A Transaction Framework for Web Applications in Haskell

Master Thesis

Florian Micheler
Matrikel-Nummer 912583

Betreuer  Priv.-Doz. Dr. Frank Huch
Written in Markdown (with X\LaTeX–template) and compiled to PDF-file using pandoc ver. 1.12.3.3
http://johnmacfarlane.net/pandoc/

**Fonts used in this thesis**
Main font: Linux Libertine, http://www.linuxlibertine.org
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Kiel, 9. Juni 2014

___________________________
Florian Micheler
## Contents

Abstract ............................................. 7

1 Fundamentals ...................................... 8
   1.1 Transactions .................................. 8
       1.1.1 ACID Principle ........................... 8
       1.1.2 User Interaction within Transactions .... 8
   1.2 Deadlocks ..................................... 9
   1.3 Software Transactional Memory (STM) ........ 9
       1.3.1 STM in Haskell .......................... 10
   1.4 Web Application Frameworks .................. 13
   1.5 Sessions ...................................... 13

2 Happstack ........................................ 15
   2.1 Haskell Source with XML (HSX) ............... 15
       2.1.1 Template ................................ 16
   2.2 ACIDState .................................... 18
       2.2.1 Data-type for Persistent Storage ....... 18
       2.2.2 IxSet .................................... 19
       2.2.3 Set Operations ........................... 20
       2.2.4 Queries .................................. 22
       2.2.5 Global Lock ............................. 25
   2.3 Type-safe Forms ............................... 26

3 Integration of the Transaction Framework ........ 33
   3.1 Transaction Base .............................. 34
   3.2 The Trans Monad ................................ 37
   3.3 atomicTransactionPart ......................... 38
   3.4 Read Set and Write Set ......................... 39
   3.5 Status ........................................ 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6 Touched Values</td>
<td>40</td>
</tr>
<tr>
<td>3.7 Sequence- and Interaction-Diagram</td>
<td>42</td>
</tr>
<tr>
<td>3.8 Example: Edit User Transaction</td>
<td>44</td>
</tr>
<tr>
<td>3.8.1 Parsing Parameters</td>
<td>46</td>
</tr>
<tr>
<td>3.8.2 Transaction Instructions</td>
<td>48</td>
</tr>
<tr>
<td>3.8.3 HTML Construction</td>
<td>51</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.1 GHC Language Extensions</td>
<td>55</td>
</tr>
<tr>
<td>4.1.1 Record Wild Cards</td>
<td>55</td>
</tr>
<tr>
<td>4.1.2 Existential Quantification</td>
<td>55</td>
</tr>
<tr>
<td>4.2 The Monad</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1 The Monad’s State</td>
<td>58</td>
</tr>
<tr>
<td>4.3 atomicTransactionPart</td>
<td>58</td>
</tr>
<tr>
<td>4.3.1 runcode</td>
<td>61</td>
</tr>
<tr>
<td>4.4 Functions of Trans–monad</td>
<td>68</td>
</tr>
<tr>
<td>4.5 Timeout Handler</td>
<td>73</td>
</tr>
<tr>
<td>4.6 Removing Sessions</td>
<td>75</td>
</tr>
<tr>
<td>4.7 Clean up Touched Values</td>
<td>76</td>
</tr>
<tr>
<td>4.8 Overview MVars</td>
<td>78</td>
</tr>
<tr>
<td>4.8.1 Removing sessions</td>
<td>80</td>
</tr>
<tr>
<td>4.8.2 Creating and reusing sessions</td>
<td>80</td>
</tr>
<tr>
<td>4.8.3 Deadlock Avoidance</td>
<td>82</td>
</tr>
<tr>
<td>5 Perspectives</td>
<td>83</td>
</tr>
<tr>
<td>6 Appendix</td>
<td>84</td>
</tr>
<tr>
<td>6.1 Building Environment</td>
<td>84</td>
</tr>
<tr>
<td>GHC Package Listing</td>
<td>84</td>
</tr>
<tr>
<td>7 References</td>
<td>88</td>
</tr>
</tbody>
</table>
List of Figures

1. Sequence diagram showing data flow from client to server and back. The place of the transaction framework in the server is between web framework and database. .......................................................... 33
2. On the right: Sequence and Interaction Diagram .................................... 42
3. Flowchart of transaction instructions .................................................. 48
4. Sketch of interaction with MVars ...................................................... 79
5. Data Structure .................................................................................. 87

List of Tables

1. Listing of \texttt{readVal} and \texttt{writeVal} behavior in relation to database and read- and write-set. .......................................................... 39
2. Parameters of the web-app to instruct status change. .............................. 47
3. How to handle Touched Values removal after commit for each key ............ 67
4. Return Values of \texttt{fst <$> (readVal \ k)} in relation to values in database and read- and write-set. Compare to table 1 on page 39 ........................................... 71

List of Screenshots

1. Step 1, “StatusEdit” ........................................................................... 44
2. Step 2, “StatusCheck”. Values that differ in database are printed red. ........ 44
3. Step 3, StatusSaved ........................................................................... 45
4. Another person has committed a dataset of the same user ID in the mean time. Option to reload from database. .................................................. 45
5. Warning about another person editing the same data set simultaneously. ... 46
Abstract

The concept of this web transaction framework is influenced by sessions\(^1\), transactions\(^2\) and software transactional memory\(^3\). To web-developer it offers sessions with integrated transaction management that has an included server-side storage of key-value pairs. The transactions can be spanned over multiple web requests and only affect the database when a commit is instructed. Additionally, the sessions have access to the (not yet committed) key-value pairs and the status (e.g., the last requested page) of all other sessions that are active. As sessions have to be canceled after a given timeout, the framework has a garbage collector that safely cleans up the timed out sessions. The framework introduces a monad which encapsulates all the logic of the transactions and frees them of side effects. This is a concept known from the STM–monad and like in STM, the transactions cannot dead-lock each other.

\(^1\)see page 13
\(^2\)see page 8
\(^3\)see page 9
1 Fundamentals

1.1 Transactions

A transaction is a set of instructions that are executed as a unit. A transaction must successfully complete or completely fail. The goal is to maintain a system’s integrity. This means that only completed transactions should be allowed to modify the state of a system.

It can be clarified with the aid of the following bank-transfer example: A money transfer from one account to another consists of withdrawing the given amount from the source account and depositing the same amount to the target account. In this example, a partially complete transaction could dissolve the money that is being transferred if the transaction was able to influence the state after withdrawing the amount from the source account and for some reason the depositing failed.

This is why a transaction management system has to ensure that either all operations succeed or none of them change the system’s state at all to ensure the integrity. If a transaction fails in an operation after some operations have already been completed, the transaction has to revoke all operations that were completed to restore the consistent, known state from before the transaction started. This is accomplished by a “roll back”.

1.1.1 ACID Principle

Transactions management systems have to guarantee the ACID properties: [1]

- **Atomicity** An operation either completes or completely fails
- **Consistency** Every operation turns the database from one consistent state to another consistent state.
- **Isolation** While one process is doing an update-operation, read-actions are permitted and the resulting data stems either from before or after the completed update-operation. Hence, a read-operation does not return data of an inconsistent state that results of a partial update-operation.
- **Durability** As soon as any operation is finished, it is guaranteed that the resulting state is saved even if the program crashes or a power blackout occurs.

1.1.2 User Interaction within Transactions

Transactions as described do not allow user-interaction within running transactions because they are executed in a side-effect free environment. All user interaction must be complete before the transaction can start.

Web based applications differ from local running applications. Web based user interaction usually goes like this: A user gets an input form rendered by the browser. After filling in the
data which can be split in sections of several web pages, the user usually receives a final web page stating the data again to verify. After confirming the input data, the “transaction” spanned over these several steps is complete. Since web based applications usually interact with a very large user base, it is not possible to lock resources for a longer amount of time.

What happens if relevant data changes while an “interactive” transaction is in progress?
This thesis describes one possible solution to this problem.

1.2 Deadlocks

One problem in concurrent programming is mutual exclusion of resources and the resulting possibility of deadlocks. Deadlocks occur when two (or more) processes simultaneously block a resource and wait for another to release their resource. For example, a process P1 has to take locks on resource R1 and then resource R2. At the same time, a process P2 takes both locks in reverse order. If process P1 has successfully taken resource R1 and P2 has taken R2, both processes are now waiting for each other to release the opponent’s lock. Because this release is not possible, the program is caught in a deadlock and these two processes will sleep or rather wait forever.

According to Coffman et al. [2] deadlocks occur only when the following four general conditions are operative:

1. Tasks claim exclusive control of the resources they require (“mutual exclusion” condition).
2. Tasks hold resources already allocated to them while waiting for additional resources (“wait for” condition).
3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion (“no preemption” condition).
4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain (“circular wait” condition).

1.3 Software Transactional Memory (STM)

A naive way to ensure the atomicity of an operation is to lock all the resources needed before starting the operation. To avoid deadlocks, it is possible to lock the resources in a linear order. But these conventional techniques force developers to take every lock- and release-action into account which is difficult and error-prone, often resulting in the easiest approach that turns the concurrent program into a rather sequential one.

The goal is to increase concurrency by limiting number and size of critical sections. Nir Shvit and Dan Touitou [3] suggested a transactional approach on software basis (instead of hardware implementations) and introduced Software Transactional Memory (STM).
A transaction in that matter is a set of read and/or write instructions. Each transaction either fails or successfully completes. Every transaction writes a log of all read and write instructions where the write actions do not yet change the shared memory. They are only noted in the log until verification succeeds. This very optimistic approach has the advantage of increased concurrency because there is no need to wait for a lock.

A transaction fails, if any other transaction concurrently made changes to the memory it read from. In case of a failure it can just start over again (called rollback) until it finishes successfully. Any transaction may abort or roll back at any time because no permanent data has been written, so that the overall state stays consistent.

On success the transaction performs a commit, which means it writes the changes to the shared memory to make it permanent. Only in this phase it needs to lock its resources or may even take a global lock. (This is a brief overview of one locking scheme called “commit-time locking”. There also are a few other approaches.)

### 1.3.1 STM in Haskell

The STM module defines a monad in which all transactional instructions are to be executed. This ensures that no side effects can influence the transaction. The provided function `atomically` serves as a lift function from the STM–monad into the IO–monad. It receives the STM action as an argument and returns its result into the IO–monad:

```
• atomically :: STM a -> IO a
```

The shared data is to be encapsulated with the data type `TVar a`. The following functions create and access TVars:

```
• newTVar :: a -> STM (TVar a)
• readTVar :: TVar a -> STM a
• writeTVar :: TVar a -> a -> STM ()
```

For the last expression of a do block of the STM–monad one can use

```
• return :: a -> STM a
```

or a function called

```
• retry :: STM a
```
retry aborts the transaction immediately and turns the running thread into a waiting state. As soon as one of the transactional variables gets updated the transaction will be restarted [4].

The function orElse “tries” two transactions in sequence: If the first transaction calls retry then the second transaction will be run. If the latter also calls retry, the result of orElse will be retry, too.

* orElse :: STM a -> STM a -> STM a

Example

This is a simple STM transaction that reads an Int–value from a TVar \( v \), and, if the read value equals 0, the transaction will be aborted and the thread will sleep until the TVar has been changed. Otherwise the Int within the TVar will be decremented. Example taken from [4].

\[
\text{decT} :: \text{TVar Int} \rightarrow \text{IO ()} \\
\text{decT v} = \text{atomically} \quad (\quad \text{do} \\
\text{x} \leftarrow \text{readTVar} v \\
\text{if} \ x == 0 \\
\text{then} \ \text{retry} \\
\text{else} \ \text{return} () \\
\text{writeTVar v} (x-1)
\]

Basic Implementation Sketch

GHC has an integrated STM library which makes use of many low level C-libraries. For this thesis it is more interesting to have a look at a higher level approach implemented by Frank Huch and Frank Kupke [5] and very slightly modified by me. It is interesting because the Trans–monad for the transaction framework described in this thesis has a similar structure. I will leave out details on how retry and orElse are implemented because the transaction framework does not have such functionality. In the following I sketch only the STM–monad instance implementation.

The STM–monad is a wrapper of the state transformer monad:

\[
\text{newtype} \quad \text{STM} a = \text{STM} \ (\text{runSTM} :: \text{StateT StmState IO (STMResult a)})
\]

There is the StmState that collects all the information of the transaction that is being executed (by atomically):
1 Fundamentals

```haskell
data StmState = TST {
    touchedTVars :: [(ID,Any)],
    isValid     :: IO Bool,
    commits     :: [IO ()],
    notifys     :: [IO ()],
    wait        :: IO (),
    retryMVar   :: MVar ()
}
```

Functions `newTVar`, `writeTVar` and `readTVar` manipulate only the StmState when run by atomically. `atomically` starts the `stmAction` with an `initialState`.

```haskell
state <- initialState
runStateT (runSTM stmAction) state
```

where

```haskell
stmAction :: STM a
initialState :: IO StmState
runSTM :: STM a -> StateT StmState IO (STMResult a)
runSTM stmAction :: StateT StmState IO (STMResult a)
runStateT (runSTM stmAction) :: StmState -> IO (STMResult a, StmState)
```

The `STMResult` consists of four outcomes where placeholder `a` is the return type of a successful transaction.

```haskell
data STMResult a = Retry
    | Invalid
    | Success a
    | Exception STMException
```

You can can see that this is an optimistic approach because the STM action itself decides whether the result is invalid. The STM monad instance only continues the transaction as long as the `STMResult` is `Success`. In all other cases it immediately stops the transaction and returns.
instance Monad STM where
    return = STM . return . Success
    x >>= f = STM $ do
        stmRes <- runSTM x
        case stmRes of
            Success a ->
                runSTM $ f a
            Retry -> return Retry
            InValid -> return InValid
            Exception e -> return $ Exception e

1.4 Web Application Frameworks

Web application frameworks (or web frameworks for short) are software frameworks for developing dynamic web pages, web applications and also web services. Such frameworks simplify common tasks and reduce overhead of web-development and encourage reuse of code.

Usually web frameworks come with mechanisms that make it easy to generate HTML with templating and manage URL mappings. Such frameworks also make it easier to communicate with data bases and allow programmers to work with higher-level concepts.

This thesis makes use of a web application framework for Haskell called Happstack.

1.5 Sessions

Through sessions, a client (browser) becomes recognizable over multiple requests to a web server. This is necessary for the server in order to keep a separate state for each client. Such a state can be used to store information, e.g., of a shopping cart in an E-commerce system.

The protocol HTTP of the World Wide Web has a stateless design. The connection is usually not kept alive from one request to another, from client (browser) to server (web server). The protocol is also not designed for identifying and distinguishing different clients. To accomplish this, however, it is necessary to realize sessions in the application layer of the ISO OSI model.

A session can be initialized by an HTTP-request. The server generates a state and a corresponding unique session ID. A session ID is often a long, randomly generated string. This ID is send back to the client with the HTTP-response to be saved, e.g., as a cookie value. This way the session ID is send to the specific web server with every request from then on. The session state is saved only on the web server, identified by the session ID. Another method of storing the session ID at the client-side is to embed it in every link and form of all HTML-documents so it is sent as parameter in every GET or POST request.
Examples for sessions are complex transactions in E-commerce systems, log-in state for websites with user accounts, or, as a minimalistic example, an empty state to test if the browser has sent a request before.

Finalizing a session can be explicitly done by a specific request (e.g., logging out of a user account or placing an order in an E-commerce system) or by timeout. The latter happens if web-surfing users close their web browser or if they do not use the web site for the specified time. The timeout length is defined on the server and the user usually has no information about it. When a session times out the server deletes the associated state. So after the timeout the session ID is not known to the server anymore and if a request is sent to the server with a timed-out ID, the server can either reply with an error page, reuse the session ID with a new empty state, or give the client a new session ID.
2 Happstack

Happstack stands for Haskell Application Stack.

Happstack is an open-source community dedicated to building the next generation of Haskell web technology. Our work includes libraries for templating, form validation, persistence, HTTP, and more [6].

In the following chapters I will walk you through the usage of some Happstack modules that will later be included in examples for the transaction framework.

2.1 Haskell Source with XML (HSX)

HSX allows inserting XML or more specifically HTML code into Haskell source code files. The GHC parameter -F -pgmF trhsx tell the compiler that the external preprocessor trhsx should be used to transform the source code before GHC compiles it. Through this transformation the source code usually expands and the error line numbers displayed refer to the transformed source code which is then misleading.

A simple line (plus the compiler options) could be:

{-# OPTIONS_GHC -F -pgmF trhsx #-}

html = <p>Hello world!</p>

It is possible to embed Haskell code into the XML code. For that, these special tags <% %> have to surround it. The function between these tags have to instantiate the class

class XMLGen m => EmbedAsChild m c.

It is also possible to embed lists of child elements with <%> and </%>:

html :: (XMLGenerator m) => XMLGenT m [ChildType m]
html =
  <%
    <a href="/">sum:</a>
    <$ show $ sum [1 ++ (10 :: Int)] $>
  </%>

The XML tree structure is transformed to a nested Haskell data structure and every XML element is translated to a function genElement. It takes as arguments the name, the attribute list and the list of child elements.

The above transformed looks like this:
html :: (XMLGenerator m) => XMLGenT m [ChildType m]

html = (asChild
  ([asChild
    (genElement (Nothing, "a")
      [asAttr (("href" :: String) := ("/" :: String))]
      [asChild ("sum:" :: String))],
      asChild ((asChild (show $ sum [1 .. (10 :: Int)]))))]))

with

genElement :: XMLGen m => Name
  -> [XMLGenT m [AttributeType m]]
  -> [XMLGenT m [ChildType m]]
  -> XMLGenT m (XMLType m)
type Name = (Maybe String, String) -- / Pairs of strings can represent names,
  -- meaning a name qualified with a domain.

The type of the hole XML tree is XMLGenT m XML. By using the unlift function
unXMLGenT :: XMLGenT m a -> m a you can get the data out of the monad.

2.1.1 Template

This is a simple template for all pages of an example website.

The following function template takes title, headers and content of a web page and embeds all
of it into one HTML page. The title argument is a String that will be put into the <title>
HTML tag. headers and body are of type EmbedAsChild (ServerPartT IO) headers. This
basically means it is an XML type.
The language extension FlexibleContexts allows to constrain the type to (ServerPartT IO) instead of a type variable in the signature of appTemplate.

Here the type Server is defined that later will become important in chapter “Type-safe forms” on page 26.

Example

```haskell
homepage = appTemplate "Homepage" () $
        <p>Welcome!</p>
```

This usage of the appTemplate function results in a web page that has the title “Homepage”, no additional headers and the body content is the paragraph “Welcome!”.

Since the resulting type is

Server Response which is equivalent to XMLGenT (ServerPartT IO) Response

it has to be “unlifted” with unXMLGenT to

ServerPartT IO Response
which is a type that matches the signature of the server’s main function \texttt{simpleHTTP} from module \texttt{Happstack.Server}.

This is a possible main function that starts up the Happstack server which answers all request with the homepage from above:

```
main :: IO ()
main =
    simpleHTTP nullConf {port = 8080} $ unXMLGenT homepage
```

### 2.2 ACIDState

ACID – in computer science – is a generally known abbreviation and stands for \textit{Atomicity, Consistency, Isolation and Durability} \cite{7}. The package \texttt{acid-state} offers an ACID-implementation in Haskell. In contrast to relational databases, this implementation is designed to persistently save arbitrary Haskell data types. Queries are also written in Haskell, in contrast to SQL for relational databases. This way, no manual parsing and converting of data structures is required and type-safety is guaranteed.

One can imagine that the ACID-state behaves like an \texttt{MVar} which contains all persistent data and additionally guaranties all ACID properties. (See chapter Transactions on page 8.)

The following sections show an example implementation of a user database.

#### 2.2.1 Data-type for Persistent Storage

At first there has to be the basic (Haskell) data type that contains all the wanted properties of the user. In contrast to relational databases a record data type declaration is used instead of a table and as it is also common in rel. databases a unique key – in this case \texttt{userId} – is used to identify each dataset. The \texttt{newtype UserId} is a wrapper-type that provides type safety.
import Data.Acid
import Data.Time
import Data.IxSet

data User = User
  { userId :: UserId
  , lastName :: String
  , firstName :: String
  , username :: String
  , password :: String
  , email :: String
  , birthday :: Day
  } deriving (Eq, Ord, Read, Show, Typeable)

ewtype UserId = UserId { unUserId :: Integer }
  deriving (Eq, Ord, Read, Show, Enum, Typeable)

The instance of class Typeable is needed for operations that are later described in section IxSet. By deriving class Enum by UserId the function succ :: UserId -> UserId is made available with which the unique key will be incremented.

2.2.2 IxSet

The package ixset offers a data type for sets with multiple indexed keys. This set is later saved to the ACID state. The different keys are matched by type. So they have to be of a unique type for the respective IxSet. For the many String record members in the User data type it is managed by wrapper data types to differentiate the keys.

newtype LastName = LastName String deriving (Eq, Ord, Data, Typeable)
newtype FirstName = FirstName String deriving (Eq, Ord, Data, Typeable)
newtype EMail = EMail String deriving (Eq, Ord, Data, Typeable)
newtype UserName = UserName String deriving (Eq, Ord, Data, Typeable)
newtype Password = Password String deriving (Eq, Ord, Data, Typeable)

All several keys are combined by the instantiation of class Indexable. The correlation to User is described by defining empty.
instance Indexable User where

empty = ixSet [ ixFun $ \user -> [ userId user] 
  , ixFun $ \user -> [ LastName $ lastName user] 
  , ixFun $ \user -> [ FirstName $ firstName user] 
  , ixFun $ \user -> [ UserName $ firstName user] 
  , ixFun $ \user -> [ Password $ firstName user] 
  , ixFun $ \user -> [ EMail $ email user] 
  , ixFun $ \user -> [ birthday user] ]

empty :: IxSet a
isSet :: [Ix a] -> IxSet a
 ixFun :: (Ord b, Typeable b) => (a -> [b]) -> Ix a

With a being assigned to User and b assigned to the suitable unique key from above. Type b
has to instantiate class Typeable so that the required matching of the type to the index type is
possible.

2.2.3 Set Operations

IxSet allows the following operation:

• insert :: (Typeable a, Ord a, Indexable a) => a -> IxSet a -> IxSet a
  The given dataset will be added to the given IxSet.

• delete :: (Typeable a, Ord a, Indexable a) => a -> IxSet a -> IxSet a
  The given dataset will be removed from the given IxSet.

In the following the type k stands for the type to match on.

• deleteIx :: (Indexable a, Ord a, Typeable a, Typeable k)
  => k -> IxSet a -> IxSet a
  The dataset with the given key will be removed from
  the given IxSet. If more than one dataset is found, nothing will be deleted.

• (@=) :: (Indexable a, Typeable a, Ord a, Typeable k)
  => IxSet a -> k -> IxSet a
  This request returns all datasets with the given key in their indices.

• (@<), (@>), (@<=), (@>=) (same signature as (@=))
  These requests are similar to the above with the difference that (<), (>), (<=) or rather
  (>=) is used as compare operator instead of (==).
Requests with interval (k stands for the type, not value)

- \((@><)\) :: \((\text{Indexable} \ a, \text{Typeable} \ a, \text{Ord} \ a, \text{Typeable} \ k)\)
  \(\rightarrow \text{IxSet} \ a \rightarrow (k, k) \rightarrow \text{IxSet} \ a\)
  Request with open interval (k,k)
- \((@>=<)\) (same signature)
  Request with half open interval [k,k)
- \((@><=)\) (same signature)
  Request with half open interval (k,k]
- \((@><=)\) (same signature)
  Request with closed interval [k,k]

Requests with lists

- \((@+)\) :: \((\text{Indexable} \ a, \text{Typeable} \ a, \text{Ord} \ a, \text{Typeable} \ k)\)
  \(\rightarrow \text{IxSet} \ a \rightarrow [k] \rightarrow \text{IxSet} \ a\)
  This request returns the datasets that contain at least one of the keys in their index from the given list.
- \((@*)\) (same signature)
  This request returns the datasets that contain all of the keys in their index from the given list.

To get sorted list results, it is necessary to determine a type that indicates which field is to sort. For this the data type Proxy is used.

```
data Proxy a = Proxy
```

Proxy has a type a and always the same value. Its only purpose is to indicate its parametrized type. It is passes to one of the following functions with constructor and type signature like this: (Proxy :: Proxy LastName).

- `toAscList :: (Indexable a, Typeable a, Typeable k) \(\rightarrow \text{Proxy} \ k \rightarrow \text{IxSet} \ a \rightarrow [a]\)`
  Transforms the IxSet into an ascending sorted list.
- `toDescList` (same signature)
  Transforms the IxSet into a descending sorted list.

**Example**

The result of the following snippet is a list of type User sorted in descending order of last name.
To get direct access to the current UserId (the counter) the data type UserBase is introduced, which also contains the IxSet “users”. This is also necessary because IxSet does not offer an auto increment functionality. This data structure is also used to store the data persistently. To create the dataset an initial state is used. More on that in section [global lock].

The initial UserId should be 1, so that the 0 is free for User data types that are not yet persistently saved.

### 2.2.4 Queries

Queries are completely expressed in Haskell. They are programmed in two phases: First, a query is directed to the embedded IxSet. These are not thread-safe. To realize thread safety, boilerplate-Data types and -instances are generated with the help of Template-Haskell. Those are used by a second query which will be used as the actual query, or rather: Those will be exported for external use, and the internal IxSet-queries will be hidden to other modules.

#### Queries with IxSet

First we have to distinguish between queries that only read from the database and those that have at least one write action. First, I will have a look at a query that has a write action. The following function takes a User as argument that it will save to the database.
2.2 ACIDState

```haskell
insertUserUpdate :: User -> Update UserBase UserId
insertUserUpdate user = do
    u@UserBase<.<.<.<< <- get
    put $ u { users = IxSet.insert user { userId = nextUserId } users
            , nextUserId = succ nextUserId }
    return nextUserId
```

Queries are expressed with the help of the `Update` or `Query` monad from package `acid-state`. The return type of `Update` is just a simple wrapper around the type `State` from module `Control.Monad.State`. The first type argument of `Update` stands for the data type of the state (UserBase) and the second one is the return data type of the query (UserId). The data type `State` implements the class `MonadState`. Because of that, the monadic functions `get` and `put` are available, which in addition to getting and putting the state of the monad, take a lock, and release the lock again, respectively.

Due to the language extension `RecordWildCards`, the code line `u@UserBase<.<.` is valid Haskell syntax. `get` return a date of type `UserBase`. By using this notation variables are set up for every record member. The names of the set-up variables are identical with the record member’s names. In this case `users` and `nextUserId`. This is identical with `u@(UserBase nextUserId users) <- get`.

The function `IxSet.insert` inserts the `User` “user” with index “`nextUserId`” into the `IxSet` “users”.

Alternatively the function `Ixset.updateIx` modifies existing entries. The entry to overwrite is detected by the index of the (unique) type k:

```haskell
updateIx :: (Indexable a, Ord a, Typeable a, Typeable k) => k -> a -> IxSet a -> IxSet a
```

The value of the record member `nextUserId` is incremented by 1.

By using `put` the modified `UserBase` is written back to the `MonadState`.

Queries that only `read` are to be of type `Query` which is a wrapper of the `Reader`-monad from module `Control.Monad.Reader`. The following function return the `User` that corresponds to the given `userId` if it exists. (Maybe `User`). In the `Query`-monad you use `ask` instead of `get`. The resulting difference is that no lock will be taken and other threads can concurrently do other actions on the same database.

**Example**

The following example query expects a user id and returns data of type `User` if a user with the given ID is existent:
2 Happstack

userByIdQuery :: UserId -> Query UserBase (Maybe User)
userByIdQuery uid = do
  UserBase(...) <- ask
  return $ getOne $ users @= uid

The operator (@=) is known from section IxSet. Since UserId is globally unique the IxSet resulting of (users @= uid) always has only one entry. The function getOne takes the first entry and returns it as Maybe, (Just) if present, or Nothing if not.

Queries with ACIDState

The following Template-Haskell function makeAcidic generates some boilerplate data types and instances that ensure the ACID properties and write a log for data recovering after a crash. It gets two arguments. The first one is the name of the data type that contains the database. The seconds argument is a list of the names of all query functions:

$(makeAcidic ''UserBase [ 'insertUserUpdate, 'userByIdQuery ])

As a result, some data types are generated that have the same name as the functions but begin with a capital. Those instantiate the class UpdateEvent and QueryEvent, respectively, depending on the return type of the functions they are based on.

In the end, the actual query is done with the help of the functions query’ or update’, respectively.

update’ :: (UpdateEvent event, MonadIO m) => AcidState (EventState event) -> event -> m (EventResult event)
query’ :: (QueryEvent event, MonadIO m) => AcidState (EventState event) -> event -> m (EventResult event)

These get the ACID-state and the data type constructors as arguments that are generated by makeAcidic with optional parameters. That sums up to:

userById :: MonadIO m => AcidState (EventState UserByIdQuery)
  -> UserId -> m (EventResult UserByIdQuery)
userById acid uid = do
  query’ acid (UserByIdQuery uid)

insertUser :: MonadIO m => AcidState (EventState InsertUserUpdate)
  -> User -> m (EventResult InsertUserUpdate)
insertUser acid user = do
  update’ acid (InsertUserUpdate user)
Alternatively, the underlying functions `query` and `update` can be used which force the IO-monad.

```haskell
query :: QueryEvent event -> AcidState (EventState event) -> event -> IO (EventResult event)
```

The following program writes a `User` into the database and reads it back again. (This is not a useful idea, it is just implemented for demonstration purposes.)

```haskell
insertAndRenderUser :: AcidState UserBase -> User -> Server Response
insertAndRenderUser acid user = do
  asssinedUserId <- insertUser acid user
  -- if userById returns Nothing, error is executed!
  userFromDB <- fromMaybe (error "Database error") <$> userById acid assinedUserId
  appTemplate "Your Registration" () $
    <dl>
      <dt>lastname:</dt>     <dd><% lastName userFromDB %></dd>
      <dt>firstname:</dt>    <dd><% firstName userFromDB %></dd>
      <dt>username:</dt>     <dd><% username userFromDB %></dd>
      <dt>email:</dt>        <dd><% email userFromDB %></dd>
      <dt>birthday:</dt>     <dd><% show (birthday userFromDB) %></dd>
    </dl>
```

### 2.2.5 Global Lock

The IO–function `openLocalState` opens the local ACID-state or sets it up if no state is found on disk. All other programs do not get access to the state as long as it is locked. With termination of the program the lock is removed.

```haskell
openLocalState :: (Typeable st, IsAcidic st) -> IO (AcidState st)
```

The following function helps to initialize the ACID-state at starting time of the server. With the help of `bracket`, crashes and exceptions are caught and in such cases the state will be closed and saved with `createCheckpointAndClose`.

```haskell
withStateDo serve = do
  bracket (openLocalState initialUserBaseState)
    createCheckpointAndClose
    serve
```
In module `Main` the function `withStateDo` can be used to get the ACID-state variable:

```haskell
main :: IO ()
main =
    withStateDo $ \acid -> simpleHTTP nullConf {port = 8080} $ route acid
```

With `route` being the function that takes care of the route in the Happstack web server. `route` delegates the `acid`-argument to the sub functions that deal with the database.

The persistent data is stored on the file system in folder “state”.

### 2.3 Type-safe Forms

Due to the package reform-happstack it is possible to implement HTML-forms that guarantee data validation and type safety.

For that, the module `Text.Reform.Core` offers a frame that can be fitted with individual data types which supply information about validation and return type.

```haskell
newtype Form m input error view proof a = Form
    { unForm :: FormState m input
    (View error view, m (Result error (Proved proof a)))}
```

In that matter `m` is a monad, `input` is of type `Happstack.Server.Input` and represents a HTML input field, `error` is a type that implements `FormError`, `view` is for rendering, and `a` is the return type of the form.

The following introduces the type `UserForm` that assigns some of the type-parameters of `Form`:

```haskell
type UserForm =
    Form Server [Input] AppError [Server (XMLType (ServerPartT IO))] ()
```

With the type variables assigned as follows: `m = Server`, `input = [Input]`, `error = AppError`, `view = [Server (XMLType (ServerPartT IO))]`, `proof = ()` with

```haskell
type Server = XMLGenT (ServerPartT IO)
```

Notice that `a` stays unassigned and is to be considered as a (type) parameter of `UserForm`. `Input` is imported from `Happstack.Server` and `AppError` is to be defined here:
2.3 Type-safe Forms

data AppError
  = Required
  | NotANatural String
  | NotADate String
  | PassNotIdent
  | AppCFE (CommonFormError [Input])
  deriving Show

Here, every constructor stands for an error that can occur somewhere in the form. AppError has to instantiate FormError:

instance FormError AppError where
  type ErrorInputType AppError = [Input]
  commonFormError = AppCFE

Similar to a Show instance, here a EmbedAsChild instance is necessary:

instance (Monad m) => EmbedAsChild (ServerPartT m) AppError where
  asChild Required = asChild $ "required"
  asChild PassNotIdent = asChild $ "Passwords don’t match"
  asChild (NotANatural str) = asChild $ "Could not decode as a positive integer: " ++ str
  asChild (NotADate str) = asChild $ "Not a valid date: " ++ str
  asChild (AppCFE cfe) = asChild $ commonFormErrorStr show cfe

Validation functions

The validation functions get the user form input from an input field and return an Either data type. Left will be returned with one of the above defined errors. Right will be returned containing the validated value. The value does not have to be of the same data (type) as the input data.

required :: String -> Either AppError String
required [] = Left Required
required str = Right str

requireDate :: String -> Either AppError Day
requireDate str = case reads $ str of
  [(day,""),] -> Right day
  _ -> Left $ NotADate str
requireMatchingPass :: (String, String) -> Either AppError String
requireMatchingPass (pass1,pass2) = if pass1 == pass2
  then Right pass1
  else Left PassNotIdent

There is a set of functions that each represent a different type of input field:

- inputText
- inputPassword
- inputSubmit
- inputReset
- inputHidden
- textarea
- buttonSubmit

They all have the type signature in common:

inputText :: (Monad m, FormInput input, FormError error,
            ErrorInputType error ~ input) => String -> Form m input error Html () String

The String argument defines the default input value.

The function transformEither is used as a link between input fields and validation.

transformEither :: Monad m
  => Form m input error view anyProof a -> (a -> Either error b)
  -> Form m input error view () b

As easily to identify from the type signatures, this function transforms one form type into another – similar to map with the addition that an error “breaks through”. Combined with an input field it gives the following type:

inputText "" `transformEither` required
  :: (Monad m, Monoid view)
  => Form m input AppError view () String

or alternatively with transformation to Day:

inputText "" `transformEither` requireDate
  :: (Monad m, Monoid view)
  => Form m input AppError view () Day
The following operator allows joining form elements of the same type:

\[
(\leftrightarrow) :: (\text{Monad } m, \text{Monoid} \; \text{view}) \\
\Rightarrow \text{Form } m \text{ input error view } () () \\
\rightarrow \text{Form } m \text{ input error view proof } a \\
\rightarrow \text{Form } m \text{ input error view proof } a
\]

\[
(\leftarrow\rightarrow) :: (\text{Monad } m, \text{Monoid} \; \text{view}) \\
\Rightarrow \text{Form } m \text{ input error view proof } a \\
\rightarrow \text{Form } m \text{ input error view } () () \\
\rightarrow \text{Form } m \text{ input error view proof } a
\]

As one can see by looking at the type signature, this is an infix operator that returns or rather passes the results of the the form element in “direction of the arrow”. The other argument in each case only serves as a layout element.

Example

\[
\text{errorList} \leftrightarrow \text{label } "\text{Last name:}" \\
\leftrightarrow (\text{inputText } "" \text{'}transformEither\text{'} \text{ required}) \leftarrow\rightarrow \text{br}
\]

with

\[
\text{errorList} :: (\text{Monad } m, \text{XMLGenerator } x, \text{EmbedAsChild } x \; \text{error}) \\
\Rightarrow \text{Form } m \text{ input error } [\text{XMLGenT } x (\text{XMLType } x)] () ()
\]

\[
\text{label} :: (\text{Monad } m, \text{XMLGenerator } x, \text{StringType } x \sim \text{Text}, \\
\text{EmbedAsAttr } x (\text{Attr } \text{Text } \text{FormId}), \text{EmbedAsChild } x \; \text{c}) \\
\Rightarrow c \rightarrow \text{Form } m \text{ input error } [\text{XMLGenT } x (\text{XMLType } x)] () ()
\]

\[
\text{errorList} \text{ and label are imported from module Text.Reform.HSP.String and constrain the} \\
\text{type variable view to } [\text{XMLGenT } x (\text{XMLType } x)]^{4} \text{ The type of UserForm matches: It assigns} \\
\text{view to } [\text{Server } (\text{XMLType } (\text{ServerPartT } \text{IO})))]. \text{ So the } x \text{ from above will be assigned to} \\
\text{ServerPartT IO.}
\]

By analyzing the type signatures of errorList and label you can easily guess which functionality these functions have if you look at EmbedAsChild x error and EmbedAsChild x c. This first one embeds the errors into x. The second one embeds some type c into x. c can be anything that can be embedded into x. Through the instance EmbedAsChild (ServerPartT m) AppError from above it is defined how data of type AppError (error is assigned to AppError) is to be embedded.

Hence, the line from the example does the following:

\[\text{errorList} \leftrightarrow \text{label } "\text{Last name:}"\]

\[\leftrightarrow (\text{inputText } "" \text{'}transformEither\text{'} \text{ required}) \leftarrow\rightarrow \text{br}\]

\[\text{errorList} \text{ and label are imported from module Text.Reform.HSP.String and constrain the type variable view to } [\text{XMLGenT } x (\text{XMLType } x)]^{4}\text{ The type of UserForm matches: It assigns view to } [\text{Server } (\text{XMLType } (\text{ServerPartT } \text{IO})))]. \text{ So the } x \text{ from above will be assigned to ServerPartT IO.}\]

\[\text{By analyzing the type signatures of errorList and label you can easily guess which functionality these functions have if you look at EmbedAsChild x error and EmbedAsChild x c. This first one embeds the errors into x. The second one embeds some type c into x. c can be anything that can be embedded into x. Through the instance EmbedAsChild (ServerPartT m) AppError from above it is defined how data of type AppError (error is assigned to AppError) is to be embedded.}\]

\[\text{Hence, the line from the example does the following:}\]

\[\text{errorList} \leftrightarrow \text{label } "\text{Last name:}"\]

\[\leftrightarrow (\text{inputText } "" \text{'}transformEither\text{'} \text{ required}) \leftarrow\rightarrow \text{br}\]

\[\text{errorList} \text{ and label are imported from module Text.Reform.HSP.String and constrain the type variable view to } [\text{XMLGenT } x (\text{XMLType } x)]^{4}\text{ The type of UserForm matches: It assigns view to } [\text{Server } (\text{XMLType } (\text{ServerPartT } \text{IO})))]. \text{ So the } x \text{ from above will be assigned to ServerPartT IO.}\]

\[\text{By analyzing the type signatures of errorList and label you can easily guess which functionality these functions have if you look at EmbedAsChild x error and EmbedAsChild x c. This first one embeds the errors into x. The second one embeds some type c into x. c can be anything that can be embedded into x. Through the instance EmbedAsChild (ServerPartT m) AppError from above it is defined how data of type AppError (error is assigned to AppError) is to be embedded.}\]

\[\text{Hence, the line from the example does the following:}\]

\[\text{errorList} \leftrightarrow \text{label } "\text{Last name:}"\]

\[\leftrightarrow (\text{inputText } "" \text{'}transformEither\text{'} \text{ required}) \leftarrow\rightarrow \text{br}\]

\[\text{errorList} \text{ and label are imported from module Text.Reform.HSP.String and constrain the type variable view to } [\text{XMLGenT } x (\text{XMLType } x)]^{4}\text{ The type of UserForm matches: It assigns view to } [\text{Server } (\text{XMLType } (\text{ServerPartT } \text{IO})))]. \text{ So the } x \text{ from above will be assigned to ServerPartT IO.}\]

\[\text{By analyzing the type signatures of errorList and label you can easily guess which functionality these functions have if you look at EmbedAsChild x error and EmbedAsChild x c. This first one embeds the errors into x. The second one embeds some type c into x. c can be anything that can be embedded into x. Through the instance EmbedAsChild (ServerPartT m) AppError from above it is defined how data of type AppError (error is assigned to AppError) is to be embedded.}\]

\[\text{Hence, the line from the example does the following:}\]
• List all errors if existing. (In this case only submission of an empty input field)
• Write label "Last name:"
• Write input field (empty if no data submitted, otherwise the value submitted or rather the value returned by required)
• Write line break

The type of the entire line is `UserForm String`. Now the goal is to get a value of type `User` by combining several input fields. For that, it is necessary to lift the constructor `User` into the `Server-monad` and pass to it all the input values as arguments.

Through `instance Funktor m => Funktor (Form m input error view ())` (from `Text.Reform.Core`) it is possible to use `(<$>)` and `(<>*)` as operator for `UserForm`.

\[
(<$>) :: \text{Functor } f \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

\[
(<>*) :: \text{Applicative } f \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

The constructor’s first argument is an `UserId` that will be reassigned later when storing to database. That is why the value at this point is only a place holder and not relevant. When partly applying `User` with an `UserId` and that again partly applying to `(<>*)` we get the following type. (\(f\) from above is at first forced to be `UserForm`, from the second code segment on it is type inferred)

\[
(User \ (UserId \ 0) \ <$>)
\quad :: \text{UserForm String}
\quad \rightarrow \text{UserForm } (\text{String } \rightarrow \text{String } \rightarrow \text{String } \rightarrow \text{String } \rightarrow \text{Day } \rightarrow \text{User})
\]

\[
(User \ (UserId \ 0) \ <$> \ (inputText "")
\quad :: \text{UserForm } (\text{String } \rightarrow \text{String } \rightarrow \text{String } \rightarrow \text{Day } \rightarrow \text{User})
\]

\[
(User \ (UserId \ 0) \ <$> \ (inputText "") \ <> \ (inputText "")
\quad :: \text{UserForm } (\text{String } \rightarrow \text{String } \rightarrow \text{Day } \rightarrow \text{User})
\]

\[
:
\]

\[
(User \ (UserId \ 0) \ <$> \ (inputText "") \ <> \ . . . <> \ (inputText "")
\quad <> \ (inputText "" \ ‘transformEither’ \ requireDate)
\quad :: \text{UserForm User}
\]

To be complete, the form needs a submit button which is added using `(<>*)`. That way, the input data of the button will be discarded and not passed to the `User` constructor.
2.3 Type-safe Forms

Now this form has to be integrated into Happstack and it has to be defined what should be done with the User data type when the form is successfully transmitted. For that the helper function `reform` is provided which expects several arguments. The first one is a function that generates the `<form>` elements. For that another helper function named `form` is provided. `form` partly applied with its first argument (the action attribute of the HTML form element) fits the type of the first argument of `reform`. Second argument of `reform` is a string that is used as a prefix of all form fields. So that several forms on one page do not conflict with field names. The next argument is a handler that will be called when the form successfully validates. After that, `reform` can be passed an optional handler that will be called on failure. If you omit it (with `Nothing`), the same form will be displayed again on failure with the user input and the errors will be included by `errorlist`.

```haskell
reform :: (ToMessage b, Happstack m, Alternative m, Monoid view)
    => ([(String, String)] -> view -> view) -- ^ wrap raw form html inside a <form> tag
    -> String -- ^ prefix
    -> (a -> m b) -- ^ success handler used when form validates
    -> Maybe ([(FormRange, error)] -> view -> m b) -- ^ failure handler used when form does not validate
    -> Form m [Input] error view proof a -- ^ the formlet
    -> m view
```

```haskell
form :: (XMLGenerator x, EmbedAsAttr x (Attr String action)) =>
    action -- ^ action url
    -> [(String, String)] -- ^ extra hidden fields to add to form
    -> [XMLGenT x (XMLType x)] -- ^ children
    -> [XMLGenT x (XMLType x)]
```
The combination of the function from above results into the following:

```haskell
registerForm :: UserForm User
registerForm =
  User (UserId 0)
    <$> errorList ++> label "Last name:"
    ++> (inputText "" (transformEither' required) <$> br
    <$> errorList ++> label "First name:"
    ++> (inputText "" (transformEither' required) <$> br
    <$> errorList ++> label "User name:"
    ++> (inputText "" (transformEither' required) <$> br
    <$> errorList ++>
      (,,) <$> errorList ++> label "Password:"
      ++> (inputPassword (transformEither' required) <$> br
    <$> errorList ++> label "Retype password:"
      ++> (inputPassword (transformEither' required) <$> br)
      (transformEither' requireMatchingPass
    <$> errorList ++> label "E-mail:"
    ++> (inputText "" (transformEither' required) <$> br
    <$> errorList ++> label "Birthday:"
    ++> (inputText "" (transformEither' requireDate) <$> br
    <*> inputSubmit "Register"
```

The following code integrates the form into Happstack. `insertAndRenderUser` has been introduced in section ACIDState on page 25. In contrast to the function `insertAndRenderUser` from above, here it should not store to the database and load the same data again.

```haskell
register :: AcidState UserBase -> ServerPart Response
register acid = unXMLGenT $
  appTemplate "register" () $
  <$> reform
    (form "/register")
    "register-"
    (insertAndRenderUser acid)
    Nothing
    registerForm
  %>
```
3 Integration of the Transaction Framework

The transaction framework takes its place between web framework and database on the server.

To work properly, it is important that only the transaction framework changes the relevant key-value pairs within the database. Since the framework keeps copies of key-value pairs as a cache, changes by other processes might be ignored.

Terms The concept of this transaction framework makes the terms Session, Transaction and TSession very similar or even identical (in the context of the framework).

The framework’s transactions are spanned over several requests which is the main aspect of a session. The data type for this “transaction session fusion” is called TSession. In this thesis the terms are sometimes used interchangeably because most of the times it does not matter if you write transaction or session.

A Transaction Part is a single request to the transaction framework which includes one call of the function atomicTransactionPart. A Transaction/Session usually consists of several transaction parts.

This concept forces strict division of application logic and view generation as known from the model view controller (MVC) concept. The logic resides in the side-effect free monad within the transaction framework and returns data to the view generation. The latter is located within the web application framework.
3 Integration of the Transaction Framework

3.1 Transaction Base

This framework supports transactions within sessions spanned over several requests. Every session or rather transaction needs a TransactionsBase as foundation. All sessions of one TransactionBase can communicate with each other. The TransactionBase data type is to be constructed as the web server starts. There can be several of its kind within one application. Different TransactionsBases define different transaction types or classes (not to be understood a Haskell-sense). E.g., one TransactionBase could be defined for a user administration application and another for an order placement application. These bases define several properties. This is the underlying data structure:

```haskell
data TransactionBase k v status = TransactionBase
    { _trBaseName :: String
    , _trBaseLoadFun :: k -> IO (Maybe v)
    , _trBaseSaveFun :: k -> v -> IO ()
    , _trBaseTouchedValues :: MVar (Map.Map k [TouchedValue v status])
    , _tsessions :: TSessions k v status
    , _tsessionIdGenerator :: TSessionIdGenerator
    , _tsessionInitStatus :: status
    , _tsessionTimeOut :: NominalDiffTime
    , _tsessionTimeoutIndex :: IORef (Map.Map TSessionId Timeout)
    }
```

**Code design** All record field names start with an underscore.

One can notice the three type variables \( k, v \) and \( status \). \( k \) and \( v \) will be assigned with the types of the key-value pairs. \( k \) stands for the type of the key and \( v \) stands for the type of the value. These types have to match the corresponding types of the database save and load functions. More on that later. The type variable \( status \) will be assigned with a custom data type that indicates the status of each session. This could be e.g.,

```haskell
data UserEditStatus = StatusEdit | StatusCheck | StatusSaved
```

That saves the status of the last request and indicates the status of this session to all other sessions. In other words, this \( status \) can be used to store information about each session that is not supposed to be saved to the database and also does not have to be in key-value structure. \( status \) is always defined for a session. A new session starts with the designated initial status. If a status is not needed, type and value () can be used for it.

\(^5\)Sessions that have different TransactionsBases are not able to communicate with each other.
The TransactionBase is to be constructed with a helper function that has the following type.

\[
\text{createTransaction} :: \text{Ord} \ k \\
\quad \Rightarrow \text{String} \quad \text{-- TransactionBase name} \\
\quad \Rightarrow (g \rightarrow \text{TSessionId}, g, g) \quad \text{-- Tuple of ID generation function} \\
\quad \Rightarrow \text{status} \quad \text{-- Initial status of new tsession} \\
\quad \Rightarrow (k \rightarrow \text{IO} \ (\text{Maybe} \ v)) \quad \text{-- Function for loading from database} \\
\quad \Rightarrow (k \rightarrow v \rightarrow \text{IO} \ ()) \quad \text{-- Function for storing to database} \\
\quad \Rightarrow \text{NominalDiffTime} \quad \text{-- Timeout in seconds} \\
\quad \Rightarrow \text{IO} \ (\text{TransactionBase} \ k \ v \ \text{status})
\]

the first string argument corresponds to the \(_\text{trBaseName}\) attribute from above. This string is used to separately store session ID of different transaction bases in one cookie. The developer has to take care that this string is unique and not used by another transaction base.

The next argument is a tuple of the session ID generation function and its state. The generation function’s purpose is the generation of a session ID. Every time the functions is called, it is passed its state. In addition to the session ID, it returns a new state which will be its argument at the next call. This is similar to Haskell’s random number generator which will be used in the example later. Notice that \text{type TSessionId = Integer}.

The argument of type \text{status} defines the status that a newly created session gets assigned.

The next two arguments are for loading and storing data from and to the database.

The timeout defines what period of time (in seconds) has to pass without request to a specific session until that session will be garbage collected. An \text{Int} argument will automatically be converted to \text{NominalDiffTime}.
Example

This is an example which uses Haskell’s random number generator and ACIDState⁶ and the data type UserEditStatus mentioned above.

```haskell
createUserEditTransaction :: AcidState UserState -> IO (TransactionBaseUserId User UserEditTransStatus)
createUserEditTransaction acid = do
  gen <- getStdGen
  createTransaction "editUser"
    (randomR (1000000000000000000,9999999999999999999),gen)
  StatusEdit
    (\uid -> userId acid uid)
  (\uid user -> insertUser acid user { userId = uid } >>= return ())
```

This returns a TransactionBase with the type variables assigned as follows: k ~ UserId, v ~ User and status ~ UserEditTransStatus. It gets the example functions of section ACIDState for loading and storing datasets of type User indexed by UserId. Through type inference the types of key and value are defined.

This newly created TransactionBase is now to be passed to the function atomicTransactionPart which is to be used in every request which uses the transaction. atomicTransactionPart is being passed code that expresses the logic of the transaction which is encapsulated in the Trans–monad.

⁶See page 18
3.2 The Trans Monad

Functions that interact with the transaction are to be used within the Trans-Monad.

```haskell
newtype Trans k v status a
instance Monad (Trans k v status)
```

This Monad has three type parameters k, v and status as described above, and a return type a. The following functions are provided for use within Trans:

- **writeVal ::** Ord k => k -> v -> Trans k v status ()
  writeVal is used to save a key-value pair in the session’s key-value store. It returns (). More information on page in section Read Set and Write Set on page 39.

- **readVal ::** (Ord k, Eq v) => k -> Trans k v status (Maybe v, TouchedValues v status)
  readVal is used to read a value by a given key from the database or if it has been read (written) before by the session then the value from the read set (write set) will be returned even though it could have changed in the database. Additionally, it returns all values that have been touched by all other active sessions for that key. Also see section Touched Values on page 40.

- **safeLoadVal ::** Ord k => k -> Trans k v status (Maybe v)
  This function loads the value respective to the given key from the database and not from the read or write set.

- **setStatus ::** Eq status => status -> Trans k v status ()
  This function sets the status of the calling session. More information on page 40.

- **getStatus ::** Trans k v status status
  This function gets the status of the calling session.

- **commit ::** Trans k v status ()
  commit tells the transaction controller to store the values, that have been written to the session’s key-value store, to the database as soon as the request is finished.

- **finishSession ::** Trans k v status ()
  This function tells the transaction controller to remove the calling session after the request is complete.

- **return ::** a -> Trans k v status a
  The monadic return function should be used to return everything needed for the resulting web page.
3 Integration of the Transaction Framework

Example

changePassword
:: UserId -> String -> Trans UserId User UserEditTransStatus User
changePassword uId password =
do
  (userM, _touchedVals) <- readVal uId
case userM of
    Nothing -> error "User undefined"
    (Just user) -> do
      writeVal uId $ user { password = password }
      setStatus StatusSaved
      commit
      return user

This snippet of code reads a user from the session’s read- or write-set, or rather from the database if not in either of the sets. In case of successful loading, the user is stored to the write set with the given password. The code also sets the status and requests a commit. This code has to be passed to the function atomicTransactionPart in order to be executed:

3.3 atomicTransactionPart

The function atomicTransactionPart acts as a lift function for the Trans–monad. It takes the TransactionBase and a function of type Trans k v status a as arguments and returns the result of the function into the calling monad:

atomicTransactionPart
:: (Ord k, Eq v, MonadIO m,
   Functor m, HasRqData m, FilterMonad Response m, MonadPlus m)
  => TransactionBase k v status
  -> Trans k v status a -> m a

Happstack Variant This function atomicTransactionPart is a variant and imported from Control.TSession.Happstack. Currently, it is the only one available. Since it is only a simple wrapper around the atomicTransactionPart function from module Control.TSession with the addition of the Happstack specific cookie getter and setter functions, it is easy to adapt it for other web frameworks. The second line of the type variable constraints are Happstack specific and added by the wrapper function. This is the only difference in the signature compared to the wrapped function.
Example

transaction ← createUserEditTransaction acid
atomicTransactionPart transaction $ changePassword uid password

For acid see section ACIDState on page 18 and for createUserEditTransaction see example on page 36.

3.4 Read Set and Write Set

Each session has two own sets of key-value pairs. The read set stores every key-value pair that the transaction reads from the database. After once read, the read action will return the value from the read set (which may be different from the database if the value there has been changed).

The write set stores every key-value pair that the transaction writes. After a write action, the read action will return the respective value from the write set.

<table>
<thead>
<tr>
<th>Action</th>
<th>Return Value</th>
<th>Read Set</th>
<th>Write Set</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
<td></td>
<td>(name,Flo)</td>
<td></td>
<td>(name,Flo)</td>
</tr>
<tr>
<td>readVal name</td>
<td>Flo</td>
<td>(name,Flo)</td>
<td></td>
<td>(name,Flo)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(name,Flo)</td>
<td></td>
<td>(name,Einstein)</td>
</tr>
<tr>
<td>readVal name</td>
<td>Flo</td>
<td>(name,Flo)</td>
<td></td>
<td>(name,Einstein)</td>
</tr>
<tr>
<td>writeVal name</td>
<td>Nadja ()</td>
<td>(name,Flo)</td>
<td>(name,Nadja)</td>
<td>(name,Einstein)</td>
</tr>
<tr>
<td>readVal name</td>
<td>Nadja</td>
<td>(name,Flo)</td>
<td>(name,Nadja)</td>
<td>(name,Einstein)</td>
</tr>
<tr>
<td>commit⁹</td>
<td>()</td>
<td>(name,Flo)</td>
<td>(name,Nadja)</td>
<td>(name,Nadja)</td>
</tr>
</tbody>
</table>

Table 1: Listing of readVal and writeVal behavior in relation to database and read- and write-set.

⁷Touched Values are left out for a simpler overview.

⁸Database may differ from read set if other transactions change data. safeLoadVal always gives the actual values from the database.

⁹commit does not immediately store values to the database. They will be stored after the function atomicTransactionPart successfully validates the transaction part.
3.5 Status

For each TransactionBase it is necessary to define an arbitrary type status. There are no constraints to this type whatsoever. The status can be used to store data for each session that is not supposed to be stored persistently to the database and does not have to be in key-value structure. If this is not needed, the type and value () can be used for this. The status can also be used for passive inter-transaction/session communication. As you will learn from the next section Touched Values, each transaction can access data of all other active transactions including the status. This way a web-developer can easily define how sessions interact with each other.

It can be used for different things, e.g., for finding out if two users are editing the same form and in which step of editing the other user was last. If both users submit the form data, the data of the first user to submit would be overwritten by the second submit. By checking the status of all other transactions the web-developer could either disallow editing for more than one session or the second user could be warned that someone else is editing the same form at the same time.

Examples

- ()
- data CountStatus = CountStatus Int
- data UserEditStatus = StatusEdit | StatusCheck | StatusSaved
- (CountStatus, UserEditStatus)

3.6 Touched Values

Every transaction can inspect the key-value pairs not only of the database but also the key-value pairs of all other active transactions. The latter are summarized as “Touched Values”. Touched Values are categorized in three types: It can be a value that is read by the transaction, it can be a value that has been written into the transaction’s own write set or it can be a value that has been committed to the database. readVal expects a key k as argument and returns a tuple of the corresponding value (Maybe) and all Touched Values corresponding to the given key k: readVal :: (...) => k -> Trans k v status (Maybe v, TouchedValues v status)

The data type is defined as follows:

data TouchedValues v status = TouchedValues
{ _touchedValuesAll :: [TouchedValue v status]
, _touchedValuesAfterLastLoad :: [TouchedValue v status]
, _touchedValuesWithoutMe :: [TouchedValue v status]
}
deriving Show
It consists of three lists of type `TouchedValue v status` which is defined as shown below. The first list `_touchedValuesAll` contains all available Touched Values in order of occurrence to the given key that are available in all active transactions regarding the given key \( k \). The two other lists are filtered variants (or rather sublists): `_touchedValuesAfterLastLoad` contains all Touched Values that were touched since the last page request of the calling session and `_touchedValuesWithoutMe` is a list of all Touched Values except the value that is touched by the calling transaction itself.

As mentioned above a Touched Value is one of the categories `ValueCommitted`, `ValueRead` or `ValueWritten`.

```haskell
data TouchedValue v status =
    ValueCommitted TSessionId v |
    ValueRead TSessionId v status |
    ValueWritten TSessionId v status
  deriving (Show, Ord, Eq)
```

All Touched Values store the TSessionID and the actual value of the key-value pair within the respective transaction. The value with `ValueCommitted` always equals the value from the database. This is convenient for checking for a database change with `readVal` (in combination with `_touchedValuesAfterLastLoad`). Since the attribute \( v \) of `ValueCommitted` is always identical to the database, it will be removed as soon as no active transaction is left that has touched the corresponding key \( k \).

Additionally, the `ValueRead` and the `ValueWritten` also store the status of the respective transaction/session.

**Constraint**

In a list of Touched Values corresponding to an arbitrary key \( k \), there is at most one entry for each transaction ID. Given a transaction \( t_1 \) that has a value \( v_1 \) in read set and a value \( v_2 \) in write set, both corresponding to the same key \( k \). The Touched Value \( tv_2 \) that corresponds to value \( v_2 \) hides the Touched Value \( tv_1 \) corresponding to value \( v_1 \). Other transactions only see `ValueWritten t1 v2 status` (besides Touched Values of transactions other than \( t_1 \)) when retrieving Touched Values by `readVal k`.

If transaction \( t_1 \) calls a commit, then the Touched Value will change to `ValueCommitted t1 v2` in the global Touched Values.
3.7 Sequence- and Interaction-Diagram

The purpose of the following diagram is to show the interaction of the transactions with the database, with the Touched Values and with each other. It shows two transactions sequentially where time flows from top to bottom. The two TSessions T1 and T2 execute code blocks alternately. Each block of code is to be understood as written in do-notation of the Trans–monad and each executed with a call of atomicTransactionPart. The code-instructions for T1 and T2 are listed under the heading “Actions” of the respective TSession. The return values of each instruction is printed next to it on the right. The read- and write-sets are printed out for every code block where they change. For the point of time between the code blocks, the current Touched Values are shown left of the TSessions. Finally, on the very left the database key-value pairs are displayed.

To keep things simple the diagram has the following name convention: The keys are A and B and statuses start with S. Values have type Int.

By following the lines printed on the transparent overlay one can see how the data is related between the columns.

What to observe

- **readVal** puts the value to read set and a corresponding Touched Value is created.
- **writeVal** puts the value to write set and not to database and a corresponding Touched Value is created. If a ValueRead is in the Touched Values with the same key, it will be replaced.
- A second readVal for the same key, returns the value of the read set, not of the database.
- **commit** stores the write set to database and changes all respective ValueWritten to ValueCommitted and puts them to the head of list.
- If a Touched Value exits, with every readVal or writeVal, it jumps to the head of the list.
- **setStatus** overrides the status in all Touched Value of the respective transaction.
- **finishTransaction** all Touched Values of the calling transaction (except ValueCommitted).
- **safeLoadVal** returns the actual value that is in database. However, if a matching ValueCommitted exists, then this is returned because it is always identical.
- A ValueCommitted–entry in the Touched Values is deleted as soon as no active transaction is available that has a value in read- or write-set with the same key as the ValueCommitted. (Here after timeout of T2)
- In the last code block of TSsession T2, the key A is loaded safely from database. This is a pessimistic strategy. In that way both TSessions add 2 to the value corresponding to key A. If T2 did not safely load the value, the committed value of TSsession T1 would be lost/overridden.

Figure 2: On the right: Sequence and Interaction Diagram
<table>
<thead>
<tr>
<th>Database</th>
<th>Touched Values</th>
<th>TSession T1</th>
<th>TSession T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A,1), (B,2)</td>
<td></td>
<td>setStatus S1 ()</td>
<td>setStatus S2 ()</td>
</tr>
<tr>
<td>(A,1), (B,2)</td>
<td></td>
<td>readVal A (Just 1,[])</td>
<td>readVal A (Just 1, [ValueRead T1 1 S1])</td>
</tr>
<tr>
<td>(A,1), (B,2)</td>
<td>A-&gt;[ValueRead T1 1 S1]</td>
<td>A-&gt;1</td>
<td>A-&gt;1</td>
</tr>
<tr>
<td></td>
<td>(A,1), (B,2)</td>
<td>A-&gt;[ValueRead T2 1 S2, ValueRead T1 1 S1]</td>
<td>writeVal A (A+2) ()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A-&gt;1</td>
<td>readVal B (Just 2,[])</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>readVal A (Just 3, [ValueRead T2 1 S2, ValueRead T1 1 S1])</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A-&gt;3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A,5), (B,2)</td>
<td>A-&gt;[ValueCommitted T2 5]</td>
<td>finishTransaction ()</td>
<td>commit ()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A,5), (B,2)</td>
<td>A-&gt;[ValueCommitted T2 5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table represents a series of actions and return values for two sessions, TSession T1 and TSession T2, over a database with values (A,1), (B,2). The actions include setting status, reading, writing, and committing values. Return values are also shown.
3.8 Example: Edit User Transaction

In the following I will demonstrate a use case of the transaction framework. As a working example I will present an editing web page wizard with which you can edit a user data set in three steps. Browser-screenshots of these steps are displayed in screenshots 1–3 on the following pages.

**Screenshot 1: Step 1, "StatusEdit"**

![Screenshot 1: Step 1, "StatusEdit"](image1)

**Screenshot 2: Step 2, "StatusCheck". Values that differ in database are printed red.**

![Screenshot 2: Step 2, "StatusCheck"](image2)

The three steps/statuses are:

- Edit the user in a HTML form: StatusEdit
- Check if the data you typed in at the previous page is correct: StatusCheck
- Result page showing the committed data: StatusSaved
Additionally to just editing a User dataset, the web page wizard informs you when someone else is simultaneously editing the same dataset or if the dataset has been committed by someone else in the time since your first page request. In the latter case you can safely load the new values from the database, which will override your uncommitted input. (Displayed in screenshots 5 and 4)

This results in the datatype that is used for the status type argument of the transaction base:

```haskell
data UserEditTransStatus = StatusEdit | StatusCheck | StatusSaved
  deriving (Show, Read, Eq, Enum, Bounded)
```

The function for creating the transaction base has already been introduced in section Transaction Base on page 34. Now I introduce the function `editUser`, which is to be called from the Happstack main function that implements the user editing wizard. It takes the transaction base as argument and returns a response into Happstack’s `ServerPartT I0`-monad.

```haskell
statusCheck

warning! This user has been changed!!

reload

in data base

- lastname: Florian
- firstname: Micheler
- username: fmi
- email: fmi@informatik.uni-kiel.de
- birthday: 1983-02-15

back

save
```

Screenshot 3: Step 3, StatusSaved

Screenshot 4: Another person has committed a dataset of the same user ID in the mean time. Option to reload from database.
3 Integration of the Transaction Framework

editUser :: TransactionBase UserId User UserEditTransStatus
           ServerPartT IO Response
editUser transaction = do
  uIdStr <- look "uid"                      -- (1)
  let uId = UserId (read uIdStr)           -- (2)
  statusSubmitted <- optional $ look "status" -- (3)
  reloadM <- optional $ look "reload"      -- (4)
  -- ↓ (5)
  result <- unXMLGenT $ eitherForm environment prefix (registerForm emptyuser)

Note that the key-value pairs are of type (UserId, User). There is not a separate pair for each field of the User record.

3.8.1 Parsing Parameters

The web page expects some parameters from the browser (GET or POST). One for the UserId that is desired to be edited, which is obligatory (1) – (2), and some optional ones for overriding the status (3) and for instructing the web app to safe-reload the User data set from the database (4). By using optional the returned variable is of type Maybe. The following is a possible URL for the web-app:

```
http://happstack-server/editUser?uid=1
```

The use of eitherForm in (5) is a work-around: Later, the form function happstackEitherForm will be used. The latter handles the input from the browser and builds the result-view in one
3.8 Example: Edit User Transaction

Unfortunately, in the transaction framework, the transmitted data has to be present before the view is generated because `atomicTransactionPart` has its place in between. That is why an additional `eitherForm` must be put here. It is actually a utility function used within `happstackEitherForm`. It returns an `Either` type of which only the Right side will be used.

```haskell
result :: Either view User
```

The following table offers an overview of some parameters (except uid) that can be sent from the browser to the web-app by either POST or GET method.

<table>
<thead>
<tr>
<th>Status to change to</th>
<th>HTTP-parameters</th>
<th>reload</th>
<th>status</th>
<th>result (form-data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no change, default=StatusEdit</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>StatusCheck</td>
<td></td>
<td>-</td>
<td>status=StatusSaved</td>
<td>User¹⁰</td>
</tr>
<tr>
<td>StatusSaved</td>
<td></td>
<td>-</td>
<td>status=StatusSaved</td>
<td>-</td>
</tr>
<tr>
<td>StatusEdit</td>
<td></td>
<td></td>
<td>status=StatusEdit</td>
<td>-</td>
</tr>
<tr>
<td>StatusEdit with reload</td>
<td>reload</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Parameters of the web-app to instruct status change.

If no parameter is sent (except the mandatory “uid”) the internal status stays with default `StatusEdit`. This is the step where the web-user starts the wizard. The other parameters are either submitted by the form by clicking “edit” on screenshot 1 or they are embedded into links where the web-user clicks on to navigate. These links are “reload” on screenshot 4 and “save” on screenshot 3.

Cross referencing to figure 1, the code at (1)–(5) is described by the step “Request-data Parsing” within the web framework. Here, “Routing” is left out since that should be part of the Happstack main function. The arrow “Transaction Instructions” symbolizes the do-block of the Trans–monad in the following code blocks. (7) et seqq.

The arrow “Return Data” symbolizes the returned quintuple which has information to be used for building the HTML-view. It has the type `(UserEditTransStatus, User, User, Bool, Bool)` (6)

The HTML-construction is described on page 51. It needs the status, the `User`-variable of the user that is being edited and the `User`-variable of the user that is present in the database and the two `Bool`-variables are for the two warning messages as seen on screenshots 5 and 4 respectively.

¹⁰The variable `result` is not actually one single HTTP-parameter. Instead the `User` data type is constructed by `eitherForm` which collects the attributes from several HTTP-parameters.
3.8.2 Transaction Instructions

The flowchart in figure 3 shows all possible decisions within the transaction instructions depending on the parameters displayed in table 2 on the previous page. It gives a good overview of the code that follows on the next few pages.

First, the do block instructs the transaction framework to read the user at the given UserId, which also returns the relevant Touched Values, and to safely reload the actual value of that user from the database. (8)–(9) If no user can be loaded under the given UserId, then the program exits with the error function.
In the following description follows the sequence of the code. This might not seem intuitive since it is not the order of the steps that the web-user goes through.

The following code block is executed if a web-user clicks on the reload link as seen in screenshot 4.

```haskell

Just user -> do
  case reloadM of
    -- reload:
    Just _ -> do
      setStatus StatusEdit
      let user' = fromMaybe user $ userFromDBM
      writeVal uId user'
      return (StatusEdit, user', userFromDBM, warningEdit touchedVals,
               warningChanged user' touchedVals)
```

The variable `reloadM` contains a `Just`–element if the server received a parameter named “reload” from the browser. The following URL would trigger this condition:

```
http://happstack-server/editUser?uid=1&reload
```

In such case, the web-app jumps back to step 1 (StatusEdit) and reloads the user from database. In case of `userFromDBM == Nothing`–which should not happen if the app is used properly because reloading is only offered if there is a user in database– the variable `user` is used as fallback. (13)

The reloaded user-value has to be put to write set, so that the next use of `readVal` returns it again. (14)

All necessary data for the HTML-construction is returned out of the `Trans`–monad into the `Happstack`–monad. For `warningEdit` and `warningChanged` see following excursus. (15)

**Warning Indicators**

The functions `warningEdit` and `warningChanged` each compute a boolean. (15)

The former indicates if another user is editing while the latter indicates that a new user-dataset has been committed, both as seen in screenshots 5 and 4 respectively. The functions are defined as follows.

```haskell

warningEdit touchedVals = (not.null) $ Prelude.filter f $ _touchedValuesWithoutMe touchedVals
  where
    f (ValueCommitted _ _) = False
    f _ = True
```

`warningEdit` filters the `ValueCommitted`–entries out of the list of Touched Values of other transactions and indicates if the list is empty. If there is an entry of type `ValueRead` or
ValueWritted, then there is another active transaction working with the same UserId. 
ValueCommitted has to be filtered out because it does not indicate that another transaction is active.

\[
\text{warningChanged user touchedVals = (not.null) $ Prelude.filter f $}
\]
\[
\text{_touchedValuesWithoutMe touchedVals}
\]
\[
\text{where}
\]
\[
\text{f (ValueCommitted _ user')} = user /= user'
\]
\[
\text{f _ = False}
\]

warningChanged looks for the ValueCommitted–entry that contains a user that is unequal to the one returned by readVal. In that case another transaction has committed a dataset to the database.

Because of the lazy evaluation of Haskell, these defined functions return the result without filtering the lists completely if not necessary.

In case of no “reload” parameter (15), the next step is to check for submitted form-data: (15)

```haskell
-- no reload: (case reloadM of)
Nothing -> do
  case result of
    -- no form data transmitted:
    Left _ -> do
```

If no form-data is submitted (17), parse the status parameter and make a decision depending on it. (18)

```haskell
  case read <$> statusSubmitted of
    Just StatusEdit -> do
      setStatus StatusEdit
          -- (19)
    Just StatusSaved -> do
      setStatus StatusSaved
          -- (20)
      commit
      finishSession
      Nothing -> return ()
          -- (21)
    status <- getStatus
    return (status, user, userFromDBM, warningEdit touchedVals,
            warningChanged user touchedVals)
          -- (22)
```

The following URL submits the status parameter:

```
http://happstack-server/editUser?uid=1&status=StatusSaved
```
3.8 Example: Edit User Transaction

If StatusEdit was submitted, just run setStatus to store it in the TSession’s server side status variable. (19)
Do the same with StatusSaved and additionally call commit and finishSession. (20)
If no status parameter available, do nothing. (21)
As in (15), return all relevant data. (22)

If a User–data set is submitted by the form (23), then set status to StatusCheck (24) and write the submitted User to write set: (25)

```
-- form data transmitted (case result of)
Right user -> do
  setStatus StatusCheck
  writeValue uId user
  return (StatusCheck, user, userFromDBM, warningEdit touchedVals,
              warningChanged user touchedVals)
```

As before, return the relevant data for the HTML construction. (26)

3.8.3 HTML Construction

The HTML will be constructed with the help of Haskell Source with XML (HSX) as introduced on page 15. The variables within the tuple returned by atomicTransactionPart will be used: (status, user, userFromDBM, warnEdit, warnChanged)

```
unXMLGenT $ do
  appTemplate "Edit User" () $ do
  result <- happstackEitherForm toForm prefix (registerForm user) -- (27)
```

The function happstackEitherForm is supposed to be used for the HTML form view–construction and data validation or rather data construction of the sent form values. Because of the work-around mentioned on page 46, here, only the HTML-code for the form returned by happstackEitherForm will be used, which is the Left–part of the returned Either data type. (27)

The variable html will be assigned with the HTML-code for the segment of the web-page that is dependent on the status. This variable will then later be embedded between the page-head and -bottom which are independent of the status. (28)
### 3 Integration of the Transaction Framework

```haskell
let html = case status of
    StatusEdit -> do
        case result of
            Left formHtml ->
                $ formHtml %>
            _ -> asChild ()
```

In case of StatusEdit the HTML-form is to be displayed. (29)
In this case the result variable should always be of the Left constructor. The Right constructor contains the submitted form data and as indicated in table 2, submitted data triggers StatusCheck.

```haskell
StatusCheck -> do
    let value_back = show $ pred status
    let value_save = show $ succ status
    let below = <%
        <form action=myDir method="GET">
            <input type="hidden" name="uid" value=uIdStr />
            <input type="hidden" name="status" value=value_back />
            <input type="submit" value="back" />
        </form>
        <form action=myDir method="GET">
            <input type="hidden" name="uid" value=uIdStr />
            <input type="hidden" name="status" value=value_save />
            <input type="submit" value="save" />
        </form>
    %>
    renderUser below user Nothing
```

In case of StatusCheck basically only the user will be rendered with the two buttons below, as seen on screenshot 2. (30)
For renderUser see page 54. Note that the variable html contains the code for the user displayed on the left. The code for the user displayed on the right hand side of the web-page starts at l. 101.
3.8 Example: Edit User Transaction

```haskell
StatusSaved -> do
  let value_start = show StatusEdit
  let below = <%
    <b>Saved!</b>
    <form action="myDir" method="GET">
      <input type="hidden" name="uid" value="uIdStr" />
      <input type="hidden" name="status" value="value_start" />
      <input type="submit" value="back to start" />
    </form>
  </%>
  renderUser below user Nothing
```

The case of StatusEdit is very similar with the difference of displaying Saved! and that only one button is displayed that guides to step 1.

The following code is independent of the status and describes amongst others the structure of the web-page.

```haskell
<%>
  <% show status %>
  <% if warnEdit %>
    then asChild $
      <p><font color="red">warning! This user is currently edited by another instance</font></p>
  else asChild () %>
  <% if warnChanged %>
    then asChild $
      <p><font color="red">warning! This user has been changed!!</font></p>
      <a href="myDir++?uid="+uIdStr++"&reload">reload</a></p>
  else asChild () %>
  <table><tr><td valign="top">
  <br />
  <% html %>
  </td>
```

As seen on all screenshots, the status is printed on top of the web-page. This is actually for debugging purposes. (31)

Next, the warnings are included into the HTML-code if the boolean variables warnEdit or warnChanged are set respectively. The warning for the database change includes a link for reloading. (32), (33)

To display the two User datasets next to each other, a table with two columns is used. (34)

In the first column the HTML-code of the variable html from above is inserted. (35)
In the right column of the table the user is displayed, which is currently stored in the database if it differs from the one in read- or rather write-set. (36)
To make the attributes show up in red, that differ in the users as seen in the screenshots 2 and 4, the function renderUser takes both users as arguments (37): userFromDB is the one to be displayed and user serves as reference for comparing. See renderUser:

renderUser below user userDiffM = do
<div>
  <ul>
    <li>lastname: <span class="red">% renderUserAttr user userDiffM lastName %</span></li>
    <li>firstname: <span class="red">% renderUserAttr user userDiffM firstName %</span></li>
    <li>username: <span class="red">% renderUserAttr user userDiffM username %</span></li>
    <li>email: <span class="red">% renderUserAttr user userDiffM email %</span></li>
    <li>birthday: <span class="red">% renderUserAttr user userDiffM (show.birthday) %</span></li>
  </ul>
</div>

renderUserAttr user userDiffM attr =
  if (isJust userDiffM) && (attr (fromJust userDiffM) /= attr user)
  then <span class="red">% attr user %</span>
  else <span class="red">% attr user %</span>
4 Implementation

4.1 GHC Language Extensions

4.1.1 Record Wild Cards

The GHC syntax extension “RecordWildCards” helps (among others) binding variables from record fields. Given the following snippet of a function definition

```haskell
timeouthandler tr@TransactionBase{..} = do
  threadDelay $ 1000 * 1000 * (round _tsessionTimeOut) 'div' 10
  ...
```

By writing `TransactionBase{..}` (1) a variable for every record field of the record constructor “TransactionBase” will be defined. The variable name correlates to the record field name and hides its lookup-function. Thus, `_tsessionTimeOut` (2) is a variable of type `NominalDiffTime` and hides the lookup-function `_tsessionTimeOut :: TransactionBase k v status -> NominalDiffTime`. That way the syntax for accessing record fields is shorter and more clearly represented.

4.1.2 Existential Quantification

The GHC language extension “ExistentialQuantification” allows hiding a type variable within a data constructor with use of the `forall` keyword. Given the following data type declaration for `TSessionIdGenerator`.

```haskell
data TSessionIdGenerator = forall g. TSessionIdGenerator
  { _tsessionIdGeneratorRef :: IORef g
  , _tsessionIdGeneratorGenFun :: g -> (TSessionId, g) }
```

The ID generator acts like a “black box” which generates a `TSessionId`. For usage it is not necessary to know what the internal type `g` is assigned to. Since `g` is not generally known it cannot be used outside the black box. Imagine this definition of a possible ID generator:

```haskell
let idgen = TSessionIdGenerator idgenRef random
```

with

```haskell
idgenRef :: IORef StdGen and
random and StdGen from module System.Random
```

The following monadic expressions generate an ID:
let TSessionIdGenerator{..} = idgen
let tsessionId <- atomicModifyIORef _tsessionIdGeneratorRef (swap _tsessionIdGeneratorGenFun)

Whatever type g stands for, the corresponding data type is read from the IORef, passed to _tsessionIdGeneratorGenFun and then written back to the IORef.

### 4.2 The Monad

The Trans–Monad is similarly designed as the STM–Monad introduced in chapter STM in Haskell on page 10. It also is a wrapper around the state transformer monad with an inner state of type TSessionState k v status, the inner monad IO and the result data type TransactionResult a.

newtype Trans k v status a = Trans
  { runTransactionPart ::
    StateT (TSessionState k v status) IO (TransactionResult a) }

with

data TransactionResult a = Success a | Exception TransactionException

The type of the record field function runTransactionPart is:

runTransactionPart :: Trans k v status a
                    -> StateT (TSessionState k v status) IO (TransactionResult a)

The monad instance extends the StateT–Monad instance with the addition of Exception handling:

instance Monad (Trans k v status) where
  return = Trans . return . Success
  x >>= f = Trans $ do
    res <- runTransactionPart x
    case res of
      Success a ->
        runTransactionPart $ f a
      Exception e -> return $ Exception e
return injects its argument into the Trans–monad with constructor Success of data type TransactionResult. The following type signatures should make clear why the function compositions are correct:

\[
\begin{align*}
\text{Success} & : : \ a \to \TransactionResult a \\
\text{return \ . Success} & : : \Monad m \Rightarrow a \to m (\TransactionResult a) \\
\text{Trans \ . return} & : : \TransactionResult a \to \Trans k v \text{status} a \\
\text{Trans \ . return \ . Success} & : : a \to \Trans k v \text{status} a
\end{align*}
\]

with

\[
m = \text{StateT (TSessionState} \ k \ v \ \text{status}) \ IO
\]

based on instance:

\[
\text{instance Monad m \Rightarrow Monad (StateT s m)} \\
\quad \text{-- Defined in ‘Control.Monad.Trans.State.Lazy’}
\]

The sequential composition operator \( x >>= f \) of monad Trans first calls \( \text{runTransactionPart} \ x \) which results in type \( \text{StateT (TSessionState} \ k \ v \ \text{status}) \ IO (\TransactionResult a) \). Thus, the expression \( \text{res <- runTransactionPart} \ x \) causes \( \text{res :: TransactionResult a} \) with \( x :: \Trans k v \text{status} a \) and \( f :: (a \to \Trans k v \text{status} b) \).

Depending on the outcome of \( \text{res} \) being Success or Exception the following two branches are possible. With Success \( a \) the composition operator results in \( \text{Trans} \ \$ \ \text{runTransactionPart} \ \$ \ f \ a \) which is of type:

\[
\begin{align*}
f & :: (a \to \Trans k v \text{status} b) \\
f \ a & :: \Trans k v \text{status} b \\
\text{runTransactionPart} \ \$ \ f \ a & :: \text{StateT (TSessionState} \ k \ v \ \text{status}) \ IO (\TransactionResult a) \\
\text{Trans} \ \$ \ \text{runTransactionPart} \ \$ \ f \ a & :: \Trans k v \text{status} b
\end{align*}
\]

With Exception \( e \) the composition operator results in \( \text{Trans} \ \$ \ \text{return} \ \$ \ \text{Exception} \ e \) which matches the type \( \Trans k v \text{status} b \). In this case the Exception "breaks through", meaning that \( \text{runTransactionPart} \ \$ \ f \ a \) is never being called and thus the transaction is discontinued or, if you will, canceled.
4 Implementation

4.2.1 The Monad’s State

Since the Trans–monad is an extension of the state monad it has a state within.

```haskell
data TSessionState k v status = TSessionState

{ _tsessionBase :: TransactionBase k v status, -- (1)
  _tsessionId :: TSessionId, -- (2)
  _tsessionReadSet :: Map.Map k (TouchedValue v status), -- (3)
  _tsessionWriteSet :: Map.Map k (TouchedValue v status), -- (4)
  _tsessionCommit :: Bool, -- (5)
  _tsessionFinish :: Bool, -- (6)
  _tsessionStatus :: status, -- (7)
  _trBaseTouchedValuesCopy :: Map.Map k [TouchedValue v status], -- (8)
}
```

1. _tsessionBase is a link to the transaction base needed for, e.g., access to the database.
2. _tsessionId is needed, e.g., for writing into the Touched Values.
3. _tsessionReadSet is the working read set for the Trans–monad. After each atomicTransactionPart this working read set will be put into the _tsessionVarReadSet of the TSession.
4. _tsessionWriteSet analogous to above.
5. _tsessionCommit is a flag with default value False and it is set to True when function commit is called within the Trans–monad. This boolean is checked later by atomicTransactionPart.
6. _tsessionFinish is also a flag and handled analogous to the commit flag but with the meaning that the TSession is to be deleted after the response is sent.
7. _tsessionStatus holds the current status of the TSession and can be accessed via setStatus and getStatus.
8. Since the Touched Values are inside an MVar the state gets a copy of them for : _trBaseTouchedValuesCopy

When merging the read- and write-set to the Touched Values, the copy will first be compared to the actual Touched Values, and, if a corresponding value from the TSession’s read set has been changed there, the transaction part is invalid and will be restarted.

For a full overview of all data type definitions of the framework, see figure 5 on page 87.

4.3 atomicTransactionPart

The function atomicTransactionPart is the heart of the framework. It runs the TSessions and returns the result into some monad m (constrained to MonadIO m). In the following I will walk
you through the whole function step by step. You can also follow the branches of execution and MVar interactions of the function in figure 4 on page 79. The cross references to that sketch are noted in footnotes in this chapter.

```haskell
atomicTransactionPart :: (Ord k, Eq v, MonadIO m)
  => (TransactionBase k v status -> m TSessionId)
  -> TransactionBase k v status
  -> Trans k v status a -> m a
atomicTransactionPart cookieFun tr @ TransactionBase {..} code = do
  tId <- cookieFun tr
      -- (1)
  tSessions <- liftIO $ readMVar _tsessions
      -- (2)
  let tsM = findSession tId tSessions
      -- (3)
  case tsM of
    Nothing -> do
      -- status was deleted after timeout or new transaction (4)
      tSessions <- liftIO $ takeMVar _tsessions
      -- (5)
      -- search again with lock:
      let pageVarM = findSession tId tSessions
          -- (6)
    Just _ -> do
      -- fallback
      liftIO $ putMVar _tsessions tSessions
      atomicTransactionPart cookieFun tr code
        -- (7)
```

First get the session ID from the `cookieFun`. If no cookie available, the function `cookieFun` is responsible for creating one with a new session ID. It is given as parameter so that the transaction framework itself is independent of the surrounding web framework. (1)

Then try to find the corresponding session in the contents of the MVar called `_tsessions` without taking a lock. (2), (3)

The case of `tsM == Nothing` means that the session is not known to the system. It either is a new one or the session has been deleted, e.g., because of a timeout¹¹. (4)

In order to insert the session into the frameworks session container, it is necessary to take a lock on the MVar `_tsessions`. (5)

To be sure that in the meantime no other thread has inserted the session, search for the session again, now with lock taken so that it is guaranteed to manipulate the session container exclusively. (6)

```
case pageVarM of
  Just _ -> do
    -- fallback
    liftIO $ putMVar _tsessions tSessions
    atomicTransactionPart cookieFun tr code
      -- (7)
```

¹¹Marker possibly new session in figure 4
If a session is found now, we just start over and use this session in the next run. For this, it is important to clean up the locks that were taken so far (7) and just restart the whole function (8).

```haskell
Nothing -> do  -- create new transaction instance:
  trVar <- liftIO $ newEmptyMVar  -- (9)
  let tSession = TSession trVar    -- (10)
  liftIO $ putMVar _tsessions $   -- (11)
      Map.insert tId tSession tSessions
  globalTouchedVals <- liftIO $ readMVar _trBaseTouchedValues  -- (12)
```

In case of Nothing (session was not found twice) a new session has to be constructed (13). The inner MVar of type MVar (TSessionVar k v status) is created empty and passed to the constructor of the new TSession. That way the required lock is already taken. (9), (10)

Now the new TSession has to be added to the session container and that container then put back to its MVar. (11)

For the state that the TSession needs to be run, the Touched Values have to be read from the _trBaseTouchedValues–MVar (12).

```haskell
let state =
    TSessionState
    tr
    tId
    Map.empty
    Map.empty
    False
    False
    _tsessioninitStatus
    globalTouchedVals
    runcode tSession Map.empty Map.empty
    _tsessioninitStatus tId state  -- (14)
```

After everything needed is in place, the state for the Trans–monad can be put together (13) and runcode is called. (14)

runcode is a separate function because it is called from two branches of atomicTransactionPart. See below.

---

(12) Marker session created by other thread in figure 4
(13) Marker new session in figure 4
(14) This is already at marker session locked in figure 4
4.3.1 runcode

runcode is defined within a where-block in the scope of atomicTransactionPart. Note that both branches of execution that call runcode have the MVar for the TSession (*.tSessionVar) already taken.

---

¹⁵Marker found session in figure 4

¹⁶This is also already at marker session locked in figure 4
runcode gets the references to read- and write-set and the status from before the session’s code was executed, and the session’s ID and the state as arguments. (1)

(4): To run the transaction part the record field function runTransactionPart (7) gets the session’s code (a function of type code :: Trans k v status a) and returns the inner state transformer monad: StateT (TSessionState k v status) IO (TransactionResult a) where code is argument of atomicTransactionPart.

The state transformer monad is then run with runStateT from module Control.Monad.Trans.State. (6)

Both combined, it gives an IO–function that takes a state and returns a tuple of the new state and the result of the TSession. (8)

-- The following lines are not part of runcode.
-- They just shows type signatures.

runStateT :: StateT s m a -> s -> m (a, s)  -- (6)
runTransactionPart :: Trans k v status a
                  -> StateT (TSessionState k v status) IO (TransactionResult a)

runStateT (runTransactionPart code)  -- (8)
     :: TSessionState k v status
       -> IO (TransactionResult a, TSessionState k v status)

Because the session’s code can throw exceptions, it is necessary to catch those here, so that runcode can release the locks. (3)

When an exception is caught, return a tuple of Exception :: TransactionResult a and the unchanged state. (5)
The result tuple is pattern matched in (2) where the record fields of the state are bound with the help of Record Wild Cards (page 55).

```haskell
timeout <- liftIO $ addUTCTime _tsessionTimeOut <$> getCurrentTime -- (9)
liftIO $ updateTimeout tr tId (Just timeout) -- (10)

case transactionResult of
    Exception e -> do -- (11)
        liftIO $ putMVar _tsessionVar $ TSessionVar
            timeout
            oldReadSet
            oldWriteSet
            oldStatus
        Control.Exception.throw e -- (12)
```

The session now needs a new date of timeout. That is the timeout amount of the transaction base plus the current local time. (9)
The function updateTimeout updates the timeout in the timeout index. (More information on that in section Timeout Handler on page 73) (10)
In case the transaction result from above turned out to be an Exception, it is time to release the lock: The only lock taken is the session’s TSessionVar. This is put back with the new timeout and the read- and write-set and the status from the old state17. (11)
Finally, the exception is thrown to the calling monad. (12)

If the result did not turn out to be an Exception but Success a, then several steps have to be completed:

- Check if transaction part is valid (like in STM)
- Check if committing is requested
- Check if transaction is requested to be finished (deleted)

17Marker error in figure 4
Success a -> do
    -- check if values from read set are saved in the mean time: ↓ (13)
    let newValsInReadSet = Map.difference _tsessionReadSet oldReadSet
    globalTouchedVals <- liftIO $ takeMVar _trBaseTouchedVals -- (14)
    let invalid = or $ map f (Map.toList newValsInReadSet) -- (15)
      where
        f :: (k, TouchedValue v status) -> Bool -- ↓ (16)
        f (k, touchedValRS) = case Map.lookup k globalTouchedVals of
            Nothing -> False -- ↓ (17)
            Just touchedVal -> case lookupValueCommitted touchedVal of
                Nothing -> False
                Just v -> getVal touchedValRS /= v -- (18)

where

getVal :: TouchedValue v status -> v

To prepare for handling the first bullet point, first the fresh Touched Values in the read set have
to be collected. (13)

A fresh Touched Value in read set is a Touched Value that has been added to the read set
within the last run of the transaction part. In contrast to an existing value which ex-
isted in the read set before the effective atomicTransactionPart was started.

The transaction part is invalid if a value, corresponding to key k, has been committed to the
database (by another transaction) in the mean time (while the effective transaction part
was running) and there is a fresh Touched Value in the read set corresponding to the same
key k. Then the transaction would have computed its result with outdated data; Or from
another perspective: The committed value (of the other transaction) would be ignored
(Lost Update/Write–write conflict).

Global Touched Values are the Touched Values that are stored in the MVar
_trBaseTouchedValues that contains a Map with a List of type [TouchedValue]
to every key k. In contrast to the Touched Values that each TSession has stored in its
read- and write-sets, which have a single TouchedValue as Map value. When a server
uses several transaction bases, then each transaction base has its own global Touched
Values.

Then a critical block is entered where the global Touched Values have to be manipulated exclu-
sively by one thread. This is because a change in the global Touched Values would make the
testing for invalidity outdated. (14)
To check for invalidity, every new value in the read set is mapped to type Bool by function f,
and, if at least one of them gets mapped to True, the transaction part is invalid (logical disjunction). The function `Map.toList` returns a list of tuples of key and values of the map. Here that is type: `(k, TouchedValue v status)` (15)

A committed value with key `k` is kept in the global Touched Values as a (passive) notification of database change for active transactions. It remains there as long as it exists at least one transaction that has a Touched Value with key `k` in its read set or write set. In other words: Key `k` will be removed from the global Touched Values if the Map-entry of `k` is a list with only the ValueCommitted entry (or an empty list).

Function `f` does two things:
First, check if the given key of the fresh Touched Value is available in the (global) Touched Values. (16)
If this is not the case, then return `False`, meaning this fresh Touched Value is valid.
Secondly, if a list of corresponding Touched Values is found, check if it contains a ValueCommitted entry. (Constructor of data type TouchedValue, page 41). (17)

Note that `touchedVal :: [TouchedValue v status]` and `lookupValueCommitted :: [TouchedValue v status] -> Maybe v`.
If no value was committed (case `Nothing`), then this new Touched Value is valid.
If a commit entry is found, it is only relevant if it is not equal to the value in the transaction’s read set. This is either the case if the commit entry is old (from before the call of `atomicTransactionPart`) or the value was committed that is equal to the read set value. Both cases are valid. Transaction part is invalid when read set value is not equal to (prior) committed value. (18)

```haskell
if invalid
then do -- rollback
  liftIO $ putMVar _trBaseTouchedValues globalTouchedVals
  liftIO $ putMVar _tsessionVar $ TSessionVar timeout oldReadSet oldWriteSet oldStatus
  atomicTransactionPart cookieFun tr code
```

If transaction part is invalid, then a rollback has to be performed¹⁸. Both of the open locks have to released. The MVar `_trBaseTouchedValues` is put with `globalTouchedVals` which has not

¹⁸Marker invalid in figure 4
been changed since last `takeMVar`. (14), (19)
The MVar `_tsessionVar` gets the `TSessionVar` with the old values which it contained before but with new timeout. (20)
Then the rollback is initialized by restarting the whole function `atomicTransactionPart` with the exact same parameters. (21)

- Check if committing is requested

```haskell
else do -- not invalid
  if _tsessionCommit -- (23)
    then do
      -- write write-set to data base:
      liftIO $ -- (24)
      Map.traverseWithKey -- (25)
        (\k (ValueWritten _ v _) -> _trBaseSaveFun k v) -- (26)
      _tsessionWriteSet -- (27)
      let globalTouchedVals' = -- (28)
          Map.foldrWithKey -- (29)
            turnWrittenToComitted -- (30)
          globalTouchedVals _tsessionWriteSet
      liftIO $ putMVar _trBaseTouchedValues globalTouchedVals'--(31)
```

The boolean `_tsessionCommit` indicates if a commit is requested by the session’s code (by calling `commit` inside the `TransMonad`). (23)
If so, the transaction’s save function¹⁹ is run with each key-value pair from the write set. (24)–(27)
Each key-value pair is passed to the function `_trBaseSaveFun` with usage of `traverseWithKey` (25) where the value is deconstructed from type `TouchedValue`. As the value is in the write set, it can only be constructed with `ValueWritten`. (26)
 `_trBaseSaveFun` returns type `IO ()` which is why `liftIO` is necessary. (24)

-- The following lines are not part of runcode.
-- They just shows type signatures.
`traverseWithKey :: Applicative t => (k -> a -> t b) -> Map k a -> t (Map k b)`
`Map.foldrWithKey :: (k -> a -> b -> b) -> b -> Map k a -> b`
`turnWrittenToComitted` :: `Ord k` => `k` -> `TouchedValue v status` -- each element of _tsessionWriteSet
  -> `Map.Map k [TouchedValue v status]` -- globalTouchedVals

¹⁹see section Transaction Base on page 34
4.3 atomicTransactionPart

\[ \text{Map.Map } k \ [\text{TouchedValue } v \ \text{status}] \quad \text{-- globalTouchedVals'} \]

Map.foldrWithKey turnWrittenToComitted :: Ord k

\[ \text{Map.Map } k \ [\text{TouchedValue } v \ \text{status}] \quad \text{-- globalTouchedVals} \]

\[ \text{Map.Map } k \ (\text{TouchedValue } v \ \text{status}) \quad \text{-- _tsessionWriteSet} \]

\[ \text{Map.Map } k \ [\text{TouchedValue } v \ \text{status}] \quad \text{-- globalTouchedVals'} \]

Note that the commit phase is in a critical section and protected by MVar _trBaseTouchedValues.

The global Touched Values have to be cleaned up after committing because the “touched values” are not only “touched” anymore but stored permanently to database. This means, for every key \( k \) in write set, the ValueWritten entry in the list for key \( k \) in global Touched Values with the TSession’s ID has to be removed and a respective ValueCommitted has to be inserted at the head of the list, and, if available, the old ValueCommitted entry has to be deleted.

This is done repeatedly for each key by function turnWrittenToComitted. \((29), (30)\)

<table>
<thead>
<tr>
<th>Touched Values of TSession</th>
<th>What to do with global Touched Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ValueWritten id v status)</td>
<td>Insert (ValueCommitted id v) and filter out all Touched Values with id and all ValueCommitted</td>
</tr>
</tbody>
</table>

Table 3: How to handle Touched Values removal after commit for each key

Then the new version of global Touched Values is put back to the MVar _trBaseTouchedValues and thus the critical section is left. \((31)\)

\[ \text{else do -- no commit} \]

\[ \text{-- if val in write and read set, use val from write set (argument order of union):} \]

\[ \text{let unionReadWriteSet =} \quad \text{-- (32)} \]

\[ \text{Map.union _tsessionWriteSet _tsessionReadSet} \]

\[ \text{let globalTouchedVals' =} \quad \text{-- (33)} \]

\[ \text{manageTouchedValues globalTouchedVals tId unionReadWriteSet} \]

\[ \text{liftIO}$ \ putMVar _trBaseTouchedValues globalTouchedVals' \]

In case no commit is requested, the global Touched Values have to be merged with the new Touched Values of the TSession’s read- and write-set.

The new set of the Touched Values, which is to be merged to the global Touched Values, is the union of read- and write-set. The argument order of function Map.union defines which one of two values corresponding to one key is prioritized if one value is in the read and one in the write set. Here, when in conflict, values from write set override values from read set. \((32)\)
4 Implementation

-- The following lines are not part of runcode.
-- They just shows type signatures.

manageTouchedValues :: Ord k
                   => Map.Map k (TouchedValue v status) -- globalTouchedVals
                   -> TSessionId
                   -> Map.Map k (TouchedValue v status) -- unionReadWriteSet
                   -> Map.Map k [TouchedValue v status] -- globalTouchedVals'

The function manageTouchedValues then adds the Touched Values of the union of read- and write-set to the global Touched Values, where each new value is added to the head of the list for the respective key k²⁰. (33)

Next, the TSession's critical section can be left by putting back the TSessionVar to the TSession's MVar with the updated timeout and the updated status and read- and write-set. (34)

  • Check if transaction is requested to be finished (deleted)

    when _tsessionFinish $
    liftIO $ removeSessions tr Nothing [_tsessionId]    -- (35)
    return a                                            -- (36)

Last but not least, the session is to be removed if it was requested which is indicated by boolean _tsessionFinish. See section Removing Sessions on page 75 for details. (35)

Finally, function atomicTransactionPart returns the return data of TSession’s code. (36) a is bound at line 76 by deconstructing transactionResult to Success a.

4.4 Functions of Trans–monad

The functions of type Trans k v status a work with the TSessionState. Function getStatus is trivial. It looks up the status inside the state and returns it. The variable _tsessionStatus is bound with the help of the language extension Record Wild Cards.

²⁰Remember: The lists are ordered by occurrence. (cp. chapter Touched Values, page 40)
4.4 Functions of Trans-monad

getstatus :: Trans k v status status
getstatus = Trans $ do
    state@TSessionState{..} <- get
    return $ Success _tsessionStatus

Function commit is just as trivial. It sets the boolean record field _tsessionCommit of the state within the monad to True and returns ()

commit :: Trans k v status ()
commit = Trans $ do
    state <- get
    put $ state { _tsessionCommit = True }
    return $ Success ()

Function finishSession operates analogously to above but with record field _tsessionFinish instead of _tsessionCommit. Both boolean record fields are flags for instructing atomicTransactionPart.

finishSession :: Trans k v status ()
finishSession = Trans $ do
    state <- get
    put $ state { _tsessionFinish = True }
    return $ Success ()

The function setStatus not only sets the status within the TSessionState but also overrides the status in every element in read- and write-set. The latter is important for inter-transaction-communication. Other transactions can access the status by reading a value and checking the status of the Touched Values.

The function expects the status as argument. The status for the elements in read- and write-set are manipulated by a map function.

setStatus :: Eq status => status -> Trans k v status ()
setStatus status = Trans $ do
    state@TSessionState{..} <- get -- Record Wild Cards extension
    when (_tsessionStatus /= status) $ do
        let transactionReadSet = Map.map
            (\(ValueRead id v _) -> (ValueRead id v status))
            _tsessionReadSet
Function `writeVal` manipulates the write set. The function fetches the state and inserts the given key-value pair to the write set–map within the state. The key-value pair has to be wrapped with `ValueWritten` because if the data structure design²¹.

Function `readVal` expects a key `k` and returns a tuple of the corresponding value `v` –if available– and the corresponding Touched Values. The value `v` is retrieved either from database or read- or write-set. How to decide where the data is to be loaded from is outlined in the following table.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Read Set</th>
<th>Write Set</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing</td>
<td>Just <code>a</code></td>
<td><code>a</code></td>
<td></td>
</tr>
</tbody>
</table>

²¹cp. figure 5: Data Structure, page 87
4.4 Functions of Trans–monad

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Read Set</th>
<th>Write Set</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just b</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Just b</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>Just c</td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Just a</td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Just b</td>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Just b</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

Table 4: Return Values of `fst <$> (readVal k)` in relation to values in database and read- and write-set. Compare to table 1 on page 39

```haskell
readVal :: Ord k => k -> Trans k v status (Maybe v, TouchedValues v status)
readVal k = Trans $ do
  state@TSessionState{..} <- get                                  -- (1)
  let lookup_touchedVals = fromMaybe [] $ Map.lookup k _trBaseTouchedValuesCopy -- (2)
  let touchedVals = TouchedValues -- (3)
      from
      { _touchedValuesAll = lookup_touchedVals
        , _touchedValuesAfterLastLoad = takeWhile filterTId lookup_touchedVals
        , _touchedValuesWithoutMe = filter filterTId lookup_touchedVals
      }
    where
      filterTId :: TouchedValue v status -> Bool
      filterTId tVal = (getId tVal) /= _tsessionId
```

First, the state is retrieved. (1)
The Touched Values corresponding to the given key `k` are looked up in `_trBaseTouchedValuesCopy`, which is a record field variable of `TSessionsState` bound with the help of Record Wild Cards extension. If `Map.lookup` returns `Nothing`, then `fromMaybe []` will transform it into an empty list. Otherwise, the returned list of Touched Values will be unwrapped from the `Just` constructor. (2)
The TouchedValues data type consists of three lists where the last two are filtered sublists of the first one. These lists are bound to the constructed data type’s record fields. (3)
As seen in table 4, if a value is found in write set, this is the one to be returned. So, before checking the database or read set, lookup the key k in write set. (4)
If found, then add the value to the read set and write back state with the new read set to the monad. (5)
Return the value and the Touched Values. (6)
In case of no match in write set, table 4 specifies that a possible value in read set is to be returned before loading from the database. So, next check for the key k in read set. (7)
If found there, it is not necessary to manipulate the read set because it was just loaded from there. So, just return value from the read set and Touched Values. (8)

If the key k did not give results in write- nor in read-set, then the returned value has to be loaded from database. Since the ValueCommitted–entries in the Touched Values are always
equal to the values in the database, searching there for a match makes them a cache. In case of a match, insert value to read set and return the tuple of value and Touched Values as before. lookupValueCommitted is defined at ll. 61. (9)

```
-- finally load from DB:
let tr@TransactionBase{} = _tsessionBase          -- (10)
vM <- liftIO $ _trBaseLoadFun k                  -- (11)
case vM of                                       -- (12)
  Just v -> do
    put $ state
    { _tsessionReadSet =
      Map.insert
        k
        (ValueRead _tsessionId v _tsessionStatus)
        _tsessionReadSet
    }
    return $ Success (Just v, tuchedVals)        -- (13)
  Nothing -> return $ Success (Nothing, tuchedVals)  -- (14)
```

The last possibility of finding a value to the given key k is loading it from the database with the database load function defined in the transaction base: _trBaseLoadFun :: k -> IO (Maybe v) (11) The function name _trBaseLoadFun is bound in (10) with the help of Record Wild Cards. If the load function returned a value to the given key k, then insert it to the read set and return value and Touched Values as before. (12),(13) Otherwise, return a tuple of Nothing and the Touched Values. (14)

```
lookupValueCommitted :: [TouchedValue v status] -> Maybe v
lookupValueCommitted [] = Nothing
lookupValueCommitted ((ValueCommitted _ v):_) = Just v
lookupValueCommitted (_:xs) = lookupValueCommitted xs
```

4.5 Timeout Handler

Sessions need to be sorted out after a given period of time in which they have not been accessed. This is done by a separate garbage collection thread for every TransactionBase which is spawned at creation time by the function createTransaction. Its task is to check for timed out sessions, delete them and clean up the Touched Values of these sessions. The cleanup is initiated at regular intervals depending on the defined timeout period.

The function timeoutHandler gets the TransactionBase as argument. (1)
The garbage collection thread sleeps between cleanups for a duration of a tenth of the timeout. This means that a session can survive no longer than its timeout plus a tenth of it\(^2\) after its last request. (2)

The transaction base holds a timeout index exclusively for the timeout handler: 

\[ \_\text{tsessionTimeOutIndex} :: \text{IORef } (\text{Map.Map TSessionId Timeout}) \]. (3)

It is a Map of session IDs and the corresponding moment of timeout, wrapped in an IORef necessary for access of different threads. (To remember: type Timeout = UTCTime).

To find all timed out sessions a folding over the timeout index is performed. (4)

The module Map offers the fold function

\[
\text{foldlWithKey} :: (a \to k \to b \to a) \to a \to \text{Map k b} \to a
\]

where \(k\) is the key and \(b\) the value of the map. In this case \(k\) is assigned to TSessionID and \(b\) to Timeout. The result of foldlWithKey is of type \(a\) which is assigned to [TSessionId]. The starting or rather identity element of the fold is an empty list.

The passed function \(f\) that is called with every single element of the timeout index, simply

\[^2\]\text{plus the time of the timeout handler needs to run to be exact}
4.6 Removing Sessions

The following function takes care of removing sessions respective to the given session IDs and cleaning up all their related data. As arguments it expects the transaction base, an optional argument that indicates the point of time for a timed out sessions and the list of session IDs that are to delete. (1)

\[
\text{removeSessions} : \text{Ord k} \\
\rightarrow \text{TransactionBase k v status} \\
\rightarrow \text{Maybe UTCTime} \\
\rightarrow [\text{TSessionId}] \\
\rightarrow \text{IO ()}
\]

\[
\text{removeSessions tr@TransactionBase{..} nowM trIdsToRemove = do} \\
\text{tSessions <- takeMVar _tsessions} \\
\text{-- delete given Instances and collect TouchedValues that are to delete:} \\
\text{(touchedValsToDelete, tSessions') <-} \\
\text{foldlM (removeSessionAndFindTVtoDel nowM) ([], tSessions) trIdsToRemove} \\
\text{putMVar _tsessions tSessions'} \\
\text{touchedVals <- takeMVar _trBaseTouchedValues} \\
\text{let touchedVals' = cleanupTouchedValues touchedVals touchedValsToDelete} \\
\text{putMVar _trBaseTouchedValues touchedVals'}
\]

The interaction of this function with the several MVars is drafted in figure 4 on the right hand side of page 79. First, it holds a lock on _tsessions in (2) – (4) and then after release it holds a lock on _trBaseTouchedValues in (5) – (7). The tryTakeMVar–call on _sessionVar is part of the monadic folding in (3).

removeSessions does not block all necessary MVars at ones and then takes care about the data, instead it works in two blocks. First the MVar “_tsessions” is taken for the TSessions to be removed and after the lock has been released again the second MVar “_trBaseTouchedValues” is taken for taking care of the Touched Values. The advantage is less lock overhead because of coarse lock granularity. Between these two locks other processes can access and manipulate data.
4 Implementation

Why is this correct? One can argue that sessions that get executed between the `putMVar` of `_tsessions` and the `takeMVar` of `_trBaseTouchedValues` get wrong data to work with and the outcome of the session results in wrong data because the session has access to data of at least one session that has been removed shortly before.

Answer: It does not matter. Imagine a session was executed just before the `takeMVar` of `_tsessions` so that the result is undoubtedly correct. Then the resulting web page is sent to the requesting browser and arrives there after the other session is deleted. The result is the same, although now everything is done in correct order. Since a web page is not dynamically updated, it can show outdated data. It can already be outdated when it arrives at the browser, so it does not make any difference to be absolutely exact in this matter.

4.7 Clean up Touched Values

This section describes how given Touched Values are removed from the global Touched Values. The function `cleanupTouchedValues` is defined inside a where-block in scope of `removeSessions`. It gets as arguments the global Touched Values and a list of tuples of key and Touched Value pairs that are to remove and returns the new global Touched Values. The function is purely functional. Thus, the calling function (`removeSessions`) has to take care of the locks. The global Touched Values’ structure is a `Map` of keys `k` and corresponding lists of Touched Values for the respective keys. So, the function has to

1. iterate over all key-value pairs that are to delete,
2. lookup the list of Touched Values for each key in the global Touched Values–map and
3. filter out the Touched Values in the list found corresponding to the effective key.

```haskell
cleanupTouchedValues :: Ord k => Map.Map k [TouchedValue v status] -- ^ global Touched Values (input)
                      -> [(k, TouchedValue v status)] -- ^ list of Touched Vals to remove
                      -> Map.Map k [TouchedValue v status] -- ^ global Touched Values (output)
cleanupTouchedValues touchedVals touchedValsToDelete =
  foldl cleanupTouchedValue touchedVals touchedValsToDelete
```

1. Iteration over all key-value pairs is done with the help of `foldl` over `touchedValsToDelete`. That way the helping function `cleanupTouchedValue` is called repeatedly with each key-value pair. The starting element of the fold is the Touched Values–map which is manipulated by each call of `cleanupTouchedValue` and then returned.
**4.7 Clean up Touched Values**

```haskell
4.7 Clean up Touched Values

```cleanupTouchedValue
:: Ord k => Map.Map k [TouchedValue v status] -- ^ global Touched Values (input)

- (k, TouchedValue v status) -- ^ Pair of key and TV to remove

- Map.Map k [TouchedValue v status] -- ^ global Touched Values (output)

```cleanupTouchedValue touchedVals (k, touchedVal) =
Map.alter (delIdInList (getId touchedVal)) k touchedVals
```

2. **cleanupTouchedValue** gets the global Touched Values and a key–Touched Value pair as arguments. Its task is to remove the given Touched Value from the global Touched Value–Map.

To accomplish that, the list, found at key \( k \) in the Map, has to be manipulated. This list has type \([TouchedValue v status]\). The Touched Value to be removed in this list is the one which has the same TSessionID as the Touched Value in the argument-tuple.

The Map is modified with the help of Map.alter which expects a function \( f \) of type \((Maybe a -> Maybe a)\), the key \( k \) and the Map that is to be modified.

```haskell
Map.alter :: Ord k => (Maybe a -> Maybe a) -> k -> Map k a -> Map k a with
  a - [TouchedValue v status]
```

The function \( f \) is given Nothing if the effective key is not found in the map and if it is found, \( f \) is given the value with the Just constructor.

\( f \) can return Just with a modified value or Nothing to delete the key in the map.

Here Map.alter is called with delIdInList in combination with the effective TSessionId.

```haskell
delIdInList :: TSessionId -- ^ ID to remove

- Maybe [TouchedValue v status] -- ^ Touched Value List (input)

- Maybe [TouchedValue v status] -- ^ Touched Value List (output)

delIdInList _ Nothing = Nothing

delIdInList tId (Just valList) =
  let newValList = filter filterTId valList in
  if isEmptyOrValueCommitted newValList
    then Nothing
    else Just newValList

  where

  filterTId (ValueRead tId’ _) = tId /= tId’
  filterTId (ValueWritten tId’ _) = tId /= tId’
  filterTId _ = True

  isEmptyOrValueCommitted [] = True
  isEmptyOrValueCommitted [(ValueCommitted _ _)] = True
  isEmptyOrValueCommitted _ = False
```

77
3. The helper function delIdInList gets a TSessionId and a list of Touched Values (within Maybe). Its task is to delete every Touched Value with the given TSessionId except the ValueCommitted entries in the given list. The committed values have to remain because they act as a notification for database changes. They can only be removed if there is no active transaction that has a Touched Value with the corresponding key \( k \). In other words: If the given list remains with only one entry and that entry is a ValueCommitted, then the entire key \( k \) can be deleted in the global Touched Values. In that case the function returns Nothing.

4.8 Overview MVars

The TSession module defines three different MVars that are used for different purposes. In figure 4 on page 79 you can see a graphical sketch that shows what the three different types of MVars do and how they interact with each other. With that sketch I will also later argue that the concept is deadlock free.

The graphic shows two columns. One stands for each major function that uses the MVars. That is atomicTransactionPart and removeSessions. All other functions (besides these major functions that use the MVars) are only called from within these two. So they are implicitly included and can be left out.

The MVars are symbolized by the blue vertical blocks that show the MVars' names inside of them. These blocks indicate three sub columns for each function column. The text above the MVar names shows "global" and "local". This is to indicate that the global MVars are constructed at the creation of the TransactionsBase and there is only one instance each. In contrast to the local MVar of which one is constructed for each session.

The MVar called _trBaseTouchedValues contains all key-value pairs of all sessions of the corresponding TransactionBase. It is important to know that the MVar has to be taken for modifying the key-value store and is only read (not taken) for a copy of the key-value store that the sessions get for accessing.

The _tSessions–MVar contains all sessions. It has to be taken for inserting new sessions and deleting old ones.

Finally the _sessionVar contains the data for one session. For each session one _sessionVar is created.

Beneath the MVar blocks you can see a schematic flow of code execution from top to bottom concentrated to show only the MVar specific parts. The ovals that show the interactions with MVars are sorted into the MVar columns they interact with. A single horizontal line separates different parts of the code and a double horizontal line indicates the end of execution (of the function). The vertical lines that start at a “take”–oval indicate that the MVar of that column is taken until the respective line stops at a “put”–oval.

The arrows right and left of the MVar columns are possible execution branches.
Figure 4: Sketch of interaction with MVars
4 Implementation

4.8.1 Removing sessions

It is easier to start with `removeSessions` because it is less complex. As mentioned, for deleting a session `_tsessions`–MVar has to be taken. This can be seen in the first step. After it is successfully taken the `_sessionVar` can be deleted. How this is implemented in detail, see page 75. After deletion, `_trBaseSession` will be put again with the new set of `_sessionVars`. After that the “Touched Values”–MVar will be taken to delete the values that are not needed anymore.

4.8.2 Creating and reusing sessions

Now I will show how `atomicTransactionPart` handles the MVars. This function is called with the web-developer’s code as argument which it is to execute. It can contain errors, `commit` or no `commit`, or it can turn out to have an invalid result because the read set changed while the code was being executed. In that case a rollback is to be performed. If the browser that sends the request has no session ID or a timed-out ID, a new session has to be initialized.

In the following, all possible branches will be depicted.

4.8.2.1 New session

With every request the session ID is transmitted by a cookie value or a new one is created. In the following, timed-out and new generated session ID are treated as the same. To check if the session ID is already known the `_tsessions`–MVar is first read. In case the session ID is not found in the containing data structure, the session has to be added to it. To do this, the MVar has to be taken to lock it. You can see this symbolized by the arrow that points from the read–oval to “possibly new session”. It says “possibly” because in the mean time another thread could have inserted the session ID. To find out the `_trBaseSession`–MVar is taken so that this thread is guaranteed to be the only one to manipulate the contents of the MVar. While taken the session is searched again. It is most likely not to be found but if so, then the execution jumps to “session created by other thread”. Then the function orders to put back the one MVar that is taken with unchanged data and restart the function shown by the arrow that points up to the start. In the next run the function will behave differently because the session ID is now known.

Rewinding to the last branch where this time the other option is taken. The case where the session ID is actually new. MVar is taken so that a new session can be safely created and inserted into the data structure that stores the sessions. You can see the “new”–oval that symbolizes the creation of a taken (end therefore empty) `_sessionVar` (Control.Concurrent.MVar.newEmptyMVar). The `_sessionVars` have to be taken while run-

---

23The MVar is just read first because it does not need to be taken for existing sessions. Only reading will not block other threads in that case.
ning the session code. Now the insertion of the new session is done and the _tsessions–MVar can be put again.

Result for now:

- The corresponding _sessionVar for the specific session is taken.
- No other MVar is taken.

4.8.2.2 Finding Existing Session

Rewinding to the start of the function where a “read” of _tsessions takes place. If the session ID sent by the cookie value is found in it, it is a session continuation. The branch to “found session” will then be chosen. To run the code given by the web-developer as argument, the session has to be locked. So the _sessionVar is to be taken.

Result for now:

- The corresponding _sessionVar for the specific session is taken.
- No other MVar is taken.

4.8.2.3 Running Session’s Code

Now the state of the two branches for creating a new session and continuing a known session in term of locked MVars is identical. So it is safe to merge the two paths of execution. Let us assume the process of execution is at the label “session locked”. The session’s code needs access to the Touched Values which include the key-value sets of all other sessions and a short\(^{24}\) history of changes. The session gets a copy of the Touched Values so that the _trBaseTouchedValues–MVar is not blocking the other threads while running. To get the copy, the MVar is just read as shown by the “read”–MVar. Now the code can be run. All sessions with different IDs can run concurrently because the only MVar blocking is each session’s own one. That means only other sessions from the same browser (with the same ID) have to wait for this one to complete. 

If the code produces an error, it will be caught so that the _sessionVar can be put again (See left arrow).

After the session’s code is complete, all values of the session’s read set have to be compared to the original (not the copy) of the Touched Values. Also, the sessions new key-value pairs have to be merged with the global Touched Values. For that the _trBaseTouchedValues–MVar is used to block the data access.

It can turn out that a value which is part of the session’s read set changed while the session’s code was being executed. In that case the state is invalid and a rollback has to be fulfilled. See

\(^{24}\)a history of all relevant changes
branch to the marker “invalid” on the right. In this case both MVars that are taken have to be put back and `atomicTransactionPart` is called again with the same arguments. In case of a valid result both MVars are also put back and the function successfully returns with the session’s result.

4.8.3 Deadlock Avoidance

It surely is no proof! But with good argumentation it should be clear that this sketch shows that concurrent execution of these functions should be deadlock free.

4.8.3.1 Nested MVars

According to Coffman et al. [8], for a program to be deadlock free, at least one of the four conditions mentioned in section Deadlocks (see page 9) has to be dissatisfied at all times.

The first and the third conditions are always true for MVars. So the solution must be found in condition two or four.

To proof that there will never be a “circular waiting” of tasks, one can assign numbers consecutively to all MVars and show that each process takes all MVars in ascending order and thereby eliminating the possibility of a circular chain. Let’s say `_trBaseTouchedValues` is assigned to 1, `_tsessions` is assigned to 2, and all the other `_sessionVar`s get a number in order of creation starting at 3.

MVar 1 and 2 are never simultaneously taken by one thread. This is shown by the vertical lines that indicate the lock of the two MVars which are never (horizontally) neighboring. In other words, no horizontal line can be drawn that intersects both a lock-indicating line of MVar 1 and a lock-indicating line of MVar 2.

The chart shows that one process takes MVar 2 and the same process then takes an MVar n with n > 2 without prior releasing MVar 2. It does not matter what n is because every process only takes one MVar n with n > 2 at a time and therefore the ascending order is satisfied.

4.8.3.2 Taking MVars without Releasing

A process that takes an MVar and does not release it when leaving a “semantical block” in which the MVar is allowed to be taken keeps the MVar’s lock forever. That would not only block all other processes forever that wait to take this MVar, but it could also happen that the same process is ordered to take the MVar again and then has to wait for itself.

The chart clearly shows that at every possible end of the functions, all MVars are released.
5 Perspectives

This thesis is one proposal of interactive and communicating transactions with session management. It takes aspects from good concepts like STM. Haskell as programming language was a good choice as it eliminates most errors through type safety.

The main focus was defining a clear API, correctness and deadlock avoidance. Memory usage, efficiency and performance were not the main concern. But of course, for this framework to be in productive service, it first needs serious testing.

In my opinion the API is clearly defined and easy to learn. The developer can define clear transaction instructions without worrying about the problems occurring with concurrency. I was myself surprised how rapidly I could implement the user-edit example application.

Software is said to never be in final stage. So I would be happy if this framework will be maintained, extended and/or corrected by interested developers. Here are some ideas:

Proposals

- Every TSession gets a copy of the Touched Values with every run. This can surely be implemented in a more elegant way. One proposal for the next release is to introduce a versioning system for the Touched Values, similar to the TVar versions in STM.

- More interactivity: Modern web technology offers dynamic data exchange in the background of fully loaded pages. This technique in combination with the transaction framework would allow new possibilities if a transaction got actively notified with Touched Values when relevant data is being changed by other active transactions. As a result, a displayed web-page could be dynamically updated. As an example, the warning messages of the user-edit application would show up instantly and without reloading the entire web page.

- Communication between TSessions of different transaction bases.

- Allow other processes to manipulate database. Because of cache, changes from outside of the transaction framework can happen to be ignored.
6 Appendix

6.1 Building Environment

Machine

$ uname -rsm
Linux 3.8.0-38-generic x86_64

$ lsb_release -d
Description: Ubuntu 12.04.4 LTS

GHC Version

$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.4.1

GHC Package Listing

$ ghc-pkg list # (output rearranged)
/var/lib/ghc/package.conf.d

    Cabal-1.14.0                  ghc-7.4.1
    GLUT-2.1.2.1                 ghc-prim-0.2.0.0
    HTTP-4000.2.2                haskell-src-1.0.1.5
    HUnit-1.2.4.2                haskell2010-1.1.0.1
    MonadCatchIO-mtl-0.3.0.4     haskell98-2.0.0.1
    OpenGL-2.2.3.1               hoopl-3.8.7.3
    QuickCheck-2.4.2             hpc-0.5.1.1
    array-0.4.0.0                html-1.0.1.2
    base-4.5.0.0                 integer-gmp-0.4.0.0
    bin-package-db-0.0.0.0        mtl-2.0.1.0
    binary-0.5.1.0               network-2.3.0.10
    bytestring-0.9.2.1           old-locale-1.0.0.4
    cgi-3001.1.8.2               old-time-1.1.0.0
    containers-0.4.2.1           parallel-3.2.0.2
    deepseq-1.3.0.0              parsec-3.1.2
    directory-1.1.0.2            pretty-1.1.1.0
    extensible-exceptions-0.1.1.4 process-1.1.0.1
    fgl-5.4.2.4                  random-1.0.1.1
    filepath-1.3.0.0             regex-base-0.93.2
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex-compat</td>
<td>0.95.1</td>
</tr>
<tr>
<td>regex-posix</td>
<td>0.95.1</td>
</tr>
<tr>
<td>rts-1.0</td>
<td></td>
</tr>
<tr>
<td>stm-2.2.0.1</td>
<td></td>
</tr>
<tr>
<td>syb-0.3.6</td>
<td></td>
</tr>
<tr>
<td>template-haskell</td>
<td>2.7.0.0</td>
</tr>
<tr>
<td>text</td>
<td>0.11.1.13</td>
</tr>
<tr>
<td>time</td>
<td>1.4</td>
</tr>
<tr>
<td>transformers</td>
<td>0.2.2.0</td>
</tr>
<tr>
<td>unix</td>
<td>2.5.1.0</td>
</tr>
<tr>
<td>xhtml-3000</td>
<td>2.0.5</td>
</tr>
<tr>
<td>zlib</td>
<td>0.5.3.3</td>
</tr>
</tbody>
</table>

```
/home/fmi/.ghc/x86_64-linux-7.4.1/package.conf.d
```
### Appendix

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>th-lift</td>
<td>0.5.6</td>
</tr>
<tr>
<td>th-orphans</td>
<td>0.7.0.1</td>
</tr>
<tr>
<td>threads</td>
<td>0.5.0.2</td>
</tr>
<tr>
<td>time-compat</td>
<td>0.1.0.3</td>
</tr>
<tr>
<td>transformers-base</td>
<td>0.4.1</td>
</tr>
<tr>
<td>transformers-compat</td>
<td>0.1.1.1</td>
</tr>
<tr>
<td>tsession</td>
<td>0.1</td>
</tr>
<tr>
<td>tsession-happstack</td>
<td>0.1</td>
</tr>
<tr>
<td>unordered-containers</td>
<td>0.2.3.3</td>
</tr>
<tr>
<td>utf8-string</td>
<td>0.3.7</td>
</tr>
<tr>
<td>vector</td>
<td>0.10.9.1</td>
</tr>
<tr>
<td>web-routes</td>
<td>0.27.2</td>
</tr>
<tr>
<td>web-routes-happstack</td>
<td>0.23.4</td>
</tr>
<tr>
<td>web-routes-hsp</td>
<td>0.23.0</td>
</tr>
<tr>
<td>web-routes-th</td>
<td>0.22.1</td>
</tr>
</tbody>
</table>
data TransactionBase k v status = TransactionBase 

\begin{verbatim}
  _trBaseName :: String
  _trBaseLoadFun :: k \rightarrow IO (Maybe v)
  _trBaseSaveFun :: k \rightarrow IO ()
  _trBaseTouchedValues :: Map k ([TouchedValue v status])
  _tSessions :: TSession k v status
  _tSessionIdGenerator :: TransactionIdGenerator
  _tSessionInitStatus :: status
  _tSessionTimeout :: NominalDiffTime
  _tSessionTimeoutIndex :: IORef (Map TSessionId Timeout)
\end{verbatim}

---

data TSessionState k v status = TSessionState 

\begin{verbatim}
  ( _tSessionBase :: TransactionBase k v status
    , _tSessionId :: TransactionId
    , _tSessionReadSet :: Map k ([TouchedValue v status])
    , _tSessionWriteSet :: Map k ([TouchedValue v status])
    , _tSessionCommit :: Bool
    , _tSessionFinish :: Bool
    , _tSessionStatus :: status
    , _trBaseTouchedValuesCopy :: Map k ([TouchedValue v status])
\end{verbatim}

---

data TouchedValue v status = TouchedValue 

\begin{verbatim}
  ValueCommitted TransactionId v | ValueRead TransactionId v status |
  ValueWritten TransactionId v
\end{verbatim}

---

data TouchedValues v status = TouchedValues 

\begin{verbatim}
  ( _touchedValuesAll :: [TouchedValue v status]
    , _touchedValuesAfterLastLoad :: [TouchedValue v status]
    , _touchedValuesWithoutTime :: [TouchedValue v status]
\end{verbatim}

---

type TSessions k v status = MVar (Map TSessionId (TSession k v status))

---

newtype TSession k v status = TSession 

\begin{verbatim}
  ( _tSessionVar :: MVar (TSessionVar k v status)
\end{verbatim}

---

data TSessionVar k v status = TSessionVar 

\begin{verbatim}
  ( _tSessionVarTimeout :: Timeout
    , _tSessionVarReadSet :: Map k ([TouchedValue v status])
    , _tSessionVarWriteSet :: Map k ([TouchedValue v status])
    , _tSessionVarStatus :: status
\end{verbatim}

---

type TransactionId 

\begin{verbatim}
  = Integer
\end{verbatim}

---

data TSessionIdGenerator = forall g. TSessionIdGenerator 

\begin{verbatim}
  ( _tSessionIdGeneratorRef: IORef g
    , _tSessionIdGeneratorGenFun: g \rightarrow (TSessionId, g)
\end{verbatim}

---

Figure 5: Data Structure
7 References

http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/HaerderReuter83.pdf


http://www.cse.ohio-state.edu/~agrawal/788-su08/Papers/week4/shavit95software.pdf


http://www.informatik.uni-kiel.de/~fhu/projects/stm.pdf

http://happstack.com

http://en.wikipedia.org/wiki/ACID