Writing Internet of Things Applications with Task Oriented Programming

Mart Lubbers(✉) , Pieter Koopman , and Rinus Plasmeijer

Radboud University, Nijmegen, Netherlands
{mart,pieter,rinus}@cs.ru.nl

Abstract. The Internet of Things (IoT) is growing fast. In 2018, there was approximately one connected device per person on earth and the number has been growing ever since. The devices interact with the environment via different modalities at the same time using sensors and actuators making the programs parallel. Yet, writing this type of programs is difficult because the devices have little computation power and memory, the platforms are heterogeneous and the languages are low level. Task Oriented Programming (TOP) is a declarative programming language paradigm that is used to express coordination of work, collaboration of users and systems, the distribution of shared data and the human-computer interaction. The mTask language is a specialized, yet full-fledged, multi-backend TOP language for IoT devices. With the bytecode interpretation backend and the integration with iTask, tasks can be executed on the device dynamically. This means that—according to the current state of affairs—tasks can be tailor-made at run time, compiled to device-agnostic bytecode and shipped to the device for interpretation. Tasks sent to the device are fully integrated in iTask to allow every form of interaction with the tasks such as observation of the task value and interaction with Shared Data Sources (SDSs). The entire IoT application—both server and devices—are programmed in a single language, albeit using two embedded Domain Specific Languages (EDSLs).

Keywords: Task Oriented Programming · Interpretation · Functional Programming · Internet of Things

1 Introduction

1.1 Internet of Things

The IoT is growing rapidly and it is changing the way people and machines interact with the world. The term IoT was coined around 1999 to describe the communication of Radio-frequency Identification (RFID) devices. RFID became more and more popular the years after but the term IoT was not associated with it anymore. Years later, during the rise of novel network 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. At the time of writing, there is about one connected device per person in the world of which many are part of an IoT system. Gartner estimates that of these connected devices, there are about 5.8 billion IoT devices or endpoints connected[1]. They 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. While the number of devices seems to be growing exponentially fast, programming IoT applications is difficult. The devices are a large heterogeneous collection of different platforms, protocols and languages resulting in impedance problems.

The devices in IoT systems are equipped with various sensors and actuators. These range from external ones such as positioning, temperature and humidity to more internal ones like heartbeat and respiration [12]. When describing IoT systems, a layered architecture is often used to compartmentalize the technology. For the intents and purposes of this paper the four layer architecture defined by ITU-T (International Telecommunications Union - Telecommunication Standardization Sector) will be used as visualized in Fig. 1.
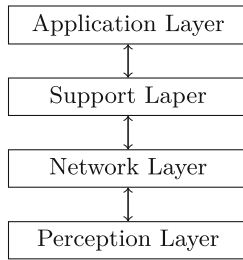

Fig. 1. The four layered IoT architecture as described by the ITU-T.

The first layer is called the perception layer and contains the actual endpoints with their peripherals. For example in home automation, the sensors reading the room and the actuators opening the curtains are in the perception layer. As a special type of device, it may also contain a Sensor Network (SN). A SN is a collection of sensors connected by a mesh network or central hub. The network layer is the second layer and it consists of the hardware and software to connect the perception layer to the world. In home automation, this layer may consist of a specialized IoT technology such as Bluetooth Low Energy (BLE) or ZigBee but it may also use existing technologies such as WiFi or wired connections. The third layer is named support 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 between the user and the

---

[1] Gartner (August 2019).

IoT system. In home automation, this layer contains the apps for to read the measurements and control the devices.

The perception layer often is a heterogeneous collections of microcontrollers, each having their own peculiarities, language of choice and hardware interfaces. 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 compiled firmware. This firmware is often written in an imperative language that needs to be flashed to the program memory. It is possible to dynamically send the program to the program memory using Over the Air (OTA) programming [6,7]. Program memory typically is flash based and only lasts a couple of thousand writes before it wears out[2]. While devices are getting a bit faster, smaller, and cheaper, they keep these properties to an extent. The properties of the device greatly reduce the flexibility for dynamic systems where tasks are created on the fly, executed on demand and require parallel execution. These problems can be mitigated by dynamically sending code to be interpreted to the MCU. With interpretation, a specialized interpreter is flashed in the program memory once that receives the program code to execute at runtime.

1.2   Task Oriented Programming

TOP is a declarative programming paradigm designed to model interactive systems [39]. A task is an abstract representation of a piece of work that needs to be done. It provides an intuitive abstraction over work in the real world. Just as with real-life tasks and workflow, tasks can be combined in various ways such as in parallel or in sequence. Furthermore, tasks are observable which means it is possible to observe a—partial—result during execution and act upon it by for example starting new tasks. Examples of tasks are filling in a form, sending an email, reading a sensor or even doing a physical task. The task itself abstracts away from implementation details such as the interface, the communication and the sharing of data.

In many implementations the value observable in a task is a three state value that adheres to the transition diagram seen in Fig. 2. If a task emits no value, it means that the task has not made sufficient progress to produce a complete value. It might be the case that some work has been done but just not quite 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 for filling in a form is an example of a task that always emits an unstable value since the contents may change over time. Stable values

---

[2] Atmel, the producer of AVR microprocessors, specifies the flash memory of the MCU in the Arduino UNO to about 10,000 cycles. This specification is a minimal specification and most likely the memory will be able to sustain many more writes. However, even if the memory can sustain ten times the amount, it is still a short time. .

never change. When the continue button has been pressed, the contents of the web editor is relayed, the values can never change, hence it is stable.

$$NoValue \longleftrightarrow Unstable \longrightarrow Stable$$

Fig. 2. State diagram for the legal transitions of task values

Tasks can communicate using task values but this imposes a problem in many collaboration patterns where tasks that are not necessarily related need to share data. Tasks can also share data using 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 external impure data such as the time, random numbers or sensory data. Similar to tasks, transformation and combination of SDSs is possible. In this architecture, tasks function as lightweight communicating threads.

1.3   iTask

The iTask system originated as a system for developing distributed collaborative interactive web applications and the TOP paradigm grew from it [37]. It is suitable to model collaboration in almost any domain (see Subsect. 5.2).

The iTask system is implemented as an EDSL hosted in Clean [9]. Compiling the embedded TOP specification results in a multi-user distributed webserver offering an interface to users for actually doing the work. By default, implementation details such as the graphical user interface, serialization and communication are automatically generated. Section B gives a non-comprehensive overview that is sufficient for the exercises and examples in this paper.

In iTask a task is implemented as an event-driven stateful rewrite function. This means that, when there is an event, the function is executed with the current state of the system and the event as arguments. As a result, it produces a new state and either a value or an exception. If a value is produced, it consists of a task value, an update to the user interface and a rewritten function. The current state of a task can be represented by the structure of the tasks and their combinators and is dubbed the task tree [29].

SDSs in iTask are based on Uniform Data Sources (UDSs). UDSs are a type safe, uniform and composable abstraction over arbitrary data through a read-/write interface [34]. This interface is extended with parametric lenses to also allow fine-grained control over accessing subsets of the data and filtering notifications [13]. Any type in the host language Clean is an SDS when it implements the `RWShared` class collection that contains the read, write and notification functions. The iTask library contains SDSs for storing data in files, databases, memory but also to provide access to system information such as date, time and random streams. Furthermore it contains combinators to apply all types of transformations to SDSs. Multiple SDSs can be combined to form new SDS,

SDSs modelling collections can be filtered, information of an SDS can determine the lens on another one and the data modelled by an SDS can be transformed.

Examples. Example 1 shows a simple example of an iTask application, more examples are available in Sect. B. In the application, the user can enter a family tree and when they are finished, view the result. The screenshots in Figs. 3 and 4 show this workflow. Lines 1 to 7 define the data types, `Family` and `Person` are record types with named fields and `Gender` is an algebraic data type. For any first order type, the necessary machinery housed in the iTask generic function collection can be derived automatically [3]. The collection contains functions for deserialization, serialization, editors, pretty printing and equality. Line 9 shows the derivation of the generic functions for the types in this example. The actual task is of type `Task Family` and shown at Line 11. The workflow consists of two tasks, the first task is for entering (Line 13) and the second one for viewings (Line 14). They are combined using a sequential task combinator (`>>=`) that results in a continue button being shown to the user. At the start of the workflow, the form is empty, and thus the continue button is disabled. When the user enters some information, the continue button enables when there is a complete value. However, the value may still change, as can be seen in the third figure when the partner tickbox is ticked and a recursive editor appears.

```
1   :: Family = { person    :: Person,  partner     :: Maybe Person
2               , children  :: [Family]
3               }
4   :: Person = { firstName :: String,  surName     :: String
5               , gender    :: Gender,  dateOfBirth :: Date
6               }
7   :: Gender = Male | Female | Other String
8
9   derive class iTask Family, Person, Gender
10
11  enterFamily :: Task Family
12  enterFamily
13    =        Hint "Enter a family tree:" @>> enterInformation []
14    >>= λres→Hint "You Entered:"         @>> viewInformation  [] res
```

Example 1. Source code for some example iTask tasks.

1.4   TOP for the IoT

IoT devices are often doing loosely related things in parallel. For example, they are reading sensors, doing some processing on the data, operating actuators and communicating with the world. The TOP paradigm is an intuitive description language for theses tasks. Furthermore, due to the execution semantics of tasks, seemingly parallel operation due to interleaving comes for free. Unfortunately, running iTask tasks on the device is not an option due to the high memory requirements of the software. Therefore, mTask has been created, a

Fig. 3. The initial user interface and the enabling of the continue button for the example application.

Fig. 4. The user interface after the user ticks the Partner box.

TOP language for small memory environments like IoT devices that also contains constructions to interact with the peripherals as well. It compiles the tasks to bytecode and sends them to the IoT device at run time. This allows the creation of dynamic applications, i.e. applications where tasks for the IoT devices are tailor-made at runtime and scheduled when needed.

1.5   Structure of the Paper

This section contains the introduction to IoT, TOP and iTask. The mTask ecosystem is explained in Sect. 2 followed by a language overview in Sect. 3. Section 4 contains gradually introduces more mTask concepts and provides a step by step tutorial for creating more interesting IoT applications. Section 5 contains the related work and Sect. 6 concludes with discussions. Background material on EDSL techniques is available in Sect. A. An iTask reference manual containing all the tasks and functions required for the exercises can be found in Sect. B and Sect. C contains detailed instructions on setting up an mTask development distribution.

Inline code snippets are typeset using a `teletype` font.

```
Program definitons are typeset in listings with a double left vertical border
```

Definition 1. This is an example definition.

```
Program examples are typeset in listings with a single left and bottom border
```

Example 2. This is an example example.

Exercise 0 (The title of the example exercise). Exercises are numbered and typeset like this. The filename of the skeleton—located in the distribution, see Sect. C—is typeset in teletype and placed between brackets (`fileName`).

2   mTask system architecture

2.1   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. The *Hello World* equivalent on microcontrollers blinks this LED.

Example 3 shows how the logic of a blink program might look when using the Arduino C++ dialect. The main event loop of the Arduino language continuously calls the user defined `loop` function. Blink's `loop` function alternates the state of the pin representing the LED between `HIGH` and `LOW`, turning the LED off and on respectively. In between it waits for 500 ms so that the blinking is actually

visible for the human eye. Compiling this results in a binary firmware that needs to be flashed onto the program memory.

Translating the traditional blink program to mTask can almost be done by simply substituting some syntax as seen in Example 4. E.g. `digitalWrite` becomes `writeD`, literals are prefixed with `lit` and the pin to blink is changed to represent the actual pin for the builtin LED of the device used in the exercises. In contrast to the imperative Arduino C++ dialect, mTask is a TOP language and therefore there is no such thing as a loop, only task combinators to combine tasks. To simulate this, the `rpeat` task can be used, this task executes the argument task and, when stable, reinstates it. The body of the `rpeat` contains similarly named tasks to write to the pins and to wait in between. The tasks are connected using the sequential `>>|.` combinator that for all intents and purposes executes the tasks after each other.

Exercise 1 (Blink the builtin LED). Compile and run the blink program to test your mTask setup (`blinkImp`). Instructions on how to install mTask and how to find the example code can be found in Sect. C.

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

Example (3) Blink in Arduino.

```
blink :: Main (MTask v ()) | mtask v
blink = {main = rpeat (
        writeD d2 (lit True)
    >>|. delay (lit 500)
    >>|. writeD d2 (lit False)
    >>|. delay (lit 500)
  )}
```

Example (4) Blink in mTask

2.2   Language

The mTask language is a TOP EDSL hosted in the pure lazy functional programming language Clean [9]. An EDSL is a language embedded in a host language created for a specific domain [23]. The two main techniques for embedding are deep embedding—representing the language as data—and shallow embedding—representing the languages as function. Depending on the embedding technique, EDSLs support one or multiple backends or views. Commonly used views are pretty printing, compiling, simulating, verifying and proving properties of the program. Deep and shallow embedding have their own advantages and disadvantages in terms of extendability, type safety and view support that are described in more detail in Sect. A.

2.3   Class Based Shallow Embedding

There are also some hybrid approaches that try to mitigate the downsides of the standard embedding techniques. The mTask language is using class-based—or tagless—shallow embedding that has both the advantages of shallow and deep embedding while keeping the disadvantages to a minimum [10]. This embedding technique is chosen because it allows adding backends and functionality orthogonally, i.e. without touching old code. E.g. adding functionality orthogonally is useful to add constructions for interact with new peripherals without requiring other backends to implement them. At the time of writing there is bytecode generation, symbolic simulation and pretty printing available as a backend.

Definition 2 shows an illustrative example of this embedding technique using a multi backend expression language. In class-based shallow embedding the language constructs are defined as type classes (`intArith` and `boolArith`). In contrast to regular shallow embedding, functions in class based shallow embedding are overloaded in the backend and in the types. Furthermore, the functions can be overloaded and contain class constraints, i.e. type safety is inherited from the host language. 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. But 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.

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

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

Definition 2. A minimal class based shallow EDSL.

A backend in a class based shallowly EDSL is just a type implementing some of the classes which makes adding backends relatively easy. It is even possible to create partial backends that do not support all classes from the language. The type of the backend are often—e.g. in the `PrettyPrinter` type—phantom types, only there to the resulting expression type safe. Example 5 shows an example of two backends implementing the expression Domain Specific Language (DSL).

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

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

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

instance intArith Evaluator where ...
instance boolArith Evaluator where ...
```

Example 5. A minimal class based shallow EDSL.

A downside of using classes instead of functions is that the more flexible implementation technique makes the type errors more complicated. Also, as a consequence of using classes instead of data, a program wanting to use the same expression twice has to play some tricks (see Example 2). If the language supports rank-2 polymorphism, it can use the same expression for multiple backends. Another solution is to create a combinator backend that combines the two argument backends in a single structure.

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

:: Two l r a = Two (l a) (r a)
printAndEval' :: (Two PrettyPrinter Evaluator t) → (String, t)
printAndEval' (Two (PP t) (Eval a)) = (t, a)

instance intArith (Two l r) | intArith l & intArith r where
  lit x = Two (lit x) (lit x)
  add (Two lx rx) (Two ly ry) = Two (add lx ly) (add rx ry)
instance boolArith (Two l r) | boolArith l & boolArith r where
  eq (Two lx rx) (Two ly ry) = Two (eq lx ly) (eq rx ry)
```

Example 6. Using multiple backends simultaneously in a shallow EDSL.

2.4   DSL design

To leverage the type checker of the host language, types in the mTask language are expressed as types in the host language, to make the language type safe. However, not all types in the host language are suitable for microcontrollers that may only have 2KiB of RAM so class constraints are therefore added to the EDSL functions (see Definition 3). The most used class constraint is the `type` class collection containing functions for serialization, printing, iTask constraints etc. Many of these functions can be derived using generic programming. An even stronger restriction on types is defined for types that have a stack representation. This `basicType` class has instances for many Clean basic types such as `Int`, `Real` and `Bool` but also for tuples. The class constraints for values in mTask are omnipresent in all functions and therefore often omitted throughout this paper for brevity and clarity.

Furthermore, expressions overloaded in backend add all mTask classes as constraints. To shorten this, a class collection is defined that contains all standard

mTask classes to relieve this strain. However, classes for peripherals—or other non standard classes that not all backends have—need to be added still.

```
class type t | iTask, ... ,fromByteCode, toByteCode t
class basicType t | type t where ...

class mtask v | arith, ..., cond v

someExpr :: v Int | mtask v
readTempClass :: v Bool | mtask, dht v
```

Definition 3. Classes and class collections for the mTask EDSL.

The mTask language is a TOP language and therefore supports tasks. For seamless integration, the `TaskValue` type from iTask is used for task values in mTask as well (see Definition 4). The leafs are basic tasks (i.e. editors) and the forks are task combinators. Every evaluation step, the task tree is traversed from the root up and nodes are rewritten while at the mean time keeping track of the task value of the tree as a whole. This means that there is a difference in execution between expressions and tasks. Expressions are always evaluated completely and therefore block the execution. Tasks on the other hand have small evaluation steps to allow seemingly parallel execution when interleaved.

```
:: TaskValue t = NoValue | Value a Bool //from iTasks
:: MTask v t :== v (TaskValue t)
```

Definition 4. The mTask task types.

2.5 Backends

The classes are just a description of the language. It is the backend that actually gives meaning to the language. There are many backends possible for a TOP programming language for tiny computers. At the time of writing, there is a pretty printing, symbolic simulation and bytecode generation backend. These lecture notes only regard the bytecode generation backend but the other backends will be briefly discussed for completeness sake.

Pretty Printer. The pretty printing backend produces a pretty printer for the given program. The only function exposed is the `showMain` (Definition 5) function which runs the pretty printer and returns a list of strings containing the pretty printed result as shown in Example 7. The pretty printing function does the best it can but obviously cannot reproduce the layout, curried functions and variable names.

```
:: Show a // from the mTask Show library
showMain :: (Main (Show a)) → [String] | type a
```

Definition 5. The entrypoint for the pretty printing backend.

```
blink :: Main (MTask v Bool) | mtask v
blink =
  fun λblink = (λstate→
          writeD d13 state
    >>|. delay (lit 500)
    >>=. blink o Not)
  In {main = blink true}

Start :: [String]
Start = showMain blink

// output:
// let f0 a1 = writeD(D13, a1) >>=λa2.(delay 1000) >>| (f0 (Not a1)) in (f0 True)
```

Example 7. Pretty printing backend example.

Simulator. The simulation backend produces a symbolic simulator embedded in iTask for the given program. When task resulting from the simulate function presents the user with an interactive simulation environment (see Definition 6, Example 8 and Fig. 5). From within the environment, tasks can be rewritten, peripheral states changed and SDSs interacted with.

```
:: TraceTask a // from the mTask Show library
simulate :: (Main (TraceTask a)) → [String] | type a
```

Definition 6. The entrypoint for the simulation backend.

```
Start :: *World → *World
Start w = doTasks (simulate blink) w
```

Example 8. Simulation backend example.

2.6   Bytecode

Programs written in mTask are compiled to bytecode to be integrated in iTask. The microcontroller stores the tasks in their RAM, leaving the program memory untouched. In TOP, it is not uncommon to create tasks every minute. Writing the program memory of an MCU every minute would wear a typical MCU out within a week (see footnote 2).

A complete specification of an mTask program—including the SDSs and peripherals—of type t has the following type in the host language: :: Main (MTask BCInterpret t). Under the hood, BCInterpret is a monad stack that generates the bytecode when executed. Interplay between mTask and iTask happens via three different constructions that are visualized in Fig. 6.
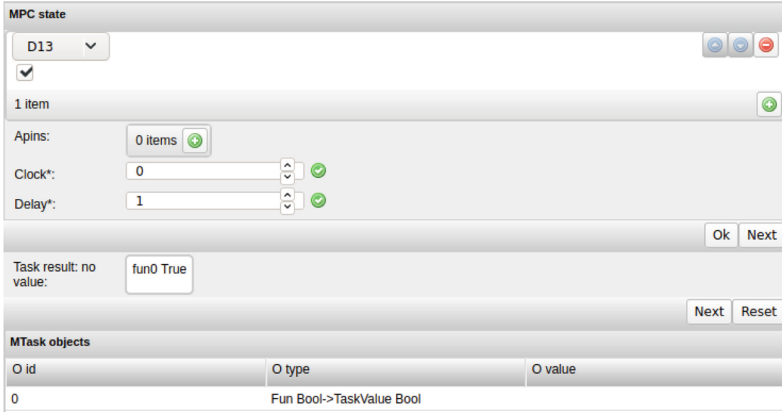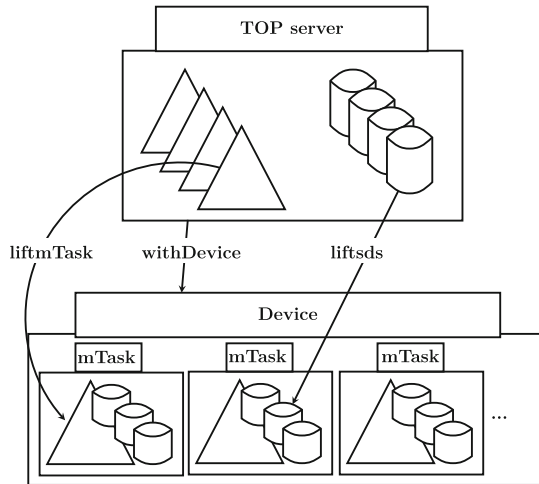
Fig. 5. Simulator interface for the blink program.

Fig. 6. The world of TOP applications that supports devices.

Connecting Devices. For a device to be suitable for mTask, it needs to be able to run the Run-time System (RTS). At the time of writing, the RTS is ported to Arduino compatible xtensa boards such as the LOLIN D1 Mini and NodeMCU, Arduino compatible AVR boards such as the Arduino UNO, and for platforms running OSs such as Linux, Windows or MacOS regardless of the architecture.

The `withDevice` function offers access to a specific device given the communication specification (see Definition 7). The first argument of the function contains the information about the connection that is used to communicate with the device. Any reliable sequential character based connection is suitable as a means of communication between the device and the server. In the mTask system

at the time of writing, `channelSync` instances are available for TCP connections
and serial port connections.

The second argument is a function that—given a device handle—produces
a task that can do something with the device. The task resulting from the
`withDevice` function will first setup a connection to the device and exchange
specifications. After the initialization, the task retrieved from the function in
the second argument is executed. When this task is finished, the connection
with the device is closed down again.

```
:: 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

instance channelSync TCPSettings, TTYSettings
```

Definition 7. Connecting mTask devices to an iTask server

Lifting Tasks. Sending a task to a device always occurs from within iTask
and is called *lifting* a task from mTask to iTask. The function for this is called
`liftmTask` (see Definition 8). The first argument is the mTask program and the
second argument is the device handle. The resulting task is an iTask proxy task
representing the exact state of the mTask task on the device.

Under the hood it first generates the bytecode of the mTask task by evalu-
ating the monad. This bytecode is bundled with metadata of the (lifted) SDSs
and peripherals and sent to the device. The device executes the task and notifies
the server on any changes in task value or when it writes a lifted SDS. These
changes are immediately reflected in the server resulting either in a changed
observable task value or a server side write to the SDS to which the lifted SDS
was connected. On the server side, the `liftmTask` task also subscribes to all lifted
SDS so that when the SDS on the server changes, the device can be notified as
well. The result is that this lifted task reflects the exact state of the mTask task.

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

Definition 8. Lifting an mTask to an iTask task.

2.7   Skeleton

Subsect. 2.1 showed an example mTask task that blinks the builtin LED. This
is not yet a complete Clean/iTask program that can be executed. A skeleton
follows that can be used as a basis for the exercises that is explained line by line.
Future snippets will again only give the mTask code for brevity.

```
1   module blink
2
3   import StdEnv, iTasks                 //iTasks imports
4   import Interpret, Interpret.Device.TCP //mTask imports
5
6   Start :: *World → *World
7   Start w = doTasks main w
8
9   main :: Task Bool
10  main = enterDevice >>= λspec→withDevice spec
11     λdev→liftmTask blink dev -|| viewDevice dev
12  where
13     blink :: Main (MTask v Bool) | mtask v
14     blink = ... //e.g. blink from Listing 4
```

Example 9. An mTask skeleton program.

Line 1 declares the name of the module, this has to match the name of the filename. Line 3 import `StdEnv` and `iTasks` libraries, these imports are required when using `iTasks`. Line 4 imports the `Interpret`—the mTask bytecode backend—and `Interpret.Device.TCP`—the TCP device connectivity modules. Both imports are always required for these exercises. Line 6 and 7 gives the `Start` function, the entry point for a Clean program. This start function always calls the iTask specific entry point called `doTasks` that starts up the iTask machinery and launches the task `main`.

The `main` task first starts with an editor on Line 10. This editor presents an interface to the user connecting to the server for it to select a device as seen in Fig. 7. The `enterDevice` task allows selecting devices from presets and allows changing the parameters to select a custom device. After entering the IP address the device shows, the task continues with connecting the device `withDevice` that takes a function requiring a device and resulting in a task. This function (Line 11) executes the `blink` task and shows some information about the device at the same time. Line 13 and 14 contain the actual task, for example the task shown in Example 9.
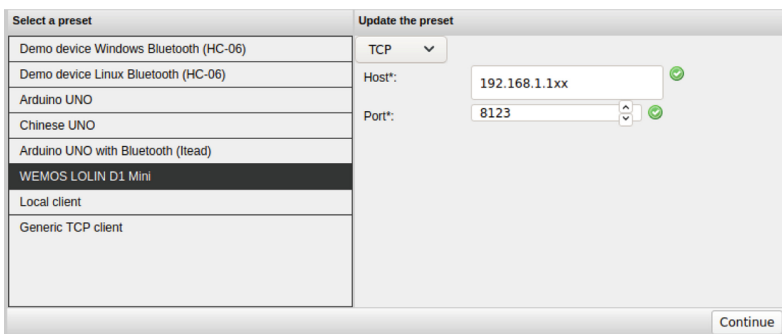

Fig. 7. The interface for the `enterDevice` task.

3   mTask language

3.1   Expressions

The classes for expressions—i.e. arithmetic functions, conditional expressions and tuples—are listed in Definition 9. Some of the class members are oddly named (e.g. +.) to make sure there is no name conflict with Clean's builtin overloaded functions that are of a different kind (∗ instead of ∗→∗). There is no need for loop control due to support for tail call optimized recursive functions and tasks. The types speak for themselves but there are a few functions to explain. The lit function lifts a value from the host language 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
  ...
  (==.) infix 4 :: (v a) (v a) → v Bool | basicType, == a
  ...
class cond v where
  If :: (v Bool) (v t) (v t) → v t | type t
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
```

Definition 9. The mTask classes for arithmetic, conditional and tuple expressions.

3.2   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 and are first class citizens in the DSL. They 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 (see Definition 10). Functions may only be defined at the top level and to constrain this, the main type is introduced to box a program.

```
:: 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) | ...
```

Definition 10. The mTask classes for functions definitions.

For every possible arity of the function, a separate implementation for the fun class has to be defined (see Example 10) The listing gives example instances for arities zero to two for backend T. Defining the different arities as tuples

of arguments instead of a more general definition forbids the use of curried functions. All functions are therefore known at compile time and when a function is called, all arguments are always known which is beneficial for keeping the memory requirements low.

```
:: 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
```

Example 10. Different class instances for different arities in mTask functions.

To demonstrate the use, Example 11 shows examples for two functions. The `type` constraint on the function arguments forbid the use of higher order functions because functions do not have instances for all classes of the collection. The functions (`sum`, `factorial`) constructs the program that calculates the result of the arguments. In the bytecode backend, there is full tailcall optimization and therefore, writing `factorial` as `factorial'` pays off in memory usage.

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

factorial :: Int → Main (v Int) | mtask v
factorial x =
  fun λfac = (λi→
    If (i ==. lit 0) (lit 1) (i *. fac (i -. lit 1)))
  In {main = fac (lit i)}

factorial' :: Int → Main (v Int) | mtask v
factorial' x =
  fun λfacacc = (λ(n,a) →
    If (n ==. lit 0) a (facacc (n -. lit 0, n *. a)))
  In fun λfac = (λi→
    facacc (i, lit 1))
  In {main = fac (lit i)}
```

Example 11. Example mTask functions.

Functional Blinking. The mTask blink implementation does not show the advantage of function or TOP. With functions, the blink behaviour can be lifted to a function to make the program more functional and composable (see Example 12). The function takes a single argument, the state and recursively calls itself. It creates an infinite task that first waits 500 ms. Then it will write the current state to the pin followed by a recursive call to with the inverse of the state.

```
blinkTask :: Main (MTask v Bool) | mtask v
blinkTask
  = fun λblink = (λx→
          delay (lit 500)
    >>|. writeD d2 x
    >>=. blink o Not)
  In {main = blink (lit True)}
```

Example 12. A functional mTask translatation of Hello World! (`blink`)

Exercise 2 (Blink the builtin LED with a different interval). Change the blinking interval of the functional blink program (`blink`).

3.3 Basic Tasks

Definition 11 shows the classes for the basic tasks in mTask. Interaction with peripherals also occurs through basic tasks and they are shown later. 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. The `rpeat` task continuously executes the argument task, restarting it when it yields a stable value. The resulting compound task itself never yields a value. The `delay` task emits no value while waiting for the elapsed number of milliseconds. When enough time elapsed, it returns the number of milliseconds that it overshot the target time as a stable value.

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

Definition 11. The mTask classes for basic tasks.

3.4 Parallel Task Combinators

Task combinators can be divided into two categories, namely parallel and sequential combinators. In parallel combination, the evaluation of the two tasks are interleaved, resulting in seemingly parallel execution. In contrast to iTask, there are only two parallel combinators available in mTask. Definition 12 shows the class definitions. Both combinators execute the two argument tasks in an interleaved fashion resulting in parallel execution.

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

Definition 12. The mTask classes for parallel task combinators and the rules for combining the value.

The resulting task value for the conjunction combinator .&&. is a pair of the task values of the children. The resulting task value for the disjunction combinator .||. is a single task value, giving preference to the most stable one. The exact task value production is explained as a Clean function in the listing below.

```
(.&&.) :: (TaskValue a) (TaskValue b) → TaskValue (a, b)
(.&&.) (Value l s1) (Value r s2) = Value (l, r) (s1 && s2)
(.&&.) _               _              = NoValue

(.||.) :: (TaskValue a) (TaskValue a) → TaskValue a
(.||.) (Value _ True) _            = Value l True
(.||.) (Value _ _)    (Value r True) = Value r True
(.||.) NoValue        r            = r
(.||.) l              _            = l
```

Definition 13. The rules for the task value of the parallel combinators.

When using the parallel combinator .&&. the result is something of type v (a, b). This means that it is a tuple in the mTask language and not in the host language and therefore pattern matching the tuple directly is not possible. For that, the topen macro is defined as can be seen in the listing together with an example of the usage.

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

firstPinToYield :: MTask v Int
firstPinToYield = readA A0 .||. readA A1 >>~. rtrn

sumpins :: MTask v Int
sumpins = readA A0 .&&. readA A1 >>~. topen λ(x, y)→rtrn (x +. y)
```

Example 13. An example of the usage of the parallel combinators.

3.5 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 Example 14.

```
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);
}
```

Example 14. Naive approach to multiple blinking patterns in Arduino C++.

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 [17]. Example 15 shows how to implement three different blinking patterns 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 difficult in general when the thread timings are determined at run time. Manual interleaving is very error prone, requires a lot of pointer juggling and generally results in 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);
}
```

Example 15. Threading three blinking patterns in Arduino.

Blinking multiple patterns in mTask is as simple as combining several calls to an adapted version of the `blink` function from Example 9 with a parallel combinator as shown in Example 16. The resulting task tree of a single blink function call can then be visualized as in Fig. 8.
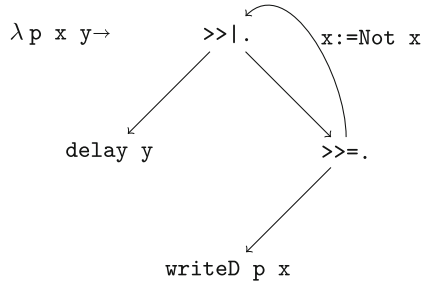
Fig. 8. The task tree for the blink task.

```
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)}
```

Example 16. An mTask program for blinking multple patterns. (`blinkThread`)

Exercise 3 (Blink the builtin LED with two patterns). Adapt the program in Example 16 so that it blinks the builtin LED with two different patterns concurrently. The times for the patterns are queried from the user.

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

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

You should `enterInformation` to get the information from the user (see Sect. B.2).

3.6   Sequential Task Combinators

The second way of combining tasks is sequential combination in which tasks are executed after each other. Similar to iTask, there is one Swiss army knife sequential combinator (`>>*.`) which is listed in Definition 14. The task value yielded by the left-hand side 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. When 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 class member instances. Their implementation can therefore be overridden to provide a more efficient implementation. For example, the `>>=.` combinator is very similar to the monadic bind, it continues if and only if a stable value is yielded with the task resulting from the function. 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 | ...

  (>>=.) 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)
```

Definition 14. 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..]]}
```

Example 17. An example task using sequential combinators.

3.7   Shared Data Source

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.

The `sds` class contains the function for defining and accessing SDSs. With the `sds` construction function, local SDSs can be defined that are 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.

Definition 18 and Example 15 present 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
```

Definition 15. The mTask class for SDS tasks.

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

Example 18. An example mTask task using SDSs.

3.8   Lifted Shared Data Sources

The `liftsds` class is defined to allow iTask SDSs to be accessed from within mTask tasks. The function has a similar type as `sds` and creates an mTask SDS from an iTask SDS so that it can be accessed using the class functions from the `sds` class. Definition 16 and Example 19 show an example of this where an iTask 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
```

Definition 16. The mTask class for iTask SDSs.

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

Example 19. An example mTask task using iTask SDSs.

3.9   Interactive Blinking

Example 17 showed that Clean can be used as a macro language for mTask, customizing the tasks using runtime values 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. Example 20 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 [] delayShare <<@ Title "Interval"
6   where
7     blink :: (Shared s Int) → Main (MTask v Bool) | mtask, liftsds v & RWShared s
8     blink delayShare =
9       liftsds λdelaysh=delayShare
10      In fun λblink = (λx→
11             writeD d2 x
12         >>|. getSds delaysh
13         >>~. delay
14         >>|. blink (Not x))
15      In {main = blink (lit True)}
```

Example 20. An mTask program for interactively changing the blinking frequency.
(`blinkInteractive`)

Line 3 shows the creation of the controlling iTask SDS using `withShared` (see
Sect. B.4).

Line 4 and 5 compromise the device function for `withDevice`. It lifts the `blink`
task to iTask and provides the user with an `updateSharedInformation` for the
delay SDS. The `blink` task itself is hardly modified. Line 9 lifts the SDS to an
mTask SDS using `liftsds` (see Subsect. 3.8). 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 12). The value this yields is
immediately fed to `delay`. The mTask machinery takes care of synchronising the
SDSs, when the user changes the delay, it is automatically reported to the device
as well.

Exercise 4 (Blink the builtin LED on demand). Adapt the program in Exam-
ple 20 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
```

3.10   Peripherals

Interaction with the General Purpose Input/Output (GPIO) pins, and other
peripherals for that matter, is also captured in basic tasks. Some peripherals
need initialization parameters and they are defined on the top level using host
language functions similar to SDSs and functions. Typically from tasks reading
peripherals such as sensors an unstable value can be observed.

General Purpose Input/Output. For each type of pin, there is a function that creates a task that—given the pin—either reads or writes the pin. The class for GPIO pin access is shown in Definition 17. 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. Note that the digital GPIO class is overloaded in the type of pin because analog pins can be used as digital ones as well.

```
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
```

Definition 17. The mTask classes for GPIO tasks.

Peripherals. All sensors have the same general structure in their classes and to illustrate this, the Digital Humidity and Temperature sensor (DHT) and LED matrix are shown. Using the `DHT` function, the device can be initialized with the correct parameters and used safely within the task. The `temperature` and `humidity` task respectively query the temperature and the relative humidity from the sensor and yield it as an unstable task value. This interface matches the C++ interface very closely but the semantics have been transformed to be suitable as a task. Note that this class is not part of the `mtask` class collection and needs to be added as a separate constraint. At the time of writing, mTask supports in a similar fashion DHTs, LED matrices, ambient light sensors, passive infrared sensors, sound level sensors and air quality sensors.

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

class dht v where
  DHT         :: p DHTtype ((v DHT) → Main (v b)) → Main (v b) | pin p & ...
  temperature :: (v DHT) → MTask v Real
  humidity    :: (v DHT) → MTask v Real
```

Definition 18. The mTask classes for the DHT.

```
:: 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 ()
```

<div align="center">

Definition 19. The mTask classes for the LED matrix.

</div>

4   IoT applications with TOP

The following subsections are a hands-on introduction to writing more complex applications in mTask and iTask. Both mTask and iTask are hosted in Clean which has a similar syntax to Haskell. Peter et al. provide a concise overview of the syntactical differences [1]. The skeletons for the exercises are listed between brackets and can be found in the `mTask/cefp19` directory of the distribution[3]. Section C contains detailed setup instructions. Solutions for all exercises are available in Sect. D.

4.1   Hardware and Client

For the examples we use the WEMOS LOLIN D1 mini[4] (Fig. 9). The D1 mini is an ESP8266 based prototyping board containing 1 analog and 11 digital GPIO pins and a micro USB connection for programming and debugging. It can be programmed using MicroPython, Arduino or LUA.

It is assumed that they are preinstalled with the mTask RTS and that it has the correct shields attached. Details on how to compile and run the mTask RTS on the device can be found in Sect. C.4.

The devices are installed on a three-way splitter and setup with an OLED, SHT and Matrix LED shield. The OLED shield is used for displaying runtime during operation. When booting up, it shows the WiFi status and when connected it shows the IP address that one should enter in the device selection screen of the server application. 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 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.

4.2   Temperature

Reading the ambient temperature off the device is achieved using the DHT sensor connected as a shield to the main board. The DHT shield contains an SHT30 sensor. When queried via I^2C, the chip measures the temperature with a $\pm 0.4\,°C$ accuracy and the relative humidity with a $\pm 2\%$ accuracy.

It is accessed using the mTask `dht` class (see Subsect. 3.10). For example, the following program will show the current temperature and humidity to the

---

[3] https://ftp.cs.ru.nl/Clean/CEFP19/.
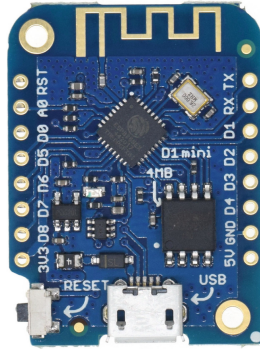[4] https://wiki.wemos.cc/products:d1:d1_mini.

Fig. 9. The mainboard of the WEMOS LOLIN D1 mini.

user. The yielded values from the `temperature` and `humidity` tasks are in tenths of degrees and percents respectively instead of a floating point value. 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}
```

Example 21. An mTask program for measuring the temperature and humidity. (`tempSimple`)

Exercise 5 (Show the temperature via an SDS). Modify the application so that it writes the temperature in an 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 Subsect. 3.10) 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.

Exercise 6 (Simple thermostat). Modify the previous exercise so that a thermo-
stat is mimicked. The user enters a temperature target and the LED will turn on
when the temperature is below the target. To quickly change the temperature
measure, blow some air in the sensor. 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
  ...
```
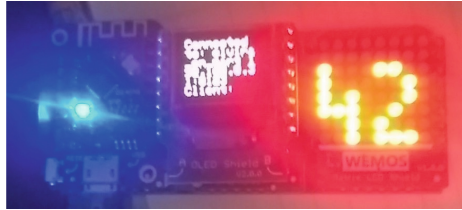
4.3   LED matrix

Fig. 10. *The Answer* printed on the LED matrix.

The LED matrix shield can be used to display information during the execu-
tion of the program. Every LED of the 8×8 matrix can be controlled individually
using the functions from Subsect. 3.10. The program in Example 22 shows an
iTask program for controlling the LED matrix. It allows toggling the state of a
given LED and clear the display.

To present the user with a nice interface (Fig. 11), a type is created that
houses the status of an LED in the matrix. The main program is very similar to
previous programs, only differing in the device part. The `>^*` combinator is a spe-
cial kind of `parallel` combinator that—instead of stepping to a continuation—
forks off a continuation. This allows the user 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.
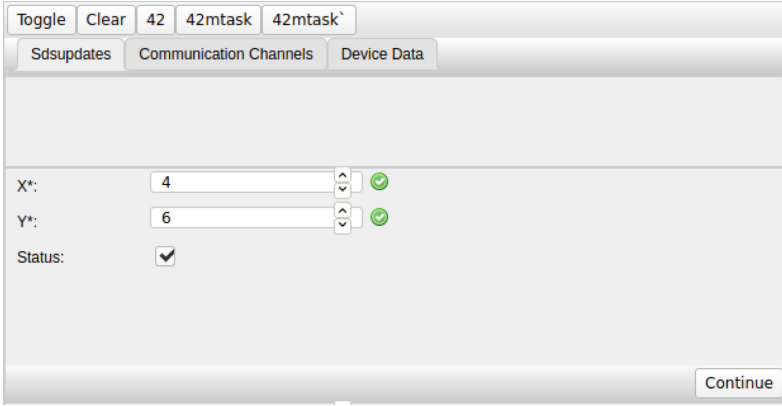
Fig. 11. The user interface for the LED matrix application

The `toggle` and `clear` tasks are self-explanatory and only use LED matrix mTask functions (see Definition 19).

```
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}
```

Example 22. 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 because for every action, a task needs to be launched. Extend the program so that there is a new button for printing the answer to the question of life, universe

and everything[5] as seen in Fig. 10. There are two general approaches possible that are presented in Assignment 7 and 8.

Exercise 7 (LED Matrix 42 using iTask). Write 42 to the LED matrix using only the `toggle` and the `clear` mTask tasks and define all other logic in iTask You can add the continuations as follows (`matrixBlink`):

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

The iTask task 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.

Exercise 8 (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:

```
mTask42 :: Main (MTask v ()) | mtask, LEDMatrix v
```
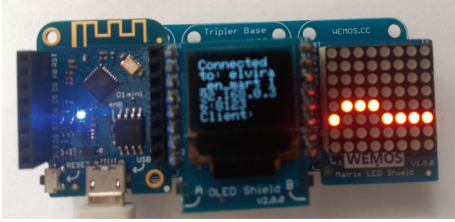
4.4   Temperature Plotter

This final exercise is about creating temperature plotter with an alarm mode. This application uses all components of the device and communicates with the server extensively. 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 12a shows an implementation in action. Figure 12b shows the user interface for it.

Exercise 9 (Temperature plotter). There are several tasks that the plotter needs to do at the same time

Plot The main task of the program is to plot the temperature over time on the LED matrix. The range of the graph is specified in the `limitsShare` and may be changed by the user.
Report The temperature has to be reported to the server every interval. This is achieved by writing the current temperature in the lifted `tempShare` SDS. The server is automatically notified and the user interface will update accordingly Preferably it only writes to the SDS when the temperature has changed.

---

[5] The exact question is left as an exercise to the reader but the answer is 42 [2].

(a) The temperature plotter in action.   (b) The temperature plotter UI.
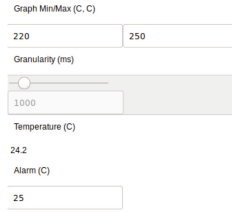
Fig. 12. The reference implementation of the plotter in action

Set alarm When the temperature is higher than a certain limit, the builtin
  LED should turn on. The current limit is always available in the lifted
  `alarmShare`.
Unset alarm When the alarm went off, the user should be able to disable it
  by pressing the A button that resides on the OLED shield.

  The exercise is quite elaborate so please keep in mind the following tips:

– 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 alternatively 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. Simplify it by either writing it as a Clean function that
  generates all the code or an mTask function that is called.

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

```
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 =
  ...
```

5   Related Work

The novelties of the mTask system can be compared to existing systems in several categories. It is an interpreted (Subsect. 5.1) TOP (Subsect. 5.2) language that may seem similar at first glance to Functional Reactive Programming (FRP) (Subsect. 5.3), it is implemented in a functional language (Subsect. 5.4) and due to the execution semantics, multithreading is automatically supported (Subsect. 5.5).

5.1   Interpretation

There are a myriad of interpreted programming languages available for some of the bigger devices. For example, for the popular ESP8266 chip there are ports of MicroPython, LUA, Basic, JavaScript and Lisp. All of these languages except the Lisp dialect uLisp (see Subsect. 5.4) are imperative and do not support multithreading out of the box. They lay pretty hefty constraints on the memory and as a result do not work on smaller MCUs. A interpretation solution for the tiniest devices is Firmata, a protocol for remotely controlling the MCU and using a server as the interpreter host [44]. Grebe et al. wrapped this in a remote monad for integration with Haskell that allowed imperative code to be interpreted on the MCUs [18]. Later this system was extended to support multithreading as well, stepping away from Firmata as the basis and using their own RTS [19]. It differs from our approach because continuation points need to be defined by hand there is no automatic safe data communication.

5.2   Task Oriented Programming

TOP as a paradigm with has been proven to be effective for implementing distributed, multi-user applications in many domains. Examples are conference

management [36], coastal protection [27], Command & Control (C2) [8], incident coordination [28], crisis management [24] and telemedicine [48]. In general, TOP results in a higher maintainability, a high separation of concerns and more effective handling of interruptions of workflow. IoT applications contain a distributed and multi-user component, but the software on the device is mostly follows multiple loosely dependent workflows A TOP language μTasks developed by Piers is specialized for embedded systems. It is a non-distributed TOP EDSL hosted in Haskell designed for embedded systems such as payment terminals [35]. They showed that applications tend to be able to cope well with interruptions and be more maintainable. However, the hardware requirements for running the standard Haskell system are high.

5.3   Functional Reactive Programming

The TOP paradigm is often compared to FRP and while they appear to be similar—they both process events—, in fact they are very different. FRP was introduced by Elliot and Hudak [15]. The paradigm strives to make modelling systems safer, more efficient, composable [5]. The core concepts are behaviours and events. A behaviour is a value that varies over time. Events are happenings in the real world and can trigger behaviours. Events and behaviours may be combined using combinators. Stutterheim et al. showed that FRP concepts such as events, behaviours and signal transformers can be expressed in TOP using tasks and SDSs as well [45].

The way FRP, and for that matter TOP, systems are programmed stays close to the design when the domain matches suits the paradigm. The IoT domain seems to suit this style of programming very well in just the device layer[6] but also for entire IoT systems.

For example, Potato is an FRP language for building entire IoT systems using powerful devices such as the Raspberry Pi leveraging the Erlang Virtual Machine (VM) [47]. It requires client devices to be able to run the Erlang VM which makes it unsuitable for low memory environments. The authors state that it should be possible to create lesser demanding node software using other languages such as C or Java but this is future work.

The emfrp language compiles a FRP specification for a microcontroller to C code [41]. The Input/Output (IO) part, the bodies of some functions, still need to be implemented. These IO functions can then be used as signals and combined as in any FRP language. Due to the compilation to C it is possible to run emfrp programs on tiny computers. However, the tasks are not interpreted and there is no communication with a server.

Juniper [21] and arduino-copilot [22] are FRP language for creating Arduino programs by compiling the specification to C++. The languages do not contain built-in interaction with the server nor do they support interpretation.

---

[6] While a bit out of scope, it deserves mention that for SN, FRP and stream based approaches are popular as well [46].

5.4   Functional Programming

Haenisch showed that there are major benefits to using functional languages for IoT applications. They showed that using function languages increased the security and maintainability of the applications [20]. Traditional implementations of general purpose functional languages have high memory requirements rendering them unusable for tiny computers. There have been many efforts to create a general purpose functional language that does fit in small memory environments, albeit with some concessions. For example, there has been a history of creating tiny Scheme implementations for specific microcontrollers. It started with BIT [14] that only required 64KiB of memory, followed by PICBIT [16] and PICOBIT [43] that lowered the memory requirements even more. More recently, Suchocki et al. created Microscheme, a functional language targeting Arduino compatible microcontrollers. The *BIT languages all compile to assembly while Microscheme compiles to C++, heavily supported by C++ lambdas available even on Arduino AVR targets. An interpreted Lisp implementation called uLisp also exists that runs on microcontrollers with as small as the Arduino UNO [25].

5.5   Multitasking

Applications for tiny computers are often parallel in nature. Tasks like reading sensors, watching input devices, operating actuators and maintaining communication are often loosly dependent on each other and are preferably executed in parallel. MCUs often do not benefit from an OS due to memory and processing constraints. Therefore, writing multitasking applications in an imperative language is possible but the tasks have to be interleaved by hand [17]. This results in hard to maintain, error prone and unscalable spaghetti code.

There are many solutions to overcome this problem in imperative languages.

If the host language is a functional language (e.g. the aforementioned Scheme variants) multitasking can be achieved without this burden relatively easy using continuation style multiprocessing [49]. Writing in this style is complicated and converting an existing program in this Continuation Passing Style (CPS) results in relatively large programs. Furthermore, there is no built-in thread-safe communication possible between the tasks. A TOP or FRP based language benefits even more because the programmer is not required to explicitly define continuation points.

Regular preemptive multithreading is too memory intensive for smaller microcontrollers and therefore not suitable. Manual interleaving of imperative code can be automated to certain extents. Solutions often require an Real-Time Operating System (RTOS), have a high memory requirement, do not support local variables, no thread-safe shared memory, no composition or no events as described in Table 1 adapted from Santanna et al. [40, p. 12]. The table compares the solutions in the relevant categories with mTask.

Table 1. An overview of imperative multithreading solutions for tiny computers with their relevant characteristics. The characteristics are: sequential execution, local variable support, parallel composition, deterministic execution, bounded execution and safe shared memory (Adapted from Santanna et al. [40, p. 12]).

Language		Complexity				Safety	
Name	Year	Seq. ex.	Loc. var.	Par. comp.	Det. ex.	Bound. ex.	Safe. mem.
Preemptive	many	✓	✓			rt	
nesC	2003				✓	async	
OSM	2005		✓	✓			
Protothreads	2006	✓			✓		
TinyThreads	2006	✓	✓		✓		
Sol	2007	✓	✓	✓	✓		
FlowTask	2011	✓	✓				
Ocram	2013	✓	✓		✓		
Céu	2013	✓	✓	✓	✓	✓	✓
mTask	2018	✓	✓	✓	✓	✓*[a]	✓[b]

[a] Only for tasks, not for expressions.
[b] Using SDSs.

5.6 mTask history

A first throw at a class-based shallowly EDSL for MCUs was made by Pieter Koopman and Rinus Plasmijer in 2016 [38]. The language was called Arduino Domain Specific Language (ARDSL) and offered a type safe interface to Arduino C++ dialect. A C++ code generation backend was available together with an iTask 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. The name then changed from ARDSL to mTask.

Mart Lubbers extended this in his Master's Thesis by adding integration with iTask and a bytecode compiler to the language [31]. SDS in mTask could be accessed on the iTask server. In this way, entire IoT systems could be programmed from a single source. However, this version used a simplified version of mTask without functions. This was later improved upon by creating a simplified interface where SDSs from iTask could be used in mTask and the other way around [32]. 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 [26]. This paper updated the language to support functions, tasks and SDSs but still compiled to C++ Arduino code. Later the bytecode compiler and iTask integration was added to the language [33]. Moreover, it was shown that it is very intuitive to write MCU applications in a TOP language [30]. One reason for this is that a lot of design patterns that are difficult using standard means are for free in TOP (e.g. multithreading). Furthermore, Erin van der Veen has been working on a green computer analysis and is working on support for bounded data types.

6   Discussion

These lecture notes give a complete introduction to the design and use of the mTask system. Furthermore it provides a hands-on tutorial for writing IoT applications with it.

The number of IoT devices is increasing evermore but programming them is as difficult and error-prone as it ever was. Most programs written for IoT devices are collections of loosely dependent parallel tasks which makes programming the devices using TOP very natural. The mTask language is a multi-backend device-agnostic TOP language specialized for IoT tasks. It contains a backend that will compile the program to bytecode that is then sent to the device. The backend is fully integrated in iTask which means that tasks that are sent to the device act as regular iTask tasks, i.e. their task value can be observed and they can interact with SDSs on the server. There is no impedance problem in the mTask ecosystem since all code is written in a single language, albeit in two EDSLs. The bytecode generation backend of mTask—and iTask for that matter—make heavy use of generic programming techniques to relieve the programmer of the burden to hand-craft specifics such as the user interface, the communication protocol or serialization. The execution semantics of the tasks makes them similar to lightweight threads—for which there is typically no support on microcomputers due to the lack of an OS. This allows programmers to create multitasking applications just by using parallel combinators. Reasonably complex IoT applications spanning all layers of IoT can be written in a concise and safe way using the mTask system.

Future work may be practical topics such as extending the number of supported platforms or extending the language with more features. For example, adding lenses and combinators to SDSs may improve the expressiveness of the language. Also, type errors in the DSL are presented to the programmer as type errors in the host language. As a result of class based shallow embedding, the type errors are quite complicated. It would be interesting to see whether techniques for mitigating this problem can be applied to mTask as well [42]. The execution model of the mTask system lets the server send arbitrary code to the device to be executed. This may pose a problem if the server, the communication technique is not to be trusted or can be snooped on. At the time of writing a student is working on analysing this problem in his thesis. Finally it would be interesting to allow the user instead of the programmer to write mTask tasks from scratch. This can be achieved by creating a type-safe editor in iTask that constructs tasks.

A   Embedded Domain Specific Language Techniques

An EDSL is a language embedded in a host language created for a specific domain [23]. 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, type safety 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.

A.1   Deep Embedding

A deeply 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. Definition 20 shows an example implementation for the expression DSL.

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

Definition 20. A deeply embedded expression DSL.

Deep embedding has the advantage that it is easy to build and views are easy to add. On the downside, the expressions created with this language are not necessarily 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. Extending the Algebraic Datatype (ADT) is easy and convenient but extending the views accordingly is tedious since it has to be done individually for all views.

The first downside of this type of EDSL can be overcome by using Generalized ADTs (GADTs) [11]. Example 21 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 [11]. 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
  =     Lit  a                     → Expr a
  |∃e: Var   String                → Expr e
  |    Plus  (Expr Int) (Expr Int) → Expr Int
  |∃e: Eq    (Expr e)   (Expr e)   → Expr Bool & == e
```

Definition 21. A deeply embedded expression DSL using GADTs.

A.2   Shallow Embedding

In a shallowly 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 Definition 22. Note that much of the internals of the language can be hidden using monads.

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

Definition 22. 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, i.e. `Lit True +. Lit 4` is rejected.

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 reimplementing all functions so that they run all backends at the same time. This will mean that every component will have to implement all views rendering it slow for multiple views and complex to implement.

B   iTask reference

This appendix gives a brief overview of iTask. It is by far extensive but should cover all iTask constructions required for the exercises. Some examples from [45] can be found in Sect. B.6.

B.1   Types

The class collection `iTask` is used throughout the library to make sure the types used have all the required machinery for iTask. This class collection contains only generic functions that can automatically be derived for any first order user defined type. Example 23 shows how to derive this class.

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

Example 23. Derive the `iTask` class for a user defined type.

B.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 Definition 23.

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

Definition 23. The definitions of editors in iTask.

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 Definition 24. 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
```

Definition 24. The definitions of editors in iTask.

Example 24 shows an example of such an editor using a lens. The user enters a temperature in degrees Celsius and the editor automatically converts the result to a temperature in Fahrenheit which is in turn the observed task value.

```
tempFahrenheit :: Task Real
tempFahrenheit = enterInformation "Enter the temperature in degrees Celsius"
  [EnterUsing λc→c*(9.0/5.0) + 32]
```

Example 24. An example of an editor that converts the entered value to a different unit in iTask.

B.3   Task Combinators

There are two flavours of task combinators, namely parallel and sequential that are all specializations of their Swiss-army knife combinator `step` and `parallel` respectively.

Parallel Combinators. The two main parallel combinators are the conjunction and disjunction combinators shown in Definition 25.

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
```

Definition 25. The definitions of parallel combinators in iTask.

Example 25 shows an example of a task that, using the disjunction combinator, asks the user for a temperature either in degrees Celsius or Fahrenheit using the task from Example 24. Whichever editor the user edits last, will be the observable task value.

```
askTemp :: Task Real
askTemp =   enterInformation "Temperature in Fahrenheit" []
       -||- tempFahrenheit
```

Example 25. An example of parallel task combinators in iTask.

Sequential Combinators. All sequential combinators are derived from the `>>*` combinator as shown in Definition 26. 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 & ...
:: 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
```

Definition 26. The definitions of sequential combinators in iTask.

Example 26 shows an example of the step combinator that forces the user to enter a number between 0 and 10. If the user enters a different value, the continue button will remain disabled.

```
numberBetween0and10 :: Task Int
numberBetween0and10 = enterInformation "Enter a number between 0 and 10" []
  >>* [OnAction (Action "Continue") $ ifValue (λi→i <= 10 && i >= 0) $ λi→return i]
```

Example 26. An example of parallel task combinators in iTask.

Derived from the >>* combinator are all other sequential combinators such as the ones listed in Definition 27 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 another.
sequence :: [Task a] → Task [a] | iTask a
```

Definition 27. The definitions of derived sequential combinators in iTask.

B.4   Shared Data Sources

Data can be observer via task values but for unrelated tasks to share data, SDSs are used. There is an publish subscribe system powering 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 an SDS. Definition 28 shows two methods for creating an SDS, they both yield a `SimpleSDSLens` but they can be used by any task using an SDS.

```
sharedStore :: String a → SimpleSDSLens a | iTask a
withShared  :: b ((SimpleSDSLens b) → Task a) → Task a | iTask a & iTask b
```
Definition 28. The definitions for SDSs in iTask.

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 Definition 29). `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. The implementation uses a publish subscribe system to evaluate the watch task only when the value of the SDS changes.

```
get   ::           (sds () r w) → Task r | iTask r & iTask w & RWShared sds
set   :: w         (sds () r w) → Task w | iTask r & iTask w & RWShared sds
upd   :: (r → w) (sds () r w) → Task w | iTask r & iTask w & RWShared sds
watch ::           (sds () r w) → Task r | iTask r & iTask w & RWShared sds
```
Definition 29. The definitions for SDS access tasks in iTask.

For all editors, there are shared variants available as shown in Definition 27. 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 | ...
```
Definition 30. The definitions for SDS editor tasks in iTask.

```
sharedUpdate :: Task Int
sharedUpdate = withShared 42 λsharedInt→
       updateSharedInformation "Left"  [] sharedInt
  -||- updateSharedInformation "Right" [] sharedInt
```
Example 27. An example of multiple tasks interacting with the same SDS in iTask.

B.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. Definition 31 shows some combinators that might be useful in the exercises.

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

Definition 31. The definitions for hybrid combinators in iTask.

B.6   Examples

Some workflow task patterns can easily be created using the builtin combinator
as shown in Examples 28.

```
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 (\now → 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) [] ()
```

Example 28. Some workflow task patterns.

C   How to Install

This section will give detailed instructions on how to install mTask on your
system. The distribution used also includes the example skeletons.

C.1   Fetch the CEFP distribution

Download the CEFP version of mTask distribution for your operating system as given in Table 2 and decompress the archive. The archives is all you need since it contains a complete clean distribution. The windows version contains an IDE and Clean Project Manager (`cpm`). Mac and Linux only have a project manager called `cpm`.

Table 2. Download links for the CEFP builds of mTask.

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

C.2   Setup

Linux. Assuming you uncompressed the archive in `~/mTask`, run the following commands in a terminal.

```
# Add the bin directory of the clean distribution to $PATH
echo 'export PATH=~/mTask/clean/bin:$PATH' >> ~/.bashrc
# Correctly set CLEANHOME
echo 'export CLEANHOME=~/mTask/clean' >> ~/.bashrc
# Source it for your current session
source ~/.bashrc
```

Windows. You do not need to setup anything on windows. However, if you want to use `cpm` as well, you need to add the `;C:\Users\frobnicator\mTask\clean` to your `%PATH%`[7].

MacOS. Assuming you uncompressed the archive in `~/mTask`, run the following commands in a terminal.

```
# Add the bin directory of the clean distribution to $PATH
echo 'export PATH=~/mTask/clean/bin:$PATH' >> ~/.bash_profile
# Correctly set CLEANHOME
echo 'export CLEANHOME=~/mTask/clean' >> ~/.bash_profile
# Source it for your current session
source ~/.bashrc
```

---

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

C.3   Compile the Test Program

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

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 .

`cpm` Enter the following commands in a command prompt or PowerShell session:

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

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 enter the following commands:

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

C.4   Setup the Microcontroller Unit

For setting up the RTS for the MCU, the reader is kindly referred to here[8].

D   Solutions

```
main :: Task Bool
main = enterDevice
  >>= λspec→enterInformation "Enter the intervals (ms)"
  >>= λ(i1, i2)→withDevice spec
    λdev→liftmTask (blink i1 i2) dev -|| viewDevice dev
where
```

---

[8] https://gitlab.science.ru.nl/mlubbers/mTask/blob/cefp19/DEVICES.md.

```
blink :: Int Int → Main (MTask v Bool) | mtask v
blink x y
  = fun λblink = (λ(p, x, y)→
        delay y
    >>|. writeD p x
    >>=. λx→blink (p, Not x, y))
  In {main = blink (d4, true, lit x)
        .||. blink (d4, true, lit y)}
```

Solution 3. Blink the builtin LED with two patterns

```
main :: Task Bool
main = enterDevice >>= λspec→withDevice spec
 λdev→withShared True λblinkOk→
        liftmTask (blink blinkOk) dev
    -|| updateSharedInformation "Blink Enabled" [] blinkOk
where
  blink :: (Shared s Bool) → Main (MTask v Bool) | mtask, liftsds v & RWShared s
  blink blinkShare = liftsds λblinkOk=blinkShare
    In fun λblink = (λx→
          writeD d2 x
      >>|. delay (lit 500)
      >>|. getSds blinkOk
      >>*. [IfValue (λx→x) (λ_→blink (Not x))])
    In {main = blink (lit True)}
```

Solution 4. Blink the builtin LED on demand

```
temp :: (Shared s Int) → Main (MTask v ()) | mtask, dht, liftsds v & RWShared s
temp tempShare =
  DHT D4 DHT22 λdht→
  liftsds λsTemp = tempShare
  In fun λmonitor = (λx→temperature dht
    >>*. [IfValue ((!=.)x) (setSds sTemp)]
    >>=. monitor)
  In {main = monitor (lit 0)}
```

Solution 5. Show the temperature via an SDS

```
temp :: (Shared s1 Int) (Shared s2 Int) → Main (MTask v ())
  | mtask, dht, liftsds v & RWShared s1 & RWShared s2
temp targetShare tempShare =
  DHT D4 DHT22 λdht→
    liftsds λsTemp = tempShare
  In liftsds λsTarget = targetShare
  In fun λmonitor = (λx→temperature dht
    >>*. [IfValue ((!=.)x) (setSds sTemp)]
    >>=. monitor)
  In fun λheater = (λst→getSds sTemp .&&. getSds sTarget
    >>*. [IfValue (tupopen λ(temp, target)→temp <. target &. Not st)
```

```
    λ_→writeD d4 (lit True)
      ,IfValue (tupopen λ(temp, target)→temp >. target &. st)
    λ_→writeD d4 (lit False)]
  >>=. heater)
In {main = monitor (lit 0) .||. heater (lit True)}
```

Solution 6. Simple thermostat

```
iTask42 :: MTDevice → Task ()
iTask42 dev = liftmTask clear dev
  >-| sequence [liftmTask (toggle {x=x,y=y,status=True}) dev\\(x,y)←fourtytwo] @! ()
        //Four
fourtytwo = [(0, 5), (0, 4), (0, 3), (0, 2) ,(1, 2), (2, 2), (2, 3) ,(2, 1), (2, 0)
        //Two
          ,(4, 5), (5, 5), (6, 4), (6, 3), (5, 2), (4, 1), (4, 0), (5, 0), (6, 0)]
```

Solution 7. LED Matrix 42 using iTask

```
mTask42 :: Main (MTask v ()) | mtask, LEDMatrix v
mTask42 = ledmatrix D5 D7 λlm→{main = LMClear lm >>|.
  foldr (>>|.) (LMDisplay lm) [dot lm {x=x, y=y, status=True} \\ (x,y) ← fourtytwo]}
```

Solution 8. LED Matrix 42 using mTask

```
temp :: (Shared s1 (Int, Int)) (Shared s2 Int) (Shared s3 Int) (Shared s4 Int)
  → Main (MTask v ()) | ...
temp limitsShare delayShare tempShare alarmShare =
  DHT D4 DHT22 λdht→
  ledmatrix D5 D7 λlm→
      liftsds λsLimits = limitsShare
  In liftsds λsDelay  = delayShare
  In liftsds λsTemp   = tempShare
  In liftsds λsAlarm  = alarmShare
  In fun λprint = (λ(targety, currentx, currenty)→
    If (currenty ==. lit 8)
      (LMDisplay lm)
      (LMDot lm currentx currenty (targety ==. currenty)
        >>|. print (targety, currentx, currenty +. lit 1)))
  In fun λmin = (λ(x, y)→If (x <. y) x y)
  In fun λcalcy = (λ(up, down, val)→
    min (down, (val -. down) /. ((up -. down) /. lit 7)))
  In fun λplot = (λx→
        getSds sLimits
    >>~. tupopen λ(gmin, gmax)→temperature dht
    >>~. λy→print (min (lit 7, calcy (gmin, gmax, y)), x, lit 0)
    >>|. setSds sTemp y
    >>|. getSds sDelay
    >>~. delay
    >>|. plot (If (x ==. lit 7) (lit 0) (x +. lit 1))
  )
```

```
In {main = plot (lit 0)
    .||. rpeat (readD BUILTIN_LED >>*. [IfValue Not (writeD ABUTTON o Not)])
    .||. rpeat (getSds sAlarm .&&. getSds sTemp
    >>*. [IfValue (tupopen λ(a, t)→t >. a) λ_→writeD ABUTTON (lit False)]
)}
```

Solution 9. Temperature plotter

References

1. Achten, P.: Clean for Haskell98 Programmers (2007)
2. Adams, D.: The Hitchhiker's Guide to the Galaxy Omnibus: A Trilogy in Four Parts, vol. 6. Pan Macmillan (2017)
3. Alimarine, A.: Generic Functional Programming. Ph.D., Radboud University, Nijmegen (2005)
4. Amazonas Cabral De Andrade, M.: Developing real life, task oriented applications for the internet of things. Master's thesis, Radboud University, Nijmegen (2018)
5. Amsden, E.: A survey of functional reactive programming. Technical report (2011)
6. Baccelli, E., et al.: Reprogramming low-end IoT devices from the cloud. In: 2018 3rd Cloudification of the Internet of Things (CIoT), pp. 1–6. IEEE (2018)
7. Baccelli, E., Doerr, J., Kikuchi, S., Padilla, F., Schleiser, K., Thomas, I.: Scripting over-the-air: towards containers on low-end devices in the internet of things. In: IEEE PerCom 2018 (2018)
8. Bolderheij, F., Jansen, J.M., Kool, A.A., Stutterheim, J.: A mission-driven C2 framework for enabling heterogeneous collaboration. In: Monsuur, H., Jansen, J.M., Marchal, F.J. (eds.) NL ARMS Netherlands Annual Review of Military Studies 2018. NA, pp. 107–130. T.M.C. Asser Press, The Hague (2018). https://doi.org/10.1007/978-94-6265-246-0_6
9. Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J.: Clean — a language for functional graph rewriting. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 364–384. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_20
10. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(05), 509 (2009). https://doi.org/10.1017/S0956796809007205
11. Cheney, J., Hinze, R.: First-class phantom types. Technical report, Cornell University (2003)
12. Da Xu, L., He, W., Li, S.: Internet of things in industries: a survey. IEEE Trans. Ind. Inform. **10**(4), 2233–2243 (2014)
13. Domoszlai, L., Lijnse, B., Plasmeijer, R.: Parametric lenses: change notification for bidirectional lenses. In: Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages, p. 9. ACM (2014)
14. Dubé, D.: BIT: a very compact Scheme system for embedded applications. In: Proceedings of the Fourth Workshop on Scheme and Functional Programming (2000)
15. Elliott, C., Hudak, P.: Functional reactive animation. In: ACM SIGPLAN Notices, vol. 32, pp. 263–273. ACM (1997)
16. Feeley, M., Dubé, D.: PICBIT: a scheme system for the PIC microcontroller. In: Proceedings of the Fourth Workshop on Scheme and Functional Programming, pp. 7–15. Citeseer (2003)

17. Feijs, L.: Multi-tasking and Arduino: why and how? In: Chen, L.L., et al. (eds.) Design and Semantics of form and Movement. 8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013), Wuxi, China, pp. 119–127 (2013)

18. Grebe, M., Gill, A.: Haskino: a remote monad for programming the arduino. In: Gavanelli, M., Reppy, J. (eds.) PADL 2016. LNCS, vol. 9585, pp. 153–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28228-2_10

19. Grebe, M., Gill, A.: Threading the Arduino with Haskell. In: Van Horn, D., Hughes, J. (eds.) TFP 2016. LNCS, vol. 10447, pp. 135–154. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14805-8_8

20. Haenisch, T.: A case study on using functional programming for internet of things applications. Athens J. Technol. Eng. **3**(1), 29–38 (2016)

21. Helbling, C., Guyer, S.Z.: Juniper: a functional reactive programming language for the Arduino. In: Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design, pp. 8–16. ACM (2016)

22. Hess, J.: Arduino-copilot: arduino programming in haskell using the Copilot stream DSL (2020). http://hackage.haskell.org/package/arduino-copilot

23. Hickey, P.C., Pike, L., Elliott, T., Bielman, J., Launchbury, J.: Building embedded systems with embedded DSLs. In: ACM SIGPLAN Notices, vol. 49, pp. 3–9. ACM Press (2014). https://doi.org/10.1145/2628136.2628146

24. Jansen, J.M., Lijnse, B., Plasmeijer, R.: Towards dynamic workflows for crisis management (2010)

25. Johnson-Davies, D.: Lisp for microcontrollers (2020). https://ulisp.com

26. Koopman, P., Lubbers, M., Plasmeijer, R.: A task-based DSL for microcomputers. In: Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL 2018, Vienna, Austria, pp. 1–11. ACM Press (2018). https://doi.org/10.1145/3183895.3183902

27. Lijnse, B., Jansen, J.M., Nanne, R., Plasmeijer, R.: Capturing the netherlands coast guard's sar workflow with itasks (2011)

28. Lijnse, B., Jansen, J.M., Plasmeijer, R., others: Incidone: a task-oriented incident coordination tool. In: Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM, vol. 12 (2012)

29. Lijnse, B., Plasmeijer, R.: iTasks 2: iTasks for end-users. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 36–54. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16478-1_3

30. Lubbers, M., Koopman, P., Plasmeijer, R.: Multitasking on microcontrollers using task oriented programming. In: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, pp. 1587–1592 (2019). https://doi.org/10.23919/MIPRO.2019.8756711

31. Lubbers, M.: Task oriented programming and the internet of things. Master's thesis, Radboud University, Nijmegen (2017)

32. Lubbers, M., Koopman, P., Plasmeijer, R.: Task oriented programming and the internet of things. In: Proceedings of the 30th Symposium on the Implementation and Application of Functional Programming Languages, Lowell, MA, p. 12. ACM (2018). https://doi.org/10.1145/3310232.3310239

33. Lubbers, M., Koopman, P., Plasmeijer, R.: Interpreting task oriented programs on tiny computers. In: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL 2019, Singapore, Singapore. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3412932.3412936

34. Michels, S., Plasmeijer, R.: Uniform data sources in a functional language, p. 16. Unpublished manuscript (2012)
35. Piers, J.: Task-oriented programming for developing non-distributed interruptible embedded systems. Master's thesis, Radboud University, Nijmegen (2016)
36. Plasmeijer, R., Achten, P.: A conference management system based on the iData Toolkit. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 108–125. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74130-5_7
37. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. ACM SIGPLAN Not. **42**(9), 141–152 (2007)
38. Koopman, P., Plasmeijer, R.: A shallow embedded type safe extendable DSL for the Arduino. In: Serrano, M., Hage, J. (eds.) TFP 2015. LNCS, vol. 9547, pp. 104–123. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39110-6_6
39. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, pp. 195–206. ACM (2012)
40. Sant'Anna, F., Rodriguez, N., Ierusalimschy, R., Landsiedel, O., Tsigas, P.: Safe system-level concurrency on resource-constrained nodes. In: Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems, p. 11. ACM (2013)
41. Sawada, K., Watanabe, T.: Emfrp: a functional reactive programming language for small-scale embedded systems. In: Companion Proceedings of the 15th International Conference on Modularity, pp. 36–44. ACM (2016)
42. Serrano, A.: Type error customization for embedded domain-specific languages. Ph.D. thesis, Utrecht University (2018)
43. St-Amour, V., Feeley, M.: PICOBIT: a compact scheme system for microcontrollers. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 1–17. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16478-1_1
44. Steiner, H.C.: Firmata: towards making microcontrollers act like extensions of the computer. In: NIME, pp. 125–130 (2009)
45. Stutterheim, J., Achten, P., Plasmeijer, R.: Maintaining separation of concerns through task oriented software development. In: Wang, M., Owens, S. (eds.) TFP 2017. LNCS, vol. 10788, pp. 19–38. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89719-6_2
46. Sugihara, R., Gupta, R.K.: Programming models for sensor networks: a survey. ACM Trans. Sensor Netw. **4**(2), 1–29 (2008). https://doi.org/10.1145/1340771.1340774
47. Troyer, de, C., Nicolay, J., Meuter, de, W.: Building IoT systems using distributed first-class reactive programming. In: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 185–192 (2018). https://doi.org/10.1109/CloudCom2018.2018.00045
48. van der Heijden, M., Lijnse, B., Lucas, P.J.F., Heijdra, Y.F., Schermer, T.R.J.: Managing COPD exacerbations with telemedicine. In: Peleg, M., Lavrač, N., Combi, C. (eds.) AIME 2011. LNCS (LNAI), vol. 6747, pp. 169–178. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22218-4_21
49. Wand, M.: Continuation-based multiprocessing. In: Proceedings of the 1980 ACM Conference on LISP and Functional Programming - LFP 1980, Stanford University, California, United States, pp. 19–28. ACM Press (1980). https://doi.org/10.1145/800087.802786