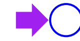
Assigning to a Variable

What is the result of this program?

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

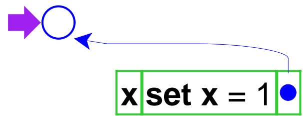
Is it 0 or 1?

Assigning to a Variable



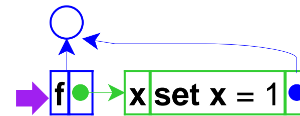
```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assigning to a Variable



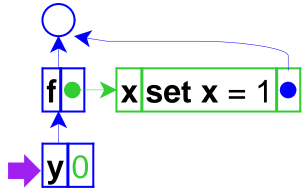

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assigning to a Variable



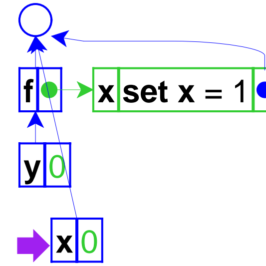

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assigning to a Variable



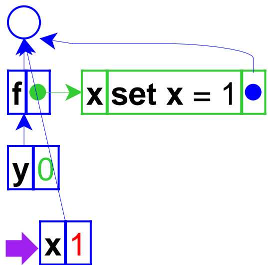

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assigning to a Variable



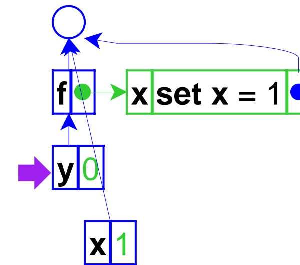

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assigning to a Variable




```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assigning to a Variable



So the answer is 0

```
let f = proc(x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Variables in C++

```
void f(int x) {
  x = 1;
}

int main() {
  int y = 0;
  f(y);
  return y;
}
```

The result above is 0, too

Variables in C++

```
void f(int& x) {
  x = 1;
}

int main() {
  int y = 0;
  f(y);
  return y;
}
```

But the result above is 1

Variables in C++

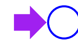
```
void f(int& x) {
  x = 1;
}

int main() {
  int y = 0;
  f(y);
  return y;
}
```

This example shows ***call-by-reference***.

The previous example showed ***call-by-value***.

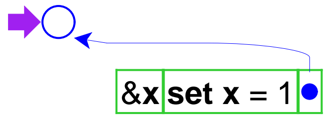
Assignment and Call-by-Reference



Adding call-by-reference parameters to our language

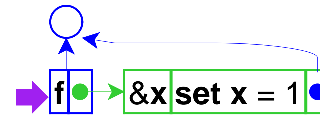
let **f** = **proc**(&**x**) **set** **x** = 1
  in **let** **y** = 0
    in { (**f** **y**);
      y   }

Assignment and Call-by-Reference



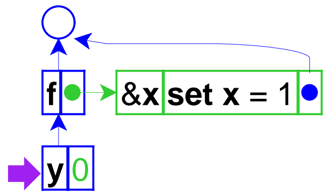

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assignment and Call-by-Reference



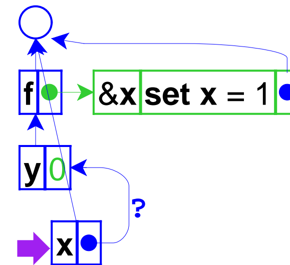

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assignment and Call-by-Reference




```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

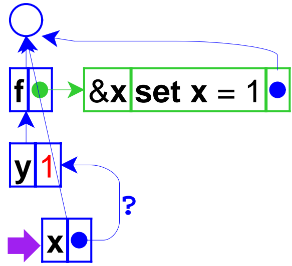
Assignment and Call-by-Reference



The pointer from one environment frame to another is questionable, because frames are supposed to point to values

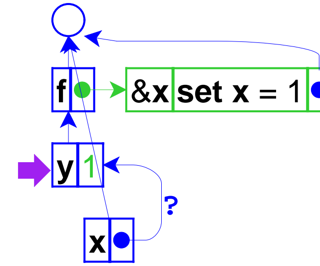
```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assignment and Call-by-Reference




```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Assignment and Call-by-Reference




```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Interpreter Changes

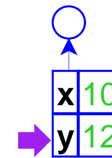
Same as before:

- Expressed values: Number + Proc
- Denoted values: Ref(Expressed Value)

The difference is that application doesn't always create a new location for a new variable binding

=> Separate *location* creation from *environment* extension

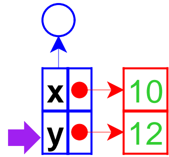
Assignment and Call-by-Reference



The old way

```
let x = 10
    y = 12
  in +(x,y)
```

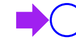
Assignment and Call-by-Reference



The new way

```
let x = 10
    y = 12
 in +(x,y)
```

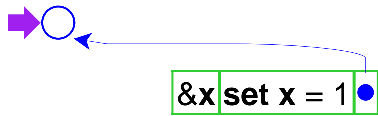
Call-by-Reference



Do the previous evaluation the new way...

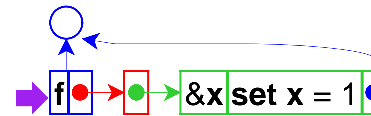
```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                  }
```

Call-by-Reference



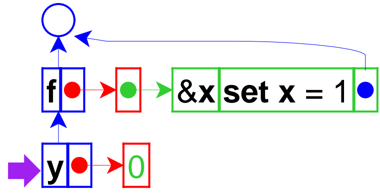

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                  }
```

Call-by-Reference



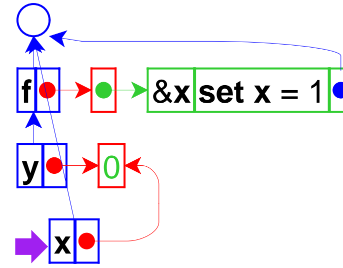

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                  }
```

Call-by-Reference




```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

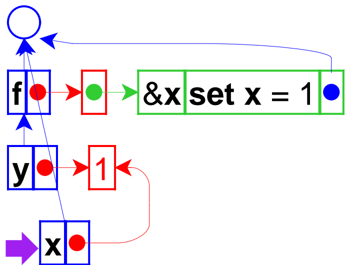
Call-by-Reference



This time, the new environment frame points to a location box, which is consistent with other frames

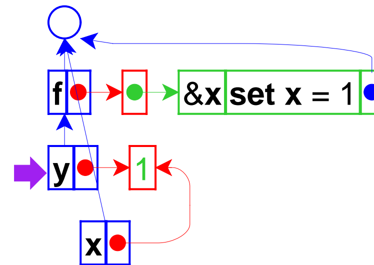
```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Call-by-Reference



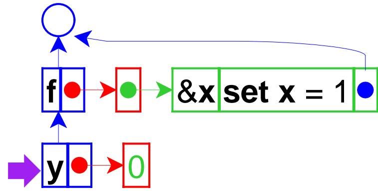

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

Call-by-Reference




```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f y);
          y                    }
```

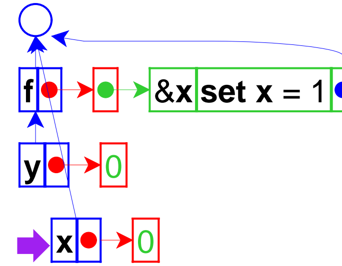
Call-by-Reference with Non-variables



If call-by-reference argument is not a variable...

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f 0);
          y                    }
```

Call-by-Reference with Non-variables



... create a location

```
let f = proc(&x) set x = 1
  in let y = 0
     in { (f 0);
          y                    }
```

Interpreter Changes

- Add call-by-reference arguments (indicated by &)
- New **var** datatype, with `cbv-var` and `cbr-var` variants
- Create explicit locations for variables

```
location : expval -> location
location-val : location -> expval
location-set! : location expval -> void
```

- Change variable lookup to de-reference locations
- Change **set** to work on locations
- Add `eval-fun-rands` and change `apply-proc`

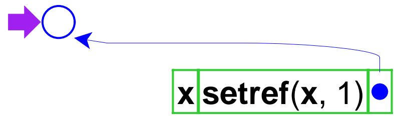
& versus * in C++

```
void f(int* x) {
  *x = 1;
}

int main() {
  int y = 0;
  f(&y);
  return y;
}
```

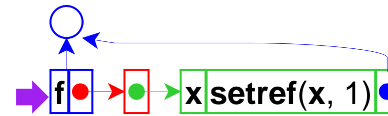
- This is back to ***call-by-value***, but with a reference as a value
- To study this form of call, we can add explicit references to our language, too

Call-by-Value with References



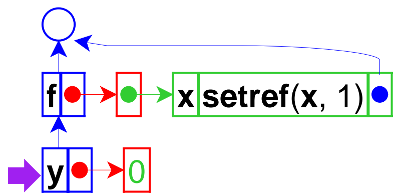

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                        }
```

Call-by-Value with References



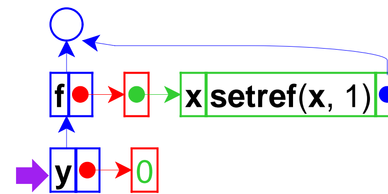

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                        }
```

Call-by-Value with References



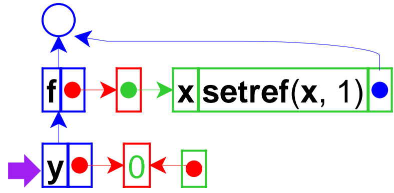

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                        }
```

Call-by-Value with References



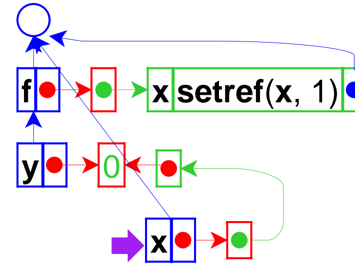

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                        }
```

Call-by-Value with References



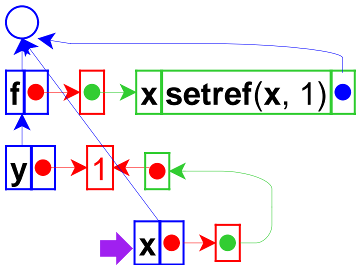

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                              }
```

Call-by-Value with References



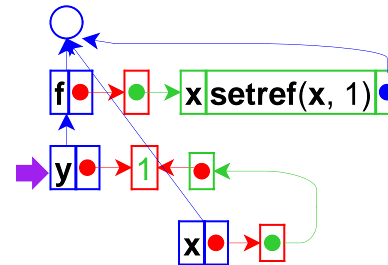

```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                              }
```

Call-by-Value with References




```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                              }
```

Call-by-Value with References




```
let f = proc(x) setref(x, 1)
 in let y = 0
    in { (f ref(y));
         y                              }
```

Interpreter Changes for References

Revised language:

- Expressed vals: Number + Proc + Ref(Expressed Val)
- Denoted vals: Ref(Expressed Val)

Interpreter changes:

- Add reference values
- Add **`ref`** form and **`setref`** primitive