Why side effects?

- **Purely functional programs are computationally complete.**
- **Why bother with side effects?**
  - Reminder: "side effect" = anything that's not evaluation
  - e.g.: changing the value in an updatable (**mutable**) data location, printing to screen

To model world?

- "World changes --- to model it, need side effects"
- Wrong --- can always model changing world using function of type

```
World -> World
```

- Takes "old world", returns "new world"
- Like list reverse, which returns fresh list instead of updating old list

So why then?

1. Efficiency
2. Expressiveness
3. Permissiveness
4. Interaction with outside world
5. Abstraction/ease of evolution

1. Efficiency

- **Purely functional programs make many copies of data**
  - e.g., list functions return new lists
- **Naive compilers will produce code that spends time and space constructing all these copies**
- **Solutions...**
  - compilers
  - type systems

Smart compilers

- Can eliminate some (not all) copies by analysis
- However:
  - Require considerable investment to write
  - May have slow compilation time
  - May require whole-program knowledge
  - Still doesn't get all the copies
- Ongoing research problem

Smart type systems

- **"Linear type systems" can restrict uses of data**
  - can make some data types "uniquely pointed to"
  - if argument to reverse is unique pointer to that list, the cells can be reused instead of being copied (no other client can access the previous list value; it is garbage)
- **However:**
  - **Can be difficult for programmers to learn**
  - **Can be too restrictive for many practical programming idioms**
- **Ongoing research problem**

(On the other hand)

- **Use of immutable data can encourage sharing**
  - Different users of a data structure don't need to worry about one mutating it in an unacceptable way
- **Sometimes this sharing leads to efficiency *gains***
- **However, these benefits can be realized in an impure language simply by using immutable data structures**

2. Expressiveness

- **Some data structures *inherently* hard to express in pure languages, e.g.:**
  - **Cyclic data structures**
    - doubly linked lists
    - trees where nodes have parent pointers
  - **Incrementally initialized data structures**
    - arrays where element values depend on previously computed element values

Doubly linked lists

```
datatype 'a DList =
    DEmpty
  | DNode of {elem:'a,
             prev:'a DList,
             next:'a DList};

val empty_dlist = DEmpty
val single_dlist =
  DNode {elem=25,
         prev=DEmpty,
         next=DEmpty};
```

Doubly linked lists

```
datatype 'a DList =
     DEmpty
   | DNode of {elem:'a,
               prev:'a DList,
               next:'a DList};

fun prepend x Empty =
      DNode {elem=25, prev=DEmpty, next=DEmpty}
  | prepend x (DNode {elem, prev, next}) =
      DNode {elem=x, prev=DEmpty,
             next=(DNode {elem=elem,
                          prev=(XXX?),
                          next=(YYY?)})};
```

Incrementally initialized arrays

- **Hard to write array constructor expression if later elements' values are computed from previous ones**

```
[2, f(this[0]), ... ]?
```

- **Purely functional solutions tend to be baroque**
- **Can make constructors into primitives (like Array.fromList)...**
  - (But then you're just admitting defeat.)

3. Permissiveness

```
fun copy (w:world) = (w, w);
```

- But there should only be one world
- No such problem if world is implicit (just current state of memory)
- Again, linear type systems can help, with caveats mentioned previously

4. Interaction

- **I/O inherently "side-effecting"**
- **E.g., network card buffer:**
  - When data arrives, that *specific spot* in memory changes
  - When you need to send data, you'd better put the new data in *that specific spot* in memory
- **Can push down into runtime system; again, this is admitting defeat**
- (Haskell is pure; it uses *monads* for I/O, which are nice but suffer from analogous problem to "threading-the-world problem" (next slide))

5. Evolution/abstraction

- **When modeling side effects by explicit "world" argument/return, all potentially side-effecting functions must take and return world**
  - e.g., If f takes an int and updates the world, it must be of type

    ```
    int * World -> World
    ```
- **So f's callers must *also* take/return the world**
- **Result: world gets "threaded" through call chain, with some annoying results**

Evolution example

- Suppose f initially is pure...

```
fun f x = x + x;
```

- ...but evolves to require a side effect:

```
fun f x =
    let val _ = Log.append "debug: x = "
                ^ (Int.toString x)
    in x + x end;
val f : fn int -> int
```

- In impure language, this is simple

Evolution example

- In pure language, we must pass/return a "world" to model side effects
- So, we must add a "world" to x's arg and return value

```
fun f (x, l:Log) =
    let val newLog = Log.append ...
    in (x + x, newLog);
val f = fn : (int * Log) -> (int * Log)
```

- We must now update all of f's callers, and their callers, etc. recursively up the call chain!

Abstraction

- Evolution problem is really special case of more general problem:
  - **In purely functional code, impossible to abstract away side effects**
  - Caller forced to know about fn's side effects
  - Often good (side effects are important, & should be documented), but not always
  - e.g., if function has "pure" interface but internally may cache previously computed values for efficiency

Conclusion

- My belief: With language and compiler technology available in 2004, side effects are a necessary evil in a practical language.
- (Caveat: Haskell community disagrees, & they have successfully written large programs.)