The Environment as an Argument

Context-Aware Functional Programming

Pedro Martins
Julie A. McCann
Susan Eisenbach

PADL 2012

Context-awareness

"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

Dey, 2000

Why Context-awareness matters

- Widespread availability of mobile computing devices.
- Appearence of practical context-aware applications.
- Existing research didn't see widespread practical adoption:
    - Inflexible and heavyweight.
    - A more declarative approach needed?

Our goals

- Context should be invisible in the eyes of the programmer.
- Contextual values should not behave differently from regular values.
- Common functionality should be derived:
  - Fetching information from sensors.
  - Known relationships between data and context.
- Safety guarantees must be provided.

Example scenario

- We wish to present a sorted list of shops on a mobile device.
- Abstracting away common functionality:
  - Fetching location from sensors.
  - Sorting by proximity.
- Example in an ideal Haskell-like language.

Example scenario

Application code:

```
nearestShops = sortC location allShops

main = loop (realize (print (take 10 nearestShops)))
```

Example scenario

Application code:

```
nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops

main = loop (realize (print (take 10 nearestShops)))
```

Example scenario

Application code:

```
nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops

main = loop (realize (print (take 10 nearestShops)))
```

Example scenario

Application code:

```
nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops

main = loop (realize (print (take 10 nearestShops)))

-- note that
take  :: Int → [a] → [a]

take 10 nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
```

Example scenario

Application code:

```
nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops

main = loop (realize (print (take 10 nearestShops)))
```

Example scenario

Application code:

```
nearestShops :: [Shop] :↓ { User ▷ IsLocatedAt }
nearestShops = sortC location allShops

main = loop (realize (print (take 10 nearestShops)))

-- main is equivalent to:
main = loop do
  loc ← fetchLocation
  User ▶ IsLocatedAt := loc
  print (take 10 nearestShops)

-- custom do-notation provides type safety
```

Outline: Context-awareness in a declarative language

We address two main aspects of context-awareness:

- Defining computations that depend on context values.
- Managing a global knowledge base of context.

Context-dependent values

- Context-dependent values are (isomorphic to) functions:
  $\texttt{a} :\downarrow \texttt{c} \cong \texttt{c} \rightarrow \texttt{a}$
- We can apply them to a context to get their value.
- The challenge is then to make these behave as values.
  - So we can apply regular functions to them.

Making context-dependent values first-class

- We can start by using the applicative functor idiom:

$$\texttt{pure} :: \texttt{a} \rightarrow \texttt{a} :\downarrow \texttt{c}$$
$$\texttt{pure x} = \lambda \texttt{c. x}$$

$$(\circledast) :: (\texttt{a} \rightarrow \texttt{b}) :\downarrow \texttt{c} \rightarrow \texttt{a} :\downarrow \texttt{c} \rightarrow \texttt{b} :\downarrow \texttt{c}$$
$$\texttt{cf} \circledast \texttt{cx} = \lambda \texttt{c. (cf c) (cx c)}$$

$$[\![\texttt{f } x_1 \; x_2 \; \ldots \; x_n]\!] = \texttt{pure f} \circledast x_1 \circledast x_2 \circledast \ldots \circledast x_n$$

Making context-dependent values first-class

- The type of $\circledast$ in the previous slide is arguably too restrictive.

  $(\circledast) :: (a \to b) :\downarrow c \to a :\downarrow c \to b :\downarrow c$

- It only works for a single context type!
- It would be better if we could combine different context types.
- As a first approach, consider:

  $(\circledast_\times) :: (a \to b) :\downarrow c_1 \to a :\downarrow c_2 \to b :\downarrow (c_1 \times c_2)$

  $af \circledast_\times ax = (af \circ fst) \circledast (ax \circ snd)$

- This scheme works, but produces duplicates.

Making context-dependent values first-class II

- We need to define strongly typed heterogenous sets.
- We define $\pi_{\subseteq}$ as a way to extract subsets of context:

  $\pi_{\subseteq}$ :: ($c_1 \subseteq c_2$) $\Rightarrow$ $c_2 \rightarrow c_1$

- We also define set union at the type level.
- We can now define a much more flexible operator:

  ($\circledast_{\cup}$) :: (a $\rightarrow$ b) :↓ c_1 $\rightarrow$ a :↓ c_2 $\rightarrow$ b :↓ ($c_1 \cup c_2$)

  af $\circledast_{\cup}$ ax = (af $\circ$ $\pi_{\subseteq}$) $\circledast$ (ax $\circ$ $\pi_{\subseteq}$)

- This solves the problem.
  - Collects all dependencies in the arguments in a minimal set.
  - Allows us to apply regular functions to context-dependent values.

Managing the knowledge base

- We now turn our attention to the knowledge base problem.
- It should interact well with the abstractions so far.
- The typing information allows for safety guarantees.
- We use a parameterised state monad for this.

Managing the knowledge base II: Parameterised monads

- We can extend $:\downarrow$ to keep track of eventual produced context:

  $\texttt{CR}(c_1, c_2, a) \cong c_1 \to (a \times c_2)$

- In our case:
  - c_1 represents the required context set.
  - c_2 represents the resulting context set.
- This forms a (well-known) parameterised monad:

```
return :: a → CR(c, c, a)
(>>=) :: CR(c1, c2, a) → (a → CR(c2, c3, b))
       → CR(c1, c3, b)
```

- We can use custom do-notation as seen in the example.

Managing the knowledge base III: Parameterised monads

- Execution of these computations requires the empty context:
  `runCR :: CR(∅, cr, a) → (a, cr)`
- When we add knowledge to the KB, we keep track of it:
  `pushC :: c → CR(cs, {c} ∪ cs, ())`
- Injection from :↓ to CR enforces existence of context:
  `inContext :: a :↓ cs → CR(cs, cs, a)`

Managing the knowledge base IV: Realisable

- The $:\downarrow$ type contains the required context information.
- If we know how to fetch the required contextual set:

  realize :: (Realizable c) $\Rightarrow$ a $:\downarrow$ c $\rightarrow$ CR($\emptyset$, c, a)

In summary

- Standard functions applied over contextual values, seamlessly.
- Contextual dependencies accumulated in the type.
  - Using the modified applicative idiom: `pure` and $\circledast_{\cup}$.
- Information added to the KB manually is tracked in the type.
- `realize` automatically derives safe retrieval code.
- Safety is enforced.
  - Using the `CR` parameterised state monad.

Future Work

- A translation from the ideal language to the EDSL (completed).
- Richer variants of the retrieval-usage loop (FRP?).
- Dealing with historical data.

Conclusion

- Embedding of context-awareness semantics.
- Easier and safer to develop context-aware applications.
- Can hopefully make FP more attractive to developers.

Thank you for listening. Questions?