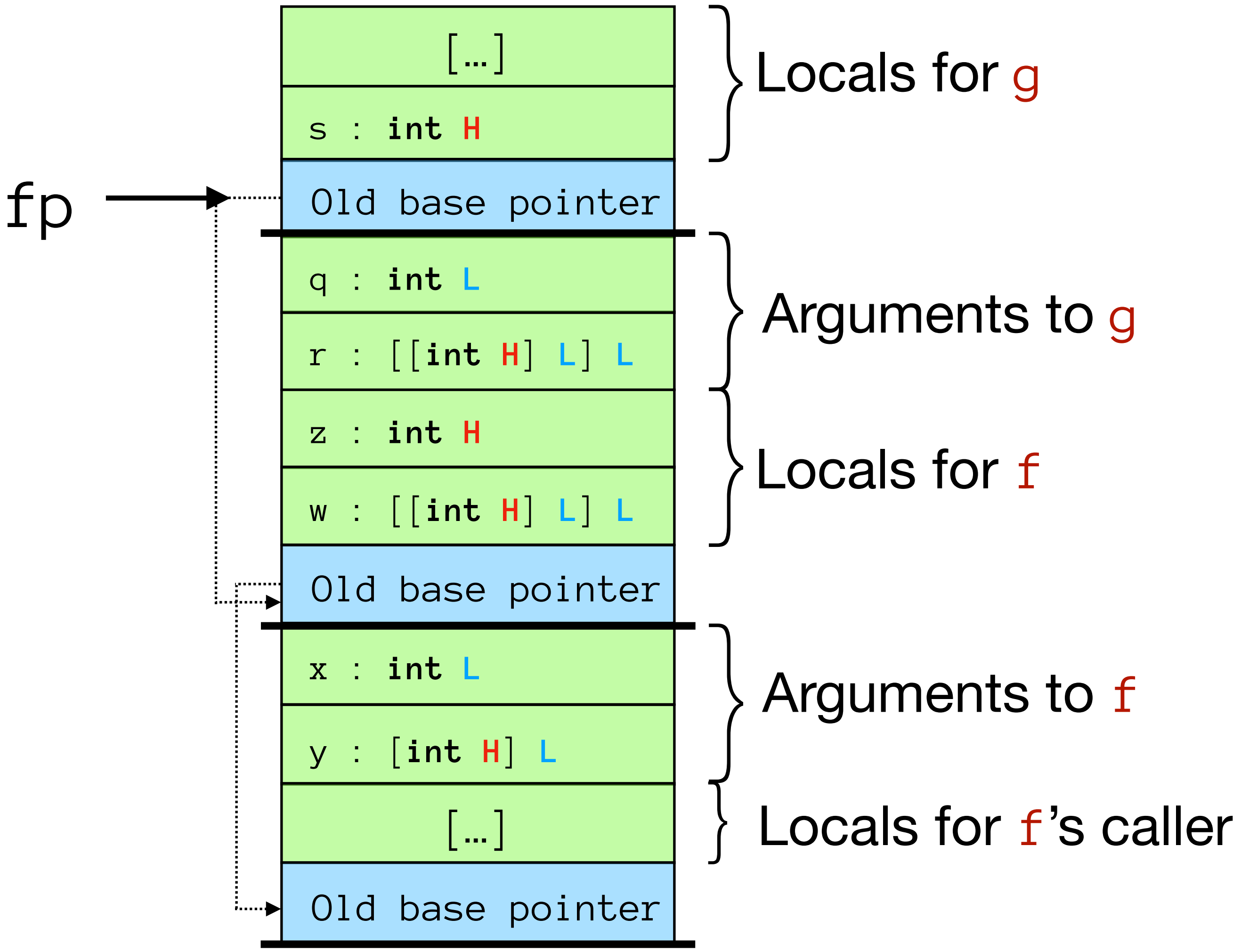
Locals for f

Arguments to f

Locals for f’s caller

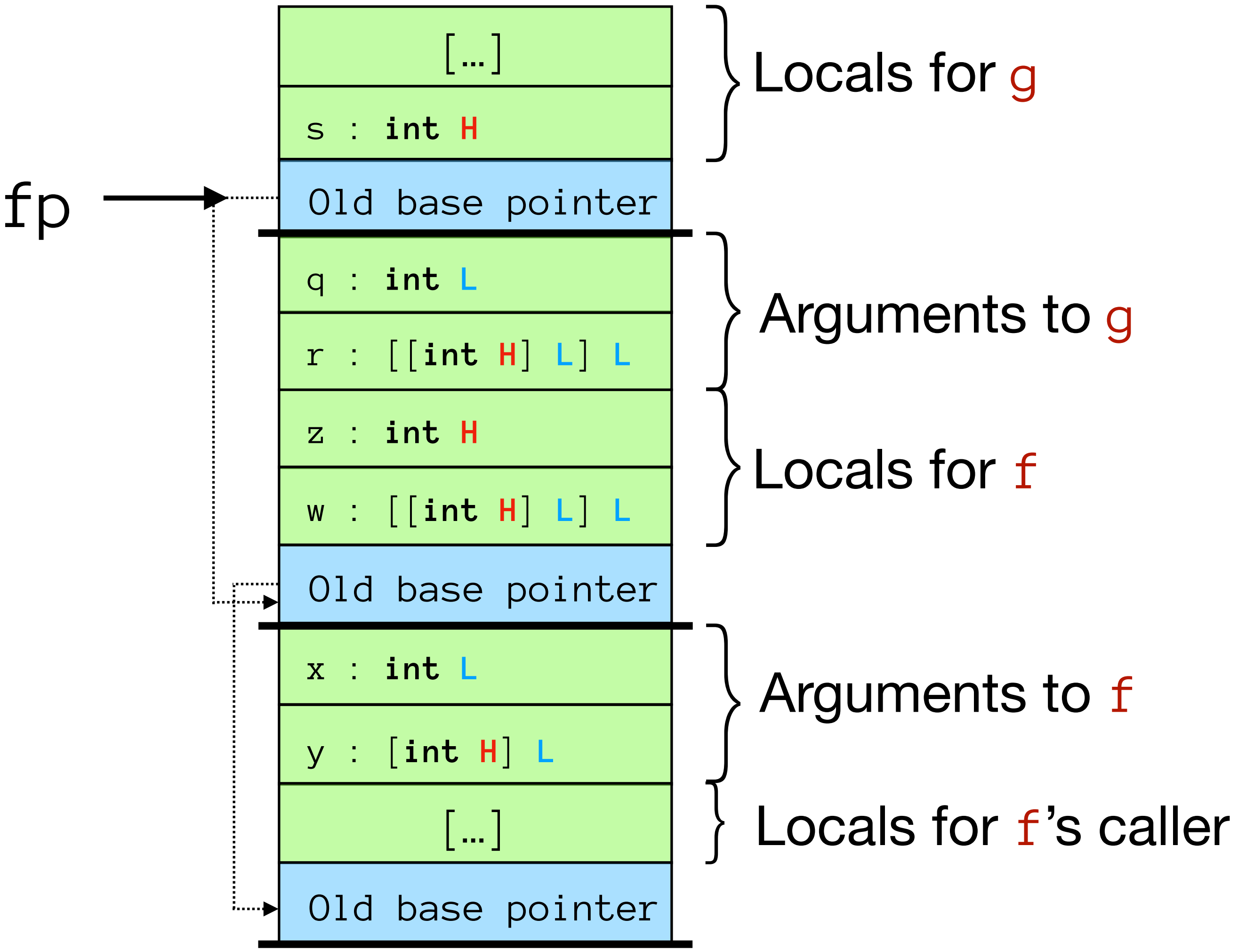
```
fp : μ a : type . ∃ b : type . [a ∗ b]
```

Typing the stack pointer



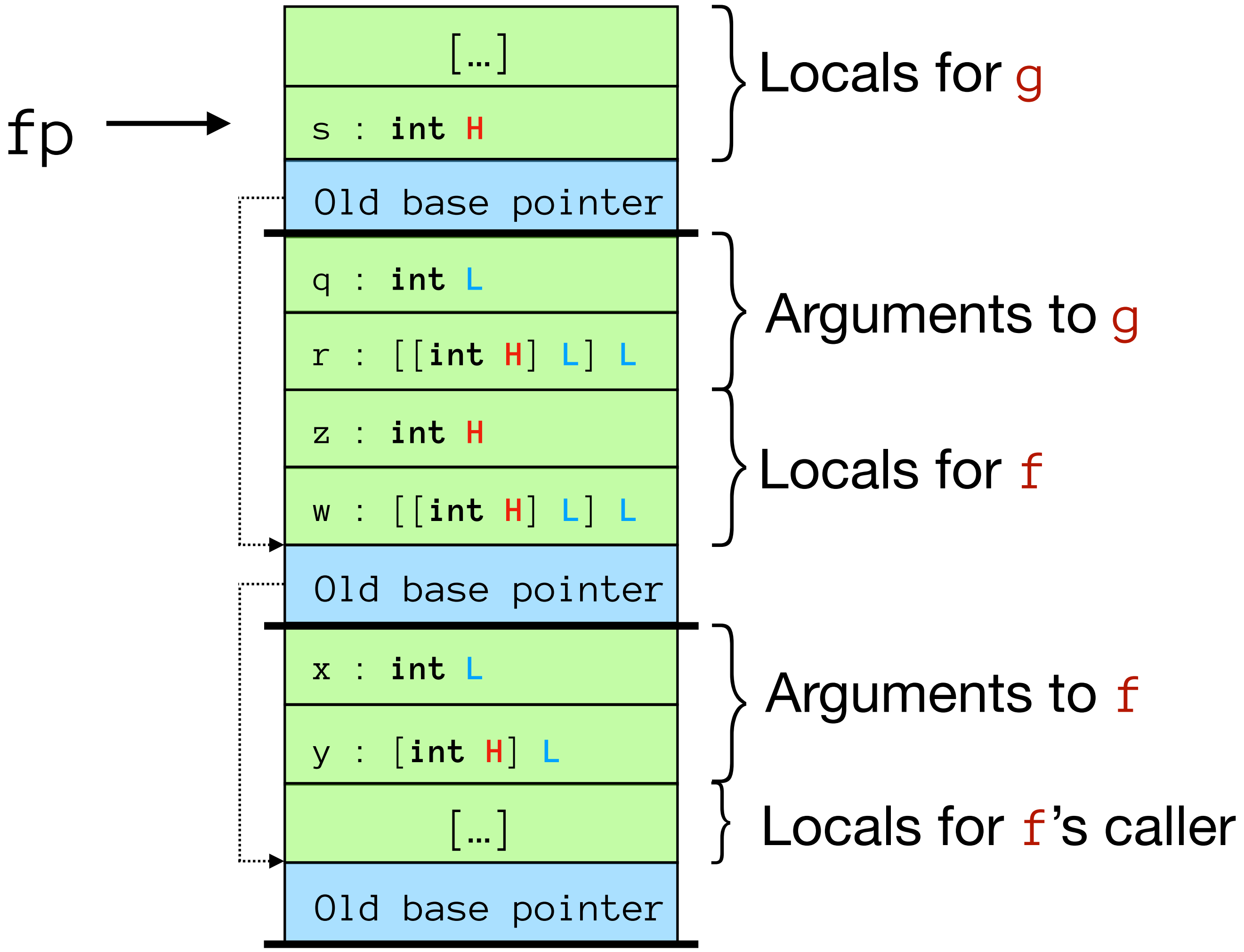
fp : μ a : **type** . ∃ b : **type** . [a ∗ b]

Typing the stack pointer



fp : μ a : **type** . ∃ b : **type** . a @ b

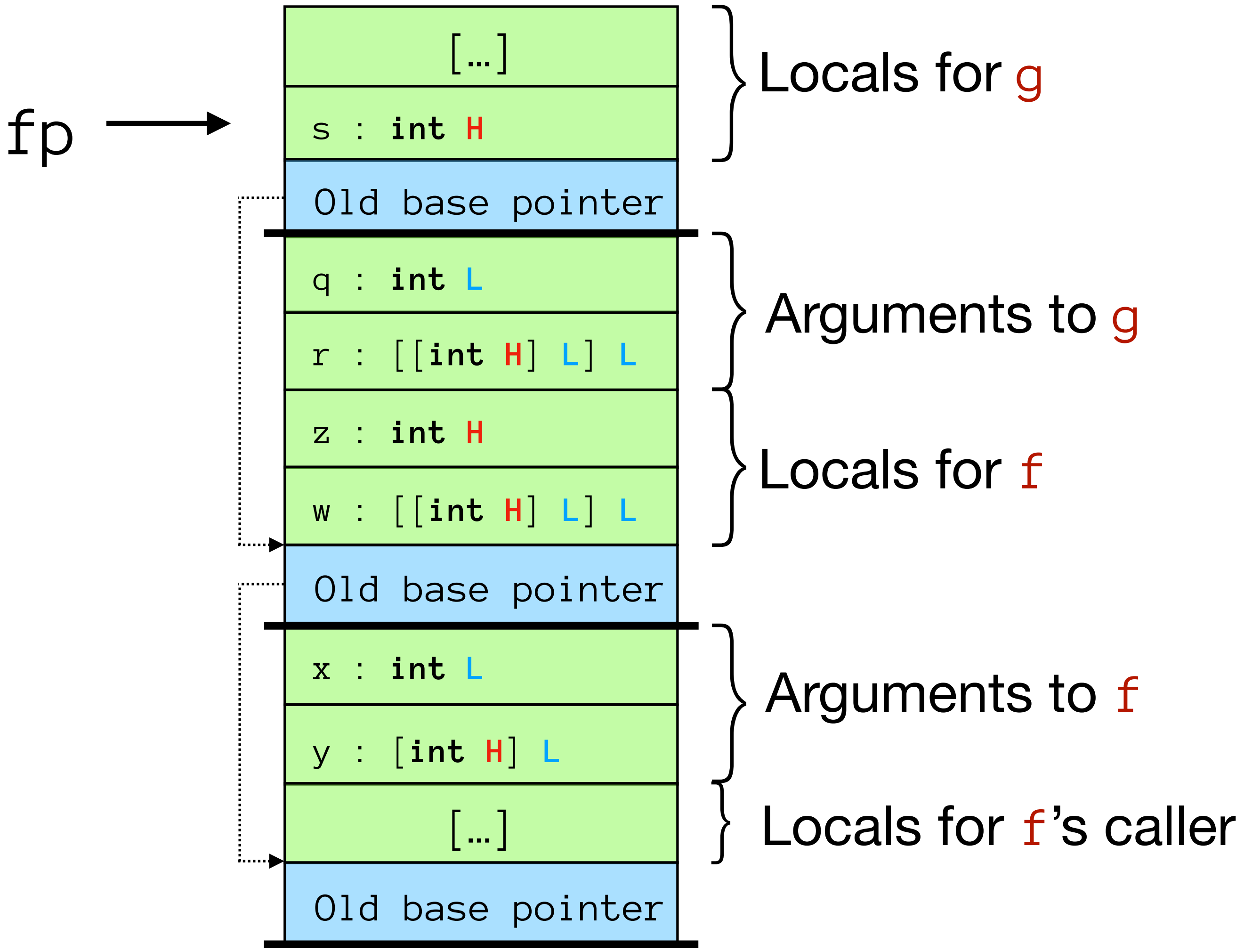
Typing the stack pointer



```
fp : μ a : type . ∃ b : type . a @ b
```

Typing the stack pointer

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$



fp

[…]

s : int H

Old base pointer

q : int L

r : [[int H] L] L

z : int H

w : [[int H] L] L

Old base pointer

x : int L

y : [int H] L

[…]

Old base pointer

Locals for g

Arguments to g

Locals for f

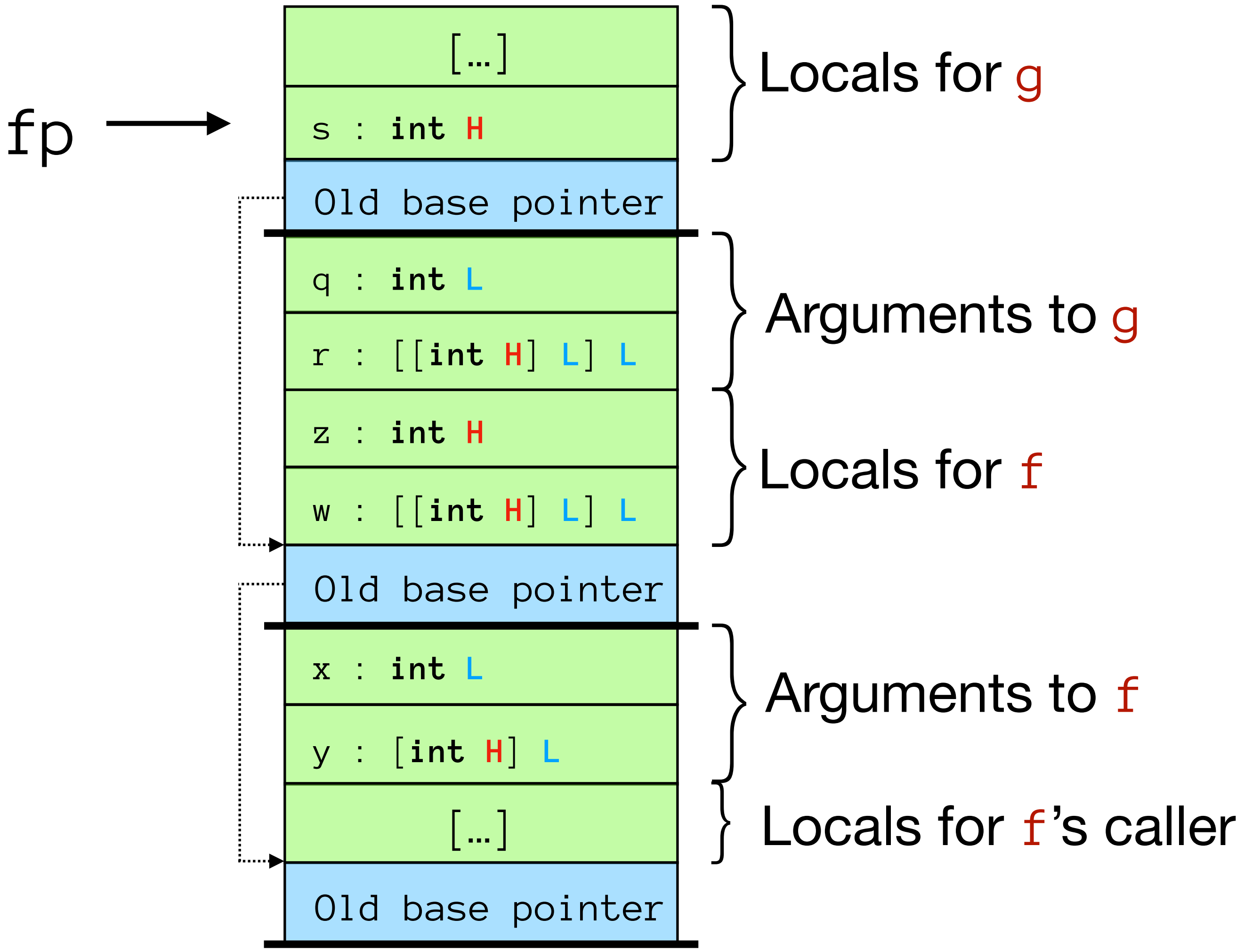
Arguments to f

Locals for f’s caller

$$fp : T_{st}$$

Typing the stack pointer

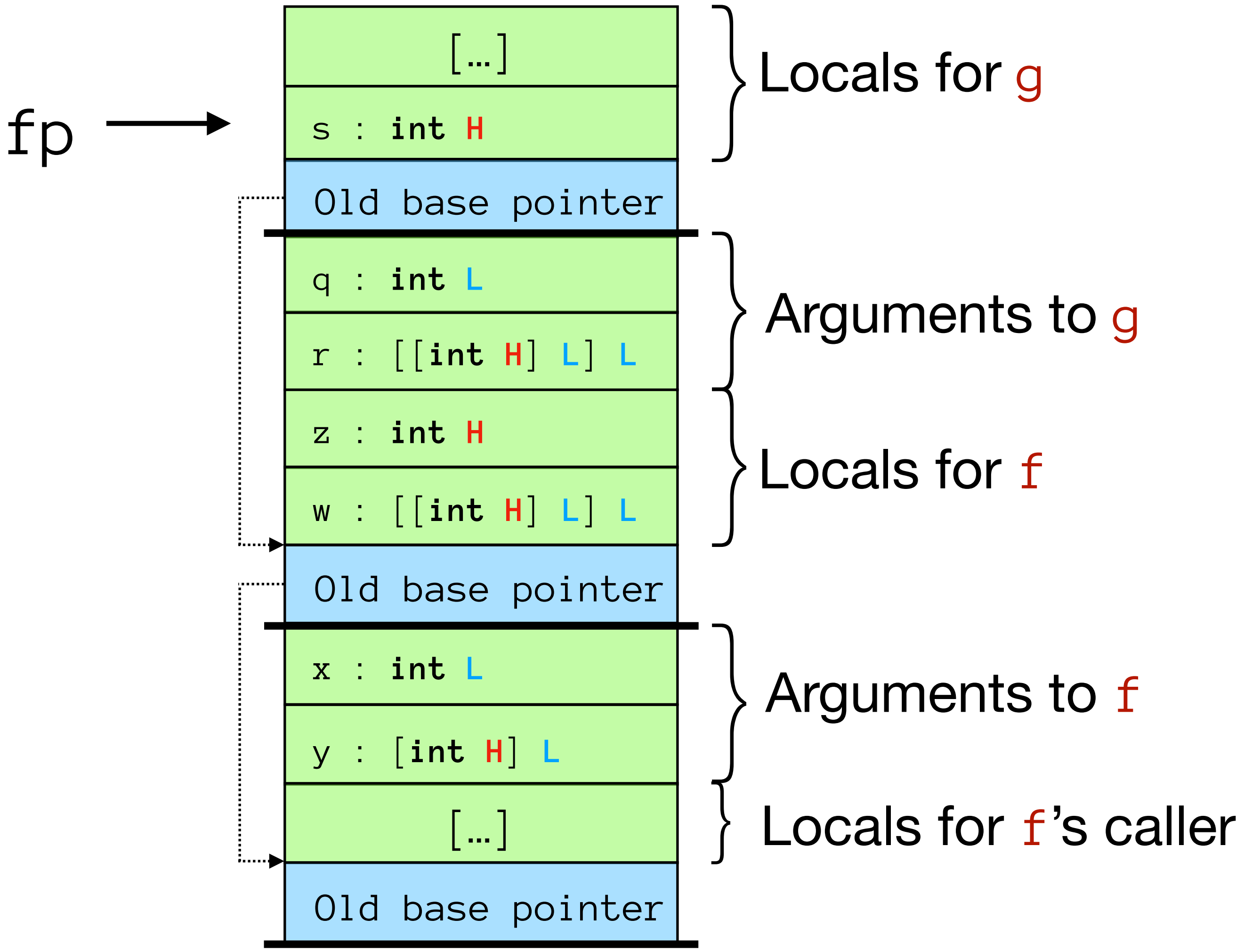
$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$



$$\mathbf{unroll}\ fp\ :\ (\exists\ b : \mathbf{type}\ .\ a\ @\ b)[T_{st}\ /\ a]$$

Typing the stack pointer

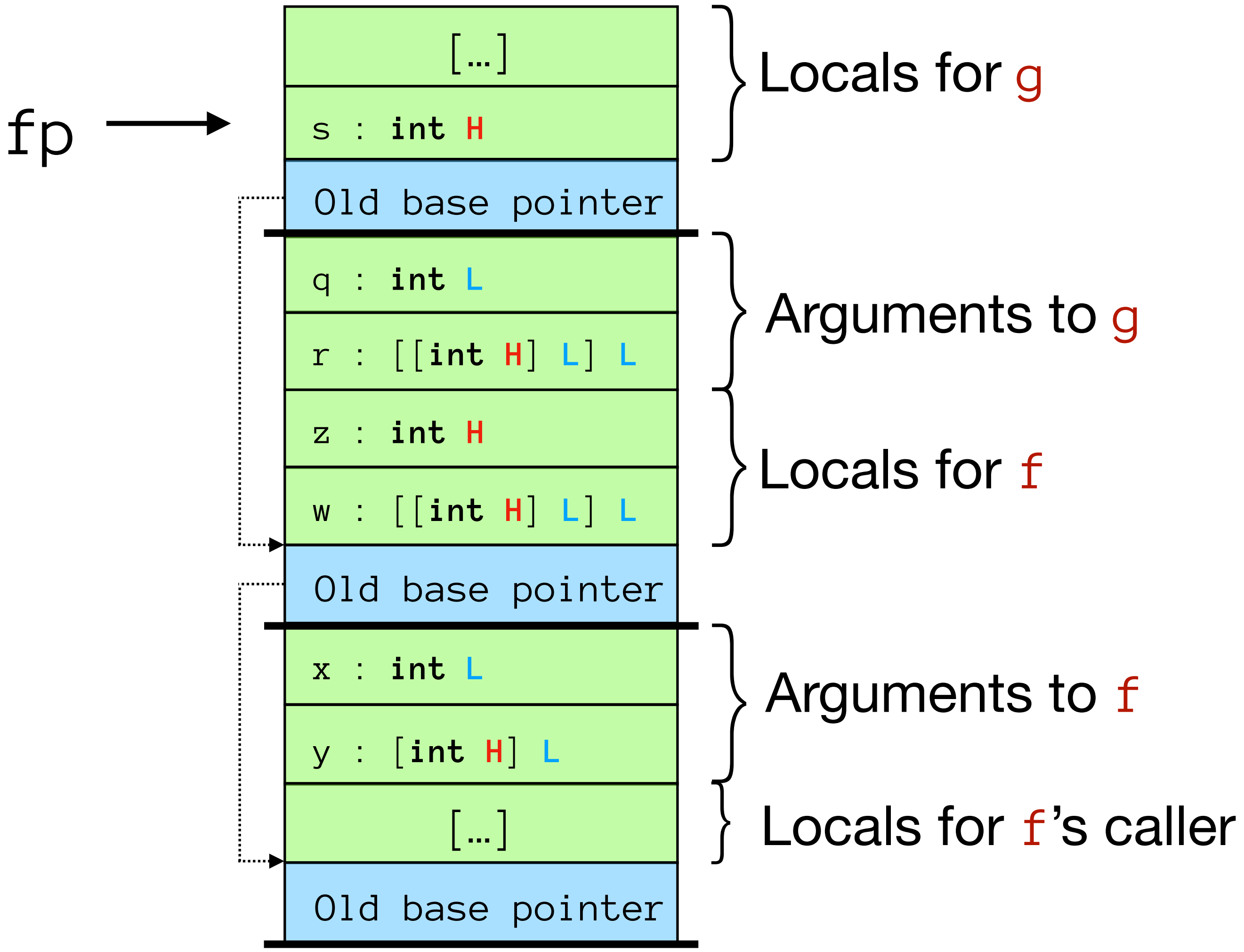
T_{st} = μ a : **type** . ∃ b : **type** . a @ b



unroll fp : ∃ b : **type** . T_{st} @ b

Typing the stack pointer

T_{st} = μ a : **type** . ∃ b : **type** . a @ b



(b, p) := **unpack** (**unroll** fp)

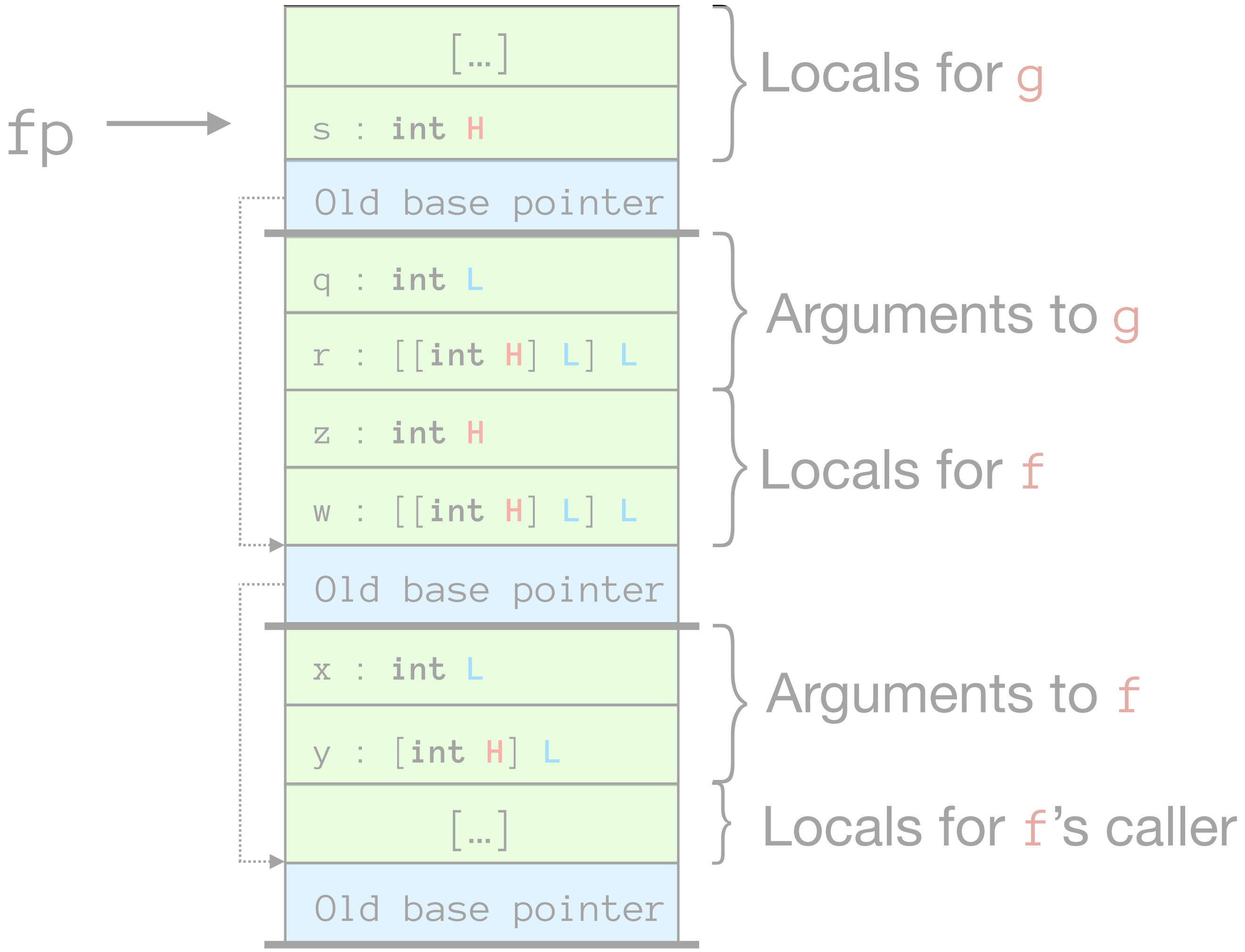
b : type

p : T_{st} @ b

Typing the stack pointer

T_{st} = μ a : **type** . ∃ b : **type** . a @ b

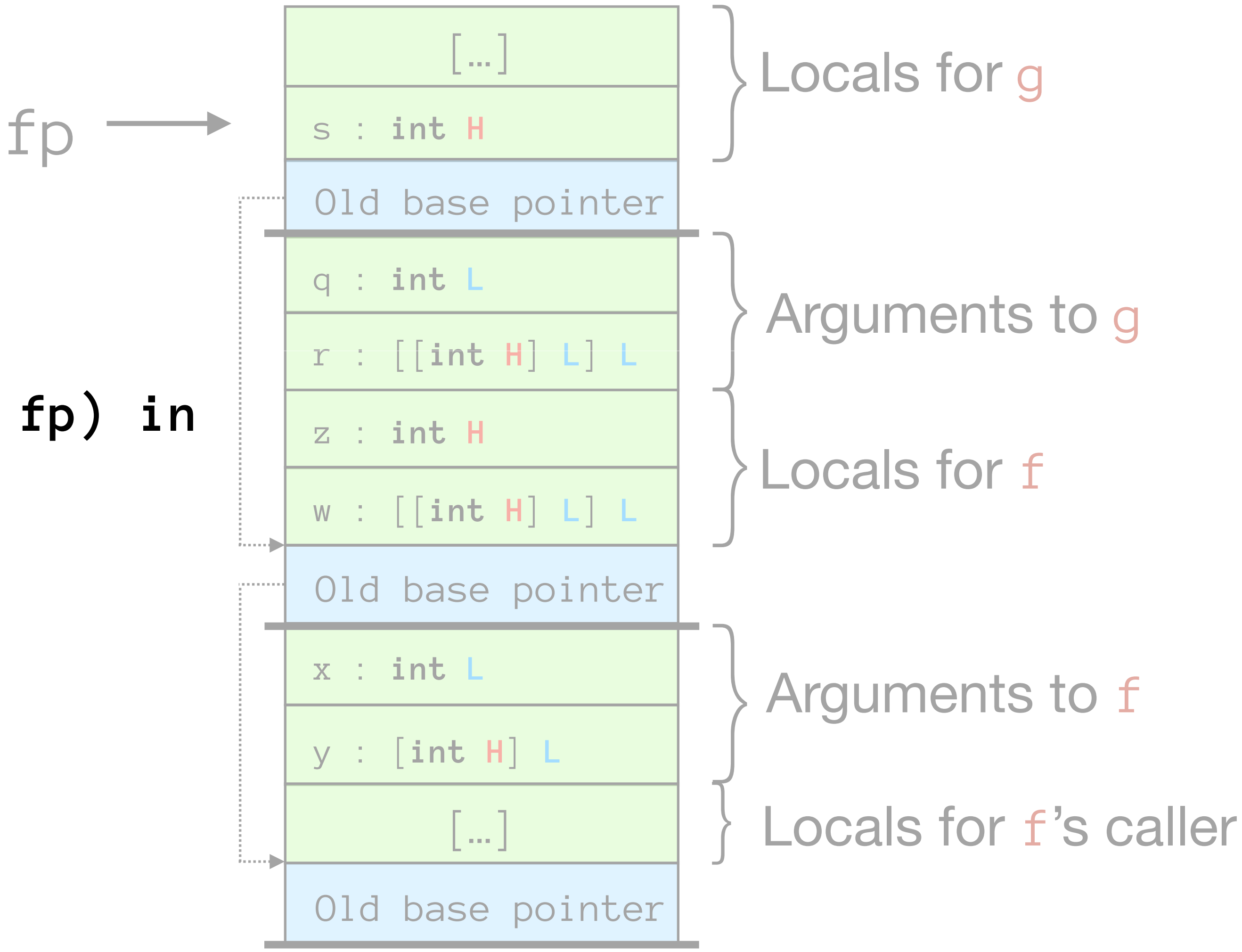
```
def g(q : int L, r : [[int H] L] L) =
  s : int H := x in
  w : [int H] L := &y in
  ...
```



Typing the stack pointer

```
Tst = μ a : type . ∃ b : type . a @ b
```

```
def g(q : int L, r : [[int H] L] L) =
  s : int H := x in
  w : [int H] L := &y in
  (b : type, p : Tst @ b):= unpack (unroll fp) in
  ...
```



fp

[…]

s : int H

Old base pointer

q : int L

r : [[int H] L] L

z : int H

w : [[int H] L] L

Old base pointer

x : int L

y : [int H] L

[…]

Old base pointer

Locals for g

Arguments to g

Locals for f

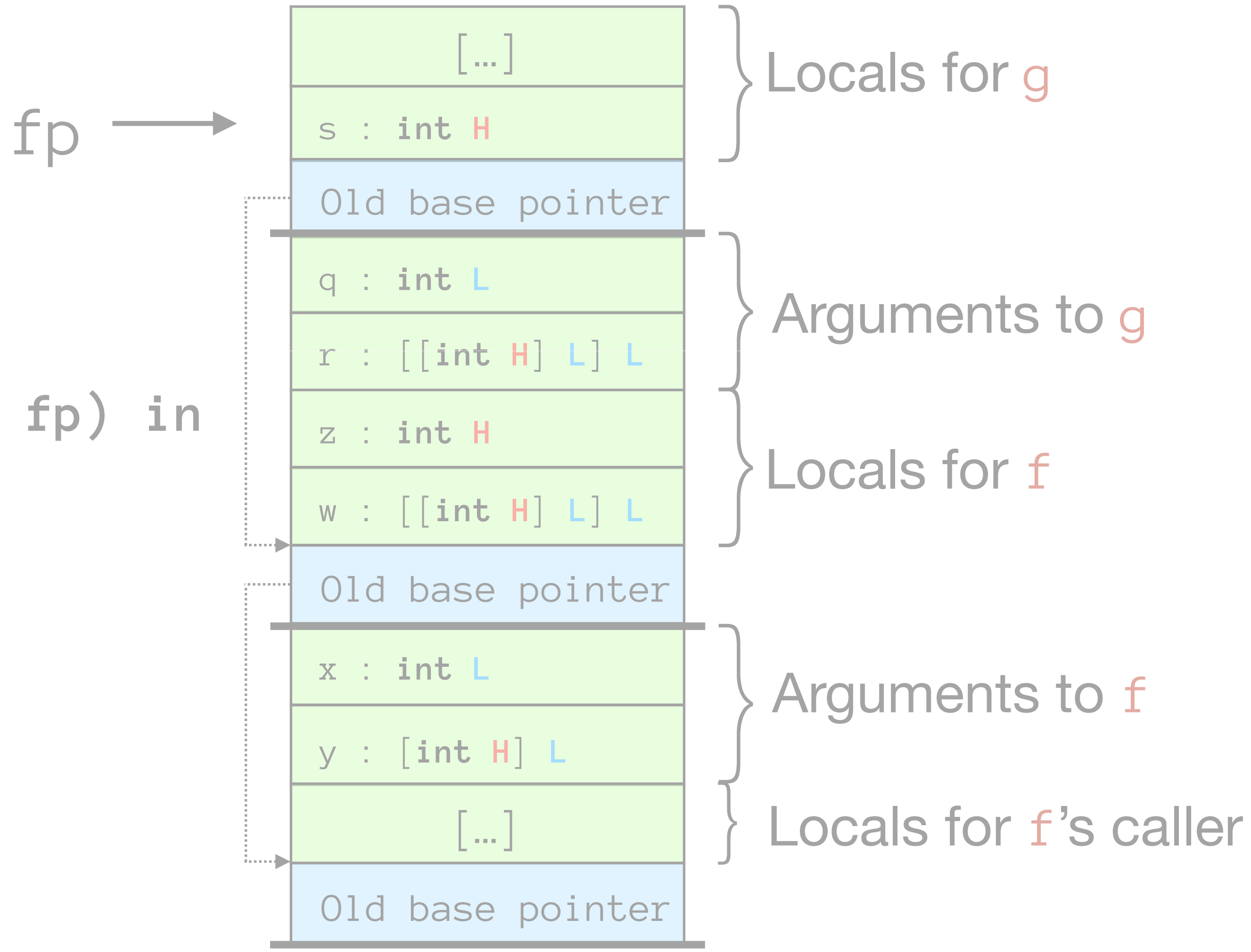
Arguments to f

Locals for f's caller

Typing the stack pointer

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$

```
def g(q : int L, r : [[int H] L] L) =
  s : int H := x in
  w : [int H] L := &y in
  (b : type, p : Tst @ b):= unpack (unroll fp) in
  ...
```



fp

[…]

s : int H

Old base pointer

q : int L

r : [[int H] L] L

z : int H

w : [[int H] L] L

Old base pointer

x : int L

y : [int H] L

[…]

Old base pointer

Locals for g

Arguments to g

Locals for f

Arguments to f

Locals for f’s caller

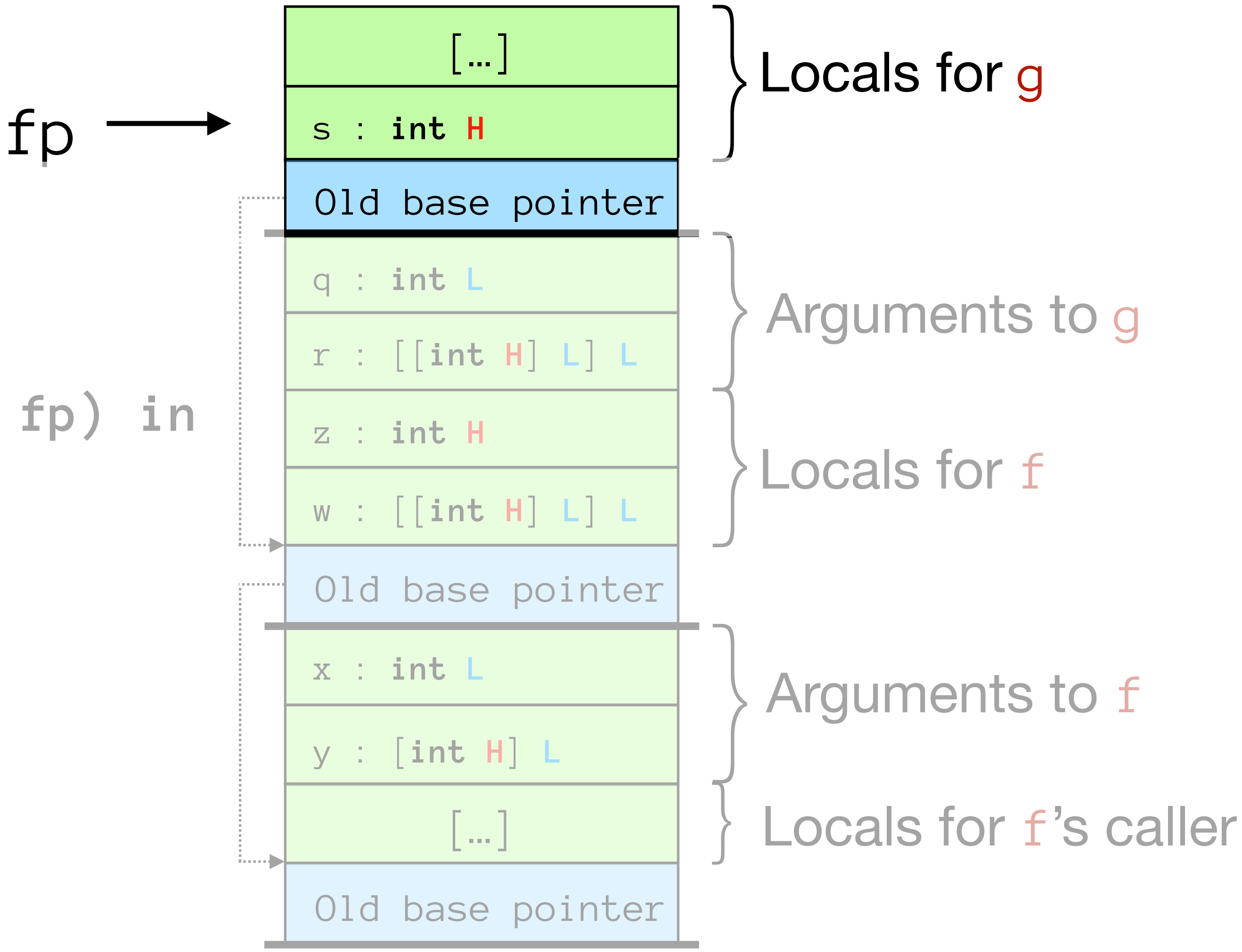
```
b = int H * [int H] L
p = 0x967a0c9d
```

Typing the stack pointer

```
Tst = μ a : type . ∃ b : type . a @ b
```

```
def g(q : int L, r : [[int H] L] L) =
  s : int H := x in
  w : [int H] L := &y in
  (b : type, p : Tst @ b):= unpack (unroll fp) in
  ...
```

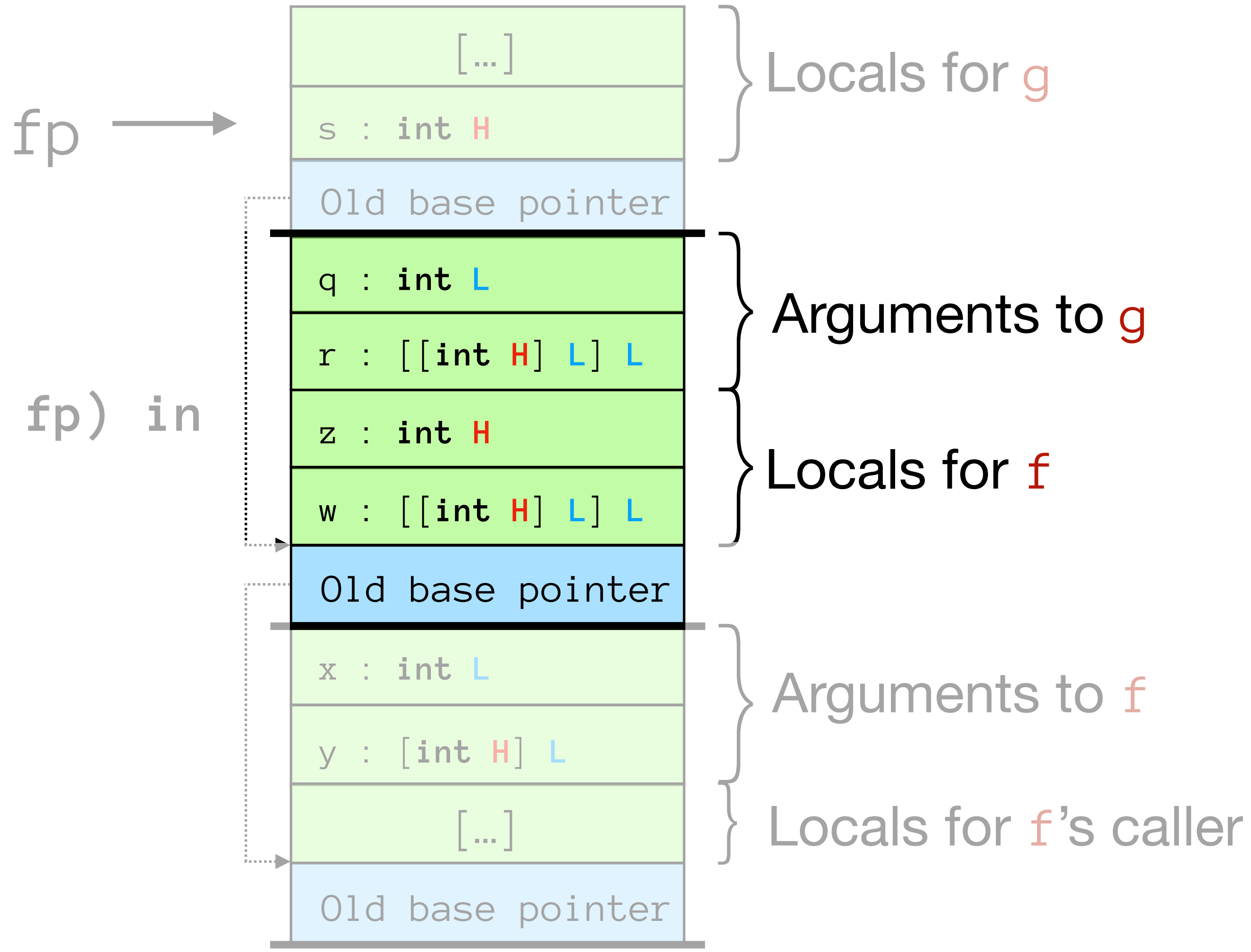
```
b = int H * [int H] L

p = 0x967a0c9d

    p : Tst @ b
    p - sizeof(Tst) : @ Tst * b
```



Typing the stack pointer

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$

```
def g(q : int L, r : [[int H] L] L) =
  s : int H := x in
  w : [int H] L := &y in
  (b : type, p : Tst @ b):= unpack (unroll fp) in
  ...
```



$b = \mathbf{int}\ H * [\mathbf{int}\ H]\ L$

$p = \mathtt{0x967a0c9d}$

$p : T_{st}\ @\ b$

$p - \mathbf{sizeof}(T_{st}) : @\ T_{st} * b$

$*(p - \mathbf{sizeof}(T_{st})) : T_{st}$

$\mathbf{unroll}\ *(p - \mathbf{sizeof}(T_{st})) : \exists\ b : \mathbf{type}\ .\ T_{st}\ @\ b$

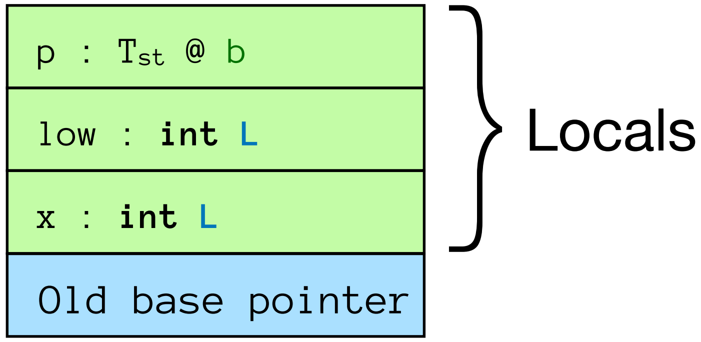
Labels on types

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$

Runtime representation of types → Types can leak information!

Labels on types

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$



p : T_{st} @ b

low : **int** L

x : **int** L

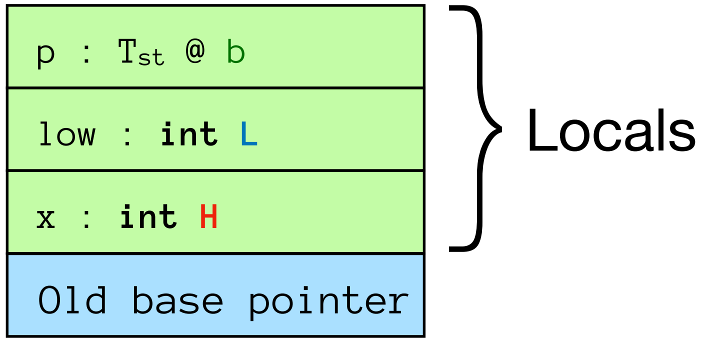
Old base pointer

Locals

```
inspectCurrentFrame1() =
  x : int L := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Labels on types

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$



p : T_{st} @ b

low : **int** L

x : **int** H

Old base pointer

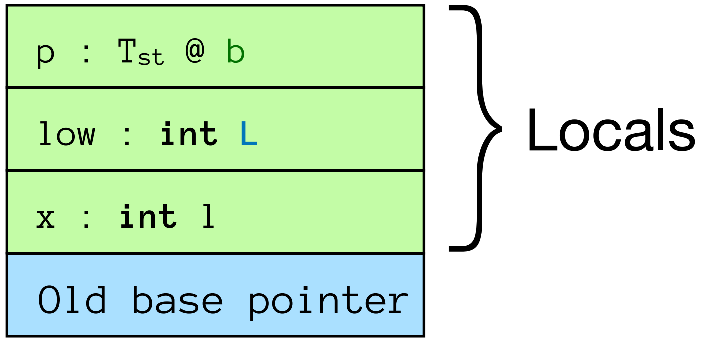
Locals

```
inspectCurrentFrame2() =
  x : int H := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

```
inspectCurrentFrame1() =
  x : int L := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Labels on types

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ .\ a\ @\ b$$



p : Tst @ b

low : int L

x : int l

Old base pointer

Locals

```
inspectCurrentFrame3(l : level H) =
  x : int l := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Leak from H to L

```
inspectCurrentFrame1() =
  x : int L := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

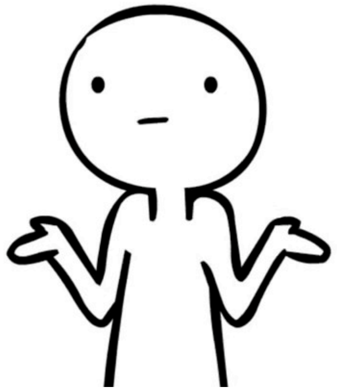
```
inspectCurrentFrame2() =
  x : int H := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Labels on types

$$T_{st} = \mu\ a : \mathbf{type}\ ?\ .\ \exists\ b : \mathbf{type}\ ?\ .\ a\ @\ b$$

Let’s focus on this one

pc?



```
inspectCurrentFrame3(l : level H) =
  x : int l := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Labels on types

T_{st} = μ a : **type** . ∃ b : **type** pc . a @ b

```
inspectCurrentFrame3(l : level H) =
  x : int l := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Labels on types

$$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ pc\ .\ a\ @\ b$$

pc = “Upper bound on information that affects control flow”

pc

```
inspectCurrentFrame3(l : level H) =
  x : int l := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

```
b : type pc
  = type L
```

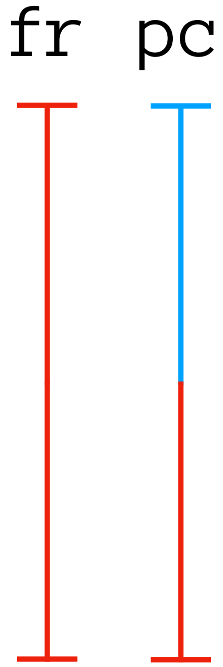
Labels on types

$T_{st} = \mu\ a : \mathbf{type}\ .\ \exists\ b : \mathbf{type}\ fr\ .\ a\ @\ b$

pc = “Upper bound on information that affects control flow”

fr = “Upper bound on information that affects frame layout”

```
b : type fr
  = type H
```



```
inspectCurrentFrame3(l : level H) =
  x : int l := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Labels on types

Upper bound on fr

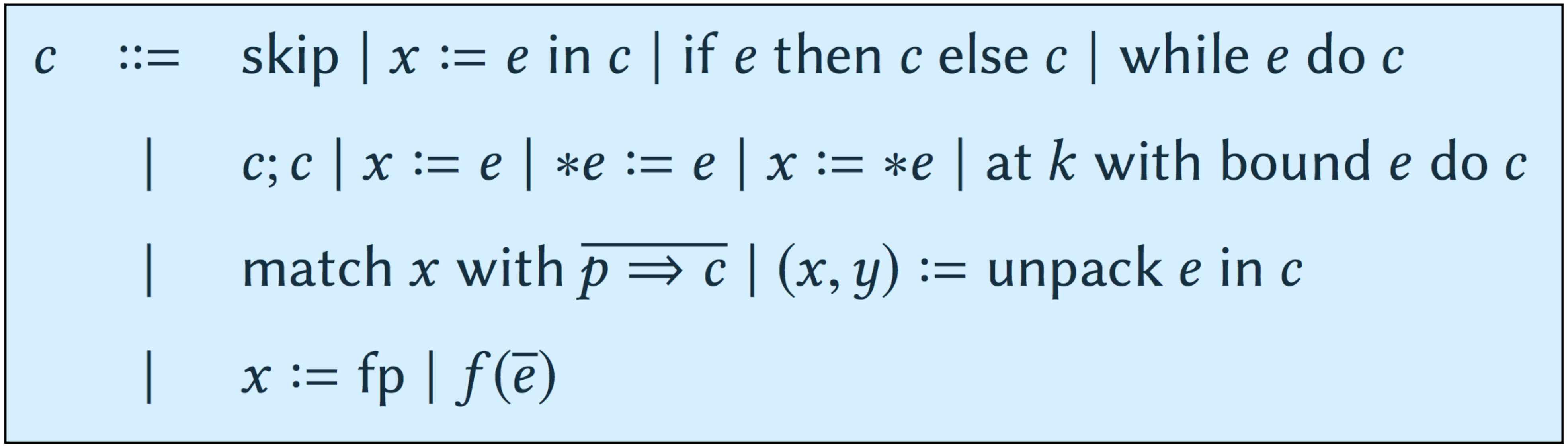
```
                                   H
inspectCurrentFrame3(l : level H)   =
                                   L
  x : int l := 42 in
  low : int L = 0 in
  (b, p) := unpack (unroll fp) in
  match b with
    int L * _ → low := 1
  | _ → skip
```

Lower bound on pc

Okay since `l : level H` and H ⊑ fr

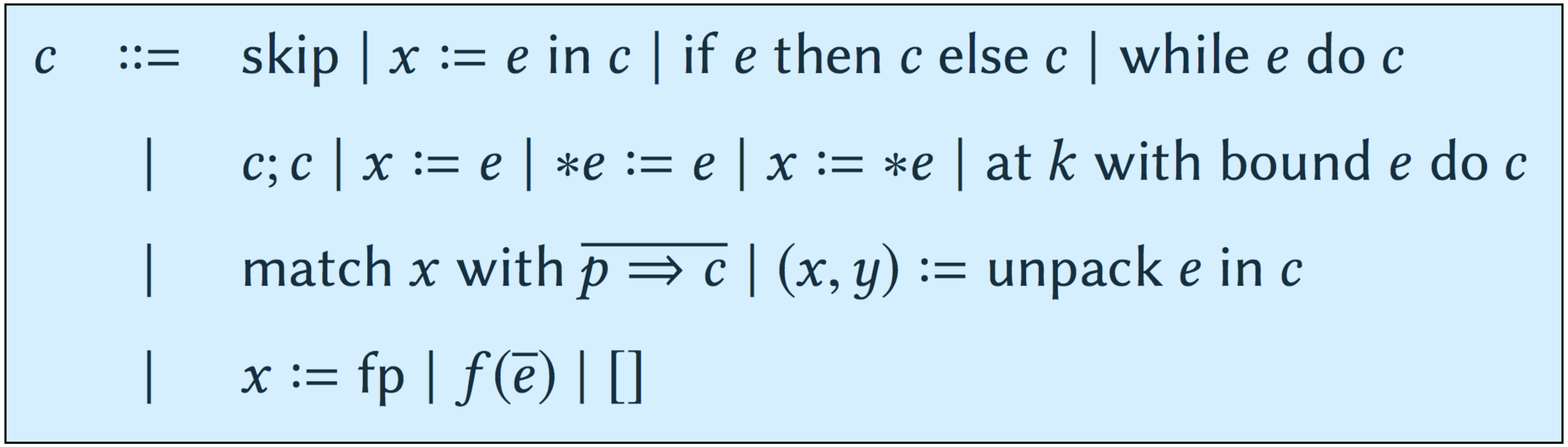
pc = H. Cannot assign to L

The Zee language



$$
\begin{array}{lcl}
c & ::= & \mathsf{skip} \mid x := e \;\mathsf{in}\; c \mid \mathsf{if}\; e \;\mathsf{then}\; c \;\mathsf{else}\; c \mid \mathsf{while}\; e \;\mathsf{do}\; c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\; k \;\mathsf{with\ bound}\; e \;\mathsf{do}\; c \\
 & \mid & \mathsf{match}\; x \;\mathsf{with}\; \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\; e \;\mathsf{in}\; c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e})
\end{array}
$$

The Zee language

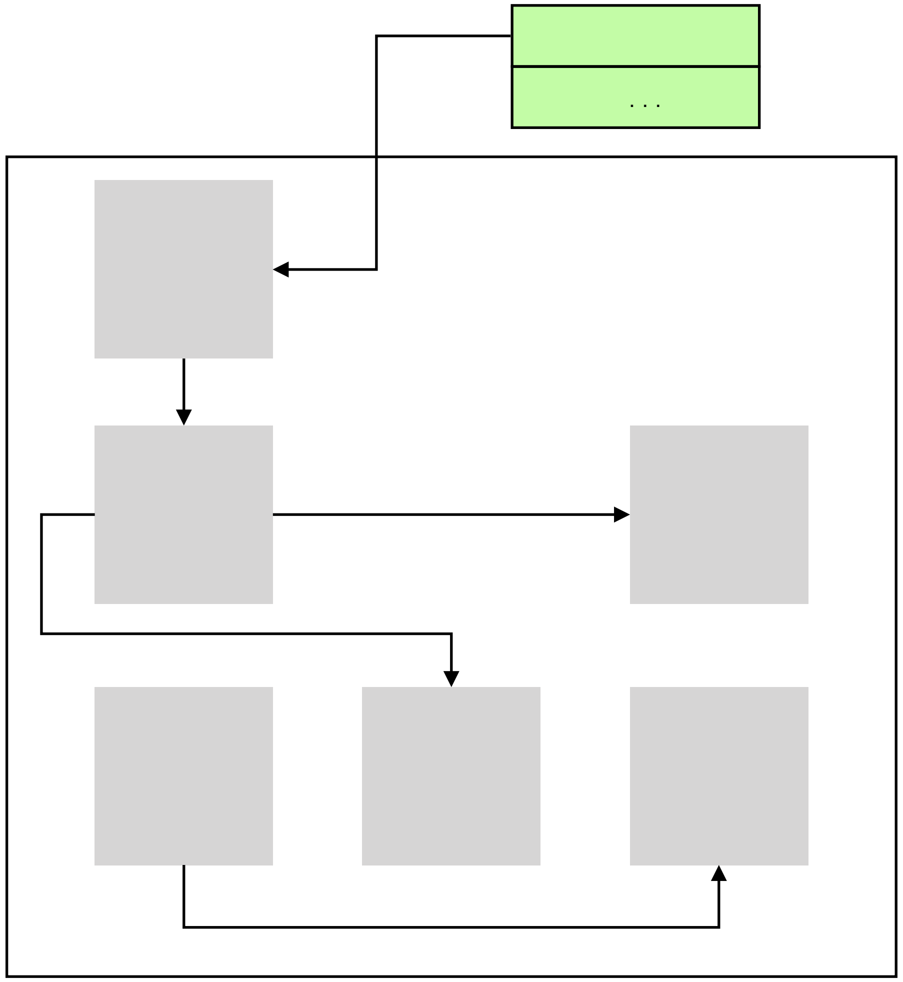


$$
\begin{array}{lcl}
c & ::= & \mathsf{skip} \mid x := e \;\mathsf{in}\; c \mid \mathsf{if}\; e \;\mathsf{then}\; c \;\mathsf{else}\; c \mid \mathsf{while}\; e \;\mathsf{do}\; c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\; k \;\mathsf{with\ bound}\; e \;\mathsf{do}\; c \\
 & \mid & \mathsf{match}\; x \;\mathsf{with}\; \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\; e \;\mathsf{in}\; c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

The Zee language

$$\begin{array}{lll}
c & ::= & \mathsf{skip} \mid x := e \ \mathsf{in}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\ k\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\
 & \mid & \mathsf{match}\ x\ \mathsf{with}\ \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\ e\ \mathsf{in}\ c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid []
\end{array}$$

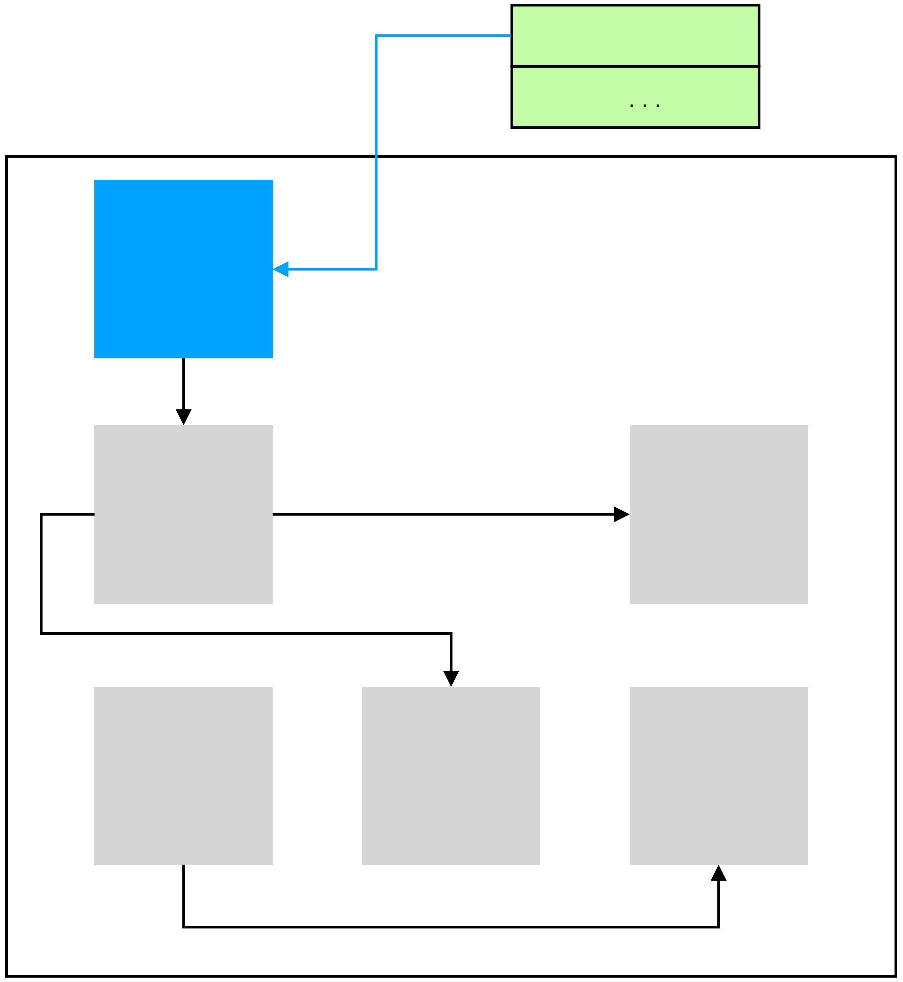
Consider two different garbage collector implementations



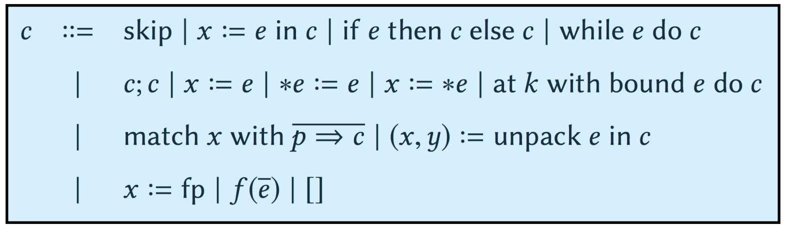
The Zee language

$$
\begin{array}{lcl}
c & ::= & \mathsf{skip} \mid x := e \ \mathsf{in}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\ k\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\
 & \mid & \mathsf{match}\ x\ \mathsf{with}\ \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\ e\ \mathsf{in}\ c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

Consider two different garbage collector implementations

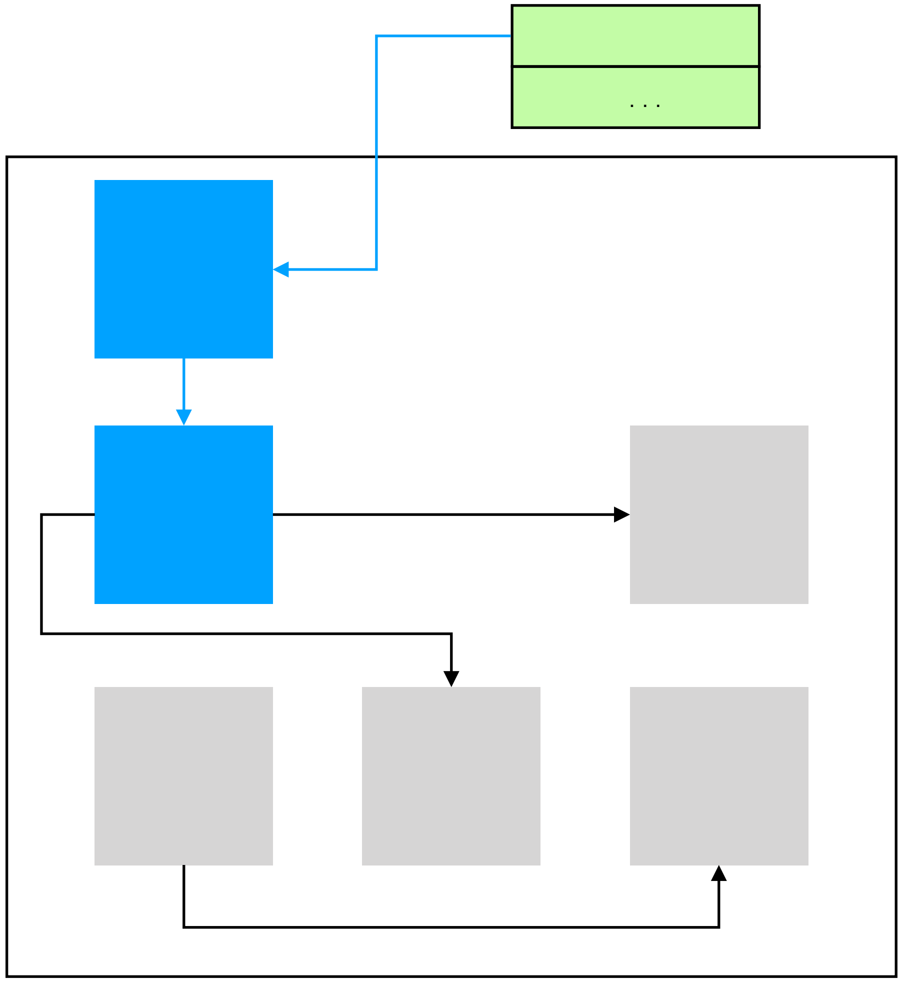


The Zee language



$$\begin{array}{lll} c & ::= & \mathsf{skip} \mid x := e \ \mathsf{in}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\ & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\ k\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\ & \mid & \mathsf{match}\ x\ \mathsf{with}\ \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\ e\ \mathsf{in}\ c \\ & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid [] \end{array}$$

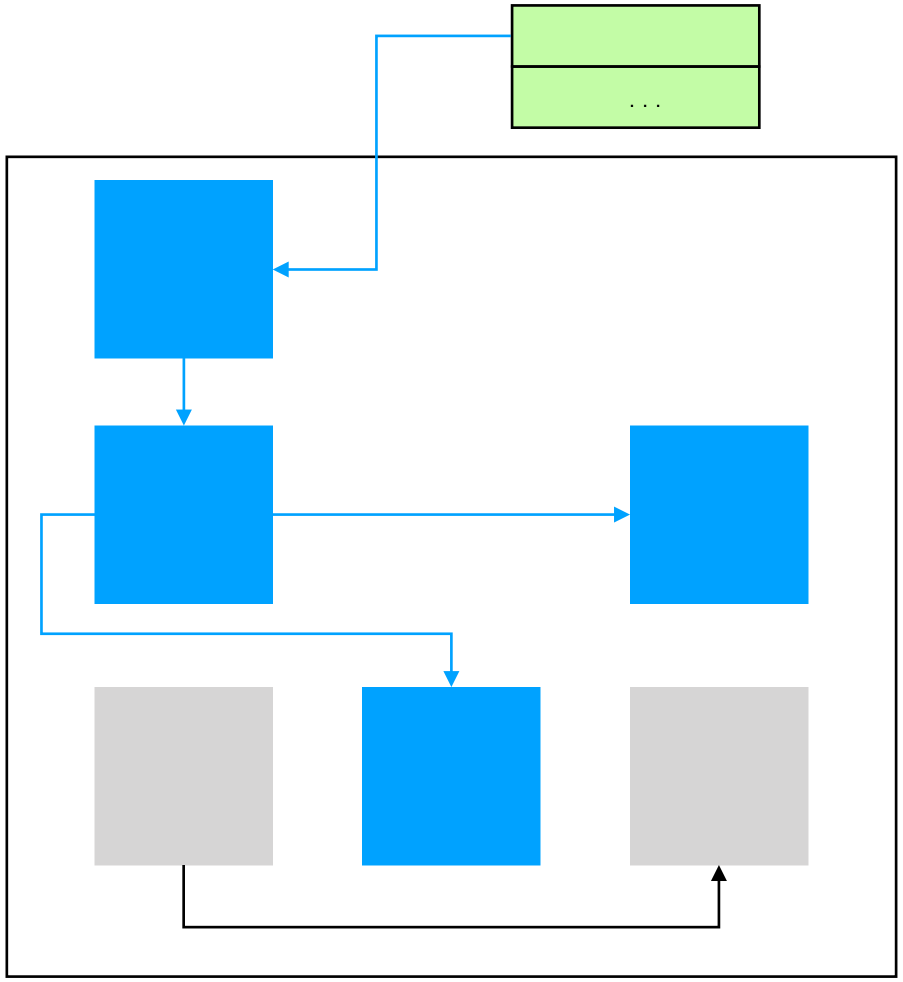
Consider two different garbage collector implementations



The Zee language

$$\begin{array}{rcl} c & ::= & \text{skip} \mid x := e \text{ in } c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \\ & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \text{at } k \text{ with bound } e \text{ do } c \\ & \mid & \text{match } x \text{ with } \overline{p \Rightarrow c} \mid (x, y) := \text{unpack } e \text{ in } c \\ & \mid & x := \text{fp} \mid f(\overline{e}) \mid [] \end{array}$$

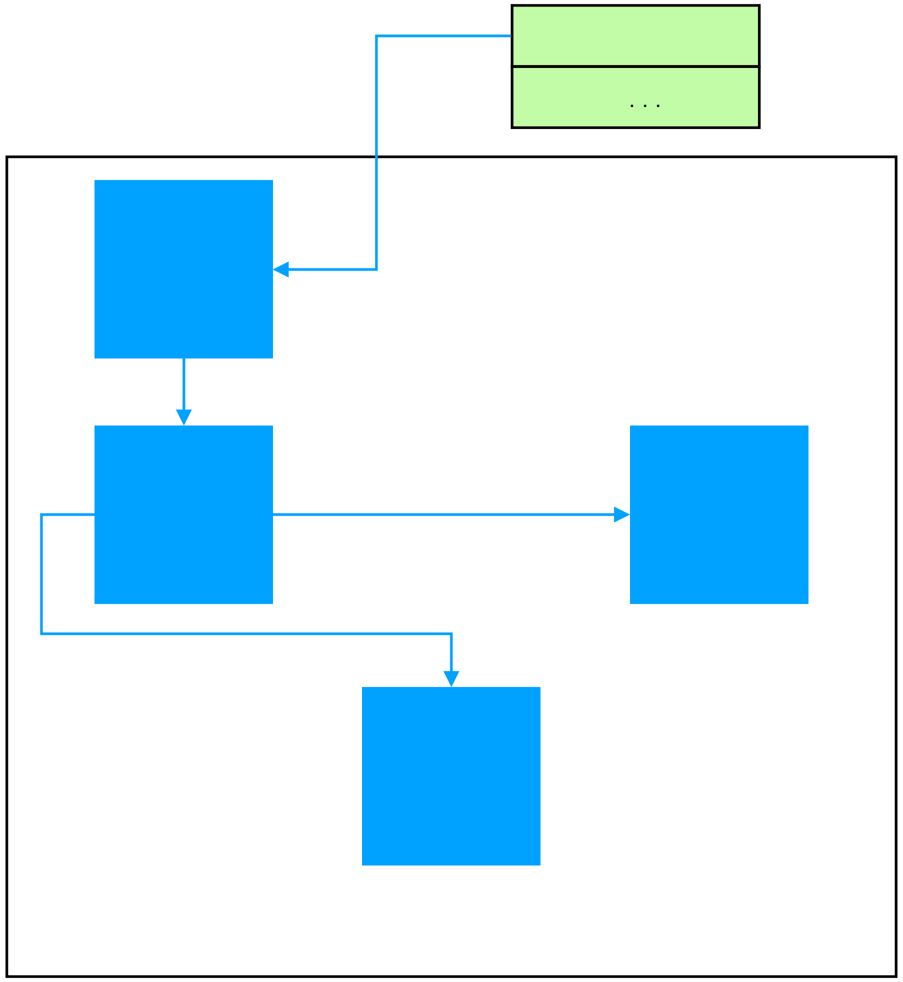
Consider two different garbage collector implementations

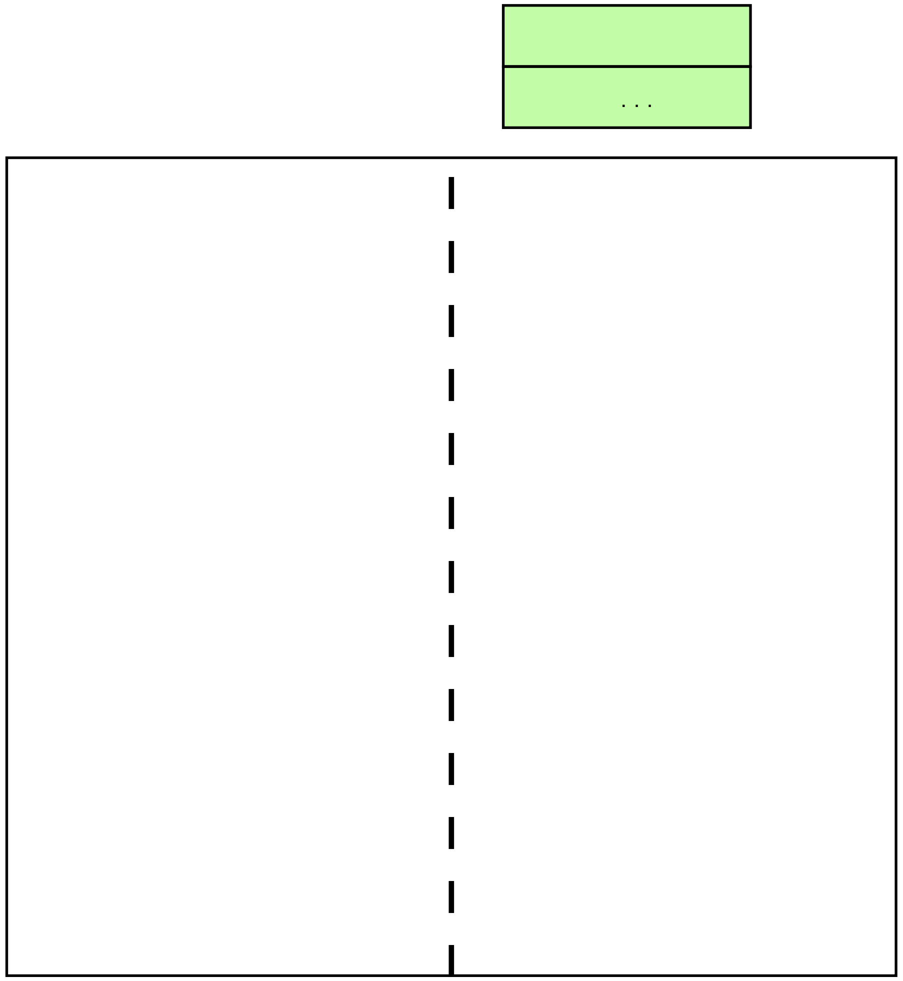


The Zee language

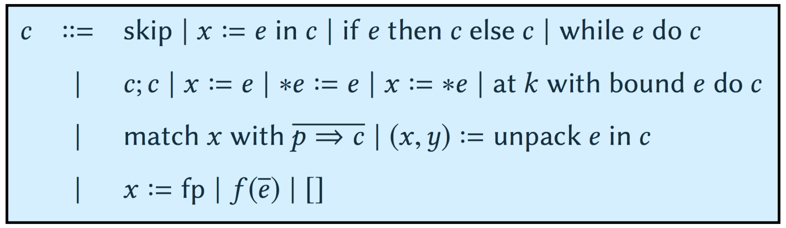
$$
\begin{array}{lll}
c & ::= & \mathsf{skip} \mid x := e \;\mathsf{in}\; c \mid \mathsf{if}\; e \;\mathsf{then}\; c \;\mathsf{else}\; c \mid \mathsf{while}\; e \;\mathsf{do}\; c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\; k \;\mathsf{with\ bound}\; e \;\mathsf{do}\; c \\
 & \mid & \mathsf{match}\; x \;\mathsf{with}\; \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\; e \;\mathsf{in}\; c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

Consider two different garbage collector implementations



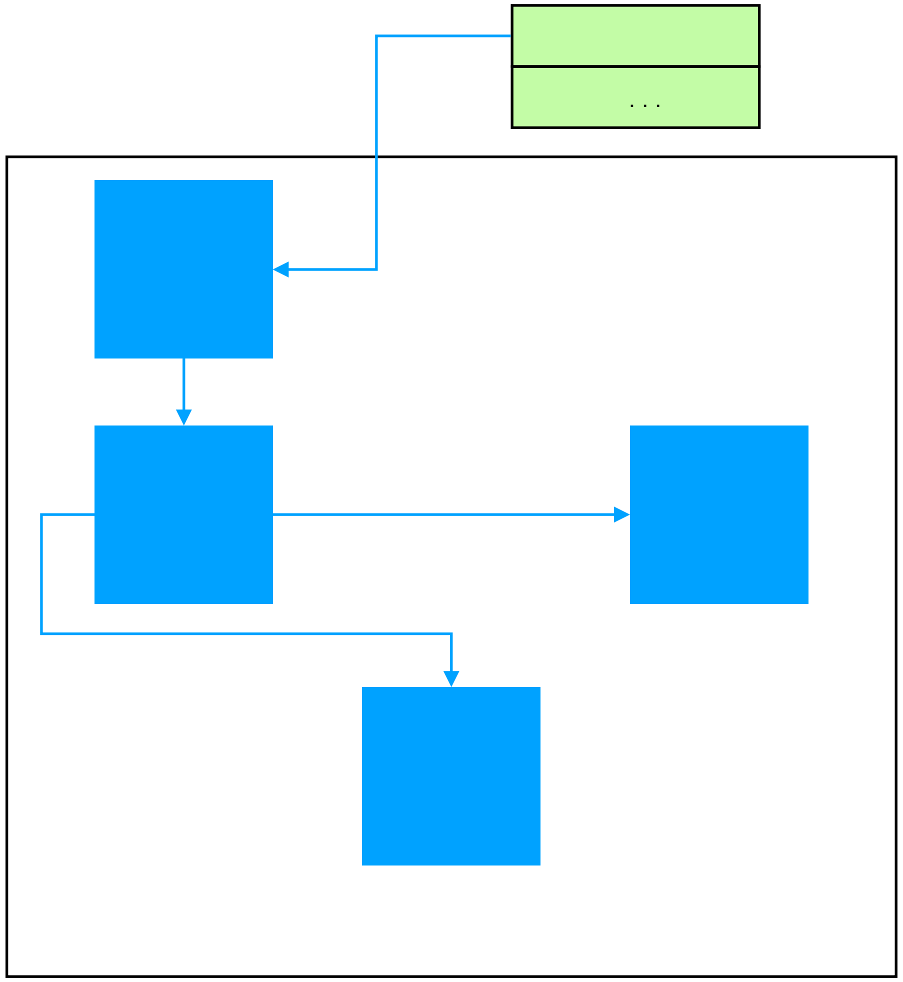


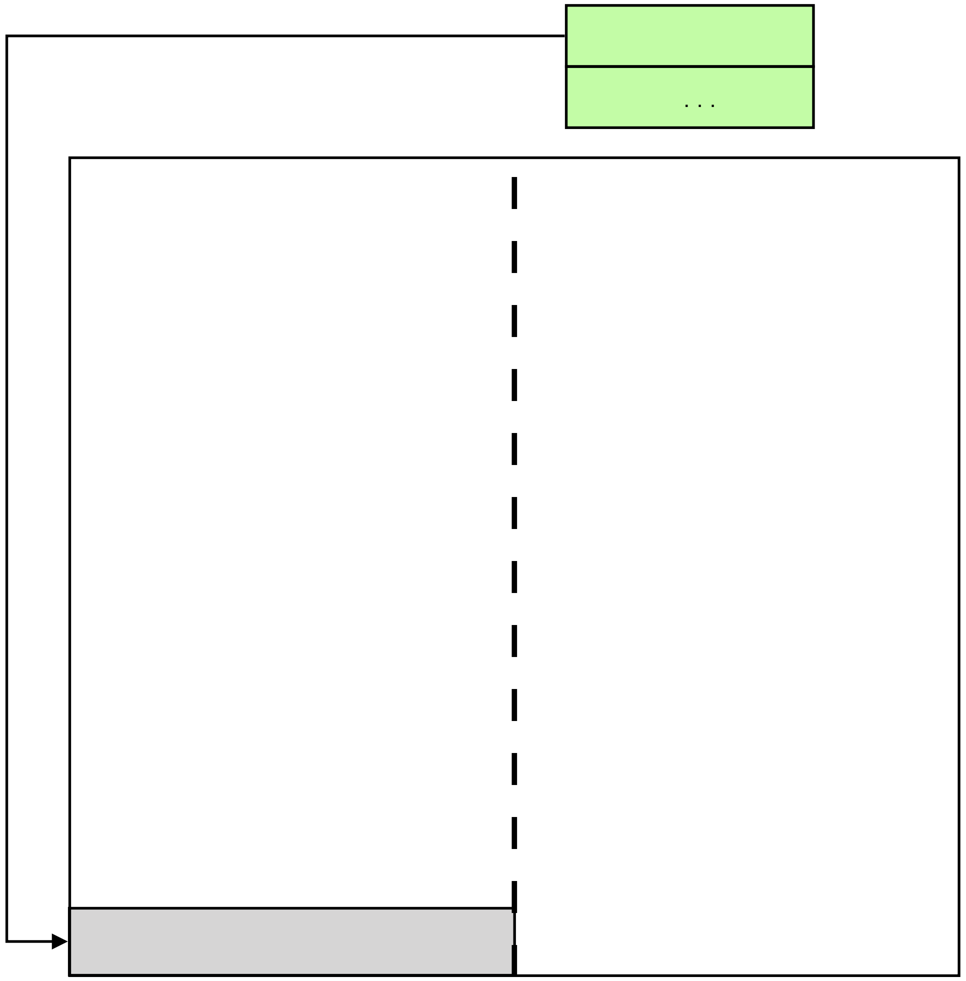
The Zee language



$$\begin{array}{rcl} c & ::= & \mathsf{skip} \mid x := e \;\mathsf{in}\; c \mid \mathsf{if}\; e \;\mathsf{then}\; c \;\mathsf{else}\; c \mid \mathsf{while}\; e \;\mathsf{do}\; c \\ & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\; k \;\mathsf{with\ bound}\; e \;\mathsf{do}\; c \\ & \mid & \mathsf{match}\; x \;\mathsf{with}\; \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\; e \;\mathsf{in}\; c \\ & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid [] \end{array}$$

Consider two different garbage collector implementations

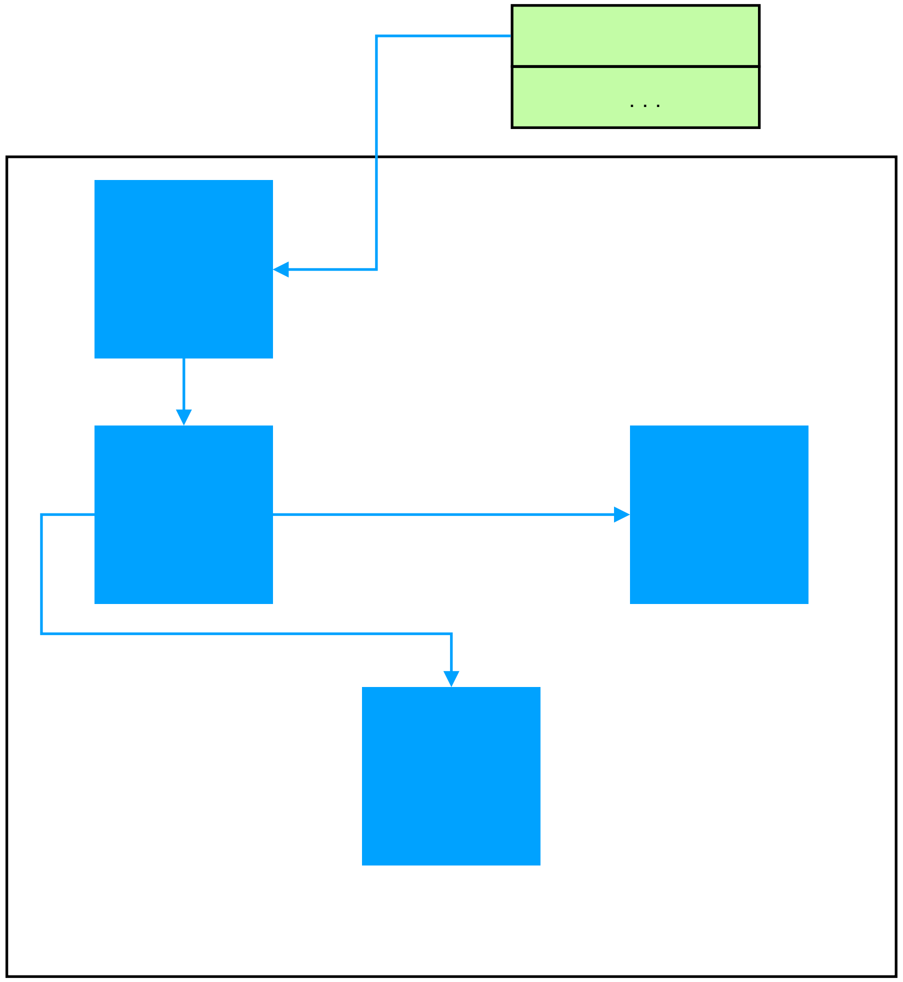


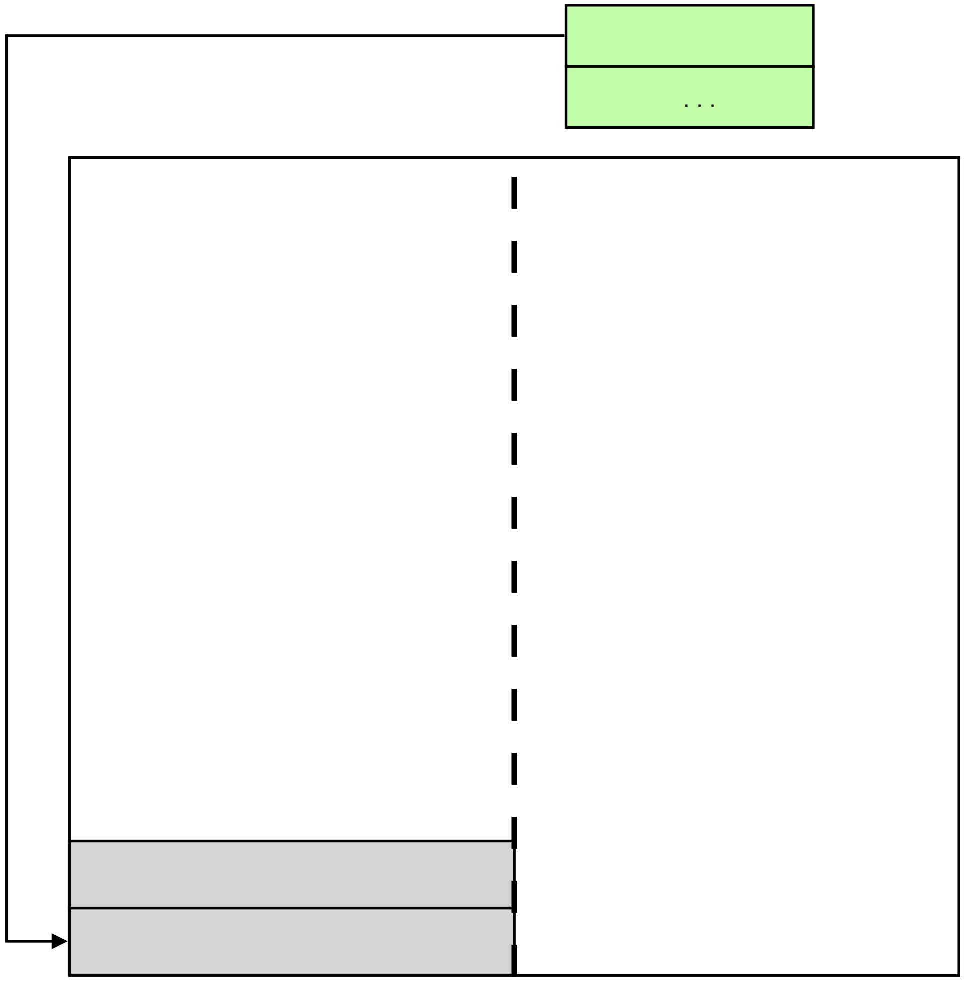


The Zee language

$$
\begin{array}{lll}
c & ::= & \text{skip} \mid x := e \text{ in } c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \\
 & \mid & c ; c \mid x := e \mid *e := e \mid x := *e \mid \text{at } k \text{ with bound } e \text{ do } c \\
 & \mid & \text{match } x \text{ with } \overline{p \Rightarrow c} \mid (x, y) := \text{unpack } e \text{ in } c \\
 & \mid & x := \text{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

Consider two different garbage collector implementations

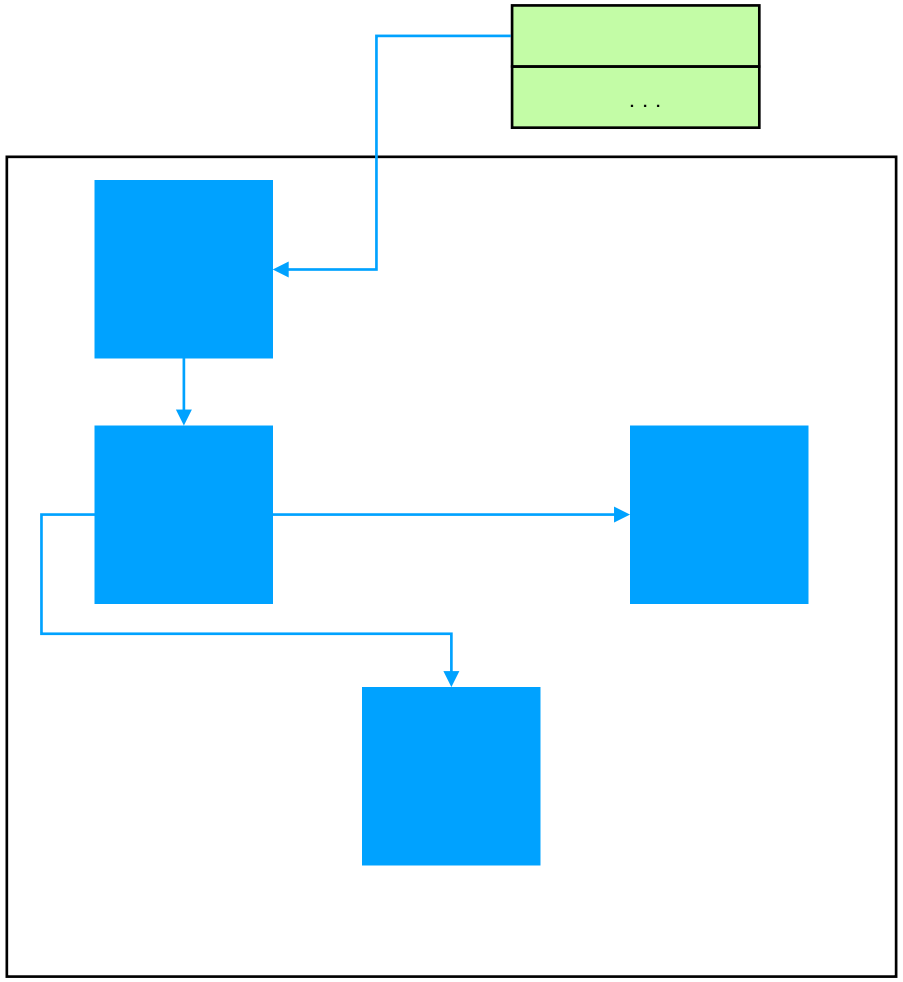


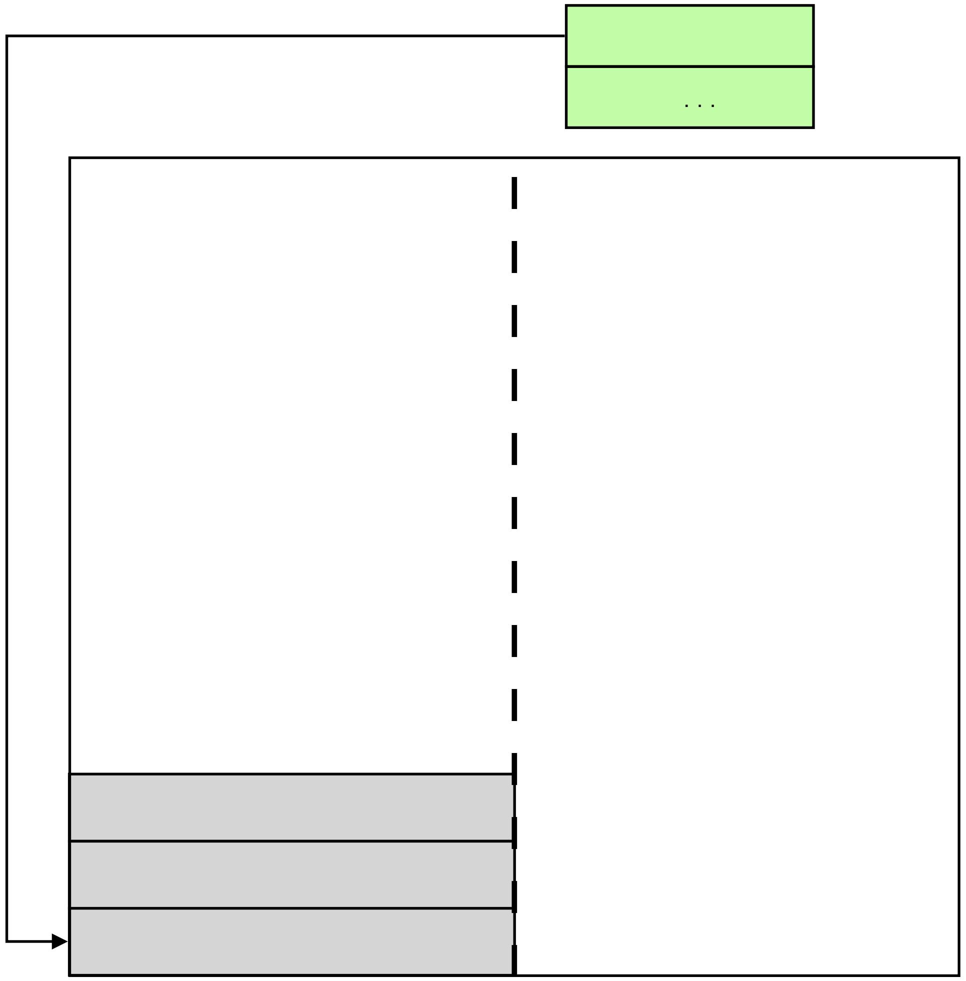


The Zee language

$$
\begin{array}{lcl}
c & ::= & \mathsf{skip} \mid x := e \ \mathsf{in}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\ k\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\
 & \mid & \mathsf{match}\ x\ \mathsf{with}\ \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\ e\ \mathsf{in}\ c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

Consider two different garbage collector implementations

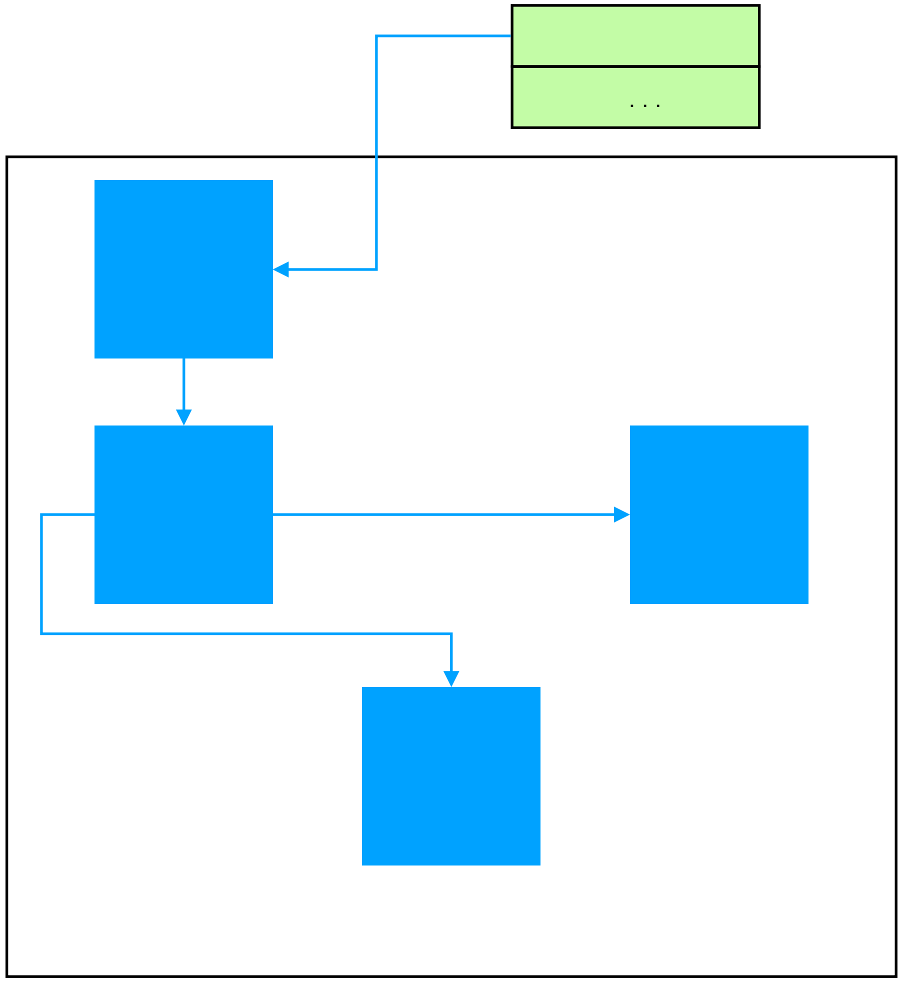


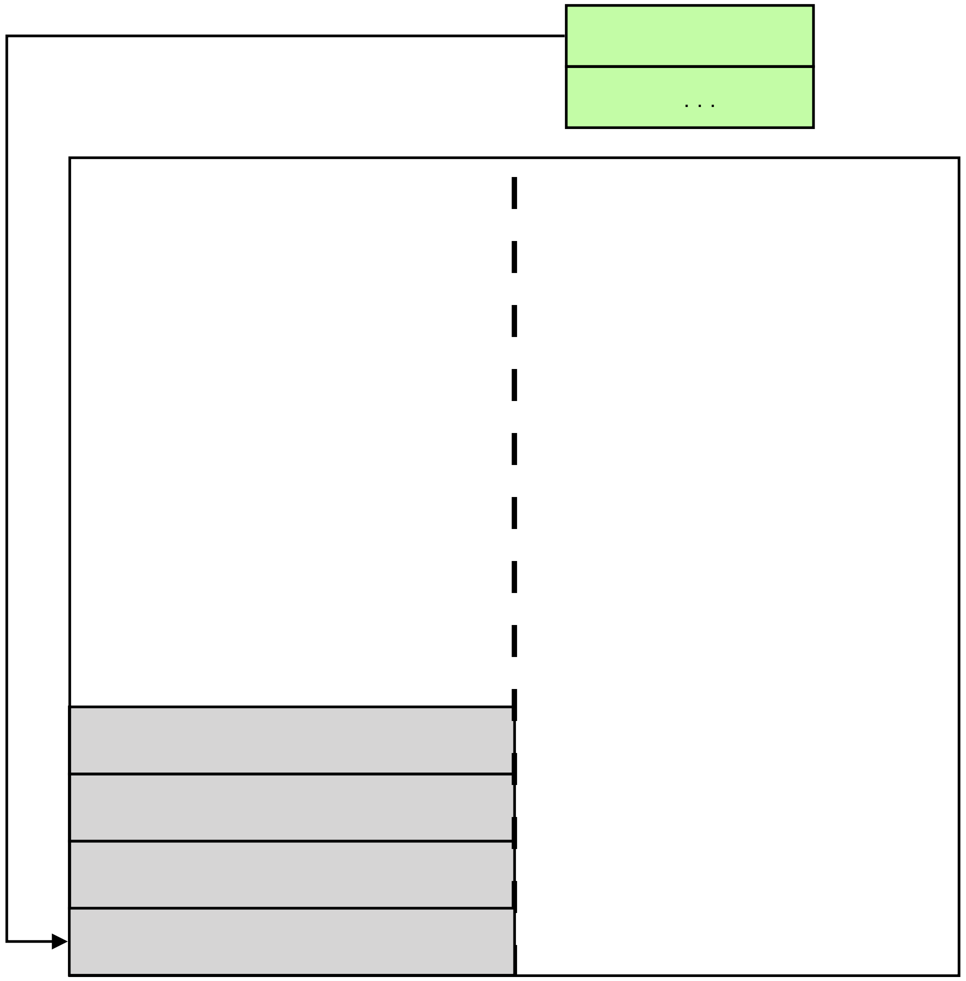


The Zee language

$$
\begin{array}{rcl}
c & ::= & \mathsf{skip} \mid x := e \ \mathsf{in}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\ k\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\
 & \mid & \mathsf{match}\ x\ \mathsf{with}\ \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\ e\ \mathsf{in}\ c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

Consider two different garbage collector implementations

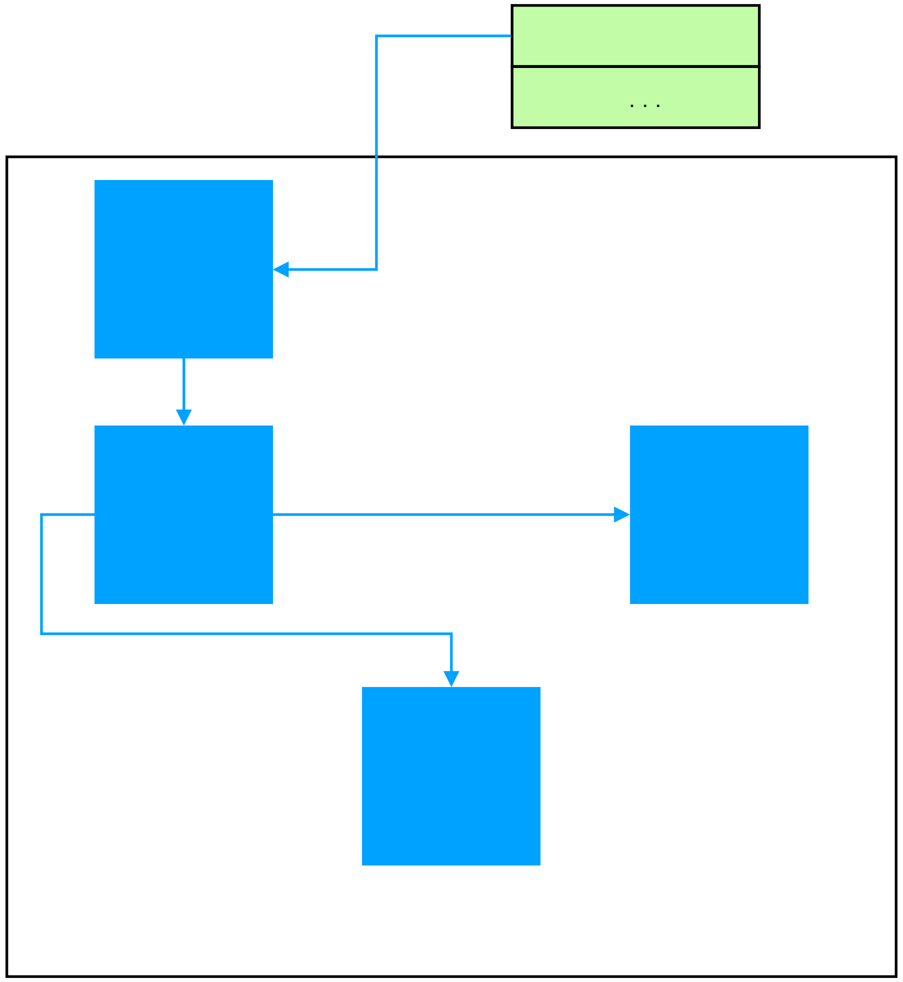


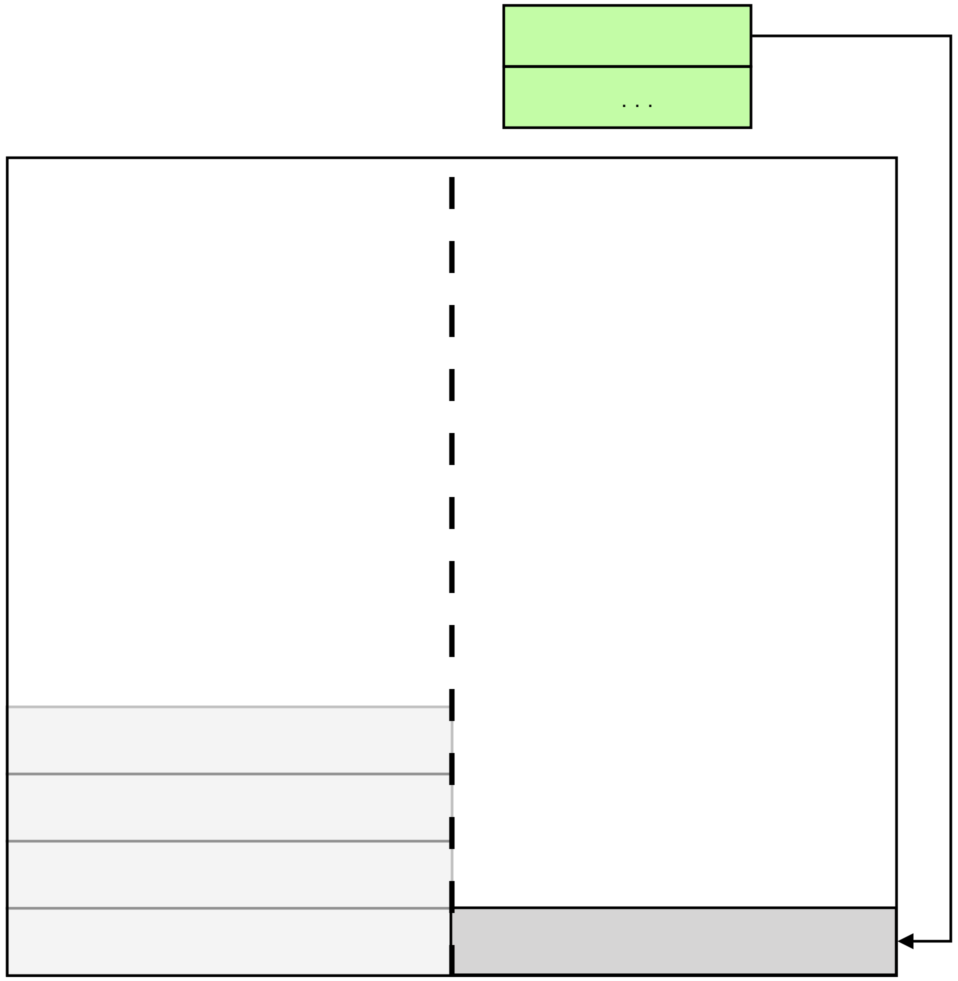


The Zee language

$$
\begin{array}{lll}
c & ::= & \mathsf{skip} \mid x := e \;\mathsf{in}\; c \mid \mathsf{if}\; e \;\mathsf{then}\; c \;\mathsf{else}\; c \mid \mathsf{while}\; e \;\mathsf{do}\; c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\; k \;\mathsf{with\ bound}\; e \;\mathsf{do}\; c \\
 & \mid & \mathsf{match}\; x \;\mathsf{with}\; \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\; e \;\mathsf{in}\; c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid [\,]
\end{array}
$$

Consider two different garbage collector implementations

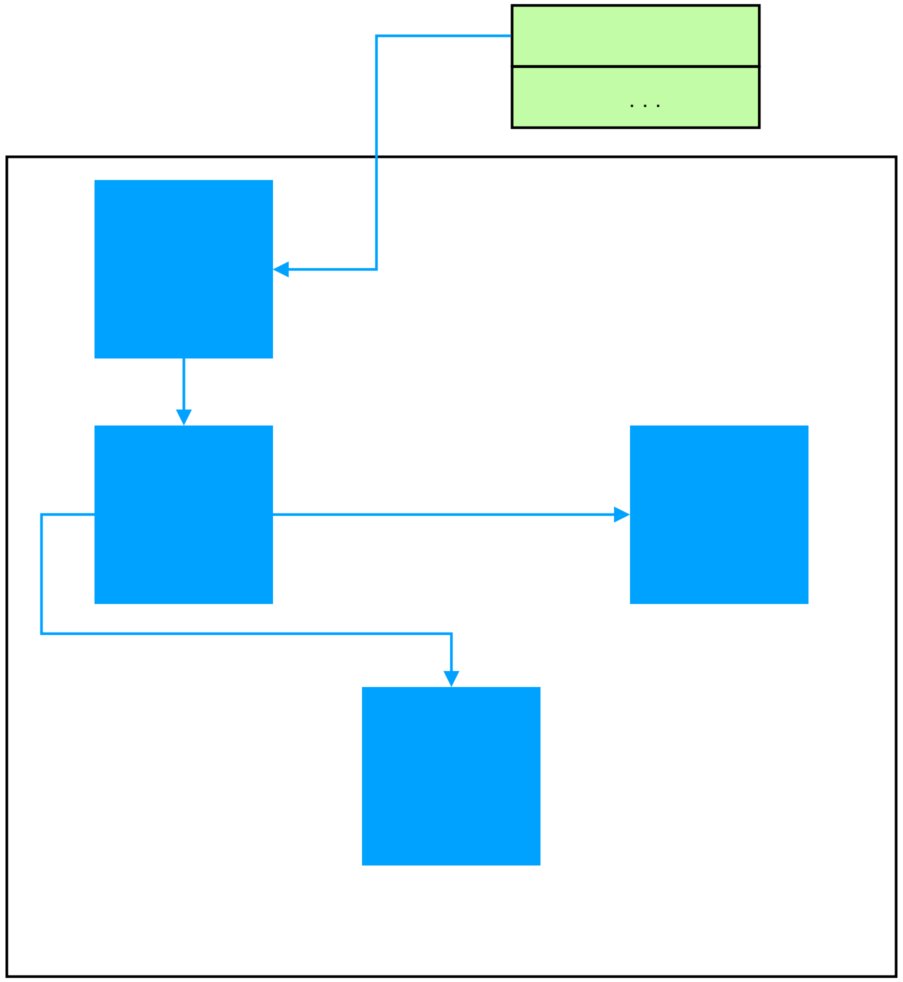


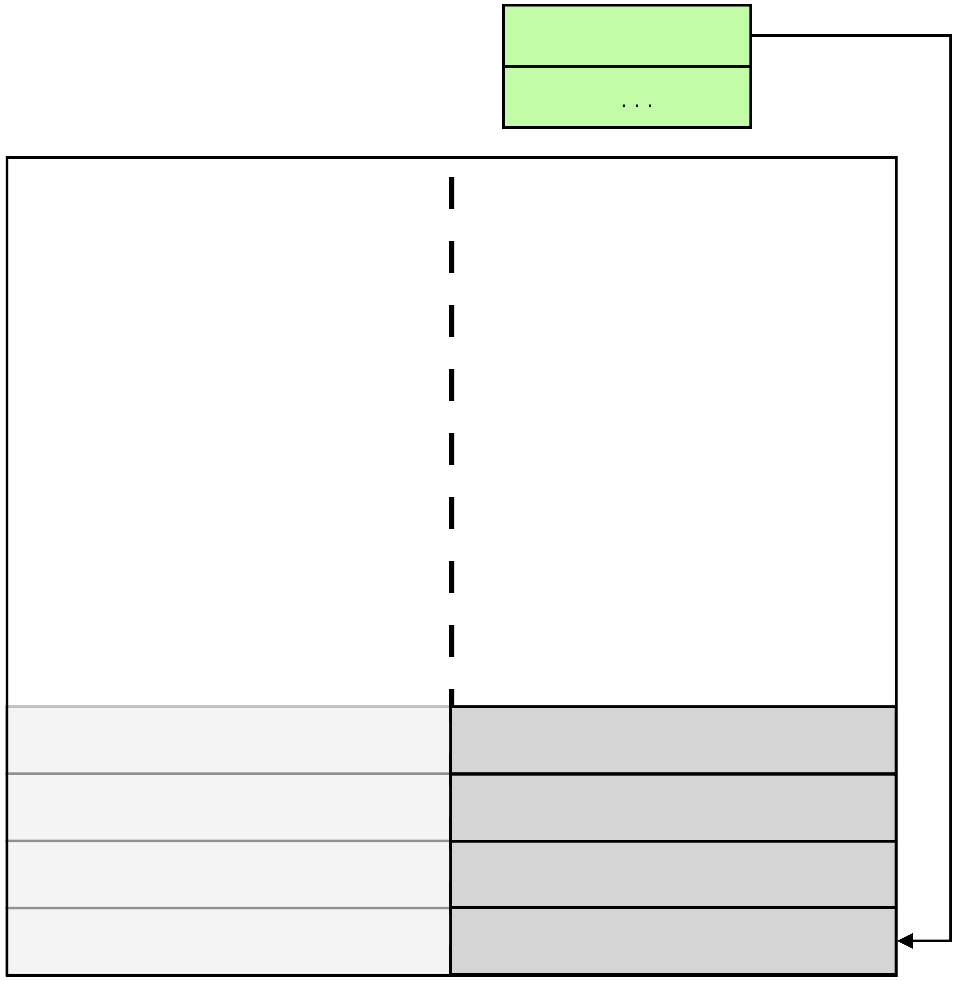


The Zee language

$$\begin{array}{lll}
c & ::= & \mathsf{skip} \mid x := e \ \mathsf{in}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \mathsf{at}\ k\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\
 & \mid & \mathsf{match}\ x\ \mathsf{with}\ \overline{p \Rightarrow c} \mid (x, y) := \mathsf{unpack}\ e\ \mathsf{in}\ c \\
 & \mid & x := \mathsf{fp} \mid f(\overline{e}) \mid [\,]
\end{array}$$

Consider two different garbage collector implementations





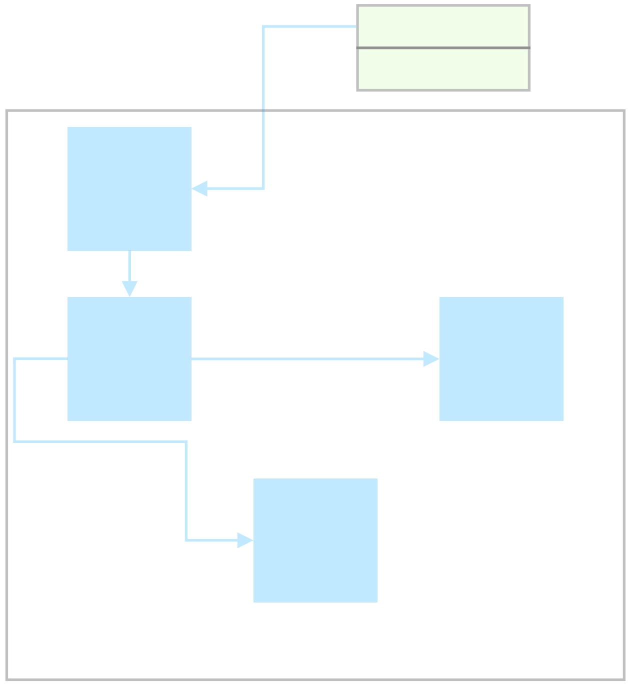
The Zee language

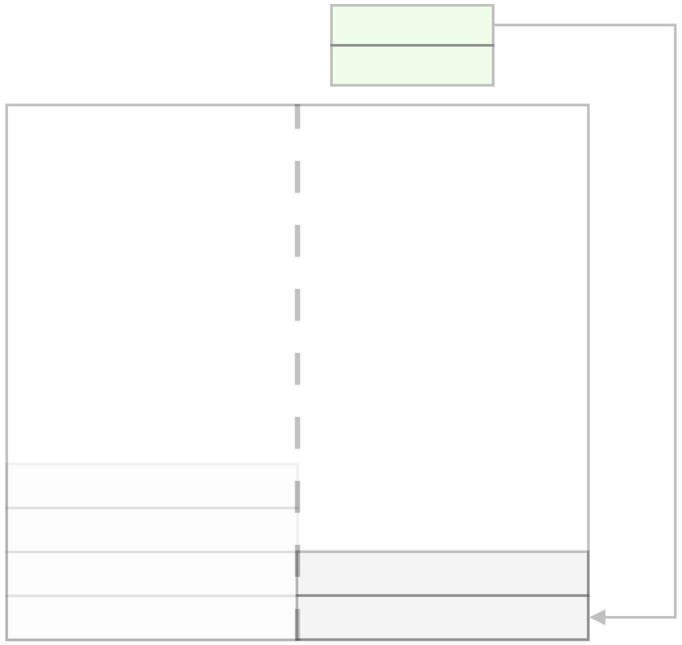
$$
\begin{array}{lcl}
c & ::= & \text{skip} \mid x := e \text{ in } c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \\
 & \mid & c; c \mid x := e \mid *e := e \mid x := *e \mid \text{at } k \text{ with bound } e \text{ do } c \\
 & \mid & \text{match } x \text{ with } \overline{p \Rightarrow c} \mid (x, y) := \text{unpack } e \text{ in } c \\
 & \mid & x := \text{fp} \mid f(\overline{e}) \mid []
\end{array}
$$

Consider two different garbage collector implementations

$$
\begin{array}{lcl}
d & ::= & \text{mark}(e) \mid \text{unmark}(e) \\
 & \mid & x := \text{alloc}(e, e, k) \mid \text{free}(e) \\
 & \mid & x := \text{get}(e) \mid \text{set}(e, e)
\end{array}
$$

$$
\begin{array}{lcl}
d & ::= & \text{set\_semispace}(e) \mid \text{set\_fwd}(e, e) \\
 & \mid & x := \text{get\_fwd}(e) \mid x := \text{alloc}(e, e, k) \\
 & \mid & x := \text{get}(e) \mid \text{set}(e, e)
\end{array}
$$





Security guarantees

$\rightarrow \subseteq c \times c$ $\Gamma \vdash c$

$\rightarrow \subseteq d \times d$ $\Gamma \vdash d$

$\rightarrow \cup \rightarrow \subseteq c[d] \times c[d]$ $\Gamma \vdash c[d]$

Theorem

If $\Gamma \vdash c[d]$ then

$c[d]$ satisfies noninterference

Timing-sensitive & termination-insensitive

Security guarantees

Theorem

If $\Gamma \vdash c[d]$ then

$c[d]$ satisfies noninterference

If $m_1 \approx_L m_2$

and $\langle c, m_1, 0 \rangle \Longrightarrow^* \langle \text{stop}, m_1', t_1 \rangle$

$\langle c, m_2, 0 \rangle \Longrightarrow^* \langle \text{stop}, m_2', t_2 \rangle$

then $m_1' \approx_L m_2'$ and $t_1 = t_2$

Implementation

Type checker and interpreter in Haskell (3500 LOC)

Case studies

Secure cooperative **thread scheduling** (650 LOC)

Noninterference → Scheduling of L threads is independent of **H** threads

Secure mark-and-sweep **garbage collection** (300 LOC)

Noninterference → Garbage collection of L allocations is independent on **H** allocations

Conclusion

- Zee supports provably secure usage of
  - Higher-order functions
  - Runtime type analysis
  - Heterogeneous arrays

- Allows for the implementation of timing-sensitive
  - Garbage collectors
  - Thread schedulers

Questions?