Asynchronous Actions in a Synchronous World

Asynchronous Share Rewriting in iTasks

Haye Böhm

January 14, 2019

Supervisor
prof. dr. dr.h.c. ir. M.J. Plasmeijer

Second reader
M. Lubbers MSc

Radboud University

Abstract

This thesis introduces a system for asynchronous communication in the iTasks framework. The framework is written in Clean, a pure, lazy, functional language. Tasks need to be able to access data in the system and retrieve data from all kinds of data sources. The share system allows tasks to read arbitrary data sources and provides a simple interface that allows composition of different data sources. This system allows tasks to share and store data in an efficient, re-usable way.

A disadvantage of the share system is that it does not allow asynchronous evaluation. When one task is using a share, other tasks have to wait for the full evaluation of this share before they can be evaluated. This has the effect that users in the iTasks framework must wait on other users. This results in poor user experience.

We implement a share system which, by way of *share rewriting*, allows asynchronous evaluation. The system can be used to communicate with arbitrary services on the internet, as well as to communicate between different iTasks servers in a distributed context.

We show how asynchronous shares are implemented and what the limitations are. We also show multiple practical examples of using asynchronous shares. The new system can be effectively used to consume services on the internet. It fits nicely into existing iTasks programs and requires few changes in existing programs.

Acknowledgements

I would like to thank Rinus and Mart for the weekly meetings, their encouragement to keep improving my work, and for their eye for detail. Further, I would also like to thank Bas for answering my questions about the iTasks system and his availability as Rubber Duck. I also thank John, for debugging weird behaviour of the graph serialization system. Lastly, I would like to thank Veerle for supporting me during the many stages of this thesis.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Structure . . . 2

2 iTasks **5**
2.1 Tasks . . . 5
2.2 Task Combinators . . . 6
2.3 Shares . . . 7
2.4 Task evaluation . . . 7

3 Shares in iTasks **9**
3.1 Using shares . . . 10
3.2 Interfacing with the world through shares . . . 11
3.3 Combining shares . . . 13
3.3.1 SDSLens . . . 13
3.3.2 SDSSelect . . . 15
3.3.3 SDSParallel . . . 16
3.3.4 SDSSequence . . . 17
3.4 Semantics . . . 18
3.5 Distributed Shares . . . 18
3.6 Implementation . . . 18
3.6.1 Internal share functions . . . 18
3.6.2 Registration . . . 19
3.7 Summary . . . 20

4 Shares as Classes **21**
4.1 Motivation . . . 21
4.2 Shares as type classes . . . 22
4.3 Usage . . . 24
4.4 Limitations . . . 25
4.5 Possible improvements . . . 25

5 Asynchronous shares **27**
5.1 Definitions . . . 27
5.2 Asynchronous Functional Programming . . . 28
5.3 Asynchronous iTasks . . . 29
5.4 Technical Challenges . . . 29
5.5 Requirements . . . 30

5.6 Share Rewriting 30
5.7 Share Context 31
5.8 Asynchronous Results 32
5.9 SDSService: Communicating with the Internet 33
5.10 Example: Retrieving Weather Information 35
5.11 Asynchronous Share trees 37
5.12 Share Operations 37
5.12.1 Read 37
5.12.2 Write 38
5.12.3 Modify 39
5.13 Example: MPD client 39
5.14 Error handling 42
5.15 Asynchronous SDSSource 42
5.16 Results 43

6 Distributed Shares **45**
6.1 Distributed iTasks 45
6.2 Distributed Share System 46
6.3 Requirements 46
6.3.1 SDSSource: Communication between iTasks systems 46
6.4 Example: Distributed Proxy 47
6.4.1 The Problem with Modifying 48
6.4.2 SDSLens 50
6.4.3 Remote registration 51
6.5 Task trees with multiple asynchronous shares 52
6.6 Error Handling 54
6.7 Example: Distributed Blockchain 54
6.8 Distributed iTasks 57

7 Conclusion and Discussion **59**
7.1 Limitations and Future Work 59
7.2 Related work 60

Chapter 1

Introduction

The goal of software frameworks is to make it easier to write software. The framework abstracts away from certain tedious details and allows the programmer to focus on the functionality of the software they are writing. A framework provides a foundation on which to build extensive applications, directing the programmer to solve problems in a certain way.

The iTasks framework [14, 15] is an implementation of Task Oriented Programming (TOP), a paradigm that focusses on abstract task definitions. These tasks can be combined and sequenced into bigger tasks, forming workflows and fully executable programs. TOP is a *declarative* paradigm: Tasks are specifications of *what* needs to be done instead of specifications of *how* it should be done. The iTasks framework is written in Clean [16], a pure lazy functional programming language.

The iTasks framework is an implementation of TOP focussed on building web apps. This entails, amongst other things, that programmers do not have to worry about the details of managing application state or communication between clients and servers. They do not have to worry about HTML, CSS, or JavaScript. They can build task specifications which are automatically converted into graphical user interfaces. The framework also provides ways to do distributed computing, by supporting tasks to be shared across different machines.

An example of such a specification is a simple program to enter a date and time into the system, which is then displayed to the user:

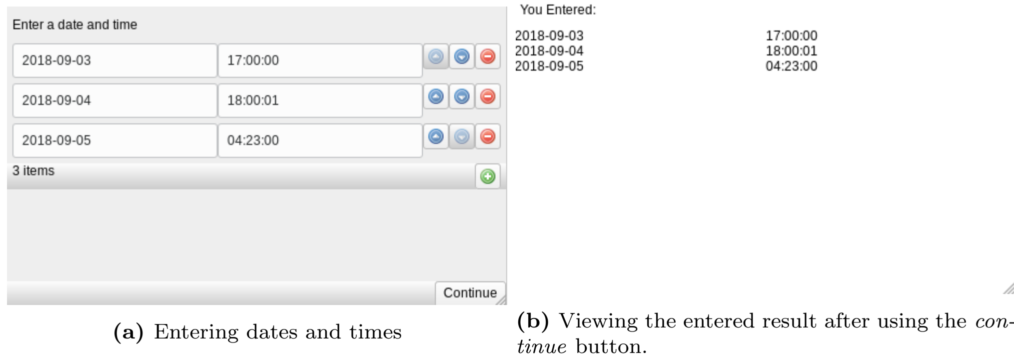
```
enterDateTime :: Task [(Date, Time)]
enterDateTime
    =   enterInformation "Enter a date and time" []
    >>= viewInformation "You Entered:" []
```

Here, the necessary GUI elements are generated automatically by the system. The transition between entering the information and viewing the result is handled by a continue button. Sequencing tasks like this forces the programmer to divide the work flow into separate steps which yield a result. It is also possible perform tasks in parallel.

The main building blocks of the iTasks system are editors, task combinators, and Shared Data Sources (shares).

A task can be a simple piece of work for a user (enter a number), a piece of work for the iTasks server (send a message to a certain network address), or a combination of multiple other tasks. A task is a state transformer function, continuously evaluated on the basis of events.

Task combinators are used to combine multiple tasks. The `>>=` combinator in `enterDateTime` is the bind operator: It feeds the result of one task into a function to create another task, creating



(a) Entering dates and times

(b) Viewing the entered result after using the *continue* button.

Figure 1.1

a bigger task with the type of its second argument. Combinators can also be used to perform tasks in parallel, or to attach actions to the user interface of a task.

Editors handle user interaction in tasks. In the example of `enterDateTime`, `enterInformation` returns a task. This task uses an editor under the hood to generate the input fields shown in Fig. 1.1. The editor is responsible for handling user interaction and maintaining state between the client and server. Editors are not the focus of this thesis, so we will not consider them further.

Shares are what allow tasks to store and retrieve data in iTasks. They can also be used as a means of communication between different tasks. For instance, task A might communicate that the user has entered the number 14 into a field, which may be used by task B to calculate 14! and display the result to another user. A disadvantage of this share system is that storing and retrieving data is done in a synchronous way.

It is possible to use iTasks in a distributed context. This entails that there are multiple machines which run an iTasks server. Tasks can be shared between these machines. One machine may be too busy and delegate the execution of a task to another machine.

1.1 Problem Statement

The iTasks framework is used more and more to communicate with external data sources. Tasks might retrieve data from the internet or connect to an external database. These data sources cannot be relied upon to reply in a timely manner, or to even reply at all. We want to provide an abstraction so that external data sources can be used more easily. A mechanism is needed that can handle communication with these sources while handling delays and errors in communication. In this document, we introduce a modification of the iTasks system that allows *asynchronous* evaluation of shares. This in turn allows asynchronous communication with all kinds of services on the internet as well as between iTasks servers. The modification consists of a new version of the share system.

1.2 Structure

In the next chapter, we will dive deeper into the iTasks system. We assume proficiency with the Clean programming language. We will explain what tasks are, how tasks can be composed using task combinators, and how tasks are evaluated. Then, in Chapter 3, we explain the share system:

The different kind of shares, the operations that can be performed on shares, and the semantics of these share operations. Chapter 4 explains a small but essential modification of the share system, namely a different way of expressing shares. Understanding this change is necessary to understand the main contribution of this document.

Chapter 5 introduces the extension which allows asynchronous evaluation of shares and how this can be used to connect to the internet. We first show the idea behind asynchronous shares and motivate the chosen solution. We also show multiple applications of the asynchronous share system. The following chapter describes how communication between iTasks servers is achieved. This chapter also includes multiple examples.

Chapter 7 summarizes the work, discusses the current limitations and possible improvements, and relates the work to other solutions to similar problems.

Chapter 2

iTasks

2.1 Tasks

A task is an state transformer function which is evaluated continuously when events are received for that task. Executing a task function always yields a new version of the task. This new version is used the next time when the task is evaluated. Evaluating a task produces a task value.

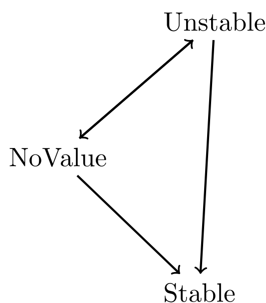


Figure 2.1: Different state transitions of a task value.

A value can either be *stable* or *unstable*. A stable task value is guaranteed to remain the same in the future. Note that it is possible for tasks to never reach a stable result. A task can either result in a value, or in no value. When it results in a value the stability of that value expresses whether the given value can change in the future. In the case of a stable value the task guarantees that future evaluations of the task will yield the same value. Figure 2.1 shows this as a state diagram.

A task is evaluated when an event is received by the system. For user tasks, such an event could be the user entering a value in a text field in their browser. For a task which communicates with an external service, such an event might be received when the service sends some data to the iTasks server.

An iTasks server can contain multiple tasks which are waiting for events to be evaluated. Multiple users can connect to a server in different browsers and perform different tasks. Tasks are evaluated sequentially. This entails that there is an event loop. This loop waits until one or more events are received and handles the events one at a time. The iTasks system is thus always evaluating at most one task at any given moment. The iTasks framework is designed with the idea that task evaluation is quick. To the end-user, it may seem that multiple tasks are

evaluated in parallel because task evaluation is near instantaneous. The client code for iTasks does run asynchronously.

The following functions are often used to create tasks:

```
enterInformation  :: String [EnterOption m]      -> Task m | iTask m
updateInformation :: String [UpdateOption m m] m -> Task m | iTask m
viewInformation   :: String [ViewOption m] m     -> Task m | iTask m
```

`enterInformation` generates a user interface according to the type `m`. This user interface lets the user fill in a form in their browser. `updateInformation` lets the user fill in a form in their browser using existing data. `viewInformation` is used only for viewing data.

The `String` parameter is used as the title to the generated user interface. Each of these functions is also supplied with a list of options. Options can change the way data is displayed to the user, or the way data is entered by the user.

Note that there are type restrictions. In `enterInformation`, the type variable `m` is restricted by the `iTask` class. This restriction ensures that the required functions are available to generate user interfaces. Note that the actual functions in the iTasks library are slightly different. They include more type restrictions. These restrictions are not essential to understanding the iTasks framework, so we leave them out.

2.2 Task Combinators

Tasks can be combined using combinators. These combinators express relations between tasks. For instance, one might want to execute two tasks sequentially, or in parallel. See the following listing for an overview of the most important combinators.

```
// Bind operator for Tasks.
(>>=) infixl 1 :: (Task a) (a -> Task b) -> Task b | iTask a & iTask b

// Decorates the left task with continuations.
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)] -> Task b | iTask a & iTask b

// Continues only when the left task has a stable value
(>>-) infixl 1 :: (Task a) (a -> Task b) -> Task b | iTask a & iTask b

// Continues when the left task has a stable or unstable value
(>>~) infixl 1 :: (Task a) (a -> Task b) -> Task b | iTask a & iTask b

// Do both tasks in parallel, yielding the result of the task that yields
// a value first.
(-||-) infixr 3 :: (Task a) (Task a) -> Task a | iTask a

// Do both tasks in parallel, yielding the result of the right task.
(||-)  infixr 3 :: (Task a) (Task b) -> Task b | iTask a & iTask b

// Do both tasks in parallel, yielding the result of the left task.
(-||)  infixl 3 :: (Task a) (Task b) -> Task a | iTask a & iTask b
```

```
// Do both tasks in parallel, yielding the result of both tasks in a tuple.
(-&&-) infixr 4 :: (Task a) (Task b) -> Task (a,b) | iTask a & iTask b
```

Note that the `>>*` combinator that allows attaching a list of *continuations* to a task. An example is the `OnAction` continuation.

```
task >>* [OnAction ActionContinue (always t)]
```

This continuation attaches a continue button to the task. Upon clicking the button, users will start working on the task `t`. `OnAction` allows inspecting the current value of `task` in order to determine whether there is a rest task. In this instance, we always move to task `t`.

The task functions mentioned earlier can be combined using these combinators. See the following task `collectUserInformation`:

```
collectUserInformation :: Task UserInformation
collectUserInformation
=   (       enterInformation "Enter a name" []
        -&&-
            enterInformation "Enter a phone number" []
    )
    >>= \info. viewInformation "Personal details" [] info
    >>| enterInformation "Enter a address" []
    @ UserInformation info
```

This task allows the user to enter a name and a phone number at the same time. It shows these details. When the user clicks the continue button, the user is asked to enter an address.

2.3 Shares

Some task combinators can be used to communicate values between different tasks. The bind combinator `>>=` communicates the stable value of the left task to the right task function. This is not always convenient. To communicate in that way makes it difficult to persistently store task values. It also complicates defining tasks because the programmer must always ensure that the required information is in scope. Sharing data between tasks which are otherwise unrelated is impossible. In iTasks, another mechanism is available to achieve communication between tasks. Shares are functions which allow tasks to access (possibly persistent) data. This data can either be available on the iTasks server, or available from other data sources like a database.

This thesis focusses on the share system. In the next chapter, we explain more thoroughly what shares are and how they can be used.

2.4 Task evaluation

Users of an iTasks program use their web browser to do their work. The program, depending on the URL, will add a new task instance to the environment for the user session. For example, when a user navigates to the `/pizza` endpoint, an instance of the “order a pizza” task is added. When a user navigates to the `/pay` endpoint, an instance of the “pay a bill” task is added.

These instances are added and started immediately. Starting an instance initializes the task instance state and generates a user interface when applicable. Not all tasks have a user interface.

The task instance is re-evaluated based on events. One possibility is that the user has edited a value in a text field, or clicked on a button. The event originates from an iTasks client, which is usually a web browser. Another possibility is that the iTasks system has received an event from another source. This source could be in the same iTasks server, or it could be another iTasks server.

Task evaluation is done with an environment `*IWorld`. Uniqueness typing (`*`) is used to ensure that the correct environment is passed around the system. Compile-time type errors are generated when this is not the case. The `*IWorld` holds the internal state for the iTasks system. The current time is kept here, amongst other things. `*IWorld` also contains the `*World`. This world is what allow tasks to perform side effects, like reading from a file or creating a TCP connection.

Task evaluation yields the following results:

- An optional `Value`, which can be stable or unstable as mentioned earlier.
- An optional change in the user interface for a user when the task has a user interface.
- A new version of the task state. This state will be stored in the system and used the next time the task function is evaluated.
- An updated `*IWorld`

The next chapter gives a thorough overview of the share system in its current state.

Chapter 3

Shares in iTasks

The purpose of this chapter is to give a thorough overview of the current share system in iTasks. This is necessary to understand the main contribution of this thesis. We first show how shares are used in the iTasks framework. We then explain the different types of shares. We conclude by giving some semantic properties of shares, as well as some implementation details.

Shares are an abstraction on data. A share is the single interface by which tasks can access this data. It is helpful to think of a share as a set of functions used to manipulate data. A share itself does not contain data, rather it is a set of functions to read from or write to the actual data source.

An example of share usage is the `withShared` function. The function creates a share for use in another task.

```
withShared :: b ((SDS () b b) -> Task a) -> Task a | iTask a & iTask b

sharedString = withShared "this string is shared"
    \share.
            viewSharedInformation "viewSharedInformation" [] share
        -&&-
            updateSharedInformation "updateSharedInformation" [] share
```

The `sharedString` function defines a task using `withShared`. The `share` variable is used as a share by `viewSharedInformation` and `updateSharedInformation`. See Fig. 3.1 for the user interface that is generated by this task definition. The share allows both parallel tasks to communicate. In this case, they communicate a simple `String`. Note that any type satisfying the `iTask` restriction could be used instead. The value of the share is initialised with *`"this string is shared"`*. The `updateSharedInformation` task is allowed to modify the current share value, while the `viewSharedInformation` task can only read the current value of the share.

In general, a share has type `SDS p r w`. The `r` and `w` type variables represent the read and write type of the share, respectively. The read type is the type of value retrieved when reading from the share. The write type is the type of value required when writing to the share. These types do not necessarily correspond. The `p` type variable denotes the type of the *parameter*. A share might need extra information in order to perform a read or write operation. We will touch upon the exact function of this parameter later.

In the case of the `sharedString` example, the type of the created share is `SDS () String String`.

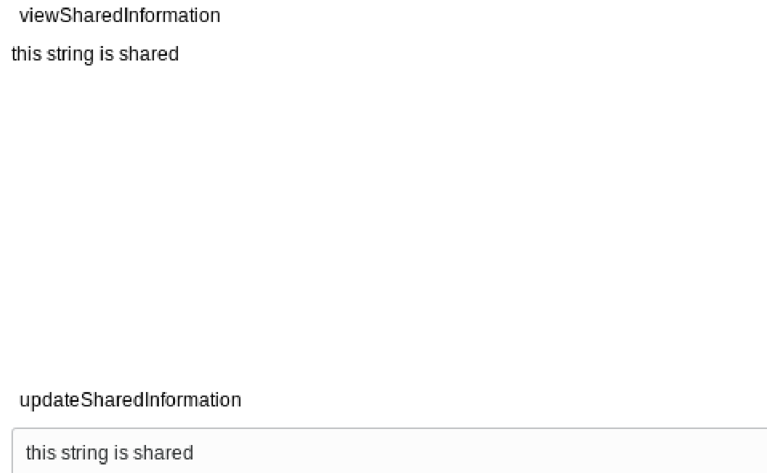


Figure 3.1: User interface generated for the `sharedString` task. The top half of the screen shows the current value of the share. It is updated automatically when the value of the share is changed. The bottom half of the screen allows the user to change the current value of the share.

Communication between tasks is not the only feature of the share system. Another important aspect is that it provides an abstraction layer on real data sources, like the file system or a server on the internet. We first explain how shares may be used in tasks. Then, we explain how this abstraction works. After that, we will show how shares can be composed.

3.1 Using shares

The iTasks system provides a number of function to interact with shares.

```
get   ::          (SDS () r w) -> Task r | iTask r
set   :: w        (SDS () r w) -> Task w | iTask w
upd   :: (r -> w) (SDS () r w) -> Task w | iTask w & iTask r
watch ::          (SDS () r w) -> Task r | iTask r
```

The `get` function retrieves the current `r` value of a share. The `set` function writes the given `w` value to the share. The `upd` function applies a function to the value of a share. This allows in-place updating of the share without having to `get` and `set` the value yourself. `watch` reads the value from a share. It also *registers* to changes in the share. Registration involves that the

iTasks system will re-evaluate a task whenever the value of the share changes. This is used to provide automatically updating user interfaces, amongst other uses. Writing and updating a share both trigger evaluation of all tasks that are currently watching the value of the share.

In addition to these operations, we also provide interaction primitives using shares that are analogous to `viewInformation` and `updateInformation`.

```
viewSharedInformation   :: String [ViewOption r] (SDS () r w)
    -> Task r | iTask r
updateSharedInformation :: String [UpdateOption r w] (SDS () r w)
    -> Task r | iTask r & iTask w
```

The `viewSharedInformation` function takes a share as an argument. It creates a task which registers to the value of the given share. The corresponding user interface is updated automatically when the value of the share is changed.

The `updateSharedInformation` also registers to the value of the given share. This function is used when the user must also be able to edit the value of the share. The user interface is automatically updated when the value of the share changes. Edits on the value made by the user are immediately applied to the value of the share.

We have shown how shares can be used in iTasks. We will now dive deeper into the share system, explaining how shares provide an abstraction on external data sources, the different types of shares, and how shares can be composed to form new shares.

3.2 Interfacing with the world through shares

Tasks must be able to interface with the world outside iTasks. An example of this is a task which reads data from a file. The share system provides the interface through which to interact with the outside world. The specific share type that allows us to do this is `SDSSource`.

```
:: SDS p r w = SDSSource (SDSSource p r w)
    | ...
:: SDSNotifyPred p :== TimeSpec p -> Bool
:: SDSSource p r w
  = { name  :: String
    , read  :: p *IWorld -> *(r, *IWorld)
    , write :: p w *IWorld -> *(SDSNotifyPred p, *IWorld)
    }
```

The `SDSSource` record contains two functions: `read` and `write`. The first function retrieves a value of type `r`. This can for instance be used to read the contents of a file, or to retrieve data from the internet by making an HTTP request. The second function writes a value of type `w`. This corresponds to replacing the contents of a file with new contents, or sending a PUT or POST message to an HTTP server.

The `write` function returns a notification predicate parametrised by the parameter of the share. This predicate is used to determine which tasks need to be notified when the source share is written to.

Note that both the `read` and `write` functions can result in errors.[1].

[1] For brevity and clarity, we leave out the corresponding `MaybeError` type in this document. The type of the `read` function is in reality `read :: p *IWorld -> (MaybeError TaskException r, *IWorld)`

```
:: FileName :== String

// Read the contents of a file as String
readFile :: FileName *IWorld -> (String, *IWorld)

// Write the value to a file
writeFile :: FileName String *IWorld -> *IWorld

valueShare :: SDS FileName a a | toString, fromString a
valueShare = SDSSource { name = "valueShare"
    , read = read
    , write = write}
where
    read filename iworld = appFst fromString (readFile filename iworld)

    write filename w iworld
    = (Ok (\otherFilename ts. filename == otherFilename),
        writeFile filename (toString w) iworld)
```

The given example share supports reading a `String` from a file and interpreting it as a Clean type using the `fromString` function. The inverse is true for the `write` function.

This share also shows how the *parameter* can be used. In this case, the parameter allows us to control the filename that is used to store and retrieve the value. The `valueShare` definition can now be used to store different objects of different types in different files, so that they do not interfere. Defining a new share for each object is unnecessary.

Another possibility to provide the filename to the share is to change the function signature to `valueShare :: FileName -> SDS () a a | toString, fromString a`. We will show later why using the parameter is the preferred way.

We can use this share in a task in the following way, assuming the required `toString` and `fromString` instances are available:

```
sdsFocus :: p (SDS p r w) -> SDS p` r w

integersShare = sdsFocus "integers.txt" valueShare

integersTask :: Task [Int]
integersTask = get integersShare
    >>= \integers. viewInformation "Current value" [] stringValue
```

For now, it suffices to say that `sdsFocus` is a function that supplies a parameter value to a share. We will look at the exact definition later in this chapter.

We have shown how the `SDSSource` share can be used to interact with the environment. One might wonder what the advantage is of using a share. After all, it is possible to achieve the same results in a task function because it also has access to the `*IWorld`.

The main advantage of shares is that they can be combined or composed. `SDSSource` is not the only type of share. Other shares allow constructing a new share from one or more other shares. For example, the `integersShare` from the previous example could be placed inside a share which filters out all odd numbers. Combining shares using combinators allows for filtering

data, zooming in on different parts of a data structure, or combining data sources into one. This allows using the same share definition multiple times, reducing code duplication. Another advantage is that registration to a share value is handled automatically, no matter the complexity of the share definition. The programmer does not have to implement updating tasks themselves, they can rely on the registration to provide automatic updating.

3.3 Combining shares

A share can be thought of as a tree. A leaf in the tree is a share which does not have any children shares. It can interact directly with the environment if necessary. `SDSSource` is an example of a leaf in a share tree. A node in the tree has one or more children. It uses these children to retrieve their values and combine them in some way. `SDSLens`, `SDSSelect`, `SDSParallel`, and `SDSSequence` are all nodes in a share tree. See Fig. 3.2 for a visual representation of a share tree.

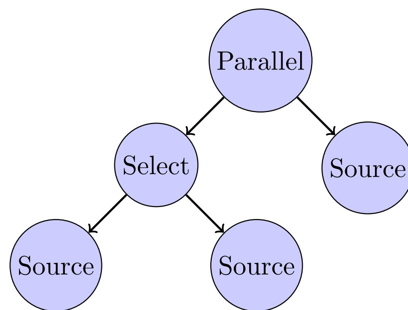


Figure 3.2: An example of a share tree. The top node is an `SDSParallel` share, which has an `SDSSelect` share and an `SDSSource` share as children. The left `SDSSelect` has two `SDSSource` shares as children.

3.3.1 SDSLens

The example share `integersShare` in the previous subsection can be used effectively to view and update a list of numbers.

Shares can contain all kinds of data structures. It is often the case that we are only interested in a part of the data structure provided by another share. In the case of a list of numbers, we might only be interested in the number at a specific index. In the case of a record, we might only be interested in a single field. The iTasks framework provides the `SDSLens` share which can transform or zoom in on data provided by another share.

A parametrized lens (introduced in their current form in iTasks by Domoszlai *et al.* [5]) can be used to transform the read or write type of a underlying share. For example, a lens can transform a share of type `SDS () [a] [a]` to `SDS Int a a`, focussing only on a specific element of the list. Another example of lens usage is to transform a share to be *read only*.

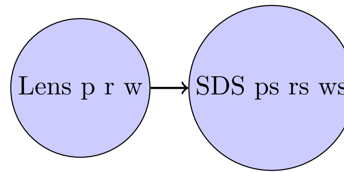


Figure 3.3: An example of a `SDSLens`. The type of the lens is `SDS p r w`. The lens contains functions to transform the read type of the underlying share (`rs`) into the read type of the lens itself (`r`). It contains an analogous function from `w` to `ws`.

Fig. 3.3 shows a visual representation of a lens.

```
:: SDS p r w =
    ...
    | E.ps rs ws: SDSLens (SDS ps rs ws) (SDSLens p r w ps rs ws) & iTask ps
    ...

:: SDSLens p r w ps rs ws =
    { param  :: p -> ps
    , read   :: p rs -> r
    , write  :: p rs w -> (Maybe ws)
    , notify :: pw rs w -> SDSNotifyPred pq
    }
```

Reading from a lens reads the value from the underlying share and applies the `read` function, transforming the `rs` value into the `r` value of the lens. Writing requires reading the value of the underlying share, applying the `write` function to it, and optionally writing the new value to the underlying share. Writing to the underlying share is optional. This is for example used to make shares *read-only*.

A lens also defines a `notify` function. This function is used when writing to the underlying share. Tasks registered to the value of the lens should only be notified when relevant data is changed. In the example of building a lens to focus on a single element of a list, tasks registered to the lens should only be notified whenever the value at the specific index is changed.

As an example, we define the following `singleItem` share. It is used to create a share of a single item in a list.

```
(!?) :: [a] Int -> Maybe a

singleItem :: SDS () [a] [a] -> SDS Int a a
singleItem = sdsLens "singleItemLens" (const ()) read write notify
where
    // Reading indexes into the list.
    read i items = mb2error "List index out of range" (items !? i)

    // Writing replaces the item at the index.
    write i items item = mb2error "List index out of range" (items !? i)

    // Notify only when the customer at the index is written to.
    notify i items item = \ts ii. i == ii
```

The `!?` operator retrieves an argument from a list at the specified index if the index is valid.

Note the type of the share: `SDS Int Customer Customer`. The parameter is the index of the customer in the list. However, all share operation functions (`get`, `set`, etc.) require a share of type `SDS () r w`. In other words, we must provide a value for the parameter. The previously mentioned `sdsFocus` function is another example of a `SDSLens`. It transforms the parameter type by providing a value for the existing parameter of a share.

```
sdsFocus :: p (SDS p r w) -> SDS p` r w | iTask p
```

This function fixes a specific value for the parameter and transforms the share into a share where the parameter can be any type. It can be used in the following way:

```
viewItem :: Int -> Task a | fromString, toString a
viewItem id = viewSharedInformation "viewItem" [] (sdsFocus id singleItem)
```

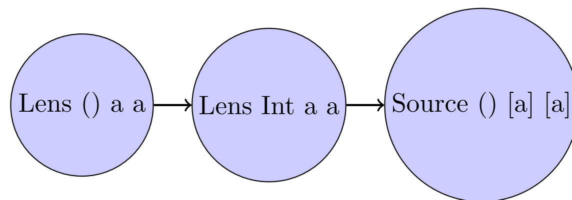


Figure 3.4: The visual structure of the lens used by the `viewItem` task.

See Fig. 3.4 for a visual representation of the structure of the lens used by `viewItem`.

Using this share, two tasks can work with different elements of data is a list without interfering with one another. Task_1 can safely update the list item at index 1 without causing task_2 to receive unnecessary update events. The share also shows why the parameter is useful: It allows us to combine different shares, while still being able to configure them.

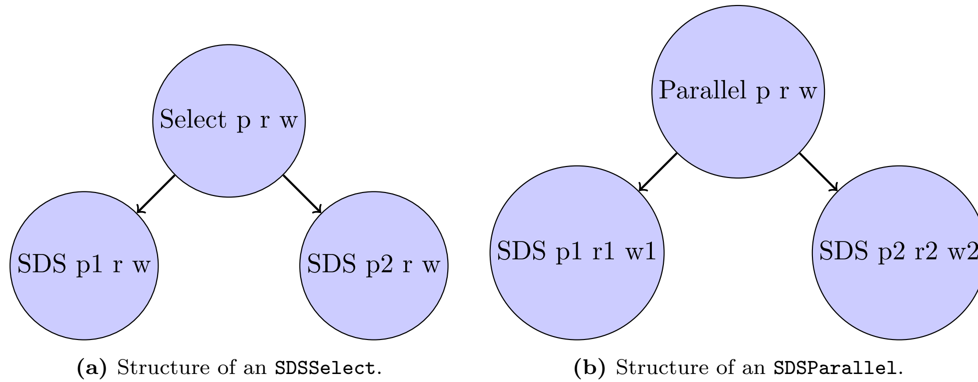
This example of retrieving a specific element in a list shows only a small portion of the capabilities of a lens. It can also be used to apply a function to the underlying data structure or to transform a share into a read-only share.

3.3.2 SDSSelect

A selecting share can choose between accessing one of the two underlying shares depending on the parameter. We can view an `SDSSelect` as a disjunction, choosing between two different other shares of the same read and write type. The definition of an `SDSSelect` share is as follows:

```
:: SDS p r w =
    ...
    | E. p1 p2: SDSSelect (SDS p1 r w) (SDS p2 r w)
        (SDSSelect p p1 p2 r w) & iTask p1 & iTask p2

:: SDSSelect p p1 p2 r w =
    { select  :: p -> Either p1 p2
    , notifyl :: p1 r w -> SDSNotifyPred p2
    , notifyr :: p2 r w -> SDSNotifyPred p1
    }
```



(a) Structure of an `SDSSelect`. **(b)** Structure of an `SDSParallel`.

Figure 3.5

The select share uses its current parameter to create either a parameter value for the left share, or a parameter value for the right share. Depending on its value it delegates all operations to either one of the underlying shares. The two notify functions are used to determine whether the select share should be notified when one of the underlying shares is written to.

3.3.3 SDSParallel

In order to combine data from multiple sources, a parallel share can be used. In contrast to the `SDSSelect`, this share combines the result of two other shares. Where the select share is a disjunction, this share combinator models a conjunction. See Fig. 3.5b for the structure of a parallel share.

```
:: SDS p r w =
    ...
    | E.p1 r1 w1 p2 r2 w2: SDSParallel (SDS p1 r1 w1) (SDS p2 r2 w2)
        (SDSParallel p1 r1 w1 p2 r2 w2 p r w) & iTask p1 & iTask p2

:: SDSParallel p1 r1 w1 p2 r2 w2 p r w =
    { param  :: p -> (p1, p2)
    , read   :: (r1, r2) -> r
    , writel :: p r1 w -> Maybe w1
    , writer :: p r2 w -> Maybe w2
    }
```

Suppose there are two shares defined as follows:

```
listItemShareA :: SDS Int [a] [a]
listItemShareB :: SDS Int [b] [b]
```

The combinator `>*<`

```
(>*<) infixl :: (SDS p rx wx) (SDS p ry wy) -> SDS p (rx,ry) (wx,wy) | iTask p
```

is used to create a share of type

```
combinedItems :: SDS Int ([a], [b]) ([a], [b])
combinedItems = listItemShareA >*< listItemShareB
```

The `sdsFocus` can still be used to select a specific index in the combined list of items. Notification is handled automatically: When either of the two underlying shares is written to, tasks registered to the value of the parallel share are notified.

3.3.4 SDSSequence

`SDSSequence` allows expressing dependencies between two shares. For instance, reading from one share may require reading from another. `SDSSequence` uses the value of the left share as parameter for reading and writing to the right share.

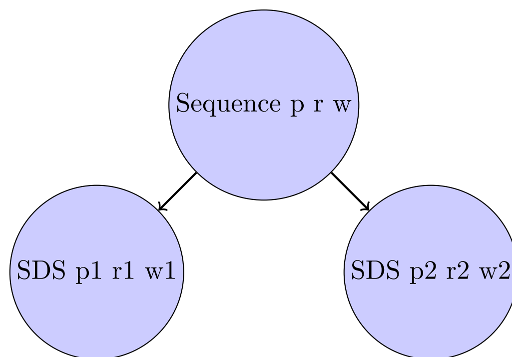


Figure 3.6: Structure of a `SDSSequence`.

```
:: SDS p r w =
    ...
    | E.p1 r1 w1 p2 r2 w2: SDSSequence (SDS p1 r1 w1) (SDS p2 r2 w2)
        (SDSSequence p1 r1 w1 p2 r2 w2 p r w) & iTask p1 & iTask p2

:: SDSSequence p1 r1 w1 p2 r2 w2 p r w =
    { paraml :: p -> p1
    , paramr :: p r1 -> p2
    , read   :: p r1 -> Either r ((r1, r2) -> r)
    , writel :: p r1 w -> Maybe w1
    , writer :: p r2 w -> Maybe w2
    }
```

The options contain a `paraml :: p -> p1` function, which transform the parameter for the sequence share to the parameter for the left share. This is then used to read a value from the left share. We then use the resulting value and the `read` function from `SDSSequence` to determine whether we still need to read from the right share. If we do, we use the `paramr` function to calculate the parameter for the right share and use the normal read function.

In practice, a sequence share is often used to store some kind of configuration. In the case of a share which communicates with a database, the left share of a sequence could contain

the database configuration parameters. Using a sequence share simplifies building shares upon the share database connection share, as we would otherwise need to focus the configuration parameters manually. This also allows for dynamic change of those parameters. For example, writing to this share could choose between using an internal in-memory database or an external database.

3.4 Semantics

The share system adheres to a number of properties that ease reasoning about it.

- Shares are completely opaque. A user of a share does not need to know how a share is defined or how it functions internally. This implies that a programmer can always safely use the built-in functions to work with shares without worrying about interactions between different types of shares or certain forbidden combinations.
- Reading and writing from a share is done atomically. It is guaranteed that while a task is being evaluated and it interacts with a share, no other task can change the value of the share during the operation. Consequently, there can be no race conditions when using shares.
- Shares are always updated atomically. While reading and writing from a share through `get` and `set` are separate steps, and could be interleaved by another task performing some share actions, updating is a single action.

These properties follow from the fact that tasks are evaluated in a blocking way. Only one task is being evaluated at any time, which in turn guarantees that only one share operation can be performed at once.

3.5 Distributed Shares

In a distributed setting, with multiple iTasks servers, these guarantees still hold. When performing a share operation on another instance, it is still guaranteed that the operation is done atomically on that machine. Care should be taken that the `upd` function is used as much as possible. On a single machine, first `get`-ing and then immediately `set`-ing is atomic because task evaluation is done synchronously. On multiple machines, this is no longer the case. Sending a `get` message and afterwords sending a `set` may be interleaved by operations of other iTasks instances. `upd` is still always performed atomically because it is a single action.

3.6 Implementation

It is useful to highlight a few implementation details which will be referred to in the coming two chapters.

3.6.1 Internal share functions

We have shown the four high-level operations on shares. These functions rely on more low-level functions to perform the required operations. The main contribution of this thesis will involve these low-level functions, so we show their definitions here.

```
read :: (SDS () r w) *IWorld
    -> *(MaybeError TaskException r, *IWorld)
write :: (SDS () r w) w *IWorld
    -> *(MaybeError TaskException (), *IWorld)
modify :: (SDS () r w) (r -> w) *IWorld
    -> *(MaybeError TaskException (), *IWorld)
register :: (SDS () r w) TaskId *IWorld
    -> *(MaybeError TaskException r, *IWorld)
```

Note that the `read` and `write` functions should not be confused with the `read` and `write` fields in specific `SDS` records.

The `read` function takes `SDS () r w` as argument. The read function uses the definition of the share to perform the required read operations. Reading yields a value of type `r`.

The `write` function takes `SDS () r w` as argument. It also takes a value of type `w`. It uses the definition of the given share to write the value to the share.

The `modify` function takes `SDS () r w` as argument. It also takes a function of type `r -> w`. It uses the `read` function to read the value of type `r` from the share. If reading was successful, it applies the modification function to the `r` value which yields a value of type `w`. The final step is to write this value to the share using the `write` function.

3.6.2 Registration

The `register` function has almost the same signature as `read`. The difference is that registration requires a `TaskId`. Registering to a share returns the read value of that share.

Registration to a share is achieved by storing a collection of notify requests in the environment. A notify connects a task to a share identity, denoting that the task is watching the share. A task can be dependent on multiple shares. For instance, when a task uses a value from a `SDSLens`, the task is not only dependent on the lens but also on the underlying share. After all, another task could use the underlying share without using the same lens. See Fig. 3.7 for an example.

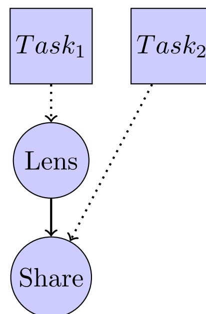


Figure 3.7: $Task_1$ uses the lens to retrieve a value. $Task_2$ uses the underlying share directly to retrieve a value. $Task_1$ should only be notified of changes relevant to it, taking into account the lens definition. $Task_2$ should be notified of all changes to share, even when the lens is written to.

When a task registers to the value of a share, the register implementation adds a notification request to the environment for all shares in the share tree. Whenever one of those shares is written to, the environment is checked for matching notification requests. These requests are

filtered to account for the specific share configuration, like different values for a parameter. The write operation returns a set of notify requests. For each request, a refresh event is queued for the respective task.

In order to efficiently keep track of registrations for each share, each share needs to be identifiable. This identify is a `String` and there is an internal function `sdsIdentity :: (SDS p r w) -> String` to do this.

3.7 Summary

We have shown that the share system provides a powerful abstraction for communication between tasks and working with external data sources. Shares can be combined using one of the provided share types. This simplifies development of an iTasks application and prevents code duplication. The next chapter shows a new modification to the share system that allows expressing the capabilities of a share, instead of the structure.

Chapter 4

Shares as Classes

This chapter explains an modification of the share system which allows us to express a share in terms of its *capabilities*. This is necessary to be more strict about which type of share will be accepted by the different share-manipulating functions in iTasks. The next chapter will use shares in the way explained by this chapter.

4.1 Motivation

Shares are implemented as an Algebraic DataType (ADT). The definition looks as follows, along with some useful type aliases:

```
:: SDS p r w = SDSSource ...
    | SDSLens ...
    ...
:: Shared r :== SDS () r r
:: ROShared r :== SDS () r ()
```

Modulo a few type synonyms which make it easier to work with shares (`Shared`, `ROShared`), constructors of `SDS p r w` are the only constructors on which share operations can be done. These constructors do not convey much information about the share. For example, expressing that a share is read-only is done by fixing the write type `w` to be `()` using the `toReadOnly` function. Writing can then still be done, but nothing is actually written to the share and no other tasks are notified of the write action.

We implemented an improvement that allows expressing *what* can be done with a share. This improvement has the following advantages:

- Separating the different actions that can be performed with a share allows us to create shares with partial capabilities such as a read only share. This allows for more fine-grained control on shares.

- Following from the previous point: The compiler can give errors when a specific action is not possible for a share. Writing to a read-only share was a no-op in the previous system. With this extension, writing to a read-only share results in a compile-time error.

- The extension allows to express certain share combinations in an explicit way. For example, only a `SDSSource` may be cached using an `SDSCache`[1]. This extension allows expressing such restrictions and gives compile-time errors when they are not satisfied.
- Extending the share system with new types of shares is easier. The logic of a new share implementation is not tied to iTasks, but rather to the application which defines and uses the new share type.

4.2 Shares as type classes

We distinguish several different behaviours which can be performed with shares:

Identify We can transform a share into its identity.

Read We can read a value from a share.

Write We can write a value to a share.

Modify We can apply a function to a share and retrieve the resulting value, also writing this value to the share.

Register We can read the value of a share and register to future changes.

Registration requires that we can identify a share in some way. We do this by generating a share identity. This identity uniquely identifies the share configuration. The identify includes the current parameter `p` for the share. Reading also requires the ability to identify the share.

Reading from a share and registering to a share are closely related. The only difference is that registering involves storing registration requests into the environment. Because of this, reading and registering are implemented by the same function. This function takes an argument which denotes whether to read or to register. Note that we still separate reading and registering to a share. This enables us to create a share which can be read, but which cannot be registered to.

Writing to a share requires identification as well. After writing to a share, we use the current parameter and the identity of the share to retrieve matching registration requests from the environment.

Modifying a share is implemented as first reading from the share, applying the modification function, and then writing to the share. Modifying a share thus requires the ability to read and write.

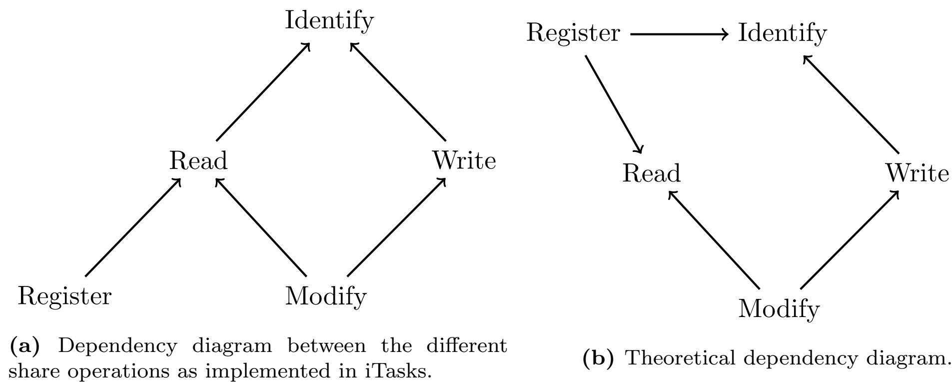
We can express this as a graph in Fig. 4.1a. In Fig. 4.1b we show what the graph would look like if reading and registering were implemented independently.

The class which expresses that a share can be identified provides a single function. It takes a share and produces a list of strings using an accumulator.

```
class Identifiable sds
where
    nameSDS :: (sds p r w) [String] -> [String]
```

We can only read from a share if we can also identify it. The `Readable` definition therefore contains a restriction that `sds` should also be a member of the `Identifiable` class for it to be a member of the `Readable` class. Note again that we leave out the `MaybeError` types.

[1]The previous chapter has not mentioned this share type, as the only purpose is to improve performance.



(a) Dependency diagram between the different share operations as implemented in iTasks.

(b) Theoretical dependency diagram.

Figure 4.1

```
class Readable sds | Identifiable sds
where
    readSDS :: (sds p r w) p (Maybe TaskId) String *IWorld
        -> *(r, *IWorld)
```

The `readSDS` function takes a share, a parameter, a possible task identifier and a share identity as parameter. The task identifier is used for registration purposes. The share identity is used to track the original share which has triggered the read operation.

In the same manner we define when a share is `Writable`. Remember that we need to be able to identify the share in order to find matching registration requests. This gives rise to the `Identifiable sds` restriction.

```
class Writeable sds | Identifiable sds
where
    writeSDS :: (sds p r w) p w *IWorld -> *((), *IWorld)
```

Registering to the value of a share only requires that it is `Readable`. We also get the restriction that it should be identifiable from the definition of `Readable`.

```
class Registrable sds | Readable sds
where
    readRegisterSDS :: (sds p r w) p TaskId *IWorld -> *(r, *IWorld)
```

Lastly, modifying a share requires that it is both `Readable` and `writeable`:

```
class Modifiable sds | Readable, Writeable sds
where
    modifySDS :: (r -> w) (sds p r w) p *IWorld -> *(w, *IWorld)
```

We also define a little helper class which allows us to shorten writing the restrictions on `sds` in the most common cases:

```
class RWShared sds | Modifiable, Registrable sds
```

With these classes we can define separate ADT's for each of the types of shares. We could define shares as record types directly. However, we will see that the next chapter makes use of these ADT's.

For example, `SDSSource` is now defined in the following way, where we leave out the actual implementation of the instances:

```
:: SDSSource p r w = SDSSource (SDSSourceOptions p r w)

:: SDSSourceOptions p r w =
    { name :: String
    , read :: p *IWorld -> (r, *IWorld)
    , write :: p w *IWorld -> (SDSNotifyPred p, *IWorld)
    }

instance Identifiable SDSSource
instance Readable SDSSource
instance Registrable SDSSource
instance Writeable SDSSource
instance Modifiable SDSSource
```

The `readSDS` instance does not replace the `read` function in the iTasks internals. The `read` function uses the `readSDS` implementation for the given share. `readSDS` is then used recursively on the children of the share.

4.3 Usage

Using type classes to represent shares has the effect that the API for iTasks functions like `updateSharedInformation` or `get` changes slightly. Where we would first require instances of the SDS ADT to be supplied to the functions, we now require that any type suffices as long as the appropriate classes are implemented for it.

```
// Before
updateSharedInformation :: String [UpdateOption r w] (SDS () r w)
    -> Task r | iTask r
// After
updateSharedInformation :: String [UpdateOption r w] (sds () r w)
    -> Task r | iTask r & RWShared sds

// Before
get :: (SDS () a w) -> Task a | iTask a
// After
get :: (sds () a w) -> Task a | iTask a & Readable sds
```

A consequence of using classes is that whenever a function returns a share, we need to specify the exact type of the returned share. The `mapRead` function, which transforms only the read value for a share using a lens, illustrates this:

```
// Before
mapRead :: (r -> r`) (RWShared p r w) -> RWShared p r` w

// After
mapRead :: (r -> r`) (sds p r w) -> SDSLens p r` w | RWShared sds
```

It is only necessary to explicitly specify the type when exporting the share definition from a definition module.

4.4 Limitations

There are several remarks to be made on this change:

- It is no longer possible to write type aliases in the same way for the most common share usages. First, we could write `Shared r` to abbreviate `RWShared () r r`. The fact that we need to refer to the type of the share from the list of context restrictions means we can no longer abbreviate shares like this. We can, however, still create some simple aliases for each of the share instances when we includes the `sds` type variable in the alias:

```
:: Shared sds a :== sds () a a

:: SimpleSDSLens a :== SDSLens () a a
```

  Note that we still need to write the correct restrictions on the `sds` type parameter.

- Due to the internals of the iTasks engine we have to be overly restrictive in some cases. Take for instance `viewSharedInformation`:

```
viewSharedInformation :: (sds () r w) String [UpdateOption r w]
    -> Task r | iTask r & RWShared sds
```

  The type `sds` is restricted to belong to the `RWShared` class, while intuitively we only need to be able to register to the value of the share. The cause of this is that `updateSharedInformation` and `viewSharedInformation` share large parts of their implementation under the hood, so we need to restrict `sds` so that it satisfies the restrictions on both functions.

4.5 Possible improvements

Take a look at the definition of SDSLens:

```
:: SDSLens p r w  = E. ps rs ws sds: SDSLens (sds ps rs ws)
    (SDSLensOptions p r w ps rs ws) & RWShared sds
```

The context restriction on `sds` is `RWShared`, a very strong condition. The `SDSLens` constructor requires that we pass in a share which is an instance of `RWShared`.

It should be possible to create a lens on a `Readable` share, which should yield a `Readable` lens. We have tried to explore solutions to this problem, but have not found one which is satisfactory.

One possible solution is to model the types of underlying shares explicitly. We add another type parameter to a share:

```
:: SDSLens sds p r w = E. ps rs ws: SDSLens (sds ps rs ws)
    (SDSLensOptions p r w ps rs ws) & ...
```

We could then change the instance of `Readable` for `SDSLens`.

```
instance Readable (SDSLens sds) | Readable sds
```

This allows us to express that a lens is readable only when the underlying share is readable as well. A similar approach can be used for all other share types and share actions.

The major downside of this approach is that we need to explicitly specify the full type of a lens, including the types of all underlying shares when exporting shares in a definition module. In iTasks, it is often the case that shares are deeply nested. Specifying the full type would be very cumbersome. This addition would not improve the user experience for programmers using the share system. We have therefore chosen not to implement this.

Chapter 5

Asynchronous shares

As seen in Chapter 3, shares allow tasks in the iTasks system to communicate and share data. Operations on shares are *safe*, in the sense that they are atomic. This property is due to the sequential nature of task evaluation by the system. When multiple tasks need to be evaluated, they are done so one after another. Every task evaluation is performed until there is a new version of the task. The atomic properties of shares are for free. An important effect of these properties is that race conditions do not exist.

However, this way of evaluating shares poses a problem. Suppose a programmer creates a new `SDSSource` share which initiates a connection to a database on the other side of the world. Chances are that sending queries and retrieving results are quite slow. This could be due to network performance, database performance, or other factors. The sequential task evaluation mentioned earlier causes evaluation of all other tasks to be postponed until one task retrieves a result from the database share. This is highly undesirable: All other tasks in the system cannot be re-evaluated until the time-consuming share has evaluated to a value, even the tasks which have no relation to the database share. This has the effect of a seemingly unresponsive iTasks application. Another result of this is that the share cannot keep a connection open between different evaluations, slowing down the evaluations.

A solution to this problem is some way to wait for a result without blocking other tasks in the system. To this end, we will define *asynchronous* shares and an asynchronous share system, allowing non-blocking actions. This system can be used to communicate with an external service, like a database or a HTTP service.

Before we can dive into the asynchronous share system, we first explain some prerequisites. We give some definitions that will be used in this chapter. We continue by identifying the requirements on an asynchronous share system. We look at other solutions to asynchronous programming in other languages and frameworks. Based on these findings, we present our solutions and motivate the choices made. After that, we explain all the ins and outs of an asynchronous share system. We show how the new system is used through multiple examples. We conclude by giving the results.

5.1 Definitions

Shares cannot be evaluated by themselves. They are always evaluated in the context of an action like reading, writing, or modifying. However, int this chapter it makes more sense to refer to these actions as share evaluation.

An asynchronous share is a share that may require waiting. Most often we are waiting for a network response. However, an asynchronous share does not necessarily need to involve communicating over a TCP connection. It could also involve reading a very large file, which is done in chunks of a certain size so that we do not block other tasks in the system. However, due to iTasks' dependency on TCP connections to generate events, we will only consider asynchronous shares which use the network. At the end of this chapter, we propose an unimplemented solution to this problem.

An asynchronous job is a TCP connection which has sent a message and is awaiting a response. This waiting is done in a non-blocking way. For example, on a POSIX system this is achieved using the `select()` system call.

5.2 Asynchronous Functional Programming

Programming language designers can approach asynchronous computing in many different ways. We highlight a couple of these ways in order to motivate our solution.

Functional Reactive Programming (FRP) is a programming paradigm that focusses around *streams* of events, and how these streams change. A stream is a sequence of values. A reactive application defines computations in terms of those streams. When the stream receives a new value, all computations which use the stream are re-evaluated. FRP is often used to build interactive applications which respond to change automatically, without manual updating of state or UI components.

Elm [2] is an implementation language for Event-driven FRP [20]. Elm can be compiled to JavaScript and is used to define responsive GUI applications in a web browser. In Elm, an event could be some interaction by the user (a button is clicked, an value in a text field is updated), or an event generated by a different source like a timer. Elm calls these events *signals*. Signals can be combined into new signals. GUIs can be defined using current signal values. These GUIs are automatically updated when the value for a signal is changed.

The way by which Elm achieves asynchronous computations is the *async* keyword. The evaluation of a signal can be made asynchronous in this way. This supports defining arbitrary asynchronous computations in conjunction with the ability to define multiple signals. This is the recommended way to perform HTTP requests.

Reynders *et al.* [17] propose *Multi-tier* Functional Reactive Programming, a way to use the same programming language to program servers and clients. The approach can be used effectively to create web applications in a single programming language, where asynchronous communication is handled automatically.

Haskell has multiple ways to do asynchronous programming, which are summarised in a tutorial by Simon Marlow [7]. It it possible to *fork* the evaluation of a expression. This spawns a new Haskell thread that is cheaper than an operating system thread. Communication between these threads is achieved through the `MVar` abstraction. This is a *mutable* data structure allowing multiple Haskell threads to safely communicate values. Haskell also provides *Software Transactional Memory*, a technique that enables a programmer to group operations atomically. This allows efficient communication and data sharing between threads, without having to build complicated locking mechanisms.

JoCaml by Fournet *et al.* [6] uses message channels and separate processes (threads) to perform asynchronous operations. There are two possibilities to communicate between processes: Though mutable data structures, or through a synchronous First-in First-out (FIFO) queue. The combination of these channels allows for declarative-style asynchronous programming. The authors give multiple examples how functions or expressions can be evaluated asynchronously.

See Syme *et al.* [18] for the asynchronous programming model used in F#. Dolan [4] *et al.* provide an extension to Multicore OCaml called *effect handlers*, allowing interaction between operating system threads and the OCaml runtime system.

5.3 Asynchronous iTasks

All solutions mentioned in the previous section rely on threads to work. An asynchronous operation can be performed in roughly two ways. The first way is to start a typed thread and wait for the result, after which the main thread can use the result to resume execution. The second way is to provide a thread with a callback to be executed when the asynchronous computation yields a result.

Clean does not have sufficient support for using threads in asynchronous work. However, we can start external processes and communicate with them. Starting an external process is expensive. Communication overhead with the process is also high. Therefore, these processes are not a good candidate for asynchronous share evaluation.

Another option is to extend Clean with a notion of threads. This would require major work in the language. One of the challenges would be the uniqueness typing. Additionally, the term-rewriting semantics of Clean might be changed with the addition of threads. On top of this work, iTasks would need to be rewritten to be able to deal with threads. A possible implementation would involve spawning a new thread for every top-level task. The challenge would then be to provide a way for tasks to communicate with one another safely. Clean is strictly a *pure* language, which means that are no mutable data structures. Mutable data structures would also need to be implemented to facilitate communication between different tasks.

We conclude that the best way to implement asynchronous shares is inside the existing infrastructure. We will not use threads, external processes, or mutable data structures. Because task evaluation is synchronous and done is a specific unique context, we cannot make use of callbacks or CPS-style continuations when evaluating a share. Rewriting the whole iTasks engine to be able to deal with threads or callbacks is too much work for the scope of this document.

5.4 Technical Challenges

There are certain challenges to consider when trying to design an asynchronous share system in the existing iTasks framework.

First, we need a solution that fits inside the current iTasks system. In terms of code, but also in terms of design and computation model. The solution needs to integrate well into the existing way tasks are evaluated. This yields the following considerations when designing an asynchronous share system:

- iTasks has been developed with the idea that task evaluation is quick. Consequently, share evaluation is also assumed to be quick. We must take care that asynchronous share evaluation is still quick.

- Tasks are evaluated even when no event has been received for that specific task. This is often the case when using a parallel combinator like `-&&-`. A refresh event for one of the tasks in the parallel combinator triggers re-evaluation of all tasks in the combinator. Shares must be able to be re-evaluated, even when no change has occurred which is relevant for that share.

The second challenge is that we need to keep intermediate state between different evaluations of the same asynchronous share. For instance, if a asynchronous share initiates a TCP connection, we need to remember this connection and be able to refer to it later, when data is received on the connection. If an asynchronous share reads from a file, we might want to remember the current cursor and the data that has been read so far.

The third challenge involves an important feature of the share system: The ability to combine shares using functions or combinators. We must be able to use an asynchronous share in all places where a share can be used currently. It must be possible to place a lens on an asynchronous share, to use it in a parallel combinator like `>*<`, or to select between two different asynchronous shares.

These challenges are *technical* in nature. Programmers using the iTasks framework should not have to consider the solutions to these technical challenges. We also present a set of requirements from the perspective of an iTasks programmer.

5.5 Requirements

Table Table 5.1 gives an overview of the requirements.

The most important requirement concerns the essence of asynchronous shares: Evaluation of asynchronous shares should not lead to waiting for a result in a blocking way.

Implementation of asynchronous shares should not affect existing programs. We want to make it as easy as possible to work with shares. We therefore require that an asynchronous implementation has no effect on the semantics of existing programs which do not use asynchronous shares. Consequently, programmers should not have to change their programs to keep the same behaviour.[1]

It is likely that certain actions take a varying amount of time. For example, an external service may be unreliable. Sometimes it replies within a second, sometimes it does not reply at all. A time-out mechanism should ensure that programmers can deal with unreliable data sources, yielding an error when an asynchronous action takes too long.

5.6 Share Rewriting

Aforementioned requirements and technical challenges lead us to design a system that can evaluate a share and remember the evaluation state when asynchronous actions are performed, without any changes to existing task-level share function like `get` or `watch`. We also require as few global state as possible, so that different evaluations of the same share can never interfere.

We introduce *share rewriting*. This design entails that evaluating a share may yield a result directly, or yields a new version of the share. If a new share is returned, it means that the share could not be evaluated in a single operation: There are asynchronous components in the share tree. This is done recursively. Evaluating the child of a node in the share tree may also result in a new version of the share, signalling that some asynchronous process is started to retrieve the value.

This solution fits nicely into the existing framework. Shares can be evaluated multiple times and the signatures of the `get` and `watch` functions does not have to change.

The high level idea of share rewriting is the following:

1. A task calls either the read, write, register, or modify functions with a share as argument.

[1] This does not consider the extension mentioned in Chapter Chapter 4, which requires changes to type signatures.

Category	Name	Description
Usability	**US**	I should not have to consider whether a share is asynchronous. The built-in functions which I use to build or use shares must not require me to consider whether a share is asynchronous.
Semantics	**SEM**	I should not have to adapt my existing iTasks programs to ensure they keep working.
Error handling		Evaluating a share can result in errors:
	ERR1	An error in evaluating an asynchronous share must be propagated to the task relating to the share.
	ERR2	An error in evaluating an asynchronous share must not affect the evaluation of other shares or tasks.
Time-out	**TIME**	I can configure time-outs and try-again options to ensure that I can make effective use of external services which may be slow or which may sporadically not respond.
Non-blocking	**BLOCK**	I can do any share operation without worrying about blocking other users, or making the system hang. I do not need to consider how other users might use shares.

Table 5.1: Requirements on an asynchronous share system from the perspective of an iTasks programmer.

2. Upon reaching an asynchronous share in the tree, initiate a connection to retrieve the value and rewrite the asynchronous node to keep track of the connection id. Return the new version of the asynchronous share.
3. When other shares retrieve an asynchronous share from one of their children, that share cannot be directly evaluated to a value and must also return an asynchronous result.
4. The task evaluating the share retrieves either a direct result, or a rewritten share. When it receives a rewritten share, it needs to store this share and await further refresh events related to asynchronous operations for this share.
5. When the connection receives a response, re-evaluate the task corresponding to the connection. The task re-evaluates the rewritten version of the share. The asynchronous node checks for a result in the environment using the stored connection id.
6. If a result is found, rewrite the node to incorporate the value into the tree and return the result. The result value can now be used higher up in the tree by parent shares to construct further results. We store the value in the tree so that the share may be evaluated multiple times without repeatedly performing asynchronous actions.

A task needs to store intermediate results from evaluating a share. This share may have initiated any number of connections. Each of these connections has to be able to send refresh events to the corresponding task. We need some way to associate a connection with a task. To this end, we define a *context* in which share evaluation is done.

5.7 Share Context

Evaluating a share can be done in different contexts. A possible context is a task: A programmer uses for instance `viewSharedInformation` to view the value of a share. Another possible context

is when a share is evaluated by the iTasks system itself. The internals of the iTasks engine use shares extensively. It relies on the fact that shares are evaluated synchronously. It has never been developed with asynchronous shares in mind. Changing the internal administration of the iTasks system would be a major rewrite outside the scope of this document.

We therefore choose to model this context explicitly, so that system shares cannot be evaluated asynchronously. Asynchronous evaluation may only be done in the context of a task. The iTasks engine can still rely on synchronous evaluation of its internal shares.

We introduce a type `TaskContext`:

```
:: TaskContext = SystemContext
    | TaskContext TaskId
```

We adapt the existing share functions so that they can take the context into account:

```
read :: (sds () r w) TaskContext *IWorld
    -> *(r, *IWorld) | Readable sds
readRegister :: (sds () r w) TaskId TaskContext *IWorld
    -> *(r, *IWorld) | Registrable sds
write :: (sds () r w) w TaskContext *IWorld
    -> *((), *IWorld) | Writeable sds
modify :: (r -> w) (sds () r w) TaskContext *IWorld
    -> *(w, *IWorld) | Modifiable sds
```

We also adapt the classes introduced in the previous chapter:

```
class Readable sds | Identifiable sds
where
    readSDS :: (sds p r w) p TaskContext (Maybe TaskId) SDSIdentity *IWorld
         -> *(r, *IWorld)

class Writeable sds | Identifiable sds
where
    writeSDS :: (sds p r w) p TaskContext w *IWorld
        -> *(Set TaskId, *IWorld)

class Registrable sds | Readable sds
where
    readRegisterSDS :: (sds p r w) p TaskContext TaskId *IWorld
        -> *(r, *IWorld)

class Modifiable sds | Readable, Writeable sds
where
    modifySDS :: (r -> w) (sds p r w) p TaskContext *IWorld
        -> *(w, *IWorld)
```

5.8 Asynchronous Results

Internal share functions (`read`, `write`, etc.) have to be adapted to deal with returning a new, rewritten version of a share when a share has asynchronous components. For each of the opera-

tions, we introduce a result type. This type contains either a result directly, or a new version of the share.

```
:: ReadResult p r w =
    E. sds: ReadResult r (sds p r w) & RWShared sds
    | E. sds: AsyncRead (sds p r w) & RWShared sds

:: WriteResult p r w =
    E. sds: WriteResult (Set TaskId) (sds p r w) & RWShared sds
    | E. sds: AsyncWrite (sds p r w) & RWShared sds

:: ModifyResult p r w =
    E.sds: ModifyResult (Set TaskId) r w (sds p r w) & RWShared sds
    | E. sds: AsyncModify (sds p r w) & RWShared sds
```

Each of the share operations has an associated result type. The `Result` constructors denote that a result is available directly, or that a previously asynchronous operation has yielded a result. The `Async` constructors denote that an asynchronous job has been started.

Note that the `Result` constructors also return a rewritten share. This is necessary because evaluating a subtree may have made changes within that tree, which we need in the future to evaluate the whole tree. For example, a parallel share with two asynchronous children may yield a `ReadResult` for the left child, while the right child still yields an `AsyncRead`. The left child share has yielded a value after performing an asynchronous operation. This means that that share has been rewritten to incorporate the value in the tree. By returning the rewritten share as well as the value, we ensure that the value can be used in the future by subsequent evaluations of the parallel share.

The `read` function and `Readable` class have to be adapted to return these asynchronous results. The changes for the other share operations are similar.

```
read :: (sds () r w) TaskContext *IWorld
    -> *(ReadResult () r w, *IWorld) | Readable sds

class Readable sds | Identifiable sds
where
    readSDS :: (sds p r w) p TaskContext (Maybe TaskId) SDSIdentity *IWorld
        -> *(ReadResult p r w, *IWorld)
```

5.9 SDSService: Communicating with the Internet

We have seen how the existing infrastructure has been adapted to deal with asynchronous shares. We now introduce one such asynchronous share: `SDSRemoteService`.

```
:: SDSRemoteService p r w
    = SDSRemoteService (Maybe ConnectionId) (WebServiceShareOptions p r w)

:: WebServiceShareOptions p r w
    = HTTPShareOptions (HTTPHandlers p r w)
    | TCPShareOptions (TCPHandlers p r w)
```

```
:: HTTPHandlers p r w =
    { host :: String
    , port :: Int
    , createRequest :: p -> HTTPRequest
    , fromResponse :: HTTPResponse p -> r
    , writeHandlers :: Maybe (
        p w -> HTTPRequest,
        p HTTPResponse -> MaybeErrorString (SDSNotifyPred p)
        )
    }

:: TCPHandlers p r w =
    { host :: String
    , port :: Int
    , createMessage :: p -> String
    , fromTextResponse :: String p Bool -> (Maybe r, Maybe String)
    , writeMessageHandlers :: Maybe (
        p w -> String,
        p String -> MaybeErrorString (Maybe (SDSNotifyPred p))
        )
    }
```

A service share is a share which communicates with the internet in order to perform operations. For reading, it uses the current parameter value to create a message. This message can either be a TCP message, or a HTTP request. The share definition also contains methods which transform the response (a `String` in the case of TCP, a `HTTPResponse` otherwise) into a usable Clean value of the correct `r` type.

Writing is performed in much the same way, with the exception that write support is optional. Writing in the case of TCP consists of sending a message and receiving some kind of acknowledgement that the message has been accepted. Writing in the case of HTTP consists of sending a POST or PUT request. Not all services on the internet support these kinds of operations, so we allow the programmer to express that they are not possible. When `writeHandlers` are not present, writing to a service share is a no-op.

Registration is not supported in the case of a HTTP service. It is supported when defining a TCP share. We cannot make use of the normal registration mechanism in iTasks. However, some TCP services support keeping a connection open until a change is made in the service. When the change is made, they use the open connection to notify the subscriber of the change. An example is a database: One could subscribe to changes in a specified table. The database would then notify when the table is modified, allowing the subscriber to retrieve the necessary data.

We take a closer look at the `fromTextResponse` function in `TCPHandlers`:

```
fromTextResponse :: String p Bool -> (Maybe r, Maybe String)
```

It takes as arguments the received message, the current parameter, and whether we need to register to the value. When the message can be parsed correctly, we return `Just r`. It could be the case that the message has been split up in multiple parts. When not all parts have been

received yet, we return `Nothing` to denote that the message is not yet complete. The second element of the returned tuple is used to optionally reply with a registration message.

This service share can be used anywhere in a share tree. This allows transforming data in the same way as you would normally do, like using lenses to zoom in on relevant parts. We now show an example how such a share can be used.

5.10 Example: Retrieving Weather Information

We now have all components required to construct a simple example of share usage. The general approach to defining a new asynchronous service share is the following:

1. We model the data using Clean types
2. We then specify whether the service uses TCP or HTTP and provide appropriate handlers, so that we may send messages and parse the response.
3. We create a share using the provided `remoteService` function

We use Open Weather Map[2] to illustrate this. This service allows us to retrieve the weather for a city using HTTP requests. It responds using JSON with weather information like temperature and humidity. It does not support updating weather requests, so writing is not supported.

We first model the response by Open Weather Map:

```
:: OpenWeatherResponse =
    { main :: OpenWeatherResponseMain
    , weather :: [OpenWeatherResponseWeather]
    }

:: OpenWeatherResponseWeather =
    { id :: Int
    , main :: String
    , description :: String
    , icon :: String
    }

:: OpenWeatherResponseMain =
    { temp :: Real
    , pressure :: Int
    , humidity :: Int
    , temp_min :: Real
    , temp_max :: Real
    }
```

`OpenWeatherResponse` maps directly onto the JSON returned by performing a GET request to Open Weather Map. We can now specify how the request should be sent and how the response should be parsed:

[2] https://openweathermap.org/

```
:: ApiKey :== String
:: Query :== String

createRequest :: (ApiKey, Query) -> HTTPRequest
createRequest (apiKey, city)
= {HTTPRequest|newHTTPRequest
    & server_name = "api.openweathermap.org"
    , req_path = "/data/2.5/weather"
    , req_query = "?q=" +++ city +++ "&APPID=" +++ apiKey
}

parseResponse :: HTTPResponse (ApiKey, Query)
    -> MaybeErrorString OpenWeatherResponse
parseResponse {rsp_code, rsp_reason, rsp_data} _
| rsp_code < 200 || rsp_code >= 300
    = Error (toString rsp_code +++ ": " +++ rsp_reason)
= case fromJSON (fromString rsp_data) of
    Nothing = Error "Could not transform JSON"
    (Just v) = Ok v
```

Finally, we define the share by creating a `WebServiceShareOptions` value with the appropriate HTTP handlers. We do not provide `writeHandlers`, because doing a POST request is not supported by Open Weather Map.

```
weatherOptions :: WebServiceShareOptions (ApiKey, Query)
    (MaybeErrorString OpenWeatherResponse) ()
weatherOptions = HTTPShareOptions handlers
where
    handlers = {HTTPHandlers| createRequest = createRequest
        , fromResponse = parseResponse
        , writeHandlers = Nothing
        , host = "api.openweathermap.org"
        , port = 80}

weather = remoteService weatherOptions
```

The resulting `weather` can be used to retrieve the weather for different cities. The parameter is used to select different cities.

An example of using the share:

```
serviceTask :: Task ()
serviceTask = enterInformation "Enter city" []
    >>= \city. get (sdsFocus ("APIKEY", city) weather)
    >>= \result. viewInformation result
    >>= \_. return ()
```

The above example illustrates that 1. We can define an asynchronous share that communicates over HTTP 2. The share can be used anywhere inside a share tree. This is illustrated by focussing

a parameter using `sdsFocus`. 3. The external API for `get` has not changed. 4. The programmer *using* the share when defining a task does not consider whether the share is asynchronous.

5.11 Asynchronous Share trees

See Fig. 5.1 for a share tree that contains asynchronous components. We use colours or patterns to denote the different states a asynchronous share can be in.

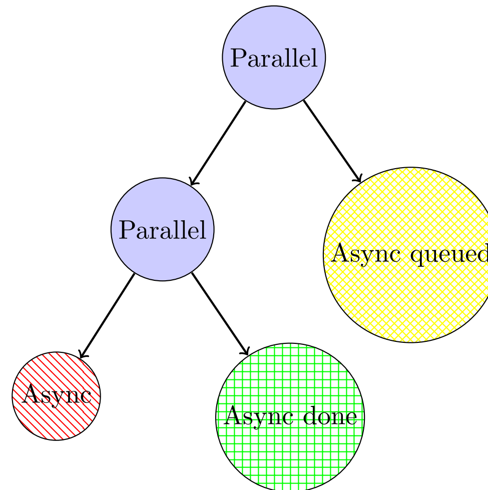


Figure 5.1: An example of a share tree with an asynchronous component. A green (horizontal grid) node denotes that the node is asynchronous and has already completed an operation. A red (diagonal line pattern) node denotes that it is asynchronous still in the initial state. A yellow (diagonal grid pattern) node denotes that the node is asynchronous and that some work has been queued to calculate a value.

5.12 Share Operations

5.12.1 Read

Reading involves walking down the tree until we reach either a `SDSSource` or a `SDSRemoteService`.

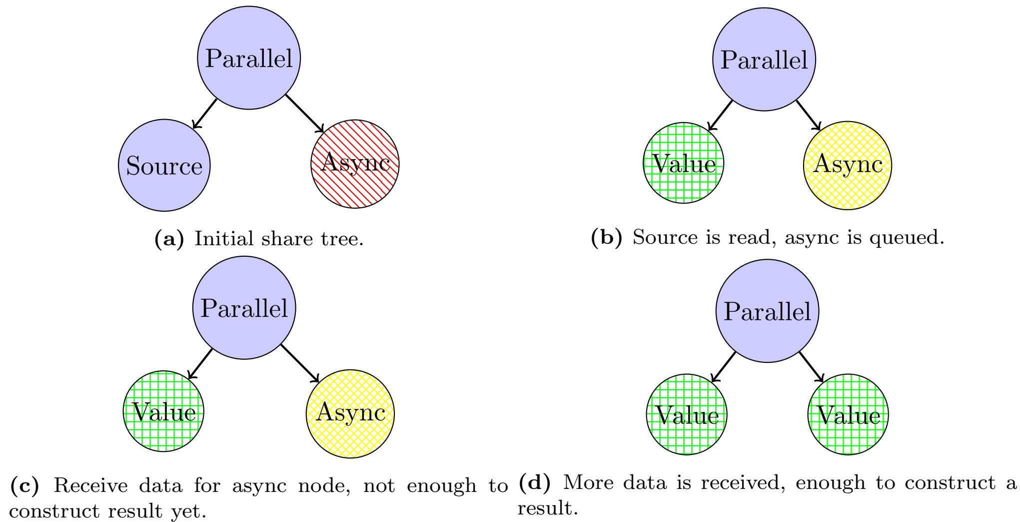
```
:: SDSSource p r w = SDSSource (SDSSourceOptions p r w)
   | E. sds: SDSValue Bool r (sds p r w) & RWShared sds
```

In the case of `SDSSource`, we simply execute the read function corresponding to the source definition to retrieve the value. This is done in a blocking way. After retrieving a value, the `SDSSource` rewrites itself to `SDSValue`, a new constructor of the data type. This denotes that the read has been performed and prevents it from being done over and over again when evaluating multiple asynchronous shares in the same tree.

As `SDSValue` is also used in writing, it contains a boolean to keep track of whether the underlying share has already been written to. Additionally, a reference to the original share is kept so that writing a value after reading is possible. This is used when modifying a share.

In the case of `SDSRemoteService`, we use the respective HTTP or TCP handlers to create a message and send that message over a TCP connection. We then rewrite the share so that we keep track of the connection identifier.

Other implementations of `read` use the values returned from reading their children in order to construct their own read result. For example, `SDSParallel` only yields a value when both of its children have been evaluated to a value. See Fig. 5.2 for a visual example. `SDSSelect` only yields a value when one of its children has been evaluated to a value, depending on the value of the parameter. SDSLens only yield a value when the underlying share has also yielded a value.



(a) Initial share tree.

(b) Source is read, async is queued.

(c) Receive data for async node, not enough to construct result yet.

(d) More data is received, enough to construct a result.

Figure 5.2: The evaluation of a parallel share. Each subfigure denotes a rewrite step of the associated task. Every rewrite step is done in response to an event. The parallel share keeps rewriting itself in order to include rewritten children. For example, in step *b*, it rewrites itself to `SDSParallel (SDSValue ...) (SDSRemoteService ...)`. In step c, the connection receives some data. This data cannot be parsed yet as a complete response, so the share does not yield a value yet. In the final step, it does not rewrite itself but rather directly returns a value.

When a task has executed the `read` function, it unpacks the result:

1. When a value is directly returned using the `ReadResult` constructor, the task can use that value to proceed with its own evaluation.

2. When a new share is returned using the `AsyncRead` constructor, the task must wait for a result from one or multiple asynchronous actions. Whenever it receives a refresh event, it re-evaluates the new share, repeating this until reading yields a value. The task does not know which shares are asynchronous or how many tasks are still queued. It can only re-evaluate the share to check whether it yields a value. We have adapted the existing share functions (`get`, `set`, `upd`, `watch`, `viewSharedInformation`, `updateShareInformation`) to show a loading animation when it is waiting for an asynchronous result.

5.12.2 Write

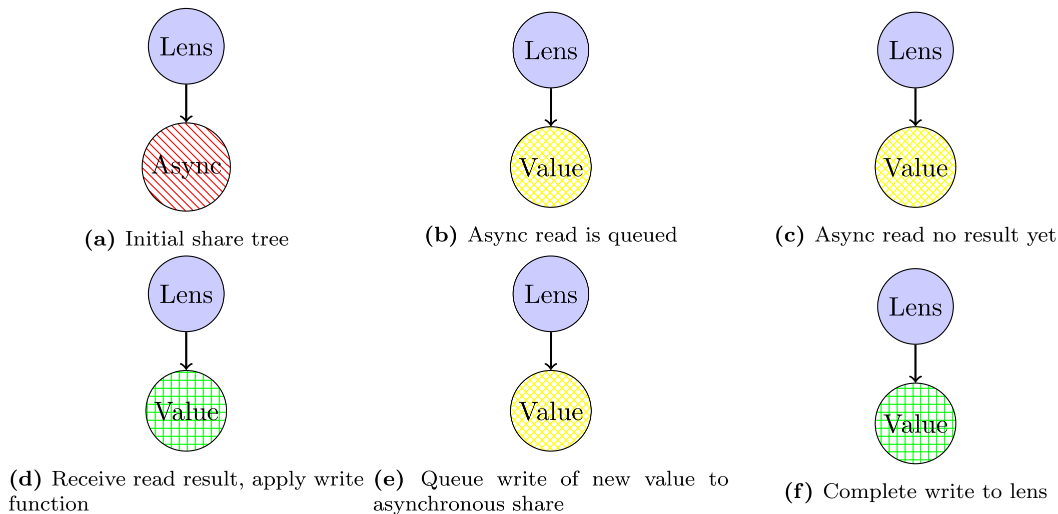
For the most part, writing does not need to be concerned with asynchronous shares. We already have a value `w` for the top most node in the share tree, which we divide using the respective

`write` functions for the different share types. When we encounter a asynchronous share, we send the write message to the host but do not need to wait for the result.

However, there is one share which does not fit into this approach. `SDSLens` needs to read the underlying share, apply the write function to the value of the underlying share, the value to be written to the lens, and the parameter, and write the result to the underlying share. See Fig. 5.3 for an example. This implies that we cannot, in general, make the assumption that writing can be done synchronously or in a "fire-and-forget" way. The modification for writing is much the same as for reading, with the `write` function either returning a result (`()`) or a new share which needs to be used to continue writing whenever a refresh event is received.

5.12.3 Modify

Modification of an `SDSService` is be done by first reading the value, and then writing the updated value from the modification function. By storing the value read in `SDSValue` in addition to the original `SDSRemoteService`, we ensure that we can directly write the value to an `SDSRemoteService` through the `SDSValue`. We rewrite to `SDSValue` to support multiple asynchronous shares in a single share tree, where these shares can retrieve results at different times.



(a) Initial share tree **(b)** Async read is queued **(c)** Async read no result yet

(d) Receive read result, apply write function **(e)** Queue write of new value to asynchronous share **(f)** Complete write to lens

Figure 5.3: Example of how a lens share is written to. It illustrates that reading the underlying share may be required in order to write to a lens.

5.13 Example: MPD client

The implementation of asynchronous shares makes it easier to create applications which talk to services through TCP. To show this, we choose Music Player Daemon [1], software to discover and play music on a wide range of operating systems. We chose MPD because the TCP interface is relatively simple. MPD communicates using a plain-text protocol. See Fig. 5.4 for an example session.

Asynchronous shares allow for communicating with MPD. First, we model the different commands and responses we can get from MPD. We leave out the full types as they are not necessary to understand the approach.

```
< OK MPD 0.2.0
>> currentsong
< file: Aitua/Elements/Aitua_-_01_-_Wings_-_I_Andante.mp3
< Last-Modified: 2018-10-12T08:36:00Z
< Artist: Aitua
< AlbumArtist: Aitua
< Title: Wings - I Andante
< Album: Elements
< Track: 1
< Date: 2018-10-12T04:33:56
< Genre: Classical
< Time: 89
< duration: 89.417
< Pos: 0
< Id: 1
< Ok
>> stop
< OK
```

Figure 5.4: A simple MPD session.

```
:: MPDCommand = Play | Pause | ...
:: MPDResponse = OkResponse | CurrentSongResponse (Maybe SongInformation)
    | ...
```

Then, we can create a share using the `remoteService` function:

```
mpdShare :: MPDCommand -> SDSLens () MPDResponse ()
mpdShare command = sdsFocus command lens
    where
        lens :: SDSRemoteService MPDCommand MPDResponse ()
        lens = remoteService (TCPShareOptions handlers)
```

We have to specify how incoming and outgoing messages are handled, as well the server to which to connect. We leave out the definition of `parseMPDResponse`, as parsing is outside the current scope. It is important to note that the `fromTextResponse` has type `:: String MPDCommand Bool -> (Maybe MPDResponse, Maybe String)`. The parser can use the sent command to parse the response more efficiently, and possibly send another message to the server.

```
handlers =
    { host = "localhost"
    , port = 6600
    , createMessage = maybe "" toString
    , fromTextResponse = \response command register. case command of
        Nothing = Ok (Just AckResponse, Nothing)
        Just command = case parseMPDResponse command response of
            Error e         = Error e
```

```
            Ok Nothing         = Ok (Nothing, Nothing)
            Ok (Just v)     = Ok (Just v,
                if register (registerString command) Nothing)
    , writeMessageHandlers = Just (toWriteMessage, fromWriteMessage)
    }

toWriteMessage p w = toString w
fromWriteMessage p message = case parseAck message of
    Error e = Error e
    Ok _ = Ok (Just (\_ _. False))

registerString CurrentSong  = Just "idle player"
registerString PlaylistInfo = Just "idle playlist"
registerString ListAllInfo  = Just "idle database"
registerString _ = Nothing
```

The `mpdShare` supports reading the current state of MPD, as well as writing commands to perform certain actions like advancing to the next song.

The only thing left to do is to define tasks which use this share:

```
task = (currentSongTask
        -&&- playlistTask <<@ ArrangeHorizontal)
    >^* actions
where
    currentSongTask = viewSharedInformation "Current Song" []
        (mpdStatusRequest CurrentSong)

    playlistTask = viewSharedInformation "Playlist" []
        (mpdStatusRequest PlaylistInfo)

    actions = [ OnAction (Action "Previous") (mpdAction Previous)
              , OnAction (Action "Play")  (mpdAction $ Play Nothing)
              , OnAction (Action "Pause") (mpdAction $ Pause On)
              , OnAction (Action "Next") (mpdAction Next)
              , OnAction (Action "Set volume") (always (
                    enterInformation "Volume" []
                >>= \vol. mpdShareCommand (SetVolume vol)))
              , OnAction (Action "Stop") (mpdAction Stop)]

    mpdAction w = always (mpdShareCommand w)
    mpdShareCommand a = set a (mpdShare Nothing)

    mpdStatusRequest r = mpdShare (Just r)
```

This generates a user interface in which the user can control MPD, as well as view the current song which is playing and the current playlist. The current song and playlist are updated automatically whenever they change, even when using another MPD client like `ncmpcpp`.

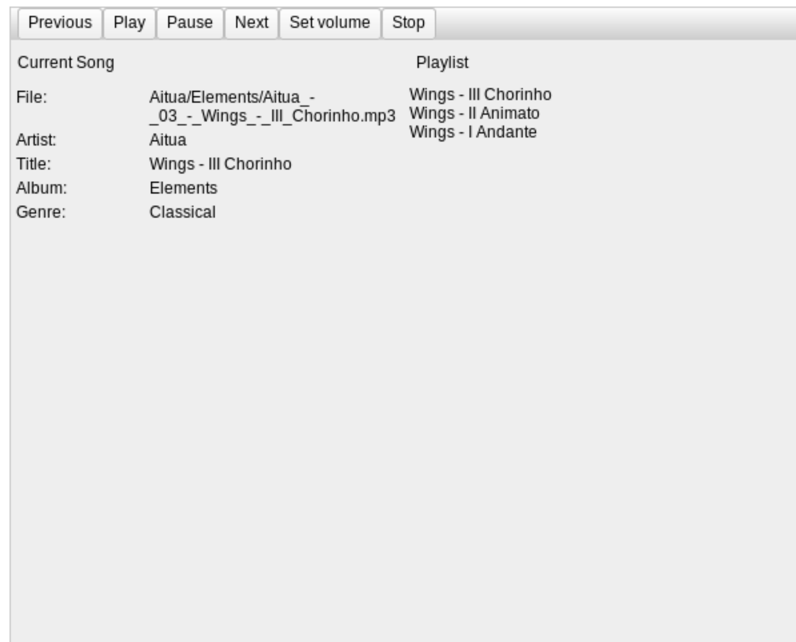


Figure 5.5: Music Player Deamon UI in iTasks.

5.14 Error handling

Evaluating a share is not guaranteed to succeed. Reading from a file could result in a error or an external service could return an error. Reading, writing, modifying, or registering could all fail.

When an error is encountered while evaluating a asynchronous share, the evaluation results in an `Error TaskException` value. This value is propagated to the task evaluating the share. The programmer can explicitly handle these exceptions, allowing him to deal with failures on a share level. When the programmer does not catch these exceptions, they are propagated to the end user.

5.15 Asynchronous SDSSource

Earlier in this chapter we mentioned that `SDSSource` is always evaluated in a blocking way. This complicates things when trying to read or search through a large file, for example. It is currently not possible to perform small pieces of a computation, while still allowing other tasks and shares to perform their work. However, we will try to show how this can be implemented. The actual implementation of this is still future work due to time constraints.

First, the definition of `SDSSource` must be changed:

```
:: SDSSource p r w = SDSSource (Maybe r) (SDSSourceOptions p r w)
    | E. sds: SDSValue Bool r (sds p r w) & RWShared sds

:: SDSSourceOptions p r w =
    { name          :: String
    , read          :: p *IWorld -> *(ReadResult r, *IWorld)
    , write         :: p w *IWorld -> *(WriteResult w, *IWorld)
    }
```

The main changes concern the `read` and `write`. We allow these functions to return partial results, so that these results can be used later to resume the computation. One possible implementation is to rewrite the `read` or `write` functions in `SDSSourceOptions`, keeping track of intermediate results by way of currying.

The remaining obstacle is to ensure that the task is refreshed, so that the computation can be resumed. This can be achieved by the `read` and `write` by queueing refresh events for the evaluating task after evaluation. Because the refresh event is added to the end of the queue, other tasks are evaluated before the share is evaluated again. Another option is registering the task to values of the clock. This ensures the share is evaluated at a constant frequency.

5.16 Results

We proposed a system allowing asynchronous evaluation of shares. The main idea is *rewriting* shares, keeping track of jobs which need to be completed before the share operation can yield a value. See Table Fig. 5.6 for a concise overview of the results for each requirement.

Programmers using shares do not need to know whether a share is asynchronous. All required functions to build shares or use them in task definitions have been adapted to be able to handle asynchronous shares.

Existing iTasks programs may need to be adapted in order to work with the interface changes of the `iTasks.Internal.SDS` module. Most programs do not directly depend on these definitions, however, and are able to be run without major modifications.

Asynchronous share evaluation can yield errors which are propagated to the associated task. These errors do not involve the evaluation of other tasks or other shares, as they are separated per task, per share evaluation.

All this is done in a *non-blocking* way, ensuring multiple users can work on the same system without interference.

There is no time-out mechanism in place to avoid endless waiting when a server does not respond. There is also no fall-back or try again mechanism in place. Therefore, tasks which require an asynchronous value in order to yield a result are not guaranteed to ever stabilize with a value or an error, depending on the availability of the data provider. The only reason for not implementing a time-out mechanism is a lack of time. We do not see any technical reasons why this could not be implemented.

We have not tested the performance of asynchronous shares. We suspect they will improve the user experience. This is mainly due to the fact that long calculations involving external services no longer involve waiting for a result in a blocking way. This should improve the responsiveness of the iTasks server and reduce the time a user needs to wait for a response from the server.

Requirement	Fullfilled
US	Yes
SEM	Yes
ERR1	Yes
ERR2	Yes
TIME	No
BLOCK	Yes

Figure 5.6: Requirements and whether the asynchronous share implementation full fills them

Chapter 6

Distributed Shares

Distributed computing is gaining more and more popularity. Open source frameworks like Kafka, MapReduce, and Apache Spark allow programmers to easily design distributed systems capable of processing large amounts of data. In iTasks, distributed computing involves sending tasks to other iTasks systems by using the distributed extension introduced by Oortgiese *et al.* [12]. This extension supports sharing tasks between devices. Devices can be other iTasks servers or Android phones. It also supports observing task values on other machines, assigning tasks to users on other machines, and transferring an existing task from one machine to another.

Sharing data between devices can be rather cumbersome. Retrieving data from another machine for example is done by creating a task that uses the `read` function to retrieve the value of a share on the other machine. The task is serialized and sent to the other machine, where it is executed. This results in unnecessary overhead: It should be possible to retrieve a share value from another iTasks server without using a task.

In this chapter, we show how we can communicate between different iTasks servers using the share system. We introduce a new share type that allows reading, writing, and modifying a share on another iTasks server. We show how this share can be used in practice through multiple examples, and how the semantics of the share system change when using the new share type.

6.1 Distributed iTasks

A distributed environment (as implemented by Oortgiese *et al.* [12]) is an environment with multiple iTasks servers. These can be physical servers, normal desktop PCs, and laptops and mobile phones. iTasks allows users to send tasks to a different device in a distributed environment, evaluate tasks which are sent to their devices, and retrieve or update information on another device.

The distributed extensions uses *graph serialization* for communication. The computation model of Clean is based on Term Graph Rewriting [13], which entails that all computations consist of applying rewrite rules to a program graph. This graph is stored directly in heap memory. It can thus be serialized to text, sent over to another server, de-serialized, and executed.

There are prerequisites for this to work. Either the executables need to be exactly the same, or the client needs to use debug symbols so that the server can use their list of debug symbols to de-serialize the graph. The graph is lazily evaluated and may contain all kinds of references to functions. Those functions may not have the exact same memory address across different machines or architectures. By storing the symbols and their relative addresses, we can always get the correct memory address for the current process to resolve function calls.

6.2 Distributed Share System

A share tree consists of leaves and nodes. Leaves can interact directly with the environment. `SDSSource` allows for interaction with the system on which the iTasks server is running. `SDSRemoteService` allows to interact with services on the internet in an asynchronous way.

The main idea of a distributed share system is that we can perform share evaluation on another machine. More precisely, when performing an arbitrary share operation, we allow subtrees in the share tree to be evaluated on other machines. In other words, we support moving arbitrary sub-share evaluation to different machines.

6.3 Requirements

See Table Table 6.1 for a concise overview of the requirements of a distributed share system.

The main requirement is that shares can be used to communicate between multiple iTasks servers. The standard `read`, `write`, `modify`, and `register` functions must all be supported.

The second requirement is that we can rely on the type system to ensure that communication is done correctly between iTasks servers.

The third requirement is especially important in a distributed environment. Dealing with many servers, it is important to identify which machine caused an error and what should be done to solve it. We therefore require that all errors are propagated to the machine which originally requested the operation. The error should also be descriptive, detailing where in the distributed environment the error occurred and what the exact error was.

Category	Name	Description
Communication	**COM**	As an iTasks programmer, I can use shares to communicate with other iTasks systems, reading from, writing to, and updating shares on that system.
Error handling	**ERR3**	An error on a another iTasks system must be propagated in such a way that it can be handled by the programmer, using the existing exception mechanism.
Type safety	**TYP**	As an iTasks programmer, I can rely on the type checker to ensure that distributed share use is type safe. I will not get unexpected runtime errors.

Table 6.1: Requirements on an distributed, asynchronous share system from the perspective of an iTasks programmer.

6.3.1 SDSSource: Communication between iTasks systems

We introduce a new share type that uses *graph serialization* to communicate the required operations to other iTasks servers.

```
:: SDSRemoteSource p r w
    = E. sds: SDSRemoteSource (sds p r w) (Maybe ConnectionId)
        SDSShareOptions & RWShared sds

:: SDSShareOptions =
    { domain :: String
```

```
    , port :: Int
    }
```

This new share type can be used as a node in the share tree. The underlying share is sent over to the host machine to be evaluated. `SDSShareOptions` can be used to configure the host and port to which the share message should be sent.

Evaluation of a `SDSRemoteSource` involves sending a message to another iTasks server. This message must contain enough information for the other server to perform the computation. We introduce a new type by which servers can communicate.

```
:: SDSRequest p r w = E. sds: SDSReadRequest (sds p r w) p & Readable sds
    | E. sds: SDSRegisterRequest (sds p r w) p SDSIdentity SDSIdentity
        TaskId Int & Registrable sds
    | E. sds: SDSWriteRequest (sds p r w) p w & Writeable sds
    | E. sds: SDSModifyRequest (sds p r w) p (r -> MaybeError TaskException w)
        & Modifiable sds
    | SDSRefreshRequest TaskId SDSIdentity
```

Each of the alternatives contains enough information to resume the computation on another machine. We send over the share that should be evaluated and the parameter with which the share should be evaluated. The context restrictions ensure that all required functions for performing the required operation on the remote machine are available when receiving the request. Depending on the operation we provide additional information. For example, modifying requires we send over the modification function.

Values of type `SDSRequest p r w` are serialized to `String` using aforementioned graph serialization. The share works in the same way as `SDSRemoteService`: A TCP connection is queued, and the share rewrites itself to incorporate the connection identifier. When a refresh event is received, evaluating the remote source share involves checking the environment for a result. If a result is found, task evaluation is resumed in the usual way.

We provide an example to illustrate simple distributed share usage. Then, we explain how share modification and share registration are done.

6.4 Example: Distributed Proxy

One of the advantages of distributed computing is that resources may be shares between devices. One of these resources is a connection to the internet. We show how an iTasks server can be used as a proxy, sharing its connection to the internet. This example also highlights that asynchronous shares can be nested.

Similar to the first example, we define a service share which supports retrieving the latest news from NewsAPI[1]. This API returns a JSON list of all the latest headlines for a specific country, in this case the Netherlands.

```
newsOptions :: WebServiceShareOptions () (Either String NewsApiResponse) ()
newsOptions = HTTPShareOptions handlers
where
handlers = { host = "newsapi.org"
    , port = 80
```

[1] http://newsapi.org/

```
    , createRequest = \_. {HTTPRequest|newHTTPRequest &
        req_protocol = HTTPProtoHTTPS
        , server_name = "newsapi.org"
        , req_path = "/v2/top-headlines"
        , req_query = "?country=nl&apiKey=" +++ newsapikey}
    , fromResponse = \resp _. fromResp resp
    , writeHandlers = Nothing
}

fromResp {rsp_code, rsp_reason, rsp_data}
| rsp_code < 200 || rsp_code >= 300
    = Ok (Left (toString rsp_code +++ ": " +++ rsp_reason))
= case fromJSON (fromString rsp_data) of
    Nothing = Ok (Left "Could not transform from JSON")
    Just d = Ok (Right d)
```

We define a share which focusses only on the list of news headlines, and does not allow writing to the service:

```
mapReadError :: (r -> MaybeError TaskException r`) (sds p r w)
    -> SDSLens p r` w | iTask p & RWShared sds

newsShare :: SDSLens () [NewsApiArticle] ()
newsShare = mapReadError read (remoteService newsOptions)
where
    read (Left error) = Error (exception error)
    read (Right {articles}) = Ok articles
```

We have defined a share which can retrieve headlines, however it does so on the current machine. We need to be able to evaluate this share on another machine in order to have access to the internet. We can do this using the newly introduced `SDSRemoteSource`, constructed by using the `remoteShare` function:

```
newsProxyShare :: SDSRemoteSource () [NewsApiArticle] ()
newsProxyShare = remoteShare newsShare {domain = "TEST", port = 9090}

Start world = doTasks (get newsProxyShare
    >>= viewInformation "The current news" []) world
```

These are all the definitions that are needed to use another iTasks server as proxy. The `newsProxyShare` is serialized, sent over to the proxy server, and evaluated on the proxy server. There, it retrieves the value of the share asynchronously and responds this value to the task server that made the original request.

All in all, it is easy to use another iTasks instance as proxy and requires no additional programming or configuration, aside from defining a simple remote share.

6.4.1 The Problem with Modifying

See the share tree in Fig. 6.2. We have a parallel share with two underlying remote shares. Reading from this share is easy enough: We just queue two asynchronous jobs and calculate the

The current news

Title:	Nagebouwd schip Willem Barentsz te water gelaten
Description:	Het duurt nog anderhalf jaar voordat het schip echt af is. De ultieme droom van de bouwers is om dezelfde tocht als zeevaarder Willem Barentsz af te leggen.
Author:	
Content:	Na acht jaar bouwen ging hij vandaag eindelijk te water: de replica van het schip van de Nederlandse zeevaarder Willem Barentsz. In Harlingen doopte prinses Margriet het schip De Witte Swaen. Het is het levenswerk van bouwmeester Gerald de Weerdt. Het projectâ¦ [+992 chars]
Url:	https://nos.nl/artikel/2256626-nagebouwd-schip-willem-barentsz-te-water-gelaten.html
Title:	Bertens verwijt zichzelf weinig na nipte nederlaag tegen Svitolina
Description:	Kiki Bertens kan zichzelf naar eigen zeggen weinig verwijten na de nederlaag van zaterdag tegen Elina Svitolina in de halve finales van de WTA Finals. De Zuid-Hollandse verloor nipt na drie zenuwslopende sets: 5-7, 7-6 (5) en 4-6.
Author:	NU.nl
Content:	Kiki Bertens kan zichzelf naar eigen zeggen weinig verwijten na de nederlaag van zaterdag tegen Elina Svitolina in de halve finales van de WTA Finals. De Zuid-Hollandse verloor nipt na drie zenuwslopende sets: 5-7, 7-6 (5) en 4-6. "In de eerste twee sets haalâ¦ [+1869 chars]
Url:	https://www.nu.nl/tennis/5537753/bertens-verwijt-zichzelf-weinig-nipte-nederlaag-svitolina.html
Title:	Drie gewonden bij ontploffing in woning in Alkmaar
Description:	
Author:	

Figure 6.1: Simplistic UI of the news proxy application

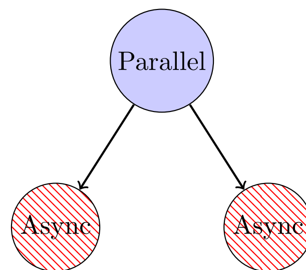


Figure 6.2: A share configuration which does not guarantee atomic updating of the underlying asynchronous shares

value for the whole share whenever we have a result for both asynchronous shares. The same holds for writing.

Modifying this share is different. We do not have a value which we can directly split across the two asynchronous shares, as we do when writing to the share. We have a function `f :: r -> w` which we need to apply to the whole share. In order to get a value of type `r` we need to read from both underlying shares and apply the `read :: (r1, r2) -> r` function which is available from the parallel definition to both results. We could wrap `f` with `read`, but that would still only yield a function of type `(r1, r2) -> w` requiring results from both shares.

In other words, we cannot split the modify function across the two underlying shares and this implies we lose the atomic updating of those shares. We resort to first reading the whole parallel share, applying the function, and writing the result.

This illustrates that it is not in general possible to modify a share atomically any more. When there are no asynchronous share in a share tree we can still guarantee that modifying is done atomically. When there are asynchronous shares in the tree, it depends on the configuration of

the tree and the implementation of the modify function for different shares. The only way to determine whether a share can be modified atomically is to look at the *structure* of the tree.

If the root node of a share tree is `SDSRemoteSource` and there are no other asynchronous components in the tree, we can guarantee that modification is done atomically. Evaluating the rest of the tree is done on another iTasks system in a blocking way just like a normal share without asynchronous components.

The following function illustrates the logic which should be used to determine whether a share can be modified atomically. It is *not* real Clean-code, as the different shares do not belong to the same algebraic data type. `isLocal` returns whether there are any asynchronous shares in a sub tree.

```
atomicallyModifiable (SDSSource _) = True
atomicallyModifiable (SDSSequence left right opts)
    = isLocal left && atomicallyModifiable right
atomicallyModifiable (SDSLens sub opts)
    = isLocal sub || isJust opts.reducer && atomicallyModifiable sub
atomicallyModifiable (SDSParallel left right opts)
    = isLocal left && isLocal right
atomicallyModifiable (SDSSelect left right opts)
    = atomicallyModifiable left && atomicallyModifable right
atomicallyModifiable (SDSRemoteSource source opts)
    = atomicallyModifiable source
```

In the case of `SDSParallel` it is not possible to update underlying asynchronous shares atomically. `SDSSequence` is only atomically modifiable when the left share is a local share which can be directly read as parameter for the right share.

In the case of `SDSSelect` we can always guarantee atomic updating, as the two child shares have the same type as the select. This allows us to just pass the same modification function to the children.

6.4.2 SDSLens

Whether we can atomically modify a lens is a bit more difficult. Recall the lens definition:

```
:: SDSLens p r w  = E. ps rs ws sds: SDSLens (sds ps rs ws)
    (SDSLensOptions p r w ps rs ws) & RWShared sds
:: SDSLensOptions p r w ps rs ws =
    { name        :: String
    , param       :: p -> ps
    , read        :: p r -> rs
    , write       :: p w rs -> ws
    , notify      :: p p w rs -> NotifyPred p
    }
```

We have a modify function `r -> w`. In order to atomically modify the underlying share we need a function typed `rs -> ws`. We can wrap the modify function with the read and param functions so that we get a function of type `rs -> w`. Now we only need a function from `w` to `ws`. Again, we use the existing write function which yields us a modification function `rs -> ws`.

Now it is possible to apply the wrapped modify function to the underlying share. There is still one problem: The function yields a value of type `ws`, whereas modifying the whole share should yield a value of type `w`. We need a function from `ws` to `w`.

We introduce a *reducer* which does this exact job:

```
:: SDSLensOptions p r w ps rs ws =
    { ...
    , reducer      :: Maybe (p ws -> w)
    }
```

Note that the type of reducer field is `Maybe (p ws -> w)`. It is not always possible to define a proper reducer. In that case, there is no other option than to first read the whole lens and then write the value of applying the modification function to the lens.

6.4.3 Remote registration

Registration to a share in a distributed context can involve registering to a share on another server. To properly record such a remote registration we need to record enough information to later send a message to refresh the relevant task. We need a hostname and a port to which we can send a message.

A notify request has the following type:

```
:: SDSNotifyRequest =
    { reqTaskId     :: TaskId
    , reqSDSId      :: SDSIdentity
    , cmpParam      :: Dynamic
    }
```

A request contains the task which should be notified, the share to which the task has registered, and the parameter which the task used to register. These requests are stored in the `IWorld`.

We expand this type to be able to record the necessary information:

```
:: RemoteNotifyOptions =
    { hostToNotify :: String
    , portToNotify :: Int
    , remoteSdsId  :: SDSIdentity
    }

:: SDSNotifyRequest =
    { ...
    , remoteOptions :: (Maybe RemoteNotifyOptions)
    }
```

Now we can expand `TaskContext` to contain the required information so that the proper notify request can be created:

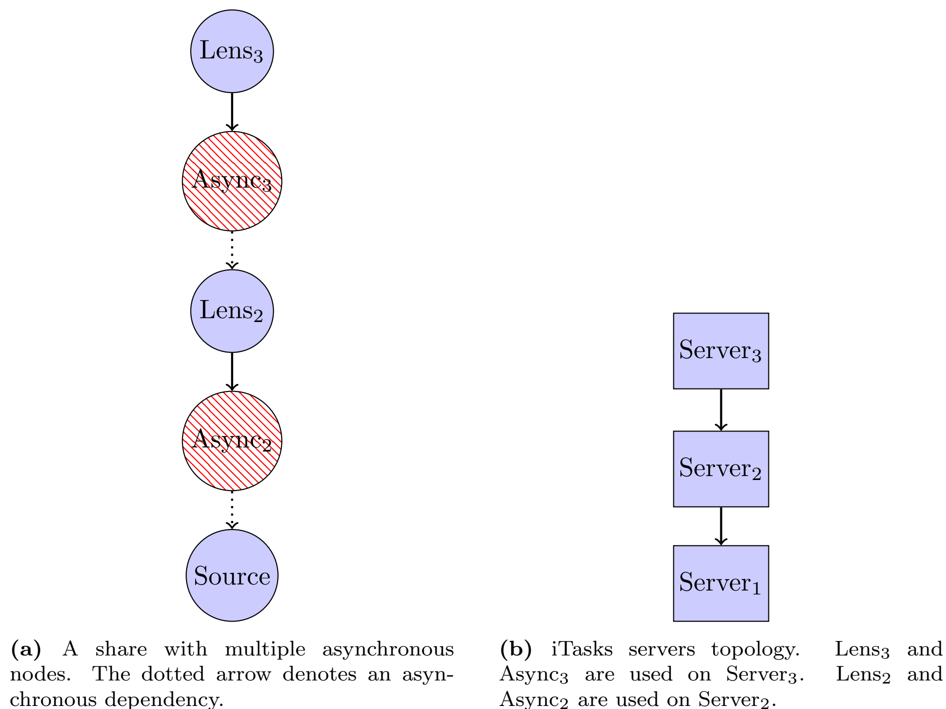
```
:: TaskContext =
...
| RemoteTaskContext
    TaskId  // The id of the original task reading the share
    TaskId // The id of the current task handling the request
    SDSIdentity // The id of the share on the remote server
    String // The hostname of the original task reading the share
    Int  // The port to which to send a refresh notification
```

Whenever a register request is received, we create a `TaskContext` with the relevant information and apply the normal `readRegister` function. The `readRegister` implementation takes care of creating the correct `SDSNotifyRequest` depending on the context type.

A refresh message is sent to all registered clients whenever a share is written to. We use the stored `SDSIdentity` to refresh all tasks registered to the share on the remote machine.

6.5 Task trees with multiple asynchronous shares

It is possible to nest multiple asynchronous shares. See Fig. 6.3 for an example.



(a) A share with multiple asynchronous nodes. The dotted arrow denotes an asynchronous dependency.

(b) iTasks servers topology. Lens_3 and Async_3 are used on Server_3. Lens_2 and Async_2 are used on Server_2.

Figure 6.3

Whenever someone uses the share on Server_3, a message needs to be sent to Server_2, which in turn requires a message to be sent to Server_1. The instance *in the middle* needs to keep state of

both the outgoing messages which still need to be completed and which incoming message they correspond to.

In order to enable this server to keep track of all this, we introduce a task which has the sole responsibility to receive messages from other iTasks instances, perform the requested operation by evaluating shares, and send a reply whenever the operation is finished. We will refer to this task as the *share service task*. This task is started automatically when the iTasks server is started using the `--distributed` option or enabled at compile time in the engine.

The service consists of two parts: It has connection handlers to deal with incoming connections from other clients. It also responds to normal events generated by the system. This second part allows the task to send messages and receive responses to those messages asynchronously.

Whenever a message is received by the share service task, it unpacks the message. The service tasks performs the required operation (`read`, `write`, etc.). If the operation yields a asynchronous result, any number of asynchronous jobs have been queued. These jobs are awaiting a response from another server. They are connected to the share service task by way of the task identifier. We store the rewritten share in relation to the incoming connection identifier. We use a share for this purpose.

When the share service task receives a refresh event, one of the asynchronous jobs for one of the pending share requests has received more data. The service task reads from its own share to retrieve the current state for all connections. It then evaluates all pending share requests. If a request yields a value, it is sent in response to the original client that sent the request. Otherwise we store the rewritten share and wait for additional data.

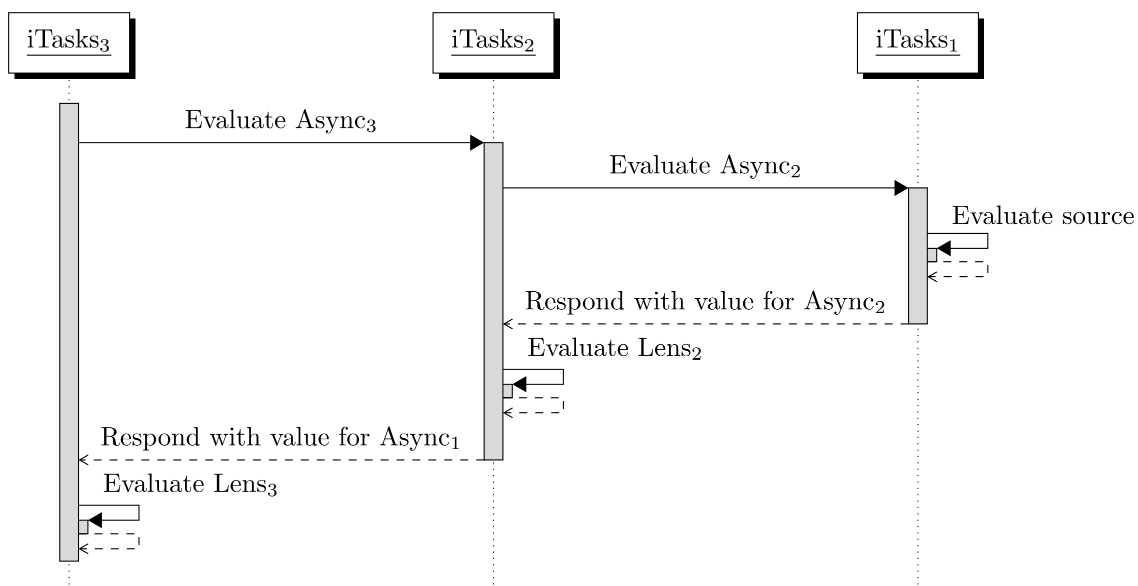


Figure 6.4: A sequence diagram of evaluating the share in Fig. 6.3a. Note that the time between sending a message and receiving a response is not spent waiting for a response. Rather, the server can do other work until a response is received. "Evaluate source", "Evaluate $Lens_2$", and "Evaluate $Lens_3$" are done in a blocking way on the respective servers.

6.6 Error Handling

In distributed systems, error handling is important. An error should give a good indication what went wrong, as well as where in the distributed system the error occurred.

When a share evaluation yields an error, the error is sent across multiple machines to the original client which requested the evaluation. In this way, it is possible to get a trace in the share tree which helps the programmer to determine where the error came from. The error is sent across devices is typed. See Fig. 6.5 for a example of a trace which is shown to a user when the exception is not handled.

Any errors in share evaluation are propagated to the iTasks server which initiated the request. Errors in communicating between iTasks servers are also handled. There is no time-out mechanism in place. Once a connection between iTasks servers has been established, it is assumed that the server that accepted the connection will eventually reply. When the accepting server closes the connection without replying (due to an internal error, for instance), the requesting server detects this. The task related to the operation receives an error.

Exception

SDSRemoteSourceQueued read get value
Remote to TEST:9998: Exception onData:SDSRemoteSource read queu
Remote to TEST:9999: Failed to connect to host TEST:9999

Figure 6.5: Screen shot of how an exception is shown. The message denotes that evaluating an asynchronous share to host TEST:9998 has failed, because it has in turn failed to evaluate an asynchronous share to host TEST:9999.

6.7 Example: Distributed Blockchain

Blockchain [11] is a great example of a distributed system working together to achieve a common goal. The aim is to store data in a *block chain*, which ensures that it cannot be tampered with once it is stored in the chain. An application of blockchain is storing financial transactions in a safe way. We refer to the paper by Nakamoto [11] for a thorough explanation of blockchain.

We implement a simple version of block chain in iTasks. In this version, there exists a single iTasks instance which contains the current version of the blockchain. In addition to the blockchain host, there is an arbitrary number of *workers*. These iTasks instances are tasked with mining new blocks. Each of the workers contains a list of data they need to integrate into the blockchain. Each workers attempts to mine a new block given the their data queue. When they find a block which fits, they attempt to update the blockchain on the server.

Multiple workers are mining at the same time. This could cause race conditions: If one worker has just appended a new block to the chain, other workers are still trying to mine using the old version of the block chain. There is no easy solution to this: We cannot continuously check whether the chain has been updated, because that would cause unacceptable slowdowns in mining. Our solution is to let the blockchain host accept or reject new blocks. If the host accepts the new block, it adds it to the chain. If the host does not accept the block, it will respond with an error to the worker who then restarts mining.

We start with defining what a block looks like:

```
:: Block =
    { nonce :: String
    , prevHash :: String
    , data :: String
    , hash :: String
    }
```

A block is dependent on the previous block by referencing the hash of the previous block. It has some arbitrary data, represented as a string. The nonce is what a miner can manipulate in order to let the hash adhere to the validation properties.

We then define a share which contains the blockchain, and another share which gives access to the most recent block:

```
blockchain = sharedStore "chain" [{nonce = "", prevHash = "",
    data="INITIALBLOCK", hash="INITIALBLOCK"}]

dataQueue :: SDSLens () [String] [String]
dataQueue = sharedStore "data" (map toString [1..200])

hdChain :: SDSLens () Block Block
hdChain = mapRead readLast
    ((sdsLens "verifyChain" id read write notify reduce) blockchain)
where
    readLast [b : bs] = b
    read = SDSRead \p rs. (Ok rs)
    write = SDSWrite \_ rs newBlock.
            if (b.Block.hash <> newBlock.prevHash
                || nonceNotUnique newBlock.nonce rs)
            (Error (exception "Block does not fit into chain!"))
            (Ok (Just ([newBlock : rs])))
    notify = SDSNotify \pw rs w ts p. True
    reduce = Nothing

    nonceNotUnique _ [] = False
    nonceNotUnique nonce [b : bs]
    = nonce == b.nonce || nonceNotUnique nonce bs
```

The first definition is straight-forward, the blockchain share contains a list of blocks. We also define a queue of initial data to be added, `dataQueue`. The second definition is more complicated. Reading from `hdChain` yields the most recent block in the chain. Writing a new block to the share will check whether the new block references the most recent block in the chain, and whether the nonce used in the block in unique in the current chain. If this is not the case, an exception is thrown.

We then define a task which allows for adding data to the head of the queue, as well as viewing the current queue and chain:

```
viewChain = viewSharedInformation "Current blockchain" [] blockchain
appendQueue = forever (enterInformation "Enter data to store in blockchain" []
    >>= \data. upd (\d. d ++ [data]) dataQueue)
```

We can then define the worker task to be executed on different machines:

```
remoteMine :: Task ()
remoteMine = enterInformation "Enter blockchain host" []
    >>= \remoteOptions. (appendQueue -&&- mineWithData remoteOptions) @! ()
where
    mineWithData :: SDSShareOptions -> Task ()
    mineWithData remoteOptions
    = whileUnchanged remoteShare hdChain remoteOptions
        (\lastBlock. get dataQueue
            >>= \data. case data of
                [] = return () @? const NoValue
                [todo:rest] = get currentTimestamp
                    >>= \ts.
                        return (mine sha1 (startsWith "00")
                            lastBlock.hash todo ts)
                    >>= \newBlock. catchAll
                        (updateChain remoteOptions newBlock rest)
                        (\e. return ())
        )
    where
        updateChain remoteOptions newBlock rest
            = set newBlock (remoteShare hdChain remoteOptions)
            >>| set rest dataQueue
            >>| viewTitle "Done mining" @! ()

Start w = doTasks tasks w
where
    tasks = [ publish "/" (const viewChain)
            , publish "/mine" (const remoteMine)]
```

This task first requires a user to enter the connection details for the blockchain host. Then, it will indefinitely try to read from the data queue and mine new blocks. Mining involves reading the most recent block in the chain, retrieve the current time (for random seed initialization), and adjusting the nonce until the hash is valid. Then, we try to write the new block to the chain. If it fails, we restart the whole mining process with reading the data queue.

We leave out the exact definition of the `mine` function.

We can effectively use this method to mine with multiple workers at the same time, which speeds up the mining process. Multiple miners will be notified whenever the blockchain is changed, after which they restart their mining process.

6.8 Distributed iTasks

We implement the distributed extension, originally built by Oortgiese [12], using shares instead of manually defining a distributed client task and a distributed server task. Asynchronous shares are a good candidate to implement sharing tasks with another device, to register to a distributed task value, or to claim a task from another device. They ensure that communication is done asynchronously and rely on the type system for correct communication.

We will not explain the whole distributed extension. We show how sharing a task and selecting a task to work on can be done.

Sending a task to another server can be done in the following way:

```
addTaskToDomain :: Domain (Task a) -> Task ()
addTaskToDomain domain task = appendDomainTask task domain
    >>= \taskId. viewInformation "Added task" [] taskId @! ())
```

Here, `appendDomainTask` is a predefined task which adds a task to a domain by writing to a share on the domain host. The current list of domain tasks can be retrieved by using existing task administration shares:

```
selectTask = enterChoiceWithShared "Select task" []
    (sdsFocus testUser taskInstancesForUser)
    >>= \{TaskInstance|instanceNo}. workOn instanceNo @! ()
```

The `selectTask` function allows a user to select one of the available tasks, after which they start working on it directly.

The task result in a domain can be registered to by using the following predefined function:

```
viewTaskResult :: TaskId Domain (Task a) -> Task () | iTask a
```

Here, the share system is used to register to the result share on another iTasks server. When the task value is updated on the other machine, the `viewTaskResult` task is also updated.

Chapter 7

Conclusion and Discussion

The iTasks framework is a multi-user, distributed framework that allows easy definition of tasks and the data required to execute those tasks. This data is represented with and abstracted over using shares, providing a single interface for tasks to interact with the outside world. Using shares is done synchronously, with the major disadvantage that IO operations are done in a blocking way.

We introduced an asynchronous share system that provides a way to evaluate a share without blocking an iTasks server. We used the system to implement asynchronous communication with the internet. Multiple tasks can use these shares without interfering with each other, and without having to wait for other tasks to complete their work. Most guarantees provided by the share system are maintained. The share API most often used by iTasks programmers has not changed. Existing programs do not need to be adapted to maintain the same behaviour.

We have also used the asynchronous share system to realize asynchronous, type-safe communication between iTasks servers in a distributed context. These distributed shares allow share evaluation on another system. Arbitrary share tree evaluation can be moved to another machine. This provides an easy API for iTasks programmers to distribute computations across multiple machines. Distributed shares can be arbitrarily nested, which means that arbitrary iTasks server topologies are supported.

The semantics of various share operations have changed with distributed shares, although only when they are used. In most cases, the semantics of reading and writing remains unchanged. The semantics of modifying have changed, as we can no longer guarantee atomic modification of a share in general when distributed shares are involved.

Finally, we have shown practical uses of asynchronous shares through multiple examples. It is easy to build tasks and shares upon asynchronous shares. This is essential when communicating with the internet or when performing other lengthy operations.

7.1 Limitations and Future Work

Asynchronous shares are a powerful tool. However, there are still some concerns and limitations to the system.

First, a drawback is that it is not possible to see from the type of a share whether it is asynchronous or not. Although not strictly necessary for the execution of a program, a programmer may find such information useful when reasoning about shares. A related issue is that we have to have very strong restrictions when building shares. Take for example the lens definition:

```
:: SDSLens p r w  = E. sds: SDSLens (sds ps rs ws) ... & RWShared sds & ...
```

If we build a lens atop a share which is only `Readable`, we would like for the lens to also be only `Readable`. Additionally, if we build a lens on an asynchronous share, we would like the lens to also be asynchronous.

Second, there is no time-out mechanism. A time-out mechanism is essential when dealing with external services which may never respond, either due to network problems or performance issues. We expect this not to be very difficult to implement. We think that implementing this should not be too difficult.

Third, it is not possible to keep state between different evaluations of a share. This is best explained by example: Suppose we have an asynchronous share which opens a TCP connection to some service. We register a task to the changes of that service, in such a way that the service sends us a message when the data is changed. When we receive such a message, the associated task is re-evaluated and the connection is closed. The task registers itself again, opening a new connection to the TCP service. It would be better to keep hold of the old connection to prevent continuously closing and opening connections. To this end, it would be great if we could keep some state between different evaluations of the same share. We do not expect this to be an easy problem to solve.

Fourth, due to the fact that there are no constraints on the structure of a share, it is possible to create two asynchronous shares on two different machines which continuously refer to each other, creating a non-terminating program.

We have not investigated the performance of distributed shares which make use of graph serialization. Some investigation should be done to determine what the overhead is of distributed shares, and whether the overhead is acceptable.

We have done some testing for error handling between iTasks servers. However, we have not done an extensive investigation into all possible kind of failures. It is future work to determine whether the existing error handling mechanism is sufficient.

Finally, it is currently not possible to define asynchronous shares which do not use TCP connections. Further investigation is required to implement the possible solution from the end of Chapter 5.

7.2 Related work

Mogk *et al.* propose REScala [8], a multi-tier reactive distributed programming language. The authors extends the language to be fault-tolerant, in the sense that different devices can rollback their internal state to the last known good state, and that errors are propagated between these devices. Programmers can decide how to handle errors in a signal, for instance when a signal is disconnected. The main difference with iTasks and the asynchronous share system is REScala is reactive and that it allows for explicit error recovery with automated rollback.

Another way to handle failures in a distributed context in proposed by Myter *et al.* [10]. They focus especially on partial failures and enable programmers to deal with them explicitly. They introduce *leased* signals, where the provider of a signal and a consumer of a signal make an agreement about the validity of the signal. When this agreement is violated, the signal provider is assumed to have failed by the signal consumer. Another contribution by the same authors is [9], where they propose a domain specific language which can manage reactive events as well as replication of data across multiple machines.

Weisenburger *et al.* [21] propose ScalaLoci, a reactive distributed programming language that uses *placement types* to denote that a signal or event should be retrieved from another device.

Other popular approaches to distributed programming are MapReduce [3] and Apache Spark [19]. These services allow for easy distribution of a computation across a cluster. The main appeal of these approaches is that they enable working with large datasets without having to deal with managing the distribution of the dataset across a cluster. The downside it that they are not designed to work with interactive applications like iTasks is.

Bibliography

[1] Music Player Daemon. https://www.musicpd.org/. Accessed: 2018-10-23.

[2] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[4] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K.C. Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *International Symposium on Trends in Functional Programming*, pages 98–117. Springer, 2017.

[5] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. Parametric lenses: change notification for bidirectional lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, page 9. ACM, 2014.

[6] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *International School on Advanced Functional Programming*, pages 129–158. Springer, 2002.

[7] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School*, pages 339–401. Springer, 2012.

[8] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[9] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. I now pronounce you reactive and consistent: handling distributed and replicated state in reactive programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*, pages 1–8. ACM, 2016.

[10] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Handling partial failures in distributed reactive programming. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 1–7. ACM, 2017.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[12] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. A distributed dynamic architecture for task oriented programming. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages*, page 7. ACM, 2017.

[13] Marinus Plasmeijer. Clean: a programming environment based on term graph rewriting. *Electronic Notes in Theoretical Computer Science*, 2:215–221, 1995.

[14] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices*, 42(9):141–152, 2007.

[15] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 195–206. ACM, 2012.

[16] Rinus Plasmeijer and Marco van Eekelen. Clean language report (version 2.2)(2002), 2002.

[17] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 55–68. ACM, 2014.

[18] Don Syme, Tomas Petricek, and Dmitry Lomov. The f# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*, pages 175–189. Springer, 2011.

[19] KMM Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[20] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven frp. In *International Symposium on Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

[21] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):129, 2018.