Mutable References

15-150, April 21, 2020

Frank Pfenning

Learning Goals

- How do we integrate mutability into functional languages?
- What are the consequences?
  - For reasoning
  - For parallelism
  - For data structures
  - For programming in general
- Benign effects
- Major example: memoizing streams

Outline

- The basics
  - type `t ref`
  - Aliasing and extensional equality revisited
- Example: accounts and transactions
- Example: a random number generator
- Example: memoizing streams

Mutability in Functional Languages

We only show the ML approach

- Fundamentally, two different approaches
  - Every expression can mutate references (SML, OCaml, …, Lisp)
  - Mutability is isolated (Haskell, …, Algol)
- Interesting trade-offs
  - SML/OCaml: programming is simplified (somewhat)
  - Haskell: reasoning is simplified (somewhat)
- Ultimately, we cannot escape the complications!
- Use mutability sparingly!

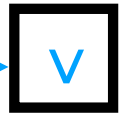
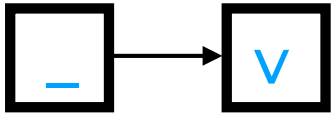
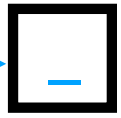
Type `t ref`

- For any type `t`, we have a new type `t ref`
- Values: Cells containing values of type `t`
  - In essence, an address (with no source-level representation)
- Constructor: `ref e : t ref` provided `e : t`
- Destructor: `!e : t` provided `e : t ref`
- Mutator: e_1 `:=` e_2 `: unit` provided e_1 `: t ref` and e_2 `: t`
- Discriminator: e_1 `=` e_2 for e_1 `: t ref` and e_2 `: t ref`

Dynamics

Write  **for a cell with contents** `v` (blank if irrelevant)

Cells are never “empty”

- `ref e ==>`  if `e ==> v` where cell is freshly allocated
- `!e ==> v` if `e ==>` 
- e_1 `:=` e_2 `==> ()` if e_1 `==>`  , e_2 `==> v`
  - with effect 
- e_1 `=` e_2 `==> true` iff e_1 `==>`  e_2 `==>`  where cells are the same (have the same address)
  - e_1 `=` e_2 `==> true` if e_1 and e_2 are aliases

Example: Bank Accounts

Let’s code!

Sequential Composition

```
(e1 ; e2 ; … ; en)
==
let val _ = e1
    val _ = e2
    …
    val v = en
in
  v
end
```

Some Observations

- Equality is no longer reflexive on expressions
  - Example: `ref 0 = ref 0 ==> false`
- Carefully need to reason about effects and their interactions
- Aliasing is tricky
- We never explicitly deallocate cells, but the garbage collector will remove them just like other data
- Evaluating expressions in parallel can lead to race conditions if they might access the same memory

Example: Random Numbers

Let’s code

Memoizing Streams

- An example of “benign effect”
  - Hidden behind abstraction boundary
  - Can still change behavior if suspension itself modifies state
- May be necessary if stream does input or output
  - Do not want input or output to be repeated!

Computation / Data & Functions

	Persistent Data / Pure Functions	**Ephemeral Data / Effectful Functions**
Sequential Computation	Functional programming is an excellent tool	Reasoning is more complicated (use only benign effects whenever possible)
Parallel Computation	Functional programming is an excellent tool	Reasoning is hard due to race conditions and deadlocks

Summary

- The basics
  - type `t ref`
  - Aliasing and extensional equality revisited
- Example: accounts and transactions
- Example: a random number generator
- Example: memoizing streams