Transforming Declarative Models Using Patterns in MDA

Srinivas Nedunuri and William Cook, Dept. of Computer Sciences, University of Texas at Austin, Austin, TX 78712

Abstract In this paper we propose to specify platform independent models using a functional language with a view to transforming them with architectural and design patterns by applying equational or algebraic reasoning. We illustrate our idea using a couple of known architectural and design patterns.

Introduction

The domain based approach to software development [CE00, MB02] views software development as the composing together of various *domain models*, and if necessary performing transformations on those models to arrive at an efficient implementation on some *target architecture (*itself possibly defined by a domain model*)*. In order to bring this idea a step closer to reality, the OMG[1] has recently proposed an initiative called *Model Driven Architecture* [MDA]. The idea behind MDA is to transform a *platform independent model* (PIM) to a *platform specific model* (PSM). The PIM is roughly a subset of a domain model [Fr03]. It is the model in which the software solution is most clearly and elegantly expressed. The PSM on the other hand is the “ugly but efficient” form that runs on a specific platform. The essential idea is to facilitate reuse by allowing the PIM to be re-targeted at a variety of different architectures and platforms.

Based on the experience of one of the authors (Nedunuri) in building an MDA tool for generating J2EE applications, we believe it is quite possible to generate quite a reasonable class of applications (for example CRUD applications) from PIMs by applying straightforward code templates, that may even incorporate some simple patterns such as Proxy [GHJV95]. However, as the complexity and sophistication of the generation grows, it becomes necessary to understand the transformations, how they interact, and how they may be modified, composed, and used to optimize models. This is going to be difficult without precise definitions of the models and the transformations. Providing such definitions is a challenging problem that the MDA community is only just beginning to tackle in earnest. We believe that *functional languages* provide not only the required precision (which could also be done in OO, see for example the work of the precise UML group [pUML]), but equally importantly, allow for *equational* or algebraic reasoning by virtue of referential transparency. Any functional program derived using equational reasoning is a valid working program and is, furthermore, guaranteed correct by construction.

In this paper, we look at transformations that effect design improvements. An example of the kind of design improvements we would like to make are captured by design and architectural patterns [GHJV95], [BMR+96]. In fact, as suggested in [TB95] we can view patterns themselves as transformations. Here we show how patterns can be viewed as transformations on declarative models (specified as functional programs) that can in principle be derived by equational reasoning.

Why Declarative Models?

It has been suggested that avoiding side effects makes it easier to both develop correct programs [Bl01] and subsequently comprehend them [DLO+03]. Unfortunately, it is not easy to completely avoid side effects in most programming languages such as Java, C++, etc and still have an idiomatic efficient program. A related approach to developing correct programs known as Design by Contract [Me97] is to specify the program declaratively using a contract, and then write the body of the specification imperatively. However, it is rare to find a model that is fully specified using contracts. Part of the problem, we believe, is that after writing a declarative contract, one still has to write the imperative model, which increases, not decreases, the work effort. Functional languages take another approach to this problem: A functional program can be considered as a specification that is both side effect free and executable. This gain can sometimes come at the cost of efficiency. Note, however, that in MDA, the idea is to develop the PIM with correctness, rather than efficiency, being the main criterion. Efficiency is addressed when transforming the PIM to a PSM. For this reason, we believe that functional language may be highly suitable for specifying the PIM. Functional languages also have a long history of research into program transformation [Da82], [BW89]. We hope to leverage off some of this work for transformation of the PIM to a PSM.

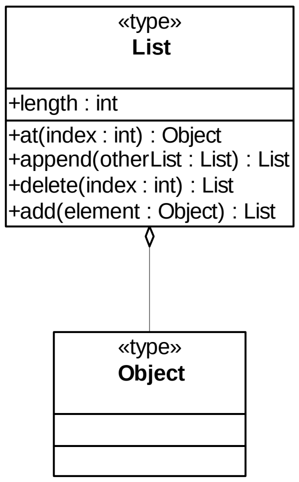
As a consequence of choosing a functional specification language, we can employ *algebraic* or *equational* transforms. That is, the expression (model) resulting from applying a transform is algebraically equal to the expression (model) to which it was applied. This very useful property enables us to view proofs as derivations and vice versa. That is, the resulting synthesized efficient model is *correct by construction*. Furthermore, because the derivations are also expressed in the same language, every intermediate stage in the derivation is also a valid program.

[1] The Object Management Group -- responsible for the UML, CORBA, and IIOP standards, amongst others

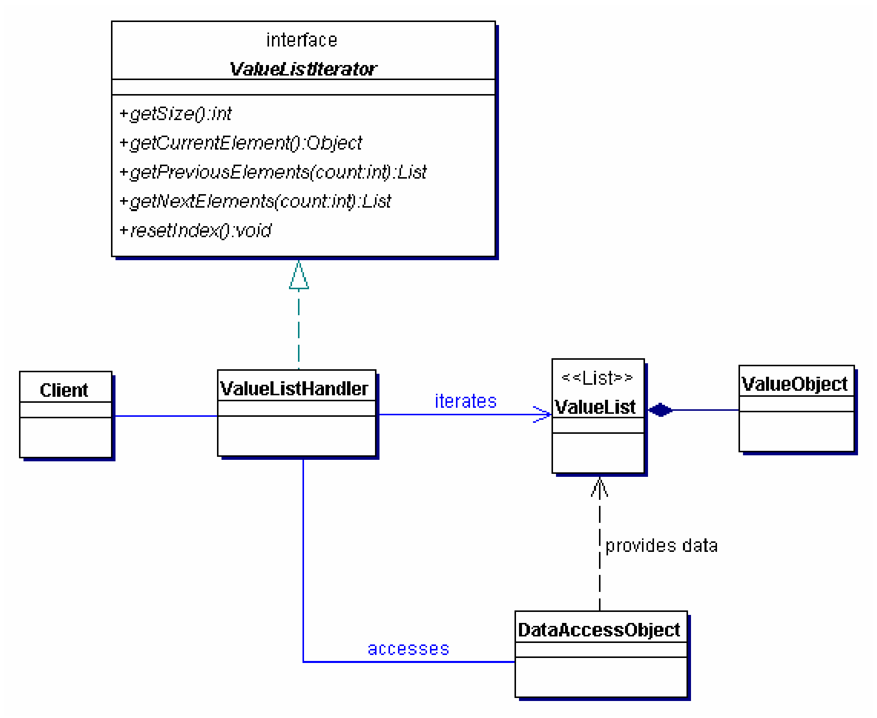
By choosing a specification language with powerful language capabilities such as higher order functions, lazy evaluation, type inference, pattern matching, etc. we hope to make the specification process itself more productive. Our first candidate specification language has been Haskell.

The Value List Handler J2EE Pattern

Consider a very simple model for lists.


«type»
List
+length : int
+at(index : int) : Object
+append(otherList : List) : List
+delete(index : int) : List
+add(element : Object) : List
«type»
Object

Now consider mapping (implementing) this to a distributed architecture, such as J2EE in which the list (perhaps a long directory of names or objects) needs to be displayed on the client. Making a round trip call to the server to access each element on demand is not acceptable from a performance standpoint. On the other hand, if the list is a very long one, pulling the entire list from the server may not work because the client may not have enough cache space. It is also wasteful in the (many) cases where the user simply wishes to look at the first few elements and then dismiss the page. To handle situations such as this, Sun has defined a number of J2EE design patterns, which have been captured in a recent book [ACM01]. One pattern addressing this problem is the *Value List Handler*. This pattern is illustrated by the generic model below:



The `ValueListHandler` (implementing the `ValueListIterator` interface) provides an iterator for iterating over the list. The list is a collection of `ValueObjects`. The `ValueObjects` are serializable transfer objects constructed from data supplied by the `DataAccessObjects` which interface with the database. The essence of this pattern, which is what we will focus on here, is that the list resides on the server, and is lazily fetched to the client where it is cached. One such lazy strategy is when a request is made for an element that is not in the cache, to fetch all the elements up to and including the desired element. The resulting program is designed to run efficiently on such an architecture. However, when defining the domain of lists, we want the modeler to write their solution in the clearest way possible. This means we have to transform the clear version into the optimized version. Note that although the pattern above is considered a J2EE pattern, there is nothing about it specific to the J2EE platform. It could apply equally well to other distributed architectures, such as CORBA, .Net, or indeed even a simple client server architecture. For this reason, we refer to the PIM that is optimized with such patterns as a *Design PIM*, rather than a PSM. The Design PIM is subsequently transformed to a PSM by applying platform specific (e.g. J2EE specific) optimizations.

A Declarative PIM

Consider the rough analog of the above models in a pure functional language. In the PIM, `List` becomes an abstract data type, defined in Haskell as a module that exports the required methods:

```
module List (length, at, append, delete, add) where
<module definition>
```

In what follows, we will interpret methods on an OO type as functions which take the type as the first argument[2]. Thus the method for accessing an element in the list, called "at" becomes the “at” function in Haskell (also known as the “!!” operator in the standard library), defined as follows:

```
at :: [t] -> Int -> t
at (x:xs) 0 = x
at (x:xs) (n+1) = at xs n
at [] n = ⊥
```

The first line is a type declaration that states that `at` is a function that takes a list of any type (designated by the type parameter `t`) and an `Int` and returns an element of the list type `t`. The type parameter plays a similar role to the type parameter in Java’s generics. The next 3 lines define the `at` function by case. The first case is where we want the first element in the list (position = 0). The correct value to return is simply the first element of the list (`x:xs` is one of several standard patterns in Haskell - in this case one that matches a list to decompose it into its head (the first element) and the tail (the rest of the list)). The second case is the recursive case where we want some element other than the 0th (indicated by the pattern `n+1`). Finally, the error case is where we are attempting to access an element of an empty list (denoted by the pattern `[]`). The returned value in this case is the nonterminating computation ⊥.

A Declarative Design Oriented PIM

What we want to do here is depicted in the following diagram:

```
      PIM
       |
       |
       v
 “design” PIM
```

There are a number of design improvements we can make to a List. For example, we may want to implement it as a mutable structure. Or we would like to incorporate the architectural pattern discussed earlier. In order to enable this, we will need to define the design version of the List ADT somewhat differently. The design definition will be based on monads. Monads are described in [Wa95]. They provide a way of conveniently wrapping housekeeping work, such as state update or error propagation, so it does not clutter up regular code. For our purposes, the `List` monad could be defined as follows

```
data List t u = List(ListHandle t -> (u, ListHandle t))
```

A ListHandle is an opaque reference to a list structure, perhaps on the server. It is not necessary to expose it to clients because the List monad datatype above allows us to define a number of access functions or *actions* (of type

[2] A more OO like solution is possible in one of the functional OO languages such as O’Haskell [No99], but we do not pursue that here as our goal is to present the essence of the idea.

`ListHandle t -> (u, ListHandle t)`) with which lists can be manipulated. Actions differ from simple values like 2 or `“hello”` in that evaluating the action causes some useful “side effect” to be carried out. Examples are such things as updating a table, or reading a line from IO. Each action takes the list handle, performs a computation and returns a pair consisting of the result of the computation of interest to the client and, conceptually, a new list handle. We say conceptually because the implementation of the list handle may in fact use a mutable structure. Monads allow us to elegantly isolate such details from the rest of the declarative code.
Now we can provide some useful functions on top of the basic actions, called commands. In Haskell, commands are just expressions that reduce to actions (as opposed to ordinary expressions that reduce to values). An example of such a command is the `at` function discussed above, which we call `at’` to distinguish it from the one in the PIM

```
at’ :: int -> List t u
at’ = λi->List (λlistH->(at (dataFromHandle listH) i, listH))
```

This defines `at’` using a lambda abstraction. The lambda abstraction defines an anonymous function that takes a parameter `i`, and returns another instantiated lambda abstraction that actually carries out the action. This second abstraction takes a list handle, and returns a pair consisting of the required value at the given index and the new list handle, using `dataFromHandle`, a (server side) function that maps handles to lists. Note that an expression such as “`at’ 3`” does not do anything until the returned action is evaluated.

Client code to access lists would now look something like:

```
do i <- <some command calculates an index>
   x <- at’ i       -- get the value at index i and assign to x
   y <- at’ 10      -- get the value at index 10 and assign to y
   …compute using x and y…
   return <result>
…etc…
```
The do syntax is a way of sequencing commands so that the list handle is automatically passed from one command to the next.

Incorporating a cache

In order to get the more efficient caching form of this monad, the state now has to incorporate the cache. For this we replace the `ListHandle` type with `CList` (for Cached List). That is:

```
data List t = List(CList t -> (u, CList t))
data CList t = (Cache t, ListHandle t)
data Cache t = [t]
```

the definitions of the two required monadic operators (>>= and `return`) are as follows:
```
instance Monad List where

(>>=) :: List t -> (t -> List u) -> List u
List cmd1 >>= lambda_cmd2
      = List(λcList -> cmd2 refreshedCList
                    where List cmd2 = lambda_cmd2 cmd1Value
                          refreshedCList = refresh newCList
                          (cmd1Value, newCList) = cmd1 cList
```

That is, the left command `cmd1` is evaluated first. The second element of the result, namely the possibly updated list is passed to `refresh` to refresh the cache. Then the lambda-wrapped second command is unwrapped[3] using the computed value returned by `cmd1` and evaluated with the refreshed list.

```
return :: t -> List t
return x = List (λcList ->(x, cList))
```

`return` is just a way of turning a value into a command.

[3] technically, a β reduction

(>>= is a slight variation on the usual definition for a state monad, with the inclusion of a call to the refresh function, which refreshes the cache, as defined below)

Now instead of `at'`, we have a function `cat'` (for caching `at'`) which, is very similar to `at'`, except that reads its data from the cache part of the incoming object instead of the list handle.

```
cat' :: int -> List t
cat' = λi–>List (λcList–>(at cache i, cList))
       where cache = fst cList
```

Since all the details of the cache refresh are inside the >>= operator it would have been nice to simply reuse the original at' command. However, they have slight differences in their operation which prevents that.

`refresh` is a function that looks at `cache` and the desired index (`i`) and determines whether or not the cache needs to be extended. Its definition is:

```
refresh :: CList t -> Int -> CList t
refresh (cache, handle) i =
       if length cache > i
       then (cache, handle)
       else (fetchDataFromServer (cache, handle) i, handle)
```

fetchDataFromServer is a primitive function on the server which appends the next n elements to the given list's cache and returns the new cache. Its behavior is *specified* as shown below. Note however, that this is not how it is likely to be actually implemented

```
fetchDataFromServer (cache, handle) n =
    cache ++ additionalElements
    where
       additionalElements = sublist (length cache) n (dataFromHandle handle)
       sublist m n = take (n-m+1) . drop m
```

A proof outline of the correctness of the transformation of the straightforward version of at to the optimized version cat' is sketched out in the Appendix.

The next example looks at deriving the composition of patterns in a functional language.

Combining Patterns: Iterator and Observer

The Iterator pattern [GHJV95] allows flexibility in traversing a collection, such that a number of ways of traversing the collection are available independent of the exact data representation of the collection. The Observer pattern [GHJV95] allows an arbitrary number of observers to be notified of changes to an entity, such that the entity itself need not be aware of which and how many observers are interested in the changes. In some situations it is common for several observers to be observing a fairly large dataset (for example information returned from a database query). If we naively combine the two patterns, each observer will be notified in turn of changes to the dataset and will then iterate over that dataset to take whatever action it needs. For large datasets, this could be quite inefficient. This is something we would like to address when transforming the PIM to a design PIM. A solution to this problem, called the Iterator-Observer Pattern is described in [Bi03]. Iterator-Observer replaces each observer iterating separately over the dataset with a single iteration over the dataset where each observer is notified in turn at each data element. Below we show how to define the functional language equivalent of Iterator-Observer.

First we need to introduce some rough functional analogs of the standard Iterator and Observer patterns.

Observer

For the functional analog of the Observer pattern, we assume there are n update functions (or callbacks) that are to be called when an Observable (a piece of data) changes. The function required to do this is somewhat the compliment of the Haskell library function `map`, except that it takes a list of functions and a single piece of data (as opposed to a single function and a list of data) and applies each function in turn to the piece of data. We will call this function `mapfs` and define it as follows:

```
mapfs :: [t->u] -> t -> [u]
mapfs [] x = []
mapfs (observer:observers) x = (observer x):mapfs observers x
```

Iterator

In the OO world, Iterator allows traversal over a collection, calling a fixed method at each element. To keep things simple, we will approximate this in the functional world with the use of the library function `map` which applies a function to each element in a collection. The following expression does the trick:

```
map update collection
```

Note that each update takes a piece of data and returns a result.

Notification

The function to carry out notification of changes to an Observable we call notify.

```
notify :: [t->u] -> [t] -> [[u]]
```

Notify takes a list of functions (updates) and invokes them in turn over the dataset. In the absence of Iterator-Observer, Notify is implemented by straightforward composition of the two patterns:

```
notify updates dataset =
      mapfs (map map updates) dataset                                        (1)
```

Intuitively, the innermost expression (`map map updates`) maps the `map` function over the list of update functions (`updates`) to arrive at a list of closures, each of which can map an `update` function. `mapfs` then "closes" each closure with the `dataset` argument. As an example, suppose the `dataset` is

```
[d1, d2,…,dn]
```

and the list of update functions is

```
[fa, fb]
```

Then `map map updates is`

```
[map fa, map fb]
```

Therefore `mapfs (map map updates) dataset is`

```
[map fa dataset, map fb dataset]
```

=

```
[[fa d1, fa d2, …, fa dn]
 [fb d1, fb d2, …, fb dn]]
```

Iterator-Observer

Now consider introducing the Iterator-Observer pattern. To do this, the first iteration must be over the dataset, applying each update in turn to a data element.

```
map (mapfs updates) dataset
```

In the above the map represents the "outer" iteration, the mapfs represents the "inner" iteration applying each update to the data element. However, the result will be transposed w.r.t. to the required result above. Therefore we finally apply a transpose function to the intermediate result to arrive at the transformed form of notification

```
notify updates dataset =
      transpose (map (mapfs updates) dataset)                              (2)
```

Taking the example earlier,

```
map (mapfs updates) dataset
```

gives

```
[mapfs [fa fb] d1,
 mapfs [fa fb] d2,
 …
 mapfs [fa fb] dn]
```

=

```
[[fa d1, fb d1],
 [fa d2, fb d2],
 …
 [fa dn, fb dn]]
```
Transposing this list gives
```
       [[fa d1, fa d2, …, fa dn]
        [fb d1, fb d2, …, fb dn]]
```

Notice that in the informal description of the pattern in [Bi97], it is not immediately obvious that application of Iterator-Observer is slightly different to applying Iterator to each invocation of Observer. The danger is that such subtle differences may not show up until run time and produce unexpected behavior (in the OO case, if each observer for example updates some visual display as it iterates over the dataset, the two forms will produce slightly different visual effects). On the other hand, the functional form, because it can be algebraically derived, makes it clear that the transpose function is required to really equate the two.

Future Work

We would like to look at more sophisticated example of functional program transformation for the purposes of mapping large scale functional programs to implementations. Again, we wish to draw inspiration from the use of design patterns, architectural patterns, and model transformation in use in the OO community. Of course, this is only possible if we have precise definitions of the source domains. Formulating such definitions is another area for exploration which we feel will be fruitful.

Complementary to the problem of mapping to a target architecture is the equally challenging problem of weaving together functional specifications, the way aspects are weaved together.

Good tool support to carry out transformations and manage libraries of transformations is going to be essential. We plan to look at tools such as MAG [MS] and XT [JVV01] from the program transformation world.

We would also like to investigate “fusion” languages such as Scala [OAC+04] and O Haskell [No99] that provide support for OO concepts such as classes and inheritance within a functional programming paradigm. This would make for a smoother conversion from OO models (defined for example in a declarative UML subset) to the specification language. We hope to be able to use one of the model transformation tools such as GMT[BEW03] for this purpose.

Related Work

As noted earlier, viewing patterns as transformations was suggested by Tokuda and Batory [TB95] in the context of object oriented programs. Opdyke in his PhD thesis [Op92] suggested refactorings be viewed as transformations. And recently [TR01] refactoring has also been applied to functional programs. However, refactorings are a simple case of transformation in which the structure, but not the meaning, of the code is changed. Meanwhile, in the functional programming community, program transformation has been studied for quite some time. Over twenty years ago, Darlington discussed program transformation based on equational reasoning [Da82]. Many books on functional programming ([BW89], [Th96]) have at least a chapter or two on synthesizing or proving properties of functional programs. However, much of the attention on transformation in functional programming has been focussed on reducing algorithmic complexity and not on structural or model transformation. *We want to leverage such work towards solving a slightly different problem*.

Our work also shares the same goals as much of the work in the fields of domain modeling and of MDA. The MIC group at Vanderbilt has been very active in the area of model transformation and aspect weaving. For example, [SAL+03] uses graph grammars to transform domain models. However, their emphasis is primarily on structural transformation. We believe that transforming behavior is at least as important as transforming structure. Also, they do not discuss how to prove or derive their transformations. Gray et.al. [GZL+04] apply transformation rules to legacy code using the DMS toolkit from Semantic Designs [SD]. However, their goal is reverse engineering rather than forward synthesis.

The primary difference is in our use of declarative models, and of algebraic transforms on those models.

References

[ACM01] D Alur, J Crupi, D Malks, *Core J2EE Design Patterns: Best Practices and Design Strategies*, Prentice Hall PTR, 2001

[BEW03] J Bettin, G van Emde Boas, E Willink, “Generative Model Transformer: An Open Source MDA Tool Initiative”, *Proc. OOPSLA*, 2003

[Bi97] P Bishop, “Java Tip 38: The trick to “Iterator Observer””, *Java World*, October 1997.

[Bl01] J Bloch, *Effective Java Programming Language Guide*, Addison-Wesley, 2001.

[BMR+96] F Buschmann, R Meunier, H Rohnert, et.al., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[BW89] R Bird, P Wadler, *Introduction to Functional Programming*, Prentice Hall, 1989

[CE00] Czarnecki and E Eisenecker, *Generative Programming: Tools, Techniques, and Methods*, Addison Wesley, 2001.

[Da82] J Darlington, “Program Transformation”, in *Functional Programming and its Applications*, Cambridge Univ. Press, 1982.

[Fr03] D Frankel, *Model Driven Architecture*, Addison-Wesley, 2003.

[GJHV95] E Gamma, R Helm, R Johnson, J Vlissides, *Design Patterns: Elements of Reusable Software*, Addison-Wesley, 1995.

[GZL+04] J Gray, J Zhang, Y Lin, et.al., “Model-driven Program Transformation of a Large Avionics Framework”, to appear in *3rd Intl. Conf. on Generative Programming and Component Engineering (GPCE 04)*, 2004

[JVV01] M de Jonge, E Visser, J Visser, “XT: A bundle of program transformation tools”, *Electronic Notes in Theoretical Computer Sci*., 44, #2, 2001

[MB02] Steve Mellor and Marc Balzer, *Executable UML*, Addison Wesley, 2002.

[MS99] O de Moor and G Sittampalam, “Generic Program Transformation”, Proc. 3rd Intl. Summer School on Advanced Functional Programming, LNCS v.1608, Springer Verlag, 1999

[No99] J Nordlander, *Reactive Objects and Functional Programming*, PhD Thesis, Dept. of Comp Sci., Chalmers Univ. of Technology, Sweden, 1999.

[OAC+04] M Odersky et.al, *An Introduction to Scala*, Programming Methods Laboratory, EPFL, Switzerland, June 2004

[OMG] OMG, *Model Driven Architecture* homepage, www.omg.org/mda.

[Op92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992. Tech. Report UIUCDCS-R-92-1759

[pUML] Precise UML group home page, http://www.cs.york.ac.uk/puml/.

[SAL+03] J Sprinkle, A Agrawal, T Levendovszky, F Shi, G Karsai, “Domain Model Translation Using Graph Transformations”, *10th IEEE International Conf. and Workshop on the Engineering of Computer Based Systems*, 2003.

[SD] The Design Maintenance System, Semantic Designs, www.semdesigns.com

[TB95] L Tokuda, D Batory, *Automating Software Evolution via Design Pattern Transformations*, Univ. of Texas at Austin, 1995.

[Th96] S Thompson, *The Craft of Functional Programming*, Addison-Wesley, 1996

[TR01] S Thompson. C Reinke, *Refactoring Functional Programs*, Computing Laboratory, University of Kent, 2001.

[Wa95] P Wadler, “Monads for Functional Programming”, in J Juering & E Meijer (eds), *Advanced Functional Programming*, Springer Verlag LNCS 25, 1995.

Appendix – Sketch of Proofs

For our transformed function (`cat′`) to be correct w.r.t. to the simple version, `at′`, we need to show that

```
       fst (at′ i xsh) = at xs i
```

where `xs` is an abbreviation for `dataFromHandle xsh`
and that

```
       cat′ i (c, xsh) = at′ i xsh
```

when `c <= xs`
where `c <= xs` means c is a prefix of `xs` (that is, `c` = `take n xs` for some n < `|xs|`)

The first requirement is straightforward to prove:

fst (at' i xsh)
= {unfold defn of at'}
fst ((λi->List (λlistH->(at (dataFromHandle listH) i, listH))) i xsh)
= {β reduction}
fst (at (dataFromHandle xsh) i, xsh))
= {xs = dataFromHandle xsh}
fst (at xs i, xsh)
= {defn of fst}
at xs i
□

The essence of the proof of the second requirement is to show that refresh preserves cache coherence. That is, we wish to show that

```
       fst (refresh (c, xsh) i) <= xs
```

when `c <= xs`

it is sufficient to show `c` is always a prefix of `xs` that is "long enough" (i.e. of length at least `i+1`). Below is a sketch of a proof of that c is a prefix of `xs` (the full proof can be obtained from the authors).

```
       fst (refresh (c, xsh) i)
= {case: i < |c|}
       fst (c, xs)
= {defn of fst}
       c
<= {assumption}
       xs

       fst (refresh (c, xs) i)
= {case: i >= |c|}
       fst fetchDataFromServer (c, xsh) i
= {defn of fetchDataFromServer }
       c ++ sublist xs (length c) i
= {defn of sublist}
       c ++ take (i – length c + 1).drop (length c) xs
<= {lemma, see below}
       xs
```

To complete this proof we need a technical lemma, which we do not prove here: for any lists `xs`, `ys`:

```
       xs ++ (take i).(drop |xs|) xs++ys  <= xs ++ ys, for any i < |ys|
```

We can demonstrate that c is "long enough", by noting that the resulting list is always at least `i+1` elements long, and is therefore "long enough".

A more detailed proof of the correctness of the transformation is available from the authors.
□