Multi-Purpose Shared Data Sources in a Functional Language

Steffen Michels and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
s.michels@science.ru.nl, rinus@cs.ru.nl

Abstract. Modern software systems deal with huge amounts of data from different sources. Data is stored at different locations, like in memory, in files, or in databases. Multiple threads, processes and remote machines access the same data concurrently. Many different solutions exist for solving different issues emerging from this. This include methods for abstracting from the actual storage, controlling access and keeping data consistent in case of concurrent accesses.
In this paper multi-purpose shared data sources, abstracting from all details but the type of data provided and received, are introduced. They provide a uniform way for dealing with various kinds of data sources, providing solutions for all of the issues mentioned above. Access control is achieved using the type system. Compositional, safe operations can be defined using atomic transactions. Finally, functional projections can be used to change the way data is accessed and sources can be combined. This makes is possible to abstract from how data is stored and distributed, which leads to highly reusable code.

1 Introduction

Modern software has to deal with a huge amount of data from different sources. Data can be stored at different locations, like in memory, in files, or in databases, and also in different formats. Multiple threads, processes and remote machines access the same data concurrently. But also the current time or measurements can be viewed as shared data. Figure 1 illustrated that sharing occur on multiple levels and includes various kinds of sources.

Many solutions exist for dealing with data sources. For instance, databases provide tables or key-value pairs abstracting from the actual storage of data. In object-oriented programming certain patterns are used, providing an interface for accessing data, hiding the actual source. For complex pieces of software, proper software engineering is essential. Access control is an important aspect, e.g. one wants to enforce that a part of the program can retrieve and change data only in a certain restricted way. In the object-oriented world getters and setters are used to control access. Those kind of interfaces also help to achieve loose coupling, making pieces of software reusable. *Lenses* [2] can be seen as a solution to access only parts of a data structure in a certain way in the functional programming world. Finally, for defining concurrent accesses several solutions exist. An

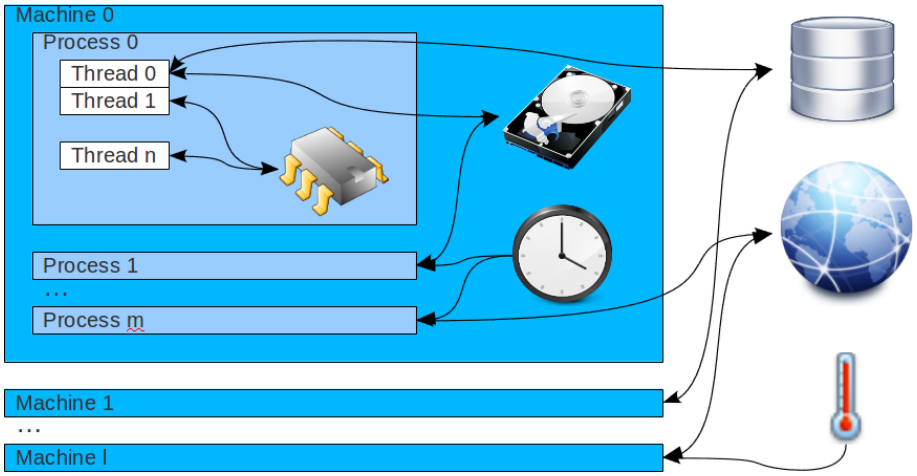

Fig. 1. Sharing is Everywhere

advanced solution for dealing with shared memory, allowing composition and avoiding common errors like deadlocks, are atomic transaction, as implemented in *Haskell* [4].

In this paper we propose a uniform way for dealing with different kinds of shared data sources, making it possible to deal with all issues discussed above. This enables programmers to abstract from the way data is stored and retrieved, such that one only has to deal with accessing data of certain types. Therefore, code can be reused when the way data is stored changes. This kind of abstraction perfectly fits in a pure functional language, like *Clean* which is used in this paper. The shared data source references introduced in the paper have the following advantages.

First, data sources are multi-purpose. They can represent arbitrary kinds of data storages, like memory, files or databases. After creation all data sources can be used in the same way. The actual operations can for example be mapped to operations on memory shared by threads, or on XML encoded files shared by different processes. Actually, even data sources other than data storage can be represented. For instance, it is possible to create a data source providing a stream of random numbers, or it can be a sensor to measure the current temperature.

Second, access control is statically enforced using the type system. The type of data read and written can be different. This makes read-only sources, but also more complex access restrictions possible. For example, it is possible to create a data source providing a list of appointments of several people, but only allowing to update the appointments of one particular person.

Third, data sources can be used inside atomic transactions. This allows to define operations on multi-purpose data sources, in a composable, safe way.

Fourth, version numbers are kept for each data source. They can be used to efficiently determine whether data is changed. They are not only used internally,

but also exposed to the user. This is for example useful to detect edit conflicts when human users are editing data and it is desirable that other users can edit the same data at the same time. An example is a Wiki where multiple users can change the same article.

Finally, data sources are composable. Basically this means that it is possible to create new data sources building on existing ones. Either functional projections can be used to change the type of data read or written or multiple data sources can be combined to a new one. The new data sources behave like basic ones. All operations are still possible, data remains consistent in case of concurrent accesses and version numbers can be used to detect changes. This makes is possible to reuse code even when information is organised in a different way, as long as there is a functional mapping between provided and required data sources.

In this paper first, Section 2 summarised the concept of *software transactional memory* (STM) and defines an interface for *Clean*. This will later serve as basis for defining the semantics of multi-purpose data source representing shared memory. How data sources can be generalised to multi-purpose sources is discussed in Section 3. In Section 4 it is described how data sources can be composed out of other ones. It is shown that this is a powerful abstraction mechanism, allowing code to become independent of how data is actually stored. Section 5 shows that our multi-purpose data sources provide a powerful way to deal with data sources in a *task-oriented programming* setting (the *iTask* system [7]). Finally, related work is discussed and conclusions are drawn in Sections 6 and 7.

2 Software Transactional Memory

Atomic transactions are a well known abstraction to deal with data in shared memory which is concurrently accessed. It has the advantage that programmers can specify and compose operations which should be done atomically as if they were normal operations. Actually performing changes as one atomic operation is solved by the system. This kind of abstraction prevents a common source of deadlocks, caused by circular dependencies.

Concurrent Haskell's transactions [4] shows that a strongly-typed, pure functional language is perfectly suited for this kind of abstraction. They can be implemented in software as STM. Although this implementation is here restricted to data shared in memory between threads, we use *Haskell*-style STM to define the semantics of our multi-purpose data sources for the case they represent shared memory. Hereafter we show that the concept can be generalised to deal with all kinds of data sources.

First, we define a *Clean* interface for *Haskell*-like STM. The basic idea of atomic transactions is that all operations are simulated in first instance. At the end of the transaction changes are actually committed, only if they can be done consistently. This means it must be possible to perform all changes made during the transaction as one atomic operation. Otherwise, the transaction is discarded and has to be redone.

Because operations are simulated, no arbitrary side-effects are allowed in transactions, but only a fixed set. In *Haskell* this is achieved using an `STM` monad on which only a restricted set of operations can be performed. Since in *Clean* I/O is realised using *uniqueness typing* [1], in this paper an abstract, unique environment, on which transaction operations are performed, is used:

```
:: *Trans *env
atomic :: ((*Trans *env) → (TRes a, *Trans *env)) *env → (a ,*env)
:: TRes a = YieldResult a | Retry
```

Using the unique `Trans` environment ensures that no side-effects can occur during a transaction. Since the transaction environment still contains the actual environment, its type is parametrised with this environment's type. In *Clean* this is typically `World` but it could also for instance be the file system. Here no context restriction is enforced for the environment. The environment restricts which data sources can actually be created, as discussed later.

The function defining the transaction can either yield an arbitrary result (`YieldResult`) or block until one the variables read inside the transaction change (`Retry`). Then the transaction is just performed again.

Operations are performed on variables (`TVar`s) representing memory shared between multiple threads. They have to be created explicitly, which is only possible for the `Memory` environment or an environment containing it (such as for example `World`)[1]:

```
:: TVar a *env
newTVar :: a *env → (TVar a *env, *env) | MemoryEnv env
instance MemoryEnv *Memory; instance MemoryEnv *World
```

The only operations which can be performed within an atomic transaction are reading and writing `TVar`s. Exceptions are omitted here. The semantics are equivalent to the one defined for *Haskell* transactions [4]:

```
readTVar  ::   (TVar a *env) (*Trans *env) → (a, *Trans *env)
writeTVar :: a (TVar a *env) (*Trans *env) → *Trans *env
```

Example 1. Adding an integer to a shared list of integers and yielding the modified list can be done by the following atomic operation[2]:

```
append :: Int (TVar [Int] *env) → ([Int], *env)
append i sharedList = atomic transaction env
where  transaction tr
         # (list,tr) = readTVar sharedList tr
         # list      = [i:list]
         # tr        = writeTVar list sharedList
         = YieldResult list
```

3 Multi-purpose Data Sources

The idea behind multi-purpose sources is the following: because the concept of transactions is a powerful abstraction for dealing with memory shared between threads, why not use it to concurrently access other kind of data sources as

[1] In *Clean* context restrictions are given at the end of a type signature after a '|'.
[2] The `#` is a compact notation for a let expression in *Clean*.

well? In this section the concept of transactional variables stored in memory is generalised such that the operations provided can also be used to handle different kinds of other data sources. Compile-time access control and the notion of versions is added. The semantics of this extension are expressed in terms of STM, as defined in the previous section. Finally, practical issues regarding the actual implementation for several other kinds of data sources are discussed.

3.1 Multi-purpose Source Representation

Examples of data storages we want to include are files shared between several processes, and databases shared between multiple machines. But data sources do not have to be restricted to data storages. A data source can also be for instance the current time or temperature. Actually each provider of data which is shared and changes over time can be seen as a data source. This is similar to *UNIX* files to which arbitrary data sources can be mapped.

Having data sources representing concepts like the current temperature gives the problem of access control. It is possible to read the temperature but not to change it. The strong type system can be used to statically enforce these kind of restrictions. We achieve this by parametrising data sources with two type parameters: the data which can be read and the data which can be written. This leads to a generalised type for representing multi-purpose data sources:

```
:: RWShared r w *env
```

The common case that a value of a certain type is shared (like with a `TVar`) is a special case with `r = w`. We use the type synonym `Shared` for this. It is also possible to express read-only and write-only data sources as special cases of the general type[3]:

```
:: Shared   a env :== RWShared a a    env
:: ROShared r env :== RWShared r Void env
:: WOShared w env :== RWShared Void w env
```

Although the implementation of the operations for the different kind of data sources will be quite different, for a programmer this is completely transparent. The only visible difference is how these data sources are created. Here creation functions for different kinds of sources are given:

```
sharedFile   :: Path (String → a) (a → String) *env → (Shared a *env, *env) | FileEnv   env
sharedMemory :: a                               *env → (Shared a *env, *env) | MemoryEnv env
time         :: ROShared Timestamp *env | WorldEnv env  // the current time
temp         :: ROShared Timestamp *env | WorldEnv env  // the current temperature
random       :: ROShared Int       *env | WorldEnv env  // a stream of random numbers
null         :: WOShared a         *env                 // thrash can, writing has no effect
```

3.2 Exposed Version Numbers

Before defining the semantics of multi-purpose data sources we introduce another useful concept, used for efficiently determining changes and conflicts. Internally, for the implementation of transactions, an efficient way to determine if a source

[3] `Void` is *Clean*'s unit type.

has changed, and therefore the transaction cannot be committed, is needed. This is solved by using version numbers which are increased if a value changed. Comparing two version numbers is more efficient than comparing (large) values.

We observed that version numbers are not only useful internally, but also for the user of data sources. An example is an application allowing human users to edit information, like *iTask* [7]. It is undesirable that such shared data sources are locked during the entire editing process. Instead editing conflicts should be detected and reported to the user.

Our concept of shared data sources allows very different kinds of implementations of actual sources. To make it possible to still reason about their behaviour, they have to obey some properties connected with the version number:

Property 1 (Version Stability). The version number of a data source is only increased because of a write operation. If no such operation is performed by any processes, the version number does not change.

Property 2 (Value Version Equality). To determine if the value has changed it is sufficient to compare the version number. If the version number is the same as for a previous read operation, the value is the same, too.

Property 3 (Version Increasingness). The version number is increased if the value is updated. It is never decreased. It can remain the same if the write operation does not change the value (e.g. of read-only data sources). This property is essential to make it possible to determine which of two versions is more recent. Also version numbers can be combined by just summing them up, which is essential for composition of data sources (Section 4.2).

For data sources just storing data (like shared memory or files), the behaviour of version numbers is obvious. They are incremented when a write operation is performed. Other abstractions, whose values change automatically, are modelled by imaginary processes updating them. For instance, the current time data source can be modelled with an imaginary process updating the timestamp each second. The data source providing random numbers can be imagined as being updated always just before a read operation is performed. So, it is possible to express the behaviour of version numbers of that kind of data sources with the same semantic model as data sources just storing a value in memory.

3.3 Shared Memory Semantics in Terms of STM

The semantics of shared memory sources are defined in terms of STM, as defined in Section 2. Since `TVar`s work on shared memory only, the semantics does not capture other kinds of data sources. In the next section (Section 3.4) it is shown how this can be generalised, by replacing the `TVar`s used in this section by a set of low level functions, which has to be implemented for each kind of data source.

We continue defining multi-purpose sources and operations on them, in terms of types and operations defined for STM. The type `RWShared` is represented by two `TVars` and two functions:

```
:: RWShared r w *env =∃ b: Shared (SharedRec b r w env)
:: SharedRec b r w *env = { value   :: TVar b       env , get :: b   → r
                          , version :: TVar Version env , put :: w b → Maybe b }
```

The two `TVars` store the basic value and the version number. Intuitively the basic value is the actually stored value. In this case the representation of the value in memory. For the generalised version discussed later, data could be stored in files as well. Then the functions could be used to map data to and from a string representation, for example JSON or XML. The basic value is of existentially quantified type `b`. The only way to access it is to use one of the two functions.

The first one (`get`) is used to retrieve a value of the read type form the basic value. The second one (`put`) is used to put a value of the write type back to the basic value. It also uses the current basic value since values of the write type might contain less information than the basic type. This idea is similar to functional lenses [2, 12]. The put function does not have to give a new value, the value can be left unchanged, for instance in the case of read-only data sources. The reason that `get` has to return a value is that a read operation always yields a value. For write-only sources this is always `Void`. The shared memory defined here, is a simple case where the basic, read and write types are equal:

```
sharedMemory :: a *env → (Shared a env, *env) | MemoryEnv env
sharedMemory x world
  # (val, world) = newTVar x world
  # (ver, world) = newTVar 0 world
  # shared = Shared { value= val, version = ver, get = id, put = const o Just }
  = (shared, world)
```

The `atomic` function and the unique transaction environment of STM can still be used. New read and write operations have to be define for `RWShared`. They have to make use of the get and put functions. Additionally the write operation increments the version[4]. Since the operation works on a transaction state writing the value and updating the version is done atomically[5]:

```
transRead :: (RWShared r w *env) (*Trans *env) → (r, (*Trans *env))
transRead (Shared {value, get}) env
  # (b, env) = readTVar value env
  = (get b, env)

transWrite :: w (RWShared r w *env) (*Trans *env) → *Trans *env
transWrite w (Shared {value, put, version}) env
  # (b, env) = readTVar value env
  = case put w b of  Nothing = env
                     Just b‘
                       # env        = writeTVar b‘ value env
                       # (ver, env) = readTVar version env
                       = writeTVar (inc ver) version env
```

An additional operation is added to expose the version numbers which can simply be read from the corresponding `TVar`:

```
:: Version :== Int
transGetVersion :: (RWShared r w *env) (*Trans *env) → (Version, *Trans *env)
transGetVersion (Shared {version}) env = readTVar version env
```

[4] Incrementing the version by one is actually an arbitrary choice. As long as the new version is greater than the old one Property 3 is fulfilled.

[5] The syntax {x,y,...} in a pattern is used to match fields of a record.

Example 2. Example 1 can straightforwardly be rewritten for multi-purpose sources using the new operations:

```
append :: Int (Shared [Int] *env) → ([Int], *env)
append i sharedList = atomic transaction env
where  transaction tr
         # (list,tr) = transRead sharedList tr
         # list      = [i:list]
         # tr        = transWrite list sharedList
         = YieldResult list
```

3.4 Extending the Implementation for Different Kinds of Sources

The semantics captured by the implementation in terms of STM, given in the previous section, were restricted to shared memory. To abstract from the actual kind of data source, we identify a set of basic operations needed to realise the high level operations, defined before.

Obviously, it must be possible to read data from a source. Concrete examples of read functions are functions reading data from memory or files and yielding a value of the basic type. A read function could also retrieve the current time or use a hardware sensor to measure a value. Because data is accessed concurrently, it must be made sure that data is not changed while it is read by another process. Therefore low level functions for locking and unlocking a source have to be provided, too[6].

At the end of a transaction the actual changes are committed. This consists of two steps: checking for consistency and then possibly writing the changes. This has to be done atomically which can be achieved by using the locking functionality. During the transaction, a transaction log keeps track of version numbers and possibly values to write. This requires a low level function for retrieving the version. For storages it can just be a counter incremented each time a value is written. Other kinds of sources have to choose a convenient way determining a version number, obeying the properties defines in Section 3.2. For instance the version number of the current time could be a timestamp.

With the low level functions defined so far, checking for consistency becomes comparing integers. If the transaction can be committed consistently, changes have to be written, for which another low-level function is needed. The function has the obvious task for data storages and might be undefined for read-only sources, like the current time.

Finally, after a retry there must be an efficient way (no busy waiting) to wait for changes of a set of data sources. This becomes more complex as the scope data sources are shared becomes larger. For memory shared within a process, the program has full control when data is written to a source, and the implementation is relatively easy. Detecting changes of files requires notification methods of the OS. This case has been implemented as well. Efficiently waiting until a database accessed by several machines has changed, seems to be doable as

[6] For performance reasons a shared source can make a difference between an exclusive and a shared lock. This allows multiple processes to read the same data simultaneously in case it is not changed.

well, but remains future work. For some non-storage sources one has to choose a refresh interval, determining for instance how frequently the temperature is measured.

4 Shared Data Source Combinators

The multi-purpose sources allow to use different kinds of data sources using the same abstraction. In this section it is shown that an even higher level of abstraction can be achieved making it possible to derive new data sources from existing ones using combinators.

4.1 Projections

It has already been shown that access control can be achieved using different types for data read and written. For some data sources, such as the current temperature, it is obvious that they are created with certain access restrictions. However, for software engineering reasons, one might want to change the way data of an existing data sources can be accessed. For this purpose, projection functions which change the read and write type of data sources, can be used.

The read type can be changed by providing a function from the old to the new read type. For changing the write type a more sophisticated function is needed since values of the write type might contain less information than that of the read type and writing is optional. Projections can be implemented by composing the projection functions with the put and get functions stored in the source[7]:

```
mapRead :: (r → r') (RWShared r w *env) → RWShared r' w *env
mapRead get' (Shared share=:{SharedRec | get}) = Shared {SharedRe c| share & get = get' o get}

mapWrite :: (w' r → Maybe w) (RWShared r w *env) → RWShared r w' *env
mapWrite put' (Shared share=:{SharedRec | get,put})
  = Shared {SharedRec | share & put = λw' b → maybe Nothing (λw → put w b) (put' w' (get b))}
```

Example 3. Turning a data source in a read-only one is a special case of projecting the write type:

```
toReadOnly :: (RWShared r w *env) → ROShared r *env
toReadOnly shared = mapWrite (λ_ _ → Nothing) shared
```

A combinator for projecting both types of a data source can simply be derived:

```
mapRW :: (r → r', w' r → Maybe w) (RWShared r w *env) → RWShared r' w' *env
```

A functional lens which is a well known method to access only a part of a larger data structure is a special case of this combinator. When `r = w` and writing is obligatory the type of `mapRW`'s first argument becomes `(s → v, v s → s)` which is a simple get/set notation of a lens.

[7] {`record & x = ..., ...`} denotes a record with a number of fields updated.

Example 4. Using a lens, from a shared tuple of two values, a data source sharing only the first value can be derived:

```
fstLens :: (Shared (a,b) *env) → Shared a *env
fstLens tuple = mapRW (fst, λx (_,y) → Just (x,y)) tuple
```

Because pure functional projections are used to map values to a different type, the behaviour of data sources after projection is not essentially different. In particular all version properties, defined in Section 3.2, do still hold.

4.2 Composition

Projection provides a powerful way to provide data of a certain type abstracting from what kind of data is actually stored. It still has the restriction that all data must be retrieved from one single source. One might want to provide a data source, which is actually a combination of several different sources. To overcome this restriction a combinator for composing two data sources (⋊⋉) is introduced.

A second constructor is added to `RWShared` to represent composed data sources. Composition can be applied repeatedly to build arbitrary large composed data sources. Since the type remains `RWShared`, composition is completely transparent for the user.

```
:: RWShared r w *env =∃ b:           Shared         (SharedRec b r w env)
                     | ∃ rx wx ry wy: ComposedSource (Composition r w rx wx ry wy env)
:: Composition r w rx wx ry wy *env =
    { srcX :: RWShared rx wx env, get :: rx ry   → r
    , srcY :: RWShared ry wy env, put :: w rx ry → Maybe (wx, wy) }

(⋊⋉) infixl 6 :: (RWShared rx wx *env) (RWShared ry wy *env) → RWShared (rx,ry) (wx,wy) *env
(⋊⋉) shareX shareY = ComposedSource
   { srcX = shareX, srcY = shareY, get = λrx ry → (rx,ry), put = λ(wx,wy) _ _ → Just (wx,wy) }
```

Similar to the get and put function of a basic share there are functions to combine data retrieved from both shares and to write data back.

Example 5. The combination of composition and projection is very powerful. A concatenation operation for two read-only lists yielding a new list can be defined like this:

```
concat :: (ROShared [a] *env) (ROShared [a] *env) → ROShared [a] *env
concat x y = mapRead (λ(x,y) → x ++ y) (x ⋊⋉ y)
```

Composition makes it possible to combine different kinds of data sources, which is completely transparent for the user. An example of a composed source using different kinds of storages is given in Figure 2.

The concepts are powerful enough to put for instance a *symmetric lens* [5] between two data sources. A derived combinator can be defined as this[8]:

```
symLens :: (a b → b) (b a → a) (Shared a *env) (Shared b *env) → (Shared a *env, Shared b *env)
symLens putr putl sharedA sharedB = (newSharedA, newSharedB)
where  sharedAll  = sharedA ⋊⋉ sharedB
       newSharedA = mapReadWrite (fst, λa (_,b) → (a,putr a b)) sharedAll
       newSharedB = mapReadWrite (snd, λb (a,_) → (putl b a,b)) sharedAll
```

[8] A simple notation without *complement* (see [5]) is used here for pragmatic reasons, without consequences for the expressiveness.

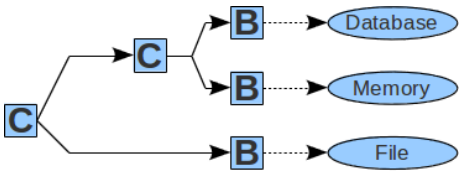

Fig. 2. Composed Sources (C) consisting of Basic Source (B) Referring to Different Kinds of Storages

One thing to note is that the type of the arguments and resulting shared sources remains the same. From the outside nothing changes, but there is a hidden connection between those two data sources. For the user of such a data source this is completely transparent, but if one is changes the other is changed as well.

For all operations using shares, a case for handling the composed case has to be added. Reading and writing is achieved by using the functions stored in the composition and performing read or write operations on the single shares. This happens atomically since the operation can be performed within a transaction only:

```
transRead :: (RWShared r w *env) (*Trans *env) → (r, (*Trans *env))
transRead (ComposedSource {srcX, srcY, get}) tr
  # (x, tr) = transRead srcX tr
  # (y, tr) = transRead srcY tr
  = (get x y, tr)

transWrite :: w (RWShared r w *env) (*Trans *env) → *Trans *env
```

After composition the properties defined in Section 3.2 should still hold. This can be achieved by simply adding the version numbers of the components. Since all individual version number increase, it is guaranteed that a unique, combined values belongs to each combined version.

Finally, similar as for the basic case, projection can be achieved by composing functions:

```
mapRead :: (r → r`) (RWShared r w *env) → RWShared r` w *env
mapRead get` (ComposedSource share=:{Composition|get})
  = ComposedSource {share & get = λrx ry → get` (get rx ry)}

mapWrite :: (w` r → Maybe w) (RWShared r w *env) → RWShared r w` *env
mapWrite putb` (ComposedSource share=:{Composition|put, get}) = ComposedSource
            {share & put = λw` rx ry → maybe Nothing (λw → put w rx ry) (put` w` (get rx ry))}
```

5 Example: *iTask*– Using Multi-purpose Sources in a Task-oriented Programming Setting

The *iTask* library allows to program webservices interacting with the outside world in a task-oriented style. This means that tasks are the main building blocks of such programs. The idea is to describe the tasks to do on a high level of abstraction. As much implementation details as possible are hidden. The system automatically keeps track of the state, generates serialisations and ways to interact with the system [7].

A very important part of the system is the ability to let users view or update data models, just by defining functional relations between the actual model and views used by the user. Details including generating an interactive webform and performing validation, such that only values of the proper type can be created, are all handled automatically. The system is not only able to deal with data local to a task, but also with shared data.

5.1 Products & Orders – Updating Data Models in the *iTask* System

A simple webshop is used to illustrate how shares can be used in a task-oriented programming setting. This kind of application can naturally be described in terms of tasks assigned to different human and non-human users: Customers using a web interface for generating orders, people maintaining the product database and printers printing invoices. To begin we define a few simple types describing products and orders:

```
:: Product = {id :: ProductID, name :: String, description :: String, price :: Int}
:: Order = {date::Date, customer::Customer, products::[(Int, Product)], status::OrdStatus}
:: OrdStatus = New | InvoicePrinted | Shipped
```

In the system there is one store containing products and one for orders. They are represented by shared data sources[9]:

```
products :: Shared [Product]; orders :: Shared [Order]
```

In order to allow users to interact with the system special tasks making it possible to change local or shared states are provided. The task updating a shared state is similar to the one defined in [10][10] [11]:

```
:: Description :== String
updateSharedInformation :: Description (RWShared r w) → Task r | iTask r & iTask w
```

For example the following task one-liner automatically generates a webform a human user can use to update the product store, as shown in Figure 3:

```
updateSharedInformation "Update product store" products
```

We use two kinds of powerful abstractions here. We abstract from the actual updated data store by using a share introduced in this paper. The actual data could be stored in, for instance, files or some kind of other database. Additionally we use the abstraction provided by the task having the goal of updating a data source, abstracting from how this is done. The system automatically generates a webform and ensures that all fields are filled in correctly, such that type-safety is remained.

[9] Within the *iTask* system a type synonym is provided such that `Shared` can be used without the environment parameter. All shares within the system use the same internal *iTask* state.

[10] The `iTask` context restriction includes all utility functions used by the system, for instance for storing the state, generating webforms and verifying inputs.

[11] In the actual system the task is more complex. The most important difference is that it is possible to combine local and shared data to a single view.

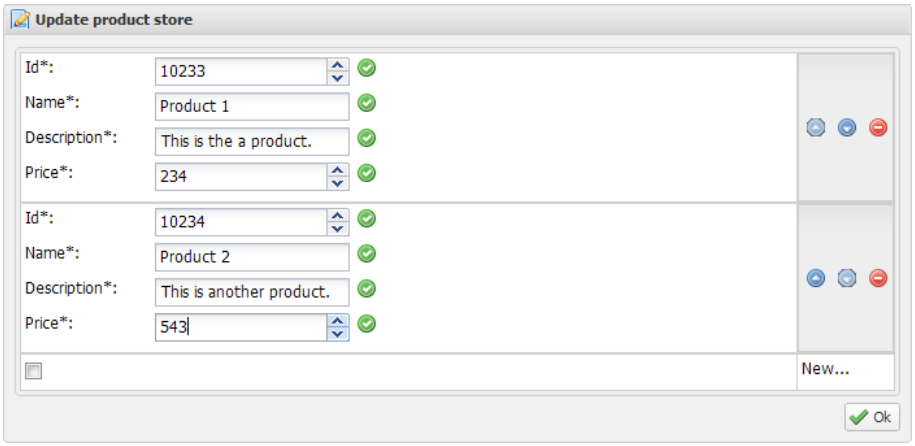

Fig. 3. Automatically Generated GUI for Updating a Product Store

To build more complex tasks those basic tasks can be combined to more complex ones, basically using sequencing an parallel branching. Also it is possible to tune the layout of the generated user interface. The details are out of the scope of this paper.

5.2 Printing Invoices – Making Tasks Independent from Data Storage

The concept of tasks is not only powerful for describing interactions with human users but also with other machines, such as printers. In this example we assume that there are several printers producing invoices for new orders and changing the status of orders to `InvoicePrinted`. Without giving the implementation we use the following task for this:

```
printInvoices :: (Shared [Order]) → Task Void
```

The task does not yield a result. There could be one instance of such a task for each printer waiting for work and running until stopped from the outside. While this is a working solution, bad encapsulation is achieved. The task `printInvoices` has to extract all information needed for printing an invoice from an order. When the representation of orders change in the system, the task has to be refactored, too. Further, it can access and change too much information. Actually the task could change the date or customer of an order while its purpose is to sorely print invoices and change the status from `New` to `InvoicePrinted`.

To solve this, first we introduce a new type containing only the information needed to perform this tasks:

```
:: Invoice = {data :: Date, customer :: Customer, products :: [(Int, Product)], printed :: Bool}
```

Using projections (Section 4.1) it is possible to map the order source to an invoice one. Additionally write access is restricted to the `printed` field. The new

task working on such a data source cannot change the date or customer of an order[12]:

```
invoices:: (Shared [Order]) → RWShared [Invoice] [Bool]
invoices orders = mapRW (get, put) orders
where  get ords           = [{Invoice|date = date,...} \\ {Order|date,...} ← ords]
       put printed orders = Just [{Order|o & printed = p} \\ o ← orders & p ← printed]

printInvoices :: (RWShared [Invoice] [Bool]) → Task Void
```

To show how powerful this approach is we examine the case the order type is changed. In the original type the products themselves are stored which makes it hard to keep the system in a consistent state when a product changes and wastes storage space because products are copied into each order. A better approach would be to store the IDs of products which leads to a new type for orders:

```
:: Order = {date::Date, customer::Customer, products::[(Int, ProductId)], status::OrdStatus}
```

Using a combination of projection and composition (Section 4.2) a mapping from the order and product store, to a data source usable by `printInvoices`, can be defined straightforwardly. The implementation of `printInvoices` can remain unchanged.

```
invoices :: (Shared [Order]) (Shared [Product]) → RWShared [Invoice] [Bool]
```

Summarized compositional shared data sources add a new kind of abstraction to the *iTask* system. The task is not only a high-level description of its purpose, abstracting from implementation details. It can also abstract from how data storage is done by using a shared source providing and processing exactly the information it needs. This leads to highly reusable task descriptions.

6 Related Work

Concurrent Haskell's *MVar*s [11] provide a way to share data between and synchronise concurrent processes in a functional language. They behave similar as channels and are more suited to synchronise the access to a shared data source rather than providing the data by themselves. Variables used within atomic transactions (*TVar*s) [4] are more similar to the shared data references introduced in this paper but are less powerful. For example they are not compositional and are restricted to data stored in memory, shared between threads.

LINQ [9] provides a solution for abstracting from the actual implementation of a data source and provides a query language to retrieve data from a collection. This collection could for instance be a collection of objects from the host programming language, relational or *XML* data. Queries are only used to retrieve data, not to update it, and only collections are handled while the approach in this paper can deal with data sources of arbitrary type. Further the language is restricted to operations like traversal, filtering, and projection. The approach in

[12] Ideally one would want to express that the length of the list of invoices read equals the length of flags written back. It is not possible to enforce this statically using *Clean*'s type system. Therefore this issue has to be handled at runtime.

this work, using functions, is more powerful but might be less efficient. Similar approaches exist for functional languages [6, 3].

Finally, there is work showing that relational databases can automacially be mapped to types of a functional language. This also allows writing back changes [8]. Even if less general than the approach proposed here (restricted to databases, not arbitrarily composable, ...), this work might give ideas for an efficient way to map relational databases to shared data sources, introduced in this paper.

7 Conclusions & Future Work

We introduced multi-purpose shared data sources, a uniform way of dealing with issues related to sharing data. After creation the actual implementation is completely hidden for the user. The only information given is the type of the data which can be read from and the type of the data which can be written to the data source. Using two types allows enforcing access restrictions statically using the type system.

The semantics for the shared memory case are defined in terms of STM and it is shown how the concept is extended to be able to deal with all kinds of data sources. This is implemented for shared memory and files. Therefore, different kinds of multi-purpose sources can be used together in atomic transactions, making it possible to define composable, safe operations.

Another powerful abstraction is introduced using shared source combinators. The way data is accessed can be changed using functional projections. Sources can also be combined, making it possible to abstract from the distribution of data.

Finally, we showed how this abstraction can be used in a task-oriented programming framework. Shared data sources can now be defined with the same level of abstraction as interaction with the user. Code using data sources becomes highly reusable.

While multi-purpose data sources provide a very general and powerful abstraction, they can become very inefficient. Using functional projections requires reading all data the original sources provide and then possibly discarding a huge amount of it. So finding a way to improve the efficiency of projections is an important direction for future work. Especially, shares representing databases should only retrieve data really needed. Other open issues related to databases are integrating their various synchronisation mechanisms and finding ways to efficiently wait for changes.

References

1. Peter Achten, John van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In John Launchbury and Patrick Sansom, editors, *Proceedings of the 5th Glasgow Workshop on Functional Programming, GFP '92, Ayr, UK*, Workshops in Computing, pages 1–17. Springer-Verlag, 1992.

2. Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.
3. George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *Revised selected papers of the 22nd international symposium on Implementation and Application of Functional Languages, Alphen aan den Rijn, Netherlands*, volume 6647 of *Lecture Notes in Computer Science*. Springer, 2010. Peter Landin Prize for the best paper at IFL 2010.
4. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.
5. Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA*, pages 371–384. ACM, 2011.
6. Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35:109–122, December 1999.
7. Bas Lijnse and Rinus Plasmeijer. iTasks for End-users. In Marco Morazán, editor, *Proceedings of the International Symposium on the Implementation and Application of Functional Languages, IFL '09*, volume SHU-TR-CS-2009-09-1, pages 22–23. Seton Hall University, South Orange, NJ, USA, September 23-25 2009. To appear as revised paper.
8. Bas Lijnse and Rinus Plasmeijer. Between types and tables - Using generic programming for automated mapping between data types and relational databases. In Sven-Bodo Scholz and Olaf Chitil, editors, *Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL '08*, volume 5836 of *LNCS*, pages 272–290, Hatfield, UK, 2011. Springer.
9. Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.
10. Steffen Michels, Rinus Plasmeijer, and Peter Achten. iTask as a new paradigm for building GUI applications. In Jurriaan Hage and Marco T. Morazán, editors, *Proceedings of the 22nd International Symposium on the Implementation and Application of Functional Languages, IFL '10, Selected Papers*, volume 6647 of *LNCS*, pages 153–168, Alphen aan den Rijn, The Netherlands, 2011. Springer.
11. Simon Peyton Jones, Andres Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd International Symposium on Principles of Programming Languages, POPL '96*, pages 295–308, St.Petersburg Beach, Florida, January 1996. ACM Press.
12. Rinus Plasmeijer and Peter Achten. The implementation of iData - A case study in generic programming. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Revised Selected Papers of the 17th International Workshop on the Implementation and Application of Functional Languages, IFL '05*, number 4015 in LNCS, pages 106–123, Dublin, Ireland, 19-21, September 2006.