The Environment as an Argument
Context-Aware Functional Programming

Pedro M. Martins⋆, Julie A. McCann, and Susan Eisenbach

Imperial College London,
{pm1108,jamm,susan}@doc.ic.ac.uk

Abstract. Context-awareness as defined in the setting of Ubiquitous Computing [3] is all about expressing the dependency of a specific computation upon some implicit piece of information. The manipulation and expression of such dependencies may thus be neatly encapsulated in a language where computations are first-class values. Perhaps surprisingly however, context-aware programming has not been explored in a functional setting, where first-class computations and higher-order functions are commonplace. In this paper we present an embedded domain-specific language (EDSL) for constructing context-aware applications in the functional programming language Haskell.

1 Introduction

With widespread availability of mobile computing devices such as mobile phones and tablets, practical implementations of context-aware applications have started to appear. However, we observe a divide between the solutions proposed by researchers and the practical solutions adopted by implementers. We believe that this is because the former solutions are too heavyweight and rigid, and force developers to sacrifice some freedom in designing their applications, for little practical gain. As a result, practical implementations are typically based on bespoke implementations of context-aware behaviour. This prevents reusability of behaviour, and makes it easier for subtle bugs and programming errors to be repeated throughout implementations of the same behaviour. It has been argued that this is an inevitable consequence of context-awareness. Indeed, Lieberman and Selker [8] present a simple model for context-awareness and postulate that due to the dynamic nature of context-aware applications, it is hard to specify a module's behaviour in a way that will allow it to be reused at all.

In this paper we show that through a deeper embedding of context-awareness semantics into a programming language, we are able to specify this behaviour and provide natural programming language constructs for it. In addition to this, by being aware of the semantics of context-awareness, the compiler for our language is able to verify statically whether a certain number of properties that we believe should be true for this type of behaviour actually hold. This allows us to reuse

context-aware behaviour in a controlled and automatically validated way, with minimal loss of expressivity.

Our contributions are as follows:

– A composable representation of context-aware computations that *automatically* derives the context dependencies needed at the type level (Section 3.1).
– An abstraction for knowledge bases which does not enforce any representation or reasoning procedure upon the knowledge base, over which we define all of our abstractions (Section 3.3).
– A *parameterized monad* [1] that encapsulates adding context to a knowledge base, and statically verifies whether the required context information will be available at the call site of one of the previous context-aware computations (Section 3.5).
– A Haskell library that captures all of these abstractions in an *embedded domain specific language* (EDSL) (Section 3.6).

2 An Example Application

We present a simple implementation example of the declarative data-driven coding style for context-aware applications that we advocate in this paper. The syntax for the example is that of a pure declarative context-aware language with Haskell-like syntax, resembling the final syntax of our EDSL. Our simple scenario is one where a user is walking home from work and wishes to pick up something to eat on the way. The user does not want the food to get cold by the time they reach their home, so they wish to know where the nearest shops to their current location are, and how far each of these shops are from their home. Code listings 1.1 and 1.2 implement the main features needed for this functionality, namely a sorted list of shops and a routine that shows the user how close the nearest shop is from home. This example shows the definition of the domain of contextual information the application is going to manipulate, the relevant data types and the context-aware computation that is intrinsic in the given specification.

We begin by defining the domain of interesting contextual information for the application. Individuals are the entities of the domain that we are concerned with, in this case the user (1). Features are the properties of the individuals that we wish to inspect and manipulate, in this case where the user and their home are located at (3). The syntax i $\triangleright$ f is a type-level representation for feature f of individual i. We then define the normal data types that we will be manipulating in the application, namely shops (5). The connection between normal data and the contextual domain is provided in this case through a relevance relation. It states that locations are more relevant to the user the closer they are to them (10). We assume a data type `Location` is provided by some language library. Using the relevance relation, we sort a list of shops by contextual information, using the primitive `sortC`. In this case we are sorting the list of shops by their location field, using the applicable relevance relation with context (16). This

```
1    individual User
2
3    feature IsLocatedAt :: Location
4
5    data Shop = Shop { name :: String, location :: Location }
6
7    allShops :: [Shop]
8    allShops = ...
9
10   relevant Location (User ▷ IsLocatedAt) by distance
11
12   distance :: Location → Location → Double
13   distance = ...
14
15   nearestShops :: [Shop] ↓ { User ▷ IsLocatedAt }
16   nearestShops = sortC location allShops
17
18   main = loop do
19     loc ← fetchLocation
20     User ▶ IsLocatedAt := loc
21     print (take 10 nearestShops)
```

Listing 1.1. An application example.

creates a computation that is context dependent, nearestShops. Its type reflects the contextual dependencies that have to be satisfied in order for its value to be computed. The type a ↓ c represents a value of type a, with contextual dependencies c, where c is a set of context types.

In order to execute this computation we need to provide it with context. The do keyword, similarly to Haskell, allows us to enter a sequential execution context. In this case the keyword will also provide a global knowledge base for storing and retrieving context. Usage of the knowledge base will be tracked and validated to ensure that contextual dependencies have been satisfied appropriately before context dependent values are used. In the main loop of the application, we first fetch a location from the device's GPS (19) and add it to the knowledge base with the primitive expression $i \blacktriangleright f := v$, which allows us to assign the value of a feature f for the individual i as having value v. In this case, we are assigning the IsLocatedAt feature for the individual User as the location we have just retrieved (20). We then print the ten most relevant shops to the screen. The usage of context in nearestShops is statically verified by the compiler. Indeed, if we remove the line adding context to the knowledge base, we will get a compiler error specifying that the context of the type we removed is not available at the call site of nearestShops.

One of the main driving goals mentioned in the introduction was composability and code reuse. In that vein, we should be able to use our context dependent list in the same way that we would use a regular list. In the final line of the

```
1   individual Home
2
3   distanceFromHome ::
4     Location → Double ↓ { Home ▷ IsLocatedAt }
5   distanceFromHome loc = distance loc (π (Home ▷ IsLocatedAt))
6
7   nearestShopDistanceFromHome ::
8     Double ↓ { User ▷ IsLocatedAt, Home ▷ IsLocatedAt }
9   nearestShopDistanceFromHome = distanceFromHome (head nearestShops)
10
11  exampleHomeDistance = loop do
12    loc ← fetchLocation
13    hloc ← askUserForHomeLocation
14    User ▶ IsLocatedAt := loc
15    Home ▶ IsLocatedAt := hloc
16    print nearestShopDistanceFromHome
```

Listing 1.2. Merging contextual information.

example, we use the standard library function `take` on the list of shops. This function is completely independent from the context library, and has the type:

`take :: Int → [a] → [a]`

We can use this function for both regular lists and context dependent lists. The application of this function to the sorted shops list will however push the contextual dependencies to the type of the return value:

`take 2 nearestShops :: [Shop] ↓ { User ▷ IsLocatedAt }`

The example so far shows that context-aware values are first-class and can interact naturally with standard library functions. Moreover, if we were to use two contextual values in a single expression, such as a value depending on the home location and another depending on the user location, those two context dependencies would be merged appropriately. This will be seen in the next example. The primitive π is provided by the library, and allows us to manually project context from the knowledge base by type. We have used it in listing 1.2 to calculate the distance to the user's home of the closest shop to them. Note how the type of `nearestShopDistanceFromHome` (7-8) reflects the contextual dependencies that we are required to satisfy, namely, `User ▷ IsLocatedAt`, coming from `nearestShops` and `Home ▷ IsLocatedAt` coming from `distanceFromHome`. The application semantics of this language collect the contextual dependencies we use, in the type of the resulting value. This allows us to validate the state of the global knowledge base. In `exampleHomeDistance`, if we removed either line 14 or 15, we would no longer be adding necessary context to the knowledge base, and we would get a compile time error. This shows the basic behaviour that our EDSL provides. The next sections describe our implementation, along with the compromises that we had to take to conform to the host language.

3 A DSL for Context-Aware Programming

The application example in section 2 shows that there are two main facets to context-awareness. Firstly, defining computations that depend on implicit values, without breaking composability and type safety. Secondly, managing a global knowledge base of context, that can be accessed to provide context to the previous computations. We approach the former in sections 3.1 through 3.3 and the latter in section 3.4. All of the following definitions are written in Haskell, with liberal use of extensions provided by its flagship compiler GHC.

3.1 Context-aware Computations

We start by representing context-aware computations as pure functions from a contextual value to the desired output. Hinting at the fact that this input is implicit, we define a new type for these functions, which is isomorphic to the basic Haskell arrow type:

```
newtype ContextF c a = ContextF {runContextF :: c → a}
  deriving (Functor, Applicative, Monad)
type a :↓ c = ContextF c a
```

Semantically, $:\downarrow$ declares that a function's argument is contextual and should be considered implicit. `runContextF` then allows us to take this context-aware value and apply it to a context to return a pure value. However, context-aware values differ from regular functions in that we want to think about them as having the type of the return value. Indeed, when applying regular functions to these values, the argument of the context-aware value should be treated as implicit and become the implicit argument of the final value returned by the application. This effect can be achieved thus:

```
apply :: (a → b) → a :↓ c → b :↓ c
apply f ca = ContextF (λc → f (runContextF ca c))
```

This definition is that of `fmap` for the `Reader` functor. Extending this behaviour to accepting multiple arguments in a curried manner leads to the definition of ⊛ from the `Applicative` instance of `Reader` [9]:

```
(⊛) :: (a → b) :↓ c → a :↓ c → b :↓ c
ff ⊛ fa = ContextF (λc → (ff `runContextF` c)
                          (fa `runContextF` c))
```

However, this abstraction is exceedingly restrictive in the type of context it is able to deal with, as it forces `c` to be constant. In our case, this would require the definition of a "universe" product type for context types, which is impractical. We would like the product type to be *automatically derived* as we use more and more contexts. Effectively, what we want is to *parameterise* the applicative functor so that it is able to manage context dependencies appropriately. In this vein, let us define a new operator $⊛_\times$ which combines the contextual dependencies of both the function and the argument in a product type:

```
(⊛ₓ) :: (a → b) :↓ c₁ → a :↓ c₂ → b :↓ (c₁ × c₂)
```

This is the operator we need to implement the application semantics we outlined in section 2. In the next paragraphs, we will describe its implementation.

3.2 Application over Context-Aware Values

For a constant type `c`, the existing `Applicative` instance for `Reader` would be enough to achieve the behaviour we want. To see how we might generalise this approach to define $\circledast_\times$, let us specialize the type of $\circledast$:

```
(⊛) :: (a → b) :↓ (c₁ × c₂) → a :↓ (c₁ × c₂) → b :↓ (c₁ × c₂)
```

It seems that the only thing that we need to do to unify this type with that proposed for $\circledast_\times$ is to provide functions that generate this "universe" type. All we need to do is to precompose both functions with an appropriate projection function; of type $c_1 \times c_2 \to c_1$ for the first one and $c_1 \times c_2 \to c_2$ for the second one. In this way, the type of a composite computation can emerge from its components in a canonical way. In order for this scheme to apply to n-ary functions, however, we need to be able to represent and handle cartesian products effortlessly in Haskell. We will use the HList library as presented by Kiselyov et al [6], which represents type-level lists as iterated products with a fixed structure, and provides utility functions and error handling. We use an extended version to obtain set semantics and operations. Other than the typical set operations we will use the `hProject` function, which allows us to retrieve subsets of context:

```
hProject :: (c₁ ⊆ c₂) ⇒ c₂ → c₁
```

In all other cases we will use regular set notation in the code listings and refer the reader to our online implementation for details [1]. We can thus rely on precomposition with `hProject` to derive the universe type that we referred to previously. Then, we can just use the classic applicative instance for $((\to)\ c)$, for all c, and we get the desired functionality. We can therefore generalize to get the $\circledast_\times$ operator:

```
(⊛ₓ) :: (a → b) :↓ c₁ → a :↓ c₂ → b :↓ (c₁ ∪ c₂)
af ⊛ₓ ax = ContextF ((runContextF af) . hProject) ⊛
               ContextF ((runContextF ax) . hProject)
```

This definition of $\circledast_\times$ has a more general principal type than the one we originally discussed, and generalizes to n-ary functions. We can also present a mapping between the "application" of an n-ary function to context-aware values and our combinators. Note that $<\$>$ is just infix `fmap`:

```
⟦ f x1 x2 .. xn ⟧ = f <$> x1 ⊛ₓ x2 ⊛ₓ ... ⊛ₓ xn
```

```
evalC :: (c₁ ⊆ c₂) ⇒ a :↓ c₁ → c₂ → a
evalC ca k = ca `runContextF` hProject k
```

```
mkC1 :: (c → a) → a :↓ { c }
mkC1 f = ContextF (f . hHead)
```

```
mkC :: (c → a :↓ cs) → a :↓ (cs ∪ { c })
mkC = comb . mkC1
```

---
[1] Available at `http://www.doc.ic.ac.uk/~pm1108/hcontext`

```
where comb :: (a :↓ c₁ :↓ c₂) → (a :↓ (c₁ ∪ c₂))
      comb cca = ContextF $ λk → (cca 'evalC' k) 'evalC' k
```

`evalC` allows us to evaluate a context dependent computation by providing it with the necessary context (or a superset). `mkC` and `mkC1` allow us to build context-aware computations. `mkC1` will have to be used when the return value of the function is not context dependent.

3.3 Abstract knowledge bases

We now turn to the issue of context representation. The abstractions that we have created clearly define semantics for context-aware values and ways to meaningfully combine them. However, we have not yet modelled access to context providers. In the sections that follow we assume that there is a language which is able to describe the full spectrum of context information that we might need. For the purposes of this paper we assume that all context information that we retrieve is encoded in the same language. Moreover, we will assume that all context providers will use the same ontology when describing concepts. This is a very strong assumption, however solving this issue is not the focus of this paper, and constitutes its own field of research [10]. To detach the current presentation from the previous semantics, we use a different syntax for `HProject`, $k :\triangleright c$, which is to be interpreted as a constraint that holds when we have a knowledge base of type `k` from which we can extract context information of type `c`, a set of context types. We also take this opportunity to add additional structure to our context information. We provide support for individuals and features through the following type:

```
type family FeatureType a :: ∗
data Feat a = a := (FeatureType a)
```

We then represent individuals as data types, and assign features to them with a new data type. The type family `FeatureType` allows us to embed the type system of features into that of Haskell. This is coupled with an arbitrary projection function, whose arguments serve solely as witnesses for the types corresponding to the individual/feature pair desired:

```
data individual ▷ feature = individual ▷ (Feat feature)
π :: a → f → FeatureType f :↓ { a ▷ f }
π _ _ = mkC1 $ λ(_ ▷ (_ := v)) → v
```

With these definitions, we have now implemented everything needed to produce the context-aware value `nearestShopDistanceFromHome`, we discussed in section 2:

```
data User = User
data Home = Home
data IsLocatedAt = IsLocatedAt
type instance FeatureType IsLocatedAt = Location

distanceFromHome loc = distance loc <$> (π Home IsLocatedAt)
nearestShopDistanceFromHome =
  distanceFromHome <$> (location . head <$> nearestShops)
```

In order to implement the example in section 2, the only feature missing in our context representation is a notion of relevance of a piece of data for a user, given a set of contextual information. Relevance is realised as a predicate, stating whether a contextual value is relevant to the sorting of another non-contextual value. We define a restriction of this notion in order to aid the type checker, where we constrain the relation $\mathcal{R}(c, k)$, to instead be a *function*. This is represented as the associated type $\mathcal{R}$, which behaves as a type function, assigning a relevant context type to a regular type:

```
class Relevant a where
  type R a :: *
  relevance :: a → R a → Double
```

The `Location` example in section 2 would become:

```
instance Relevant Location where
  type R Location = User ▷ IsLocatedAt
  relevance l1 (User ▷ (IsLocatedAt := l2)) = distance l1 l2
```

An example of this in action is the `sortC` function we introduced in section 2:

```
sortC :: (Relevant c) ⇒ (a → c) → [a] → [a] :↓ { R c }
sortC contextfn xs =
  let sortfn c x y = compare (relevance (contextfn x) c)
                             (relevance (contextfn y) c)
  in  ContextF (λc → sortBy (sortfn . hOccurs $ c) xs)
```

3.4   Managing a global knowledge base

Our abstractions allow us to model context-aware computations and sources in a programming language. In order to make context truly implicit we would like to represent context as a shared knowledge base, that is populated by retrieving information from context sources and queried by context-aware computations. We should also be able to exploit all the typing information that we have been managing to make sure that this interaction is well-formed. It turns out that all of this is possible, using the formalism of parameterised monads. [1] First, we combine a context-aware computation and a contextual information producer into one single abstraction, that of stateful computations, which is a straightforward parameterisation of the `State` functor available in the Haskell libraries. By using the parameterised monad corresponding to this functor [1], we keep track of which knowledge is in the knowledge base at the type level. The approach of using parameterized monads to provide static guarantees over a DSL has been used before. Sackman and Eisenbach[11] show how to provide security guarantees for an imperative language embedded in Haskell. In Haskell, parameterised monads can be defined as a minor generalisation of the `Monad` type class:

```
class PMonad m where
  return :: a → m c c a
  (>>=) :: m c₁ c₂ a → (a → m c₂ c₃ b) → m c₁ c₃ b
```

GHC's support for rebindable syntax allows us to recover `do` notation for parameterized monads. Qualified importing of libraries may be used where traditional

monadic behaviour is desired. The types for the parameterised context monad (and monad transformer) then become:

```
newtype ContextRuntime c₁ c₂ a =
  CR { runContextRuntime :: c₁ → (a, c₂) }
newtype ContextRuntimeT m c₁ c₂ a =
  CRT { runContextRuntimeT :: c₁ → m (a, c₂) }

liftCRT :: Monad m ⇒ m a → ContextRuntimeT m c c a
```

We omit the PMonad instances and transformer combinators as they are essentially the same as the ones provided by the regular state monad. Note that our parameterised "monad transformer" is not a fully general parameterised monad transformer as it only works for non-parameterised monads. However, this is enough for the purpose of interacting with most monads present in the Haskell libraries. We then need to define an injection from the parameterised applicative functor to the monad:

```
inContext :: (k :▷ cs) ⇒ ContextF cs a → ContextRuntime k k a
inContext cf = CR $ λk → (evalC cf k, k)
```

We must also provide combinators to add to and update the knowledge base, all whilst performing the required type-level updates. We define a function that operates on type-indexed products, which updates a value by type if it is in the product, and appends it otherwise, called `hUpdateAtTypeOrAppend` (the definition is ommitted for space reasons). Using this, updating a context value in the knowledge base simply becomes:

```
(▶) :: HUpdateAtTypeOrAppend (i ▷ f) c₁ c₂
    ⇒ i → Feat f → ContextRuntime c₁ c₂ ()
individual ▶ feat = CR $
  λc' → ((), hUpdateAtTypeOrAppend (individual ▷ feat) c')
```

We may now add context values to the knowledge base represented by an HList. Note that because of the constraints in the type of `inContext`, we can only use an injected function if the required contextual information is present in the knowledge base. The final step we must take before executing context-aware computations in this monad is enforcing an empty starting context. Thus, we now define a set of execution functions for the parameterised monad that enforce this restriction. These were inspired by the ones provided for the `State` monad in the Haskell standard library.

```
runCR :: ContextRuntime HNil k a → (a, k)
runCR ca = runContextRuntime ca hNil
```

`evalCR` and `execCR` are defined as the appropriate projections from the result of `runCR`. We also define `evalCRT`, `execCRT` and `runCRT` as the transformer versions of these combinators. Thus, the only way to run a context-aware computation is to start with the empty context. The compiler may track all context dependencies, and abort with a compile-time error if they are not satisfied. This characteristic is arguably one of the most interesting features of our EDSL, as we are able to reify into the type level the context dependencies of a particular computation, and thus statically guarantee that they will be fulfilled. This elim-

inates a whole class of potential bugs in context-aware applications, whereby the application attempts to use context when it is not stored in the knowledge base.

3.5 Automatically satisfying contextual dependencies

Given that our EDSL is targeting situations where the domain of contextual information can have a type system imposed on it, that uniquely identifies the type of contextual information, it is not too far-fetched to think of satisfying these implicit dependencies automatically. That is, we can use the mechanisms outlined in the previous sections to collect contextual dependencies on the main program, and we can also create a library that adds specific portions of contextual information to a global knowledge base by querying device-specific sensors. We can then tie both of these together automatically, through the type system.

To achieve this, we introduce a new type class, the instances of which specify which types of contextual information we can retrieve under the IO monad, for the device we are currently using.

```
pushC :: (Monad m) ⇒ c → ContextRuntimeT m HNil c ()
pushC c = CRT . const . M.return $ ((), c)

class Realizable c where
  realize :: a :↓ c → ContextRuntimeT IO HNil c a
  fetch :: IO c
  realize x = liftCRT fetch >>= pushC >> inContextT x
```

This allows us to completely hide context from the programmer who is using the EDSL. For example, if the programmer had a main loop and a function called in every iteration that could benefit from contextual information, this dependency could be added to the code for the function, and lifted to the top-level using the mechanisms the EDSL provides. We can then provide the necessary instances of Realizable for the device in question, and selectively import the ones corresponding to the retrieval technique we wish to use.

4 Evaluation

In order to test the expressive power of our EDSL we implemented two context-aware applications, showcasing both the abstraction capabilities provided by the library as well as the ease of interaction with existing code.

4.1 Presence Board

Implementing a presence board application that keeps track of all people that have checked into a certain context (e.g. a building), has become the canonical application for evaluating context-aware libraries. This application is interesting because the presence information can then be used for more exciting context-aware applications, as will be seen. We assume an existing instance of Realizable for Location and an online service that can be used to match a location with

the building that contains it, returning a circular area delimiting the range to be considered for that building/context:

```
locationToRange :: Location → IO (Location, Double)
```

The EDSL allows us to provide a reusable library for this device, fetching the contextual information under the IO monad. Through the realizable type class we ready this for easy use by the programmer of the final application. In our case, we simply supply an instance for Realizable, for presence information, in our own data type:

```
data User = ...
users :: [User]
fetchLocationForUser :: User → IO Location
fetchUsers :: IO [User]
newtype Presence x = Presence [(x, Bool)] deriving (Show, Eq)

instance Realizable Location where ...
instance Realizable (Presence User) where
  fetch :: IO (Presence User)
  fetch = do
    location ← fetch
    us ← fetchUsers
    ls ← mapM fetchLocationForUser us
    (l,d) ← locationToRange location
    return . Presence $ zip us (map ((<d) . distance l) ls)
```

With this we can define the application code easily:

```
displayPresence :: IO () :↓ { Presence User }
displayPresence = mkC1 $ λpresence → do -- ...
main = forever (realize displayPresence)
```

Which implements a simple presence board application. Note how the programmer writing the previous code did not need to worry about how to retrieve the presence information, as it was abstracted away into a library. Then, retrieving this contextual information from the point of view of the final presence board application is simply a matter of using it at the right type, and making it implicit, using the liftings.

4.2 Mailing List

In order to ascertain how easy it would be to add context-awareness to an existing application, we took one of the examples used by the context toolkit [4], a context-aware mailing list application. This application should forward emails to only those subscribers that are located in the specific context that the mailing list applies to, in our case, physically located in a building. We located a mailing list manager application implemented in Haskell, Mhailist, publicly available on the Hackage package database [12]. We then proceeded to implement this behaviour without using any EDSL for implicit information. At a high level this change corresponds to retrieving presence information for the mailing list subscribers and selectively forwarding emails depending on it.

```
...
(addressees, msg)   ← return $
    case action of
      SendToList  → (addresses, addHeader listIDHeader message)
...
main = do result ← runErrorT processMessage
...
```

The modification is fairly simple, we just have to pass in the presence information to the forwarding function, and calculate it in the main loop. However, this simple change implies adding an explicit argument at every call site of the forwarding function, all the way up to the main loop. This can result in fairly significant changes to the main program. Using the existent implicit arguments feature present in GHC, we are able to propagate this dependency in a more implicit way. However, we then need to satisfy these dependencies by name, and it would be rather hard to provide an EDSL that extracts from the implicit dependencies of a computation the exact fetching routine the program should undertake, as these are identified by name. Using types to identify implicit arguments however, we are able to do just that. We can, as before, propagate the implicit argument to the main loop in an easy way. Then, in order to satisfy the main loop's context requirements, we just need to call realize, and the Realizable type class will handle fetching the appropriate contextual information for the device and supplying it to the computation. We need to introduce the contextual dependency at the top level instead of using the lifting mechanisms presented, as otherwise we would have to fully desugar the do-notation and lift the binds. We also had to import the parameterized monad bind operator qualified as `PM.»=` to allow us to use both monadic semantics.

```
mkC1 $ λpresence → do
...
  (addressees, msg)   ← return $
      case action of
        SendToList  →
          ( filter ((isJust . flip lookup $ presence) addresses)
          , addHeader listIDHeader message)
...
main = evalCRT $ realize processMessage PM.>>= λpm →
          liftCRT $ do result ← runErrorT pm
...
```

5   Related Work

Existing work in context-awareness has focused on creating flexible context representations as well as design patterns for developing context-aware applications within traditional programming languages. Context Toolkit [4] is a Java based toolkit that defines an architecture for developing context-aware applications, and provides the programming support for it. The central component of the context toolkit is the widget. It is defined by attributes and callbacks. There

are several flaws with the widget abstraction, that are addressed with special components in the toolkit. Firstly, widgets appear to segment context information independently from the chosen context representation. This is accounted for with context servers that both aggregate contextual information and can choose an underlying widget depending on the request. In our representation, widgets would be an artificial abstraction. The typing information allows an application to precisely specify, at compile-time, what sort of information it is going to require. This allows us to define a universal context runtime that will produce widgets "on demand". The context runtime serves as a flexible universal context server. As pointed by Bardram [2], the context toolkit enforces a highly distributed structure for a context-aware application. This aids flexibility and allows for distribution of architectural components. However, it is also more demanding of the system where it is deployed. Through using a more lightweight solution, we are able to support a less distributed solution if required. Because of the data-driven approach that we take, we can exploit existing communication libraries if we need to distribute components. This allows the user to pick the communication protocol and representation freely.

There has also been prior research done in modelling implicit arguments in a functional programming language, most notably that of Lewis et al [7], which is implemented in the Haskell compiler GHC as an extension. Our approach shares certain characteristics with this calculus, such as the implicit "floating out" of implicit arguments in composite computations. Our approach was designed from the ground up to be customised to the typical use cases in context-awareness, and that is reflected in our choice of identifying variables with their types, as there should only be one value of each type in the knowledge base. This allows us to make queries to the knowledge base more automatic, as only the typing information is required. In Lewis et al's solution [7] all implicit arguments have a name that identifies them, and it is up to the programmer to manage assignment of values to names and scoping of those names. In our approach, types identify implicit arguments, so no manual management of names is needed. The flexibility lost lies in the fact that we cannot have two values of the same type, which their calculus allows, but in our case is not necessary, as we have specified a type system that distinguishes all individual contextual data by type. This constraint however, allows us to extract more typing information statically and be able to manage the interaction between context sources and context consumers automatically. Also, it is possible to have multiple values of isomorphic types, and use the more sophisticated plumbing mechanisms of relevance and feature projection to manage these. An example of this was given with the user and their home's location, having types that are isomorphic in the haskell EDSL, but can conceptually be thought of as equal.

Another common way to introduce implicit global semantics is to use aspect-oriented programming. We can think of contextual dependencies as cross-cutting concerns, whereby the behaviours that would be injected would be both projections from the global knowledge base and retrieval and storage of contextual information. Using aspects for this purpose would make it much harder for us

to provide safety guarantees in the knowledge base access. The manipulations performed by aspect-oriented programming are purely syntactical, and it is hard to work out which source code transformations are going to be applied to a piece of code without examining the whole application. For this same reason aspect-oriented programming is much more flexible. However, given that one of our main goals was to provide clearer semantics for context-awareness, the disadvantages of aspect-oriented programming would outweigh the advantages.

6   Future Work

We believe that the abstractions we presented are an interesting approach to modelling context-awareness and can indeed be used to develop practical applications that use context in more complex ways than we have seen to date. Our implementation in Haskell will hopefully encourage further experimentation with these abstractions in real-world scenarios, and serves as further proof that Haskell has become an extremely appropriate host language for DSLs even when the semantics are quite different from its. However, there are some quirks in the DSL that stem from the fact that our EDSL is being hosted in Haskell. For instance, the fact that creating a contextual value is not encapsulated in only one combinator, but is implemented as two separate functions `mkC1` and `mkC`. This is because we have to deal with non context-aware types and interact naturally with them. If non context-aware types were considered equal to types that are dependent on a null context, `mkC1` would be a special case of `mkC`. On the other hand, the fact that application of functions to context-aware values needs to be performed with special operators, makes this library slightly unnatural to use. Further, we have not provided abstractions for continuous retrieval of contextual information and modelling the retrieval-usage loop. We believe that we can use functional reactive programming [5] to manage context streams in a natural way. Thus, we believe it would be interesting to design a language from the ground up that is based around these concepts, as a purer exposition of these ideas, and maybe as a theory that can bring further insights into the nature of context-awareness and the interaction between context providers and consumers.

7   Conclusion

When integrating context into a system, programmers are presented with two options. To either conform to rigid frameworks or to build bespoke functions that represent contextual behaviour. The latter, though providing more freedom, is problematic in that it has been shown that these dynamical approaches limit the amount of reusability, and errors can be easily propagated where attempts to reuse are made.

This is the first work that aims to overcome these problems by presenting an abstraction whereby context is deeply embedded into the programming language. In doing so, we are able to show that static verification can be achieved;

limiting the propagation of undesirable behaviours. Representing context-aware computations as functions with implicit arguments and inference rules, we are able to provide a composable type-safe system that provides static guarantees of well-formedness for context-aware applications. We also formalise the concept of a knowledge base and by using the type information we collected we are able to automatically satisfy contextual dependencies.

As proof of concept we implement our constructs in Haskell. It proved to be a good choice for a host language as both its type system and syntax are fairly programmable and allowed us to provide an EDSL that presented significantly different semantics from those of vanilla Haskell.

In summary, our formal grounding for context-awareness, combined with its example implementation in Haskell, provides the abstractions to encourage the exploration of more complex context driven applications than have been seen to date.

References

1. R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335, June 2009.
2. J. Bardram. The Java Context Awareness Framework (JCAF)–a service infrastructure and programming framework for context-aware applications. *Pervasive Computing*, pages 98–115, 2005.
3. A. Dey and G. Abowd. Towards a better understanding of context and context-awareness. In *CHI 2000 workshop on the what, who, where, when, and how of context-awareness*, volume 4, pages 1–6. Citeseer, 2000.
4. A. Dey, G. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, 2001.
5. C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.
6. O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.
7. J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 108–118, New York, NY, USA, 2000. ACM.
8. H. Lieberman and T. Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3.4):617–632, 2000.
9. C. McBride and R. Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2007.
10. H. Pinto, A. Gómez-Pérez, and J. Martins. Some issues on ontology integration. In *IJCAI-99 workshop on ontologies and problem-solving methods (KRR5)*. Citeseer, 1999.
11. M. Sackman and S. Eisenbach. Safely Speaking in Tongues: Statically Checking Domain Specific Languages in Haskell. In *LDTA'09*, March 2009.
12. C. Sampson and L. Kotthoff. Mhailist: Haskell mailing list manager. `http://hackage.haskell.org/package/Mhailist-0.0`, April 2010.