Bachelor Thesis

Radboud University Nijmegen

Stateful programming in Clean

Author:
Niek Janssen
s4297091

Supervisor:
Rinus Plasmeijer

Second reader:
Peter Achten

december 2018

Abstract

In pure functional programming, side-effects are modeled by passing state variables manually to all functions that interact with them. These variables can clutter a lot of code. We try to reduce cluttering using bind functions, but we find that we need some extra syntax to be able to implement this. Another solution is to introduce some extra syntax. We show that this approach improves code quality significantly. We present different syntax proposals for different cases: single lines of code which need a state, blocks of code where the state is needed, and single lines of code inside these blocks where the state should be omitted.

Contents

Chapter 1

Introduction

In pure functional programming, no side effects are allowed. To work around this problem, state variables (Haskell) or world variables (Clean) were introduced to pass an external state to callee functions. These state and world variables are essentially the same. The only difference is the way they are used. Clean tends to have multiple world variables for multiple distinct states, whereas Haskell merges all state variables together in a single state variable. Quickly after the introduction of state variables, monads and bind functions were introduced to hide these pesky variables from the developers.[1] Monads work well, except when multiple state or world variables are used or when these variables change their composition, both of which occur frequently in Clean.[2]

In this thesis, we will refer to state and world variables simply as state variables. In code examples, the term world variable is still used when appropriate.

As functional programming (and software development overall) progressed, more of these complex cases occurred. This resulted in developers reverting to the old ways of simply passing the state variables manually. The main issue with this approach is that it results in some very ugly code. Some programmers tried to solve these issues by creating monad combiners and shufflers, but this obfuscated code even more because the effects of those operators are not very intuitive. In *section 1.1*, we will look at the *iTask system*[3], which is currently struggling with these problems.

The main focus of this thesis will be improving the way state variables are handled. The end goal is improving the quality of code which heavily utilizes such variables. In doing so, we can improve the way stateful, imperative and functional systems and languages work together.

1.1 Context and motivation

The *iTask system*[3], developed at the Radboud University Nijmegen, is a system which allows the programmer to create an interface based on tasks. This system is based on the functional programming language *Clean*[4] and generates its interface semi-automatically using in-browser technologies, including Javascript. This poses a problem, as a strictly typed, functional language (Clean) needs to work with a non-typed, imperative language (Javascript). Even though a state-of-the art Clean-to-Javascript compiler already exists and the main infrastructure between the two languages has been built, it is still not easy to call Javascript code directly from Clean despite the fact that Clean code can be compiled to Javascript before execution. This means it is hard to use an already existing third party Javascript library in the iTask project.

At the moment, to use a Javascript library inside Clean, bindings for this library have to be written manually. This is not really avoidable, but to do this, programmers have to write a very ugly chunk of code. A good example of real-life code is `iTasks.Extensions.GIS.Leaflet`[5], a Clean wrapper for `leafletjs`[6] which can be found in the iTask distribution. This wrapper contains a lot of code looking like the example below:

```
1  addPopup :: String Pos Marker *World -> *World
2  addPopup content position marker world
3  # (options,world) = jsEmptyObject world
4  # world           = (options .# "maxWidth" .= 1000000) world
5  # world           = (options .# "closeOnClick" .= False) world
6  # (popup, world)  = (l .# "popup" .$ options) world
7  # (_, world)      = (popup .# "setLatLng" .$ position) world
8  # (_, world)      = (popup .# "setContent" .$ content) world
9  # (_, world)      = (mapObj .# "addLayer" .$ popup) world
10 # world           = (marker .# "myPopup" .= popup) world
11 =                   world
```

Code example 1: Ugly code from Leaflet.icl

Listed below are the main problems causing the ugliness of the code found in `Leaflet.icl`.

- **Javascript is an imperative/object oriented language, Clean is not.** Javascript does not adhere to principles used in functional programming. It makes excessive use of state, represented as a Javascript World in Clean. When writing library implementations, you will need many `let` statements, and you need to pass the state variable to every statement.

- **Javascript does not have a type system.** Clean does. In Javascript, any object can be anything.[7] These properties are often excessively abused by library

developers. Because of this, getting Javascript data into a state which is safe to use with Clean code can be very difficult. It may be possible to create a small library with functions to aid the developers in this task.

In the example code, multiple statements can be found traversing Javascript objects. These statements generally contain `.=`, `.#`, `.?` *and* `.$` *symbols, used as a workaround to the way Javascript object traversal works. This workaround severely reduces the readability of the code.*

- **Types may be incompatible and may need to be converted.** Some Javascript libraries may return complex types with getter methods, whereas in Clean you would just want a tuple. This conversion should be made easy to do. Just as above, a small library of basic functions should be created.

- **Transpiling Typescript declarations into Clean declarations.** This is not really a problem, but just an automation to help developers define types for Javascript libraries in Clean. Typescript is a language that transpiles directly into Javascript.[8] In Typescript, it is possible to strongly type all objects and classes used in a library, or just create a strongly typed interface for an existing Javascript object. When it is made possible to generate Clean declaration files (.dcl) from those Typescript sources, a lot of work will be taken out of the developers hands.

As stated before, this thesis will mainly focus on the second problem: the connection between imperative and functional languages.

Chapter 2

Research Plan

As mentioned above, the aim of this thesis is improving the connection between imperative and functional systems. We can do this by improving the way state variables are handled. This thesis tries to answer the following question:

- *'How can we increase the readability and ease of use for a state variable in a functional programming language?'*

To answer these questions, we will first have to think about requirements. What cases and edge cases will this solution have to support? After that, we will try to create two different solutions: one using bind functions and another by introducing new syntax.

- What requirements and use cases will occur when using a state variable?
- In what way can we use monadic bind-functions to pass the state variables around?
- In what way can we change the syntax to easily pass the state variables around?

After we manage to create these solutions, we will try to convert parts of `Leaflet.icl`, an existing library binding, to the new syntax to determine what solution works best.

Chapter 3

Related work

3.1 Handling state uniqueness with uniqueness typing

At the core of a pure functional programming language stands the paradigm that side-effects are not allowed. This enables a programmer to reason about a program via very simple substitution rules. Because a state variable actually has an underlying state, changing it would result in changes in all copies of such a state variable as a side effect. This means, to keep the paradigm alive, we need to disallow the copying of state variables. In Haskell, the programmer can use monads with state variables to ensure the uniqueness of state variables (see *section 3.2*). In Clean, uniqueness typing has been introduced as an alternative method of keeping state variables unique.

Uniqueness typing introduces a type modifier, to make any type optionally unique. When a type is unique, the compiler will output an error when a value of this type is copied. Uniqueness typing can be used on all types, including polymorphic types. The uniqueness itself can also be polymorphic.

Normal uniqueness typing can be obtained by typing an asterisk (`*`) just before a type or type variable. Polymorphic uniqueness can be denoted by putting a `.` or a `u:` before a type, where the `u` can be any lowercase letter as uniqueness variable.

In the example below, we will start with a simple state updating function declaration and generalize it a few times using this syntax.

```
// This function does not use uniqueness typing at all. The state
// variable may be copied or deleted, which is not what we want
f :: State Int -> State

// This function uses uniqueness typing to guard the state variables
g :: *State Int -> *State

// This function has a unique polymorphic parameter a, and a normal
// polymorphic parameter b. This means it can be called with two
// arguments of any type, but the type of the first argument has to be
// unique. It will return a variable with the same unique type as the
// first argument.
h :: *a b -> *a

// This function can also be called with two arguments of any type, and
// the type of the first argument can either be unique or not. The
// argument of b cannot be unique, because this method doesn't
// guarantee that the variable will not be copied.
i :: .a b -> .a

// This function is equivalent to function i, but uses a more explicit
// way of denoting uniqueness polymorphism.
j :: u:a b -> u:a
```

Code example 2: Uniqueness typing

Correctly utilizing uniqueness typing is very important when creating solutions for handling state variables. This is especially so when functions return more than only their state variables. For more information about uniqueness typing, see the Clean book**???**.

3.2 Handling state uniqueness with monads

In Haskell, a common way of handling IO, states and side-effects is using monads and monad transformers. Haskell has also introduced the so-called do-notation to simplify the usage of monads. Firstly, we dive a little bit into monads using Clean. After that, we take a look at the do-notation introduced in Haskell and the advantages and disadvantages of using monads and do-notation.

A monad is nothing more than a type wrapper, combined with two functions: bind (`>>=`) and `return`. The type wrapper can be simply defined as `m a`, where `m` is the monad and `a` is the type parameter of the monad. The `return` transforms *values* of a type `a` to a *monad* of type `m a` that represents the same value. .

The bind function is a little bit more complicated. It can be described as an infix operator for reversed function application. In a bit more detail: it is a function of type `(m a) (a -> m b) -> m b`. As the first (left side) parameter, it will take a *monadic* value of type `m a`. The second (right side) parameter is a function which takes a *normal* value of type `a`, and returns a *monadic* value of type `m b`. It will then execute the function in the second argument with the value of the first monadic argument. The bind function for a particular monad can do some operations to determine how the right hand side function should be called, and whether it even should be called. For simplicity, the implementation of the Maybe monad is given below. The main feature of the Maybe monad is that it skips all further function execution when a `Nothing` is encountered. It only continues execution when `Just x` values are returned.

```
1  // The maybe monad type itself
2  :: Maybe x = Just x | Nothing
3
4  // The return simply packs the value of v and returns it as a Just v
5  // monad.
6  return           :: a -> Maybe a
7  return v          = Just v
8
9  // The bind function takes a Maybe a monad and a function of type
10 // (a -> m b). It will execute the function only if the Maybe a monad
11 // is a Just.
12 (>>=) infixl 0  :: (Maybe a) (a -> Maybe b) -> Maybe b
13 (>>=) (Just a) f = f a
14 (>>=) Nothing    = Nothing
15
16 // With uniqueness typing, these functions would look like this:
17 return           :: Maybe .a (.a -> Maybe .b) -> .b
18 (>>=) (Just a) f = f a
19 (>>=) Nothing    = Nothing
20
21 // To actually make this a monad, we have to create a new instance of
22 // Monad:
23 instance Monad (St .s) where
24     return x   = return x  // Use the return defined above
25     (>>=) x f  = (>>=) x f // Use the bind defined above
```

Code example 3: Maybe monad definition

When using such bind function, we use a helpful property of functional languages: arguments do not have to be given all at once. This means, when a function is called with an incomplete argument list, the result will just be a new function that takes the remaining arguments and returns the result of the original function.

This feature allows us to use monads in combination with functions requiring more arguments than just the values inside the monad. The programmer can call a function `f` with all the arguments it needs, except the value or values inside the monad This partially called function `f` can then be used with the bind function, which will extract these variables from the monad and call `f`. An example is included below.

```
// Assume a function with this type exists:
f :: Int Int -> Maybe Int

g :: Int Int Int Int -> Maybe Int
g a b c d =
    return a >>=
    (f b) >>=
    (f c) >>=
    (f d)
```

Code example 4: Maybe monad usage

In the example above, function `f` is called with a single integer. The second integer argument it needs is provided by the bind function on the line above. Of one of the calls to `f` fails, the calls to `f` will not be executed and the return value of `g` will be `Nothing`.

The State monad

Monads can also be used to implicitly pass on a state. The monad used to do this is conveniently called the State monad. The state monad assumes a single return value and a single state.

```
// Definition of the state monad
:: St s a = St {runState :: s -> (a, s) }

// Instance of the state monad, where the return and bind are defined
instance Monad (St .s) where
    return x = \s -> (x, s)

    (>>=) fl fr = \s ->
        let (x, s) = fl s
        in  fr x s
```

Code example 5: State monad definition

The main goal of this monad, and specifically its bind function, is to chain together multiple functions that need and return a state. When two functions are chained together, they form a new function with the same properties. Because the return value of the bind

function is defined to be just an instance of a monad, the state monad itself has to be a function that still needs a state. This resulting state monad can eventually be run with the function `runState`.

As stated before, the general type of a bind function is `(m a) (a -> m b) -> b`. When we substitute `m` for `St s` and expand it, we get the type:

```
// Expanded type of the (>>=) with the St s monad
(s -> (a, s)) (a -> (s -> (b, s))) -> (s -> (b, s))

// Equivalent to
s -> (a,s)) (a s -> (b, s)) -> (s -> (b, s))
```

Code example 6: Expanded type of the state monad bind

From this type, we can already see how we have to combine the functions on either side of the bind function. When we receive a final state, we first have to pass it to the left function. The result then has to be passed to the right function, and its result has to be returned. This is what happens in the implementation of `>>=` we had in code example 5.

To illustrate this process with an example, let's assume a state with type `*World`. This world includes an information screen, which displays the name, age, and living place of a person. A function `setAttribute` is given with type `setAttribute :: String String -> St *World ()`, where the first two strings represent the name and new value of an attribute shown on screen. The world is just passing through, representing the screen and the information on it. We define a second function that extracts a person with an integer id from the world: `getPerson :: Int -> St *World Person`. We now create a function `setPerson` which updates the name, age and place on the screen.

We could do this by simply using uniqueness typing and a few `let` expressions. This means we have to pass the world variable manually:

```
// setPerson takes a personId of type Int, and will update the World
// accordingly. Because it has to return the state in the State monad,
// it will return St *World (). () means there will be no value.
setPerson :: Int -> St *World ()
setPerson personId world
# (person, world) = getPerson personId world
# ((), world) = setAttribute "name" person.name world
# ((), world) = setAttribute "age" person.age world
=               setAttribute "place" person.place world
```

Code example 7: Update info screen with let expressions

Alternatively, we could omit the let expressions and use the function calls directly as arguments. This would result in some some pyramid-like code, but it allows us to omit most of the explicit 'world' variables:

```
setPerson :: Int -> St *World ()
setPerson personId world =
# (person, world) = getPerson personId world
=   setAttribute "place" person.place (
        setAttribute "age" person.age (
            setAttribute "name" person.name world
                ))
```

Code example 8: Update info screen with pyramid code

By using the bind function, we get the best of both worlds. We get the linear coding we had when using the let-expressions, but we can omit most of the world variables like we could when using the pyramid code:

```
setPerson :: Int -> St *World ()
setPerson personId =
    getPerson personId >>= \person ->
    setAttribute "name" person.name >>= \_ ->
    setAttribute "age" person.age >>= \_ ->
    setAttribute "place" person.place
```

Code example 9: Update info screen with the state monad

Note that this monad is only suited when all functions require the same, single state. When multiple states get involved, we have to resort to a workaround, such as packing all states into a single tuple.

Do-notation

Haskell has also got some syntactic sugar for using bind functions. This is called the do-notation. We can take the code from example 9 and write it in do notation:

```
setPerson :: Int -> St World ()
setPerson personId = do {
    person <- getPerson personId;
              setAttribute "name" person.name;
              setAttribute "age" person.age;
              setAttribute "place" person.place;
}

// This is equivalent to:
setPerson :: Int -> St World ()
setPerson person =
    getPerson personId >>= \person ->
    setAttribute "name" person.name >>= \_ ->
    setAttribute "age" person.age >>= \_ ->
    setAttribute "place" person.place
```

Code example 10: Do notation

The do-notation is a bit similar to the let-notation (#-notation) used in Clean. Statements will be executed on the right, and the results will be assigned to variables to the left. There are a few differences, though. Unlike the let-notation in Clean, the do-notation uses bind and lambda functions to execute the statements and when the state monad is used, also implicitly pass on the state. As a result, in do notation it makes sense to have statements without assigning a result to a variable to the left. The bind function makes sure the resulting state is passed on to the next function. In Clean, omitting the variables to the left would make no sense at all, because such expression would have no effect at all and would not even be executed.

Because the do notation is so close to the let-notation in Clean, we can execute code with multiple state or return variables in the same way. Like the Clean let-notation, however, it involves a lot of rewriting and explicitly passing state variables. In the example below, the identity monad is used.

```
// The identity monad:
:: Id a = Id a

instance monad (St .s) where
    return x = Id x

    (>>=) (Id x) f = f x

// Example using Clean let-notation
# (state1, state2) = function1
# (state1, state2) = function2 state1 state2
                   = function3 state1 state2

// The example above can be written in do-notation like this:
do
    (state1, state2) <- function1
    (state1, state2) <- function2 state1 state2
                        function3 state1 state2

// Which will in turn translate to this:
function1               >>= (\(state1, state2) ->
function2 state1 state2 >>= (\(state1, state2) ->
function3 state1 state2
```

Code example 11: Do notation with multiple states

More background material

References [9], [1], [10] and [11] provide some background reading material for both strategies, explaining some solutions created for Haskell and Lisp.

Chapter 4

State Passing

In this chapter we tackle the main problem as introduced in *chapter 2*: the verbosity of code when using a state. We try to answer the three subquestions in order. Firstly, we setup the requirements for the solution. After that, we try to create a solution using monads and bind functions. We first discuss how we would handle shuffling or selecting state variables. After that, we will introduce a new language feature we need when creating a bind function, and finally we create the actual bind function.

As an alternative solution, we try to adapt the Clean parser to let it automatically handle state variables. We propose and discuss some different changes in the Clean syntax to simplify state passing to different methods.

4.1 Requirements, assumptions and possible scenarios

These are the requirements we will need to fulfill when creating our solutions.

- **The state parameter should be passed to functions requiring it.** When a function requires a state, the state should be given as a parameter, and a newly returned state should be used afterwards. When a function does not require a state, it can return a new state, but it does not have to. This allows for the use of 'normal' Clean functions, instead of only functions requiring a state.

  A state variable will always be the last argument given. It is conventional to do so, because it is convenient for further use of this function. As a return value, it may be the only variable returned, or it will be on the last position in a tuple. We don't have to implement solutions for one-tuples, as they don't exist in Clean.

- **A function may require more than one state variable.** In line with the above, they will be the last arguments, and the last items in a tuple. The state variables will be returned in the same order as they were asked / given.

  An alternative approach would be to pack all state variables into tuples. More on this in *section 4.2 -> creating a single tuple for all states*

  - **Not all functions require the same state variables.** Some functions may not even require state variables at all. It should be possible to select or shuffle the order of the state variables between each function call.

    The same goes for normal return values. Some functions require not the return values of the function before it, but also return values of functions even further back. Some return values will not be needed at all and some return variables are needed multiple times.

4.2 Bind functions for statements using a state

In this first section, we investigate a simplified version of the bind function as we introduced in *section 3.2*. The bind function we will be creating will not handle monadic values but will just pass normal variables around. We will mainly focus on the fact that it should support multiple return values and states.

A solution supporting multiple return values and states consists of three parts. Firstly: we create a mechanism with which we can select or re-order the return values and states for each function. This mechanism is necessary, because each function has its own signature. This means each function could need other arguments and have other return values. Secondly, we need a language construction to let functions be polymorphic in their arity. This is because we do not want to set a fixed limit on how many states or return values a bind function uses or returns. Lastly, we will create the bind function itself.

Variable shuffling

We firstly assume the bind function we need already exists and focus on differences in function call signatures. To be able to shuffle and select state variables, lambda expressions prove to be very powerful. A lambda function could, for example, wrap the function requiring the state. This would be useful when return values have to be shuffled, ignored, or edited. The lambda function just takes all variables as arguments, and passes them to the next function. An example of how this would look is included below. Note that in this example, we assume the bind function exists. In *section 4.2 -> Creating the bind funciton* we will actually create and define the bind function. We use a state called `*FS`, short for file system.

```
1  // This function copies the contents of a file
2  copyFile :: String String *FS -> *FS
3  copyFile fname1 fname2 fs
4  # (fs, file)      = fopen fname1 fs
5  # (content, file) = fread file
6  # fs              = fclose fs file
7  # (fs, file)      = fopen fname2 fs
8  # file            = fwrite content file
9  # fs              = fclose fs file
10
11 // Implementation using the bind
12 copyFile :: String String *FS -> *FS
13 copyFile fname1 fname2 fs
14 = fs                $>              // Pass of fs
15   fopen fname1      $> \fs ->       // Pass of fs to lambda, file
16   fread             $> \content ->  // Pass of contents to lambda, file
17   fclose fs         $>              // Pass of fs
18   fopen fname2      $> \fs ->       // Pass of fs to lambda, file
19   fwrite content    $>              // Pass of file
20   fclose fs
```

Code example 12: A lambda function can prevent some variables from being passed

At line 15, the lambda function is executed with parameters `fs` and `file`, both returned by `fopen`. Note that the variable `file` is also a state variable, and has a unique type. The lambda function only requires `fs`, so it will be executed as soon as it has got the first variable from the bind function. Now, if the lambda function would only have contained the function `fread`, the remaining `file` variable would be directly passed to `fread`.

However, this is not actually the case, because the scope of all lambda functions contain the entire function body from the point where they are defined. This means the bind function on line 16 will be executed *before* `fread`. The bind function cannot simply take the result of the function on the left side, but has to take the function on the left side itself. It has to combine the functions on either side of the operator and return a new function that executes the right side after the left. More on this in *section 4.2 -> Creating the bind function.*

Type sequences: creating functions with polymorphic arity

When trying to create a bind function which supports this type of behavior, we hit an abstraction ceiling. The problem is that functions are defined asymmetrically: they take variables as multiple arguments, but return them as a single value or tuple. These tuples will have to be unpacked in the bind function. When trying to create an abstract bind function capable of unpacking a tuple of any length, giving its elements as arguments to

the next function and handling the return value of said next function, we will have to create a function that is polymorphic in its arity.

To do this, we need type *sequences* of template variables that are resolved at compile time. We will introduce the following new syntax to Clean:

- In the function *type declaration*, a programmer can currently use a type like `a` to be polymorphic in that type. With the new syntax, the programmer can now also use `a[~]` to allow the function to be used with any sequence of types, making the function polymorphic in its arity.
- In the *definition* of the function, all symbols representing a sequence of variables have to use the same syntax: `x[~]`. This goes for formal parameters, actual parameters and variables defined in let-expressions or where-expressions.
- Type sequences can also be used to define the elements of a tuple. Simply place the type sequence symbol inside the brackets used to define a tuple: `(a[~])` in a type declaration or `(x[~])` in a function definition.
- Just as with normal polymorphism, type sequences will be resolved and checked at compile time. The compiler will output an error when a type sequence is used in two places where the sequence of types do not match.
- It is only possible to use a type sequence when it occurs in the call signature of the function it is used in. This is because for the type inference to work, it needs to know the length of the type sequences. Those lengths can be computed for all parameters, because at compile time the types of those parameters are known.

Before we define the rewrite rules for this new syntax, we first start of with an example of what one could do with this syntax. We define an alternate tuple constructor. Without type sequences it would be bound to fixed-size tuples, but now we can write a generalized one:

```
// Alternate tuple constructor
tuple      :: a[~]  -> (a[~])
tuple         x[~]   = (x[~])

// The syntax above translates to multiple instances of the function.
// Because Clean does not support function overloading with different
// arities, the arity of the type sequence will benoted with a (|x|)
// where x is the length of the type sequence.
tuple(|1|) :: a     -> a
tuple(|1|)    x      = x

tuple(|2|) :: a b   -> (a, b)
tuple(|2|)    x y    = (x, y)

tuple(|3|) :: a b c -> (a, b, c)
tuple(|3|)    x y z  = (x, y, z)

...
```

Code example 13: An introduction to type sequences

To use this function, you can just throw some variables into it.

```
test :: (Int, String, Boolean)
test = tuple 4 "Hello World" False
```

Code example 14: Usage of type sequences

In theory, an infinite number of variations on `tuple` will be available. In practice, upon compile time only one instance is created for each distinct, used sequence of variables. In our example case, the following tuple function overload will be generated and used in the test function:

```
tuple :: Int String Boolean -> (Int, String, Boolean)
tuple a b c = (a, b, c)

test :: (Int, String, Boolean)
test = tuple 4 "Hello World" False
```

Code example 15: Expansion of type sequences

Note that type classes cannot be used internally to represent these different methods. Type classes require all implementing functions to have the same number of arguments. By using type sequence to define a list of arguments, the number of arguments can differ for each case.

Also note that one-tuples do not exist in Clean. One-tuples will be interpreted as just a value.

The syntax rewrite rules for type sequences

In this subsection we create rewrite rules or type sequences. We first look at polymorphic uniqueness. We write these rewrite rules separately, because they work the same everywhere, and it greatly reduces the complexity of creating the other rewrite rules. After that, we create rewrite rules for using type sequences in the following use cases:

- Creating a function with a type sequence in its signature
- Creating a function with two or more type sequences in its signature
- Calling a function with a type sequence in its signature

Uniqueness in type sequences

```
// Rules for uniqueness (will be omitted in other rules below).
     a[~]  ->       a_1,    a_2, ...    a_n
    *a[~]  ->      *a_1,   *a_2, ...   *a_n
    .a[~]  ->     u:a_1,  u:a_2, ...  u:a_n
   ..a[~]  ->      .a_1,   .a_2, ...   .a_n

  (  a[~]) ->   (   a_1,    a_2, ...    a_n)
 *( *a[~]) ->  *(  *a_1,   *a_2, ...   *a_n)
v:( .a[~]) -> v:( u:a_1,  u:a_2, ...  u:a_n)[v<=u]
u:(..a[~]) -> u:(u1:a_1, u2:a_2, ... un:a_n)[u<=u1, u<=u2, ... u<=un]
```

Code example 16: Uniqueness rules for type sequences

This syntax was chosen, because it is consistent with the current syntax for uniqueness typing as it exists in Clean. One difference, though, is the syntax for uniqueness polymorphism. We currently have two cases of uniqueness plymorphism in type sequences: collective polymorphism and individual polymorphism:

- Collective uniqueness polymorphism is denoted with `.a[~]`. The elements of a type sequence are now polymorphic in their uniqueness, but they are either all unique or all not unique. When used in a tuple, the uniqueness of the tuple itself can be defined separately from its elements, similar to normal tuple definitions.
- Individual uniqueness polymorphism is denoted with a `..a[~]`. With this variant, unique types can coexist with non-unique types in a type sequence. As with collective uniqueness, tuples can define their own uniqueness separately from the sequence.

The uniqueness rules apply to all rewrite rules in the rest of this section.

Creating a function with a type sequence in its signature

Functions with a type sequence in its signature will be translated to many instances of the same function, with different arities. This is the same code as already shown in example 14

```
tuple      :: a[~]  -> (a[~])
tuple         x[~]   = (x[~])

// Will be translated to
tuple(|1|) :: a     -> a
tuple(|1|)    x      = x

tuple(|2|) :: a b   -> (a, b)
tuple(|2|)    x y    = (x, y)

tuple(|3|) :: a b c -> (a, b, c)
tuple(|3|)    x y z  = (x, y, z)

...
```

Code example 17: A function with a type sequence in its signature

A non-polymorphic instance of a polymorphic function using type sequences has an 'invisible' variable for each type sequence used in that function. This means that for a function `f(|p|)` where type sequence `x[~]` is used, every occurrence of `x[~]` can simply be replaced with `p` different variables. In, for example, `tuple(|2|)`, every occurence of `x[~]` is replaced with 2 variables.

While only one type sequence can occur in an argument list, other type sequence can occur *inside* of arguments, for example in tuple arguments or in function arguments.

```
// Function rapply takes function arguments and a function, and will
// call the function with the arguments
rapply :: a[~] (a[~] -> (b[~])) -> (b[~])
rapply x[~] f = f x[~]

// Will be translated to
rapply(|n,m|) :: a1 ... an (a1 ... an -> (b1, ..., bm)) -> (b1, ..., bm)
rapply(|n,m|) x1 ... xn f = f x1 ... xn
```

Code example 18: A function with multiple code sequences in its signature

Note that one instance of this function will be generated for each n and m. We can also see that to define the signature of this function, we need to define the arities of all type sequences. This is why we need two integers in the `(|n,m|)` brackets.

Calling a function with a type sequence in its signature

When calling a function that is defined with a type sequence in its signature, the Clean compiler has to know where the type sequence ends. We will create a simple implementation: when a type sequence is encountered, all remaining arguments will be a part of the type sequence until the type sequence is broken. A type sequence can be broken by:

- The end of the line
- An operator with a weaker priority then function call
- A closing bracket

To illustrate this, we will call the function `rapply` (defined in example ...) in two different ways:

```
1 // function call          Translated function call
2    rapply x y  f          rapply(|3,p|) x y  f
3   (rapply x y) f         (rapply(|2,p|) x y) f
```

Code example 19: Calling a function with a type sequence in its signature

In a normal Clean program, the two calls of `rapply` would be exactly the same. However, because we use type sequences, the brackets now have the effect to break the sequence list.

- In the implementation on line 2, the variables `x`, `y` and `f` are all part of sequence list `a` would be arguments to the function called in the body of `rapply`. The resulting type of this function call would be `(x y f -> (z[~])) -> (z[~])`, because the last argument `f` of `rapply` has not yet been given.
- In the implementation on line 3, the variables `x` and `y`, but not `f` would be arguments to the function `f` called in the body of `rapply`. The variable `f` is *not* part of sequence list `a`. The resulting type of this function call would be a tuple, but its exact type is dependent on `f`.

Also note that the value of `p` (the length of the second type sequence) cannot be computed without knowing the return value of `f`.

Creating a single tuple for all states

Another solution might be to pass multiple states as a single tuple. This way, you don't have to unpack the tuple containing all the states, but you can just pass it on to the next function. While this solution looks promising at first look, it has some severe limitations.

First of all, we would also have to pack all return values in a single tuple. The result of almost every function would then be a 2-tuple with return values (in another tuple if there are multiple) at the left side and state variables (again in a tuple if there are multiple) at the right side. This makes creating the bind function a lot easier and removes the need for type sequences. However, it adds some complexity and clutter to almost all code, because a lot more brackets will have to be used. This is not a realistic solution when we actually want to reduce clutter in code.

Another drawback is backwards compatibility. All (old) code using other argument/return formats then the return-value/state-variable split are incompatible with this bind function and has to either be rewritten, worked around or otherwise incorporated into such new code.

Lastly, we would lose the implicit currying of functions in Clean. Because all arguments are grouped together in tuples, we cannot partially call a function anymore. This would be a very powerful feature to loose.

Altogether, it would probably not be a good idea to group return values and state variables together in tuples.

Creating the bind function

We do now have all the ingredients we need to create the bind function. A first version of the bind function would be very simple: take the tuple the function on the left returns, and feed its arguments to the function on the right. With the newly added type sequences, this becomes very easy:

```
1 ($>) infixl 0 :: .(..a[~]) (..a[~] -> .b) -> b
2 ($>) (x[~]) f = f x[~]
```

Code example 20: A simple bind function

This bind function can be used to carry all return values from a function over to the right side of the operator:

```
// Existing functions
f   :: *State -> *State
g   :: Int *State -> *State
h   :: *State -> (Int, *State)

k   :: *State -> *State
k state =
    state $>
    h $>
    (\i -> g (i+1)) $>
    f

// In this case, the three instances of >>= are created, with the
// following type declaration:
// first $> on line 8:      a: *State          b: Int, *State
($>) infix 0 :: (*State) (*State -> (Int, *State)) -> (Int, *State)
// Because of the one-tuple, this is equivalent to:
($>) infix 0 :: *State (*State -> (Int, *State)) -> (Int, *State)

// second $> on line 9:     a: Int, *State     b: *State
($>) infix 0 :: (Int, *State) (Int *State -> *State) -> *State

// third $> on line 10:     a: *State          b: *State
($>) infix 0 :: *State (*State -> *State) -> *State
```

Code example 21: Using the simple bind function

This implementation of the bind function has a problem that the let-notation does not have: it does not allow for references further back in the code. Using let-statements, we can use the variables created there from that point onward until the end of the function body:

```
// Existing functions
g   :: *State -> (Int, *State)
h   :: Int Int *State -> *State

f :: *State -> *State
f wld
# (i, state)  = g state     // At the last line, we still have access
# (j, state)  = g state     // to i and j, despite the fact that they
# ...                       // were defined a few lines ago.
=               h i j wld
```

Code example 22: We can reference variables further back in the code

When using bind functions, we need to be able to do the same. However, with the current implementation of the bind function, the lambda function needs to be closed *before* the next bind function occurs. This is because the bind function needs the values the left-hand side function returns, and cannot deal with the function itself. However, closing this lambda function has the side-effect that the variables in it's scope are not available later on. In other words: the bind function is left associative, but the usage of lambda functions need the bind function to be right associative.

```
// Existing functions
g   :: *State -> (Int, *State)
h   :: Int Int *State -> *State

f :: *State -> *State
f state = state $>
    g $>                // This will fail, as the last bind function
    (\i -> g $>         // will now have function g' as argument, which
    \j -> h i j)        // has a type incompatible with what h needs.

f :: *State -> *State
f state = state $>
    g $>                // This will fail, as i is not available
    (\i -> g) $>        // outside of it's lambda function scope.
    (\j -> h i j)
```

Code example 23: We can not reference variables further back in the code

We can,however, create a right associative bind function that handles this correctly. This bind function does not take the *result* of the function on the left-hand side, but the function itself. It then takes at the end of the parameter list all parameters the left-hand side function still needs. We will call this right associative version of the bind `$$>`. However, such infix operator would require at least three arguments, which is not possible. An infix operator must have exactly two arguments. The solution for this is to simply let the infix operator return a new function that requires the leftover arguments.

```
($$>) infixr 0 :: (..a[~] -> .(..b[~])) (..b[~] -> .c) -> (.a[~] -> .c)
($$>) fl fr = g fl fr
where
    g fl fr v[~]
    # (v[~]) = fl v[~]
    =          fr v[~]
```

Code example 24: A right associative version of the bind function

To explain why this works, we will look at the simple example below. Note that the first bind function is also new (`$>>`). This is just a right associative version of the old bind function. We need this new version, because using the left associative version here will leave the Clean compiler confused over what bind to give a higher priority. Changing the old bind function to right associative is also not an option, because it will break the old functionality.

```
1 g :: Int *State -> *State
2 g x v =
3     v       $>>
4     f x     $$> \i ->
5     f (i+5)
```

Code example 25: Using the right associative bind function

Because this version of the bind function is right associative, the bind function on line 4 has the lowest priority. This function takes the left side function, the right side function and a state variable which is not given just yet. Because this last state variable is not given yet, the result is a new function that:

- takes a state variable
- passes it to the left hand side function
- extracts the values from the tuple it returns
- feeds these values as arguments to the right hand side function
- returns the return value of the right hand side function

Lastly, the bind function on line 3 gets called. It passes the state variable to the function resulted from the bind function on line 4, and everything will be executed. This bind function is extremely powerful, especially because we combined it with the type lists introduces in this section a few pages back. This means, the following pieces of code are equivalent:

```
// Assume the following type exists:
:: *State = ...

// Pass all arguments of the first function with the first bind.
g :: Int *State -> *State
g x w =
    x w     $>>
    f       $$> \i ->
    f (i+5)

// Omit the first bind function.
g :: Int *State -> *State
g x w =
    f x w   $$> \i ->
    f (i+5)

// Omit giving it the world in the first place. This slightly changes
// the type of the function.
g :: Int -> (*State -> *State)
g x =
    f x     $$> \i ->
    f (i+5)
```

Code example 26: Simplified usage of the right associative bind

Conclusion about the bind

Two pieces of code are included below. The first one contains a shortened version of a real life function out of Leaflet.icl [5]. The second one contains the same code, but now using the bind functions we created in this section.

Old code:

```
addPopup :: String Pos Marker *World -> *World
addPopup content position marker world
# (options,world) = jsEmptyObject world
# world           = (options .# "maxWidth" .= 1000000) world
# world           = (options .# "closeOnClick" .= False) world
# (popup, world)  = (l .# "popup" .$ options) world
# (_, world)      = (popup .# "setLatLng" .$ position) world
# (_, world)      = (popup .# "setContent" .$ content) world
# (_, world)      = (mapObj .# "addLayer" .$ popup) world
# world           = (marker .# "myPopup" .= popup) world
=                   world
```

New code

```
addPopup :: String Pos Marker -> (*World -> *World)
addPopup content position marker =
    jsEmptyObject                          $$> \options ->
    (options .# "maxWidth" .= 1000000)     $$>
    (options .# "closeOnClick" .= False)   $$>
    (l .# "popup" .$ options)              $$> \popup ->
    (popup .# "setLatLng" .$ position)     $$> \_ ->
    (popup .# "setContent" .$ content)     $$> \_ ->
    (mapObj .# "addLayer" .$ popup)        $$> \_ ->
    (marker .# "myPopup" .= popup)
```

The last version of the bind function we constructed is extremely powerful and does exactly what we want. Code written and chained together with this bind function looks linear and clear. It does not look exactly like imperative code would, but that is not a goal nor a problem. It will, however, take a little getting used to this style of programming. A big drawback is that type sequences should be implemented before this is really usable. This will be a radical change to the Clean language, and a lot has to happen before this can actually be implemented.

4.3 Syntactical solutions

As an alternative to creating bind functions, we can change the syntax of the #-statement to simplify passing a state around. In this section, we will discuss a few different language constructs to do this, and create a proposal for the new syntax rules. Finally, we will transform a (shortened) method in `Leaflet.icl` to give an example about the new syntax rules. We use the same example to illustrate why these changes are necessary:

```
addPopup :: String Pos Marker *World -> *World
addPopup content position marker world
# (options,world) = jsEmptyObject world
# world           = (options .# "maxWidth" .= 1000000) world
# world           = (options .# "closeOnClick" .= False) world
# (popup, world)  = (l .# "popup" .$ options) world
# (_, world)      = (popup .# "setLatLng" .$ position) world
# (_, world)      = (popup .# "setContent" .$ content) world
# (_, world)      = (mapObj .# "addLayer" .$ popup) world
# world           = (marker .# "myPopup" .= popup) world
=                   world
```

Code example 27: Ugly code from Leaflet.icl

Testing correctness of syntactical changes

When introducing new syntax or changing old syntax, it is important that it doesn't clash with any other existing syntax. To test this, we can just feed the new syntax to the Clean compiler. If the Clean compiler gives a general syntax error, we know the syntax is not used for anything else. We will use this method to check all newly introduced syntax in this chapter. Also note that none of this new syntax is actually implemented at the time this thesis was finished.

Omitting parentheses around tuples

The need for parentheses around every composed return value creates a lot of clutter very quickly. It would be really nice to be able to skip the parentheses and just list the variables. Additinoally, the syntax proposals we do later on in this section benefit from it a lot.

At the moment, the Clean compiler throws an error when executing the following code:

```
1 ...
2 # x, y = ...
3 ...
```

*Code example 28: **Error**: Unexpected '‘,’', expected '‘=’'*

As discussed in *section 4.3*, it is safe to adapt the Clean parser to support the omitting of brackets. The decision what to do with singleton tuples is not a problem, because Clean does not support singleton tuples tuples. They will just be interpreted as variables.

State piping: omitting `state` identifiers in a single line

When using a state, it has to be given to and returned by many functions. This results in two state-variables per line, cluttering a lot of code. This is especially so when using multiple state variables. To try and prevent this, we will discuss two syntax constructs, the first being state piping.

The idea is very simple: at the left-hand side of the assignment we will denote what variables are state variables. These variables will implicitly be added at the end of the right-hand side of the assignment. In *section 4.1* we stated a few assumptions regarding state variables.

- The state variables will be returned at the end of a tuple. This means all other return values will be in the same tuple, but before the state variables.
- The state variables will be passed to the function in the same order as they will be returned.

- When a function in a #-expression cannot comply with the above assumptions /conventions, one can always revert to passing the state variable manually.

This brings us to the following syntax rules. Each function is expressed in the new syntax, with directly below it the equivalent in the old syntax.

```
# (x, y)                 = f          // Conventional function, without
# (x, y)                 = f          // state

# (x | state)            = g          // Function with state. The state can
# (x, state)             = g state    // be omitted at the end. The pipe
                                       // denotes `state` to be a state
                                       // variable.

# (x | state1, state2)   = h          // Example with multiple states
# (x, state1, state2)    = h state1 state2

# (x, y | state)         = i          // Example with multiple returns
# (x, y, state)          = i state

# (|state)               = j          // Example with only a state
# (state)                = j state

# x | state              = k          // As discussed above, brackets may
# (x, state)             = k state    // be omitted
```

Code example 29: The syntax rewrite rules for state piping

Currently, the Clean compiler will throw a syntax error when a pipe is used instead of a comma. As discussed in *section 4.3 -> Testing correctness of syntactical changes*, this new syntax can safely be implemented.

State lacing: omitting `state` identifiers in a block of code

When using a few methods using the same state variable or state variables, it could be possible to omit them altogether and just have them passed implicitly. The syntax for this should be adaptable for each chunk of code, not limited to whole function bodies. Additionally, the syntax should be notable, to ensure people can clearly see what is going on.

The general idea is simple: When a chunk of code uses the same state variable or state variables on each line, we can add a line before and after to indicate which state variable(s) should be passed between functions. We have got two proposals for the operator used in this new syntax:

```
// Proposal 1:
...
#!! state               // Set state variables in lacing to `state`
  r = f                 // Translates to (r, state) = f state
#!! state1              // Set state variables in lacing to `state1`
...
#!!                     // Set state variables to nothing
...

// Proposal 2:
...
#!{ state               // Set state variables to `state`
  r = f                 // Translates to (r, state) = f state
#!}{ state1             // Reset state variables to `state1`
...
#!}                     // Remove state variables
...
```

Code example 30: Syntax for a state lacing

The first example uses an update-based approach, where the second example uses a block-based approach. All operators are tested as discussed in *section 4.3 -> Testing correctness of syntactical changes* and are safe to use. Both examples allow the lacing of multiple state variables:

```
#!! state1, state2
  r = f
#!!

#!{ state1, state2
  r = f
#!}
```

Code example 31: Lacing multiple states

`#!!` vs `#!{` and `#!}`

Both proposed operators have their advantages and disadvantages. The `#!!` operator has a more update-based feel to it. With a `#!!` operator, you are basically saying: "from here we use this state lacing configuration", whereas with a `#!{` and `#!}` operator, you are defining a block of code a state lacing is active on. This also means (intuitively), a `#!{` block always needs to be closed off with a `#!}`, whereas an active `#!!` can just be terminated implicitly at the end of a function body.

Another consequence of using the `#!{` and `#!}` is that it would be possible to allow nesting of multiple blocks. This brings us to the next questions: Should it be allowed, and should it be accumulative? The three options are illustrated below.

```
// The following fragment will be transformed into the old
// syntax for each of the three semantic options:
#!{ state1
  x = f
  #!{ state2
    y = g
  #!}
  x = f
#!}

// Semantics of first option: Do not allow nesting
Syntax error: ...

// Semantics of second option: Allow non-accumulative nesting
# (x, state1)           = f state1
# (y, state2)           = g state2
# (x, state1)           = f state1

// Semantics of third option: Allow accumulative nesting
# (x, state1)           = f state1
# (y, state1, state2)   = g state1 state2
# (x, state1)           = f state1
```

Code example 32: The syntax rewrite rules for different implementations of a state lacing

While accumulative nesting comes more intuitively from the syntax, non-accumulative nesting is more powerful and easy to understand while programming. The main use case for these blocks is, probably, to call another type of library for a few lines. Because this can logically be seen as a small block within the code, it makes more sense to choose for non-accumulative nesting. A nice side effect is that these blocks can be used to disable state lacing, without exiting the block and re-enabling all states:

```
#!{ state1, state2
  x         = f         // These functions will have state1 and state2
  y         = f x       // laced through them
  #!{                   // Start a new lacing block without actually
    z = x + y           // lacing any state variables
    a = 1 + 2
  #!}                   // Lacing  of state1 and state2 will be
  ...                   // continued from here
#!}
```

Code example 33: temporary disabling a state lacing

My final recommendation would be to choose `#!{`-style state lacing, and allow for non-accumulative nesting. This is just a personal preference, though, and this should be discussed further among multiple people before a decision can be made.

Assignments without left-hand side

Both solutions creates an edge case, however: what would be the syntax when a function only returns the state? According to the rules above, the syntax would be:

```
#!{ state
  = f
  ...
#!}
```

Code example 34: Some statements in a state lacing only return states

This, however, conflicts with existing syntax, as the compiler would think this is the end result of the function. To fix this, we will discuss the four proposals below:

```
#!{ state
#    = f     // 1: Do not introduce extra syntax, but use the already
             // existing '#'
  _  = f     // 2: Use the already existing "nothing" indicator
             //
  ^  =       // 3: New syntax: use a caret to point to the state lacing
             // started earlier
     => f    // 4: Use an alternative '='
#!}
```

Code example 35: Syntax for expressions that only return a state

Solution 1 has the big advantage that no new syntax has to be introduced. The big disadvantage, however, is that the `#` is usually positioned an indent to the left to the other statements (as can be seen above).

Solution 2 needs to implement new syntax, but uses an already existing identifier to indicate there is "nothing". The problem is the current use is fundamentally different to the way we will be using it. In the existing syntax, the `_` shows there is something, but we just do not want to use it anymore. In the new syntax, it would mean we want to use something that is implicitly there.

Solution 3 currently seems the most promising of the three: we introduce a new symbol `^` to denote we want to use the implicit state. It is pretty intuitive, does not break indentation and does not break any existing syntax.

Solution 4 has two disadvantages: it is not very intuitive and it breaks the indentation directly after the `=`.

Lace skipping: calling a function without the state variable in a lacing

In many use cases, lines needing a state and lines not needing a state are alternating each other. When few lines need a state, state piping can be applied. When all lines need a state, state lacing can be applied. There is one use case, however, where these techniques fall a bit short. This is when almost all lines need a state, but not all of them. In this case, state piping still results in many `state` identifiers, but state lacing requires two extra lines of code around each statement or group of statements not requiring a state. A solution would be to create another syntax rule, similar but inverse to the syntax of state piping:

```
1 #!{ state
2   integer           = f
3   integer \ state   = integer - 1
4   newinteger        = f
5 #!}
```

Code example 36: Skipping an expression in a state lacing

Here we use a backslash to indicate we want to temporarily remove a state from being laced into the right hand side expression. Of course, this will also work with multiple states:

```
#!{ state1 state2
  integer           = f     // Takes both states
  integer1 \ state2 = g     // Takes only state1
  integer2 \ state1 = h     // Takes only state2
  integer3 \ state1, state2 = integer + integer1 + integer2
                            // Takes no states, both are temporary
                            // removed from the state lacing

  integer3 \        = integer + integer1 + integer2
                            // A trailing \ can be used as a shorthand
                            // to temporarily remove all states from
                            // lacing
#!}

// Is equivalent to this old syntax:
# (integer, state1, state2) = f state1 state2
  (integer, state1)         = f state1
  (integer, state2)         = f state2
  integer3                  = integer + integer1 + integer2
  integer3                  = integer + integer1 + integer2
```

Code example 37: Skipping an expression in a state lacing with multiple states

Conclusions of syntactical changes

Two pieces of code are included below. The first one contains a shortened version of a real life function out of Leaflet.icl [5]. The second one contains the same code, but now using the proposed syntax we created in this section.

Old syntax:

```
addPopup :: String Pos Marker *World -> *World
addPopup content position marker world
# (options,world) = jsEmptyObject world
# world           = (options .# "maxWidth" .= 1000000) world
# world           = (options .# "closeOnClick" .= False) world
# (popup, world)  = (l .# "popup" .$ options) world
# (_, world)      = (popup .# "setLatLng" .$ position) world
# (_, world)      = (popup .# "setContent" .$ content) world
# (_, world)      = (mapObj .# "addLayer" .$ popup) world
# world           = (marker .# "myPopup" .= popup) world
=                   world
```

New syntax

```
addPopup :: String Pos Marker *World -> *World
addPopup content position marker world
#!{ world
# options = jsEmptyObject
# ^       = (options .# "maxWidth" .= 1000000)
# ^       = (options .# "closeOnClick" .= False)
# popup   = (l .# "popup" .$ options)
# _       = (popup .# "setLatLng" .$ position)
# _       = (popup .# "setContent" .$ content)
# _       = (mapObj .# "addLayer" .$ popup)
# ^       = (marker .# "myPopup" .= popup)
#!}
=           world
```

Note that not all syntax we proposed is used in this example.

After transforming `Leaflet.icl` to the new syntax, we can count the number of occurrences of `world` to get a global indication of the clutter we removed.

File	Line count	`world` count
`Leaflet.icl`	526	476
`Leaflet-transformed.icl`	584	153

As we can see, the occurences of `world` are greatly reduced. The remaining `world` symbols occur mainly in function type declarations (such as the type `*World`), at the beginning of a lacing (as the line `##{ world`) and at places where the functions called are too diverse to actually implement a lacing. On those lines, a piping was used.

Another notable change is the addition of 58 lines. These added lines consist of opening and closing lines for world lacings.

Chapter 5

Conclusions

In chapter 4.1 we defined the requirements of how state variables are used, and what cases of usage we should implement. The purpose of this section is solely to create an overview.

In chapter 4.2 we looked at the possibility of using functions to hide the state variables from the end user. It looks promising, but it would need some far-reaching changes in the Clean language. Clean simply is not abstract enough in some parts of the language, so we would need to make some changes to the language to make it work.

In chapter 4.3 we took another approach and tried to change the syntax to do approximately the same. With some simple syntax changes we managed to improve the readability of a trial function by a lot. This solution obviously needs some changes to the Clean language, but these are limited to the Clean parser. They do not change the core functionality of Clean, unlike the adaptations proposed in chapter 4.2.

While both proposals result in elegant solutions, the solution introducing new syntax to handle state variables seems to have a few advantages over the solution using bind functions. It is a lot less complex to implement, and it is a lot less imposing on new programmers. Even though other researchers have had some success tackling this problem using monads and monad transformers, the final recommendation for this thesis will be to use the new syntax rules.

We can now answer the research question:

'How can we increase the readability and ease of use for a state variable in a functional programming language?'

The answer is: by adding some syntax to remove tuple brackets and remove a large portion of the state identifiers. The details can be found in chapter 4.3.

Chapter 6

Further research

While this paper covers a lot of the possibilities to improve the Clean syntax, some other areas are left to explore:

- **Extend type sequence syntax?** *[section 4.2 -> type sequences]*

  We introduced type sequences, a new language construct needed to define the bind function. Currently, the rules for using a function with type sequences in its signature are pretty strict: all arguments to be put in the type sequence have to be given in one single call to the function. Perhaps it is possible to extend the syntax of type lists to lift this restriction?

- **Are the `.#`, `.$` and `.=` really necessary?** *[section 1.1: third problem]*

  Beside the state passing, another ugly construct when programming for Javascript is the occurrence of `.#`, `.$` and `.=` when traversing Javascript objects. When the types used in the library are properly indexed, it should be possible to create native Clean types. Maybe some workarounds or language changes have to be done to support this. It would certainly be good to explore further.

- **Automatically generate the types mentioned above from Typescript code** *[section 1.1: fifth problem]*

  If such things as mentioned above become possible, it should be possible to generate native Clean types from Typescript implementations or definition files for a library.

References

[1] P. Wadler, "Monads for functional programming," in *Advanced functional programming*, 1995, pp. 24–52.

[2] P. Achten, J. H. G. van Groningen, and R. Plasmeijer, "High level specification of i/o in functional languages," in *Proceedings of the 1992 glasgow workshop on functional programming*, 1993, pp. 1–17.

[3] R. Plasmeijer, P. Achten, and P. Koopman, "iTasks: Executable Specifications of Interactive Work Flow Systems for the Web," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, 2007, pp. 141–152.

[4] P. Koopman, R. Plasmeijer, M. van Eekelen, and S. Smetsers, *Functional programming in clean*. 2002.

[5] "Leaflet iTasks wrapper." [Online]. Available: https://gitlab.science.ru.nl/clean-and-itasks/iTasks-SDK/blob/master/Libraries/iTasks/Extensions/GIS/Leaflet.icl.

[6] V. Agafonkin., "Leafletjs," 2017. [Online]. Available: http://leafletjs.com.

[7] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," *SIGPLAN Not.*, vol. 45, no. 6, pp. 1–12, Jun. 2010.

[8] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *ECOOP 2014 – object-oriented programming*, 2014, pp. 257–281.

[9] D. K. Gifford and J. M. Lucassen, "Integrating functional and imperative programming," in *Proceedings of the 1986 acm conference on lisp and functional programming*, 1986, pp. 28–38.

[10] S. L. P. Jones and P. Wadler, "Imperative functional programming." 1993.

[11] L. Erkök and J. Launchbury, "A recursive do for haskell," in *Proceedings of the 2002 acm sigplan workshop on haskell*, 2002, pp. 29–37.