Multi-Purpose Shared Data Sources in a Functional Language – *Extended Abstract* –

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. Web applications supporting people to collaborate have to deal with a lot of different kinds of data sources. Those different kinds of data sources use different mechanisms to retrieve and store data. As a consequence, the same operations on different kinds of data sources often require different implementations, which leads to non-reusable code and huge refactoring effort in case the data source implementation is changed. To avoid this situation the two concerns, defining **what** a data source is and **how** it is used, should be separated. In this paper we propose *shared data sources* (SDS) as a uniform way for dealing with different kinds of data sources, using functional programming techniques. SDSs abstract from the actual implementation of a data sources, provide access control and are composable. They enable programmers to separate defining and using data sources, resulting in highly reusable code.

1 Introduction

Web applications supporting people to collaborate have to deal with a lot of different kinds of data sources. Data might be stored in databases for a long term, while other data might only be needed for a short duration for accumulating a result which is not needed any more after it is processed. Furthermore, applications probably use meta data to handle the different users and processes currently performed.

Those different kinds of data sources use different mechanisms to retrieve and store data. As a consequence, the same operations on different kinds of data sources often require different implementations. Code is not reusable. When the data sources an application uses are changed, this requires refactoring of these implementations as well. This can require much effort, especially if application logic is mixed with implementing what a data source actually is.

To avoid this situation the two concerns, defining **what** a data source is and **how** it is used, should be separated. In this paper we propose *shared data sources* (SDS) as a uniform way for dealing with different kinds of data sources. They enable programmers to separate defining and using data sources, resulting in highly reusable code. We use the power of functional programming to achieve

this goal. 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.

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. 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.

The remainder of this paper is organized as follows: Section 2 gives the general idea how to solve the problem. The two concerns usage and definition are discussed in detail in Sections 3 and 4, respectively. We use the functional language *Clean* for code examples. The semantics of the proposed SDSs are defined in Section 5. Related work is discussed in Section 6 and conclusions are drawn in Section 7.

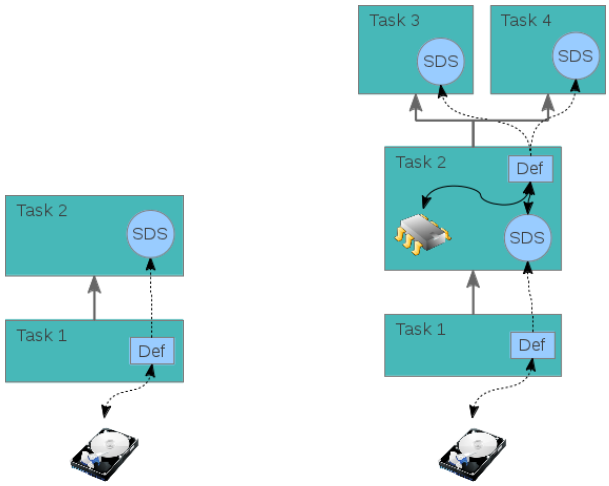
2 Separating Usage and Definition

To describe the implementation of web applications we use the concept of *tasks*. A task is the description of a piece of work that has to be done. Applications consist of basic tasks, such as entering information, which can be combined to more complex ones. Tasks can for instance be combined by executing them in sequence or in parallel. A more detailed definition of our concept of tasks is given elsewhere [7].

There are two aspects of defining tasks dealing with data sources: using a data sources and defining what reading and writing data from the source actually means. The goal of our solution is to separate those two concerns.

When using a data source one only has to know that there is a source providing and receiving data of a certain type, completely abstracting from how these operations are implemented. On the other hand, one needs to define what reading and writing data actually means. To achieve this we use an abstraction called *shared data source* (SDS).

Figure 1a illustrates this idea for a simple case. In the definition of Task 1 another task, Task 2, is used. Task 2 sorely uses an SDS and defines operations on data. That all operations are actually performed on files on a hard disc is determined by Task 1. This task defines an SDS performing operations on files and passes it to Task 2, abstracting from the actual operations this task performs. This mechanism makes Task 2 highly reusable, since it can perform operations on all kinds of data sources, depending on the SDS provided to it.



(a) Simple definition and usage (b) Combined definition and usage

Fig. 1: Definition and usage of SDSs

The situation can be more complex, as shown by Figure 1b. Tasks 3 and 4 are executed in parallel. Some of the data they share is mapped to a hard disc, but some part of the data is only needed during their lifetime to accumulate the result. This part is therefore stored in temporal memory. Tasks 3 and 4 just define operations on data, they are neither aware that they are executed in parallel nor that some of the data they work on is combined from a hard disc and temporal memory. Task 2 uses an existing SDS to define a new one. A part of the data is stored in a temporary storage created by the task. This shows that SDSs can be defined for any kind of data source. Task 2 abstracts from how the non-temporary part of the data is stored. This is determined by Task 1.

An important software engineering principle is access control. *Access control* in this context means that the SDSs a task uses only allow the operations actually needed for the task's purpose. This has two advantages. First, it prevents that a task mistakenly changes data. Second, if a task only requires operations that are really necessary it makes less assumptions about the context in which it is used. This leads to more reusable code.

This requires that SDSs support a way to indicate which part of the data can be accessed in which way. Tasks using SDSs should be defined in such a way that it requires only data which are really needed. Tasks defining SDSs then have to map the required data operation to the actual storage.

3 Using Shared Data Sources

This section deals with one of the two concerns discussed in this paper. It explains how SDSs look like and can be used by tasks.

3.1 Shared Data Source Abstraction

From the perspective of a user of an SDS, the most important property of an SDS is that it allows to abstract from the actual definition of a data source. This can be achieved by using an *abstract type* functioning as a reference to the actual source. The value cannot be inspected and only a set of operations defined elsewhere can be performed on it. Examples of these operations are discussed later in Section 3.2.

We want to make it possible to have access control. Basically, this means that what can be read is not necessarily the same than what can be written. We introduce the concept of *read* and *write types* to reflect this. Each SDS has two type parameters indicating the type of data which can be read from and can be written to the source, respectively.

Concretely, an SDS is represented by the abstract type `RWShared` with two type parameters:

```
:: RWShared r w
```

The common case that values of the same type can be read and written 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[1]:

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

3.2 Operations on Shared Data Sources

We give some possible operations on SDS as basic tasks, represented by the type `Task`. For the purpose of this paper they can just be seen as *monadic operations*. The most obvious operation on data sources are reading, writing and updating data[2]:

```
get    ::          (RWShared r w) → Task r | iTask r
put    :: w        (RWShared r w) → Task w | iTask w
update :: (r → w) (RWShared r w) → Task w | iTask w
```

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

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

The *iTask* system provides more sophisticated operations by offering tasks which make it possible to interactively update shared data. *Generic programming* techniques are used to automatically generate a user interface for any first order type. In this paper we use the following task with which a shared state can be updated interactively. It is is similar to the one defined in [6] [3]:

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

The first parameter is used to provide a human-readable explanation of the task's purpose and the third one is the SDS to operate on. The second parameter consists of two functions mapping data read to a view and a changed view back to the SDS's write type. For the case the read and write type is the same and the view should reflect the data without transformation, (`id`,`const`) can be used. For example, given there is a product store (`products :: Shared [Product]`), the following task one-liner automatically generates a webform a human user can use to update a product store, as shown in Figure 2:

```
updateSharedInformation "Update product store" (id,const) products
```

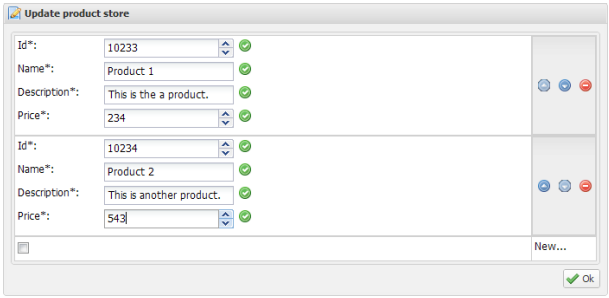


Fig. 2: Automatically Generated GUI for Updating a Product Store

We use two kinds of powerful abstractions here. We abstract from the actual updated data store by using an SDS. 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.

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 and given elsewhere [7].

[3] 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.

4 Defining Shared Data Sources

This section deals with the other concern discussed in this paper. It explains how SDSs can be defined by tasks providing them to other ones.

4.1 Basic Sources

Basically, an SDS can be defined by defining what reading and writing data means. This means that two functions have to be provided. In the *iTask* system, these functions work on the *unique* state `*IWorld` which can be used to perform arbitrary IO and additionally includes *iTask* specific meta information. The actual SDS implementation allows to use other environments as well to make the concept generally usable. This however requires to parametrise `RWShared` with the environment, which is not done in this paper to provide better readability.

The following higher order function is the basic way to create SDSs:

```
createShare :: (*IWorld → (r,*IWorld)) (w *IWorld → *IWorld) → RWShared r w
```

The programmer often does not have to provide read and write functions, but can make use of a number of pre-defined functions to create SDSs. Here, some creation functions for different kinds of sources are given:

```
// data stored in a file, functions for serialisation have to be provided
sharedFile   :: Path (String → a) (a → String) → Shared a

time         :: ROShared Timestamp // the current time
random       :: ROShared Int       // a stream of random numbers
null         :: WOShared a         // thrash can, writing has no effect
```

These examples show that SDSs 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.

Web application frameworks, like *iTask*, can provide their meta data in this way as well:

```
currentUser :: Shared User   // the current user of the task
allUsers    :: Shared [User] // all users registered in the system
```

4.2 Shared Data Source Combinators

SDSs can not only be defined in terms of basic operations, but also as composition of other SDSs. One reason for this is that a task can use SDSs and use them to define other ones, as discussed in Section 2. Another reason is that it is more convenient to use existing functions creating SDSs and combine the existing SDSs. For example, an SDS giving the current time and one for writing to a file can be combined, rather than implementing the retrieval of time and file access on low level.

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 time, 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:

```
mapRead  :: (r    → r`)      (RWShared r w) → RWShared r` w
mapWrite :: (w` r → Maybe w) (RWShared r w) → RWShared r  w`
```

Example 1. Turning a data source in a read-only one is a special case of projecting the write type:

```
toReadOnly :: (RWShared r w) → ROShared r
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) → RWShared r` w`
```

A functional *lens* [1] 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.

Example 2. 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)) → Shared a
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.

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:

```
(⋊⋉) infixl 6 :: (RWShared rx wx) (RWShared ry wy) → RWShared (rx,ry) (wx,wy)
```

Composition can be applied repeatedly to build arbitrary large composed data sources. Since the type remains `RWShared`, composition is completely transparent for the user. Reading and writing are performed from left to write. Operations on composed sources are not atomic for the outside world.

Example 3. 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]) (ROShared [a]) → ROShared [a]
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 3.

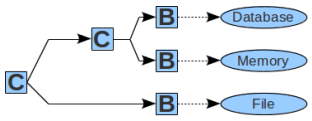


Fig. 3: Composed Sources (C) consisting of Basic Source (B) Referring to Different Kinds of Storages

The concepts are powerful enough to put for instance a *symmetric lens* [3] between two data sources. A derived combinator can be defined[4]:

```
symLens :: (a b → b) (b a → a) (Shared a) (Shared b) → (Shared a, Shared b)
```

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 these two data sources. For the user of such a data source this is completely transparent, but if one is changed the other is changed as well.

General Read and Write Combinator There are situations in which one wants to determine the SDSs to be used dynamically based on the value of some other SDS. For this we provide a more general read combinator. It first reads an SDS and uses its current value to dynamically compute another SDS which's value is given as result. The write type remains unchanged:

```
(>?>) infixl 6 :: (RWShared r w) (r → (RWShared r' wx)) → RWShared r' w
```

For writing there is a general combinators, too:

```
(>!>) infixl 6 :: (RWShared r w') (w → RWShared r' wx, w r' → [WriteShared]) → RWShared r w
:: WriteShared =∃r w: Write w (RWShared r w)
```

Since writing possibly first requires to read data, the value to write is used to determine an SDS to read (first tuple element of second argument). Then the value read from this SDS and the value to write are used to dynamically compute a number of write operations using arbitrary SDSs (second tuple element of second argument). To make it possible to combine SDSs with different write types in one list, the type `WriteShare` is used. It hides the read and write type of an SDS using existential quantification.

[4] A simple notation without *complement* (see [3]) is used here for pragmatic reasons, without consequences for the expressiveness.

5 Semantics

In this section we define the semantics of shared data sources. Thanks to the power of functional programming there is hardly a difference between the semantics and the implementation. For this reason we use *Clean* code to define the semantics. Whenever we abstract from implementation details we indicate that clearly.

The semantics is not given in this extended abstract, but is left for the full paper.

6 Related Work

Many solutions already exist for abstracting from the actual implementation of data sources. For instance, databases provide tables or key-value pairs abstracting from the actual storage of data. *UNIX* files can represent arbitrary data sources. In object-oriented programming (OOP) certain patterns are used, providing an interface for accessing data, hiding the actual source. Getters and setters are used to control access. Those solutions however are not as general as SDSs. First, unlike databases they allow to abstract from all possible implementations. Second, they use the type system to allow for very general abstractions. *UNIX* files only provide untyped strings and databases only a limited number of abstractions like tables and records. Third, they provide a more flexible way for access control using read and write types and combinators. OOP allows for similar access control, but source composition cannot be done in a convenient way due to the lack of higher-order functions.

Lenses [1] can be seen as a solution to access only parts of a data structure in a certain way in the functional programming world.

LINQ [5] 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. Similar approaches exist for functional languages [4, 2]. This provides efficient mechanisms for some common cases, but is not as general as SDSs, allowing to use arbitrary functions.

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. 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.

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.

There are situations in which atomicity of operations on composed sources plays a role. An operation specified on an SDS might be intended to be atomic and it should remain possible to abstract from whether the SDS is a basic or composed one. Since the concept of SDSs tries to capture all kinds of data sources this remains a challenge. Not for each kind of data source, for instance the current time, it is clear what it means to atomically compose operations on them with operations on other data sources. Also different implementations, for example different database system, use very different mechanisms to enforce atomicity, which would have to be combined in some way.

References

1. 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.
2. 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.
3. 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.
4. Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35:109–122, December 1999.
5. 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.
6. 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.
7. Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. Submitted to International Conference on Functional Programming, ICFP '12, 2012.