Task Oriented Programming for the Internet of Things

Mart Lubbers Pieter Koopman Rinus Plasmeijer

17th June 2019

Contents

Chapter 1

Introduction

This book will introduce the reader with the Task Oriented Programming (TOP) for the Internet of Things (IOT).

1.1 Structure of the Book

Chapter 1 contains the introductions to the IOT, TOP and iTasks. A general introduction to mTask is given in Chapter 2 containing a brief history and an overview to the architecture. Chapter 3 gives a step-by-step introduction on how to actually use the mTask with accompanying assignments. It starts with a simple blink application, continuing with threads, interaction with the server, multitasking and controlling the LED matrix, writing a thermostat. It concludes with a fairly advanced assignment in where you create a temperature sensor that plots the temperature and has an adjustable delay.

The appendices contain background information and language manuals for the used languages. Appendix A contains installation instructions for mTask. Appendix B shows background information about embedding in general. Appendix C is a complete mTask reference containing all the classes and information about how to interact with the used backends. Appendix D contains a very brief guide to functional Clean and Appendix E a similarly brief guide for iTasks. If you are already familiar with any of the subjects in the appendices you can skip them. If not, you are advised to at least glance through them. You are anyhow advised to at least read Appendix D.1 about Cloogle.

1.2 IOT

IOT technology is overtaking the world rapidly. This new technology changes the way people interact with the world.

The term IOT was coined around 1999 to describe the communication between Radio-frequency Identification (RFID) devices. RFID became more and more popular the years after, however, the term IOT was not used as much anymore. Years later, during the rise of novel networks technologies, the term IOT resurged with a slightly different meaning. Today, the IOT is the term for a system of devices that sense the environment, act upon it and communicate with each other and the world. As we speak there are around seven billion devices connected part of an IOT system. IOT devices are already in everyone's household in the form of smart electricity meters, smart fridges, smartphones, smart watches, home automation and in the form

of much more. The devices used boast various sensors ranging from more external ones such as positioning, temperature and humidity to more internal ones like heartbeat and respiration [8]. Estimations for the future vary greatly, from 26 billion[1] in 2020, 20 billion[2] in 2020, 25 billion in 2025[3], to the astronomical number of fifty billion devices by 2020[4].

When describing IOT systems, a layered architecture is often used to compartmentalize the technology. For example using the popular four layer architecture of which the first layer is called the sensing layer. The actual devices with their peripherals are in this layer. For example in home automation, the sensors reading the room, the actuators opening the curtains are all in the sensing layer. The networking layer is the second layer and it consists of the hard and software to connect the sensing layer to the world. In home automation, this layer may consist of a specialized IOT technology such as Bluetooth Low Energy (BTLE) or ZigBee network but it may also use existing technologies such as WiFi or wired connections. The third layer is named service layer and is responsible for the servicing and business rules surrounding the application. One of its goals is to provide the API, interfaces and data storage. In home automation this provides the server storing the data. The fourth and final layer in this architecture is the application layer. The application layer provides the interaction of the user with the IOT system. In home automation, this layer provides for example the apps for to read the measurements and control the devices.

IOT applications change rapidly but the devices stay roughly the same. The clients are often heterogeneous collections of micro- controllers having each their own peculiarities, language of choice and hardware interfaces. Evenmoreso, the hardware needs to be cheap, small-scale and energy efficient. As a result, the Microcontroller Units (MCUs) used to power these devices do not have a lot of computational power, a soupçon of memory, and little communication bandwidth. Typically the devices do not run a full fledged OS but a statically compiled firmwares. This firmware is often written in an imperative language that needs to be flashed to the program memory. This greatly reduces the flexibility for dynamic systems where tasks are created on the fly and executed on demand. While devices are getting a bit faster, smaller, and cheaper, they keep these properties to an extent. These problems can be solved by dynamically sending TOP code to be interpreted to the MCU.

1.3 TOP

Tasks and their task values are the first class citizens in TOP. A task represents work that needs to be done by someone or something. This work can be anything ranging from filling in a form, reading a sensor, send an email or even actual physical work. Furthermore, tasks can be transformed or combined to form new tasks.

Tasks emit a three state task value that is observable by other tasks. No value means that the task is unable to emit a *complete* value. It might be the case that some work has been done but just not enough (e.g. an open serial port with a partial message). An unstable value means that a complete value is present but it may change in the future (i.e. a side effect). A web editor is an example of a task that always emits an unstable value since the contents may change over time. Stable values never change. When the continue button has been pressed and the contents of the web editor have been relayed, the values can never change, hence it is stable. Only the state transitions in Figure 1.1 are legal.

[1] Gartner, March 2014
[2] Gartner, January 2017
[3] GSMA Intelligence, June 2018
[4] Cisco, 2016

Figure 1.1: State diagram for the legal transitions of task values

Tasks can also share data type- and thread-safe using Shared Data Sources (SDSs). SDSs are an abstraction over any data. An SDS can represent typed data stored in a file, a chunk of memory, a database etc. SDSs can also represent truly impure data such as the time, random numbers or access to sensors. Similarly to tasks, the transformation and combinator of SDSs is possible. In this architecture, tasks function as lightweight communicating threads

The reference implementation for TOP is iTasks [15], an Embedded Domain Specific Language (EDSL) hosted in Clean [5]. The iTasks system realizes TOP for developing distributed collaborative web applications.

1.4 iTasks

The iTasks system is implemented as a shallowly embedded Domain Specific Language (DSL) in Clean.

Chapter 2

mTask

The mTask language is a part of the mTask system. The language has several backends, for this book we are only using the pretty printing backend and the bytecode generation backend. With these backends, the mTask system is a programming environment for programming all layers of an IOT system from a single source.

It consists of several components:

- mTask language

  The language itself is a collection of classes and has two backends that are used in this book:

  - Pretty printing
  - Bytecode generation

- MCU Run-time System (RTS) with versions for:

  - Arduino compatible AVR microcontrollers (e.g. Arduino UNO)
  - Arduino compatible xtensa microcontrolles (e.g. NodeMCU or LOLIN).
  - Windows, Linux or MacOS compatible `x86` computers.

- iTasks integration

  The integration consists of transparent task lifting

A full reference manual of all language constructions with examples can be found in Appendix C.

2.1 History

A first throw at a class-based shallowly EDSL for MCUs was made by Pieter Koopman and Rinus Plasmijer in 2016 [16]. The language was called Arduino Domain Specific Language (ARDSL) and offered an type safe interface to Arduino C++ dialect. A C++ code generation backend was available together with an iTasks simulation backend. There was no support for tasks or even functions. Some time later an unpublished extended version was created that allowed the creation of imperative tasks, SDSs and the usage of functions. It was named mTask.

Mart Lubbers extended on this in his Master's Thesis by adding integration with iTasks and a bytecode compiler to the language [12]. SDS in mTask could be accessed on the iTasks server. In this way, entire IOT systems could be programmed from a single source. However, this version used a simplified simplified version of mTask without functions. This was later improved upon by creating a simplified interface where SDSs from iTasks could be used in mTask and the other way around [14]. It was shown by Matheus Amazonas Cabral de Andrade that it was possible to build real-life IOT systems with this integration [4].

The mTask language as it is now was introduced in 2018 [11]. This paper updated the language to support functions, tasks and SDSs. It still compiled to static C++ Arduino code. Later the bytecode compiler and iTasks intergration was added to the language [1]. Moreover, it was shown that it is very intuitive to write MCU applications in a TOP language [13]. The reason for this is that you get a lot of design patterns that are difficult using standard means for free (e.g. multithreading). Furthermore, Erin van der Veen has worked and will probably be working on a green computing analysis of the mTask language.

2.2 Architecture

[1] under review

Chapter 3

Building mTask Applications

This chapter is a hands-on introduction to writing applications in mTask and iTasks. Skeletons are listed between brackets and can be found in the `mTask/cefp19` directory.

3.1 Hardware & Client

For the examples we use the WEMOS LOLIN D1 mini[1]. The D1 mini is an ESP8266 based prototyping board boasting 1 analog and 11 digital General Purpose Input/Output (GPIO) pins and a micro USB connection for programming and debugging. It can be programmed using MicroPython, Arduino and nodemcu (LUA).

For this purpose they come preinstalled with the mTask runtime and ready to connect to the hotspot and with several shield attached. Details on how to program the device yourself can be found in Appendix A.4.

The devices come installed on a three-way splitter and is setup with an OLED, SHT and Matrix LED shield. The OLED shield is used for displaying runtime during boot. When booting up, it will show the WiFi status and when connected it will show the IP address that it uses. Furthermore, the OLED screen contains two buttons that can be accessed from within mTask to get some kind of feedback from the user.The SHT shield houses a Digital Humidity and Temperature sensor (DHT) sensor that can be accessed from mTask as well. The LED matrix can be accessed through mTask and can be used to display information to the user as well.

It is not always convenient to have to work with physical devices and therefore a simulation client is available that works on your regular x86 machine (See Appendix A.4.5). This simulation client is built from the exact same sources as the firmwares for the physical devices and can therefore be used to debug more quickly.

3.2 Blink

Traditionally, the first program that one writes when trying a new language is the so called *Hello World!* program. This program has the single task of printing the text *Hello World!* to the screen and exiting again. On microcontrollers, there often is no screen for displaying text. Nevertheless, almost always there is a rudimentary single pixel screen, namely an — often builtin — LED.

[1] `https://wiki.wemos.cc/products:d1:d1_mini`

Listing 3.1 shows the blink program when written in the Arduino C++ dialect. When flashing the generated firmware onto the device, the builtin LED[2] changes state every 500 milliseconds.

```
void setup () {
   pinMode(D4, OUTPUT);
}

void loop() {
   digitalWrite(D4, HIGH);
   delay(500);
   digitalWrite(D4, LOW);
   delay(500);
}
```

Listing (3.1) Blink in Arduino.

```
blink :: Main (MTask v ()) | mtask v
blink = {main=rpeat (
            writeD d4 (lit True)
      >>|. delay (lit 500)
      >>|. writeD d4 (lit False)
      >>|. delay (lit 500)
   )}
```

Listing (3.2) An mTask Translatation of Hello World! (`blinkImp`)

The program can be translated almost literally using mTask constructs as seen in Listing 3.2. In an Arduino program, the `loop` function is called continously during the execution of the program. To simulate this, the `rpeat` task can be used, this task executes the argument task and, when stable, reinstates it.

However, this is not very functional nor task oriented. A more natural translation would be the one in the complete program shown in Listing 3.3.

```
1  module blink
2
3  import StdEnv, iTasks
4
5  import Interpret
6  import Interpret.Device.TCP
7
8  Start :: *World → *World
9  Start w = doTasks main w
10
11 main :: Task Bool
12 main = enterDevice >>= λspec → withDevice spec
13    λdev → liftmTask blink dev -|| viewDevice dev
14 where
15    blink :: Main (MTask v Bool) | mtask v
16    blink
17       = fun λblink=(λx →
18                delay (lit 500)
19          >>|. writeD d4 x
20          >>=. blink o Not)
21       In {main=blink (lit True)}
```

Listing 3.3: A functional mTask Translatation of Hello World! (`blink`)

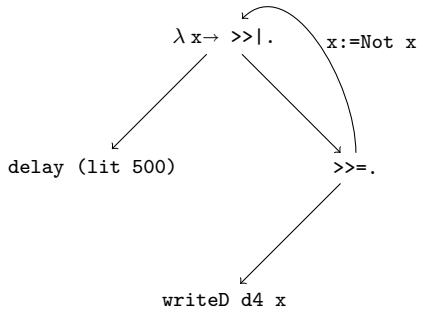
To explain the program and automatically have a nice skeleton for writing your own mTask programs we will go through the program line by line. Future snippets will only give the mTask code for brevity.

Line 1 declares the name of the module, this has to match the name of the filename (See Appendix D.2). Lines 3–6 regard the imports for the external libraries. Line 3 import `StdEnv` and `iTasks`, these imports are required when using `iTasks`. Lines 5–6 import the `Interpret` —the mTask bytecode backend — and `Interpret.Device.TCP` — the TCP device connectivity. Both imports are

[2]The builtin LED is connected to digital GPIO pin 2 on the LOLIN D1 mini. On the Arduino UNO this is digital GPIO pin 13.

always required for this book. Lines 8–9 gives the `Start` function, the entrypoint for a Clean program. This start function always calls the iTasks specific entrypoint called `doTasks` that starts up the iTasks machinery and launches the task `main`.

The `main` task first starts with an editor on line 12. This editor presents an interface to the user connecting to the server for it to select a device. The `enterDevice` task allows selecting devices from presets and allows changing the parameters to select a custom device. After selecting a device, the task continues with connecting the device `withDevice` that takes a function requiring a device and resulting in a task. This function (Line 13) executes the `blink` task and shows some information about the device in parallel. Lines 15–21 contain the actual task. To make reusing more easy, the blinking behaviour is lifted to a function. The function takes a single argument, the state and recursively calls itself continuously. It creates an infinite task that first waits 500 milliseconds. Then it will write the current state to the pin followed by a recursive call to with the inverse of the state.



Assignment 3.1: Blink the builtin LED

Compile and run the blink program to test your mTask setup (`blink`). Instructions on how to install mTask can be found in Appendix A.

3.3 Threaded Blinking

Now say that we want to blink multiple blinking patterns on different LEDs concurrently. Intuitively we want to lift the blinking behaviour to a function and call this function three times with different parameters as done in Listing 3.4.

```
void blink (int pin, int wait) {
   digitalWrite(pin, HIGH);
   delay(wait);
   digitalWrite(pin, LOW);
   delay(wait);
}

void loop() {
   blink (1, 500);
   blink (2, 300);
   blink (3, 800);
```

```
}
```

Listing 3.4: Naive approach to multiple blinking patterns in Arduino.

Unfortunately, this does not work because the `delay` function blocks all further execution. The resulting program will blink the LEDs after each other instead of at the same time. To overcome this, it is necessary to slice up the blinking behaviour in very small fragments so it can be manually interleaved [9]. Listing 3.5 shows how three different blinking patterns might be achieved in Arduino using the slicing method. If we want the blink function to be a separate parametrizable function we need to explicitly provide all references to the required state. Furthermore, the `delay` function can not be used and polling `millis` is required. The `millis` function returns the number of milliseconds that have passed since the boot of the MCU. Some devices use very little energy when in `delay` or sleep state. Resulting in `millis` potentially affects power consumption since the processor is basically busy looping all the time. In the simple case of blinking three LEDs on fixed intervals, it might be possible to calculate the delays in advance using static analysis and generate the appropriate `delay` code. Unfortunately, this is very hard when for example the blinking patterns are determined at runtime.

The manual interleaving method is very error prone, requires a lot of pointer juggling and generally results into spaghetti code. Furthermore, it is very difficult to represent dependencies between threads, often state machines have to be explicitly programmed by hand to achieve this.

```
long led1 = 0, led2 = 0, led3 = 0;
bool st1 = false, st2 = false, st3 = false;

void blink(int pin, int delay, long *lastrun, bool *st) {
   if (millis() - *lastrun > delay) {
      digitalWrite(pin, *st = !*st);
      *lastrun += delay;
   }
}

void loop() {
   blink(1, 500, &led1, &st1);
   blink(2, 300, &led2, &st1);
   blink(3, 800, &led3, &st1);
}
```

Listing 3.5: Threading three blinking patterns in Arduino.

Blinking multiple patterns in mTask is as simple as combining several calls for the `blink` function from Listing 3.3 with a parallel combinator as shown in Listing 3.6.

```
1  blink :: Main (MTask v Bool) | mtask v
2  blink
3     = fun λblink=(λ(p, x, y)→
4              delay y
5        >>|. writeD p x
6        >>=. λx→blink (p, Not x, y))
7     In {main=blink (d1, true, lit 500)
8        .||. blink (d2, true, lit 300)
9        .||. blink (d3, true, lit 800)}
```

Listing 3.6: An mTask program for blinking multple patterns. (`blinkThread`)

Assignment 3.2: Blink the builtin LED with two patterns

Adapt the program in Listing 3.6 so that it blinks the builtin LED with two different patterns concurrently. The times for the patterns can be queried from the user.

The function signature for `blink` becomes (`blinkThread`):

```
blink :: Int Int → Main (MTask v Bool) | mtask v
```

You can use `enterInformation` to get the information from the user (See Appendix E.2).

3.4 Interactive Blinking

Assignment 2 showed that Clean can be used as a macro language for mTask. Customizing the tasks when needed. SDSs can also be used to interact with the mTask tasks during execution. This can for example be used to let the user control the blinking frequency. Listing 3.7 shows how the blinking frequency can be controlled by the user using SDSs.

```
1  main :: Task Bool
2  main = enterDevice >>= λspec → withDevice spec
3     λdev → withShared 500 λdelayShare →
4             liftmTask (blink delayShare) dev
5        -|| updateSharedInformation "Interval" [updater] delayShare
6  where
7     updater :: UpdateOption Int Int
8     updater = UpdateUsing (λx → (x, x)) (const fst)
9        (panel2
10          (slider <<@ minAttr 5 <<@ maxAttr 10000)
11          (integerField <<@ enabledAttr False))
12
13    blink :: (Shared s Int) → Main (MTask v Bool) | mtask, liftsds v & RWShared s
14    blink delayShare = liftsds λdelaysh=delayShare
15       In fun λblink=(λx →
16                writeD d4 x
17          >>|. getSds delaysh
18          >>∼. delay
19          >>=. λ_ → blink (Not x))
20       In {main=blink (lit True)}
```

Listing 3.7: An mTask program for interactively changing the blinking frequency. (`blinkInteractive`)

Line 3 shows the creation of the controlling iTasks SDS using `withShared` (See Appendix E.4). Line 5 adds a task to the `withDevice` function. It adds an `updateSharedInformation` to the mix with which the user can control the delay. The standard integer editor is unbounded so a custom editor is therefore used (lines 7–11). The `blink` task itself is hardly modified. Line 14 lifts the SDS to an mTask SDS using `liftsds` (see Appendix C.1.7). Note that the `>>∼.` combinator is used since the `getSds` task always yields an *unstable* value. The lifted SDS can be accessed as usual using the `getSds` task (line 17). The value this yields is immediately fed to `delay`. The mTask machinery takes care of the rest such as the automatic SDS updates.

Assignment 3.3: Blink the builtin LED on demand

Adapt the program in Listing 3.7 so that the user can control *whether* the LED blinks or not.

The `blink` function will then have the following type signature (`blinkInteractive`):

```
blink :: (Shared s Bool) → Main (MTask v Bool) | mtask, liftsds v & RWShared s
```

3.5 Multitasking

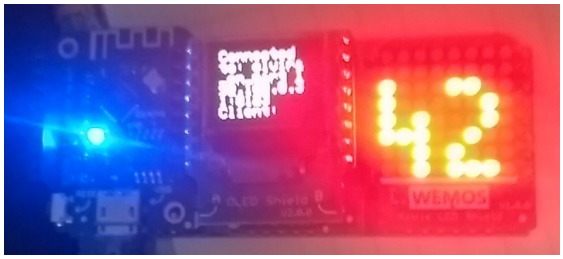


Figure 3.2: *The Answer* printed on the LED matrix.

The LED matrix shield can be used to display information during the execution of the program. The 8×8 LEDs can be controlled individually. The program in Listing 3.8 shows an iTasks program to control the LED matrix. It allows toggling the state of a given LED and clear the display.

A `Ledstatus` is a dedicated type created so that entering the information for toggling a single LED automagically gets a nice user interface. The main program is very similar to previous programs, only differing in the device part. The `>^*` combinator is a special kind of `parallel` combinator that — instead of stepping to a continuation — forks off a continuation. This allows the user to run multiple tasks to schedule many tasks in parallel. Continuations can be triggered by values or by actions. In this example, only actions are used that are always enabled. One action is added for every operation and when the user presses the button, the according task is sent to the device. The `toggle` and `clear` tasks are self-explanatory and only use LED matrix mTask functions (See Appendix C.1.8).

```
1  :: Ledstatus = {x :: Int, y :: Int, status :: Bool}
2  derive class iTask Ledstatus
3
4  main = enterDevice >>= λspec → withDevice spec
5     λdev → viewDevice dev >^*
6        [OnAction (Action "Toggle") (always (
7           enterInformation () [] >>= λs → liftmTask (toggle s) dev
8              >>~ viewInformation "done" []))
9        ,OnAction (Action "Clear") (always (
10          liftmTask clear dev
11          >>~ viewInformation "done" []))
12       ] @! ()
13 where
14    dot lm s = LMDot lm (lit s.x) (lit s.y) (lit s.status)
15
16    toggle :: Ledstatus → Main (MTask v ()) | mtask, LEDMatrix v
17    toggle s = ledmatrix D5 D7 λlm → {main=dot lm s >>|. LMDisplay lm}
18
19    clear :: Main (MTask v ()) | mtask, LEDMatrix v
20    clear = ledmatrix D5 D7 λlm → {main=LMClear lm >>|. LMDisplay lm}
```

Listing 3.8: An interactive mTask program for interacting with the LED matrix. (`matrixBlink`)

Toggling the LEDs in the matrix using the given tasks is very user intensive. Extend the program so that there is a button for printing the answer to the question of life, universe and everything [3] as seen in Figure 3.2. There are several approaches possible.

Assignment 3.4: LED Matrix 42 using iTasks

Write 42 to the LED matrix using only the `toggle` and the `clear` tasks. You can add the continuations as follows (`matrixBlink`):

```
,OnAction (Action "42") (always (iTask42 dev))
```

The iTasks tasks should then have the following type signature:

```
iTask42 :: MTDevice → Task ()
```

In this situation, a whole bunch of mTask tasks are sent to the device at once. This strains the communication channels greatly and is a risk for running out of memory. It is also possible to define printing 42 in solely in mTask. This creates one bigger task that is sent at once.

Assignment 3.5: LED Matrix 42 using mTask

Write 42 to the LED matrix as a single mTask task. This results in the following continuation (`matrixBlink`):

```
,OnAction (Action "42mtask") (always (liftmTask mTask42 dev))
```

The mTask task should then have the following type signature:

3.6 Temperature

Reading the temperature on the device can be done using the DHT sensor that is connected as a shield to the main board. The DHT chip attached to the board is the *SHT30* sensor. When asked via Inter-Intergrated Circuit (I^2C), the chip measures the temperature between with a $\pm 0.4^{\circ}C$ accuracy and the relative humidity with a $\pm 2\%$ accuracy.

It can be accessed using the mTask `dht` class (See Appendix C.1.8). For example, the following program will show the current temperature and humidity to the user. The yielded values from the `temperature` and `humidity` tasks are in tenths of degrees or percents. Therefore, a lens is applied on the editor to transform them into floating point values.

```
1 main = enterDevice >>= λspec → withDevice spec
2    λdev → liftmTask temp dev >&> viewSharedInformation () [ViewAs templens]
3 where
4    templens = maybe (0.0, 0.0) λ(t, h) → (toReal t / 10.0, toReal h / 10.0)
5
6    temp :: Main (MTask v (Int, Int)) | mtask, dht v
7    temp = DHT D4 DHT22 λdht → {main=temperature dht .&&. humidity dht}
```

Listing 3.9: An mTask program for measuring the temperature and humidity. (`tempSimple`)

[3] N.B. the answer is 42 [3]

Assignment 3.6: Show the temperature via an SDS

Modify the application so that it writes the temperature in a SDS. Writing the temperature constantly in the SDS creates a lot of network traffic. Therefore it is advised to create a function that will memorize the old temperature and only write the new temperature when it is different from the old one. Use the following template (`tempSds`):

```
main = enterDevice >>= λspec → withDevice spec
   λdev → withShared 0 λsh →
          liftmTask (temp sh) dev
      -|| viewSharedInformation "Temperature" [ViewAs templens] sh
where
   templens t = toReal t / 10.0

   temp :: (Shared s Int) → Main (MTask v ()) | mtask, dht, liftsds v & RWShared
    ↪ s
```

With the `writeD` functions from mTask (See Appendix C.1.8) the digital GPIO pins can be controlled. Imagine a heater attached to a GPIO pin that turns on when the temperature is below a given limit.

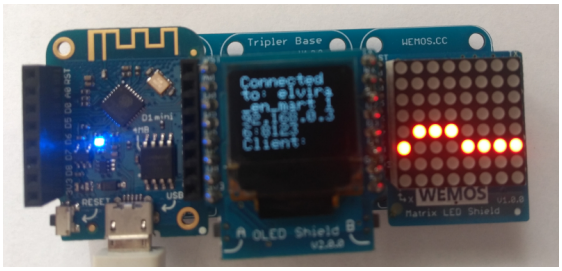
Assignment 3.7: Simple Thermostat

Modify the previous assignment so that a thermostat is mimicked. The user can enter a temperature target and the LED will turn on when the temperature is below the target. Use the following template (`thermostat`):

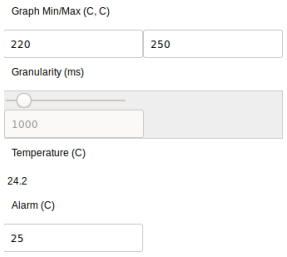
```
main = enterDevice >>= λspec → withDevice spec
   λdev → withShared 0 λtempShare →
         withShared 250 λtargetShare →
          liftmTask (temp targetShare tempShare) dev
      -|| viewSharedInformation "Temperature" [ViewAs tempfro] tempShare
      -|| updateSharedInformation "Target" [UpdateAs tempfro λ_ → tempto]
    ↪ targetShare
where
   tempfro t = toReal t / 10.0
   tempto  t = toInt t * 10

   temp :: (Shared s1 Int) (Shared s2 Int) → Main (MTask v ()) | mtask, dht,
    ↪ liftsds v & RWShared s1 & RWShared s2
   temp targetShare tempShare =
```

3.7 Temperature Plotter

For this final assignment you are going to create a temperature plotter with an alarm mode that uses all components. I.e. the LED matrix to show the plot, the OLED shield buttons to toggle the alarm, the builtin LED to show the alarm status and the DHT shield to measure the temperature. Figure 3.3a shows the reference implementation in action. Figure 3.3b shows the user interface for it.



(a) The temperature plotter in action.



(b) The temperature plotter UI.

Figure 3.3: The reference implementation of the plotter in action

Assignment 3.8: Temperature Plotter

The plotter has several jobs to do.

- Plot the temperature.

  Plotting the temperature happens on the LED matrix. The range of the graph is specified in the `limitsShare` and may be changed by the user.

- Report the temperature every interval.

  The temperature has to be written to the `tempShare` SDS so that the user interface can show an up to date temperature. Preferably it only writes to the SDS when the temperature has changed.

- Set the alarm.

  If the current temperature is higher than the alarm value (`alarmShare`), the builtin LED should turn on.

- Unset the alarm.

  The alarm can be unset by using the A button from the OLED shield.

Create the plotter using the following template (`plotter`):

```
module temp

import StdEnv, iTasks

import Interpret
import Interpret.Device.TCP

Start :: *World → *World
Start w = doTasks main w

BUILTIN_LED :== d3
ABUTTON :== d4

main = enterDevice >>= λspec → withDevice spec
  λdev → withShared (220, 250) λlimitsShare →
```

```
        withShared 1000 λwaitShare →
        withShared 0 λtempShare →
        withShared 250 λalarmShare →
         liftmTask (temp limitsShare waitShare tempShare alarmShare) dev
    -|| updateSharedInformation "Graph Min/Max (C, C)" [] limitsShare
    -|| updateSharedInformation "Granularity (ms)" [updater] waitShare
    -|| viewSharedInformation "Temperature (C)" [ViewAs tempfro] tempShare
    -|| updateSharedInformation "Alarm (C)" [UpdateAs tempfro λ_ → tempto]
  ↪ alarmShare
where
  tempfro t = toReal t / 10.0
  tempto  t = toInt t * 10

  updater :: UpdateOption Int Int
  updater = UpdateUsing (λx → (x, x)) (const fst)
    (panel2
      (slider <<@ minAttr 5 <<@ maxAttr 10000)
      (integerField <<@ enabledAttr False))
  temp :: (Shared s1 (Int, Int)) (Shared s2 Int) (Shared s3 Int) (Shared s4 Int)
    → Main (MTask v ()) | mtask, dht, liftsds, LEDMatrix v
    & RWShared s1 & RWShared s2 & RWShared s3 & RWShared s4
  temp limitsShare delayShare tempShare alarmShare =
```

Some tips are:

- Start with the preamble and a skeleton for the tasks.

  The preamble should at least lift the SDSs and define the peripherals (LED matrix and DHT).

- Use functions for state as much as possible.

  Especially for measuring the temperature, you do not want to write to the temperature SDS every time you measure. Therefore, keep track of the old temperature using a function or a local SDS.

- Write functions for routines that you do multiple times.

  For example, clearing a row on the LED matrix is a tedious job and has to be done every cycle. Make it yourself easy and either write it as a clean function that generates all the code or an mTask function that is called.

Appendix A

How to Install

This section will give detailed instructions on how to install mTask on your system.

A.1 Fetch the latest version

Download the Central European Functional Programming School (CEFP) version of mTask version for your operating system as given in table A.1 and decompress the archive. The archives is all you need since it contains a complete clean distribution. The windows version contains an Integrated Development Environment (IDE). Mac and Linux only have a project manager called Clean Project Manager (cpm).

OS	Arch	URL
Linux	x64	`https://ftp.cs.ru.nl/Clean/CEFP19/mtask-linux-x64.tar.gz` Requires GCC
Windows	x64	`https://ftp.cs.ru.nl/Clean/CEFP19/mtask-windows-x64.zip`
MacOS	x64	`https://ftp.cs.ru.nl/Clean/CEFP19/mtask-macos-x64.tar.gz` Requires XCode

Table A.1: Download links for the CEFP builds of mTask.

A.2 Setup

A.2.1 Linux

Assuming you uncompressed the archive in `~/mTask`, run the commands from listing A.1 in a terminal.

```
# Add the bin directory of the clean distribution to $PATH
echo 'export PATH=~/mTask/clean/bin:$PATH' >> ~/.bashrc
# Correctly set CLEAN_HOME
echo 'export CLEAN_HOME=~/mTask/clean' >> ~/.bashrc
```

Listing A.1: Linux setup instructions for mTask.

A.2.2 Windows

You do not need to setup anything on windows. However, if you want to use cpm as well, you need to add its directory to your `%PATH%` as follows[1].

Assuming you uncompressed the archive in `C:\Users\frobnicator\mTask`:

- Right-click on Computer. Then go to Properties » Advanced System settings » Environment Variables or press win + R and type `SystemPropertiesAdvanced.exe` and press ↵.
- Select *"Path"* from the list of system variables.
- Choose Edit and append (i.e., do not overwrite the previous value):

  `;C:\Users\frobnicator\mTask\clean`

A.2.3 MacOS

MacOS should work but is less tested and recent Clean builds have been reported to be a tad bit unstable.

Assuming you uncompressed the archive in `~/mTask`, run the commands from listing A.2 in a terminal.

```
# Add the bin directory of the clean distribution to $PATH
echo 'export PATH="~/mTask/clean/bin:$PATH" >> ~/.bash_profile
# Correctly set CLEAN_HOME
echo 'export CLEAN_HOME="~/mTask/clean" >> ~/.bash_profile
```

Listing A.2: MacOS setup instructions for mTask

A.3 Compile the test program

Note that the first time compiling everything can take a while and will consume quite some memory.

A.3.1 Windows

Assuming you uncompressed the archive in `C:\Users\frobnicator\mTask`. Connect a device or start the local TCP client by executing `C:\Users\frobnicator\mTask\client.exe`

IDE

- Open the IDE by starting `C:\Users\frobnicator\mTask\clean\CleanIDE.exe`.
- Click on File » Open or press Ctrl + O ond open `C:\Users\frobnicator\mTask\mTask\cefp19\blink.prj`.
- Click on Project » Update and Run or press Ctrl + R.

[1] Instructions from `https://hmgaudecker.github.io/econ-python-environment/paths.html`

cpm

In a command prompt or powershell session follow the commands listed in listing A.3.

```
cd C:\Users\frobnicator\mTask\mTask\cefp19
cpm blink.prj
blink.exe
```

Listing A.3: Compile and run the test program on windows with cpm.

A.3.2 Linux & MacOS

Assuming you uncompressed the archive in `~/mTask`. Connect a device or start the local TCP client by executing `~/mTask/client`. In a terminal follow the commands listed in listing A.4.

```
cd ~/mTask/cefp19
cpm blink.prj
./blink
```

Listing A.4: Compile and run the test program on Linux or MacOS with cpm.

A.4 Setup the Microcontroller

Programming the microcontroller yourself is also possible but a lot more elaborate.

A.4.1 Manual Method (All Platforms)

- Download and install the latest Arduino IDE[2].
- Open it and click on File » Preferences.
- Add the Arduino ESP8266 url[3] to the Additional Board Manager URLs.
- Open the boards manager from Tools » Board and install *esp8266* platform.
- Install all libraries from the mTask repository via: Sketch » Include Library » Add.ZIP Library...

A.4.2 Automatic Method (Linux)

This method assumes that there is no previous installation of Arduino and installs the prepped IDE in the `arduino` directory.

```
git clone https://gitlab.science.ru.nl/mlubbers/mtask
cd mtask
mkdir -p arduino
curl -SsL https://downloads.arduino.cc/arduino-1.8.9-linux64.tar.xz | tar -xJC
    ↪ arduino --strip-components=1
for f in dependencies/*.zip
do
   unzip -d arduino/libraries "$f"
done
mkdir -p arduino/hardware/esp8266
git clone --recursive https://github.com/esp8266/Arduino.git \
   arduino/hardware/esp8266/esp8266
```

[2] https://www.arduino.cc/en/Main/Software
[3] https://arduino.esp8266.com/stable/package_esp8266com_index.json

```
( cd arduino/hardware/esp8266/esp8266/tools/; python get.py; )
```

Listing A.5: Setup the arduino environment for mTask.

A.4.3 Flashing the RTS

Before flashing — and if using WiFi — make sure to set up `wifi.h`. You can do this by using the `wifi.def.h` template and fill in any connection details you want the device to connect to.

- Open the prepped Arduino IDE and click on File » Open and select `client/client.ino` from the mTask repository.
- Click on Tools » Board » LOLIN (WEMOS) D1 & R2 mini.
- Click on Sketch » Upload to upload the RTS. It might be necessary to change Tools » Port to the correct port number before compiling and uploading.

A.4.4 Macro Definitions

For specializing the devices, the code adheres to several macro definitions for specifying certain things. First the platform is determined using the macro definitions listed in Table A.2. According to the platform the specified header is included.

Definition	Header File
`PC`	`pc.h`
`ARDUINO`	`arduino.h`

Table A.2: Platform macro definitions.

Table A.3 shows all other options for customizing the firmware.

A.4.5 Simulator

A standalone program can be generated for x86_x64 machines that simulates a device. This program is built from the same source as the actual RTS that is flashed on the microcontrollers. To compile this, run `make` in the `client` directory of the repository.

The `client` accepts two command line options. The `-h` command line option displays the help. By default the simulator listens to port number 8123, with the `-p` command line option, this can be changed.

Definition	Ard.	Pc	Description
MEMSIZE	✓	✓	Number of bytes to reserve for the stack and heap
SC(s)	✓	✓	String literal decoration, useful for storing string literals in the program memory.
APINS	✓	✓	Number of analog GPIO pins.
DPINS	✓	✓	Number of digital GPIO pins.
HAVE_DHT	✓	✓	DHT attached.
HAVE_LEDMATRIX	✓	✓	LED matrix attached.
HAVE_OLEDSHIELD	✓	✗	OLED shield attached[4].
LOGLEVEL	✓	✓	Level of logging, 0=none, 1=info, 2=debug.
BAUDRATE	✓	✗	Baudrate for the serial connection.
CURSES_INTERFACE	✗	✓	Compile with the experimental ncurses frontend.
REQUIRE_ALIGNED_MEMORY_ACCESS	✓	✓	Make sure all load and store instructions are aligned to the native word length (required for xtensa processors).

Table A.3: Feature macro definitions.

Appendix B

EDSL Techniques

An EDSL is a language embedded in a host language created for a specific domain [10]. EDSLs can have one or more backends or views. Commonly used views are pretty printing, compiling, simulating, verifying and proving the program. There are several techniques available for creating EDSLs. They all have their own advantages and disadvantages in terms of extendability, typedness and view support. In the following subsections each of the main techniques are briefly explained. An example expression DSL is used as a running example.

B.1 Deep Embedding

A deep EDSL is a language represented as data in the host language. Views are functions that transform *something* to the datatype or the other way around. Listing B.1 shows an example implementation for the expression DSL.

```
:: Expr
   = LitI  Int
   | LitB  Bool
   | Var   String
   | Plus  Expr Expr
   | Minus Expr Expr
   | And   Expr Expr
   | Eq    Expr
```

Listing B.1: A minimal deep EDSL.

Deep embedding has the advantage that it is easy to build and views are easy to add. To the downside, the expressions created with this language are not type-safe. In the given language it is possible to create an expression such as `Plus (LitI 4) (LitB True)` that adds a boolean to an integer. Evermore so, extending the Algebraic Data Type (ADT) is easy and convenient but extending the views accordingly is tedious and has to be done individually for all views.

The first downside of this type of EDSL can be overcome by using Generalized ADTs (GADTs) [7]. Listing B.2 shows the same language, but type-safe with a GADT. GADTs are not supported in the current version of Clean and therefore the syntax is hypothetical. However, it has been shown that GADTs can be simulated using bimaps or projection pairs [7]. Unfortunately the lack of extendability remains a problem. If a language construct is added, no compile time guarantee can be given that all views support it.

```
:: Expr a
   =     LitI  Int                    → Expr Int
```

```
    |      LitB  Bool                    → Expr Bool
    | ∃ e: Var   String                  → Expr e
    |      Plus  (Expr Int)  (Expr Int)  → Expr Int
    |      Minus (Expr Int)  (Expr Int)  → Expr Int
    |      And   (Expr Bool) (Expr Bool) → Expr Bool
    | ∃ e: Eq    (Expr e)    (Expr e)    → Expr Bool & == e
```

Listing B.2: A minimal deep EDSL using GADTs.

B.2 Shallow Embedding

In a shallow EDSL all language constructs are expressed as functions in the host language. An evaluator view for the example language then can be implemented as the code shown in listing B.3. Note that much of the internals of the language can be hidden using monads.

```
:: Env   = ...
           // Some environment
:: DSL a = DSL (Env → a)

Lit :: a → DSL a
Lit x = λe→x

Var :: String → DSL Int
Var i = λe→retrEnv e i

Plus :: (DSL Int) (DSL Int) → DSL Int
Plus x y = λe→x e + y e

Eq :: (DSL a) (DSL a) → DSL Bool | == a
Eq x y = λe→x e + y e
```

Listing B.3: A minimal shallow EDSL.

The advantage of shallowly embedding a language in a host language is its extendability. It is very easy to add functionality because the compile time checks of the host language guarantee whether or not the functionality is available when used. Moreover, the language is type safe as it is directly typed in the host language.

The downside of this method is extending the language with views. It is nearly impossible to add views to a shallowly embedded language. The only way of achieving this is by decorating the datatype for the EDSL with all the information for all the views. This will mean that every component will have to implement all views rendering it slow for multiple views and complex to implement.

B.3 Class Based Shallow Embedding

There are also some hybrid approaches that try to mitigate the downsides. The mTask language is using class-based — or tagless — shallow embedding that has both the advantages for shallow and deep embedding [6]. In class-based shallow embedding the language constructs are defined as type classes. This language is shown with the new method in listing B.4.

This type of embedding inherits the ease of adding views from shallow embedding. Just as with GADTs, type safety is guaranteed in deep embedding. Type constraints are enforced through phantom types. One can add as many phantom types as necessary. Lastly, extensions can be added easily, just as in shallow embedding. When an extension is made in an existing class, all views must be updated accordingly to prevent possible runtime errors. When an extension is

added in a new class, this problem does not arise and views can choose to implement only parts of the collection of classes.

```
:: Eval a = Eval a
runEval :: (Eval a) → a
runEval (Eval a) = a

:: PrettyPrinter a = PP String
runPrinter :: (PrettyPrinter t) → String
runPrinter (PrettyPrinter s) = s

class intArith where
   lit   :: t → v t              | toString t
   add   :: (v t) (v t) → (v t) | + t
   minus :: (v t) (v t) → (v t) | - t

class boolArith where
   and :: (v Bool) (v Bool) → (v Bool)
   eq  :: (v t)    (v t)    → (v Bool) | == t

instance intArith Evaluator where ...
instance intArith PrettyPrinter where
   lit x   = PP $ toString x
   add x y = PP $ x +++ "+" +++ y
   ...
instance boolArith Evaluator where ...
instance boolArith PrettyPrinter where ...
```

Listing B.4: A minimal class based shallow EDSL.

A downside is that the type errors can be pretty arcane but there are some techniques to mitigate this. Furthermore, when you want to use multiple backends for the same expression, rank-2 polymorphism is required in the language as seen in listing B.5.

```
printAndEval :: (∀ v: v t | intArith, boolArith v) → (String, t)
printAndEval c = (runPrinter c, runEval c)

Start :: (PP String, Bool)
Start = printAndEval (eq (lit 42) (lit 21 +. lit 21))
```

Listing B.5: Using multiple backends simultaneously in a shallow EDSL.

Appendix C

mTask Reference

The `mTask/library` directory contains all Clean files that form the mTask system. Table C.1 shows all directories and their explanations.

Directory	Description
Generics	type indexed helper functions
Interpret	bytecode interpretation backend
Language	mTask language classes
Show	pretty printing backend
Simulate	symbolic simulation backend
AST	C code generation backend (deprecated)

Table C.1: Directory overview

C.1 The mTask language (`Language.*`)

The core of the mTask system is the mTask language — a multibackend class based EDSL. The classes are type-constructor classes and therefore, a backend implementing a class is a type of the form `v t` where `v` is the actual backend. The phantom type `t` represents the type of the construction. The type `t` is often constrained with the `type` class collection to make sure only types suitable for MCUs can be used, e.g. only serializable and bounded.

C.1.1 Type restrictions

All types used in mTask programs must implement all classes in the `type` class collection. This collection contains serialization, printing, iTasks functions, etc. Many of these functions can be derived using generic programming. A subset of the types implementing `type` also implement basic types. Some types are basic types,

```
class type t | toString, iTask ... toByteCode{|*|} t

class basicType t | type t :: t
instance basicType Int, Bool, Real, Char, ()

:: Main a = {main :: a}
```

```
:: MTask v a = ...
:: In a b = In infix 0 a b

class mtask v | arith v & cond v & ...
```

Listing C.1: Type restrictions

C.1.2 Expressions

The classes for expressions — i.e. arithmetic functions, conditional expressions and tuples — are listed in Listing C.2. Some of the class members are oddly named (e.g. `+.`) to make sure there is no name conflict with Clean's builtin functions. There is no need for loop control due to support for tail-call optimized recursive functions and tasks. An exception to the pattern is the `lit` function, which allows lifting host language values to the mTask domain. For tuples there is a useful macro (`topen`) to convert a function with an mTask tuple as an argument to a function with a tuple of mTask values as an argument.

```
class arith v where
   lit :: t → v t | type t
   (+.) infixl 6 :: (v t) (v t) → v t | basicType, +, zero t
   (-.) infixl 6 :: (v t) (v t) → v t | basicType, -, zero t
   (*.) infixl 7 :: (v t) (v t) → v t | basicType, *, zero, one t
   (/.) infixl 7 :: (v t) (v t) → v t | basicType, /, zero t
   (&.) infixr 3 :: (v Bool) (v Bool) → v Bool
   (|.) infixr 2 :: (v Bool) (v Bool) → v Bool
   Not           :: (v Bool) → v Bool
   (==.) infix 4 :: (v a) (v a) → v Bool | Eq, basicType a
   (!=.) infix 4 :: (v a) (v a) → v Bool | Eq, basicType a
   (<.)  infix 4 :: (v a) (v a) → v Bool | Ord, basicType a
   (>.)  infix 4 :: (v a) (v a) → v Bool | Ord, basicType a
   (<=.) infix 4 :: (v a) (v a) → v Bool | Ord, basicType a
   (>=.) infix 4 :: (v a) (v a) → v Bool | Ord, basicType a
class cond v where
   If :: (v Bool) (v t) (v t) → v t | type t
   (?) infix 1 :: (v Bool) (v t) → MTask v () | type t & cond v
   (?) p t = If p t (lit ())
class tupl v where
   first  :: (v (a, b))  → v a      | type a & type b
   second :: (v (a, b))  → v b      | type a & type b
   tupl   :: (v a) (v b) → v (a, b) | type a & type b

   topen :: (v (a, b) → c) (v a, v b) → c
   topen f x :== f (first x, second x)
```

Listing C.2: The mTask classes for arithmetic, conditional and tuple expressions.

Functions

Functions are supported in the EDSL, albeit with some limitations. All user defined mTask functions are typed by Clean functions so that they are type-safe. All functions are defined using the multi-parameter typeclass `fun`. The first parameter (`a`) of the typeclass is the shape of the argument and the second parameter (`v`) is the backend (Listing C.3). Functions may only be defined at the top level and to constrain this, the `main` type is introduced to box a program.

One implementation for the `fun` class is defined for every arity. The listing gives example instances for for arities zero to two for backend `T`. Defining the different arities as tuples of arguments forbids the use of curried functions.

```
:: Main a = {main :: a}
:: In a b = In infix 0 a b
class fun a v where
   fun :: ((a → v s) → In (a → v s) (Main (v u))) → Main (v u) | type s & type
    ↪ u

:: T a // a backend
instance fun () T
instance fun (T a) T | type a
instance fun (T a, T b) T | type a & type b
```

Listing C.3: The mTask classes for functions definitions.

To demonstrate the use, Listing C.4 shows examples for many functions. The `type` constraint on the function arguments forbid the use of higher order functions. This Clean function will construct the program that will calculate the factorial of the given argument. In the bytecode backend, there is full tailcall optimization. It therefore loans to write tailcall optimized functions (see `factorial‘`).

```
increment :: Int → Main (v Int)
increment x =
   fun λinc=(λi→i +. lit 1) In
   {main=inc x}

sum :: Int Int → Main (v Int)
sum x y =
   fun λsum=(λ(l, r)→l +. r) In
   {main=sum (x, y)}

factorial :: Int → Main (MTask v Int) | mtask v
factorial x =
   fun λfac=(λi→
      If (i ==. lit zero)
         (lit one)
         (i *. fac (i -. lit one))) In
   {main=rtrn (fac (lit i))}

//Tail call optimized factorial
factorial‘ :: Int → Main (MTask v Int) | mtask v
factorial‘ x =
   fun λfacacc=(λ(n,a)→
      If (n ==. lit zero)
         a
         (facacc (n -. lit one, n*.a))) In
   fun λfac=(λi→
      facacc (i, lit one)) In
   {main=rtrn (fac (lit i))}
```

Listing C.4: Example functions.

C.1.3 Basic Tasks

Tasks are viewed as trees with leafs and forks. Basic tasks are the leafs and often represent a side effect such as hardware access. Task combinators are the forks and transform some tasks to a single transformed task.

Task values in mTask are represented by the same type as tasks in iTasks (Listing C.5). To lift a value in the expression domain to the task domain, the basic task `rtrn` is used. The resulting task will forever yield the given value as a stable task value.

```
class rtrn v where rtrn     :: (v t) → MTask v t | type t
```

Listing C.5: The mTask classes for basic tasks.

```
class rpeat v :: (MTask v a) → MTask v () | type a
class delay v :: (v n) → MTask v n | type, number n
```

Listing C.6: The mTask classes for delay and rpeat.

C.1.4 Parallel Task Combinators

In contrast to in iTasks, there are only two parallel combinators (Listing C.7). The listing shows an example of querying two pins at the same time, returning the one with the highest value.

```
class .&&. v where
   (.&&.) infixr 4 v :: (MTask v a) (MTask v b) → MTask v (a, b) | type a ...
class .||. v where
   (.||.) infixr 3 v :: (MTask v a) (MTask v a) → MTask v a | type a

twopins :: APin APin → Main (MTask v Int) | mtask v
twopins p1 p2 = {main=
        readA p1 .&&. readA p2
   >>∼. topen λ(l, r) → If (l >. r) (rtrn l) (rtrn r)}
```

Listing C.7: The mTask classes for parallel task combinators.

The conjunction combinator `.&&.` combines the task values to a tuple. The disjunction combinator `.||.` combines them into single task value, giving preference to the *most stable* value. The combinators first rewrite both sides and then merge the task values according to the semantics given as a Clean function in Listing C.8.

```
(.&&.) :: (TaskValue a) (TaskValue b) → TaskValue (a, b)
(.&&.) (Value lhs stab1) (Value rhs stab2) = Value (lhs, rhs) (stab1 && stab2)
(.&&.) _                 _                 = NoValue

(.||.) :: (TaskValue a) (TaskValue a) → TaskValue a
(.||.) lhs=:(Value _ True) _                    = lhs
(.||.) (Value lhs _)       rhs=:(Value _ True) = rhs
(.||.) NoValue             rhs                 = rhs
(.||.) lhs                 _                   = lhs
```

Listing C.8: Task value semantics for the parallel combinators.

C.1.5 Sequential Task Combinators

With sequential combinators, tasks are executed after each other. With parallel combinators, the tasks are executed at the same time and the resulting task values are combined.

The step combinator (`>>*.`) is the Swiss army knife of sequential combination (Listing C.9). The value that the left-hand side of the combinator yields is matched against all task continuations (`Step v t u`) on the right-hand side, i.e. the right-hand side tasks *observes* the task value. If one of the continuations yields a new task, the combined task continues with it, pruning the left-hand side. All other sequential combinators are derived from the step combinator as default instances but their implementation can be overridden. For example, the `>>=.` combinator is very similar to the monadic bind, it continues if and only if a stable value is yielded. The `>>∼.` combinator continues when any value, stable or unstable, is yielded. The `>>|.` and `>>..` combinators are variants that do not take the value into account of the aforementioned combinators.

```
class step v where
   (>>*.) infixl 1 :: (MTask v t) [Step v t u] → MTask v u | type u & type t

   (>>=.) infixl 0 :: (MTask v t) ((v t) → MTask v u) → MTask v u | ...
   (>>=.) m f = m >>*. [IfStable (λ_→lit True) f]
   (>>∼.) infixl 0 :: (MTask v t) ((v t) → MTask v u) → MTask v u | ...
   (>>∼.) m f = m >>*. [IfValue (λ_→lit True) f]
   (>>|.) infixl 0 :: (MTask v t) (MTask v u) → MTask v u | ...
   (>>|.) m f = m >>=. λ_→f
   (>>..) infixl 0 :: (MTask v t) (MTask v u) → MTask v u | ...
   (>>..) m f = m >>∼. λ_→f

:: Step v t u
  = IfValue    ((v t) → v Bool) ((v t) → MTask v u)
  | IfStable   ((v t) → v Bool) ((v t) → MTask v u)
  | IfUnstable ((v t) → v Bool) ((v t) → MTask v u)
  | IfNoValue                           (MTask v u)
  | Always                              (MTask v u)
```

Listing C.9: The mTask classes for sequential task combinators.

The following listing shows an example of a step in action. The `readPinBin` function will produce an mTask task that will classify the value of an analog pin into four bins. It also shows how the nature of embedding allows the host language to be used as a macro language.

```
readPinBin :: Main (MTask v Int) | mtask v
readPinBin = {main=readA A2 >>*.
   [  IfValue (λx→x <. lim) λ_→rtrn (lit bin)
   \\ lim<-[64,128,192,256]
   &  bin<-[0..]]
```

Listing C.10: An example task using sequential combinators.

C.1.6 Miscellaneous Combinators

Furthermore, there are miscellaneous combinators. For example, the `rpeat` function will forever execute the argument task. When the argument task is stable, it reinstates it and starts all over again. The `delay` task will wait for the specified amount of time. This task yields no value until the given time has elapsed, then it will yield the remaining time as a stable task value. To demonstrate them, the `blink` program is given that will turn the given pin on or off every 500 milliseconds. The functionality of `rpeat` can also be simulated using recursive functions as shown in the `blinkFun` task.

```
class rpeat v where
   rpeat :: (MTask v a) → MTask v () | type a
class delay v where
   delay :: (v Int) → MTask v t | type t

blink :: DPin → Main (MTask v ()) | mtask v
blink p = {main=rpeat (
        delay (lit 500) >>|. writeD (lit True) p
   >>|. delay (lit 500) >>|. writeD (lit False) p)}

blinkFun :: DPin → Main (MTask v Bool) | mtask v
blinkFun p =
   fun λblink=(
      λst→delay (lit 500) >>|. writeD st p >>=. blink o Not
   ) In {main=blink (lit True)}
```

Listing C.11: The mTask classes for repeat and delay.

C.1.7 Shared Data Sources

In mTask it is also possible to share data between tasks type safely using SDSs. Similar to functions, SDSs can only be defined at the top level. They are well-typed parts of the monadic state.

The `sds` class contains the function for defining and accessing SDSs. With the `sds` function, local SDSs can be defined. They are also typed by functions in the host language to assure type safety. The other functions in the class are for creating `get` and `set` tasks. The `getSds` returns a task that constantly emits the value of the SDS as an unstable task value. `setSds` writes the given value to the task and re-emits it as a stable task value when it is done.

Listing C.12 presents the definitions and an example. The artificial example shows a task that mirrors a pin value to another pin using an SDS.

```
:: Sds a
class sds v where
   sds    :: ((v (Sds t)) → In t (Main (MTask v u)))
      → Main (MTask v u) | type t & type u
   getSds :: (v (Sds t))       → MTask v t | type t
   setSds :: (v (Sds t)) (v t) → MTask v t | type t

localvar :: Main (MTask v ()) | mtask v
localvar = sds λx=42 In {main= rpeat (readA D13 >>∼. setSds x)
                         .||.  rpeat (getSds x >>∼. writeD D1)}
```

Listing C.12: The mTask classes for SDS tasks.

Lifted Shared Data Sources

The `liftsds` class is used to allow iTasks SDSs to be accessed from within mTask tasks. The function has a similar type as `sds` and creates an mTask SDS from an iTasks SDS so that it can be accessed using the class functions from the `sds` class. Listing C.13 shows an example of this where an iTasks SDS is used to control an LED on a device. When used, the server automatically notifies the device if the SDS is written to and vice versa. The `liftsds` class only makes sense in the context of actually executing backends. Therefore this class is excluded from the `mtask` class collection.

```
:: Shared a // an iTasks SDS
class liftsds v | sds v where
   liftsds :: ((v (Sds t)) → In (Shared t) (Main (MTask v u)))
      → Main (MTask v u) | type t & type u

lightSwitch :: (Shared Bool) → Main (MTask v ()) | mtask v & liftsds v
lightSwitch s = liftsds λx=s In {main=rpeat (getSds x >>∼. writeD D13)}
```

Listing C.13: The mTask class for lifting iTasks SDSs.

C.1.8 Peripherals

Interaction with the GPIO pins, and other peripherals for that matter, is also captured in basic tasks.

GPIO

For each type of pin, there is a read and a write task that, given the pin, will execute the action. The class for analog GPIO pin access is shown in Listing C.14. The `readA`/`readD` task constantly

yields the value of the analog pin as an unstable task value. The `writeA`/`writeD` writes the given value to the given pin once and returns the written value as a stable task value.

```
class aio v where
   readA  :: (v APin) → MTask v Int
   writeA :: (v APin) (v Int) → MTask v Int

class dio p v | pin p where
   readD :: (v p) → MTask v Bool
   writeD :: (v p) (v Bool) → MTask v Bool

:: Pin = AnalogPin APin | DigitalPin DPin
class pin p :: p → Pin | type p
instance pin APin, DPin
```

Listing C.14: The mTask classes for GPIO tasks.

Digital Humidity and Temperature Sensor

Several types of DHT sensors are supported and they can be used with functions from the `dht` class. Note that this class is not part of the `mtask` class collection and needs to be added as a separate constraint.

```
:: DHT
:: DHTtype = DHT11 | DHT21 | DHT22

//Hundredths of degreens
class dht v where
   DHT         :: p DHTtype ((v DHT)→Main (v b)) → Main (v b) | type p & pin p &
    ↪  type b
   temperature :: (v DHT) → MTask v Int
   humidity    :: (v DHT) → MTask v Int
```

Listing C.15: The mTask classes for DHT tasks.

LED Matrix

The 8×8 LED can be accessed using the function in the `LEDMatrix` class. Note that this class is not part of the `mtask` class collection and needs to be added as a separate constraint.

```
:: LEDMatrix = LEDMatrix Int

derive class gCons LEDMatrix
derive class iTask LEDMatrix

class LEDMatrix v where
   ledmatrix   :: DPin DPin ((v LEDMatrix) → Main (v b)) → Main (v b) | type b
   LMDot       :: (v LEDMatrix) (v Int) (v Int) (v Bool) → MTask v ()
   LMIntensity :: (v LEDMatrix) (v Int) → MTask v ()
   LMClear     :: (v LEDMatrix) → MTask v ()
   LMDisplay   :: (v LEDMatrix) → MTask v ()
```

Listing C.16: The mTask classes for LED Matrix tasks.

C.2 Pretty printing (`Show.*`)

In the pretty printing backend, tasks are printed to a list of strings. To run it one can just run the `showMain` function as shown in the following listing.

```
:: Show a
showMain :: (Main (Show a)) → [String] | type a

Start = showMain {main=rtrn (lit 42)}
```

Listing C.17: Pretty printing backend functionality

C.3 Bytecode Interpretation (`Interpret.*`)

In the bytecode backend, tasks are compiled, sent and executed during runtime of the iTasks server to a specified device. The supporting Clean functions for this are listed in Listing C.18.

The `withDevice` function offers access to the device with the given specification. The first argument of the function contains the information for maintaining a connection with the device that is of a type implementing the `channelSync`. As of now, the framework has instances for `channelSync` for types describing TCP connections, serial communication and a simulator as can be seen in Appendix C.3.1. The resulting task connects the device and ascertains that the connection is set up, kept up and closed down on completion. After the connection is set up, the second argument, the task doing something with a device, is executed.

Within the argument task — besides executing iTasks tasks — the `liftmTask` task can be used. This task lifts an mTask task to an iTasks task using the specified device. The `Bytecode` type implements the mTask classes and therefore, tasks will have the form `Main (MTask BCInterpret u ↪ )`. Under the hood, this functions runs the compiler, sends the generated bytecode, listens to messages from the device and watches the lifted SDSs. The task value of the mTask task is observable from iTasks because the task is now a regular iTasks task. Furthermore, lifted SDSs can be accessed and used for communication. In a traditional setting, all these things — such as communication, data sharing, task scheduling — have to be done by hand.

```
:: MTDevice //Abstract device representation
:: Channels //Communication channels

class channelSync a :: a (Shared Channels) → Task ()
withDevice :: a (MTDevice → Task b) → Task b | iTask b & channelSync, iTask a

liftmTask :: (Main (MTask BCInterpret u)) MTDevice → Task u | iTask, type u
```

Listing C.18: Integration with iTasks

C.3.1 Device types (`Interpret.Device.*`)

TCP (`Interpret.Device.TCP`)

The `TCPSettings` type houses the connection details for TCP connected mTask devices. The `host` and `port` field are fairly common and therefore it might be necessary to disambiguate the record (See Appendix D.12).

```
:: TCPSettings = {host :: String, port :: Int}
instance channelSync TCPSettings
```

Listing C.19: TCP Device Definition

Serial (`Interpret.Device.Serial`)

The `TTYSettings` type houses the connection details for the serial port connection.

```
:: ByteSize = BytesizeFive | BytesizeSix | BytesizeSeven | BytesizeEight
:: Parity = ParityNone | ParityOdd | ParityEven | ParitySpace | ParityMark
:: BaudRate = ... | B9600 | B19200 | B38400 | B57600 | B115200 | ...

:: TTYSettings = {
      devicePath :: String,
      baudrate :: BaudRate,
      bytesize :: ByteSize,
      parity :: Parity,
      stop2bits :: Bool,
      xonxoff :: Bool,
      //* Time in seconds to wait after opening the device.
      //* Set this to 2 if you want to connect to a borked arduino
      sleepTime :: Int
   }
instance channelSync TTYSettings
```

Listing C.20: Serial Device Definition

Appendix D

Clean Reference

This chapter is an adapted version of the chapter written by Peter Achten for the lecture notes of CEFP 2013 [2].

This chapter gives a brief overview of functional programming in Clean[1] [5]. Clean is a pure lazy functional programming language. It has many similarities to Haskell[2] .

This section contains a set of brief overviews of topics in Clean. These overviews should be short enough to read while studying other parts of this paper without loosing the flow of those parts. The somewhat experienced functional programmer is introduced to particular syntax or language constructs in Clean.

D.1 Cloogle

Cloogle is a search engine for Clean [17]. It searches all included libraries such as iTasks and Platform but also some extra libraries. As of now it does not index mTask (yet).

- The web frontend[4]

  This frontend can also be accessed using the `!cloogle` bang via DuckDuckGo Furthermore it is possible to share search results, browse the libraries[5] or browse the documentation[6].
- The `:Cloogle` command or `<LocalLeader>c` in vim-clean[7].
- An email to `query@cloogle.org` with the query in the subject[9]
- The `!query` command of the IRC bot `clooglebot`[10] which often resides on the `#cloogle` and `#cleanlang` channels on freenode[11].

Table D.1 shows all possible query types.

[1] `https://clean.cs.ru.nl`
[2] A one-page guide to Clean for Haskell programmers is also available here[3] [1]
[3] `http://www.mbsd.cs.ru.nl/publications/papers/2007/achp2007-CleanHaskellQuickGuide.pdf`
[4] `https://cloogle.org`
[5] `https://cloogle.org/src`
[6] `https://cloogle.org/doc`
[7] `https://gitlab.science.ru.nl/cstaps/vim-clean`[8]
[8] Note for Linux/MacOS users: vim-clean also contains IDE functions such as searching for definitions, switching between implementation and definition and much more
[9] `https://gitlab.science.ru.nl/cloogle/cloogle-mail`
[10] `https://gitlab.science.ru.nl/clean-cloogle/clean-irc`
[11] `irc://freenode.net/#cleanlang`

Query	Description
`hd`	Functions with a name like hd
`:: a [a] → a`	Functions with a type unifiable with `a [a] → a`
`hd :: [a] → a`	A combination of the above
`:: ∀ a: [a] → a`	Type search, where a cannot be unified
`.[]`	Information about the syntax construct `.[]`
`stack overflow`	Information about the error message "stack overflow"
`using Maybe ==;`	Anything that uses Maybe and ==
`exact Text`	Anything with the exact name Text
class `Text`	Classes with the exact name Text
`type Maybe`	Types with the exact name Maybe

Table D.1: Cloogle's different search types

D.2 Modules

A module with name M is represented physically by two text files that reside in the same directory: one with file name M.`dcl` and one with file name M.`icl`.

The M.`icl` file is the *implementation* module. It contains the (task) functions and data type definitions of the module. Its first line repeats its name:

```
implementation module M
```

An implementation module can always use its own definitions. By importing other modules, it can use the definitions that are made visible by those modules as well:

```
import M1, M2, ..., Mn
```

The M.`dcl` file is the *definition* module. It contains M's interface to other modules. The first line of a definition module also gives its name:

```
definition module M
```

A definition module basically serves two purposes.

- It exports identifiers of its own implementation module by repeating their signature. Hence, identifiers which signatures are not repeated are *cloaked* for other modules.
- It acts as a serving-hatch for identifiers that are exported by other modules by importing their module names. In this way you can create libraries of large collections of related identifiers.

D.3 Operators

Operators are binary (two arguments) functions that can be written in *infix* style (between its arguments) instead of the normal *prefix* style (before its arguments). Operators are used to increase readability of your programs. With an operator declaration you associate two other attributes as well. The first attribute is the *fixity* which indicates in which direction the binding power works in case of operators with the same precedence. It is expressed by one of the keywords **infixl**, **infix**, and **infixr**. The second attribute is its *precedence* which indicates the binding power of the operator. It is expressed as an integer value between 0 and 9, in which a higher value indicates a stronger binding power.

The snapshot below of common operators as defined in the host language Clean illustrates this.

```
class (==) infix  4 a :: !a !a → Bool
class (+)  infixl 6 a :: !a !a → a
class (-)  infixl 6 a :: !a !a → a
class (*)  infixl 7 a :: !a !a → a
class (/)  infixl 7 a :: !a !a → a
class (^)  infixr 8 a :: !a !a → a
```

(These operators are *overloaded* to allow you to instantiate them for your own types.) Due to the lower precedence of `==`, the expression x `+` y `==` y `+` x must be read as $(x$ `+` $y)$ `==` $(y$ `+` $x)$. Due to the fixities, the expression x `-` y `-` z must be read as $(x$ `-` $y)$ `-` z, and x `^` y `^` z as x `^` $(y$ `^` $z)$. In case of expressions that use operators of the same precedence but with conflicting fixities you must work out the correct order yourself using brackets `()`.

D.4 Guards

Pattern matching is an expressive way to perform case distinction in function alternatives, but it is limited to investigating the structure of function arguments. *Guards* extend this with conditional expressions. Here are two examples.

```
sign :: !Int → Int
sign 0  = 0
sign x
| x < 0 = -1
sign x  =  1
```

```
instance < Date where
   < x y
   | x.year <  y.year   = True
   | x.year == y.year
      | x.mon <  y.mon = True
      | x.mon == y.mon = x.day < y.day
      | otherwise      = False
   | otherwise          = False
```

In `sign`, the first alternative matches only if the argument evaluates to the value `0`. In that case, `sign` results in the value `0`. The second alternative imposes no pattern restrictions, but it does have a guard (`| x < 0`). Even though the pattern always matches, evaluation of the guard must result in `True` if the second alternative of `sign` is to be chosen. Therefor, the value `-1` is returned only if the argument is a negative number. Finally, the last alternative has neither a pattern restriction nor a guarded restriction, and therefor matches all remaining cases, which concern the positive numbers. In those cases, the result is `1`.

The implementation of `<` for `Date` values illustrates *nested* guards. In contrast with top-level guards, nested guards must be completed with `otherwise` to catch any remaining cases. The `otherwise` keyword can also be used in top-level guards, as is shown on the last line of the `<` function. The `<` function first checks the guard on line 3 and returns `True` if the first `year` field is smaller than the second `year` field. If the guard evaluates to `False`, then the second guard on line 4 is tested. In case of equal `year` field values, evaluation continues with the nested guards on lines 5–7 that inspect the `month` fields. If the first nested guard on line 5 evaluates to `True`, then the comparison also yields `True`. In case of a `False` result, the second nested guard on line 6 is tested. In case of equal `month` field values, the comparison of the `day` values provides the final answer. Finally, to complete the nested guards, the last case on line 7 concludes that the first argument is not smaller than the second, a conclusion that is shared by the last top-level guard on line 8.

D.5 Choice and pattern matching

Unlike most programming languages, in which an *if-then-else* construct is supported in the language, it can be straightforwardly incorporated as a function in a lazy functional language, using pattern matching as well. Let's examine the type and implementation of **if**:

```
if :: !Bool a a → a
if True then else = then
if _    _    else = else
```

The *type* tells you that the `Bool` argument is strict in **if**: it must always be evaluated in order to know whether its result is `True` or `False`. The *implementation* uses the evaluation strategy of the host language to make the choice effective. The **if** function has two alternatives, each indicated by repeating the function name and its arguments. Alternatives are examined in textual order, from top to bottom. Up until now the arguments of functions were only variables, but in fact they are *patterns*. A pattern p is one of the following.

- A *variable*, expressed by means of an identifier that starts with a lowercase character or simply the `_` wildcard symbol in case the variable is not used at all. A variable identifies and matches any computation without forcing evaluation. Within the same alternative, the variable identifiers must be different.
- A *constant* in the language, such as `0`, `False`, `3.14`, `'c'`, and `"hello, world"`. To match successfully, the argument is evaluated fully to determine whether it has exactly the same constant value.
- A *composite* pattern, which is either a *tuple* `(`p_1`, ... ,`p_n`)`, a *data constructor* `(`$d\ p_1 \ldots p_n$`)` where n is the arity of d, a *record* `{`f_1`=`p_1`, ... ,`f_n`=`p_n`}`, or a *list* `[`p_1`, ... ,`p_n`]` or `[`p_1`, ... ,`p_n `: `p_{n+1}`]`. Matching proceeds recursively to each part that is specified in the pattern. In case of records, only the mentioned record fields are matched. In case of lists, p_1 upto p_n are matched with the first n elements of the list, if present, and p_{n+1} with the remainder of the list.

Patterns control evaluation of arguments *until it is discovered that it either matches or not.* Only if all patterns in the same alternative match, computation proceeds with the corresponding right-hand side of that alternative; otherwise computation proceeds with the next alternative.

Hence, in the case of **if** its second argument is returned if the evaluation of the first argument results in `True`. If it results in `False` the second alternative is tried. Because it does not impose any restriction, and hence also causes no further evaluation, it matches, and the third argument is returned.

In `firstYearPossible` the data constructors are also matched from top to bottom. The last case always matches, and returns the value `0`.

D.6 List comprehensions

Lists are the workhorse of functional programming. *List comprehensions* allow you to concisely express list manipulations. Their simplest form is:

`[` e `\\` p `<-` g `]`

Generator g is an expression that is or yields a list. (Note that g can also evaluate to an array. In that case you need to use `<-:` instead of `<-` to extract array elements.) From the generator, values are extracted from the front to the back. Each value is matched with the pattern p. If this

succeeds, then the pattern variables in p are bound to the corresponding parts of the extracted value, and expression e, that typically uses these bound pattern variables, yields an element of the result list. If matching fails, then the next element of the generator is tried.

Besides the pattern p, elements can also be selected using a *guarded condition*:

```
[ e \\ p <- g | c ]
```

Here, c is a boolean expression that can use any of the pattern variables that are introduced at generator patterns to its left. For each extracted value from the sequence for which the pattern match succeeds, the guarded condition is evaluated. Only if the condition also evaluates to `True`, a list element is added.

It is possible to use several pattern-generator pairs `p <- g` in one list comprehension. They are combined either in *parallel* with the `&` symbol or as a *cartesian product* with the `,` symbol.

- In `p1 <- g1 & p2 <- g2`, values are extracted from g_1 and g_2 at the same index positions and matched against p_1 and p_2 respectively. The shortest generator determines termination of this value-extraction process.
- In `p1 <- g1 , p2 <- g2`, for each extracted value from g_1 that matches p_1 *all* values from g_2 are extracted and matched against p_2.

Each and every one of the above ways to manipulate lists is already very expressive. However, they can be combined in arbitrary ways. This can be daunting at times, but once you get used to the expressive power, list comprehensions often prove to be the best tool for list processing tasks.

D.7 Lambda abstractions

Lambda-abstractions $\lambda x \rightarrow e$ allow you to introduce *anonymous* functions 'on the spot'. They typically occur in situations where an ad hoc function is required, for which it does not make much sense to come up with a separate function definition. This frees you from thinking of a proper identifier and perhaps a type signature as well. The bind combinator `>>=` is an excellent example of such a situation because in general you need to give a name x to the task value of the first task, and want to give an expression e that uses x. If you weren't interested in x, you would have used the naïve then combinator `>>|` instead.

D.8 Modelling side-effects

In a pure functional programming language all results must be explicit function results. This implies that a changed state should also be a function result. The type of the `Start` function in an interactiev program is `*World → *World`, this indicates that it changes the world. There are two things worth noting at this moment:

- The basic type `World` is *annotated* with the *uniqueness attribute* `*`. In a function type any argument can be annotated with this attribute. This enforces the property that whenever the function is evaluated, it has the *sole reference* to the corresponding argument value. This is useful because it allows the function implementation to *destructively update* that value without compromising the semantics of the functional programming language. This can only be done if the function body itself does not violate this uniqueness property. This is checked statically.

- The basic type `World` represents the 'external' environment of a program. If the `Start` function has an argument, the language assumes that it is of type `World`. The language provides no other means to create a value of type `World`, so if an application is to do any interaction with the external environment, it must have a `Start` function with a uniquely attributed `World` argument.

Incorporating side-effects safely in a functional language has received a lot of attention in the functional language research community. For lazy functional languages a host of techniques has been proposed. Well-known examples are *monads*, *continuations*, and *streams*. For eager functional languages, the situation is less complicated because in these languages programs exhibit an execution order that is more predictable.

D.9 Signatures

A signature x `::` t declares that identifier x has type t. An identifier x starts with a lowercase or uppercase letter and has no whitespace characters. The type t can be either of the following forms.

- It is one of the basic types, which are: `Bool`, `Int`, `Real`, `Char`, `String`, `File`, and `World`.
- It is a type variable. Their identifiers start with a lowercase character.
- It is a composite type, using one of the language type constructors `[ ]`, `{ }`, `(,)`, and →.
  - If t is a type, then `[`t`]` is the *list-of-t* type.
  - If t is a type, then `{`t`}` is the *array-of-t* type.
  - If t_1 and t_2 are types, then `(`t_1`,`t_2`)` is the *tuple-of-t_1-and-t_2* type. This generalizes to t_1 upto t_n with $2 \leq n \leq 32$, separating each type by `,`. Hence, `(`t_1`,`t_2`,`t_3`)`, `(`t_1`,`t_2`,`t_3`,`t_4`)` and so on are also tuple types.
  - If t_1 and t_2 are types, then $t_1 \rightarrow t_2$ is the *function-of-t_1-to-t_2* type. This generalizes to $t_1 \ldots t_n \rightarrow t_{n+1}$, where $t_1 \ldots t_n$ are the argument types, and t_{n+1} is the result type. The function argument types are separated by whitespace characters. So, $t_1\ t_2 \rightarrow t_3$, $t_1\ t_2\ t_3 \rightarrow t_4$ and so on are also function types.
- It is a custom defined type, using either an algebraic type or a record type. Their type names are easily recognized because they always start with an uppercase character.

Signatures can be *overloaded*, in which case they are extended with one or more *overloading constraints*, resulting in x `::` t `|` $tc_1\ a_1$ `&` $\ldots$ `&` $tc_n\ a_n$. A constraint $tc_i\ a_i$ is a pair of a type class tc_i and a type variable a_i that must occur in t. Note that $tc_1\ a$ `&` $tc_2\ a$ `&` $\ldots$ `&` $tc_n\ a$ can be shorthanded to tc_1`,` tc_2`,` $\ldots$`,`$tc_n\ a$.

D.10 Overloading

Overloading is a common and useful concept in programming languages that allows you to use the same identifier for different, yet related, values or computations. In the host language Clean overloading is introduced in an explicit way: if you wish to reuse a certain identifier x, then you declare it via a *type class*:

class $x\ a_1\ \ldots\ a_n$ `::` t

with the following properties:

- the *type variables* $a_1 \ \ldots \ a_n$ $(n > 0)$ must be different and start with a lowercase character;
- the *type scheme* t can be any type that uses the type variables a_i.

This declaration introduces the type class x with the single *type class member* x. It is possible to declare a type class x with several type class members $x_1 \ldots x_k$:

class $x \ a_1 \ \ldots \ a_n$
 where $x_1 \ :: \ t_1$
 $\vdots$
 $x_k \ :: \ t_k$

It is customary, but not required, that in this case identifier x starts with an uppercase character. The identifiers x_i need to be different, and their types t_i can use any of the type variables a_i.

Type classes can be *instantiated* with concrete types. This must always be done for all of its type variables and all type class members. The general form of such an instantiation is:

instance $x \ t'_1 \ \ldots \ t'_n \mid tc_1 \ b_1$ `&` $\ldots$ `&` $tc_m \ b_m$
where ...

with the following properties:

- the types $t'_1 \ \ldots \ t'_n$ are substituted for the type variables $a_1 \ \ldots \ a_n$ of the type class x. They are not required to be different but they are not allowed to share type variables;
- the types t'_i can be overloaded themselves, in which case their type class constraints $tc_i \ b_i$ are enumerated after | (which is absent in case of no constraints). The type variable b_i must occur in one of the types t'_i;
- the **where** keyword is followed by implementations of all class member functions. Of course, these implementations must adhere to the types that result after substitution of the corresponding type schemes t_i.

D.11 Algebraic and existential types

Algebraic types allow you to introduce new constants in your program, and give them a type at the same time. The general format of an algebraic type declaration is:

:: $t \ a_1 \ \ldots a_m$ `=` $d_1 \ t_{11} \ \ldots t_{1c_1}$ `|` $\ldots$ `|` $d_n \ t_{n1} \ \ldots t_{nc_n}$

with the following properties:

- the type constructor t is an identifier that starts with an uppercase character;
- the type variables a_i $(0 \leq i \leq m)$ must be different and start with a lowercase character;
- the data constructors d_i $(1 \leq i \leq n)$ must be different and start with an uppercase character;
- the data constructors can have zero or more arguments. An argument is either one of the type variables a_i or a type that may use the type variables a_i.

From these properties it follows that all occurrences of type variables in data constructors (all right hand side declarations) must be accounted for in the type constructor (on the left hand side). With *existential quantification* it is possible to circumvent this: for each data constructor one can introduce type variables that are known only locally to the data constructor. A data constructor can be enhanced with such local type variables in the following way:

$$\exists\ b_1 \ldots b_k\ \texttt{:}\ d_i\ t_{i1} \ldots t_{ic_i}\ \texttt{\&}\ tc_1\ x_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ tc_l\ x_l$$

with the following properties:

- the type variables b_j $(0 \leq j \leq k)$ must be different and start with a lowercase character;
- the arguments of the data constructor d_i can now also use any of the existentially quantified type variables b_i;
- the pairs $tc\ x$ are type class constraints, in which tc indicates a type class and x is one of the existentially quantified type variables b_i.

From these properties it follows that it does not make sense to introduce an existentially quantified type variable in a data constructor without adding information how values of that type can be *used*. There are basically two ways of doing this. The first is to add functions of the same type that handle these encapsulated values (in a very similar way to methods in classes in object oriented programming). The second is to constrain the encapsulated type variables to type classes.

D.12 Record types

Record types are useful to create named collections of data. The parts of such a collection can be referred to by means of a field name. The general format of a record type declaration is:

$$\texttt{::}\ t\ a_1 \ldots a_m\ \texttt{=}\ \texttt{\{}\ r_1\ \texttt{::}\ t_1\texttt{,}\ \ldots\texttt{,}\ r_n\ \texttt{::}\ t_n\ \texttt{\}}$$

with the following properties:

- the type constructor t is an identifier that starts with an uppercase character;
- the type variables a_i $(0 \leq i \leq m)$ must be different and start with a lowercase character;
- the pairs r_i :: t_i $(1 \leq i \leq n)$ determine the components of the record type. The field names r_i must be different and start with a lowercase character. The types t_i can use the type variables a_i.

Just like algebraic types, record types can also introduce existentially quantified type variables on the right-hand side of the record type. However, unlike algebraic types, their use can not be constrained by means of type classes. Hence, if you need to access these encapsulated values afterwards, you need to include function components within the record type definition.

D.13 Disambiguating records

Within a program record field *names* are allowed to occur in several record types (the corresponding field *types* are allowed to be different). This helps you to choose proper field names, without worrying too much about their existence in other records. The consequence of this useful

feature is that once in a while you need to explicit about the record value that is created (in case of records with exactly the same set of record field names) and when using record field selectors (either in a pattern match or with the *.field* notation). Type constructor names are required to be unique within a program, hence they are used to disambiguate these cases.

- When creating a record value, you are obliged to give a value to each and every record field of that type. If a record has a field with a unique name, then it is clear which record type is intended. Only if two records have the same set of field names, you need to include the type constructor name t within the record value definition.

  $\ldots$ `{` t `|` f_1 `=` e_1`,` $\ldots$`,` f_n `=` e_n`}` $\ldots$

- If a record pattern has at least one field with a unique name, then it is clear which record type is intended. The record pattern is disambiguated by including the type constructor name t in the pattern in an analogous way as described above when creating a record value, except that you do not need to mention all record fields and that the right hand sides of the fields are patterns rather than expressions:

  $\ldots$ `{` t `|` f_1 `=` p_1`,` $\ldots$`,` f_n `=` p_n`}` $\ldots$

- If a record field selection $e.f$ uses a unique field name f, then it is clear which record type is intended. A record field selection can be disambiguated by including the type constructor name t as a field selector. Hence, $e.t.f$ states that field f of record type t must be used.

D.14 Record updates

Record values are defined by enumerating each and every record field, along with a value. If r is a record (or an expression that yields a record value), then a new record value can be created by specifying only what record fields are different. The general format of such a *record update* is:

`{` r `&` f_1 `=` e_1`,` $\ldots$`,` f_n `=` e_n`}`

This expression creates a new record value that is identical to r, except for the fields f_i that have values e_i $(0 < i \leq n)$ respectively. A record field should occur at most once in this expression.

D.15 Synonym types

Synonym types only introduce a new type constructor name for another type. The general formal of a type synonym declaration is:

`::` t' a_1 $\ldots a_n$ `:==` t

with the following properties:

- the type constructor t' is an identifier that starts with an uppercase character;
- the type variables a_i $(0 \leq i \leq n)$ must be different and start with a lowercase character;
- the type t can be any type that uses the type variables a_i. However, a synonym type is not allowed to be recursive, either directly or indirectly.

Synonym types are useful for documentation purposes of your model types. Although the name t' must be new, t' does not introduce a new type: it is completely exchangeable with any occurrence of t.

D.16 Strictness

In the signature of the basic task function `return` the first argument is provided with a *strictness annotation*, `!`. Recall that iTasks is embedded in Clean, which is a *lazy* language. In a lazy language, computation is driven by *the need to produce a result*. As a simple example, consider the function `const` that does nothing but return its first argument:

```
const x y = x
```

There is absolutely no need for `const` to evaluate argument `y` to a value. However, argument `x` is returned by `const`, so its evaluation better produces a result or otherwise `const x y` won't produce a result either.

The more general, and more technical, way of phrasing this is the following. Suppose we have a function f that has a formal argument x. Let e be a diverging computation (it either takes infinitely long or aborts without producing a result). If $(f\ e)$ also diverges, then argument x is said to be strict in f. Note that this is a property of the function, and not of the argument. In case of `const`, it is no problem that argument `y` might be a diverging computation because it is not needed by `const` to compute its result. The consequence is that with respect to termination properties, it does not matter if strict function arguments are evaluated *before* the function is called. In many cases, this increases the performance of the application because you do not need to maintain suspended computations (due to lazy evaluation), but instead can evaluate them to a result and use that instead.

The strictness property of function arguments is expressed in the function signature by prefixing the argument that is strict in that function with the `!` annotation. In case of `const`, its signature is:

```
const :: !a b → a
```

Appendix E

iTasks Reference

This chapter gives a brief overview of the iTasks library. It is by far extensive but should cover all iTasks constructions you will need for the assignments. Some examples from [18] can be found in Appendix E.6.

E.1 Types

The class collection `iTask` is used throughout the library to make sure the types used have all the required machinery for iTasks. This class collection contains only generic functions that can automatically be derived. Listing E.1 shows how to derive this class for a user defined type.

```
:: MyName =
   { firstName :: String
   , lastName  :: String
   }
derive class iTask MyName
```

Listing E.1: Derive the `iTask` class for a user defined type.

E.2 Editors

The most common basic tasks are editors for entering, viewing or update information. For the three basic editors there are three corresponding functions to create tasks as seen in Listing E.2.

```
enterInformation  :: d [EnterOption m]       → Task m | toPrompt d & iTask m
updateInformation :: d [UpdateOption m m] m  → Task m | toPrompt d & iTask m
viewInformation   :: d [ViewOption m]     m  → Task m | toPrompt d & iTask m
```

Listing E.2: The definitions of editors in iTasks.

The first argument of the function is something implementing `toPrompt`. There are `toPrompt` instances for at least `String` — for a description, `(String, String)` — for a title and a description and `()` — for no description.

The second argument is a list of options for modifying the editor behaviour. This list is either empty or contains exactly one item. The types for the options are shown in Listing E.3. Simple lenses are created using the `*As` constructor. If an entirely different editor must be used, the `*Using` constructors can be used.

```
:: ViewOption a  = ∃ v: ViewAs     (a → v)            & iTask v
                 | ∃ v: ViewUsing  (a → v) (Editor v) & iTask v
:: EnterOption a = ∃ v: EnterAs    (v → a)            & iTask v
                 | ∃ v: EnterUsing (v → a) (Editor v) & iTask v
:: UpdateOption a b
   = ∃ v: UpdateAs    (a → v) (a v → b)            & iTask v
   | ∃ v: UpdateUsing (a → v) (a v → b) (Editor v) & iTask v
```

Listing E.3: The definitions of editors in iTasks.

E.3 Task combinators

There are two flavours of task combinators, namely parallel and sequential. Both are based on super combinators, `step` and `parallel` respectively.

E.3.1 Parallel combinators

The two main parallel combinators are the conjunction and disjunction combinators shown in Listing E.4.

The `-&&-` has semantics similar to the mTask `.&&.` combinator. The `-||-` has the same semantics as the mTask `.||.` combinator. The `-||` and `||-` executes both tasks in parallel but only looks at the value of the left task or the right task respectively.

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iTask a & iTask b
(-||_) infixl 3 :: (Task a) (Task b) → Task a     | iTask a & iTask b
(_||-) infixr 3 :: (Task a) (Task b) → Task b     | iTask a & iTask b
(-||-) infixr 3 :: (Task a) (Task a) → Task a     | iTask a
```

Listing E.4: The definitions of parallel combinators in iTasks.

E.3.2 Sequential combinators

All sequential combinators are derived from the `>>*` combinator as shown in Listing E.5. With this combinator, the task value of the left-hand side can be observed and execution continues with the right-hand side if one of the continuations yields a `Just (Task b)`. The listing also shows many utility functions for defining task steps.

```
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)] → Task b | iTask a & iTask b
:: TaskCont a b
   = OnValue         ((TaskValue a) → Maybe b)
   | OnAction Action ((TaskValue a) → Maybe b)

:: Action = Action String //button

always       :: b                    (TaskValue a) → Maybe b
never        :: b                    (TaskValue a) → Maybe b
hasValue     :: (a → b)              (TaskValue a) → Maybe b
ifStable     :: (a → b)              (TaskValue a) → Maybe b
ifUnstable   :: (a → b)              (TaskValue a) → Maybe b
ifValue      :: (a → Bool) (a → b)   (TaskValue a) → Maybe b
ifCond       :: Bool b               (TaskValue a) → Maybe b
withoutValue :: (Maybe b)            (TaskValue a) → Maybe b
withValue    :: (a → Maybe b)        (TaskValue a) → Maybe b
withStable   :: (a → Maybe b)        (TaskValue a) → Maybe b
withUnstable :: (a → Maybe b)        (TaskValue a) → Maybe b
```

Listing E.5: The definitions of sequential combinators in iTasks.

Derived from the >>* combinator are all other sequential combinators such as the ones listed in Listing E.6 with their respective documentation.

```
// Combines two tasks sequentially. The first task is executed first.
// When it has a value the user may continue to the second task, which is executed
    ↪ with the result of the first task as parameter.
// If the first task becomes stable, the second task is started automatically.
(>>=) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b

// Combines two tasks sequentially but explicitly waits for user input to confirm
    ↪ the completion of
(>>!) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b

// Combines two tasks sequentially but continues only when the first task has a
    ↪ stable value.
(>>-) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b

// Combines two tasks sequentially but continues only when the first task has a
    ↪ stable value.
(>-|) infixl 1
(>-|) x y :== x >>- λ_ → y

// Combines two tasks sequentially but continues only when the first task has a
    ↪ value.
(>>~) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b

// Combines two tasks sequentially just as >>=, but the result of the second task
    ↪ is disregarded.
(>>^) infixl 1 :: (Task a) (Task b) → Task a| iTask a & iTask b

// Execute the list of tasks one after a¬her.
sequence :: [Task a] → Task [a] | iTask a
```

Listing E.6: The definitions of derived sequential combinators in iTasks.

E.4 Shared Data Sources

Data can be observer via task values but for unrelated tasks to share data, SDSs are used. There is an automatic publish subscribe system attached to the SDS system that makes sure tasks are only rewritten when activity has taken place in the SDS. There are many types of SDSs such as lenses, sources and combinators. As long as they implement the `RWShared` class collection, you can use them as a SDS. Listing E.7 shows two methods for creating a SDS, they both yield a `SimpleSDSLens` but they can be used by any task using a SDS.

```
sharedStore :: String a → SimpleSDSLens a | JSONEncode{|*|} a & JSONDecode{|*|} a
    ↪ & TC a
withShared  :: b ((SimpleSDSLens b) → Task a) → Task a | iTask a & iTask b
```

Listing E.7: The definitions for SDSs in iTasks.

With the `sharedStore` function, a named SDS can be created that acts as a well-typed global variable. `withShared` is used to create an anonymous local SDS.

There are four major operations that can be done on SDSs that are all atomic (see `lst:itaskssdstasks` ↪). `get` fetches the value from the SDS and yields it as a stable value. `set` writes the given value to the SDS and yields it as a stable value. `upd` applies an update function to the SDS and returns the written value as a stable value. `watch` continuously emits the value of the SDS as an unstable task value.

```
get :: (sds () a w) → Task a | iTask a & Readable sds & TC w
set :: a (sds () r a)  → Task a | iTask a & TC r & Writeable sds
upd :: (r → w) (sds () r w) → Task w | iTask r & iTask w & RWShared sds
watch :: (sds () r w) → Task r | iTask r & TC w & Readable, Registrable sds
```

Listing E.8: The definitions for SDS access tasks in iTasks.

For all editors, there are shared variants available as shown in Listing E.9. This allows a user to interact with the SDS.

```
updateSharedInformation :: d [UpdateOption r w] (sds () r w) → Task r | ...
viewSharedInformation   :: d [ViewOption r]     (sds () r w) → Task r | ...

sharedUpdate :: Task Int
sharedUpdate = withShared 42 λsharedInt →
        updateSharedInformation () [] sharedInt
   -||- updateSharedInformation () [] sharedInt
```

Listing E.9: The definitions for SDS editor tasks in iTasks.

E.5 Extra Task Combinators

Not all workflow patterns can be described using only the derived combinators. Therefore, some other task combinators have been invented that are not truly sequential nor truly parallel. Listing E.10 shows some combinators that might be useful in the assignments.

```
//Feed the result of one task as read-only shared to a¬her
(>&>) infixl 1 :: (Task a) ((SDSLens () (Maybe a) ()) → Task b) → Task b | iTask
    ↪  a & iTask b

// Sidestep combinator.
// This combinator has a similar signature as the >>* combinator, but instead of
    ↪ moving forward to a next step, the selected step is executed in parallel
    ↪ with the first task.
// When the chosen task step becomes stable, it is removed and the actions are
    ↪ enabled again.
(>^*) infixl 1 :: (Task a) [TaskCont a (Task b)] → Task a | iTask a & iTask b

// Apply a function on the task value while retaining stability
(@) infixl 1 :: (Task a) (a → b) → Task b
// Map the task value to a constant value while retaining stability
(@) infixl 1 :: (Task a) b → Task b

// Repeats a task indefinitely
forever :: (Task a) → Task a | iTasks a
```

Listing E.10: The definitions for hybrid combinators in iTasks.

E.6 Examples

.

E.6.1 Hello World

The entry point to all iTasks programs is the `doTasks` function. This function sets all machinery going and launches the web server. Listing E.11 shows a complete hello world program in iTasks.

```
module hello

import iTasks

Start w = doTasks hello w

hello :: Task MyName
hello = enterInformation "Enter your name" []
   >>= λx→viewInformation "Hello: " [] x
//Or it can be written in a point-free style:
// >>= viewInformation "Hello: " []
```

Listing E.11: The iTasks Hello World! program.

E.6.2 Task Patterns

Some workflow task patterns can easily be created using the builtin combinator as shown in Listing E.12.

```
maybeCancel :: String (Task a) → Task (Maybe a) | iTask a
maybeCancel panic t = t >>*
   [ OnValue (ifStable (return o Just))
   , OnAction (Action panic) (always (return Nothing))
   ]

:: Date //type from iTasks.Extensions.DateTime
currentDate :: SDSLens () Date () // Builtin SDS

waitForDate :: Date → Task Date
waitForDate d = viewSharedInformation ("Wait until" +++ toString d) [] currentDate
   >>* [OnValue (ifValue (λnow → date < now) return)]

deadlineWith :: Date a (Task a) → Task a | iTask a
deadlineWith d a t = t -||- (waitForDate d >>| return a)

reminder :: Date String → Task ()
reminder d m = waitForDate d >>| viewInformation ("Reminder: please " +++ m) [] ()
```

Listing E.12: The iTasks Hello World! program.

Bibliography

[1] Peter Achten. *Clean for Haskell98 Programmers*. 13th July 2007.

[2] Peter Achten, Pieter Koopman and Rinus Plasmeijer. 'An Introduction to Task Oriented Programming'. In: *Central European Functional Programming School*. Springer, 2015, pp. 187–245.

[3] Douglas Adams. *The Hitchhiker's Guide to the Galaxy Omnibus: A Trilogy in Four Parts*. Vol. 6. Pan Macmillan, 2017.

[4] Matheus Amazonas Cabral De Andrade. 'Developing Real Life, Task Oriented Applications for the Internet of Things'. Master's Thesis. Nijmegen: Radboud University, 2018. 60 pp.

[5] Tom Brus et al. 'Clean – a language for functional graph rewriting'. In: *Conference on Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 364–384.

[6] Jacques Carette, Oleg Kiselyov and Chung-Chieh Shan. 'Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages'. In: *Journal of Functional Programming* 19.5 (Sept. 2009), p. 509. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796809007205. URL: http://www.journals.cambridge.org/abstract_S0956796809007205 (visited on 15/01/2019).

[7] James Cheney and Ralf Hinze. *First-class phantom types*. Cornell University, 2003. URL: https://ecommons.cornell.edu/handle/1813/5614 (visited on 15/05/2017).

[8] Li Da Xu, Wu He and Shancang Li. 'Internet of things in industries: a survey'. In: *Industrial Informatics, IEEE Transactions on* 10.4 (2014), pp. 2233–2243.

[9] L. M. G. Feijs. 'Multi-tasking and Arduino : why and how?' In: *Design and semantics of form and movement. 8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013)*. Design and semantics of form and movement. 8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013). Ed. by L. L. Chen et al. Wuxi, China, 2013, pp. 119–127. ISBN: 978-90-386-3462-3.

[10] Patrick C. Hickey et al. 'Building embedded systems with embedded DSLs'. In: ACM Press, 2014, pp. 3–9. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628146. URL: http://dl.acm.org/citation.cfm?doid=2628136.2628146 (visited on 23/05/2017).

[11] Pieter Koopman, Mart Lubbers and Rinus Plasmeijer. 'A Task-Based DSL for Microcomputers'. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. the Real World Domain Specific Languages Workshop 2018. Vienna, Austria: ACM Press, 2018, pp. 1–11. ISBN: 978-1-4503-6355-6. DOI: 10.1145/3183895.3183902. URL: http://dl.acm.org/citation.cfm?doid=3183895.3183902 (visited on 14/01/2019).

[12] Mart Lubbers. 'Task Oriented Programming and the Internet of Things'. Master's Thesis. Nijmegen: Radboud University, 2017. 69 pp.

[13] Mart Lubbers, Pieter Koopman and Rinus Plasmeijer. 'Multitasking on Microcontrollers using Task Oriented Programming'. In: *2019 42st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. COnference on COmposability, COmprehensibility and COrrectness of Working Software. Opatija, Croatia: IEEE, 2019, pp. 1842–1846.

[14] Mart Lubbers, Pieter Koopman and Rinus Plasmeijer. 'Task Oriented Programming and the Internet of Things'. In: *Proceedings of the 30th Symposium on the Implementation and Application of Functional Programming Languages*. International Symposium on Implementation and Application of Functional Languages. Lowell, MA: ACM, 2018, p. 12. ISBN: 978-1-4503-7143-8. DOI: 10.1145/3310232.3310239.

[15] Rinus Plasmeijer, Peter Achten and Pieter Koopman. 'iTasks: executable specifications of interactive work flow systems for the web'. In: *ACM SIGPLAN Notices* 42.9 (2007), pp. 141–152.

[16] Rinus Plasmeijer and Pieter Koopman. 'A Shallow Embedded Type Safe Extendable DSL for the Arduino'. In: *Trends in Functional Programming*. Vol. 9547. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-39109-0 978-3-319-39110-6. DOI: 10.1007/978-3-319-39110-6. URL: http://link.springer.com/10.1007/978-3-319-39110-6 (visited on 22/02/2017).

[17] Camil Staps and Mart Lubbers. *The Clean language search engine*. 2017. URL: https://cloogle.org.

[18] Jurriën Stutterheim, Peter Achten and Rinus Plasmeijer. 'Maintaining Separation of Concerns Through Task Oriented Software Development'. In: *Trends in Functional Programming*. Ed. by Meng Wang and Scott Owens. Vol. 10788. Cham: Springer International Publishing, 2018, pp. 19–38. ISBN: 978-3-319-89718-9 978-3-319-89719-6. DOI: 10.1007/978-3-319-89719-6_2. URL: http://link.springer.com/10.1007/978-3-319-89719-6_2 (visited on 14/01/2019).

Glossary

Arduino is an ecosystem and a C++ dialect for many different types of Microcontroller Units.

AVR is an architecture for microprocessors used for example in the popular Arduino UNO.

C is an imperative low level system programming language.

C++ is an imperative and object oriented low level programming language compatible with C but supporting much more such as classes..

Clean Clean Language of East-Anglia and Nijmegen, a pure lazy functional programming language based on graph rewriting.

ESP8266 is a WiFi chip that can be used as a general purpose microcontroller as well.

Haskell Haskell, a pure lazy functional programming language.

iTasks is a Task Oriented Programming implementation hosted in the purely lazy functional programming language Clean.

LOLIN is a prototyping Microcontroller Unit based on the popular ESP8266 chip.

mTask is an Embedded Domain Specific Language for running tasks on Microcontroller Units.

NodeMCU is a prototyping Microcontroller Unit based on the popular ESP8266 chip.

xtensa is a configurable processor microprocessor core and architecture used for example on the popular ESP8266 chip.

ZigBee is an open standard for short-range wireless networking designed to complement WiFi and Bluetooth.

Acronyms

ADT Algebraic Data Type.

API Application Programming Interface.

ARDSL Arduino Domain Specific Language.

BT Bluetooth.

BTLE Bluetooth Low Energy.

CEFP Central European Functional Programming School.

cpm Clean Project Manager.

DHT Digital Humidity and Temperature sensor.

DSL Domain Specific Language.

EDSL Embedded Domain Specific Language.

GADT Generalized ADT.

GPIO General Purpose Input/Output.

I²C Inter-Intergrated Circuit.

IDE Integrated Development Environment.

IO Input/Output.

IOT Internet of Things.

IP Internet Protocol.

IRC Internet Relay Chat.

LEAN Language of East-Anglia and Nijmegen.

LED Lighting Emitting Diode.

MCU Microcontroller Unit.

OLED Organic Lighting Emitting Diode.

OS Operating System.

RFID Radio-frequency Identification.

RTS Run-time System.

SDS Shared Data Source.

TCP Transmission Control Protocol.

TOP Task Oriented Programming.

List of Assignments

List of Figures

List of Tables

List of Listings