Programming Languages and Techniques (CIS120)

Lecture 13

February 17, 2014

ASMs and Aliasing

Announcements

- Homework 4 due tomorrow at midnight
- Midterm 1 will be in class on Friday, February 21st
  - ROOMS:
    - Towne 100 (here) last names: A – L
    - DRLB A1 last names: M – Z
  - TIME: 11:00 AM sharp, 50 mins
  - Covers up to Feb 12th and HW 4
    - no Abstract Stack Machine!
- Review session Wednesday, Feb 19th, 7-9PM in Levine 101
- HW 5 will be available Friday (after the exam) and due the following Friday
- Read Ch. 15 and 16

Recap

The Course So Far…

- We started out focusing on *pure* expressions with no side effects (variable mutation, etc.)
  - Strictly speaking, we did use a few "impure" features, for printing and running tests, but we omitted these from discussions of how programs evaluate
- Pure computations are all we need for a wide range of tasks
  - easier to parallelize
  - easier to reason about, for both humans and automatic tools such as typecheckers (because of the lack of "side channels")
  - simple execution model, substituting expressions by their values "in place"
- However, side-effecting computations are sometimes useful
- To understand their subtleties, a more sophisticated execution model is needed…

Abstract Stack Machine

- Three “spaces”
  - workspace
    - contains the expression the computer is currently working with
    - machine operation gradually simplifies expression to value
  - stack
    - temporary storage for variables, replacing substitution: used for `let` bindings and function parameters
    - maps variable names to atomic values (primitive values or references to heap locations)
    - also stores *workspaces* in a function call
  - heap
    - models your computer’s memory
    - storage area for large data structures (datatypes, tuples, first-class functions, records)
    - tracks the locations of data structures

Simplification

Workspace

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

4

Stack

x	22
y	24

Heap

DONE!

Simplification

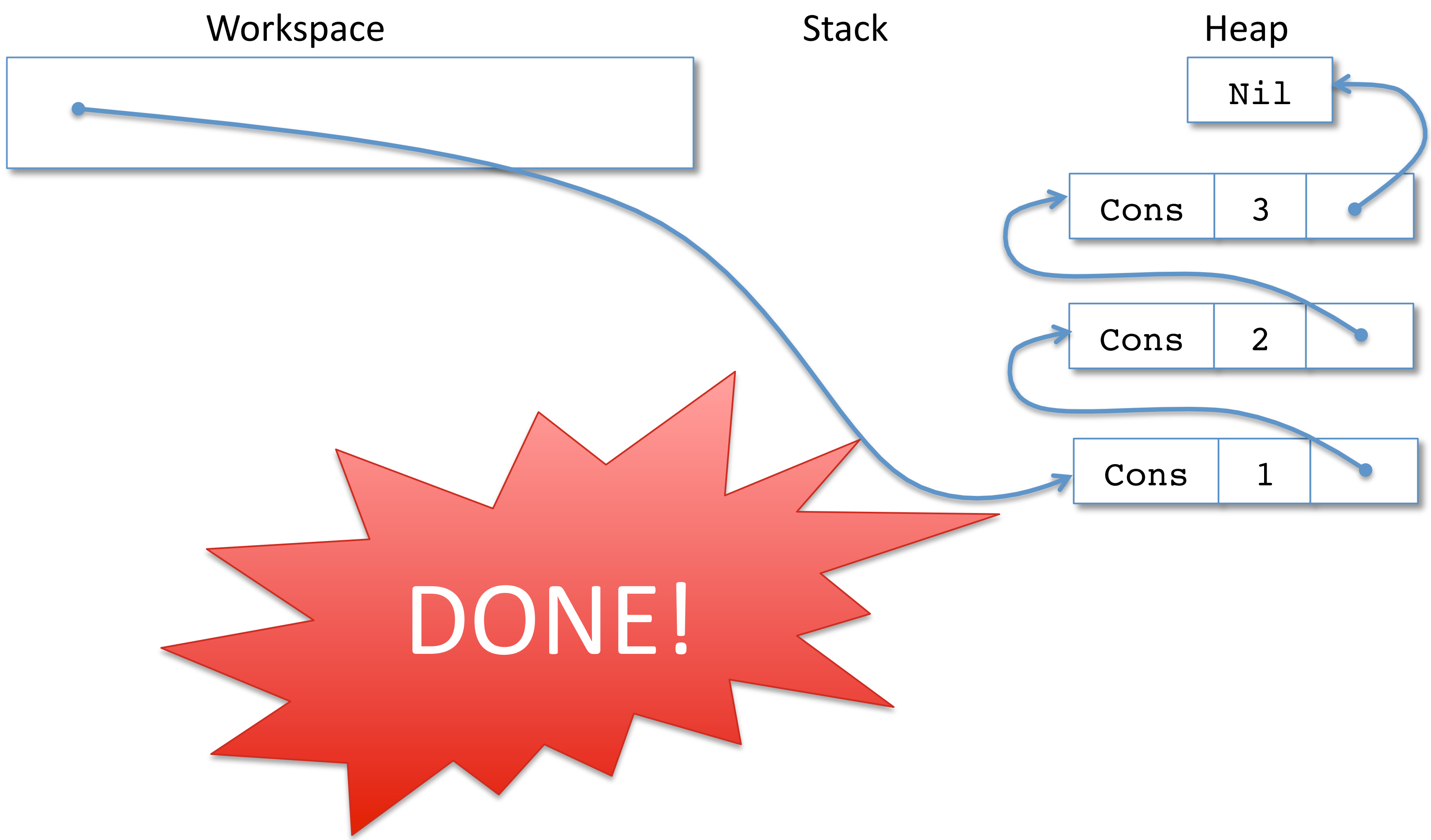
Workspace | Stack | Heap

```
Cons (1,Cons (2,Cons (3,Nil)))
```

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification



ASM and shadowing

Simplification

Workspace

```
let x = 10 + 12 in
let x = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Note that the second x *shadows* the first.

Simplification

Workspace | Stack | Heap

```
let x = 10 + 12 in
let x = 2 + x in
  if x > 23 then 3 else 4
```

Simplification

Workspace

```
let x = 22 in
let x = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 22 in
let x = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let x = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let x = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let x = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

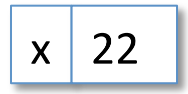
Heap

Simplification

Workspace

```
let x = 24 in
  if x > 23 then 3 else 4
```

Stack



Heap

Simplification

Workspace

```
let x = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22

x	24

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
x	24

Heap

Looking up x in the stack proceed from most recent entries to the least recent entries – the “top” (most recent part) of the stack is toward the bottom of the diagram.

Simplification

Workspace

```
if 24 > 23 then 3 else 4
```

Stack

x	22
x	24

Heap

Simplification

Workspace

```
if 24 > 23 then 3 else 4
```

Stack

x	22
x	24

Heap

Simplification

Workspace

```
if true then 3 else 4
```

Stack

x	22
x	24

Heap

Simplification

Workspace

```
if true then 3 else 4
```

Stack

x	22
x	24

Heap

Simplification

Workspace

3

Stack

x	22
x	24

Heap

DONE!

What is your current level of comfort with the Abstract Stack Machine?

1. got it well under control
2. OK but need to work with it a little more
3. a little puzzled
4. very puzzled
5. very *very* puzzled :-)

Mutable Records and the ASM

What is the value of `ans` at the end of this program?

```
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int  = p2.x <- 17; p1.x
```

1. 17
2. 1
3. sometimes 17 and sometimes 1
4. f is ill typed

Answer: 17

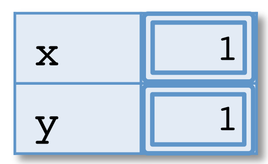
Mutable Records

- The reason for introducing all this ASM stuff is to make the model of heap locations and sharing *explicit*.
  – Now we can say what it means to mutate a heap value in place.

```
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int = p2.x <- 17; p1.x
```

- We draw a record in the heap like this:
  – The doubled outlines indicate that those cells are mutable
  – Everything else is immutable
  – (field names don't actually take up space)



A point record in the heap.

Allocate a Record

Workspace

Stack

Heap

```
let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

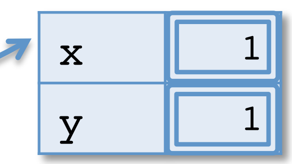
Allocate a Record

Workspace

```
let p1 : point =
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

Stack

Heap



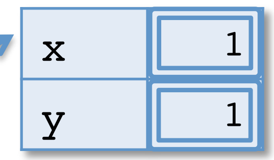
Let Expression

Workspace

```
let p1 : point =   .
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

Stack

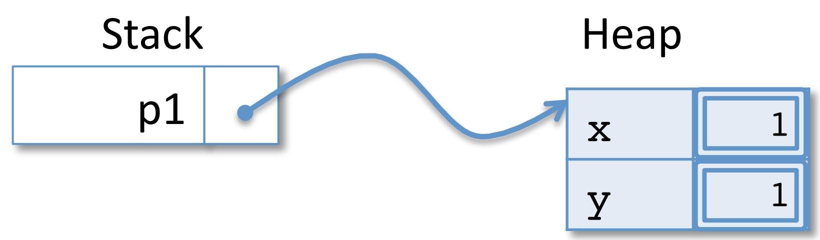
Heap



Push p1

Workspace

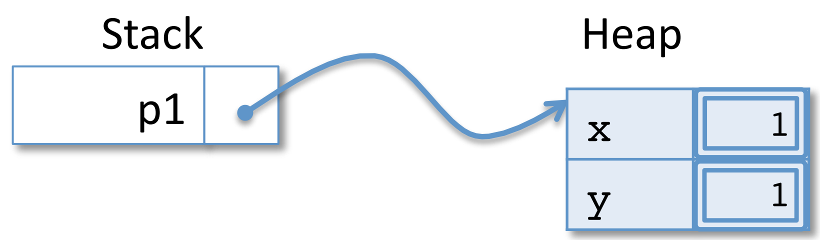
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p1’

Workspace

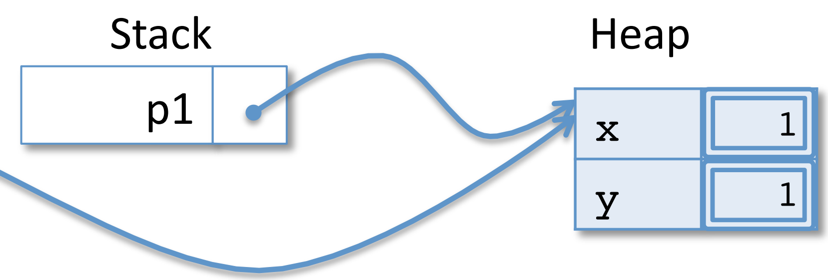
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



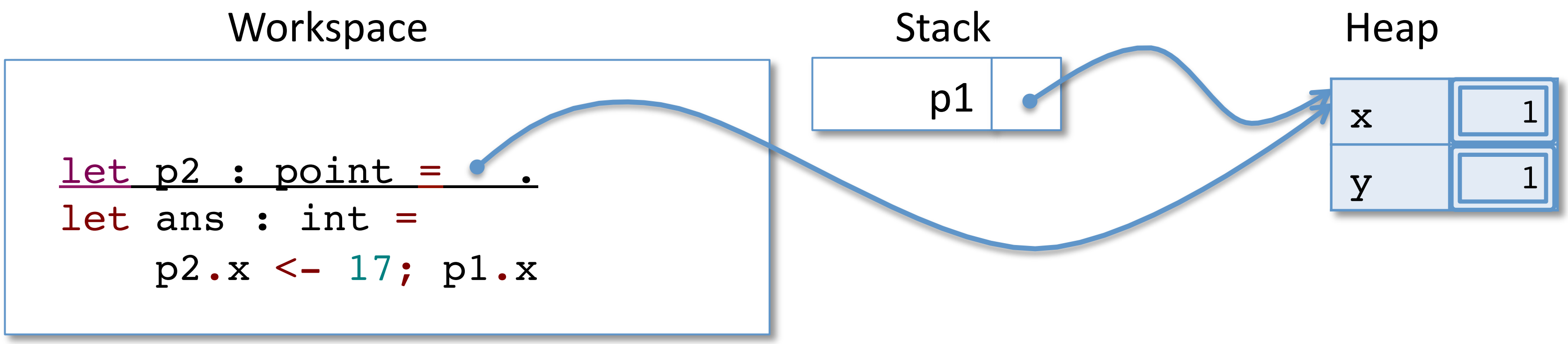
Look Up ‘p1’

Workspace

```
let p2 : point =
let ans : int =
    p2.x <- 17; p1.x
```



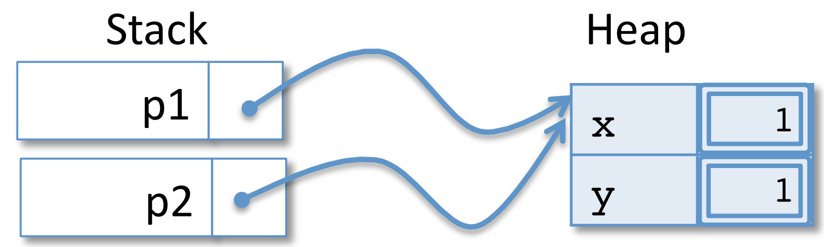
Let Expression



Push p2

Workspace

```
let ans : int =
    p2.x <- 17; p1.x
```

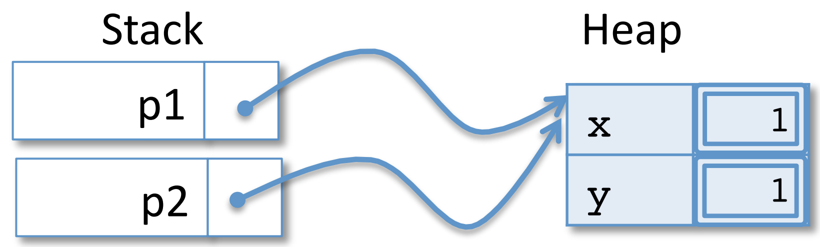


Note: p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same thing*.

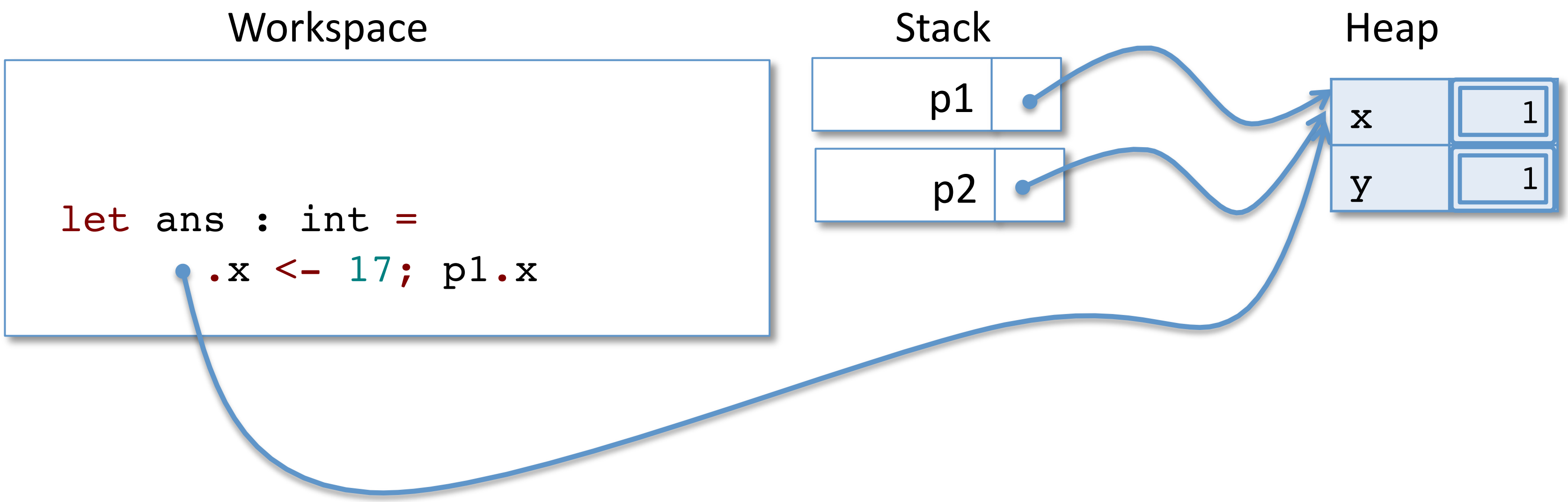
Look Up ‘p2’

Workspace

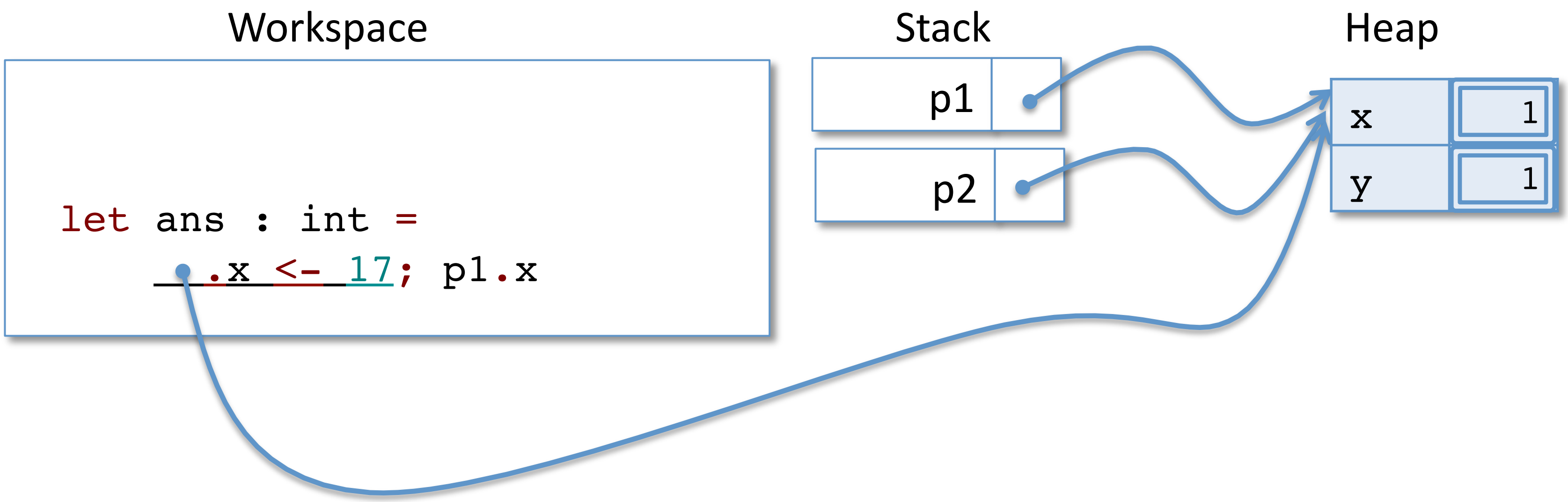
```
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p2’



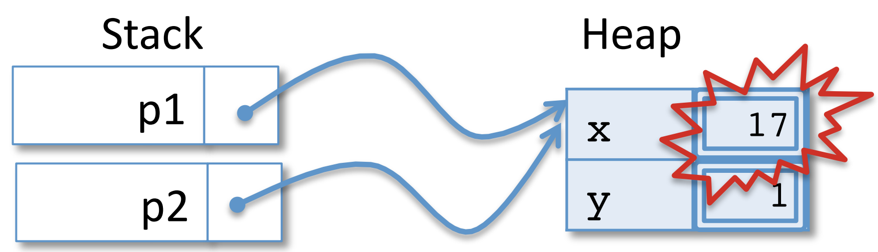
Assign to x field



Assign to x field

Workspace

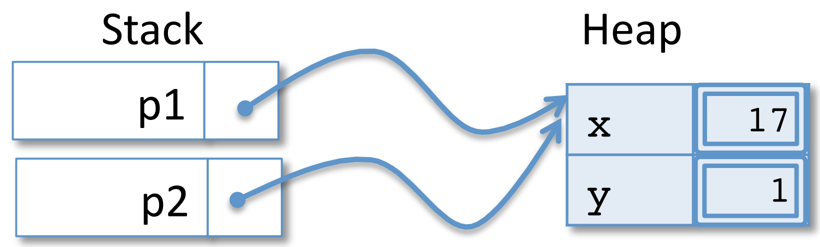
```
let ans : int =
    (); p1.x
```



Sequence ‘;’ Discards Unit

Workspace

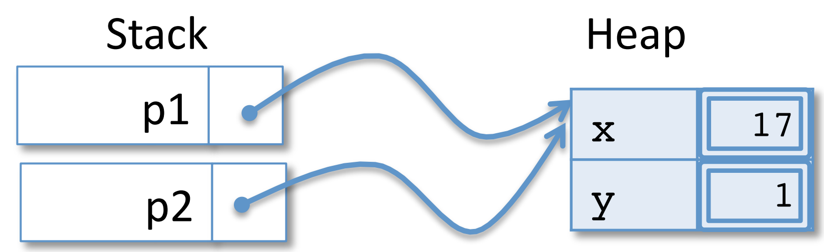
```
let ans : int =
    (); p1.x
```



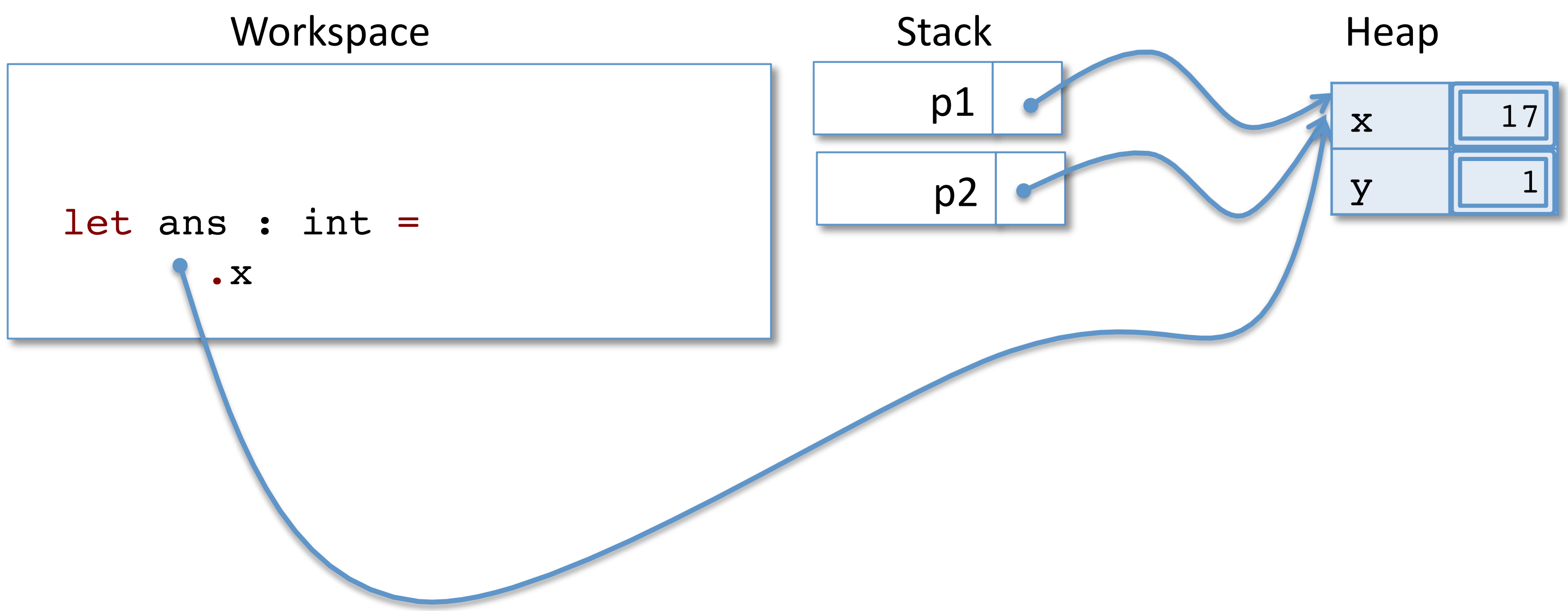
Look Up 'p1'

Workspace

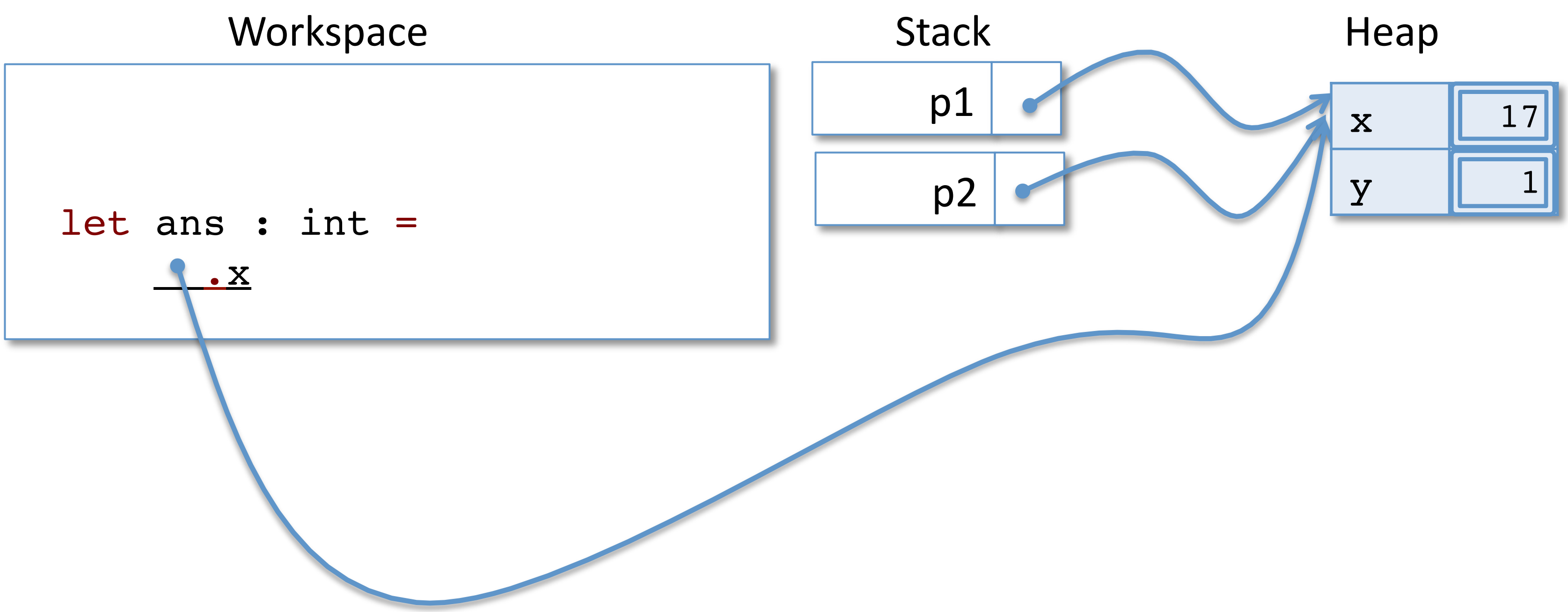
```
let ans : int =
    p1.x
```



Look Up ‘p1’



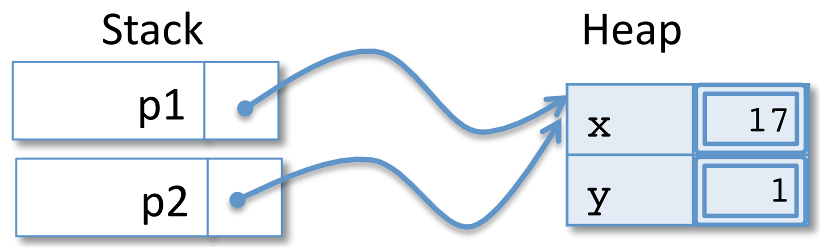
Project the 'x' field



Project the 'x' field

Workspace

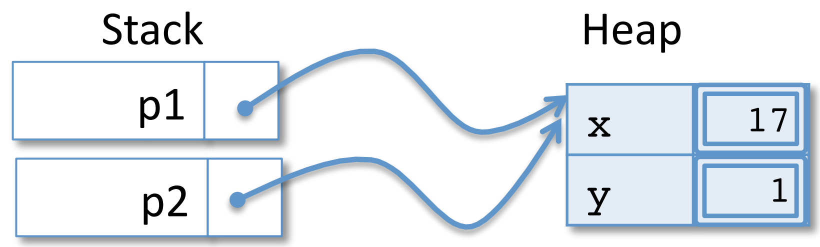
```
let ans : int =
    17
```



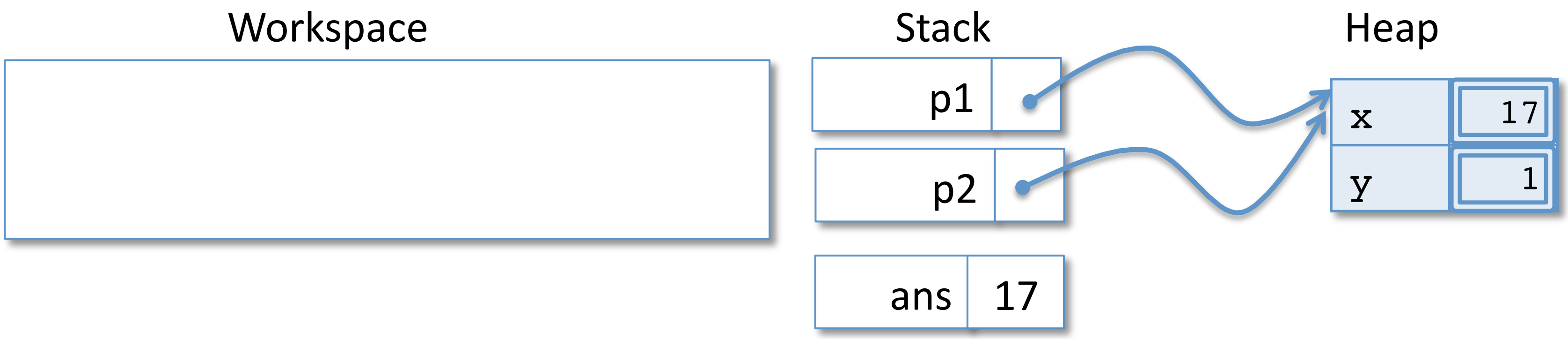
Let Expression

Workspace

```
let ans : int =
    17
```



Push ans



What answer does the following expression produce?

```
let p1 = {x=0; y=0} in
let p2 = p1 in
p1.x <- 17;
p2.x <- 42;
p1.x
```

1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: 42

What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: sometimes 17 and sometimes 42

What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: 17

Reference and Equality

= vs. ==

Reference Equality

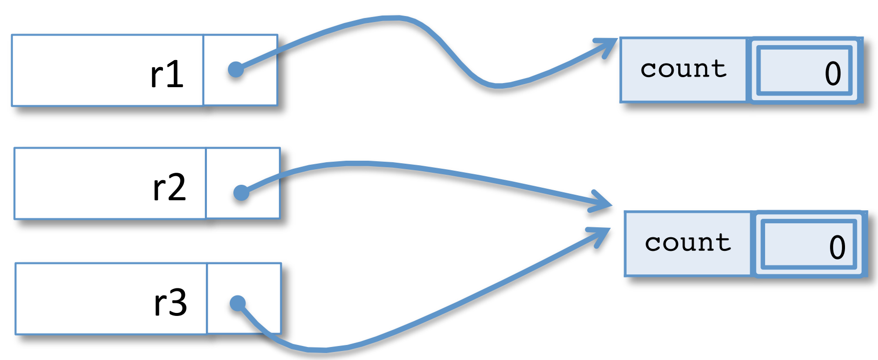
- Suppose we have two counters. How do we know whether they share the same internal state?
  - `type counter = { mutable count : int }`
  - We could increment one and see whether the other's value changes.
  - But we could also just test whether the references alias directly.

- Ocaml uses '==' to mean *reference* equality:
  - two reference values are '==' if they point to the same thing in the heap; so:

```
r2 == r3

not (r1 == r2)

r1 = r2
```



Structural vs. Reference Equality

- Structural (in)equality: `v1 = v2` `v1 <> v2`
  - recursively traverses over the *structure* of the data, comparing the two values' components for structural equality
  - function values are never structurally equivalent to anything
  - structural equality can go into an infinite loop (on cyclic structures)
  - appropriate for comparing *immutable* datatypes

- Reference equality: v1 == v2 v1 != v2
  - Only looks at where the two references point in the heap
  - function values are only equal to themselves
  - equates strictly fewer things than structural equality
  - appropriate for comparing *mutable* datatypes

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in

p1 = p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = { x = 0; y = 0; } in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: false

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 = l2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: false