## The Aggregate Update Problem

- With pure functional languages we are lumbered with the *aggregate update* problem:
  - manipulating large datastructures by copying is too expensive
  - performing I/O with copying is impossible
- The conclusion is that
  - We cannot allow arbitrary pure programs to do I/O
  - Any large aggregate updates will be very expensive
  - Why use the word "arbitrary" above ?

## Real-world programs aren't arbitrary

- A "real-world" programmer wouldn't write something like `(bigds,modify p bigds)`
- The idea of wanting to have both the original and modified versions of a large data-structure simultaneously available would seem absurd.
- Nor would any reasonable programmer want to access new and old versions of files.
- In practise, in real programs, once we modify something large or external, we never expect to see the old version.
  - — unless we make explicit provision for some form of "undo"
  - — or are required to maintain an "audit trail" !

## Single-Threadedness

- The use of a data value `ds` is "single-threaded"[1] if
  - There is only ever one live reference to it.
  - Once a function has been applied to it (`f ds`), the program no longer refers to `ds`.
- Non-single threaded examples:
  - `f ds = (g ds, h ds)`
    We have two live references to `ds` once `f` is evaluated
  - ```
    let ds1 = f ds
        ds2 = g ds in ...
    ```
    After `ds1`, we still proceed to refer to `ds` in the next line
- Single-threaded example:
  ```
  let ds1 = f ds
      ds2 = g ds1 in ...
  ```
  The application of `g` is to the *result* of the application of `f`

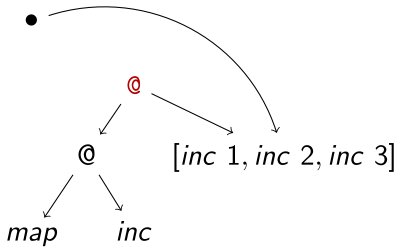
[1] Nothing to do with "threads" in concurrent programming

## Optimising Single Threadeness

- If a function is applied to a single-threaded argument ...
  - destructive update does not destroy referential transparency
- Copying is only necessary to ensure purity if arguments are "multi-threaded"
- So, the destructive update optimisation we saw for
  `map inc xs`
  is valid if no further reference to `xs` exists in the program.
- We can implement I/O (destructively) in a pure language if
  - we can ensure all references to files/outside world are single-threaded.
  ```

## Revisiting optimised(?) `map inc [1..3]`

- We might suggest that we update the list in place and swing our application pointer to indicate that update:




- If we can show there are no further live pointers to either the original list `[1,2,3]`, or the topmost `@` node, then the above destructive update is safe and does not affect referential transparency.

## Enforcing Single-Threadedness

There are two ways to enforce single-threadedness.

1. "Invent and Verify"
   - Programmers write program in their own way.
   - Compiler checks for single-threadedness where required.
   - Implemented in language "Clean"
     using the type system (uniqueness types)
   - Undecidable, so valid programs may be rejected
2. "Correctness by Construction"
   - Required single-threadedness enforced by compiler
   - i.e. Special restricted sub-language for I/O
   - Implemented in Haskell
     using abstract `IO` type and "monads".
   - Can't write all possible single-threaded programs

## I/O in Haskell

- We are now going to explore how Haskell supports I/O with all its side-effects, whilst also maintaining referential transparency w.r.t. the "natural" functional semantics.
- First we shall look at a few file operations to note their (destructive) side-effects
- Then we shall introduce the Haskell *IO* type constructor.

## File I/O Operations : File Open/Close

- `openFile`
  - Input: pathname, opening mode (read/write)
  - Effect: modifies filesystem by creating new file
  - Return Value: handle to new file
- `hClose`
  - Input: file handle
  - Effect: closes file indicated by handle, modifying filesystem
  - Return Value: none
- Real-world items affected: filesystem, file status
- opening modes: `ReadMode`, `WriteMode`, ...

## File I/O Operations : File Put/Get

- `hPutChar`
  Input: file handle, character
  Effect: modifies file by appending the character
  Return Value: none
- `hGetChar`
  Input: file handle
  Effect: reads character from file current-position, which is then incremented
  Return Value: character read
- Real-world items affected: contents of and positioning in open files

## I/O in Haskell

- Functions that do I/O (as a side-effect) use a special abstract datatype: `IO a`
- Type `IO a` denotes a "value":
  - whose evaluation produces an I/O side-effect
  - which returns a value of type `a` when evaluated.
  - such "values" are called "I/O-actions".
- I/O-actions that don't return a value have type `IO ()`
  - Type `()` is the singleton (a.k.a. "unit") type
  - It has only one value, also written `()`
- I/O-actions are usually invoked using special syntax ("`do`-notation").

## Other File I/O Types

- `data IOMode = ReadMode | WriteMode | ...`
  file opening mode
- `type FilePath = String`
  File Pathname — just a string.
- `data Handle = ....`
  File Handles — *pointers* to open files
- There are no types to represent file themselves, or the file-system.
- I/O in Haskell works by hiding references to external data that is destructively updated.

## Haskell File I/O Functions

- `openFile :: FilePath -> IOMode -> IO Handle`
  `openFile fp mode` is an I/O-action that opens a file and returns a new handle.
- `hClose :: Handle -> IO ()`
  `hClose f` is an I/O-action that closes file `f`, returning nothing.
- `hPutChar :: Handle -> Char -> IO ()`
  `hPutChar f c` is an I/O-action that writes `c` to file `f`, returning nothing.
- `hGetChar :: Handle -> IO Char`
  `hGetChar f` is an I/O-action that reads from file `f`, returning the character read.

## Haskell Console I/O Functions

- `putChar :: Char -> IO ()`
  `putChar c` is an I/O-action that writes `c` to `stdout`, returning nothing.
- `getChar :: IO Char`
  `getChar` is an I/O-action that reads from `stdin`, returning the character read.

## Composing I/O actions

- For this stuff to be useful we need some way of *composing* two actions together in a safe (that is, single-threaded) way.
- As ever in Haskell the solution is to come up with a function.
- Perhaps something with a type like this?

```
seqIO :: (IO a) -> (IO b) -> (IO b)
```

- And we could use it like this:

```
putAB = seqIO ( putChar 'a' ) ( putChar 'b' )
```

## Composing I/O actions

- Actually, this will read better if we make `seqIO` an infix function:

```
(>>)  :: IO a -> IO b -> IO b
infixl 1  >>

putAB = putChar 'a' >> putChar 'b'
```

- We need something more elaborate to handle IO actions that produce results
- If we pass on the result of the first action as an input to the second then we can make use of it in subsequent actions.

## Composing I/O actions

- `bindIO ::  (IO a) -> (a -> IO b) -> IO b`
- Obviously, we want to make this infix as well
- `(>>=) ::  (IO a) -> (a -> IO b) -> IO b`
- Easy to use this for simple compositions:
  `getput = getChar >>= putChar`
- We don't always want to make use of the result right away – one neat solution is to use a lambda abstraction:

```
swap2char
  = getChar >>=
    (\ c1 -> getChar >>=
      (\ c2 -> putChar c2 >> putChar c1))
```

## Building I/O actions

- Apart from the four file I/O actions just presented, we have ways to build our own.
- `return :: a -> IO a`
  `return x` is an I/O action that has no side-effect, and simply returns `x`.
- ```
  get2str :: IO String
  get2str
    = getChar >>=
       (\ c -> getChar >>=
          (\ d -> return [c,d]))
  ```

## Syntactic sugar for I/O actions

- We have some syntactic sugar for the `>>` and `>>=` functions.
- Consider

  ```
  getChar >>= (\ c -> getChar >>= (\ d -> return [c,d]))
  ```

- We can write a multiline version, dropping brackets, as

  ```
  getChar >>= \ c ->
  getChar >>= \ d ->
  return [c,d]
  ```

  Look at Haskell precedence and binding rules carefully.
- We can read this as:
  "`getChar`, call it `c`, `getChar`, call it `d`, and return `[c,d]`".

## Do-notation

- We have

  ```
  getChar >>= \ c ->
  getChar >>= \ d ->
  return [c,d]
  ```

- Haskell has syntactic sugar for this: "do-notation":

  ```
  do c <- getChar
     d <- getChar
     return [c,d]
  ```

- Restriction: the only way to compose actions is to sequence them.

## Invoking Actions in `do`-notation

- If action `act a b c` returns nothing, we simply call it

  ```
  act a b c
  ```

- If action `act a b c` returns a value we can either:
  - simply invoke it as is (discarding the return value)

    ```
    act a b c
    ```

  - or invoke it and bind the return value to a variable

    ```
    x <- act a b c
    ```

- The last action in a `do`-expression cannot bind its return value

  ```
  actlast a b c
  ```

  Its return value becomes that of the entire `do`-expression.
  ```

## File I/O Examples (I)

- Read character from one file, and write to another

```
fCopyChar :: FilePath -> FilePath -> IO ()
fCopyChar fromf tof
 = do ff <- openFile fromf ReadMode
      c <- hGetChar ff
      hClose ff
      tf <- openFile tof WriteMode
      hPutChar tf c
      hClose tf
```

- Notes:
  - no explicit reference to filesystem
  - no explicit reference to file/open file data, just a reference to the file handle pointer

## File I/O Examples (Ia)

- For comparison, here is the same function, but without using the "do" notation
- ```
  fCopyChar fromf tof
   = openFile fromf ReadMode >>= \ ff ->
     hGetChar ff             >>= \ c ->
     hClose ff               >>
     openFile tof WriteMode  >>= \ tf ->
     hPutChar tf c           >>
     hClose tf
  ```
- Why "do" notation was invented!
- But note that despite its imperative appearance, it is just function application using `>>` and `>>=`.
  ```