Implementing an Object Oriented Design in Curry

Herbert Kuchen

University of Münster, Steinfurter Str. 109, D-48149 Münster, Germany
`kuchen@uni-muenster.de`,
WWW home page: `http://www.wi-uni-muenster.de/pi/`

Abstract. Declarative programming languages provide a higher level of abstraction and more powerful concepts than typical imperative and object oriented languages. On the other hand, the object oriented paradigm is superseding conventional approaches to software development like SA/SD and modular design. This is, among others, due to the better reusability and simpler maintenance of object oriented components. As a consequence, only languages which enable a simple implementation of object oriented designs will remain interesting for the development of significant software systems.
The present paper shows how an object oriented design can be implemented in the functional logic language Curry. This approach can easily be adapted to other declarative languages.

1 Motivation

Many programmers love declarative languages because of their elegance, powerful programming concepts, readability, and so on. It is impossible to recall all the advantages of declarative languages here. Let us just name of a few of them. Functional languages (like e.g. Haskell [HPW92]) offer e.g. higher-order functions, which enable the development of user-defined, application oriented control structures. Using a couple of well-chosen higher-order functions, complex applications can be handled by a few lines of code. Logic languages like Prolog [CM84] have a built-in search mechanism and allow hence an easy solution of search problems. Functional logic languages like Curry [HKM95,Ha98] combine the advantages of purely functional and purely logic languages. Moreover, purely declarative languages simplify proofs of program properties, like e.g. correctness, due to their lack of side effects.

This is all very nice, but declarative languages are rarely used for the development of significant software systems. The object oriented paradigm has taken over in the area of software engineering. This is due to a couple of reasons. Only a few of them can be listed here. In particular, object oriented languages support the reusability of components better than others. With inheritance and late binding, it is possible to add a special component (a subclass) to a more general one (the superclass) without having to change or even touch components which refer to objects of the superclass, even if the actual object happens to be of the subclass. This kind of re-usability is also available in declarative languages like Haskell which support type classes. However, a second form of re-usability is only available in object oriented languages. Namely, subclasses may inherit code from superclasses. Together with the possibility of a subclass to override some of the methods of its superclass, this allows to re-use software components even if (as often) they do not fit completely. The (hopefully few) conflicting methods are just overridden by adapted ones. So-called frameworks, i.e. adaptable systems of classes (e.g. for graphical user interfaces) demonstrate the strength w.r.t. reusability in an impressive way. Moreover, the possibility to encapsulate data structures and the operations working on them is more directly supported by object oriented languages then by declarative languages, which focus on a functional (rather than a data oriented) decomposition of the software system. The recent agreement on a unified notation, the so-called unified modeling language (UML) [BRJ99], for object oriented analysis and design has strengthened the position of the object oriented paradigm. Finally, a large set of so-called design patterns [G+95], has been developed for object oriented software development. Such a pattern is a typical system of classes which has turned out to be useful for solving a class of similar design problems. The designer has to anticipate which aspects of the software system are likely to be changed in the future and to choose a corresponding pattern which ensures the required flexibility for this aspect. Thus, the focus of these patterns is on flexibility and easy adaptability of the system to changing requirements. All of these patterns make heavy use of object oriented concepts.

To summarize, declarative as well as object oriented software development both have their advantages, and it would be nice to combine them. From the point of view of the declarative programming community, this is not just nice but essential in order to keep these languages interesting in an environment where the software development process, at least as far as the analysis and design phases are concerned, tends to be based on the object oriented paradigm. If it is not easily possible to implement an object oriented design in a declarative language, people will prefer to use an (imperative) object oriented one and they will refrain from the nice features of declarative languages. In particular, it should be simple to start off from an UML model consisting of a few use-case diagrams, class diagrams, interaction diagrams, state charts, and so on and to implement such a system in a declarative language.

The present paper shows how object oriented programming concepts can be simulated in the functional logic language Curry. In particular, the following aspects will be addressed:

1. the simulation of state
2. the generation of a dynamic net of objects
3. inheritance

We will discuss these issues by means of an example application. We finish with a discussion of related work and a conclusion. In the sequel, we will assume some familiarity with Curry or Haskell as well as some basic knowledge of object oriented software development.

2 Running Example: Library

We will demonstrate our approach by means of a running example, namely the object oriented design of a library. The library maintains a set of books and loans them to clients. The core of this design is the class diagram depicted in Fig. 1. It shows the classes `library`, `book`, and `client` together with their attributes and methods. E.g. class `client` has an attribute `name` and offers the methods `LoanBook`, `ReturnBook`, and `GetName`. The edges of the class diagram denote associations between classes. The cardinalities of the associations indicate the number of neighbor objects to which each object is connected. E.g. a book object can be connected to 0 or 1 client objects (the loaner), while each client object may be connected to an arbitrary number of book objects (the loaned books). The association between book and client is the only one which is navigable in both directions. All other associations only allow navigation in one direction (indicated by an arrow) here. This example will later on be enhanced by inheritance.

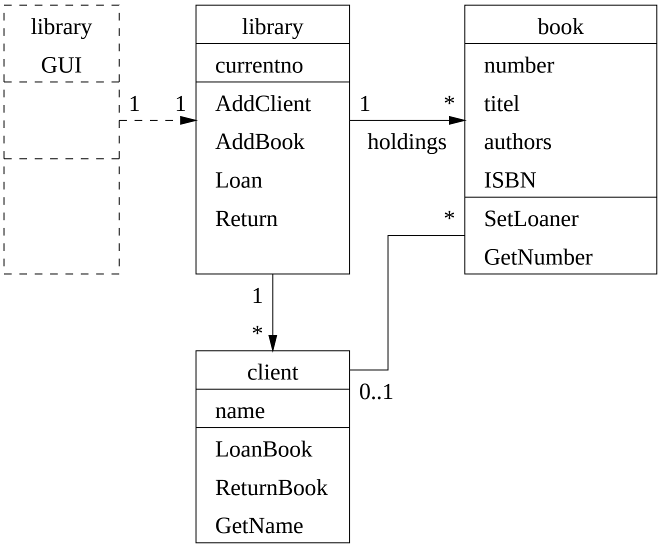


Fig. 1. class diagram for the library application

3 The Simulation of State

An important feature of an object is that it has a state. This is something that purely declarative languages cannot express directly. The usual way to simulate the state of a data structure in a declarative language is to generate a new data structure which corresponds to the state of the object after the change, e.g. when loaning a book to a client, the new state of the book could be:

$$\text{newbook} = \text{SetLoaner book client}$$

This idea is used in an object oriented extension of Haskell, named Haskell++ [HS95]. Apart from the fact that this simulation does not seem to be natural, it also does not work in the case of distributed or concurrent systems of objects. If one client computes newbook, only this client sees the "new state". Other clients keep referring to the "old state".

Another way of coping with state in a functional (logic) language is to represent an object by an expression. This expression is typically an application of a recursively defined function f , which consumes a stream of incoming messages and produces a stream of outgoing messages. Thus, the "objects" communicate via streams of messages [1]. The second argument of f is the state of the object, i.e. a tuple of its attributes. The corresponding tuple in the recursive call of f can be considered as the new state after processing one message. The following example illustrates the approach [2]:

```
data Message = SetLoaner OID | GetNumber Int | ...
type OID = Int
type BookState = (Int,String,String,String,OID)


book :: [Message] ->  BookState -> [(OID,Message)]
book (m:l) state@(no,title,authors,isbn,loaner) =
  msgs ++ book l newstate
  where (msgs, newstate) =
    case m of
      (SetLoaner oid) -> ([],(nr,title,authors,oid))
      (GetNumber result) | result =:= no  -> ([],state)
```

An application of the recursively defined function `book` to some stream of messages and some initial state represents a book object. For each method in the class diagram, there will be a corresponding kind of message. The body of `book` mainly consists of a case distinction depending on the kind of message to be processed. The processing of a message will often require some messages to be send to other objects. Moreover, it will cause the state to be changed. When processing a `GetNumber` message, an answer has to be transmitted back to the sender. In a purely functional setting, this will require an explicit result messages to be sent and processed. In a functional logic language, this can be simplified. The sender adds a logical variable to the message, and the receiver binds this variable to the computed result. The computation of the sender can continue until the result is actually needed, e.g. since it is used as an argument of a rigid function. If this happens, the sender will be suspended until the binding has been established. Note that each object only processes one message at a time.

[1] For simplicity of the presentation, we assume all streams to be infinite here.

[2] We use some Haskell features here which are not yet available in Curry.

Thus, if the computation of an object is suspended due to a missing answer no other messages will be processed either. In the above example, the solution of the constraint `result =:= nr` causes `result` to be bound to the number `no` of the considered book. If the result type is void, no logic variable for the result is needed.

Library and client objects will be analogously simulated by functions `library` and `client`, respectively.[3]

```
data Message = ... | Loan String Int | Return String Int |
               AddClient String |
               AddBook Int String String String |
               LoanBook OID | ReturnBook OID |
               GetName String | ...
type LibState = ([OID],[OID],Int)


library :: [Message] ->  LibState -> [(OID,Message)]
library (m:l) state@(clients,books,currentno) =
  msgs ++ library l newstate
  where (msgs, newstate) =
    case m of
      (Loan name bookno) | in bookOID books  &&
                           in clientOID clients ->
         ([(bookOID,GetNumber bookno),
           (clientOID,Getname name),
           (clientOID, LoanBook book)], state)
         where bookOID, clientOID free
      (Return name bookno) | in bookOID books  &&
                             in clientOID clients ->
         ([(bookOID,GetNumber bookno),
           (clientOID,GetName name),
           (clientOID, ReturnBook book)], state)
         where bookOID, clientOID free
      (AddClient name) -> ...  discussed later ...
      (AddBook no title authors isbn) ->  ...  discussed later ...


type ClientState = (String,[OID],OID)


client :: [Message] ->  ClientState -> [(OID,Message)]
client (m:l) state@(name,books,this) =
  msgs ++ client l newstate
  where (msgs, newstate) =
    case m of
```

[3] The auxiliary functions `delete` and `in` can be found in the appendix.

```
(GetName result) |  result =:= name  ->   ([], state)
(LoanBook book) ->
   ([(book, SetLoaner this)], (name, book:books, this))
(ReturnBook book) ->
   ([(book, SetLoaner 0)],
    (name, delete book books, this))
```

When `client` processes a message `GetName result`, the variable `result` is unified with the actual `name` of the client (by `result =:= name`) . If `result` was an unbound variable, the unification would bind it to the name stored in the state of the client. However here, the variable `result` is bound already, at least if the `GetName` message has been submitted in the body of `library`. There, the constraint `in clientOID clients` has been solved in order to find a clientOID, and the `GetName` message is sent to the corresponding client object. If the binding of `result` does not match the actual name of the receiving client object, the computation will fail and alternative solutions of the constraint will be tried. On the implementation level, this means that `GetName` messages will be send to client objects until one is found which has the desired name. The same technique is used to find a book object with the desired number.

Note that a client object refers to itself (`this`), and that this requires to maintain the own oid as part of the state.

A first attempt to connect our objects could work as follows[4]:

```
m1 = client  (mfilter 1 ms) ("Smith",[])
m2 = book    (mfilter 2 ms) (1,"Curry","N.N.","5-987-645-9",0)
m3 = library (mfilter 3 ms)  ([],[])
ms = merge m1 (merge m2 m3)
```

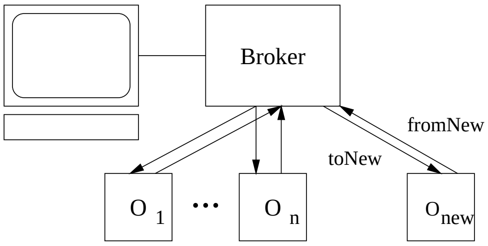
The output streams of all objects are non-deterministically merged and each object extracts the messages addressed to itself using `mfilter` and its own unique object identifier (oid).

A problem of this approach is that the resulting network of objects is completely static, while in a typical object oriented system objects are dynamically created and connected to each other. This issue will be discussed in the next section.

4 Dynamic Networks of Objects

In order to implement the dynamic creation and connection of objects, we connect the objects via a special so-called *broker* object rather than directly to each other.

[4] The auxiliary functions `mfilter` and `merge` can be found in the appendix.



Thus, the only object with a changing number of "neighbors" is the broker. In our example, it can be implemented as follows:

```
data Message = ... | NewClient String OID |
               NewBook Int String String String OID |
               NewLibrary OID | PRINT String | ...
type BrokerState = Int


broker::  [(OID,Message)] ->  BrokerState -> [(OID,Message)]
broker ((adr,m):l) lastOID = fromBroker
  where  fromBroker = msgs ++ broker toBroker newLastOID
    (msgs, newLastOID, toBroker) =
      case (adr,m) of
        (0,NewClient name resultOID) | resultOID =:= lastOID ->
           ([], lastOID+1, merge l fromNew)
           where fromNew = client toNew (name,[])
                 toNew   = mfilter lastOID fromBroker
        ...  other NewXX messages handled analogously ...
        (oid,mes) | oid > 0 -> ([(oid,mes)],lastOID,l)
        ...
```

Besides forwarding messages the broker is also responsible for the connection to the environment, i.e. (possibly monadic) input and output (omitted here for simplicity), and for creating new objects and connecting them to itself. If the broker e.g. receives a `NewClient` message (addressed to itself, i.e. where the oid is 0), it creates a new client object by calling `client` with the proper input stream and the initial state as arguments. The OID of the created object is returned to the sender as usual by binding `resultOID` to `lastOID`. Other requests to create a new object are handled analogously. All other messages are directly forwarded to the indicated receiver.

We are now in the position that we can explain how the function `library` processes `AddBook` and `AddClient` messages:

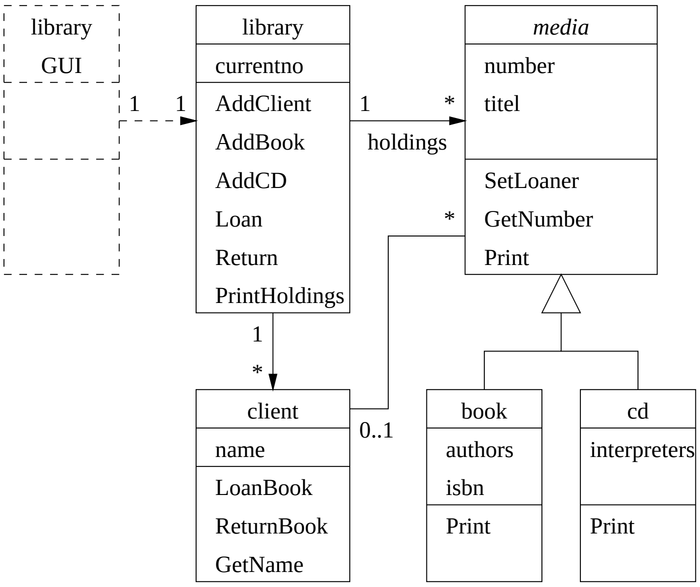
```
library (m:l) state@(clients,books,currentno) =
  msgs ++ library l newstate
  where (msgs, newstate) =
    case m of
      ... as before ...
      (AddClient name) ->
```

```
            ([(0,NewClient name resultOID],
             (resultOID:clients,books,currentno))
            where resultOID free
          (AddBook title authors isbn) ->
             ([(0,NewBook currentno title authors isbn resultOID],
              (clients,resultOID:books,currentno+1))
             where resultOID free
```

When the broker is asked to create the required objects, their OIDs are stored in the lists of books and clients, respectively.

5 Inheritance

An important aspect of the object oriented paradigm is that a class may inherit attributes and methods from a superclass. Let us extend our running example in order to introduce inheritance.



We replace class `book` by an abstract class `media`, and classes `book` and `cd` inherit e.g. `GetNumber` from it and override `Print`. According to late binding, the correct `Print` method should be used for each media, when printing the holdings, i.e. the list of all media.

Wadler and Blott [WB89] show how inheritance can be simulated by equipping each “object” with a dictionary telling which method corresponds to each message an object has to handle. This approach is used in Haskell++ [HS95]. However, in our case an other approach is simpler. Each recursive function which is used for the simulation of an object just cares about all messages the object might receive including those corresponding to methods inherited from the superclass. Since all OIDs have the same type, all the messages can in principle be sent to all objects. Of course, objects of different classes will in general process the same message differently. In the case of book and cd objects, a `Print` message will cause the different attributes to be printed.

```
book (m:l) state@(no,title,authors,isbn,loaner) =
  msgs ++ book l newstate
  where (msgs, newstate) =
    case m of
      ... SetLoaner and GetNumber as before ...
      Print -> ([(0,Print title++authors++isbn)], state)


cd (m:l) state@(no,title,interpreters,loaner) =
  msgs ++ cd l newstate
  where (msgs, newstate) =
    case m of
      (SetLoaner oid) -> ([],(nr,title,interpreters,oid))
      (GetNumber result) |  result =:= no  -> ([],state)
      Print  -> ([(0,Print title++interpreters)], state)


library (m:l) state@(clients,media,currentno) =
  msgs ++ library l newstate
  where (msgs, newstate) =
    case m of
      ... as before ...
      PrintHoldings -> ([(med,Print) | med <- media], state)
```

This approach has the advantage that it causes no runtime overhead. However, it may result in a code duplication, since the superclass and its subclasses all contain the same code to process the message. In our example, this is not serious, since every message is handled by one line of Curry. If the processing of a message is more complex, this could be done by an auxiliary function, e.g. `processGetNumber` in the library application. In a subclass like `book` we would then re-use this function. Of course, we have to restrict the state to the attributes of the superclass before and to extend it afterwards by the additional attributes of the subclass, i.e. mainly

```
extend.processGetNumber.restrict
```

6 Related Work

Some researchers have already felt the need to combine declarative programming languages and object orientation. Objective ML [RV98] uses the side effects of ML in order to simulate state. There are two object oriented extensions of the purely functional language Haskell, namely Haskell++ and O'Haskell, both from Chalmers. As mentioned already, Haskell++ [HS95] simulates state by producing new values. This approach is not applicable to concurrent and distributed systems. In O'Haskell [NC97], objects correspond to processes which communicate similarly to monadic IO.

The concurrent functional language Eden [B+98] has no direct relation to object orientation. However, it provides explicit processes (roughly: special expressions) which are connected via streams, and it is hence another obvious platform for implementing our approach. Moreover, the communication of objects via the broker can (at least in some cases) be simplified by using Eden's so-called dynamic reply channels. An alternative implementation of the broker could be based on ports [Ha99]. Here, the objects would communicate via difference lists.

7 Conclusions

We have shown how object oriented programming can be simulated in the purely functional logic language Curry. This approach can be easily adapted to other declarative languages. The main idea is to represent objects by expressions, namely calls to recursively defined functions which consume and produce a stream of messages and carry the state as an additional argument. In order to enable a dynamic network of objects, the objects are not connected directly to each other but via a special broker object, which is also responsible for the creation of new objects. The presented approach also simulates inheritance.

It is known that input/output via streams may lead to some strange order of inputs and outputs in a lazy language like Curry in case that some outputs happen not to depend on previously expected inputs and that the strictness of some functions has to be increased in order to avoid these problems [HPW92]. Let us point out that this problem does not occur in our framework, since every submitted message is really expected by the receiver. An alternative way of coping with the problem would be to use the standard transformation of stream IO to monadic IO. We prefer the communication via streams, since this does not give an imperative flavour to the system.

References

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide, Addison Wesley, 1999.

[B+98] S. Breitinger, R. Loogen, Y. Ortega-Mallén, R. Peña: Eden — Language Definition and Operational Semantics, Technical Report 96-10, Reihe Informatik, Philipps Universität Marburg 1998, http://www.mathematik.uni-marburg.de/inf/eden/

[CM84] W.F. Clocksin, C.S. Mellish: Programming in Prolog, Springer, 1984.

[G+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison Wesley, 1995.

[Ha98] M. Hanus (Ed.): Curry — An Integrated Functional Logic Language, http://www-i2.informatik.rwth-aachen.de/ hanus/curry/

[Ha99] M. Hanus: Distributed Programming in a Multi-Paradigm Declarative Language, Proceedings PPDP'99, LNCS 1702, 188-205, 1999.

[HKM95] M. Hanus, H. Kuchen, J.J. Moreno-Navarro: Curry: A Truly Functional Logic Language, ILPS'95 Workshop on Visions for the Future of Logic Programming, Portland (USA), pp. 95-107, 1995.

[HPW92] P. Hudak, S. Peyton Jones, P. Wadler (Eds.): Report on the Programming Language Haskell, SIGPLAN Notices, Vol. 27, No. 5, 1992.

[HS95] J. Hughes, J. Sparud: Haskell++: An Object-Oriented Extension of Haskell, Haskell Workshop, 1995, http://www.cs.chalmers.se/ sparud/

[NC97] J. Nordlander, M. Carlsson: A Rendezvous of Functions and Reactive Objects — Escaping the Evil I, Proceedings of Haskell Workshop, 1997, http://www.cs.chalmers.se/ nordland/ohaskell/

[RV98] D. Rémy, J. Vouillon: Objective ML: An Effective Object-Oriented Extension to ML. TAPOS 4(1): 27-50, 1998.

[WB89] P. Wadler and S. Blott: How to make ad hoc polymorphism less ad hoc. Proceedings of 16th ACM Symposium on Principles of Programming Languages, pages 60–76, 1989.

8 Appendix: Predefined Functions

```
mfilter :: OID -> [(OID,Message)] -> [Message]
mfilter i [] = []
mfilter i ((j,m):l) | i == j    =  m : mfilter i l
                    | otherwise =  mfilter i l


merge [a] -> [a] -> [a]
merge eval choice
merge [] l = l
merge l [] = l
merge (x:l1) l2 =  x : merge l1 l2
merge l1 (x:l2) =  x : merge l1 l2


delete :: a -> [a] -> [a]
delete x [] = []
delete x (y:l) | x == y    = l
               | otherwise = y : delete x l


in :: a -> [a] -> Success
in x (y:l) =  x =:= y  || in x l
```