Uniform 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. One commonly has to deal with all sorts of data sources such as memory, files, and databases when developing complex applications. Different kinds of data sources have different properties and realisations and therefore typically provide different interfaces for storing and retrieving data. As a consequence, the code one writes in an application depends on the actual resource used, which leads to non-reusable code and huge refactoring effort in case one decides to store the information differently.
In this paper we propose to use an abstract data type called *uniform data sources* (UDS) which makes it possible to deal with all kinds of different data sources in a uniform way. UDSs enable programmers to separate the definition and usage of data sources, and provide a uniform interface resulting in highly reusable code. UDSs are composable: new data sources can be defined in terms of existing ones. Reading and writing to a UDS maybe of different type which can be used to statically enforce complex access control, allowing to restrict what components can read and write to what is necessary for their functionality. Full-access, read-only access, write-only access are special cases of this mechanism. More complex access restrictions can be defined as well. We have implemented a UDS library for *Clean* which is successfully used in the *iTask* system and applied in *iTask* applications.

1 Introduction

Complex applications, such as for instance multi-user web applications, 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 anymore after it is processed. Furthermore, applications probably use meta data to handle the different users and processes currently performed.

Different kinds of data sources (memory, files, databases) typically use different mechanisms to retrieve and store data. As a consequence, the interfaces which are provided differ and depend on the type of data source being used. The code one has to write to access a certain kind of data source is therefore commonly not reusable. When the data is stored differently, the corresponding code

has to be adjusted. This may require much effort, especially if in the application logic the definition and the use of the data sources are intertwined.

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 *uniform data sources* (UDS) as a uniform way for dealing with different kinds of data sources. We furthermore want to restrict the access provided by UDS, meaning that we want to be able to restrict the type of data which can be read or written. This has two advantages. It ensures that certain data cannot be changed by mistake and fewer assumptions need to be made about the context in which a piece of code is used.

This leads to highly reusable code and allows for a layered software architecture. When using a data source in a higher level layer only has to know that there is a source providing and receiving information of a certain type, completely abstracting from where the data is actually stored and how access to it is realised. On the other hand, one needs to define what reading and writing data actually means in lower layers, in such a way that one can completely abstract from the way the data is actually being used.

We use an abstract data type and the expressive power of a functional language to achieve this goal. The uniform data source references introduced in the paper have the following advantages:

1. UDSs are *uniform*, i.e. they can represent arbitrary kinds of data storages, like memory, files or databases, and even arbitrary combination of data sources. They make it possible that all data sources can be used in the same way, even though the actual operations are mapped to e.g. operations on memory, or on XML encoded files. Actually, even data sources other than data storages 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.
2. *Access control is statically enforced* by 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.
3. 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. 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 the provided and required data sources.

The UDS concept is generally applicable. It has been implemented in *Clean* and is offered as a general library such that it can be used in any (*Clean*) application. It has proven to be very powerful in combination with the *iTask* system [14] that is used for developing multi-user distributed applications. In such applications typically many different data sources are being used (shared memory, files, databases, time, date, logged-in users, JSON encoded client server

communication) which now can all be addressed in a uniform way. UDSs have become one of the pillars of the *iTask* system and UDSs are used in large *iTask* applications such as described in [10] and [11].

The remainder of this paper is organized as follows: The two concerns usage and definition are discussed in detail in Sections 2 and 3, respectively. We use the functional language *Clean* for code examples. The semantics of the proposed UDSs are defined in Section 4. Our approach can in principle also be used for shared data sources in a concurrent environment. This is explained in Section 5. Related work is discussed in Section 6 and conclusions are drawn in Section 7.

2 Using Uniform Data Sources

This section deals with the first of the two concerns discussed in this paper. From the perspective of a user of a UDS, the most important property of a UDS is that it allows to abstract from the actual definition of a data source. This can be achieved by using an *abstract data type* functioning as a reference to the actual source. The reference itself cannot be inspected and only a pre-defined set of operations can be performed on it, as defined below and in Section 3.

UDSs provide access control. This basically means that what can be read from a UDS is not necessarily the same than what can be written. Each UDS therefore has two type parameters indicating the type of data which can be read from and can be written to the source, respectively. Concretely, a UDS is represented by the abstract type `RWUDS`:

```
:: RWUDS 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 `UDS` for this. It is also possible to express read-only and write-only data sources as special cases, where the type `Void` is used to express the inability to write c.q. read[1]:

```
:: UDS   a :== RWUDS a a
:: ROUDS r :== RWUDS r Void
:: WOUDS w :== RWUDS Void w
```

Basic operations provided on a UDS are for reading (`get`) and writing (`put`)[2]:

```
get    ::   (RWUDS r w) *World → (r, *World)
put    :: w (RWUDS r w) *World → *World
```

To obtain access to the outside world, *Clean* uses uniqueness typing instead of a monadic approach [1]. All side-effecting functions work on a uniquely attributed (*) type `World`. The functions `get` and `put` look rather straightforward, yet arbitrarily complex data structures, possibly stored in different media, can be read and written with them in one go.

Example 1. As running example in this paper we look at a typical software component which is part of a web application. It allows users to respond to an

[1] `Void` is *Clean*'s unit type and:==denotes a type synonym.
[2] In *Clean* notation the function `get` has two and `put` has three parameters.

appointment invitation. It is part of an agenda application, where users can suggest an appointment and invite a number of other users. The users can respond to this invitation by accepting or rejecting it. To make this decision, information about the appointment has to be provided to the user. This includes a description of the appointment, who initiated it and current responses of other users.

First, we define a record type `Invitation` which contains all the information needed for inviting a user:

```
:: Invitation =
  { description :: String,   createdBy :: UserDescription
  , when        :: DateTime, timeUntil :: TimeInterval
  , ownResp     :: ParticipantResp, otherParticipants :: [(UserDescription, ParticipantResp)] }
:: UserDescription = { firstName :: String, lastName  :: String }
:: ParticipantResp = NoResponse | Accepted | Rejected String | Tentative String
```

An `Invitation` to an appointment obviously includes a date and time and how much time there is to respond. To display information about the users being invited, `UserDescription`s are needed. To keep it simple they only contain the user's names. The response from an invited user to the invitation is represented by `ParticipantResp`s. If not accepting an invitation the user can input a string containing an explanation of the reason for this.

We assume that the data which plays a role in this example, is stored in different media. The data representing the invitation is constructed in memory of the application. But to accomplish this, part of the information such as the descriptions of the users invited need to be retrieved from a database, while the current date and time is obtained from the operating system.

When a user responses to an invitation, we want that only that part of the invitation can be changed. We can realise this by offering a UDS of type `RWUDS Invitation ParticipantResp`, such that the information of type `Invitation` can be read with one `get` and displayed after which the response of type `ParticipantResp` can be written with a `put`.

3 Defining Uniform Data Sources

To make it possible to read and write complex data from multiple sources in one go, we combine the different data sources involved in a layered way. Figure 1 illustrates a possible layered architecture for our running example. In this layered architecture we distinguish creation of basic data sources, and offer combinators to compose data sources from others.

3.1 Creating Basic Uniform Data Sources

When a UDS is created, it is defined what reading and writing of the data of that particular source actually means. This means that two functions have to be provided, one for reading, one for writing. Given these two functions, a UDS can be created using the following function:

```
createUDS :: (*World → (r,*World)) (w *World → *World) → RWUDS r w
```

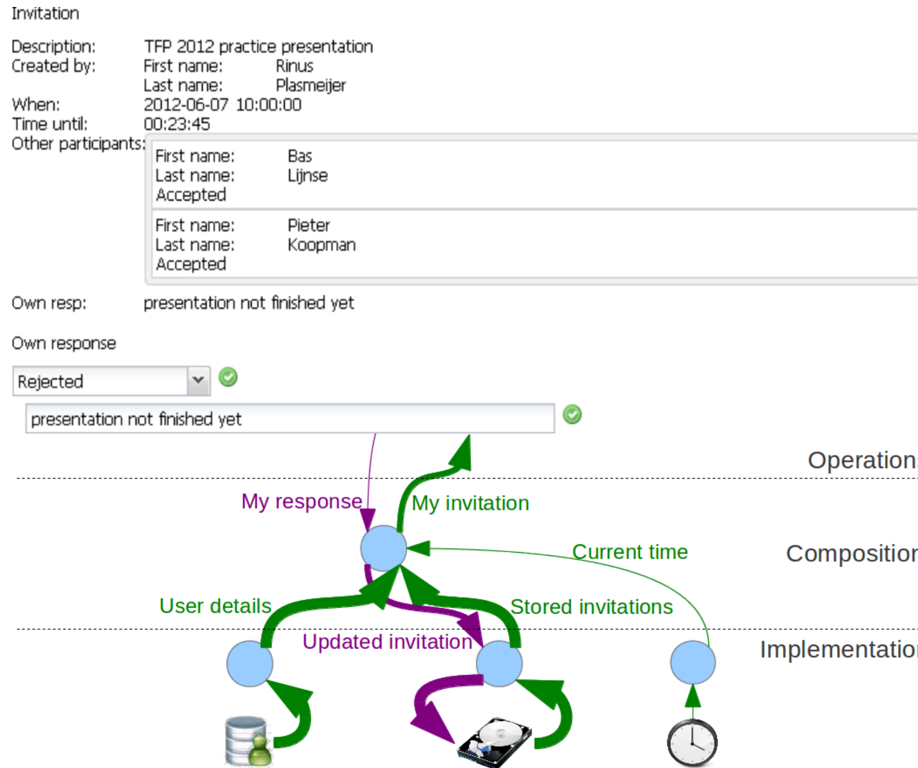

Fig. 1: Layers of the example agenda component

Hence, a uniform data source can simply be created by defining how the data can be obtained from and has to be stored back to the outside world. The complexity of these two functions depend on the actual medium where the data is stored. Any thinkable UDS can be created in this way.

To make life easier, the programmer can make use of a number of pre-defined functions we have defined in the library. One very simple example is a UDS providing a constant value:

```
constUDS :: a → ROUDS a
constUDS v = createUDS (λworld → (v,world)) (λ_ world → world)
```

To store information in a file, for instance, the following library function is provided:

```
fileUDS :: Path (String → a) (a → String) → UDS a
```

The only thing the programmer has to provide is a path, and an encode and a decode function. Data sources which do not behave like ordinary data stores are provided as well:

```
curDateTime :: ROUDS DateTime // the current time
random      :: ROUDS Int      // a stream of random numbers
null        :: WOUDS a        // thrash can, writing has no effect (like Unix /dev/null)
```

In the *iTask* system, UDSs are used to provide domain specific data sources, such as data sources for dealing with its internal administration, like for the management of the *iTask* users:

```
currentUser         :: ROUDS User
allUserDescriptions ::   UDS [(User,UserDescription)]
```

The type `User` is an abstract type representing a user of the system in some way, for instance by an identification number. This allows to abstract from the description of users which are provided by another data source.

These examples show that UDSs 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 can be seen as a uniform data source and defined as such.

3.2 Uniform Data Source Combinators

UDSs cannot only be defined in terms of basic operations, but also as composition of other UDSs. In this way a layered architecture can be achieved.

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 access restrictions are needed. However, for software engineering reasons, one also 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 a data source, can be used:

```
(>?@) infixl 6 :: (RWUDS r w) (r     → r‘)       → RWUDS r‘ w
(>!@) infixl 6 :: (RWUDS r w) (w‘ r → Maybe w) → RWUDS r  w‘
```

With the `>?@` operator, the read type of a UDS can be changed by providing a function from the old to the new read type. With the `>!@` operator, the type to be written can be changed. A more sophisticated function has to be given here since values of the write type might contain less information than that of the read type. The data source's current value might therefore be necessary to construct a value of the old write type. Furthermore, writing is optional.

Because pure functional projections are used to map values to a different type, the behaviour of data sources after projection is not changed for the outside world.

Example 2. Turning a data source in a read-only one is a special case of projecting the write type:

```
toReadOnly :: (RWUDS r w) → ROUDS r
toReadOnly uds = uds >!@ (λ_ _ → Nothing)
```

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

```
(>?!@) infixl 6 :: (RWUDS r w) (r → r‘, w‘ r → Maybe w) → RWUDS r‘ w‘
(>?!@) uds (pr,pw) = uds >!@ pw >?@ pr
```

Our >?!@ combinator is a more general case of a functional *lens* as introduced by Hoffman et al. [2]. A lens is a well known method to update only a part of a larger data structure. When r = w and writing is obligatory, the type of >?!@'s first argument becomes (s → v, v s → s) which is a simple get/set notation of a lens.

Example 3. A data source containing a tuple can be changed into a data source which only allows to access the tuple's first element using a simple lens:

```
fstLens :: (UDS (a,b)) → UDS a
fstLens tuple = tuple >?!@ (fst, λx (_,y) → Just (x,y))
```

Example 4. We finally come back to our running example. Assume we have a store for a single invitation. Access control can be achieved by applying a write projection to the UDS, turning it into a UDS of the type required by the component described in Example 1[3]:

```
restrWriteAcc :: (UDS Invitation) → RWUDS Invitation ParticipantResp
restrWriteAcc uds = uds >!@ (λresp inv → Just {inv & ownResp = resp})
```

This example illustrates a combination of both concerns. First, the projection uses an existing UDS abstracting from how the data is actually stored. Second, the projection defines a UDS which is then passed to the layer actually using it.

Static 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 actually combines a number of existing sources. To achieve this a combinator for composing two data sources (>+<) is introduced:

```
(>+<) infixl 6 :: (RWUDS rx wx) (RWUDS ry wy) → RWUDS (rx,ry) (wx,wy)
```

Example 5. A concatenation operation for two read-only lists yielding a new single list can be defined like this:

```
concat :: (ROUDS [a]) (ROUDS [a]) → ROUDS [a]
concat x y = (x >+< y) >?!@ (λ(x,y) → x ++ y, λ_ _ → Nothing)
```

Composition can be applied repeatedly to build arbitrary large composed data sources. Since the resulting type is a RWUDS again, composition is completely transparent for the user. Internally, reading and writing is performed from left to right. Composition makes it possible to combine different kinds of data sources. An example of a composed source using different kinds of storages is given in Figure 2.

The combination of composition and projection is very powerful. Our combinators are powerful enough to put for instance a *symmetric lens* [7] between two data sources[4]:

[3] The notation {rec & field = newValue} updates a field of a record.

[4] A simple notation without *complement* (see [7]) is used here for pragmatic reasons, without consequences for the expressiveness.

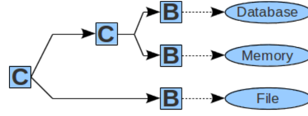

Fig. 2: Composed sources (C) consisting of basic sources (B) using different media

```
symLens :: (a b → b) (b a → a) (UDS a) (UDS b) → (UDS a, UDS b)
symLens putr putl udsA udsB = (newUdsA, newUdsB)
where  udsBoth  = udsA >+< udsB
       newUdsA = udsBoth >?!@ (fst, λa (_,b) → Just (a,putr a b))
       newUdsB = udsBoth >?!@ (snd, λb (a,_) → Just (putl b a,b))
```

Such lenses are used to keep two data sources consistent if only one of them is changed. Type of the arguments and resulting sources remains the same, but a connection between the sources is established. For the user of such a data source this is completely transparent.

Example 6. In our running example, one would probably want store the information about the planned meeting and accepted invitations. Assuming there is a store for that:

```
storedInv :: UDS StoredInv

:: StoredInv = { description :: String,   createdBy    :: User
               , datetime    :: DateTime, participants :: [(User, ParticipantResp)] }
```

The time until the event is not stored as it has to be determined dynamically. In the store it is administrated which users have accepted the invitation so far. There is no distinction between other participants and the response which can be updated as this depends on the current user. References of type `User` are used instead of descriptions. Those have to be retrieved from the already discussed source `allUserDescriptions`. In this way it is ensured that any change in the user database is immediately reflected. We assume there is a function `toDescr :: User [(User,UserDescription)] → UserDescription` to replace a reference of type `User` by the corresponding description.

From `storedInv` together with the current time and the user database a UDS of the type required by the component described in Example 1 can be created:

```
invitation :: RWUDS Invitation ParticipantResp
invitation =      (storedInv >+< curDateTime >+< currentUser >+< allUserDescriptions)
             >?!@ (projR,projW)

projR :: (((StoredInv, DateTime), User), [(User,UserDescription)]) → Invitation
projR (((inv, curDateTime), curUser), descrs) =
    { description = inv.description, createdBy = toDescr inv.createdBy descrs
    , when = inv.datetime,           timeUntil = inv.datetime - curDateTime
    , ownResp  = hd [r \\ (u,r) ← inv.participants | u == curUser]
    , otherParticipants = [(toDescr u descrs, r) \\ (u,r) ← inv.participants | u ≠ curUser] }

projW :: ParticipantResp (((StoredInv, DateTime), User), [(User,UserDescription)])
      → Maybe (((StoredInv, Void), Void), [(User,UserDescription)])
projW resp (((inv, _), curUser), users) = Just (((
    {inv & participants = [(curUser,resp) : filter (λ(u,_) → curUser ≠ u) inv.participants]}
  , Void), Void), users)
```

The write projection `projW` stores the invitation response at the proper place in the stored invitation. A possible previous response of the current user is outdated and filtered out of the list and the new one is added in front.

Dynamic Composition & Projection Core Combinators There are situations in which static composition is not desirable. The combination of first combining UDSs providing all information possibly needed and restricting access using projections is actually powerful enough to cover virtually all cases, but this solution is potentially highly inefficient. The UDS `allUserDescriptions` for instance gives a list of all user descriptions. This becomes inefficient if there is a large number of users in the system and one is interested in only the descriptions of some of them. A function for generating a UDS efficiently retrieving the description of a given user can be defined. This function can for example perform an efficient database query retrieving only the user description of the user given:

```
userDescription :: User → UDS UserDescription
```

The problem is that this function cannot be used together with the combinators discussed so far, since they require a UDS and not a function generating one.

This can be solved by combinators allowing for dynamic composition and projection. We consider those combinators as core combinators as all previously defined combinators can be expressed using them. The core read combinator >?> first reads a UDS and uses its current value to dynamically compute another UDS which's value is given as result. As before, the write type remains unchanged:

```
(>?>) infixl 6 :: (RWUDS r w) (r → (RWUDS r‘ wx)) → RWUDS r‘ w
```

For writing there is a more general core combinator >!>, too:

```
(>!>) infixl 6 :: (RWUDS r w) (w‘ → RWUDS r‘ wx, w‘ r‘ → [WriteUDS]) → RWUDS r w‘

:: WriteUDS =∃r w: Write w (RWUDS r w)
```

As before, writing possibly means that one first has to read data since the structure to write may contain not enough information to determine where to write it. In analogy with >?>, the value to write is used to determine a UDS to read from (first tuple element of second argument). This is quite flexible because in this way the write combinator does neither depend on nor change the read type of the UDS given as first argument. The value read from the dynamically determined UDS and the value to write are used to dynamically compute a number of write operations on arbitrary many UDSs (second tuple element of second argument). To make it possible to combine UDSs with different write types in one list, the type `WriteUDS` is used. It hides both the read and write type of a UDS using existential quantification.

Example 7. The core combinators can be used to define a UDS providing the current user's description in an efficient way, given the UDS `currentUser` and the function `userDescription`:

```
currentUserDescription:: UDS UserDescription
currentUserDescription =
        currentUser
    >?> userDescription
    >!> (const currentUser, λdescr curUser → [Write descr (userDescription curUser)])
```

The order in which both combinators are applied to the UDS does not matter. As discussed, the combinators are designed in such as way that they depend on and change solely either the read or the write type.

Example 8. It is finally shown how the UDS for the running example can be defined without the need to read all user descriptions:

```
invitation :: RWUDS Invitation ParticipantResp
invitation = (storedInv >+< curDateTime >+< currentUser) >?> readR >!> (readW, write)

readR :: ((StoredInv, DateTime), User) → RWUDS Invitation (Void, UserDescription)
readR ((inv, curDateTime), curUser) =
  ( mapList inv.participants (λ(u,r) → userDescription u >+< constUDS r)
    >+<
    userDescription inv.StoredInv.createdBy )
  >?@ λ(participants, creator) →
         { description  = inv.description, otherParticipants = participants
         , when         = inv.datetime,    timeUntil  = inv.datetime - curDateTime
         , createdBy = creator, ownResp = hd [r \\ (u,r) ← inv.participants | u==curUser] }

readW :: ParticipantResp → RWUDS (StoredInv, User) (StoredInv, Void)
readW _ = storedInv >+< currentUser

write :: ParticipantResp (StoredInv, User) → [WriteUDS]
write resp (inv, curUser) = [ Write { inv & participants =
    [(curUser,resp) : filter (λ(u,_) → curUser ≠ u) inv.participants] } storedInv ]
```

The function `mapList` is an auxiliary combinator, which can easily be defined using the `concat` combinator of Example 5:

```
mapList :: [a] (a → RWUDS r w) → ROUDS [r]
mapList l f = combine l (constUDS [])
where  combine []    uds = uds
       combine [e:l] uds = combine l (concat uds (toReadOnly (f e >?@ λx → [x])))
```

4 Semantics

In this section we define the semantics of uniform data sources. We use *Clean* code as formalism to give the semantics. This has the advantage that the behaviour is defined unambiguously while it also gives a good hint for an actual implementation. Such executable semantics have already been used to define the behaviour of the *iTask* system earlier [8, 14].

Implementation details we have abstracted from are the following. For simplicity reasons we define all the operations on the `*World` environment. In the actual implementation operations are overloaded such that different environment can be used. For every concrete media being used (memory, file, database) one needs to have primitives for reading and writing from that media. Depending on the kind of media used, the implementation effort can be straightforward but it can also be very complicated involve a lot of work when efficiency is a concern. We abstract from all this here in the semantics and assume that `readFromMedia :: *World → (r,*World)` and `writeToMedia :: w *World → *World` are defined for any type and any media. Another simplification is that we do not deal with error handling and assume operations always succeed.

4.1 Basic UDSs

We first give the semantics of basic UDSs. A RWUDS is an algebraic data type, defined as follows:

```
:: RWUDS r w    = BasicUDS (*World → *(r,*World)) (w *World → *World) | ...
```

In the algebraic data type the data constructors indicate with which type of UDS we are dealing with: a basic one or a composed one. Basic UDSs are represented by the constructor `BasicUDS`, the other options are explained below.

The create function creates a basic UDS and simply passes its arguments to the constructor:

```
createUDS :: (*World → (r,*World)) (w *World → *World) → RWUDS r w
createUDS readF writeF = BasicUDS readF writeF
```

The basic operations get and put can now straightforwardly be defined for the case that we have to deal with a basic UDS:

```
get :: (RWUDS r w) *World → (r,*World)
get (BasicUDS read _) world = read world

put :: w (RWUDS r w) *World → *World
put w (BasicUDS _ write) world = write w world
```

The operations become more complex for the composed cases discussed in the remainder of this section.

4.2 Core Combinators

All compositions can be expressed in terms of two basic combinations, as already discussed. The semantics given here and the actual implementation are identical.

```
:: RWUDS r w    = BasicUDS (*World → *(r,*World)) (w *World → *World)
    | ∃rx wy:     ComposedRead  (RWUDS rx w) (rx → RWUDS r wy)
    | ∃r‘ w‘ w‘‘: ComposedWrite (RWUDS r w‘) (w  → RWUDS r‘ w‘‘) (w r‘ → [WriteUDS])

:: WriteUDS =∃r w: Write w (RWUDS r w)
```

The two additional constructors, `ComposedRead` and `ComposedWrite` representing read and write composition as discussed in Section 3.2. The types of all intermediate results are hidden using existential type quantification.

The core combinators do nothing more than passing their arguments to those constructors:

```
(>?>) infixl 6 :: (RWUDS rx wx) (rx → RWUDS ry wy) → RWUDS ry wx
(>?>) uds read = ComposedRead uds read

(>!>) infixl 6 :: (RWUDS r w‘) (w → RWUDS r‘ w‘‘, w r‘ → [WriteUDS]) → RWUDS r w
(>!>) uds (read,write) = ComposedWrite uds read write
```

The get and put operations have to be extended to cover the two new cases. We start with `get`[5]:

```
get (ComposedRead uds cont) world
    # (x, world) = get uds world
    # (y, world) = get (cont x) world
    = (y, world)
get (ComposedWrite uds _ _) world = get uds world
```

[5] In *Clean* the `#` denotes a *let*.

In the case the UDS is a read composition, the original UDS is read first by invoking `get` recursively, then the UDS resulting from the continuation function is read. The case for write composition is trivial, since the behaviour is not affected by read composition.

The `put` operation has to be extended similarly:

```
put w (ComposedRead uds _) world = put w uds world
put w (ComposedWrite _ readOp writeOp) world
    # (r, world) = get (readOp w) world
    # writes     = writeOp w r
    = seqSt (λ(Write w uds) → put w uds) writes world
```

4.3 Derived Combinators

All derived combinators used in this paper can be expressed in terms of the core combinators. The read and write projection can be expressed by the core combinators in this way:

```
(>?@) infixl 6 :: (RWUDS r w) (r → r‘) → RWUDS r‘ w
(>?@) uds p = uds >?> λr → constShare (p r)

(>!@) infixl 6 :: (RWUDS r w) (w‘ r → Maybe w) → RWUDS r w‘
(>!@) uds p = uds >!> (const uds, λw‘ r → maybe [] (λw → [Write w uds]) (p w‘ r))
```

Static composition can be expressed by a combination of the read and write combinator:

```
(>+<) infixl 6 :: (RWUDS rx wx) (RWUDS ry wy) → RWUDS (rx,ry) (wx,wy)
(>+<) udsx udsy =         (udsx >?> λrx → udsy >?@ λry → (rx,ry))
                       >!> (const (constShare Void), λ(wx,wy) _ → [Write wx udsx, Write wy udsy])
```

5 Towards Shared Uniform Data Sources

As explained before, UDSs have been implemented and are offered in a special *Clean* library. The library is general applicable and can be used for the development of any (*Clean*) application. The library is heavily used in the *iTask* system. The *iTask* system offers a domain specific language for the development of distributed multi-user web-applications. *iTask* applications are used in practice in the home heath care domain (see [15]) while larger applications are being prototyped (see [10] and [11]).

In both the *iTask* system itself as well as the applications made with it, many different data sources are commonly being used, and UDSs turned out to be very helpful. However, UDSs are here not only successfully being used to abstract from the concrete data sources being used, they also turned out to be very useful for defining data which is being shared between the different tasks one is working on distributively. This specific use is rather straightforward to accomplish here because of certain characteristics of the *iTask* implementation. No synchronisation of operations on data shared between tasks is necessary, because task evaluation is done single-threadedly and all data sources are under the exclusive control of the *iTask* server. Hence, atomic access to such UDSs is guaranteed.

We have extended the UDS library to support shared use of UDSs for the general case where applications do *not* have exclusive control over the data sources being used. However, this only works for certain data sources. When sharing data between arbitrary applications, synchronisation is crucial. A first reason is that one might want to perform several operations on a number of UDSs atomically. The most important reason is that when using a UDS one should be able to completely abstract from whether it is a composed one or a basic one. A single operation on a UDS should always be one atomic action for the outside world no matter how it is actually constructed.

For certain types of data sources, this is hard to realise. First, synchronisation mechanisms of several kinds of data storages have to work together and need a uniform interface. It is not possible to use the transaction mechanism of data stores if one wants to achieve an atomic operation on multiple of them at the same time. This is only possible with more low level primitives like locking and unlocking, which possibly excludes data storages not supporting this kind of primitives. Excluding data storages not supporting sufficiently powerful synchronisation mechanisms might be a price we have to pay for getting a uniform interface.

The implementation making use of those locking primitives has to be able to deal with the dynamic nature of UDS composition, without introducing deadlocks. *Software transactional memory* (STM) [6] provides a solution for this. We implemented STM for UDSs for which the basic implementation supports a simple lock and unlock operation. This works fine, but we have not been able yet to measure the performance of this approach compared to using the transaction mechanisms provided by the data storages themselves.

The most severe problem of guaranteeing atomic access to a UDS is that not always clear what it semantically means when certain data sources are being used. For instance, what does it mean if we atomically want to read a UDS containing the current time as one of its components? Which time should we use? To extend the semantics of STM such that those kind of sources can be represented by our abstraction, while still providing atomicity guarantees, remains future work.

So, we have an implementation which supports shared use of UDSs between arbitrary applications. Atomicity is guaranteed using STM techniques in the implementation. It is not always as efficient as it could be and it is also not always clear how to handle special cases like reading the time and current temperature. More research is needed to solve these issues.

6 Related Work

One area of related work are query languages allowing to abstract from the actual data storage. Those languages however assume that data is stored in a certain format. *XML-QL* [4] is an example for a language to retrieve data from and generate new *XML* documents. In the context of relational data, a lot of work has been done about embedding database accesses into programming languages.

The technique of writing queries in a functional language and then compile it to SQL was pioneered by *Kleisli* [16]. Other approaches are *HaskellDB* [9] and *Database-Supported Haskell* [5]. Another more high-level approach allows to completely abstract from which queries are done, by generically mapping databases to types based on a data model [12]. The goal of all those solutions is to let programmers abstract from performing database access and provide type safety. The work aims at accessing tables storing collections of records in databases.

The UDS approach allows to work on arbitrary types, the data provided by an UDS could for instance be a tree. This also requires a more flexible way of defining data accesses and performing operations, based on arbitrarily complex functions and dynamic composition. The disadvantage of the flexible UDS approach is that it does not allow for optimisations, as performed by some of the above mentioned database languages. However, special query operations for data stored in a restricted structure using a certain database system, could be built on top of the flexible UDS primitives.

The most crucial difference with the mentioned work on integrating data access into programming languages is that UDS aims at providing a composable abstraction for data sources, allowing to structure software in a layered way and provide static access control.

An approach to provide uniform data sources which is more similar to our concept of UDSs is *LINQ* [13]. It provides a solution for abstracting from the actual implementation of a data source. Supported data sources are not restricted to databases, but can be objects from the host programming language or *XML* data as well. Similar to the UDS abstraction the type IEnumerable<T> is used for abstraction. Objects of this type are also composable using joins.

As for the discussed database languages *LINQ* works on collections only. UDS composition is done using functions, together with the dynamic core combinators this provides a greater flexibility. Static access control finally is not provided by the IEnumerable<T> interface, data read and written is of the same type.

Links [3] combines database access, defining the server logic and the client user interface of a web application in a single language. In this sense it is similar to *iTask* in combination with UDSs. Task descriptions in *iTask* are however on a much higher level, as they allow the programmer to abstract from the details of user interaction. In the same way UDSs provide a higher level of abstraction, as they can represent any kind of data source.

Lenses [2] can finally be seen as a theoretical concept underlying static projections on data sources. The static projection combinators for UDSs are more flexible, partially because of pragmatic reasons. Not writing, for instance, is more efficient than writing an unchanged value back. The main reason is that the concept of using different read and write types requires more flexible projections. The price of this flexibility is that *well-formed lenses* probably have more theoretical properties allowing to reason about composition. We however did not explore the theoretical properties of UDS projections.

7 Conclusions & Future Work

We introduced uniform data sources, a way to abstract from concrete data sources and compositions of them such that one can use them when writing code in a uniform way. Code using data sources becomes highly reusable.

Due to the offered abstraction, 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 different types for reading and writing makes it possible to define access restrictions statically. With the core combinators complex data combinations can be defined and it is possible to compute the location of the actual data dynamically.

The UDS concept can be implemented in any language and is general applicable. It is implemented in *Clean* and available as library. It is successfully used in practice in the task-oriented programming framework *iTask*.

UDS can also be used for shared data sources. Atomicity is guaranteed using STM techniques in the implementation. However, the implementation only supports certain data types. More research is needed if one wants to be able to deal with arbitrary data sources in an efficient way. Since the concept of UDSs tries to capture all kinds of data sources it is difficult to guarantee atomicity for all cases without scarifying efficiency. For certain data sources, such as the current time, it is unclear what it means to atomically compose operations on them with operations on other data sources. Another problem is that 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. Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Rudrapatna Shyamasundar, editor, *Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science, FSTTCS '93, Bombay, India*, volume 761 of *LNCS*, pages 41–51. Springer-Verlag, 1993.
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. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, November 2006. Springer-Verlag.
4. Alin Deutsch, Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for xml. *Computer Networks*, 31(11-16):1155–1169, 1999.
5. 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.

6. 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.
7. 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.
8. Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In Sven-Bodo Scholz, editor, *Proceedings of the International Symposium on the Implementation and Application of Functional Languages, IFL '08, Hertfordshire, UK*, pages 53–64. University of Hertfordshire, 2008.
9. Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35:109–122, December 1999.
10. Bas Lijnse, Jan Martin Jansen, Ruud Nanne, and Rinus Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In David Mendonca and Julie Dugdale, editors, *Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management, ISCRAM '11*, Lisbon, Portugal, May 2011. ISCRAM Association.
11. Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. Incidone: A task-oriented incident coordination tool. In Leon Rothkrantz, Jozef Ristvej, and Zeno Franco, editors, *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM '12*, Vancouver, Canada, April 2012. to appear.
12. 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.
13. 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.
14. Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. Paper accepted for publication in the proceedings of the International Conference on Principles and Practice of Declarative Programming, PPDP '12, 2012.
15. Maarten van der Heijden, Bas Lijnse, Peter Lucas, Yvonne Heijdra, and Tjard Schermer. Managing COPD exacerbations with telemedicine. In *13th Conference on Artificial Intelligence in Medicine, AIME '11*, volume 6747 of *LNCS*, pages 169–178, Bled, Slovenia, July 2011. Springer-Verlag.
16. Limsoon Wong. Kleisli, a functional query system. *J. Funct. Prog*, 10, 1998.