

IA010: Principles of Programming Languages

State and Side-Effects

Achim Blumensath

blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Assignments

Side-Effect:

- mutating memory and IO
- Even purely functional programs **must** support side-effects.

$$\langle expr \rangle ::= \dots \mid \mathbf{skip} \mid \mathbf{print} \langle msg \rangle \langle expr \rangle \mid \langle expr \rangle ; \langle expr \rangle$$
$$\mid \langle id \rangle := \langle expr \rangle$$

```
let x = 1;
print "x has value: " x;
x := 2;
print "now x has value: " x;
```

Ramifications

(a) evaluation turns

from $env \rightarrow val$

to $env \times state \rightarrow val \times state$

(b) identifiers turn

from **constants** with a value (**r-values**)

to **variables** with a memory location (**l-values**)

⇒ changes the notion of an environment

(c) evaluation order matters

```
let x = 0;
```

```
let y = (x := 1; 3) + (x := 2; 4);
```

```
x + y
```

⇒ makes lazy evaluation impractical

Ramifications

- (d) allows **uninitialised data structures**
 - needed for mutually recursive structures
 - source of hard to find bugs
- (e) **aliasing**
 - we need to distinguish between
 - “have the same value” and “have the same memory address”
 - might require frequent copying of data structures
- (f) **clean up** code
 - in conjunction with **error checking** and/or **exceptions**:
 - lot of work and error prone
 - **finally** and **defer** statements

Discussion

Advantages

- drastically increases expressive power
- solutions without side-effects can be substantially more complicated or inefficient (RNG, debug output,...)

Disadvantages

- error prone
- adds implicit interactions between program parts (encapsulation)

⇒ separation between pure and impure parts desirable

Parameter passing

```
let f(x) { x := 1; };  
let y = 0;  
f(y);  
y
```

Parameter modes: in, out, in/out

Calling conventions

- call-by-value
- call-by-result
- call-by-value/result, call-by-copy, call-by-copy-result
- call-by-reference
- call-by-name
- call-by-need
- call-by-macro-expansion

Call-By-Value

Call-By-Result

```
f(in x, out y, out z) {  
  x := x + 1;  
  y := x + 1;  
  z := x + 2;  
};  
let u = 0;  
f(u,u,u);
```

Call-By-Copy

```
incr(inout x) {  
  x := x + 1;  
};  
let u = 0;  
incr(u);
```


Call-By-Reference

```
let u = 1;  
let v = 0;  
f(x, y) {  
  x := x + u - v;  
  y := y + u - v;  
};  
f(u, v)
```

Call-By-Name

```
let sum(k, l, u, expr) {  
  let s = 0;  
  for k = l .. u {  
    s := s + expr;  
  };  
  s;  
};  
sum(i, 1, 100, i*i)
```

Discussion

Standard

- **call-by-value** for languages with side-effect
- **call-by-need** for those without
- **call-by-reference** for declarative languages

Notes

- call-by-value reduces **aliasing** (plus copying of data structures)
- call-by-reference can be simulated with **reference** or **pointer types**

Memory management

Kinds

- manual
- automatic
- type based

Problems

- dangling pointers
- unreachable objects

Manual memory management

- gives programmer full control
- tedious, error prone, hard to debug
- (de-)allocation of memory not cheap

Manual memory management

- gives programmer full control
- tedious, error prone, hard to debug
- (de-)allocation of memory not cheap

Automatic memory management

- reference counting
 - easy to implement
 - very slow
 - does not support cyclic data structures
- garbage collection
 - hard to implement
 - much faster
 - hard to control runtime impact

Manual memory management

- gives programmer full control
- tedious, error prone, hard to debug
- (de-)allocation of memory not cheap

Automatic memory management

- reference counting
 - easy to implement
 - very slow
 - does not support cyclic data structures
- garbage collection
 - hard to implement
 - much faster
 - hard to control runtime impact

Type based memory management

- minimal runtime overhead
- typing is very restrictive and requires more work

Loops

$\langle expr \rangle ::= \dots \mid \mathbf{while} \langle expr \rangle \{ \langle expr \rangle \}$
 $\mid \mathbf{for} \langle id \rangle = \langle expr \rangle \dots \langle expr \rangle \{ \langle expr \rangle \}$

Loops

$\langle expr \rangle ::= \dots \mid \mathbf{while} \langle expr \rangle \{ \langle expr \rangle \}$
 $\mid \mathbf{for} \langle id \rangle = \langle expr \rangle \dots \langle expr \rangle \{ \langle expr \rangle \}$

Goto

$\langle expr \rangle ::= \dots \mid \mathbf{label} \langle id \rangle \mid \mathbf{goto} \langle id \rangle$

- more expressive
- can be misused
- can improve code

Loops

$\langle expr \rangle ::= \dots \mid \mathbf{while} \langle expr \rangle \{ \langle expr \rangle \}$
 $\mid \mathbf{for} \langle id \rangle = \langle expr \rangle \dots \langle expr \rangle \{ \langle expr \rangle \}$

Goto

$\langle expr \rangle ::= \dots \mid \mathbf{label} \langle id \rangle \mid \mathbf{goto} \langle id \rangle$

- more expressive
- can be misused
- can improve code

Special cases

$\langle expr \rangle ::= \dots \mid \mathbf{break} \mid \mathbf{continue} \mid \mathbf{return} \langle expr \rangle$

Usages of side-effects

- recursive data structures

Usages of side-effects

- recursive data structures
- efficiency: reusing space, avoiding copies

Usages of side-effects

- recursive data structures
- efficiency: reusing space, avoiding copies
- passing values via global variables (RNG, logging,...)