Bachelor's Thesis Computing Science

Radboud University Nijmegen

Task Oriented Programming in Lua

Author:
Dante van Gemert
s1032684

First supervisor/assessor:
dr. Peter Achten

Second assessor:
dr. Pieter Koopman

June 27, 2022

Abstract

Task Oriented Programming (TOP) is a programming paradigm centered around *tasks*. Its implementations are written in the functional language Clean. Lua is a procedural language that is very different to Clean. This thesis explores the design space that appears when implementing TOP in Lua. We create a proof-of-concept implementation of TOP in Lua and show that Lua has some benefits over Clean for implementing TOP.

Contents

1 Introduction **2**

2 Preliminaries **3**
2.1 Task Oriented Programming 3
2.2 Lua 7

3 Research **13**
3.1 Task types 13
3.2 Tasks and task values 16
3.3 Type representation 18
3.4 Task combinators 20
3.5 Editors and user interface 25
3.6 LTasks 27
3.7 Wrap-up 33

4 Comparison **34**
4.1 Editors 34
4.2 Helper function 35
4.3 User interface 35
4.4 When to use which 36

5 Related Work **38**

6 Conclusions **39**
6.1 Future work 39

A Examples in LTasks and iTasks **43**
A.1 Breakfast 43
A.2 Multiple date input formats 45

B LTask code **48**
B.1 task.lua 48
B.2 types.lua 55
B.3 ltuiEditor.lua 58

Chapter 1

Introduction

Task Oriented Programming (TOP) is a programming paradigm where code is structured using *tasks*. Currently, the implementations of TOP are written as a shallowly embedded DSL in the pure, lazy, strongly typed, functional language Clean. This gives programming in TOP a functional taste, since using these implementations means using features of the functional host language.

In this thesis we break away the concept of TOP from its functional implementation by asking how we can develop a task oriented programming implementation in the interpreted and dynamically typed procedural language Lua. This implementation in Lua will give programming in TOP a more procedural feel. By placing TOP in an environment that is radically different than Clean, we can see what new and interesting design decisions appear that were not clear when using Clean.

We first go over the preliminary knowledge in chapter 2, where we find the essence of TOP and give a brief introduction to Lua. In chapter 3 we explore the design space of implementing TOP in Lua. With that information, we create a proof-of-concept implementation called LTasks for which we explain the choices made and how it is implemented. We compare this implementation to iTasks (one of the current implementations in Clean) in chapter 4. Chapter 5 lists the related work, and we conclude the thesis in chapter 6. Lastly, appendix A contains code examples in both LTasks and iTasks and appendix B has the most important code files for LTasks.

Chapter 2

Preliminaries

This chapter provides the necessary background information on the two most important topics in this thesis. Section 2.1 explains the concept of the task-oriented programming paradigm, and section 2.2 goes over the basics and the most important features of the Lua programming language.

2.1 Task Oriented Programming

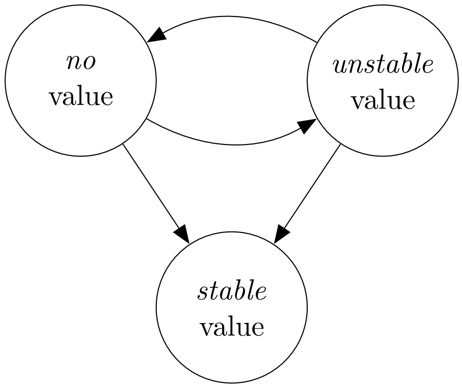
In Task Oriented Programming [12], a task is just like a task in the real world: a description of something that needs to be done, an abstract unit of work. A task can have an observable intermediate value and access to shared information. Some tasks are to be performed by a human and some can be done by a computer. By composing tasks in various ways, it is possible to create complex applications.

There are two implementations of TOP: iTask [11], written in the functional language Clean, is used for developing interactive distributed applications. mTask [6, 7], written in Clean and C++, is used for IoT devices which are constrained in their resource usage. Both of them are a shallowly-embedded domain-specific language (EDSL). In this thesis we will primarily be comparing against iTasks.

A TOP implementation must provide the concept of tasks, ways to compose the tasks where one task can read the value of another task, shared data sources and for interactive TOP systems also some form of user interface.

2.1.1 Task value

Tasks can have an *intermediate value*. A task value can be in one of three states: **no value**, **unstable** value or **stable** value [1, §4.3]. A task can switch between no value and an unstable value, and it can switch to a stable value. Tasks with a stable value have a final result and do not change anymore. Figure 2.1 shows this graphically. Even when a task has no stable



Listing 2.1: The possible states of task values

value yet, its intermediate value can be observed. Tasks can observe the value of other tasks.

In iTasks, tasks can also throw *exceptions*, implemented as returning an exception value [12, §3.1.1]. When it is clear that a task will never result in a meaningful value, it can raise an exception. This can happen for instance when a network connection fails.

2.1.2 Task composition

Tasks can be *composed* in multiple ways, falling in one of these categories: **sequential** composition and **parallel** composition. They both use the fact that a task's value can be observed. Sequential composition is named `step` in iTasks. We can provide it with multiple continuation tasks, one of which will be executed based on the observed task value. Parallel composition executes all provided tasks. The values of these tasks are combined to form the value of the parallel task, and these values can also be observed by the provided tasks. There is a way to create tasks with a stable value, called `return` in iTasks. The value and type of a task can be transformed with iTasks' `@` operator.

We use the example of having breakfast, adapted from Naus [10]. To make breakfast, you first make something to drink. This can be either tea or coffee. You also make something to eat, a sandwich in this example. You can do that already while you are waiting for your drink to be ready. When you have your drink and your sandwich, you can eat it. The whole operation of having breakfast can be seen and modelled as a task, composed of smaller tasks.

The breakfast example using the combinators `>>-` (sequential), `-&&-` (parallel and) and `-||-` (parallel or) from iTasks looks like listing 2.2.

A task composed sequentially does not have to wait for the first task to complete before starting. It can even be the case that task 2 becomes stable

```
(makeTea -||- makeCoffee) -&&- makeSandwich >>- eatBreakfast
```

Listing 2.2: Example: making breakfast.

when task 1 still has an unstable value. For example, task 1 could give an overview of free hospital beds. Task 2 could then decide to continue (have a stable value) once there is a suitable bed available, while the overview of hospital beds will never be stable. This situation looks like this in iTasks:

```
hospitalBeds >>* [OnValue (ifValue (\beds = length beds > 0)
    (\beds = return (hd beds)))]
```

Or think of a task that waits for a specific time before returning a value. Task 1 (a task that yields the current time) will always have an unstable value (because time keeps changing), but eventually the waiting task becomes stable.

2.1.3 Shared data sources

Tasks are *distributed* and *concurrent*. For instance, in a multi-user system, two tasks that are composed in parallel could be done by two different users at the same time. Note that a TOP implementation does not have to be a multi-user system. While iTasks is multi-user, mTasks is not (because having users makes little sense in an IoT environment).

Sharing data between tasks is done with shared data sources like files or sensors. These can interact with tasks. Not all shared data sources have to be both readable and writable; the current date and time are examples of read-only shared data sources. When tasks are composed in parallel, they get a read-only shared data source that reflects the current value of each other [12, §2.1]. The following is an example of a SDS that is both readable and writable, letting the user input a list of words and immediately showing the sentence made from these words.

```
wordsSDS :: SimpleSDSLens [String]
wordsSDS = sharedStore "wordsSDS" []

wordsTask = (updateSharedInformation [] wordsSDS <<@ Title "enter words")
    -|| (viewSharedInformation [ViewAs (foldl (+++) "")] wordsSDS
        <<@ Title "sentence view")
```

2.1.4 User interface

Interaction of tasks with humans happens with interactive tasks called *editors*. An editor task is a bridge between the internal world of tasks and the external real world. A TOP framework automatically generates an appropriate user interface with these editors. They also allow users to interact with shared data sources.

An editor task never has a stable value. When an editor task is composed sequentially, even when the user has pressed a "continue" button and the sequential task has moved on to the next task, the editor stays unstable behind the scenes.

User interfaces of combined tasks are composed of the user interfaces of the components. For example in iTasks, if two tasks assigned to the same user are combined in parallel, they are shown next to each other. [10, §4.2.4]

Since iTasks is written in the statically typed language Clean, the possible values a task can have are predetermined by its type. This allows us to make use of the existing HTML form input fields when generating a web interface. This is useful for two reasons. First, these form fields only allow valid input, i.e. you can't input arbitrary text in a number field. Second, they improve usability by adapting the input method, for instance displaying a date picker.

This generation of different HTML form fields is done automatically in iTasks. This is possible because Clean has generic types and is statically typed—types are available statically, before any value is present.

2.1.5 Implementation in iTasks

Since iTasks is written in the functional language Clean, tasks are modelled as functions. Functions in Clean are pure; they cannot mutate existing values. Instead, a task is implemented as a function that can process an event, using the information stored in the `IWorld` environment, and that results in a new version of itself, the task value, any user interface changes, and the `IWorld`. In listing 2.3 we can see that the task function (`Task`) returns a `TaskResult` and an `IWorld`. `IWorld` carries information about the entire TOP application such as the current clock, and we will not go into it further.

A task value is either `NoValue`, or `Value` together with a value and a boolean `Stability` signifying whether it is stable. Note that this models the three possible states from 2.1.1 correctly: a task with no value also has no stability.

```
:: Task a (=: Task` (Event -> TaskEvalOpts -> *IWorld
        -> *(TaskResult a, *IWorld)))

:: TaskResult a
    = ValueResult !(TaskValue a) !TaskEvalInfo !UIChange !(Task a)
    | ExceptionResult !TaskException
    | DestroyedResult

:: TaskValue a
    = NoValue
    | Value !a !Stability
```

Listing 2.3: The types `Task`, `TaskResult` and `TaskValue` in Clean.

2.2 Lua

Lua is an interpreted and dynamically typed programming language. The website lists its selling points: Lua is claimed to be fast, portable, embeddable, powerful but simple, small and free [5]. Lua is *fast* because the LuaJIT just-in-time implementation can yield performance that can be reasonably compared to that of C [2, fig. 11]. It is *portable* since the Lua interpreter is written in plain C and can run on any device. Lua us built as an extension language, so it is *embeddable* with the use of its C API. It is *powerful but simple*: the language has little syntax and concepts, but it provides meta-mechanisms which let you implement features yourself. The language is *small* since the interpreter and libraries take less than 800kB, and it is *free* because it is distributed under the MIT license.

The next subsections will cover the aspects and concepts of Lua that are most relevant for this thesis. Details that are not necessary are left out, but can be found in the reference manual [4].

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. [5]

The Lua logo

2.2.1 Where is Lua used?

Lua is described by the authors as an extensible extension language [3]. This means that it is made to be extended (for instance with bindings to libraries written in C), and to be used within other applications.

Lua is used for example as a language for making games (Roblox[1,2], Love2D[3]), as a scripting language within games (Minecraft mods[4,5]), as a scripting language in other programs (Adobe Photoshop Lightroom[6], LuaTeX[7], OpenResty[8]) and also in microcontrollers for IoT projects (NodeMCU[9]).

[1] https://www.roblox.com/
[2] https://luau-lang.org/
[3] https://love2d.org/
[4] https://computercraft.cc/
[5] https://oc.cil.li/
[6] https://www.adobe.com/products/photoshop-lightroom.html
[7] http://www.luatex.org/
[8] https://openresty.org/en/
[9] https://nodemcu.readthedocs.io/en/release/

2.2.2 Basics

Lua has 8 basic types: `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread`, and `table` [4, §2.1].

Some interesting notes: The type `nil` (which has a single "value", `nil`) signifies the absence of a value (`null` or `undefined` in other languages). Any variable or field with no value implicitly holds `nil`, and a non-existing variable or field cannot be distinguished from one with an explicit `nil` value. Lua has only a single `number` type, which internally switches from integers to floating point numbers automatically. In versions prior to 5.3, numbers were always floats. `userdata` is the type of data that comes from C, we can ignore it here. `function`, `thread` and `table` will be covered in depth in the next sections. All values are first-class: they can be stored in variables and tables, passed to functions and returned from functions.

By default, variables are global. To make them local, you put `local` in front of them: `local localVar = "local"` instead of `globalVar = "global"`. This also works for functions, though that is not as common. Lua supports parallel assignment: you can swap variables without using a temporary variable by writing `x, y = y, x`.

The types `table`, `function`, `thread` and `userdata` are reference types (the Lua manual calls them *objects* [4, §2.1]). This means that `{} == {}` evaluates to `false` because they refer to two separate tables. However `"" == ""` evaluates to `true` because `string` is not a reference type.

2.2.3 Tables

Lua's main (and only) composite data structure is the table. A table is an associative array and fulfills the purposes of arrays, objects and maps in other languages. A table can store any type of key except `nil`, and any type of value. Note that this means it is possible to have keys of reference types or *objects* (such as tables, defined above). Having values stored by integer keys creates a kind of list. Since Lua is dynamically typed, a table can hold both integer keys/values and other type keys/values at the same time: a single table can function as both a list and a map. Indexing a table with an arbitrary type key uses square brackets (`tbl[key]`). There is a shorthand for accessing string keys: `tbl.key` is syntactic sugar for `tbl["key"]`.

Tables as lists are 1-indexed, though nothing prevents you from assigning a value to key `0` (or negative numbers). Removing an item from a table can be done by setting it to `nil`, and attempting to access a non-existent field in a table results in `nil`. Listing 2.5 shows an example of using tables as lists.

```lua
-- Create a table as an array
local array = {"Hello", "world"}
array[3] = "from"
array[4] = "Lua"

-- Concatenate the elements of the table,
-- separated by a space, and append '!'
-- ('..' is the string concatenation operator)
local hello = table.concat(array, " ") .. "!"

-- Print "Hello world from Lua!" to stdout
print(hello)
```

Listing 2.5: Using tables as lists

2.2.4 Functions

In Lua, functions are first-class values. In fact, the function definition statement (`function f() ... end`) is syntax sugar for assigning a function expression (`f = function() ... end`).

Functions can return multiple values at once. For example in the standard library, `coroutine.resume()` (see 2.2.6) returns a success status, followed by the result values. Returning multiple values is done like this: `return 10, 20`. On assignment, any extra values are ignored, e.g. `result = coroutine.resume(co)` ignores anything after the first return value. Extra values are also ignored if the function call is wrapped in parentheses, or when it is followed by another expression. That means `y` will be `nil` here: `x, y = (coroutine.resume(co))` and here: `x, y = coroutine.resume(co), nil`.

Functions are also closures: they have access to the local variables of an enclosing scope. An example of this can be seen in listing 2.6.

Functions can also take a variable number of arguments, using a vararg expression. Adding ... to the end of a function's signature turns it into a vararg function, and you can then use ... anywhere directly inside the function to represent the extra arguments passed to the function. ... is not a first-class citizen, though: it is not possible to use it inside an inner function. To get around that, it is common to put the vararg in a table. A useful function from the standard library is `select(index, ...)`[10], which is a vararg function itself. It has two uses: when `index` is the string `"#"`, it returns the number of arguments. When it is a number, it returns all arguments after that index.

2.2.5 Metatables

A metatable is a normal table assigned to a value as metadata. It can be used to override default behaviour like operators, indexing and calling.

[10] https://www.lua.org/manual/5.4/manual.html#pdf-select

```lua
-- Cherry-pick indices from a table
function pick(tbl, ...)
    local new = {}
    -- Get number of vararg arguments
    local nArgs = select("#", ...)

    -- Loop from 1 to nArgs
    for i = 1, nArgs do
        -- Get vararg number i
        local idx = select(i, ...)
        new[i] = tbl[idx]
    end
    return new
end

-- Pick indices 1, 4 and 2 from 'array'
local array = {"Hello", "world", "from", "Lua"}
local picked = pick(array, 1, 4, 2)

-- "Hello Lua world!"
print(table.concat(picked, " ") .. "!")
```

```lua
function makePicker(tbl)
    return function(...)
        return pick(tbl, ...)
    end
end

local picker = makePicker(array)
picked = picker(1, 4, 2)
-- same as before
```

Listing 2.6: Using vararg functions and closures

Primitive types have a metatable for the entire type, but tables have individual metatables. The metatable set on strings for instance enables shorter OOP-like syntax, `("hello"):upper():reverse()` being shorthand for `string.reverse(string.upper("hello"))`.

In example 2.7a, the metatable containing `__add` and `__tostring` is set as the metatable for the table with `x` and `y`. When we do `vec + vec`, Lua looks in the metatable and executes the `__add` function. Similarly when we print the result, the `__tostring` function is executed. In this way, all operators[11] can be overridden.

Not only operators can be overridden like this. It is also possible to customise what happens when a field is not found, for example doing `tbl.foo` when `tbl` does not contain the key `foo`. When Lua cannot find a key, if the metatable's `__index` is a table, it will look there. This can be used to create a prototype inheritance chain. To facilitate this, Lua has syntax sugar: `myDog:bark()` (notice the colon) is sugar for `myDog.bark(myDog)`, and `function dog:bark()` is sugar for `function dog.bark(self)`. These concepts are demonstrated in listing 2.7b, and this is also how the shorthand string operations mentioned before work.

Another metamethod that can be used is the `__call` metamethod, which is called when trying to call a table. This can be used for instance to create

[11] https://www.lua.org/manual/5.4/manual.html#2.4

[12] example adapted from https://www.quora.com/What-is-the-__call-metamethod-in-Lua-and-what-are-some-of-its-uses-and-basic-examples/answer/Pierre-Chapuis

```lua
local mt = {}
function vector(x, y)
    return setmetatable(
        {x = x, y = y}, mt)
end
function mt.__add(a, b)
    return vector(
        a.x + b.x,
        a.y + b.y)
end
function mt.__tostring(v)
    return "("..v.x..", "..v.y..")"
end

local vec = vector(2, 1)
print(vec + vec) --> (4, 2)
```

(a) Overriding operators

```lua
local animal = {}
animal.sound = "*silence*"
animal.name = "the animal"
function animal:eat()
    print(self.name.." eats")
end
function animal:makeSound()
    print(self.sound)
end

local dog = {}
setmetatable(dog, {__index = animal})
dog.sound = "Woof!"
function dog:bark()
    self:makeSound()
end

local myDog = {name = "Doggo"}
setmetatable(myDog, {__index = dog})
myDog:eat()  --> Doggo eats
myDog:bark() --> Woof!
```

(b) Creating prototype chains

```lua
local fact = { [0] = 1 }
setmetatable(fact, {
    __call = function(t, n)
        -- Calculate and store if it does not exist yet
        if not t[n] then t[n] = n*fact(n-1) end
        return t[n]
    end
})

fact(10) --> 3628800
```

(c) Using the `__call` metamethod for memoization[12]

Listing 2.7: Using metatables

a memoised factorial function, as shown in listing 2.7c.

2.2.6 Coroutines

Lua provides asymmetric stackful coroutines. A coroutine is a first-class value of type `thread`. A coroutine represents a thread of execution in the Lua interpreter, and can be used for concurrency, but not parallelism. Functions for creating a coroutine from a function, resuming a coroutine and yielding from a coroutine are `coroutine.create()`, `coroutine.resume()` and `coroutine.yield()` respectively.

Coroutines in Lua are asymmetric: a coroutine cannot specify where to transfer control to, it always yields back to its caller [9].

Lua's coroutines are stackful, which means that a yield can happen anywhere in the call stack. Something like 2.8 is not possible with for example Python's generators.

```lua
function yieldIncr(value)
    coroutine.yield(value + 1)
end

function coroFunction()
    yieldIncr(41)
end

-- Create a coroutine
local co = coroutine.create(coroFunction)
local success, result = coroutine.resume(co)
print(result) --> 42
```

Listing 2.8: Yielding from deeper into the call stack

2.2.7 Comparison between Clean and Lua for TOP

There are some important differences between Lua and Clean. The most important one is the fact that Clean is a functional language and Lua is procedural. Like in Clean, functions in Lua are first-class. Unlike Clean, they can have side-effects and they can give a different output for equal inputs.

Clean does not have coroutines and uses functions to model tasks. In this thesis we show how we can use coroutines to model tasks (section 3.2).

Another important difference is that Clean is statically typed and compiled while Lua is dynamically typed and interpreted. Type information in Lua, like most dynamically typed languages, is not precise: all tables have type `table`, regardless of their contents. In section 3.1 we explore ways to meaningfully work with this in Lua.

Chapter 3

Research

There are many design decisions of TOP in Lua to be explored, resulting from the major differences between Clean and Lua. This research explores the design space and creates a proof-of-concept of TOP in Lua based on these decisions. The proof-of-concept is complete, when:

- it features a basic implementation of tasks,
- these tasks can be composed sequentially and in parallel (both "and" and "or"), while making use of observable task values.
- it has a way of interacting with users (editors), and some form of user interface that is automatically generated. Minimally, the editors should be able to model tables, strings, numbers and booleans.

The concepts of shared data sources (§2.1.3) and exceptions (§2.1.1) in TOP are out of scope for this bachelor thesis.

In the next section, we think about how to meaningfully work with the dynamic type system of Lua. After that, we look how we can represent tasks (§3.2) and task types (§3.3). Section 3.4 takes a look at task combinators, we discuss user interfaces in section 3.5, and lastly we discuss LTasks (§3.6).

3.1 Task types

Lua is dynamically typed, so a variable or table field can hold any type of value at any point in time. A function can take any number of arguments and return any number of values of any type. Type information is also not attached to variables/fields, but to the values. The consequence of this is that there is no type information present when there are no values yet—a field in a table without any value (i.e. the table key is assigned `nil`) simply does not exist. iTasks uses types to automatically create editor UI and to validate task inputs, but that is not possible in Lua by default. Because of this fundamental difference between Clean and Lua, we need to rethink and redesign the TOP concepts as used in iTasks. Below we explore the design space.

3.1.1 Adding types

The programmer specifies which type a task or field is supposed to have. The type information is given in a format similar to JSON schema[1]. This type information is then attached as meta-information to a task, and checked dynamically. This idea basically comes down to emulating a statically typed language and comes closest to how iTasks handles TOP. Because the editor task has an associated output type, it is possible for the implementation to automatically generate editor UI that is appropriate and specific to the type of value.

This is a weird direction to go into for a dynamically typed language: instead of choosing this method in Lua, it would be a better idea to use a statically typed language because it already has these features built-in. Alternatively, instead of trying to fight the dynamic language, we should embrace it. This is what the other ideas do.

3.1.2 Validator and conversion functions

The idea here is to not specify types, but validate or convert task input. For example a task that expects a number can use the `tonumber()` validator / conversion function and fail if the provided value is not accepted by the validator function.

Since there is no information about what type of data a task expects or needs to output, it is impossible to automatically generate type-specific editors. Instead, the user is in charge of selecting the right editor type. The user interface allows the user to change the input method—for example from text to a list (which can contain items of differing types).

This is a versatile way to enter information, compared to the static types of Clean. For example entering a date can be done in these ways, with each of them having their own function to check if the format is correct and to optionally convert it to a different format:

- Basic text field: `"2022-03-31"`
- Date picker (with a function that outputs the date in one of these other formats)
- Three number fields `year`, `month` and `day`:
  `{year = 2022, month = 3, day = 31}`
- Two number fields `year` and `day`, and a string field `month`:
  `{year = 2022, month = "march", day = 31}`

[1] https://json-schema.org/

This does make use of dynamic typing, but I doubt this is really useful. Especially the usability is a problem since the user has to select the right editor type manually. This problem is even more apparent for composite data structures: imagine as a user having to create an editor for a person with a birthday from scratch:

```
{
    firstName = "John",
    lastName = "Doe",
    birthday = {year = 2022, month = "march", day = 31}
}
```

3.1.3 Interface with JSON APIs

The de-facto communication format of the web, JSON[2], is also dynamic (when not using JSON schema[1]). JSON is a good companion to TOP in the dynamic language Lua, as all concepts in JSON map directly to Lua concepts: numbers, strings, booleans, arrays (tables), objects (tables) and null (nil). It can be used to communicate with all kinds of JSON web APIs, such as ones providing weather conditions, address information or public transit information.

With this approach, tasks do not have any type information attached and can simply fail when their input is not in a format they can work with. They can do this because we can rely on JSON APIs to yield the right format if there was no error.

If we restrict ourselves to only JSON web APIs (which are automatically served by websites), there is no longer an interactive component for users. This is problematic because we just defined (at the start of this chapter) that interactive editors are an essential component of TOP. While JSON can be hand-written by users as input to an editor, doing that is even less user-friendly than 3.1.2.

3.1.4 Structural type matching

A different way to make use of the dynamic-ness of Lua is to attach a type to all task continuations passed to the step combinator. Each continuation can accept a different type, something that is not possible with iTasks. The step combinator can then employ a matching algorithm to find which task it should execute, based on the value of the previous task. The matching algorithm should not only match primitive types, but should also be able to match more complex structures like tables as lists, or tables with specific fields. The major difference with the first option "adding types" (section 3.1.1) and with statically typed languages is that you can add continuations for multiple different types to the step combinator, and that the output type of editors can still change at runtime.

[2] https://www.json.org/json-en.html

There are many different ways to design such a matching algorithm, as there are many design considerations. When multiple task continuations match some value, the algorithm can find either the first match or the best match. Finding the best match requires defining a measure of match quality. What happens to tables that have more fields than the task requires?

You may think that an editor task before a step combinator can use the type information of tasks after the combinator to automatically deduce the right editor type to display. However, this goes against the TOP principle that tasks are autonomous: they do not depend on other tasks. What we can do is manually make a different editor for each type of output.

This is the direction we will go into for this thesis, for the following reasons: it keeps the core concepts of TOP with user interaction, it makes use of the dynamic typing of Lua, it works in a way that is not encouraged in the current TOP implementations and lastly I think it is the most interesting and novel idea.

3.2 Tasks and task values

Tasks in iTasks are modelled as an algebraic data type (listing 2.3). Lua does not have algebraic data types. Moreover, in contrast to Clean, mutation is normal and we can keep state by using tables. We can also use coroutines which makes modelling changing tasks more convenient, as execution can halt in the middle of a function and continue later on. There are three choices to be made here: whether to model the task functionality as a coroutine or as a function, whether to store that coroutine/function in a table or leave it bare, and whether to separate the actual task value from its stability. All three choices affect each other; only a few combinations actually make sense.

3.2.1 Functions or coroutines

Clean has no coroutines. The way that a single task can keep state and handle multiple events during the runtime of the program is by returning a new function to handle the next event. In Lua we can keep handling events within a single coroutine. We can keep state using local variables within the coroutine. If we choose to use functions, we return the task value and stability. When using coroutines, we yield. We will use coroutines for this thesis because they can be used to model tasks in an elegant way, which we show in section 3.6.

3.2.2 Tasks as tables

Close to how iTasks works in Clean, we can model tasks as bare functions or coroutines, where the task value is returned or yielded. Making use of what Lua gives us, we can store that coroutine/function in a table alongside

```lua
function counter(initial)
    local count = initial
    return {
        get = function()
            return count
        end,
        increment = function()
            count = count + 1
        end
    }
end

local c = counter(41)
print(c.get()) --> 41
c.increment()
print(c.get()) --> 42
```

(a) Using a `get()` function and a direct `count` upvalue.

```lua
function counter(initial)
    local self = {}
    self.count = initial
    function self.increment()
        self.count = self.count + 1
    end
    return setmetatable({}, {
        __index = self,
        __newindex = function() end
    })
end

local c = counter(42)
print(c.count) --> 42
c.count = 10
print(c.count) --> 42
getmetatable(c).__index.count = 10
print(c.count) --> 10
```

(b) Using a table with a no-op `__newindex` metamethod. With a detour, the value can still be modified from the outside.

Listing 3.1: Two ways of making values private using closures: `count` cannot be accidentally modified from the outside.

the task value and stability. The effect of this is that all tasks that have a reference to the task can read its value at any time. In iTasks this is limited to tasks that are linked together by a combinator. Another possibility that this opens up is that we can now define other functions that operate on this task's internals, however that goes against the principle that tasks should be autonomous.

The downside of this is that task values can now be altered from outside. TOP means that tasks are autonomous: only the task itself can set its task value, and one task should not be able to modify the value of another task. This can be solved by not exposing the task value itself, but rather a function that reads from a private task value. There are multiple ways to do this, listing 3.1 shows two of them. They both make use of a closure to hide the variable. Barring use of the `debug` library[3], method 3.1a makes the `count` variable truly invisible and immutable from the outside. Method 3.1b allows us to refer to the value itself instead of having to call a getter function which makes it transparent, but its downside is that it only hides the `count` variable behind a metatable. The example shows that it is possible to modify the variable with a detour.

Both of these methods work for preventing accidentally modifying a

[3] The `debug` library violates multiple core assumptions about Lua code [4], so including it in considerations would not be appropriate.

task's value. For the proof of concept however, we will not be using any of these options. While that makes it possible to violate a task's autonomicity, that will not happen in normal use.

3.2.3 Value and stability

When using functions or coroutines as tasks, we can choose to return or yield the task value and its stability separately since Lua allows returning multiple values. Closer to what iTasks does, we could also return a table containing the value and the stability. Returning the task value and its stability separately is more idiomatic in Lua. However, this can lead to problems where the value and stability need to be passed around. Especially for the `parallel` combinator because its task value is a list of task values.

We will keep the actual value and the stability separate and only pack them together when needed. In the proof of concept, this only happens in the `parallel` combinator.

3.3 Type representation

Because we decided in section 3.1.4 that task continuations have an associated type, we need some way to represent Lua types at runtime. This typing information is used by the step combinator's type match function to decide which task continuation it should choose. Lua has the `type` function that returns the type of the value passed as a string: `type(42) == "number"`. The problem is that this does not give us detailed enough information for tables; `type({10, 20})` and `type({hello = "world"})` both result in just `"table"`.

Tasks and editors require a more elaborate system that can distinguish types of composite values. We need to consider the way these types are written, how they are represented or stored at runtime, and how they are compared against each other. We will elaborate on multiple ways to solve the first two considerations now, how to compare types is left for section 3.4.4.

3.3.1 LuaRocks libraries

When looking for Lua libraries, I primarily used LuaRocks[4], which is the most used Lua package manager and package repository. There are a number of libraries that come up when searching for "types". Three of them have some way to represent composite types at runtime: luastruct[5], struct.lua[6] and Typed[7].

[4] `https://luarocks.org/`
[5] `https://luarocks.org/modules/UlisseMini/luastruct`
[6] `https://github.com/mpatraw/struct.lua`
[7] `https://luarocks.org/modules/SovietKitsune/typed`

luastruct and struct.lua

luastruct and struct.lua represent types at runtime by a default value. The example from the LuaRocks description[5] of luastruct describes the type of a table with a `name` field of type `string` (by default `"default name"`) and an `age` field of type `number` (default `0`):

```
local person = struct {
    name = "default name",
    age  = 0
}
```

struct.lua works in the same way, and this example is also valid there. This may be a very simple way to store composite types at runtime, but it has the obvious downside that every field must have a default value. For editors, this is not that big of a problem. But for specifying what type a task accepts, this can be very inconvenient. Furthermore, in this place the actual default value does not have any use: only its type will be used. A bigger problem for representing types of tables in this thesis is that these libraries are only about *structs*; they do not have a way to represent arrays.

Typed

Typed is a library for checking a function's arguments. It gives formatted error messages containing information on what type was expected. The error messages are not interesting for this thesis, but how it represents composite types is. Arrays can be represented like the string `"number[]"`, maps are written as `"table<string, boolean>"`. When multiple types are valid, they can be written as `"string | number"`. For more complicated types like what LuaStruct and Struct.lua do, it uses schemas, for example a table that contains the string field `name` and a numeric field `id` is written like this: `typed.Schema('test'):field('name', 'string'):field('id', 'number')`.

3.3.2 Lua extensions

Maidl, Mascarenhas and Ierusalimschy [8] designed a gradually typed extension of Lua called Typed Lua. It does not keep types at runtime, but it does have its own way of representing these types in code.

Pallene, developed by Gualandi and Ierusalimschy [2], is a typed subset of Lua. In contrast to Typed Lua, it does sometimes keep types for runtime type checks.

Teal[8] is a language that compiles to Lua, implemented in Lua. It has an online playground[9] that shows that types are removed at runtime.

[8]https://github.com/teal-language/tl
[9]https://teal-playground.netlify.app

Unlike Teal, Luau[10] does not compile to Lua but has its own interpreter. Like Luau however, it also does not keep types at runtime.

3.3.3 Other languages

TypeScript

TypeScript[11] is a language that transpiles to JavaScript. Like Typed Lua, Teal and Luau, its types get removed at compile time. We can still learn from the way types are written, though.

3.3.4 Typed library

The Typed library library is the most complete of the three libraries, so we will use it in the proof-of-concept for representing types at runtime. The matching of types will initially also be done by the library, but later on we will design a custom match algorithm. While the library is more complete than the rest, it is still missing some non-essential features we would like to have such as being able to describe a table which both has predetermined fields and is also an array. Implementing these is out of scope for this thesis, but the design decisions themselves will be considered in section 3.4.4.

3.4 Task combinators

3.4.1 Combinators and operators

Combinators are common in functional languages like Clean, where it is possible to define custom operators for them. For instance, iTasks defines an infix operator `>>*` for the `step` function. In order to be able to easily compare the proof of concept to iTasks, we want to come close to the notation as used in iTasks.

Lua does not allow defining custom operators, but you can change the behaviour of the pre-existing operators. To do this, we define a `task` table and use it not only to define all combinators, but also as a metatable for tasks. For changing the behaviour of, for example, the `&` function, we define the `__band` metamethod in this table. We let all tasks inherit from this prototype table using the `__index` metatable entry, see listing 3.2.

3.4.2 Parallel

If we bring the `parallel` signature from iTasks down to its essence, we get listing 3.3. It takes a list of tasks, the task it returns has as its value a list of values of the original tasks.

[10] https://luau-lang.org/
[11] https://www.typescriptlang.org/

```lua
local task = {}
task.__index = task
task.__band = function() --[[ ... ]] end

local myTask = setmetatable({}, task)
```

Listing 3.2: A simplified example showing the basic structure for inheriting the prototype and defining custom operator behaviour.

```
parallel :: [Task a] -> Task [TaskValue a]
```

Listing 3.3: The simplified `parallel` combinator's signature.

Each time the `parallel` task is resumed (we decided in section 3.2.1 that it is a coroutine), it resumes the input tasks one by one and updates its list of task values. Because the resulting task needs to also contain the task values' stability, the value of the `parallel` task is a list of task value–stability pairs. Listing 3.4 shows a very simple example of parallel "and."

```
(return "A" -&&- return "B") >>- (\x -> viewInformation [] x)
```

Listing 3.4: A simple example of using `parallel`. This shows "A" and "B" in the output.

3.4.3 Step

The step combinator executes one task and chooses another task to execute using the observable task value of the first task. The result of the step combinator is a task that has the value of the selected follow-up task. It is called *step* because when it can execute one of the follow-up tasks, it steps to that task and does not go back anymore.

In iTasks, the step combinator expects a list of *task continuations*. Such a continuation defines a task that should be executed when some event happens. Such an event can be when a task has a stable value or when a task has a value that matches some predicate (`OnValue`). It can also be when the user presses some button like 'yes', 'no', 'ok' or 'cancel' (`OnAction`). Listing 3.6 shows an example of using multiple `OnValue` continuations. The step combinator only steps to a continuation if its predicate holds. If we simplify its signature from iTasks, we get listing 3.5.

Each time the `step` task is resumed before stepping, it resumes the first task and tries to find a matching continuation task. When one such continuation task is found, it steps. Now, the `step` task acts as a proxy to the continuation task: it resumes the continuation task and updates its own task value and stability to match that task.

```
step :: (Task a) [TaskCont a (Task b)] -> Task b

:: TaskCont a b
    = OnValue ((TaskValue a) -> ? b)
    | OnAction String ((TaskValue a) -> ? b)
```

Listing 3.5: The simplified `step` combinator's signature, together with the type definition of `TaskCont` (also simplified).

```
enterInformation [] >>* [
    OnValue (ifValue isPalindrome (showInput "palindrome: ")),
    OnValue (ifValue isGreeting (showInput "greeting: "))]
```

(a) Using the step combinator with `OnValue` in iTasks. It will automatically step once the user input is either a palindrome or a greeting. `isPalindrome` and `isGreeting` are defined elsewhere, their implementation is not important. (A greeting is something like "hello" or "I am ...")

```
enterInformation [] >>* [
    OnAction (Action "Check palindrome")
        (ifValue isPalindrome (showInput "palindrome: ")),
    OnAction (Action "Check greeting")
        (ifValue isGreeting (showInput "greeting: "))]
```

(b) The same example as (a), but with `OnAction`: it will only step when the user clicks "Check palindrome" or "Check greeting."

Listing 3.6: `OnValue` and `OnAction` in iTasks. `showInput` is a convenience wrapper around the iTasks function `viewInformation`.

3.4.4 Type matching

The type of values that a continuation expects will need to be attached to the continuation, in the format just described in section 3.3. To decide what continuation to step to in Lua, we use a type matching function. As hinted at in section 3.1.4, there are many different ways for a type matching function to work. The considerations as well as the choices for this proof of concept and the reasoning behind the choices are outlined here. The syntax used here is hypothetical.

Best match or first match

When there are multiple continuations that match the current task value, we need to decide which of the continuations to execute. This possibility of having multiple continuations that match is also present in iTasks, where the first `OnValue` or otherwise the first `OnAction` match is used. Actually, in iTasks all continuations need to accept exactly the same type so it is not possible to let the system automatically find a "best" match, only manually. This is easier to do in dynamically typed languages like Lua.

We can define a *better* match to be a more *specific* one: `number` is more specific than string | number (a union), because the first one does not accept strings. table<string, number> (a table with string keys and number values) is more specific than just table, and a table {id: number, age: number} (a struct) is even more specific than both of these.

We can formalise this intuitive relation, let's write $T_1 < T_2$ if T_2 is more specific than T_1. To be able to use this relation in Lua with the table.sort function, it needs to be a strict partial order [4, §6.6]: it must be irreflexive, asymmetric and transitive. If some T_1 and T_2 do not match any of the following rules, they are either not comparable or equivalent. T denotes any type, t is any type except unions, F and G are pairs of key name and value type, and k is a string key. $T \mid T$ (same type on left and right side) is equal to just T. Order does not matter for union types: $T_1 \mid T_2$ is equal to $T_2 \mid T_1$. A struct with no pairs is equal to a table. Note that relation defined here is intended to be simple, so it does not include things like tuple types or a specified list length.

The any type is the least specific because it matches all types:

$$\text{any} < T \qquad \text{if } T \neq \text{any}$$

A union of two types is less specific than a single type:

$$T_1 \mid T_2 < t_3$$

For two unions with a corresponding type, one is less specific than the other if the non-corresponding type is less specific:

$$T_1 \mid T_2 < T_1 \mid T_3 \qquad \text{if } T_2 < T_3$$

A table of any type is less specific than one with a list type specified:

$$\text{table} < \text{table}(T)$$

The same for a table that has a key and value type specified:

$$\text{table} < \text{table}(T_1, T_2)$$

A list is less specific than another list if their element types are less specific:

$$\text{table}(T_1) < \text{table}(T_2) \qquad \text{if } T_1 < T_2$$

A table with string keys and a set value type is less specific than a struct type (given that the struct type is not empty):

$$\text{table}(\text{string}, T) < \{F_1, \ldots, F_n\}$$

For two struct types with a corresponding pair of key and value-type, one is less specific than the other if the rest of the struct types is less specific:

$$\{F_1, \ldots, F_n, k : T\} < \{G_1, \ldots, G_m, k : T\} \qquad \text{if } \{F_1, \ldots, F_n\} < \{G_1, \ldots, G_m\}$$

For two struct types with the same number of pairs and a corresponding key, one is less specific than the other if the value-type is less specific and the rest of the struct is less specific:

$$\{F_1, \ldots, F_n, k : T_1\} < \{G_1, \ldots, G_n, k : T_2\} \quad \text{if } T_1 < T_2 \text{ and } \{F_1, \ldots, F_n\} < \{G_1, \ldots, G_n\}$$

Matching lists: types and order

A list in Lua can contain values of differing types at once. What happens if the actual list contains the right types but in a different order than asked for? This goes wrong if the position of elements in the list has meaning. Typescript calls this tuple types[12]. An example of this is a continuation accepting a date as a table of `{number, string, number}` (year, month, day). When it receives a `{number, number, string}` instead, it can not know which number is the day and which is the year. Therefore, a list with a different order of types should never match.

Matching lists: length

If the continuation specifies a list length and if the actual list is longer than this length, does it still match? List elements may have semantics, so if we choose to match a list that is longer than needed, we may discard important information. This can happen for example when we have a 3D vector that is represented as a list of its coordinates. If we have two continuations, one for 2D vectors and one for 3D vectors, we should not choose the 2D vector continuation. To prevent situations like this, we should not match lists that are longer than requested. The best-match algorithm described above does not include list length, so using that does not help.

Matching tables

Analogous to list length: when a table has more fields than required, does it match? The same 2D/3D vector example applies here, but with tables containing the fields `x`, `y` and `z`. This problem can be solved in two ways: by using the best-match algorithm described above or by manually ordering the continuations, placing the continuation accepting a table with the fewest number of fields last.

[12] `https://www.typescriptlang.org/docs/handbook/2/objects.html#tuple-types`

3.5 Editors and user interface

There are many different ways of interfacing with users. iTasks uses a webpage for instance. But there are other graphical interfaces, as well as non-graphical ones. They all differ in usability for the user and ease of programming. We explored a JSON-based interface in section 3.1.3, which would be an especially non-user-friendly user interface.

3.5.1 HTML page

There is one well-known Lua library for and for interacting with the DOM through Javascript: Fengari[13]. Fengari implements a Lua VM, so Lua code runs in the browser. We can also generate HTML using h5tk[14] and serve it using LuaSocket[15], http[16], Fullmoon[17], Lapis[18], Lor[19], Sailor[20] or Pegasus[21].

We will not be going this way, because while it may be the most user-friendly option and cross-platform, we estimate that the amount of work exceeds the scope of this proof-of-concept project and other options are usable enough for a proof of concept.

3.5.2 Native application

A native application looks about the same as a HTML page, but the difference is that interaction does not go via Javascript but via an API written in C. There are some native UI libraries for Lua: fltk4lua[22], TekUI[23], AbsTK[24], libuilua[25], lui[26], lui[27] and wxLua[28] to name a few.

A native application has about the same usability as a webpage. Due to the fact that Lua is built to interoperate with C, it is easier to build a native application than a webpage. For this proof of concept, though, we will use a simpler form of user interface.

[13] `https://fengari.io/`
[14] `https://luarocks.org/modules/forflo/h5tk`
[15] `https://luarocks.org/modules/lunarmodules/luasocket`
[16] `https://luarocks.org/modules/daurnimator/http`
[17] `https://github.com/pkulchenko/fullmoon`
[18] `https://luarocks.org/modules/leafo/lapis`
[19] `https://luarocks.org/modules/sumory/lor`
[20] `https://github.com/sailorproject/sailor`
[21] `https://luarocks.org/modules/evandrolg/pegasus`
[22] `https://luarocks.org/modules/siffiejoe/fltk4lua`
[23] `https://luarocks.org/modules/luarocks/tekui`
[24] `https://luarocks.org/modules/pedroalvesv/abstk`
[25] `https://luarocks.org/modules/daurnimator/libuilua`
[26] `https://tset.de/lui/index.html`
[27] `https://github.com/zhaozg/lui`
[28] `https://github.com/pkulchenko/wxlua`

3.5.3 Terminal text-based UI

The third way of displaying tasks somewhat graphically is by using a terminal emulator. There are a couple libraries for this: AbsTK[24], ltui[29], termfx[30], lua-tui[31]. The first three are more complete UI-building libraries while lua-tui is more of a toolbox. AbsTK is not available for windows but I also have not been able to get it installed in Ubuntu on WSL. Termfx uses the no-longer-maintained termbox which needs Python 2, and I have not gotten that to work either.

For this thesis I chose to work with ltui. It is quite hard to start working with it because it has almost no documentation, but it does have the features needed for displaying tasks and editors. The way in which its example applications are structured is that there is one element of each type: one main dialog, one text input dialog, one output dialog, and so on. When one of these elements is needed, any old contents get replaced and it gets shown on screen.

3.5.4 Terminal command-line

Since a command-line application does not have a graphical interface and is closer to the implementation, this is the least involved way of interfacing with the user. The user can only type commands and the application responds. This however does make it the least user-friendly, but for a minimal proof of concept this matters less. Since it only involves text input and output, it requires no libraries. Because it requires only the minimal extra setup and effort, this is the initial interface of the proof of concept. Some features are too advanced for such a simple interface, they will only be implemented in a text-based UI.

3.5.5 Tables in editors

Tables can be visually represented as a sequence of key-value pairs, with a "+" button for adding a new pair and a "−" button for removing a pair. A value without key acts as an array entry. These entries implicitly get a numeric key, just like in Lua. They can be displayed one after the other, without keys displayed. Tables that contain tables can be represented in two ways: either by a single element that, when clicked, navigates to the inner table entirely (like entering a directory in a file explorer), or by a collapsible indented list (like the sidebar in a file explorer). They both provide the same functionality; which one to choose comes down to preference or implementation details.

[29] `https://luarocks.org/modules/waruqi/ltui`
[30] `https://luarocks.org/modules/gunnar_z/termfx`
[31] `https://github.com/daurnimator/lua-tui`

3.6 LTasks

To continue the naming scheme of iTasks and mTasks, the proof of concept implementation in this thesis is called LTasks[32]. This section goes into the details of the LTasks library and shows that it is indeed a correct implementation of TOP. To further show that the proof of concept is indeed complete for TOP, chapter 4 contains a case study comparison of the breakfast example, while appendix A contains full examples in both LTasks and iTasks. The full code is available at `https://github.com/Dantevg/LTasks`, the files `task.lua`, `types.lua` and `ltuiEditor.lua` are attached in appendix B.

3.6.1 Tasks

Most functions in the LTask library are defined in the `task` module (B.1), as functions on the `task` table. The function `task.new` (listing 3.7) creates a task, which is a table containing the task coroutine, the task value and its stability. The metatable of the task has a `__index` field pointing to the `task` table, so all operations can be done as methods on a task, and chained. The metatable also defines the custom operator behaviour (3.6.2, listing 3.9).

The function `task.resume` (listing 3.8) is for resuming a task's coroutine. When calling this function, you can give it a table of options used for the user interface: the boolean `showUI` and the task `parent`. The options will be explained further in section 3.6.5. When a task is resumed, it can resume any child tasks it has. When it is done, it yields. This creates a coroutine hierarchy.

```lua
function task.new(fn, name, value)
    local self = {}
    self.stable = false
    self.value = value
    self.__name = name or ""
    self.co = coroutine.create(fn)
    return setmetatable(self, task)
end
```

Listing 3.7: The `task.new` function.

3.6.2 Task combinators

While iTasks provides a lot of combinators, we do not need that for a proof of concept so LTasks includes only the essential combinators and some convenience wrappers around them. Here is the list, along with their operators in LTasks or their equivalent in iTasks:

[32] With capital "L" to avoid confusion with iTasks, because many fonts make the lowercase "l" look like a capital "I".

```lua
---Resumes the coroutine of the task with the given options
---@param options table
---@param showUI boolean? if set, sets `options.showUI` to this value
---@param parent table? if set, sets `options.parent` to this value
---@return any value
---@return boolean stability
function task:resume(options, showUI, parent)
    if self.stable then return self.value end
    if coroutine.status(self.co) == "dead" then return end

    options = options or {}
    if showUI ~= nil then options.showUI = showUI end
    if parent ~= nil then options.parent = parent end
    local success, err = coroutine.resume(self.co, self, options)

    if not success then error(err) end
    return self.value, self.stable
end
```

Listing 3.8: The `task.resume` function for resuming a task's coroutine.

```lua
task.__band = task.parallelAnd
task.__bor = task.parallelOr
task.__bxor = task.step
task.__concat = task.step
```

Listing 3.9: Setting the custom operator behaviour to functions defined in the `task` table.

- `constant` (`return` in iTasks)
- `step` (`~` in LTasks, `>>*` in iTasks), `stepStable` (`>>-` in iTasks) and `stepButtonStable` (`>>?` in iTasks)
- `parallel`, `anyTask`, `parallelAnd` (`&` in LTasks, `-&&-` in iTasks), `parallelOr` (`|` in LTasks, `-||-` in iTasks), `parallelLeft` (`-||` in iTasks) and `parallelRight` (`||-` in iTasks)
- `transform` and `transformValue` (`@` in iTasks)

These are the most important functions for building a TOP system, as we defined at the start of this chapter.

Step

The step combinator in LTasks implements both `OnValue` and `OnAction`. When there are multiple continuations that match some value and action, the UI shows the user a dialog to choose one of the continuations to step to. iTasks does not handle this well, due to what is probably a bug: it displays two buttons with the same name, but chooses the first task continuation

```
editor.editString("") ~ {
    {
        action = "continue",
        fn = function(value)
            return isPalindrome(value)
                and editor.viewInformation(value, "palindrome: ")
        end
    }, {
        action = "continue",
        fn = function(value)
            return isGreeting(value)
                and editor.viewInformation(value, "greeting: ")
        end
    }
}
```

Listing 3.10: The step combinator in LTasks, using multiple `OnAction` continuations of the same action. Listing 3.6b shows this example in iTasks.

regardless of which button is pressed. For this reason we used two different actions in listing 3.6b.

In the example in listing 3.10, this happens when the user inputs "Madam, I'm Adam"—which is both a palindrome and a greeting. LTasks will prompt the user which continuation to step to.

The implementation of `step` can be divided into three phases: before the step happens, choosing the continuation task, and after the step happens. Before the step happens, each time the `step` task is resumed, it resumes its first task and searches for a matching continuation task (listing 3.11). For that matching, it uses the function `matchContinuation`, which finds all continuations that have the right type and action, and have a type that is as specific as the most specific continuation. If it has found at least one continuation, it goes on to the continuation choosing phase. If there are multiple continuation tasks that match, it lets the user choose which one to step to (listing 3.12). When it has a single continuation task, it steps to that task and acts like a proxy: it sets its own value and stability to that of the continuation task (listing 3.13).

Parallel

The parallel task combinator in LTasks is a simplified version of the one in iTasks, but the most common usage is present: combining tasks into a list of task values. Listing 3.14 shows a simple example of this.

Listing 3.15 shows the most important part of the `task.parallel` function. Each time it is resumed, it resumes all of its child tasks and updates its own value to be the list of values and stabilities of the child tasks. It is itself stable if all child tasks are stable.

```
local matching = {}
self.parent = options.parent
while #matching == 0 do
    self.__name = "step (left, "..t.__name..")"
    if options.showUI then ltuiElements.stepDialog(self, conts, t) end
    t:resume(options, false, self)
    if t.value ~= nil then
        matching = matchContinuation(t.value, t.stable, options.action,
        ↪   conts)
    end
    if t.stable then break end
    if #matching == 0 then self, options = coroutine.yield() end
end

if #matching == 0 then error("no matching continuation for stable task") end
```

Listing 3.11: The first phase of the `task.step` function, before the step happens.

```
-- Step happens here
local next, nextNames = nil, {}
if #matching > 1 then
    -- Allow user to choose continuation
    for _, nextTask in ipairs(matching) do table.insert(nextNames,
    ↪   nextTask.__name) end
    app.main:insert(
        ltuiElements.choiceEditor(nextNames[1], nextNames,
            function(_, idx) return matching[idx] end, nil,
            function(val) next = val end),
        {centerx = true, centery = true}
    )
    while not next do self, options = coroutine.yield() end
else
    next = matching[1]
end
```

Listing 3.12: The continuation selection phase of the `task.step` function, asking the user which continuation task to step to.

```
options.showUI = true -- Show self to reflect stepped task
next:show(self) -- Automatically show continuation

while not self.stable do
    self.__name = "step (right, "..next.__name..")"
    if options.showUI then ltuiElements.stepDialog(self, {}, next) end
    next:resume(options, false, self)
    self.value, self.stable = next.value, next.stable
    self, options = coroutine.yield()
end
```

Listing 3.13: The last phase of the `task.step` function, after the step happens.

```
(task.constant "A" & task.constant "B") ~ {{
    fn = function(x) return editor.viewInformation(x) end
}}
```

Listing 3.14: The parallel combinator in LTasks. The output of this is `{"A", "B"}`. Listing 3.3 shows this example in iTasks.

```
self.parent = options.parent
self.value = {}
while not self.stable do
    self.__name = "parallel ("..table.concat(getTaskNames(), ", ")..")"
    if options.showUI then ltuiElements.parallelDialog(self, tasks) end
    for i, t in ipairs(tasks) do
        if not t.stable then
            t:resume(options, false, self)
            self.value[i] = {value = t.value, stable = t.stable}
        end
    end
    self.stable = allStable(tasks)
    self, options = coroutine.yield()
end
```

Listing 3.15: The content of the `task.parallel` task.

Custom operators

In Clean it is common to define lots of operators. For example, there are eight different operators for variations of the step combinator. Lua does allow for changing the behaviour of the standard operators, but only up to a point. For example, the result of the comparison operators like `<` is always converted to a boolean [4]. Perhaps the most notable library that uses operators with custom behaviour is LPeg[33]. It is not so common to redefine the behaviour of the operators in Lua, so LTasks only uses three operators: `~`, `&` and `|`. Using the `..` operator for `step` more resembles the original meaning—concatenation, putting strings after each other. However that operator does not play well with chaining multiple operators because it is right-associative [4, §3.4.8].

3.6.3 Type matching

For time reasons, we did not implement a custom type matching algorithm (one that matches the empty table `{}` for the type `"table"` for example). We just used the Typed[7] library to compare types, which is very strict in what it matches. There is a Lua library for matching data structures called Tamale[34], however it is not made for matching types and is also strict in what it matches, so we do not use it.

[33] http://www.inf.puc-rio.br/~roberto/lpeg/
[34] https://luarocks.org/modules/luarocks/tamale

We did implement the type *specificity* algorithm for ordering types from section 3.4.4. The function `types.lt` in module `types.lua` (B.2, lines 63–136) follows these rules, and returns `true` when a type is less specific than another. It parses the types the same way as the Typed library[7]. This is important, because Typed itself is first used to check whether a type is even compatible. We do not check for list length, because the Typed library does not allow specifying that. This type specificity algorithm can be seen in action for example when there are two continuation tasks, one of which accepts `"string | number"` and the other accepts `"string"`. When a string value is given, the second continuation task is chosen, even though the first one also matches.

3.6.4 Editors

Instead of providing a single function for creating all types of user input editors like iTasks does, we provide one function per editor type: `editNumber`, `editTable` etc. To create a table editor, the programmer has to provide the table editor with the sub-editors. Listing 3.16 shows what this looks like.

Each editor construction function has an optional parameter for setting the editor's prompt (called a hint in iTasks). This is done to keep the proof-of-concept simple: iTasks uses a `tune` combinator (with `<<@` operator) which can do a lot more, but that is not important for TOP.

```lua
local dateEditor = editor.editTable {
    year = editor.editNumber(),
    month = editor.editString(),
    day = editor.editNumber(),
}
```

Listing 3.16: Creating a table editor with three sub-editors for `year`, `month` and `day` (adapted from the date example in appendix A.2).

3.6.5 User Interface with LTUI

For simplicity with working with LTUI, we decided to only ever have one UI element of a type at once. Instead of creating a new element every time, the old one is re-used, displayed, and hidden when no longer needed. These re-used elements are defined and created once in `ltuiApp.lua`. The module `ltuiElements.lua` provides functions that use these reusable elements and set the contents like the task name or the current value. `ltuiEditor.lua` is the module that then converts these editors into tasks so they can be used with TOP. This module provides the same functions with the same parameters as `terminalEditor.lua`, which provides editors that use standard I/O as a command-line interface instead of a textual UI. Figure 4.5 shows the textual user interface in action.

```
function app:on_refresh()
    if self.task then self.task:resume() end

    ltui.application.on_refresh(self)
end
```

Listing 3.17: The `app.on_refresh` function.

When resuming a task's coroutine, you can pass it options related to the user interface. The boolean `showUI` is read by tasks that have a visible UI (so editors and `step`, but not `transform`). When resuming any child tasks, they set this to `false`. The task `parent` is passed by tasks that have a visible UI to their child tasks. When a task exits its UI (by selecting "back" or when an editor dialog closes), it resumes its parent task with `showUI` enabled, in order for it to update its visible content.

To create a TOP application with a LTUI user interface, `ltuiApp.lua` first creates a `ltui.application`. The main entry point for the application, `ltuiTest.lua`, expands on this by defining `app.init` and `app.on_refresh` functions. LTUI calls the `app.on_refresh` function multiple times per second, which first resumes the top-level task, and then lets LTUI handle any events and draw the UI.

3.7 Wrap-up

We explored a number of design decisions in this chapter, and decided on them for the LTasks implementation. We defined that the proof-of-concept needs to have a basic implementation of tasks that can be composed sequentially and in parallel, and it needs to have editors with a UI (start of the chapter). For handling typed tasks and editors, we decided to implement structural type matching (section 3.1). We make use of coroutines by modelling the task as a table containing a coroutine, the task value and its stability (section 3.2). The types are represented at runtime using the Typed library (section 3.3). We formalised a type specificity algorithm in section 3.4.4. We noted what happens when lists or tables have more elements than required, and when lists have types in a different order than required. For editors, we use a textual UI instead of a HTML page, a native application or a command-line application (section 3.5). In the previous section (3.6) we show how the LTasks implementation works.

Chapter 4

Comparison

This chapter will compare iTasks and LTasks using a case study of the breakfast example adapted from Naus [10] in listing 2.2. To make it into a functioning example with editors, we need to modify it a bit. `makeTea`, `makeCoffee` and `makeSandwich` are here modelled as editors. In the real world, they will be tasks that have no value initially, and a constant value once the tea, coffee or sandwich has been made. This allows us to make use of the standard task combinators without helper functions, like the example in listing 2.2. This complete but contrived example is however more interesting because it makes use of editors and the transform combinator. The entire example can be seen in appendix A.1.

The high level overview looks like listing 4.1. As you can see, they are almost identical. Lua uses different operators, and instead of an `OnValue` there is a table with a `fn` field.

```
((makeTea -||- makeCoffee) -&&- makeSandwich) >>* [OnValue maybeEatBreakfast]
```

(a) In iTasks

```
((makeTea | makeCoffee) & makeSandwich) ~ {{fn = maybeEatBreakfast}}
```

(b) In LTasks

Listing 4.1: The main part of the breakfast example.

4.1 Editors

There is a bit more difference in making editors than in the basic combinators. iTasks uses combinators to add hints to editors, while LTasks includes the hints in the function signature, for simplicity. The most important difference in this example comes from the fact that Clean is statically typed, so the transformation function cannot return different types as in Lua. Instead, it has to return a maybe (an algebraic data type), and we need to use tvFromMaybe, which takes a TaskValue of a ?None or ?Just and turns it into NoValue or Value, respectively. This is not needed in LTasks, where we can simply return nil from the transformation function.

```
makeTea = updateInformation [] False <<@ Hint "Make tea?"
        @ (\x -> if x (?Just "Tea") ?None)
        @? tvFromMaybe
```

(a) In iTasks

```
local makeTea = editor.editBoolean(false, "make tea?")
    :transformValue(function(x) return x and "Tea" or nil end)
```

(b) In LTasks

Listing 4.2: Making a boolean editor that results in either “Tea” or nothing.

4.2 Helper function

Listing 4.3 shows the helper functions that are needed in order to create a viewInformation task only if both a food and a drink are chosen. These functions are written differently because in iTasks we use the maybe type and pattern matching, while we use nil in LTasks.

```
maybeEatBreakfast (Value (drink, food) _) = ?Just (eatBreakfast drink food)
maybeEatBreakfast _ = ?None
```

(a) In iTasks

```
local function maybeEatBreakfast(value)
    if value[1] ~= nil and value[2] ~= nil then
        return eatBreakfast(value[1], value[2])
    end
end
```

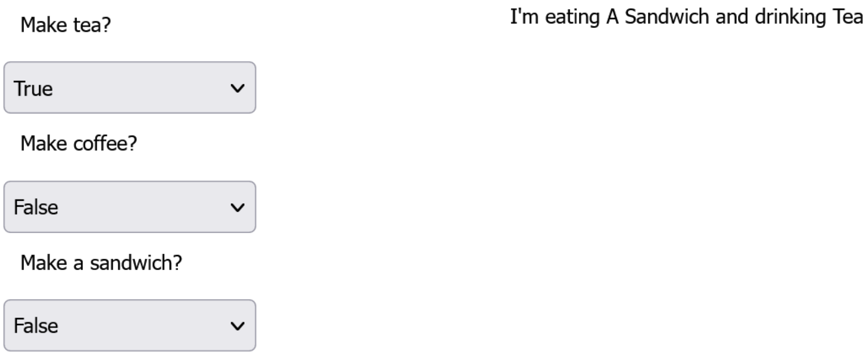
(b) In LTasks

Listing 4.3: Making a boolean editor that results in either ”Tea” or nothing.

4.3 User interface

The user interface for LTasks (shown in figure 4.5) is made to serve two purposes: to be the bare minimum for a TOP proof-of-concept, and to make clear that the behaviour is correct for TOP. For that reason, it looks very different to the iTasks UI.

Let us set aside the differences in visual display for now (LTasks uses a textual UI while iTasks uses a webpage). The textual UI of LTasks shows the structure of the task at the top. This is not only useful for seeing



(a) The UI showing all input fields at once.

(b) The UI showing the output after selecting `true` for `makeSandwich`.

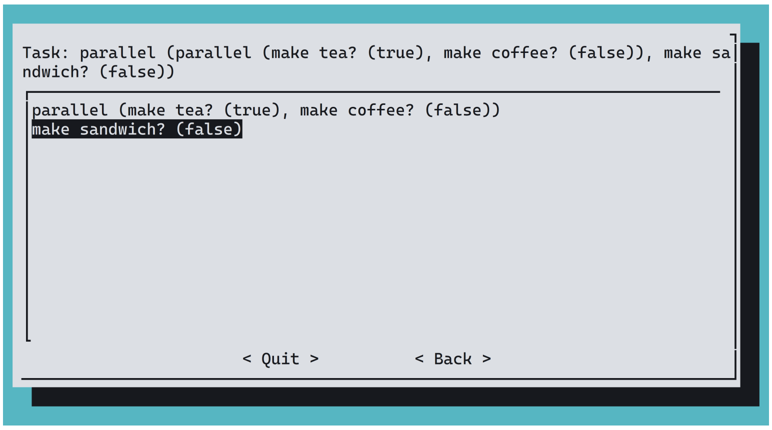
Listing 4.4: The graphical web UI of iTasks.

that the structure is indeed correct, but also for keeping a mental image of where you are navigated to. This is not necessary in iTasks because it shows everything at once instead of entering sub-menus (fig. 4.4). iTasks hides the way in which `makeTea` and `makeCoffee` are composed with `makeSandwich`, while the LTasks UI makes this more explicit.

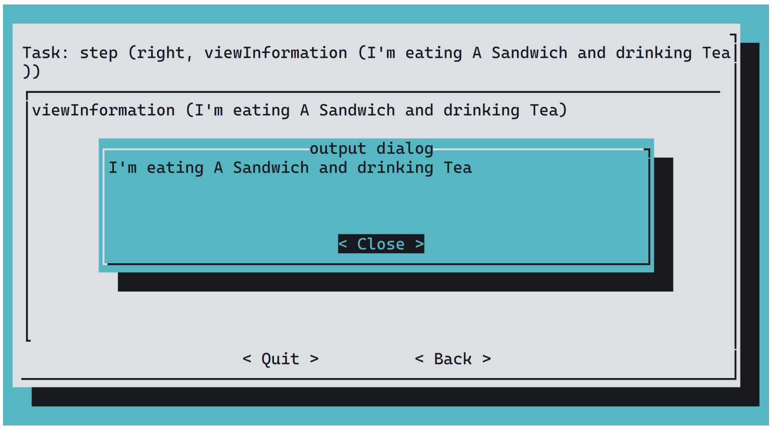
4.4 When to use which

In general, LTasks is more suited for problems that have a large dynamic aspect, and for quick prototyping. Such a dynamic problem can be allowing users to enter a date in multiple formats. It is useful for quick prototyping, because you do not need to first define the types and derive the right classes, as you would in iTasks (see the date example in A.2.2).

Bringing TOP to Lua is not all sunshine and roses, though. The small standard library of Lua means that you need to define some functions yourself, while they are provided in Clean. In the date example (A.2), Clean has the `elemIndex` function, which needs to be defined manually with LTasks. We disregard the `parseDate` function added by iTasks here, because LTasks is only a proof of concept. Lastly, while Lua is more free in what you can do, Clean—being statically typed—provides some static guarantees, which is important in some situations.



(a) The UI showing that **makeTea** is `true` and that **makeCoffee** and **makeSandwich** (selected) are both `false`.



(b) The UI showing the output after selecting `true` for **makeSandwich**.

Listing 4.5: The textual UI of LTasks.

Chapter 5

Related Work

There are currently two implementations of TOP. The iTask system [11] is a TOP implementation for creating distributed multi-user systems. The mTask system [6, 7] is not an interactive one like iTasks, but it is meant for IoT devices, which are constrained in their resource usage.

Naus describes TopHat [10], a TOP language with formal semantics. They include a clear description of the core TOP features.

There has been some research on adding a type system to Lua. Gualandi and Ierusalimschy introduce a typed companion language to Lua called Pallene [2], and Maidl, Mascarenhas and Ierusalimschy have developed Typed Lua [8], which is an optionally typed language.

Chapter 6

Conclusions

In this bachelor thesis we explored the design decisions that come up when implementing TOP in the procedural language Lua, and we have written a proof-of-concept TOP implementation called LTasks. LTasks contains the most important parts for a TOP implementation: it has tasks, these tasks can be composed sequentially and in parallel, and there is interaction with users through editor tasks.

The first major difference between Clean and Lua is that Clean is statically typed while Lua is dynamically typed. We explored a number of ways to work with Lua's dynamic types. Structural type matching is the most interesting choice here, which we use in LTasks. Lua has coroutines while Clean does not. Coroutines are more convenient than functions for modelling tasks, so a task in LTasks is a table with a coroutine. The most important choice for structural type matching is whether to choose the first match or the best match. We defined an algorithm to find that best match, which we use in LTasks. User interaction through editors in LTasks happens in a text-based user interface, because that is the simplest form of UI that can display all TOP features.

6.1 Future work

The proof-of-concept implementation uses the Typed library to represent types at runtime. However, this library can only represent a limited set of data structures and is very strict in what it matches. Further research could find a better way of representing types at runtime so that more Lua features can be used, developing a less strict type matching algorithm and a more complete type specificity relation to go along with it. One can look at Typed Lua [8] or Pallene [2] as inspiration for the types.

This research focused only on the core concepts of TOP, and left shared data sources and exceptions out of scope. Further research can go to expanding the LTask implementation by bringing SDS and exceptions to Lua.

Acknowledgements

Many thanks go out to my supervisor, Peter Achten. Not only for guiding me throughout the process and providing continuous valuable feedback on my writing, but also for helping me understand TOP and helping me fix errors in my Clean code. I also want to thank my parents and my friends for their support.

Bibliography

[1] Peter Achten, Pieter Koopman, and Rinus Plasmeijer. An introduction to task oriented programming. In *Central European Functional Programming School*, pages 187–245. Springer, 2013.

[2] Hugo Musso Gualandi and Roberto Ierusalimschy. Pallene: a companion language for lua. *Science of Computer Programming*, 189:102393, 2020.

[3] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[4] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. *Lua 5.3 Reference Manual*, 2022. `https://www.lua.org/manual/5.3/manual.html`.

[5] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua: about, 2022. `http://www.lua.org/about.html`.

[6] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. A task-based dsl for microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–11, 2018.

[7] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. Multitasking on microcontrollers using task oriented programming. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1587–1592, 2019.

[8] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed lua: An optional type system for lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–10, 2014.

[9] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[10] Nico Naus. *Assisting End Users in Workflow Systems*. PhD thesis, University Utrecht, 2020.

[11] 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.

[12] 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, 2012.

Appendix A

Examples in LTasks and iTasks

This appendix contains the full code of examples using both iTasks and LTasks, in order for comparing them. A.1 shows the full code for the breakfast example used throughout this thesis. A.2 shows an additional example of letting the user input multiple date formats.

A.1 Breakfast

A.1.1 In LTasks

```
local task = require "LTask.task"
local editor = require "LTask.ltuiEditor"

local makeTea = editor.editBoolean(false, "make tea?")
    :transformValue(function(x) return x and "Tea" or nil end)
local makeCoffee = editor.editBoolean(false, "make coffee?")
    :transformValue(function(x) return x and "Coffee" or nil end)
local makeSandwich = editor.editBoolean(false, "make sandwich?")
    :transformValue(function(x) return x and "A Sandwich" or nil end)
local eatBreakfast = function(drink, food)
    return editor.viewInformation("I'm eating "..food.." and drinking
    ↪  "..drink)
end

local function maybeEatBreakfast(value)
    if value[1] ~= nil and value[2] ~= nil then
        return eatBreakfast(value[1], value[2])
    end
end

return ((makeTea | makeCoffee) & makeSandwich) .. {{fn = maybeEatBreakfast}}
```

A.1.2 In iTasks

```
module Breakfast

import StdEnv
import iTasks

Start :: *World -> *World
Start world = doTasks breakfast world

makeTea :: Task String
makeTea = updateInformation [] False <<@ Hint "Make tea?"
        @ (\x -> if x (?Just "Tea") ?None)
        @? tvFromMaybe

makeCoffee :: Task String
makeCoffee = updateInformation [] False <<@ Hint "Make coffee?"
        @ (\x -> if x (?Just "Coffee") ?None)
        @? tvFromMaybe

makeSandwich :: Task String
makeSandwich = updateInformation [] False <<@ Hint "Make a sandwich?"
        @ (\x -> if x (?Just "A Sandwich") ?None)
        @? tvFromMaybe

eatBreakfast :: String String -> Task String
eatBreakfast drink food = viewInformation []
        ("I'm eating "+++food+++" and drinking "+++drink)

maybeEatBreakfast :: (TaskValue (String, String)) -> ? (Task String)
maybeEatBreakfast (Value (drink, food) _) = ?Just (eatBreakfast drink food)
maybeEatBreakfast _ = ?None

breakfast :: Task String
breakfast = ((makeTea -||- makeCoffee) -&&- makeSandwich)
        >>* [OnValue maybeEatBreakfast]
```

A.2 Multiple date input formats

A.2.1 In LTasks

```
local task = require "LTask.task"
local editor = require "LTask.ltuiEditor"
local typed = require "typed"
local app = require "LTask.ltuiApp"

local months = {"jan", "feb", "mar", "apr", "may", "jun", "jul", "aug",
↪  "sep", "oct", "nov", "dec"}

local validMonths = {}
for i, m in ipairs(months) do validMonths[m] = i end

local function stringToDate(str)
    local year, month, day = str:match("(%d+)-(%d+)-(%d+)")
    if not tonumber(year) or not tonumber(month) or not tonumber(day) then
    ↪  return nil end
    return {
        year = tonumber(year),
        month = tonumber(month),
        day = tonumber(day),
    }
end

local dateString = editor.editString("", "date as string:")
local dateTableNumeric = editor.editTable({
    year = editor.editNumber(),
    month = editor.editNumber(),
    day = editor.editNumber(),
}, "date as numeric table:")
local dateTableNamedMonth = editor.editTable({
    year = editor.editNumber(),
    month = editor.editOptions(months[1], months, nil, "choose a month:"),
    day = editor.editNumber(),
}, "date as named-month table:")

return task.anyTask {dateString, dateTableNumeric, dateTableNamedMonth} ~ {
    {
        type = "string",
        action = "continue",
        fn = function(dateStr)
            local date = stringToDate(dateStr)
            if date then return task.constant(date) end
        end
    },
    {
        type = typed.Schema("DateTableNumeric")
            :field("year", "number")
            :field("month", "number")
            :field("day", "number"),
        action = "continue",
        fn = function(date)
```

```
            return task.constant(date)
        end
    },
    {
        type = typed.Schema("DateTableNamed")
            :field("year", "number")
            :field("month", "string")
            :field("day", "number"),
        action = "continue",
        fn = function(date)
            date.month = validMonths[date.month]
            return task.constant(date)
        end
    },
} ~ {{fn = function(date)
    return editor.viewInformation(app.pretty(date))
end}}
```

A.2.2 In iTasks

```
module DateFormats

import StdEnv
import iTasks

import iTasks.Extensions.DateTime
import Data.Functor
from Data.List import elemIndex

Start :: *World -> *World
Start world = doTasks dateTask world

:: DateAsNamedMonth =
    { year   :: !Int
    , mon    :: !String
    , day    :: !Int
    }

:: DateFormat
    = AsString !String
    | AsNumeric !Date
    | AsNamedMonth !DateAsNamedMonth

derive JSONEncode   DateAsNamedMonth, DateFormat
derive JSONDecode   DateAsNamedMonth, DateFormat
derive gEq          DateAsNamedMonth, DateFormat
derive gText        DateAsNamedMonth, DateFormat
derive gEditor      DateAsNamedMonth, DateFormat
derive gHash        DateAsNamedMonth, DateFormat

months = ["jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep",
↪  "oct", "nov", "dec"]
```

```
toMonth :: String -> ? Int
toMonth monthName = (\x -> x + 1) <$> (elemIndex monthName months)

dateString :: Task DateFormat
dateString = updateInformation [] "" <<@ Hint "date as string:"
    @ AsString

dateNumeric :: Task DateFormat
dateNumeric = enterInformation [] <<@ Hint "date as numeric:"
    @ AsNumeric

dateNamedMonth :: Task DateFormat
dateNamedMonth = enterInformation [] <<@ Hint "date as named-month:"
    @ AsNamedMonth

toDate :: DateFormat -> ? Date
toDate (AsString date) = error2mb (parseDate date)
toDate (AsNumeric date) = ?Just date
toDate (AsNamedMonth {DateAsNamedMonth|year,mon,day}) = case toMonth mon of
    ?Just monthInt = ?Just {Date|year=year, mon=monthInt, day=day}
    ?None = ?None
toDate _ = ?None

tvToDate :: (TaskValue DateFormat) -> ? (Task Date)
tvToDate (Value date _) = case toDate date of
    ?Just date = ?Just (return date)
    ?None = ?None
tvToDate (NoValue) = ?None

dateTask = anyTask [dateString, dateNumeric, dateNamedMonth]
    >>* [OnAction ActionContinue tvToDate]
    >>- (\date -> viewInformation [] date)
```

Appendix B

LTask code

The full code repository can be found at `https://github.com/Dantevg/LTasks`, the three most important files for this thesis are attached here: `task.lua` in B.1, `types.lua` in B.2 and `ltuiEditor.lua` in B.3.

B.1 task.lua

```lua
--[[--
    This module contains functions for creating tasks and for composing them.
]]

local ltuiElements = require "LTask.ltuiElements"
local app = require "LTask.ltuiApp"
local types = require "LTask.types"

local task = {}
task.__index = task

function task.new(fn, name, value)
    local self = {}
    self.stable = false
    self.value = value
    self.__name = name or ""
    self.co = coroutine.create(fn)
    return setmetatable(self, task)
end

-- Create a task that simply holds the value given.
-- Is stable immediately.
--
-- iTasks equivalent: [`return`](https://cloogle.org/#return%20%3A%3A%20a%20-%3E%20Task%20a)
-- `a -> Task a`
function task.constant(value)
    return task.new(function(self)
        self.value = value
```

```lua
        self.stable = true
    end, "constant")
end

---Transform the resulting value and stability of task `t` with function `fn`.
---
---iTasks equivalent: [`transform`](https://cloogle.org/#transform)
---`Task a, ((a, boolean) -> (b, boolean)) -> Task b`
---@param t table `Task a`
---@param fn function `(a, boolean) -> (b, boolean)`
---@return table `Task b`
function task.transform(t, fn)
    return task.new(function(self, options)
        while not self.stable do
            t:resume(options)
            self.__name = t.__name -- Transform is transparent
            self.value, self.stable = fn(t.value, t.stable)
            self, options = coroutine.yield()
        end
    end, "transform")
end

---Transform the result of task `t` with function `fn` without changing the
---stability.
---
---iTasks equivalent: [`@`](https://cloogle.org/#@)
---`Task a, (a -> b) -> Task b`
---@param t table `Task a`
---@param fn function `a -> b`
---@return table `Task b`
function task.transformValue(t, fn)
    return task.transform(t, function(value, stable) return fn(value), stable end)
end

local function matchContinuation(value, stable, action, conts)
    local matching, matchingConts = {}, types.matchAll(value, conts)
    table.sort(matchingConts, types.lt)
    local bestType
    for _, cont in ipairs(matchingConts) do
        if bestType and types.lt(cont.type, bestType) then break end
        if cont.action == nil or action == cont.action then
            local next = cont.fn(value, stable)
            if next then
                table.insert(matching, next)
                if not bestType then bestType = cont.type end
            end
        end
    end
    return matching
end

```

```lua
---Sequential combinator. Performs task `t` followed by the task returned by
---the matchng combinator from `conts`. When a match is found, the step happens.
---
---Custom operator: `..`
---
---iTasks equivalent: [`step`](https://cloogle.org/#step)
---or [`>>*`](https://cloogle.org/#%3E%3E*)
---`Task a, [ {type: any, action: string?, fn: (a -> Task b)} ] -> Task b`
---@param t table `Task a`
---@param conts table `[ {type: any, action: string?, fn: (a -> Task b)} ]`
---@return table `Task b`
function task.step(t, conts)
    return task.new(function(self, options)
        local matching = {}
        self.parent = options.parent
        while #matching == 0 do
            self.__name = "step (left, "..t.__name..")"
            if options.showUI then ltuiElements.stepDialog(self, conts, t) end
            t:resume(options, false, self)
            if t.value ~= nil then
                matching = matchContinuation(t.value, t.stable, options.action, conts)
            end
            if t.stable then break end
            if #matching == 0 then self, options = coroutine.yield() end
        end

        if #matching == 0 then error("no matching continuation for stable task") end

        -- Step happens here
        local next, nextNames = nil, {}
        if #matching > 1 then
            -- Allow user to choose continuation
            for _, nextTask in ipairs(matching) do table.insert(nextNames, nextTask.__name) end
            app.main:insert(
                ltuiElements.choiceEditor(nextNames[1], nextNames,
                    function(_, idx) return matching[idx] end, nil,
                    function(val) next = val end),
                {centerx = true, centery = true}
            )
            while not next do self, options = coroutine.yield() end
        else
            next = matching[1]
        end

        options.showUI = true -- Show self to reflect stepped task
        next:show(self) -- Automatically show continuation

        while not self.stable do
```

```lua
            self.__name = "step (right, "..next.__name..")"
            if options.showUI then ltuiElements.stepDialog(self, {}, next) end
            next:resume(options, false, self)
            self.value, self.stable = next.value, next.stable
            self, options = coroutine.yield()
        end
    end, "step")
end

---Helper function for `step` combinator. Returns a continuation configuration
---that matches when the given type and action match and there is any value.
---@param type any the type that the continuation accepts
---@param action string? the action that is needed for the continuation
---@param cont function `a -> Task b`
---@return table
function task.onAction(type, action, cont)
    return {
        type = type,
        action = action,
        fn = function(value)
            if value ~= nil then return cont(value) end
        end
    }
end

---Helper function for `step` combinator. Returns a continuation configuration
---that matches when the given type matches and there is a stable value.
---
---iTasks equivalent: [`ifStable`](https://cloogle.org/#ifStable)
---@param type any the type that the continuation accepts
---@param cont function `a -> Task b`
---@return table
function task.ifStable(type, cont)
    return {
        type = type,
        fn = function(value, stable)
            if value ~= nil and stable then return cont(value) end
        end
    }
end

---Sequential combinator with a single continuation. Continues when task `t`
---has a stable value.
---
---iTasks equivalent: [`>>-`](https://cloogle.org/#%3E%3E-)
---`Task a, (a -> Task b) -> Task b`
---@param t table `Task a`
---@param type any the type that the continuation accepts
---@param cont function `a -> Task b`
```

```lua
---@return table `Task b`
function task.stepStable(t, type, cont)
    return task.step(t, { task.ifStable(type, cont) })
end

---Sequential combinator with a single continuation. Continues when the user
---presses "continue" (only when task `t` has a value) or when task `t` has a
---stable value.
---
---iTasks equivalent: [`>>?`](https://cloogle.org/#%3E%3E%3F)
---`Task a, (a -> Task b) -> Task b`
---@param t table `Task a`
---@param cont function `a -> Task b`
---@return table `Task b`
function task.stepButtonStable(t, type, cont)
    return task.step(t, { task.onAction("continue", type, cont), task.ifStable(type, cont) })
end

-- Returns whether all tasks in `tasks` have stable values.
local function allStable(tasks)
    for _, t in ipairs(tasks) do
        if not t.stable then return false end
    end
    return true
end

---Parallel combinator. Performs all tasks in `tasks`. The result is the list
---of results of `tasks`.
---
---iTasks equivalent: [`parallel`](https://cloogle.org/#parallel)
---`[Task a] -> Task [{value: a, stable: boolean}]`
---@param tasks table `[Task a]`
---@return table `Task [{value: a, stable: boolean}]`
function task.parallel(tasks)
    local function getTaskNames()
        local taskNames = {}
        for _, t in ipairs(tasks) do table.insert(taskNames, t.__name) end
        return taskNames
    end

    return task.new(function(self, options)
        self.parent = options.parent
        self.value = {}
        while not self.stable do
            self.__name = "parallel ("..table.concat(getTaskNames(), ", ")..")"
            if options.showUI then ltuiElements.parallelDialog(self, tasks) end
            for i, t in ipairs(tasks) do
                if not t.stable then
```

```lua
                    t:resume(options, false, self)
                    self.value[i] = {value = t.value, stable = t.stable}
                end
            end
            self.stable = allStable(tasks)
            self, options = coroutine.yield()
        end
    end, "parallel")
end

-- Perform tasks in parallel and return the first stable value, or the first
-- unstable value if there are no unstable values.
--
-- iTasks equivalent: [`anyTask`](https://cloogle.org/#anyTask)
-- `[Task a] -> Task a`
function task.anyTask(tasks)
    return task.transform(
        task.parallel(tasks),
        function(values)
            local unstableValue
            for _, v in ipairs(values) do
                if v.value ~= nil and v.stable then
                    -- Stable value found, return immediately
                    return v.value, true
                elseif v.value ~= nil then
                    -- Set first unstable value we find
                    unstableValue = v.value
                end
            end
            -- No stable value found, return first unstable value found
            return unstableValue, false
        end
    )
end

-- Perform tasks `l` and `r` in parallel, yield both values
--
-- Custom operator: `&`
--
-- iTasks equivalent: [`-&&-`](https://cloogle.org/#-%26%26-)
-- `Task a, Task b -> Task (a, b)`
function task.parallelAnd(l, r)
    return task.transform(
        task.parallel {l, r},
        function(values)
            return {values[1].value, values[2].value},
                values[1].stable and values[2].stable
        end
    )
end

-- Perform tasks `l` and `r` in parallel, yield the first available value.
--
```

```lua
-- Custom operator: `|`
--
-- iTasks equivalent: [`-||-`](https://cloogle.org/#-%7C%7C-)
-- `Task a, Task b -> Task (a | b)`
function task.parallelOr(l, r)
    return task.anyTask {l, r}
end

-- Perform tasks `l` and `r` in parallel, yield only the result of task `l`
--
-- iTasks equivalent: [`-||`](https://cloogle.org/#-%7C%7C)
-- `Task a, Task b -> Task a`
function task.parallelLeft(l, r)
    return task.transform(
        task.parallel {l, r},
        function(values) return values[1].value, values[1].stable end
    )
end

-- Perform tasks `l` and `r` in parallel, yield only the result of task `r`
--
-- iTasks equivalent: [`||-`](https://cloogle.org/#%7C%7C-)
---`Task a, Task b -> Task b`
function task.parallelRight(l, r)
    return task.transform(
        task.parallel {l, r},
        function(values) return values[2].value, values[2].stable end
    )
end



---Resumes the coroutine of the task with the given options
---@param options table
---@param showUI boolean? if set, sets `options.showUI` to this value
---@param parent table? if set, sets `options.parent` to this value
---@return any value
---@return boolean stability
function task:resume(options, showUI, parent)
    if self.stable then return self.value end
    if coroutine.status(self.co) == "dead" then return end

    options = options or {}
    if showUI ~= nil then options.showUI = showUI end
    if parent ~= nil then options.parent = parent end
    local success, err = coroutine.resume(self.co, self, options)

    if not success then error(err) end
    return self.value, self.stable
end

---Resumes the task with `options.showUI` enabled
---@param parent table
---@return any value
```

```lua
---@return boolean stability
function task:show(parent)
    return self:resume({showUI = true, parent = parent})
end

task.__band = task.parallelAnd
task.__bor = task.parallelOr
task.__bxor = task.step
task.__concat = task.step -- For backwards compatibility: `..` is
↪  right-associative

return task
```

B.2 types.lua

```lua
--[[--
    This module contains functions for comparing types and specificity.
]]

local typed = require "typed"

local types = {}

-- taken from https://github.com/SovietKitsune/typed/blob/master/typed.lua
local function trim(str)
    return string.match(str, '^%s*(.-)%s*$')
end

---Get the number of fields in a Typed schema
---@param schema table
---@return number
local function schemaLength(schema)
    local n = 0
    for _ in pairs(schema._fields) do n = n + 1 end
    return n
end

---Transform types for use with typed
function types.toType(type_)
    if type_ == nil then
        return "any"
    elseif type_ == "table" then
        return "table<any, any>"
    else
        return type_
    end
end

---Match `value` with the continuation types in `conts`
---@param value any the value to match
---@param conts table the list of continuations, of form `{type = ""}`
---@return table matching the list of matching continuations
function types.matchAll(value, conts)
```

```lua
    -- Order is important
    local matching = {}
    for _, cont in ipairs(conts) do
        local type_ = types.toType(cont.type)
        if type(type_) == "string" then
            if typed.is(type_, value) then table.insert(matching, cont) end
        else
            if type_:validate(value) then table.insert(matching, cont) end
        end
    end
    return matching
end

---Compare two types on their specificity, returns whether `a` < `b`.
---
---This relation must define a strict partial order according to the Lua docs
---(https://www.lua.org/manual/5.3/manual.html#pdf-table.sort).
---This means that it must be:
--- - Irreflexive: not `a` < `a`
--- - Asymmetric: if `a` < `b` then not `b` < `a`
--- - Transitive: if `a` < `b` and `b` < `c` then `a` < `c`
---@param a any
---@param b any
---@return boolean lt whether `a` is less specific than `b`
function types.lt(a, b)
    local aIsString, bIsString = type(a) == "string", type(b) == "string"
    local aIsSchema = type(a) == "table" and a.__name == "Schema"
    local bIsSchema = type(b) == "table" and b.__name == "Schema"

    -- empty struct (schema) is equal to "table"
    if bIsSchema and schemaLength(b) == 0 then
        b = "table"
        bIsString, bIsSchema = true, false
    end

    if a == b then
        return false -- Irreflexivity
    elseif a == "any" and b ~= "any" then
        return true -- any < T  if T != any
    elseif aIsString and a:match("|")
            and not (bIsString and b:match("|")) then
        return true -- T_1 | T_2 < t_3
    elseif aIsString and a:match("|")
            and bIsString and b:match("|") then
        -- T_1 | T_2 < T_1 | T_3  if T_2 < T_3
        local firstA, restA = string.match(a, "(.-)%s*|%s*(.+)")
        local firstB, restB = string.match(b, "(.-)%s*|%s*(.+)")
        if firstA == firstB then
            return types.lt(restA, restB) -- T_2 < T_3
        else
            -- First types (T_1) are not equal, cannot compare
            -- (assume types are ordered the same)
            return false
```

```
        end
    elseif a == "table" and bIsString and b:match("(%[%])") then
        return true -- table < table(T)
    elseif aIsString and a:match("%[%]")
            and bIsString and b:match("%[%]") then
        local listTypeA = trim(a):match("(.+)%[%]$")
        local listTypeB = trim(b):match("(.+)%[%]$")
        return types.lt(listTypeA, listTypeB)
        -- table(T_1) < table(T_2)  if T_1 < T_2
    elseif a == "table" and bIsString and b:match("table<.-,%s*.->") then
        return true -- table < table(T_1, T_2)
    elseif aIsString and a:match("table<.-,%s*.->") and bIsSchema then
        return a:match("table<(.-),%s*.->") == "string"
        -- table(string, T) < {F_1, ..., F_n}
    elseif aIsSchema and bIsSchema then
        local fields = {}
        for k, v in pairs(a._fields) do
            fields[k] = fields[k] or {}
            fields[k].a = v[1]
        end
        for k, v in pairs(b._fields) do
            fields[k] = fields[k] or {}
            fields[k].b = v[1]
        end

        -- {F_1, ..., F_n, k : T} < {G_1, ..., G_m, k : T}  if {F_1, ..., F_n} < {G_1, ..., G_m}
        local nonequalFields = {}
        for k, v in pairs(fields) do
            if v.a ~= v.b then nonequalFields[k] = v end
        end

        for k, v in pairs(nonequalFields) do
            if v.a ~= nil and v.b ~= nil then
                if not types.lt(v.a, v.b) then return false end
            elseif v.a ~= nil and v.b == nil then
                return false
            end
        end
        return true
        -- {F_1, ..., F_n, k : T_1} < {G_1, ..., G_m, k : T_2}
        -- if T_1 < T_2 and {F_1, ..., F_n} < {G_1, ..., G_m}
    else
        return false -- If none of the above rules match, a is not less specific than b
    end
end

return types
```

B.3 ltuiEditor.lua

```lua
--[[--
    This module uses the UI elements from ltuiElements.lua and converts them
    to tasks.
]]

local task = require "LTask.task"
local ltuiElements = require "LTask.ltuiElements"
local app = require "LTask.ltuiApp"

local editor = {}

---Show `value` to the user with a prompt before.
---
---iTasks equivalent: [`viewInformation`](https://cloogle.org/#parallel)
---`a, String? -> Task String`
---@param value string the value to display
---@param prompt string?
---@return table task the resulting editor task
function editor.viewInformation(value, prompt)
    local function showUI()
        app.main:insert(
            ltuiElements.stringView(app.pretty(value), prompt),
            {centerx = true, centery = true}
        )
    end

    return task.new(function(self, options)
        self.value = tostring(value)
        while true do
            if options.showUI then showUI() end
            self, options = coroutine.yield()
        end
    end, (prompt or "viewInformation").." ("..tostring(value)..")", value)
end

local function genericEditor(value, showUI, onAction, name)
    return task.new(function(self, options)
        self.parent = options.parent
        self.value = value
        while true do
            self.__name = name.." ("..app.pretty(self.value)..")"
            if options.action and onAction then
                onAction(self, options)
            elseif options.showUI then
                showUI(self)
            end
            self, options = coroutine.yield()
        end
    end, name.." ("..app.pretty(value)..")", value)
end

```

```lua
---An editor for strings.
---@param value string? the initial value
---@param prompt string?
---@return table task the resulting editor task
function editor.editString(value, prompt)
    return genericEditor(value, function(self)
        local dialog = ltuiElements.stringEditor(self.value, prompt,
            function(val)
                self.value = val
                self.__name = "editString".." ("..tostring(self.value)..")"
                if self.parent then self.parent:show() end
            end)
        app.main:insert(dialog, {centerx = true, centery = true})
        return dialog
    end, nil, prompt or "editString")
end

---An editor for numbers.
---@param value number? the initial value
---@param prompt string?
---@return table task the resulting editor task
function editor.editNumber(value, prompt)
    return genericEditor(value, function(self)
        local dialog = ltuiElements.numberEditor(self.value, prompt,
            function(val)
                self.value = val
                self.__name = "editNumber".." ("..tostring(self.value)..")"
                if self.parent then self.parent:show() end
            end)
        app.main:insert(dialog, {centerx = true, centery = true})
        return dialog
    end, nil, prompt or "editNumber")
end

---An editor for a pre-determined set of inputs.
---@param value any the initial value
---@param choices table the list of possible choices
---@param converter function|table? the function to use for converting the values
---@param prompt string?
---@return table task the resulting editor task
function editor.editOptions(value, choices, converter, prompt, name)
    return genericEditor(value, function(self)
        local dialog = ltuiElements.choiceEditor(
            self.value ~= nil and tostring(self.value) or "",
            choices, converter, prompt,
            function(val)
                self.value = val
                self.__name = (name or "editOptions").." ("..tostring(self.value)..")"
                if self.parent then self.parent:show() end
            end)
        app.main:insert(dialog, {centerx = true, centery = true})
        return dialog
```

```lua
    end, nil, prompt or name or "editOptions")
end

---An editor for a pre-determined set of inputs.
---@param value boolean? the initial value
---@param prompt string?
---@return table task the resulting editor task
function editor.editBoolean(value, prompt)
    return editor.editOptions(value, {"true", "false"}, {true, false}, prompt,
        prompt or "editBoolean")
end

---An editor for tables.
---@param editors table? the sub-editors
---@param prompt string?
---@return table task the resulting editor task
function editor.editTable(editors, prompt)
    local value = {}
    for key, ed in pairs(editors) do
        value[key] = ed.value
    end
    return genericEditor(value, function(self)
        ltuiElements.tableEditor(self, editors or {}, prompt,
            function(val) self.value = val end,
            function(name)
                editors[name] = nil
                self:show()
            end)
    end, function(self, options)
        if options.action == "add array" then
            local dialog = ltuiElements.numberEditor(#editors+1, "enter an index",
                function(key) self:resume {action = "add", key = key} end)
            app.main:insert(dialog, {centerx = true, centery = true})
        elseif options.action == "add named" then
            local dialog = ltuiElements.stringEditor("", "enter a name",
                function(key) self:resume {action = "add", key = key} end)
            app.main:insert(dialog, {centerx = true, centery = true})
        elseif options.action == "add" then
            local dialog = ltuiElements.choiceEditor("string",
                {"string", "number", "boolean", "table"}, nil, "choose a type",
                function(editorType)
                    editors[options.key] = editor.editInformation(nil, nil, editorType)
                    self:show()
                end)
            app.main:insert(dialog, {centerx = true, centery = true})
        end
    end, prompt or "editTable")
end

---An editor for strings, numbers, booleans or tables.
```

```lua
---@param value any? the initial value
---@param prompt string?
---@param editorType "string" | "number" | "boolean" | "table" the type of
↪ the editor
---@return table task the resulting editor task
function editor.editInformation(value, prompt, editorType)
    editorType = editorType or type(value)
    if editorType == "string" then
        return editor.editString(value, prompt)
    elseif editorType == "number" then
        return editor.editNumber(value, prompt)
    elseif editorType == "boolean" then
        return editor.editBoolean(value, prompt)
    elseif editorType == "table" then
        return editor.editTable(value, prompt)
    end
end

return editor
```