Questions: Semantic and Computational Issues

Veneeta Dayal and Chung-chieh Shan

October 4, 2006

1. Warm up: type constructors

A value of type `Tree a` (from last time) is a binary tree with leaves labeled with `a`.

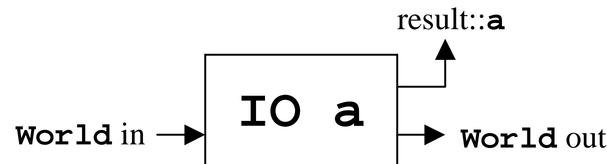
A value of type `Maybe a` (from the Haskell Prelude) is either `Nothing` or `Just` an `a`.

A value of type `List a` (well, actually it's notated `[a]`) is an ordered list of `a`.

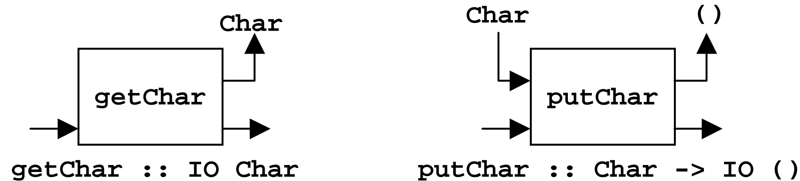
These types are comparable (belong to the type-class `Eq`) as long as `a` is comparable.

2. Actions as semantic values

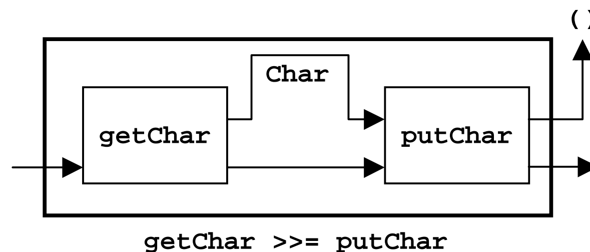
A value of type `IO a` is an action that results in `a`. It is *roughly* `World -> (a, World)`.



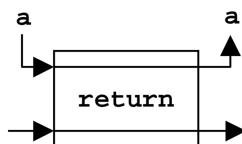
For example, reading a character (`getChar`) and printing a character (`putChar`) have the types below. (Recall that `()` is the *unit* type: it has just one value, also written `()`.)



We can combine two actions using `(>>=) :: IO a -> (a -> IO b) -> IO b`. The result of the first action (of type `a`) determines the second action (which results in `b`).



Finally, we can lift a value to an action using `return :: a -> IO a`.



These pictures are taken from "Tackling the awkward squad". Please read §§2.2–2.4.

3. Generalizing `IO` to monads

A very useful type class:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

Like `IO`, the three type constructors in §1 are instances of `Monad`. More monads:

- `Identity a = a`
- `Reader a = S -> a`, for any type `S` such as possible world
- `State a = S -> (a, S)`, for any type `S` such as information state
- `Continuation a = (a -> W) -> W`, for any type `W` such as truth value

(How do you define `return` and `>>=` for these monads other than `IO`?) In principle, "set" (questions, Hamblin) and "pointed set" (focus, Rooth) are also monads, but we cannot define `instance Monad Set` in Haskell because not all types `a` are comparable.

4. Wh-questioning as a side effect

```
data Suspend a = Done a | Ask (E -> a)
instance Monad Suspend where
    return a     = Done a
    Done a >>= k = k a
    Ask f  >>= k = Ask (\e -> f e >>= k)
ask :: Suspend E = Ask Done
```

"Who saw who?" "Who is who?"

```
> pretty (ask >>= \x -> ask >>= \y -> return (x == y))
```

"Who went to the store and bought what?"

```
> pretty (ask >>= \x -> if x == Alice then return True
                        else ask >>= \y -> return (x == y))
```