Task Oriented Programming and the Internet of Things

Mart Lubbers
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
mart@cs.ru.nl

Pieter Koopman
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
pieter@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

In the omnipresent Internet of Things (IoT), tiny devices sense and alter the environment, process information and communicate with the world. These devices have limited amounts of processing power and memory. This imposes severe restrictions on their software and communication protocols. As a result, applications are composed of parts written in various programming languages that communicate in many different ways. This impedance mismatch hampers development and maintenance.

In previous work we have shown how an IoT device can be programmed by defining an embedded Domain Specific Language (eDSL). This paper shows how IoT tasks can be seemlessly integrated with a Task Oriented Programming (TOP) server such as iTasks. It allows the specification on a high level of abstraction of arbitrary collaborations between human beings, large systems, and now also IoT devices. The implementation is made in three steps. First, there is an interface to connect devices dynamically to an iTasks server using various communication protocols. Next, we solve the communication problem between IoT devices and the server by porting Shared Data Sources (SDSs) from TOP. As a result, data can be shared, viewed and updated from the server or IoT device. Finally, we crack the maintenance problem by switching from generating fixed code for the IoT devices to dynamically shipping code. It makes it possible to run multiple tasks on an IoT device and to decide at runtime what tasks that should be.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Software and its engineering → Client-server architectures**; **Functional languages**; **Domain specific languages**.

KEYWORDS

Internet of Things, Functional Programming, Distributed Applications, Task Oriented Programming, Clean

IFL'18, August 2019, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310239

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Task Oriented Programming and the Internet of Things. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310239

1 INTRODUCTION

1.1 Internet of Things (IoT)

The term IoT stands for a whole network of (smart) devices that interact with each other and — most of all — interact with the world. IoT is booming, Gartner estimated that there will be around 21 billion IoT devices online in 2020[1]. IoT devices are already entering our households in the form of smart electricity meters, thermostats, weather stations, door locks and so on. IoT technology is emerging rapidly and is transforming the way people interact with technology and with each other.

There are typically severe limitations on the processing power of IoT devices. Because they must be cheap, very tiny computers are used with limited memory, e.g. Microcontroller Units (MCUs). The programs are usually stored in flash memory that only withstands a fairly limited number of write cycles. IoT devices communicate with the internet to share information such as sensor data and to act on demand with actuators all while using as little power, bandwidth, and memory as possible [Da Xu et al. 2014].

IoT not only encompasses the devices but all components of the system including the server, devices, and communication. There is an impedance mismatch between these which leads to isolated logic to integration problems. Every component has to be programmed separately in different languages and on different platforms resulting in high update roll-out costs. For example, rolling out updates in a device is relatively expensive since reprogramming MCUs in the field often requires physical access, while updating an app on the server is as simple as deploying an updated app.

Reprogramming devices automatically is beneficial when devices are often reprogrammed. It allows the creation of dynamic systems in which programs can be moved on demand between devices; e.g. in case of a failing device. In a compiled setting, deploying a different program on a device requires a complete reprogramming.

Interpretation can mitigate this limitation but comes with downsides as well. Sending serialized general purpose programs causes a big communication overhead and they have to be stored in the already scarce memory.

[1] Gartner (November 2015)

1.2 Task Oriented Programming (TOP)

The TOP paradigm and the corresponding iTasks implementation offer a high abstraction level for defining real world workflow tasks [Plasmeijer et al. 2007]. These tasks are described in an eDSL hosted in the purely functional programming language Clean [Brus et al. 1987; Plasmeijer et al. 2011]. Tasks are the basic building blocks of the language; they resemble actual work that needs to be done. The language contains combinators — arising from workflow modelling — to combine tasks in a sequential, parallel or conditional way. The iTasks system generates a multi-user web application to coordinate the tasks that the end users and the computer systems have to do in collaboration.

The iTasks eDSL is type-driven and built on generic functions that are created on the fly for the given types. These generic functions provide the basic TOP functionality which means the programmer has to do little to no implementation work on details such as the user interface. If needed, functions can be specialized to offer fine-grained control over this generated functionality.

1.3 Integrating IoT Devices with TOP

With TOP, one can describe arbitrary complex distributed collaborations between end users and systems without the need to worry about the technical details. In this paper we show how to program all layers of IoT from one single source and thus incorporate IoT devices seamlessly in TOP/iTasks. Adding IoT devices to the current iTasks system is difficult as it was not designed to cope with devices which are that tiny and that restricted in their communication bandwidth.

A natural way of adding clients to a server in iTasks is to use the distributed extension. Oortgiese et al. lifted iTasks from a single server model to a distributed server architecture[Oortgiese et al. 2017]. For example, Android apps can be created that run an entire iTasks core and are able to receive tasks from a different server and execute them. Although their system is suitable for dynamically sending tasks over platforms with different types of processors, their solution cannot be ported to MCUs because they are simply not powerful enough to run or store an iTasks core. Devices that run Android are still a lot more powerful than the typical IoT MCU. Moreover, sending serialized iTasks tasks over an Low Power Low Throughput Network (LTN) requires too much bandwidth.

1.4 Research Contribution

In this paper we present a novel way of controlling IoT devices in TOP/iTasks using restricted tasks for IoT devices and special interfaces to SDSs. It presents the following research contributions.

(1) Extensions for the mTask-eDSL by Koopman and Plasmeijer [2016] are given to create a language for describing imperative IoT tasks. (2) The novel backend for the eDSL generates specialized bytecode programs. (3) A Runtime System (RTS) is shown for the devices that can dynamically receive and execute this bytecode so that can be repurposed without reprogramming. The RTS is modular just as the eDSL and easily portable. (4) A method for integrating the devices with a TOP server is shown by giving an implementation in iTasks. With this glue, IoT tasks can be executed as if they were regular TOP tasks and communication with these tasks is transparently achieved via SDSs. Device and communication specific information is hidden for the programmer and user.

1.5 Structure of this Paper

In Section 2, the basic concepts of TOP as offered by the iTasks system are introduced together with an IoT application that is used as a running example. Section 3 explains the eDSL techniques and the actual eDSL used to express IoT tasks. The glue needed for the interaction between iTasks and IoT devices is discussed in Section 4. Moreover, the example from Section 2 is finished to illustrate the process of building IoT applications. Section 5 shows the bytecode compilation backend for the eDSL to dynamically generate code that can be executed on IoT devices. Section 6 covers the details of the run-time system for the IoT devices. In Section 7, the server-side implementation and integration is discussed, Section 8 describes related work and Sections 9 and 10 conclude with the conclusion and discussion.

2 BRIEF OVERVIEW OF TOP

Here we present a brief overview of the main concepts of TOP. The details are specific to the TOP implementation iTasks. More detailed information can be found in [Plasmeijer et al. 2012].

2.1 Tasks

A task is a statefull event processor that returns a value of type `:: TaskValue a = NoValue | Value a Bool` in which the `Bool` represents the stability. A `TaskValue` is special since it may change over time because its event handling function is re-evaluated on every event. Once a value is `Stable`, it does not change again. A `TaskValue` can be observed by other tasks and it can affect which other tasks are to be started. There are basic tasks and combinators to compose tasks in familiar workflow patterns.

Basic tasks come in two flavours: interactive and non-interactive. Non-interactive basic tasks consist of processing, task value manipulation, external connections, and communication with the host system.

Interactive tasks provide interaction with a user via a type driven generated web interface. A type used in iTasks must have instances for a collection of generic functions that is captured in the class `iTask`[2]. One of these generic functions is the generic web form generation that allows the user to edit a value of that type using the web browser. Basic types have specialization instances for these generic functions and they can be derived for any first-order user-defined type. When desired, derived interfaces can be fine-tuned or specialized instances can be defined.

The main interactive tasks for entering, viewing and updating values of arbitrary types are called the `*Information` tasks and shown below. These tasks spawn a web form for the user to the work with. The first argument is the title, the second argument contains the display options on the data and the third argument of the `view` and `update` variant are the initial value.

```
enterInformation :: String [EnterOption m ]   → Task m | iTask m
viewInformation  :: String [ViewOption  m ] m → Task m | iTask m
```

[2]In Clean, class constraints on overloaded functions are placed after the signature denoted by a bar, separated by ampersands

```
updateInformation :: String [UpdateOption m m] m → Task m | iTask m
```

2.2 Task Combinators

Tasks can be combined with task combinators to express sequential, parallel and conditional workflows. With these combinators, one can define how tasks depend on each other and how the information is passed between them. The resulting combination delivers a new task. Some of the — for this paper — relevant combinators are explained below.

The parallel combinators combine two or more tasks in such a way that they are offered to the user at the same time, possibly combining the result. For example the -|| emits the first task with a value and stabilizes when either one of the task has a stable value. The -&&- emits a task value only when both sides have a value and stabilizes only when both sides are stable. Specialized versions of the -|| exist that executes the two tasks in parallel but only regards the value of one side.

```
(-||-)  infixr 3 :: (Task a) (Task a) → Task a       | iTask a
(-&&-)  infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a & iTask b
(-||)   infixr 3 :: (Task a) (Task b) → Task a       | iTask a & iTask b
(||-)   infixr 3 :: (Task a) (Task b) → Task b       | iTask a & iTask b
allTasks         :: [Task a] → Task [a]              | iTask a
```

Instead of running tasks at the same time, sequential task combinators compose tasks sequentially. The value of the left-hand side is fed to the right-hand side if one of the `TaskCont` predicates hold. These predicates can be based on the stability of the value, the actual value, an action or an exception. Actions are presented to the user as buttons in the generated web form. Exceptions are thrown by tasks and can be caught using the `try` construction. The bind combinator (>>=) is implemented as a step that continues only when either the left-hand side is stable or the user presses the continue button.

```
(>>=) infixl 1 :: (Task a) (a → Task b)          → Task b | iTask a & iTask b
(>>|) infixl 1 :: (Task a) (Task b)              → Task b | iTask a & iTask b
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)] → Task b | iTask a & iTask b

:: TaskCont a b
    =     OnValue         ((TaskValue a) → Maybe b)
    |     OnAction String ((TaskValue a) → Maybe b)
    | ∃e: OnException     (e → b)                 & iTask e

try :: (Task a) (e → Task a) → Task a | iTask a & iTask e & toString e
```

2.3 Shared Data Sources

Non sequential data sharing happens in TOP via SDSs. SDSs are solely defined by their stateful read and write functions and are an abstraction on data in the broadest sense. For example, an SDS can be a file on disk, the system time, a place in memory, lenses on other SDSs, or an external database. There is a publish-subscribe system attached to SDSs which means that a task reading an SDS is automatically notified when the value has changed. This results in low resource usage because tasks do not need to poll. Lenses on SDSs can be used to map functions, combine multiple SDSs, or apply data or notification filters. An SDS is typed by three types; the read, the write type and the parametric lens type. The parametric lens is ignored for now and is fixed to `()` in all access tasks anyway. However, different parameter types are later used (see Section 7.4).

There are four atomic tasks to interact with SDSs. The `get` function retrieves the value, the `set` value sets the value and the `upd` changes the value. Finally the `watch` function is a task that constantly returns the value of the SDS when it is changed.

```
:: SDS p r w
:: Shared a :== SDS () a a

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

Moreover, the interactive `*Information` (see Section 2.1) tasks are available for SDSs as well. In this way, one can interact with shared data using the generated web forms. Below are the type signatures for these functions. The view on the SDS data in the web page is automatically updated when the SDS is updated. These functions support the same lenses as their counterparts.

```
viewSharedInformation   :: String [...] (SDS r w) → Task r | iTask r
updateSharedInformation :: String [...] (SDS a a) → Task a | iTask a
```

There are some extra functions available in the realm of SDSs that need some introduction. The >*< operator combines two SDSs. The `mapRead` function embeds a transformation function atomically in the given SDS. Derived from the `watch` function is the `whileUnchanged` function that — given an SDS and a task — executes the task every time the SDS value changes.

A way to create an SDS is by using the `withShared` function. This function creates a memory mapped SDS that is only available within scope.

```
(>*<) infixl 6 :: (SDS () rx wx) (SDS () ry wy) → SDS () (rx, ry) (wx, wy)
mapRead        :: (SDS () r w) (r → r') → (SDS r' w)
whileUnchanged :: (SDS () r w) (r → Task b) → Task b | iTask b
withShared     :: a ((SDS () a a) → Task b) → Task b | iTask b
```

2.4 Thermostat Example: iTasks

As an illustrative running example we introduce a thermostat application written in iTasks.

The user can set the limits through the generated web interface. If the temperature drops below the lower limit, the heater turns on. If it rises over the upper limit, the fan turns on.

The iTasks system provides the interaction with the data sources to the user. All of the communication is going through these data sources which are represented by SDSs. An SDS is created for the current temperature, upper and lower limit. Moreover, virtual SDSs define the on/off state of the heater and the cooler. The current temperature and the corresponding limit are combined to an SDS lens yielding a `Bool` defining the limit status.

Now suppose we are also able to execute tasks on an IoT device and suppose we have all the actual device interaction captured in the `iotThermostat` task (Section 4.3). Then we can program a thermostat as follows.

```
1  thermostat :: Task Int
2  thermostat =
```

```
3       withShared 0  (λcurrentTemp →
4       withShared 18 (λlowerTarget →
5       withShared 22 (λupperTarget →
6       let coolerSDS = mapRead (uncurry (>)) (currentTemp>*<lowerTarget)
7           heaterSDS = mapRead (uncurry (<)) (currentTemp>*<upperTarget)
8       in viewSharedInformation   "Current"     [ViewAs viewAsCelcius] currentTemp
9       -|| updateSharedInformation "Lower limit" [] lowerTarget
10      -|| updateSharedInformation "Upper limit" [] upperTarget
11      -|| viewSharedInformation   "Cooler"      [] coolerSDS
12      -|| viewSharedInformation   "Heater"      [] heaterSDS
13      -|| iotThermostat currentTemp cooler heater)))
14  where
15      viewAsCelcius s = toString s +++ "[+~1degree+]C"
```

Lines 3-5 instantiates SDSs in memory representing the current temperature and the limits. The initial value for the current temperature (type `Shared Int`) is set to 0 and we will discuss later how it is updated. The initial temperature limits are 18 and 22 degrees and are both represented by a `Shared Int`. All communication between the tasks goes via these SDSs.

Lines 6-7 contain lenses on SDSs to create virtual SDSs representing the status for the heater and the cooler. The current temperature and limit are combined using a comparison operator.

Lines 8-12 contain the tasks that generate the user interface. The interface depends on the type of the SDS. Hence, it is used to change the target temperatures, view the current temperature and view the status of the cooler and the heater. The view option from line 15 makes sure the temperature — a plain `Int` — is decorated with a unit.

Line 13 contains the IoT logic which is defined in Section 4.3. The compound task uses the SDSs to operate the thermostat. It creates an IoT task, compiles it to bytecode and executes this code on the device. The device measures the temperature and controls the cooler and the heater and communicate via the SDSs.

This code — modulo some aesthetic options — results in the interface in Figure 1. Every change in the IoT system — server *or* device — is automatically propagated to tasks watching it. This results in an interface that automatically updates when for example the temperature on the device updates. Moreover, if the user modifies one of the limits, this is automatically propagated to the device so that the IoT tasks can respond to it.

Figure 1: Thermostat user interface

3 AN eDSL FOR IoT TASKS

Regular iTasks tasks are not suitable to run on small devices because of their resource usage. However, a subset of the TOP tasks are natural to the IoT domain and we still want to express them in a type safe and extendible way. EDSLs offer a solution for creating new languages in a host language while benefiting from properties of the host language such as the type system.

3.1 Class-Based Shallow Embedding

There are several basic embedding techniques, such as shallow and deep embedding [Gibbons 2015]. Class-based shallow embedding — or tagless embedding — has the advantages of both shallow and deep embedding [Carette et al. 2009; Svenningsson and Axelsson 2012]. Here, the language constructs are defined as type classes and a backend is a type with an instance for some of the classes. This means that adding backends is easy and a backend only needs to implement the classes it needs. Moreover, type safety is guaranteed because the types can contain phantom types and constraints can be enforced by the type signatures of the class functions. Lastly, extensions can be added easily. Existing backends do not need to be updated when an extension is added in a new class. Naturally, if the extension is added in an existing class, the backends implementing the class need to be updated.

3.2 IoT EDSL

The mTask eDSL is a class-based shallowly eDSL hosted in Clean [Koopman and Plasmeijer 2016]. Their backend generates C-code for complete TOP-like programs that can run on an Arduino. The language itself is imperative of nature and programs written in it are suited to run on an MCU. However, this backend generates a self-contained system and is not suitable for our purpose because there is no connection whatsoever with the regular iTasks system.

In this paper, the mTask eDSL is extended with a new bytecode generation backend and language extensions that allow run-time assignment of tiny IoT tasks to IoT devices as well as integration of these IoT tasks with regular iTasks tasks. To avoid confusion, the extended mTask eDSL with the novel backend is called the IoT eDSL.

The IoT eDSL is a collection of classes implementable by types with two type variables. The type implementing the classes is called the backend (`b`). The first type variable (`t`) represents the type of the construction and the second type variable (`r`) the role of the construction. Type constraints are used to make sure the expressions are well typed and to disallow expressions like: `lit True +. lit 1`. Roles can be `Expr`, `Stmt` and `Upd` to denote expressions, statements and updatables. The roles form a hierarchy that is expressed in class constraints, for example, an updatable can be used as an expression but not the other way around. This type and class definitions for this hierarchy follows.

```
:: Expr = Expr
:: Stmt = Stmt
:: Upd  = Upd

class isExpr a :: a
instance isExpr Upd, Expr
class isStmt a :: a
instance isStmt Upd, Expr, Stmt
```

The constructions in the IoT language can be grouped by roles and extra categories for device access and SDS operations. The tasks are small imperative programs that are executed continuously by the RTS and therefore there is no need for loop control. Therefore we only need conditional and sequential statements.

3.2.1 Expressions. There are two classes of expressions, namely boolean expressions and arithmetic expressions. The class of arithmetic language constructs also contains the function `lit` that lifts a host language value into the IoT eDSL domain. All operators are suffixed with a full stop to avoid have name clashes with Clean's builtin operators. All standard arithmetic functions are included in the eDSL, but some are omitted for brevity.

```
class arith b where
    lit          :: t              → b t Expr
    (+.) infixl 6 :: (b t r) (b t q) → b t Expr   | + t & isExpr r & isExpr q
    ...
```

3.2.2 Statements. Both the sequence operator (`:.`) and the conditional (`IF`, `?`) statements are shown below. The `?` is a variant of the standard conditional operation where the else clause is empty.

```
class IF b where
    IF          :: (b Bool p) (b t q) (b s r) → v () Stmt | isExpr p
    (?) infix 1 :: (b Bool r) (b t q)         → v () t    | isExpr p
class seq b where
    (:.) infixr 0 :: (b t r) (b u q)          → b u Stmt
```

3.2.3 Assignables. The IoT eDSL offers an imperative language and therefore an assignment construction is very natural. Only constructs with the `Upd` role can be assigned to. Examples of constructs with the `Upd` role are variables and General Purpose Input/Output (GPIO) pins. Variables — and other decorations — can only be defined at the top level. The `Main` type statically ensures this. The type signature is complex; to illustrate the usage, an implementation example for a variable written to an analog pin is given below.

```
:: In   a b = In infix 0 a b
:: Main a   = {main :: a}
:: DigitalPin = D0 | D1 | D2 | D3 | D4 | D5 | ...

class dIO b :: DigitalPin → b Bool Upd
class var b :: ((b t Upd) → In t (Main (b c r))) → (Main (b c r)) | ...
class assign b where
    (=.) infixr 2 :: (b t Upd) (b t r) → b t Stmt | isExpr r

writeAnalog :: Main (b Bool Stmt)
writeAnalog = var λx=True In {main = dIO D3 =. x}
```

4 THE GLUE BETWEEN TOP AND IoT TASKS

With a language to express IoT tasks we still need glue to actually put the bytecode on the device at run-time for execution as well as a way of integrating them with a TOP server. From the iTasks program, IoT tasks can be executed on a device transparently as if it were iTasks tasks. The IoT tasks and iTasks tasks can communicate via SDSs that are synchronized between the device and the server. The glue functions and tasks to achieve this can be divided into three categories, namely *devices*, *tasks* and *SDSs*.

4.1 Glue Functions and Tasks

We introduce the `withDevice` task that is needed to interact with an IoT device. This task — given a specification — connects to a device and sets it up for usage with the iTasks system. The task only requires a specification that implements the `Duplex` class. The `Duplex` class' only member is the iTasks task that, given the specification, synchronizes the communication channels. Implementations of this class have been made for TCP and Serial devices. When the device is connected, the further interaction is communication agnostic. The `Device` is an abstract type representing an IoT device and passed to functions interacting with devices. It can be seen as a reference to the device and IoT/iTasks programmers should not use the structure directly. If the programmer wants to use another type of device or communication method, they only need to change the value they pass to `withDevice`.

```
withDevice :: a (Device → Task b) → Task b | Duplex a & iTask a & iTask b
class Duplex a where synFun :: a Device → Task ()
```

IoT tasks that are expressed in the IoT eDSL have to be compiled to bytecode first. Next, they are sent to the device for execution. All of this is captured in the `liftIOTTask` function. This function compiles the IoT task, sends it to the device and handles the communication. When the IoT task terminates, the lifted task becomes stable. In TOP, there is no hard limit in the number of tasks that are assigned to the same system. In analogy, multiple IoT tasks that are sent to the same device, are executed after each other. The scheduling of the IoT tasks is done by the RTS on the device, hence the lack of looping functionality in the IoT tasks. An IoT task is always accompanied by a scheduling strategy that is either a oneshot execution or a repeated execution (see Section 4.2).

```
:: Interval = OneShot | OnInterval Int
liftIOTTask :: (Device, Interval) (Main (ByteCode a Stmt)) → Task ()
```

All interaction of the iTasks system with the running IoT tasks happens via SDSs. To accommodate this, a class has been added that looks similar to the `var` class that allows iTasks SDS to be used in IoT tasks. SDSs in iTasks are automatically published to all readers when it is written. Applying this strategy to IoT SDSs could cause a large communication overhead. To mitigate this overhead, a lifted SDS that is written on the server is passed on to the device immediately, but a device writing an SDS needs to publish this explicitly using the added `pub` class.

```
class sds v :: ((v t Upd)→In (SDS t t) (Main (v c s))) → (Main (v c s)) | ...
class pub v :: (v t Upd) → v t Stmt
```

4.2 Scheduling

Tasks sent to an IoT device are accompanied by a scheduling strategy. With this strategy they behave like TOP tasks in the sense that they are continuously executed and their values can be observed, albeit through SDSs. Two scheduling strategies are available for different types of workflow.

The `OneShot` strategy can be used to execute a task only once. In IoT applications, often the status of a peripheral or system has to be queried only once on the request of the user. For example, a thermostat might read the temperature every hour but, the user might want to know the temperature at an exact moment. Then they can just send a `OneShot` task probing the temperature. If the temperature sensor is connected to GPIO analog pin 7, such a task looks like

```
sharePin :: Device (Shared Pin) → Main (ByteCode () Stmt)
sharePin dev someShare
    = liftIOTTask (dev, OneShot)
        (sds λpin=someShare In {main = pin =. aIO A7 :. pub pin})
```

Secondly, tasks accompanied with the `OnInterval Int` strategy are executed every given number of milliseconds. This strategy most closely resembles tasks as in the iTasks system and fits the use case of periodic measurements. The task shown previously can be used to measure the temperature constantly. Moreover, the strategy can be (ab)used to simulate recursion because variables and SDS are kept between executions on the device. The repeated execution can be terminated with a `return`. The following example shows this with the factorial function.

```
IOTFac :: Device Int → Main (ByteCode () Stmt)
IOTFac dev n
    = liftIOTTask (dev, OnInterval 500)
        (var λresult=1 In var λy=n In {main =
            IF (y ==. 0) return (result =. y *. result :. y =. y -. lit 1)})
```

4.3 Thermostat Example: IoT

Now that we have provided all the tooling, we can finish the example from Section 2.4. It is possible to program the thermostat with only a single task and a single device but to make the example a bit more interesting, we divided the work over two devices: `sensorDevice` and `coolerDevice`.

The first device — connected through TCP — measures the temperature and operates the heater. The temperature is read from analog GPIO pin `A0` and written in the `currentTemp` SDS. The heater is connected to digital GPIO pin `D1` and is set according to the value in the `heater` SDS.

The second device — connected via a serial connection — operates the cooling fan. The cooler is connected to digital GPIO pin `D5` and the state of the cooler is read from the `cooler` SDS. The sensor is executed every 500 milliseconds and the cooler and heater IoT tasks every 1000 milliseconds.

```
1  iotThermostat :: (Shared Int) (Shared Bool) (Shared Bool) → Task ()
2  iotThermostat currentTemp cooler heater = readTempHeat -||- operateCooler
3  where
4      sensorDevice :: TCPSettings
5      sensorDevice = {host="192.168.0.12", port=8888}
6
7      coolerDevice :: TTYSettings
8      coolerDevice = {devicePath="/dev/ttyUSB0", baudrate=B9600, ...}
9
10     readTempHeat :: Task ()
11     readTempHeat
12         = withDevice sensorDevice λsensor→
13             liftIOTTask (sensor, OnInterval 500)
14                 (sds λx=currentTemp In {main= x =. analogRead A0 :. pub x})
15         -||- liftIOTTask (sensor, OnInterval 1000)
16                 (sds λf=heater      In {main=dIO D1 =. f})
17
18     operateCooler :: Task ()
19     operateCooler
20         = withDevice coolerDevice λcoolerOper→
21             liftIOTTask (coolerOper, OnInterval 1000)
22                 (sds λf=cooler In {main= dIO LED1 =. f :. dIO D5 f})
```

Line 2 is the parallel combination of the two tasks representing the work that needs to be done on each device. Lines 4-8 give the specification for the devices.

Lines 10-16 show the work that is done on the TCP device. First the device is instantiated with the `withDevice` task. Then, two IoT tasks are sent to the device that run in parallel. The first IoT task reads and publishes the current temperature and the second operates the heater.

Lines 18-22 describe the work that is done on the cooler device. The IoT task operates the cooler through GPIO pin `D5` and shows the status on `LED1`.

5 COMPILING IoT TASKS

Sending an IoT task to a device is a multi step process under the hood.

First, the class functions from the IoT eDSL are implemented for the `ByteCode` type. This type is a boxed Reader Writer State Transformer (RWST) [Jones 1995] that transforms a compiler state while writing bytecode instructions when evaluated.

```
:: ByteCode t r = BC (RWS () [BC] BCState ())
```

Secondly, the RWST generates the appropriate bytecode but it also keeps track of the used SDSs and variables in the state. An IoT task is not only defined by its bytecode but also by its SDSs and variables that are stored in the state together with fresh identifier streams. The state is kept between compilations to not have common identifiers between tasks.

Finally, all the aforementioned data must be converted to messages that the device can understand. To keep the communication overhead small and the execution fast, the bytecode is assembled. The assembly consists of converting the instructions to bytes and resolving the labels. A device receives two types of messages for an IoT task. First, it receives the specification for all the SDSs and variables. Then, it receives the task containing the bytecode.

5.1 Instruction Set and Representation

The instruction set is defined by the `BC` type and contains basic instructions for a stack machine such as stack operations, labels, jumping and arithmetics. IoT specific instructions are included to allow interaction with peripherals. Moreover, it contains instructions for variables and SDSs (prefixed with `BCSds`). There is no typing on the instruction level, all types are boxed in the `BCValue` type. This box can contain any type for which the `IOTType` class is defined. The context restriction contains all functions needed to interact with the values (e.g. serialization and de-serialization) to send them to the device and the `iTask` class to interact with them via a web interface. Instances for the serialization classes are given for the basic types, IoT specific types — e.g. GPIO pins and LEDs — and for the box type itself.

```
:: BC
    = BCLab Int      | BCJmp Int      | BCJmpT Int | BCJmpF Int
    | BCPop          | BCPush BCValue
    | BCAdd          | BCMult         | ...
    | BCSdsStore Int | BCSdsFetch Int | BCSdsPub Int
    | BCAnalogRead AnalogPin | ...
:: BCValue =∃e: BCValue e & IOTType e
```

```
class toByteCode   a :: a → String
class fromByteCode a :: String → (Either String (Maybe a), String)
class IOTType a | toByteCode a & fromByteCode a & iTask a
```

Generating the stack machine bytecode is straightforward for the basic imperative language constructs. The next listing shows some implementation for the arithmetic and conditional classes.

```
tell :: w → RWST r w s m () //From MonadWriter

instance arith ByteCode where
    lit x             = BC (tell [BCPush (BCValue x)])
    (+.) (BC x) (BC y) = BC (x ≫| y ≫| tell [BCAdd])
    …

instance IF ByteCode where
    (?) b t = …
    IF  (BC b) (BC t) (BC e) = BC $
        freshlabel ≫=λelse → freshlabel ≫=λendif →
        b ≫| tell [BCJmpF else] ≫|
        t ≫| tell [BCJmp endif, BCLab else] ≫|
        e ≫| tell [BCLab endif]

//Fetch a label from the state
freshLabel :: RWS () [BC] BCState Int
```

5.2 Shared Data Sources (SDSs)

SDSs and variables used in an IoT task are related concepts. They are represented differently in the compiler and in the iTasks system but on the device they are the same thing. They are both stored in a list of `BCShares` that is stored in the compiler's state. In the compilation process, the method of getting the initial value is different and the `sdss` are synchronized with the referenced iTasks SDSs on execution. For a `var`, the initial value is available but for an `sds` this initial value must be queried using the `get` iTasks task. A `BCShare` consists of a device unique identifier and either the initial value or the iTasks reference.

```
:: BCShare = {sdsi :: Int, sdsval :: Either BCValue (Shared BCValue)}
```

An `sds` definition is always of the form `sds λ x=someShare In ↪ {main = …}`. The compiler adds a `BCShare` to the list and the lambda variable — named `x` in this case — is the RWST writing the `BCSdsFetch` instruction with the identifier embedded. The `BCShare` is initialized with the default value in the `var` case. The `sds` case requires some more work because IoT SDSs in a `BCShare` record are not typed anymore by the Clean type system in the compiler state but boxed in the `BCValue` type. Therefore, a lens on the linked iTasks SDS is created to map the original type to the box and the other way around. This is a potentially unsafe cast, but the `BCValue` SDS is not accessible from the outside. It is used to process publications coming from the device. The device cannot change the type and therefore this is safe.

```
mapReadWriteError :: (r → MaybeError String r`, w` r → MaybeError String (Maybe
      ↪ w)) (SDS r w) → SDS r` w`

lens :: (SDS t t) → SDS BCValue BCValue | IOTType t
lens s
    = mapReadWriteError
        (λ           t → Ok (BCValue t)
        ,λ(BCValue v) t → case fromByteCode (toByteCode v) of
            (Right (Just t), "") = Ok (Just t)
            _ = Error "Mismatch in BCValue type"
        ) v
```

5.3 Assignables

Assignables — e.g. the `dIO` construct — result in an RWST writing a *fetch* instruction. Therefore, an the left hand side of an assignments needs to be rewritten to their *store* counterpart. The `censor` function from the `Writer` monad is used to rewrite the instruction accordingly. This technique is applied for all assignables and the technique is used for the `pub` function as well to transform the `BCSdsFetch` instruction to a `BCSdsPublish` instruction.

```
instance dIO ByteCode    where dIO (BC p) = BC (tell [BCDigitalRead p])
instance assign ByteCode where (=.) (BC v) (BC e) = BC (e ≫| censor makeStore v)

makeStore [BCSdsFetch i]    = [BCSdsStore i]
makeStore [BCDigitalRead i] = [BCDigitalWrite i]
makeStore […]               = […]
```

5.4 Compilation Example

To demonstrate the compilation, the following code shows a room monitoring program that reports if the temperature is too high. The report is done by setting the alarm SDS to `True`, this will trigger an iTasks task for handling the alarm. If the temperature (read from the given pin) is over the `panic` value, the alarm will sound. Moreover, it will also set the alarm if the temperature is over the `limit` value for longer than 10 ticks. The IoT task is parametrized as a Clean function and requires a temperature pin, a limit value, a panic value and an alarm SDS. The values of the arguments can easily be obtained through an editor in iTasks. This really shows that Clean is a macro language that you can use to construct IoT tasks dynamically and according to a runtime specification. It returns a tailor made IoT task that can be sent to the device.

```
temp :: AnalogPin Int Int (Shared Bool) → Main (ByteCode () Stmt)
temp pin limit panic alarmShare =
    sds λalarm = alarmShare In
    var λcount = 0          In
    {main =
        IF (aIO pin > . lit panic)
            (alarm =. lit True :.  pub alarm )
            ( IF (aIO pin > . lit limit) (
                count =. count +. lit 1 :.
                IF ( count > . lit 10)
                    ( alarm =. lit True :. pub alarm)
                    ( noOp )
            ) ( count =. lit 0)
    }
```

Using the bytecode backend, this program is compiled to the bytecode given. The bytecode is numbered with the program memory offset. The labels are already resolved to actual addresses but for clarity the conditional structure is displayed next to the instructions. The compiler state returning includes two `BCShare` values, the second — identifier 2 initialized with the `var` construct — has the initial value `0 :: Int`. The first — identifier 1 initialized with the `sds` construct — has no initial value because it is a referenced iTasks SDS. The initial value is retrieved from the iTasks system upon sending the IoT task to the device.

```
 0. BCAnalogRead A0
 2. BCPush (panic :: Int)
 6. BCGre
 7. BCJmpF 20              / if (aIO pin >. lit panic)
 9. BCPush (True :: Bool)  |
12. BCSdsStore 1           |
15. BCSdsPublish 1         |
18. BCJmp 69               |
20. BCAnalogRead A0        + else
22. BCPush (limit :: Int)  |
26. BCGre                  |
27. BCJmpF 62              |   / if (aIO pin >. lit limit)
29. BCSdsFetch 2           |   |
32. BCPush (1 :: Int)      |   |
36. BCAdd                  |   |
37. BCSdsStore 2           |   |
40. BCSdsFetch 2           |   |
43. BCPush (10 :: Int)     |   |
47. BCGre                  |   |
48. BCJmpF 60              |   |   / if (x >. lit 10)
50. BCPush (1 :: Bool)     |   |   |
53. BCSdsStore 1           |   |   |
56. BCSdsPublish 1         |   |   |
58. BCJmp 60               |   |   + else
60. BCJmp 69               |   |   \ endif
62. BCPush (0 :: Int)      |   + else
66. BCSdsStore 2           |   |
69. END OF PROGRAM         \   \ endif
```

6 RUNTIME SYSTEM

The RTS is the single program/firmware that needs to be executed on the device for the system to be able to execute IoT tasks and integrate with iTasks. On startup, the allocated memory is initialized, followed by running a device specific setup function. In this specific setup function, peripherals can be initialized and communication can be established. If this function returns, a connection with the server has been made and the main loop is continuously executed. Only in case of a shutdown request, the memory is cleared and the device specific setup function is executed again so that the device is in the initial state again waiting for a connection.

The main loop (1) checks if there is input on the communication channel. If input is available, it reads and parses this to a message and process the message. All messages — such as new tasks, SDSs or a specification request — are processed immediately. (2) The RTS executes tasks that are ready. This means that all one shot tasks and all interval tasks for which the interval has passed are executed. The execution happens in a round robin fashion. If a one shot task is executed, it is removed from the memory. If an interval task is executed, its last run time is set to the current time. Interpretation always starts from the first cell in the program memory and ends when the program counter exceeds the size of the program or a return instruction is encountered in which case the task is also removed from the memory. (3) When one entire loop is finished, the program waits for a — compile time determined — time after which it continues. During this waiting the device idles.

6.1 Interface

The RTS is written in C and only uses standard C functions such that the same code can be used for all devices. All device specific functions are hidden in a single header file called the interface. It contains functions for accessing device specific peripherals, communication functions, setup and tear-down and it defines the specification. The interface header file is created in a modular way using conditional macros. This means that — similar to the eDSL — parts of the interface are optional. Every device has to implement the communication functions but peripherals are optional. The specification of the device is generated from the implemented functions to tell the server upon startup what its capabilities are. Therefore, porting the RTS to a new device only requires implementing the interface. The device specific interface is very simple. For example, there are only three functions regarding communication as shown below.

```
bool    input_available(void);
uint8_t read_byte(void);
void    write_byte(uint8_t b);
```

Implementations are made for *POSIX*, *mbed*[3], *ChibiOS*[4] and *Arduino*[5] compatible platforms using either TCP or Serial connections.

6.2 Memory Management

The RTS has to store both statically and dynamically allocated data. The static data contains the interpreter state, the interpreter stack and the communication buffers. The dynamic data consists of the tasks and SDSs. Tasks consists of their bytecode, an identifier and the scheduling information. An SDS is made up of the identifier and their value.

Some devices have very little memory and therefore space needs to be used optimally. While almost all MCUs support heaps nowadays, the functions for allocating and freeing memory on the heap are not very space optimal and often leave holes if allocations are not freed in a last in first out order. SDSs and tasks may be dynamically added and removed. To mitigate this problem, the RTS manages its own — compile time configurable sized — memory in the global data segment. Tasks are stored from the top down and SDSs are stored from the bottom up.

When a task or an SDS is removed, this managed space is compacted immediately so that there are no holes left. In practice this means that if the first received task is removed, all tasks received later are moved up until the hole is filled completely. Obviously, this is quite time intensive, but it cannot be permitted to leave holes in the memory since the memory space is so limited. With this aggressive memory management technique, even an Arduino UNO R3 with just $2K$ RAM can execute several tasks accessing several SDSs concurrently.

7 SERVER INTEGRATION

The server part of the system — given as an iTasks implementation — is responsible for housekeeping the IoT devices. Accessing the functionality only happens via two glue functions shown in Section 4.1. The following sections give details detail on what these functions actually do under the hood.

[3] https://mbed.com
[4] https://chibios.org
[5] https://arduino.cc

7.1 Communication with Devices

Every IoT device has an iTasks memory based SDS assigned to it that contains their communication channels. These channels form the communication-agnostic interface for all communication to-and-fro the device. The type signature for this is given below. The channels SDS is a triple containing an incoming channel, an outgoing channel and a stop flag. The synchronization task — started on device connection — synchronizes the communication channels with the device. If messages appear in the first list, the synchronization task relays them to the device. Moreover, if the device sends a message, the synchronization task places it in second list. When the stop flag is set, the synchronization function terminates because the connection with the device is closed. If a new communication method is to be added, a programmer only has to implement the synchronization function for the whole system to work.

```
:: Channels :== Shared ([MSGRecv], [MSGSend], Bool)
:: MSGRecv = MTTaskAck Int Int | MTSDSAck Int | MTPub Int BCValue | ...
:: MSGSend = MTTask Interval String | MTSds Int BCValue | ...
```

7.2 Device Storage

All devices are stored in one global SDS containing a list of `Device` records. Each record contains everything a device encompasses. Storing all devices in a single SDSs has the advantage that one can inspect all devices from anywhere in the iTasks program. This facilitates debugging and error handling. The system knows which devices are connected and which IoT SDSs are available on a device. Moreover, this approach also allows the creation of systems that reconnect devices after a restart because a persistent SDS can be used to store the devices.

Every record stores a reference to the communication channels, the compiler state, the information to setup the synchronization function, a list of IoT tasks and a list of IoT SDSs. If the device is connected, it also stores the task identifier for the synchronization function, and the hardware specification. If the device is erroneously disconnected it can set a descriptive error.

The programmer can only add devices to the system through the `withDevice` function. This function is a wrapper around several asynchronous functions that interact with the device records in the global SDS. This function (1) adds the device to the global device list (2) connects the device (3) executes the task requiring the device (4) waits for a stable value for the given task (5) requests a shutdown for the device (6) removes the device from the global device list.

Connecting a device is also a multi step process in itself. It (1) clears the channels. (2) starts a message processing task. (3) sends a device specification request and stabilizes when this request has been honored.

The message processing task is a compound task consisting of the device specific synchronization function and a device agnostic message processing function. The message processing function acts upon new messages in the incoming channels. For example, when an `MTPub` is received, the corresponding SDS in the device record is updated. Similarly, when an `MTTaskAck` is received, the task in the device record is updated with the appropriate task identifier.

7.3 Synchronizing Shared Data Sources

The server stores a proxy value for every IoT SDS in the form of an iTasks SDS. This proxy iTasks SDS stores the latest value from the device as a cache. If it is written, it will send an SDS write request to the device. Watchers are notified when the device published a new value. This cached value is stored in the device record in the `IOTShare` type. This `IOTShare` type contains the identifier, the current value and possibly the reference to the iTasks SDS.

```
:: IOTShare = { identifier:: Int
              , value     :: BCValue
              , iTaskRef  :: Maybe (Shared BCValue)
              }
```

If the device publishes a new value for an IoT SDS, the processing task updates the proxy value stored in the device record. Moreover, the processing task writes the new value to the reference iTasks SDS lens. This lens automatically translates the `BCValue` box to the correct value and writes the actual referenced iTasks SDS.

The other way around, when a task updates the referenced iTasks SDS, a watcher task — started in the `liftIOTTask` function (Section 7.4) is notified. The watcher task can then update the proxied value in the device record accordingly. The synchronizing of this proxied value with the actual device is explained in Section 7.5.

7.4 Executing Tasks

The programmer can only execute tasks on a device through the `liftIOTTask` task. In the same fashion as the `withDevice` function, it wraps several device record modifying tasks.

The function (1) compiles the IoT task to messages (2) places the messages in the channels SDS (3) adds the task to the device record (4) sets the task identifier when the task acknowledgement is received (5) waits for the task to stabilize while watching all the reference iTasks SDSs watching a reference iTasks task is done by using the `whileUnchanged` function on the `Shared BCValue` lens. Moreover, it removes the task from the device by sending a `MTTaskDel` message when the iTasks task is terminated. Termination of iTasks tasks happens for example if the task is part of a step (`>>*`) and one of the conditions matches or when one of its siblings throws an exception.

7.5 Lenses on the Device SDSs

All the device information is stored in a single iTasks SDS for reasons mentioned in Section 7.2. Unfortunately, there are also two issues with this approach.

First, it is not convenient to edit the global SDS containing the devices if you are only interested in a part of it. For example, updating a single IoT SDS value for a single device requires a complicated update function.

Secondly, a task watching the global SDS might only be interested in a very small section of it but is notified on all changes. For example, watcher tasks are launched in the `liftIOTTask` function. These tasks watch only a single reference iTasks SDS to make sure the value is proxied in the device record. However, if another task writes in the device SDS to a different place, the watchers are notified either way.

To solve the first type of problems, parametric lenses were introduced [Domoszlai et al. 2014]. The `p` that was fixed to `()` in Section 2

represents the parameter and this parameter is available during reading and writing. In the SDS access tasks, this parameter must be fixed to `()` which can be achieved using the `sdsFocus` function. This function fixes the parameter and casts it to `()` so that the SDS is usable for the standard tasks. The SDS can specify or refine the data read or written according to the given value of p. Moreover, it can place a filter on the notifications using this parameter. For example, it can be used to create a lens on a tuple — e.g. only giving write access to the first element. If the second element is written, a watcher on the first element is not notified.

```
sdsFocus :: p (SDS p r w) → SDS () r w | iTask p
```

And the type used for the parametric lens and the device SDS is defined as follows.

```
:: IOTParam = Global | Local Device | Share Device Int
deviceStore :: SDS IOTParam [Device] [Device]
```

Every constructor denotes a different type of view on the root SDS. First, the `Global` constructor is only interested in the entire list of devices. Secondly, the `Local` lens only looks at a single device. Finally, the `Share` view only focusses on a single SDS on a single device. For these SDS lenses, functions are available to access the parts of the global SDS.

```
deviceStoreNP ::                    Shared [Device]
deviceShare   :: Device             → Shared Device
shareShare    :: Device IOTShare    → Shared BCValue
```

Focussing the `deviceStore` to `Global` gives access to the global SDS. Global watchers are only be notified if the structure of the list changes, i.e. if a device is added or removed.

Accessing a single device is done with the `deviceShare` function. The SDS requires a `Device` record to know on which device to focus. This record is available within the task given to `withDevice`. Watchers of a local SDS are notified when something in the device changes. Writers can change something in the device such as the specification.

Share SDSs can be accessed through the `shareShare` function. This function focusses the global SDS on a single SDS on a single device. It is only notified when that specific proxy SDS value changes.

This brings us to the last unsolved part of the extension, namely synchronizing the proxy values with the device. Albeit not designed for it, parametric lenses can also be used to solve this problem. The parameter is known in the stateful write function. The state can be used to write to other — e.g. device's channels — SDSs.

When a task writes to this SDS, the global SDS knows this through the parameter and propagates the value to the device. When the server or the device changes the SDS, this view is notified. The SDS requires a `Device` and a `IOTShare` record. The `IOTShare` record and therefore the lens is only used internally, for example by the processing function watching the IoT SDSs to synchronize them with their iTasks references.

7.6 Task Migration and Task Construction

The thermostat example is kept simple for illustration purposes. However, it does not show the full potential of the extension. For example, it is possible to extend the thermostat with a redundant cooler. If the cooler IoT device would stop working it will be automatically reassigned to another device. The only function we need to change is the `operateCooler` function, note that the `cooler` iTasks SDS is in scope here. The `withDevice` function throws an exception if a device terminates the connection and the connection cannot be re-established. This exception can be caught and the work that is done on the device can be moved to another device.

```
operateCooler :: Task ()
operateCooler
    = try
        (withDevice coolerDevice runCoolerTask)
        (λexc → viewInformation "Exception" [] (toString exc)
            ≫| withDevice coolerDevice2 runCoolerTask)
where
    runCoolerTask dev
        = liftIOTTask (dev, OnInterval 1000)
            (sds λf=cooler In {main= dIO LED1 =. f :. dIO D5 =. f})

    coolerDevice2 :: TTYSettings
    coolerDevice2 = {devicePath="/dev/ttyACM0", baudrate=B19200, ...}
```

Moreover, IoT tasks can be sent dynamically at runtime to the device. To illustrate this, we show a task in which the user can send tasks to a device. The user selects the task from a list of predefined tasks and provides the execution strategy via the web interface as well. If this task is executed in parallel with the sensor and cooling tasks with the device, the user can use the device as well for miscellaneous other tasks. For example to blink a light, or to open window blinds on demand while the thermostat is operating.

```
interact :: Device → Task ()
interact device
    =    enterChoice "Choose a task" [ChooseFromList fst] taskList
    -&&- enterInformation "Execution Strategy" []
    >^* [OnAction (Action "Send") $ withValue λ((_, task), strat)→
            Just (task ≫=λiottask→liftIOTTask (device, strat) iottask)]
    @! ()
where
    taskList :: [(String, Task (Main (ByteCode () Stmt)))]
    taskList =
        [("faculty", enterInformation "Faculty of what?" []
            ≫=λn → var λx=1 In var λy=n In {main =
                IF (x ==. 0) return (x =. y *. x :. y -. lit 1)})
        ,("count", return $
            sds λx=0 In {main = x =. x +. lit 1 :. pub x})
        ,("blink", enterInformation "Led on which pin?" []
            ≫=λl → var λx=True In {main = x =. Not x :. dIO l =. x})
        ,("...", ...)
        ]
```

8 RELATED WORK

Related research has been conducted on the subject arising from academia and the industry. For example, MCUs such as the Arduino can be remotely controlled very directly using the Firmata-protocol[6]. This protocol is designed to allow control of the peripherals — such as sensors and actuators — directly through commands sent via a communication channel such as a serial port. This allows very fine grained control but with the cost of communication bandwidth since no code is executed on the device itself, only the

[6] https://github.com/firmata/protocol

peripherals are queried. A Haskell implementation of the protocol is also available[7]. The hardware requirements for running a Firmata client are very low because all the logic is on the server. However, the bandwidth requirements are high and therefore it is not suitable for IoT applications that communicate through LTN networks. Similarly, Grebe and Gill [2016] created *HaskIno*, a monadic interface over *hArduino* that allows remote code execution on Arduinos. Their initial tethered solution is based on Firmata but they also propose an untethered approach that is similar to compilation by storing the program in EEPROM. However, there is no communication between the device and the server that programmed it and the solution is very specific to the Arduino ecosystem. An extension has been proposed where explicit threading is supported [Grebe and Gill 2017] in which the code executed is similar to mTask's IoT tasks. It differs in the execution model and in data access between threads and there is no shared data with the server.

There are also some more general OS/RTS solutions for MCUs that allow the programmer to program MCUs on a more abstract level. They generate a static image to flash on the MCU for operation and do not support dynamic task sending and there is no out of the box typed transparent data sharing between the server and the client. For example, Levis et al. [2005] created TinyOS, which is an OS that can compile a static program for a lot of MCUs and supports threading and has a similar execution model as the new system — namely slicing programs and lacking a blocking API. Furthermore, Elsts et al. [2015] proposed ProFUN, a — similar to TOP — declarative language using Task Graphs (TGs) to create sensor networks. The TGs can be created in a graphical interface accessible with a web browser. Functionality exists for automatic logging via a server application.

Clean has a history of interpretation, for example, there is a lot of research happening on the intermediate language SAPL. SAPL is a purely functional intermediate language that can be efficiently interpreted. It has interpreters written in C++ [Jansen et al. 2007] and a compiler to JavaScript [Domoszlai et al. 2011]. Compiler backends exist for Clean and Haskell which compile the respective code to SAPL [Domoszlai and Plasmeijer 2012]. The SAPL language is a functional language and therefore requires big stacks and heaps to operate and is therefore not directly suitable for devices with little RAM such as the Arduino. It might be possible to compile the SAPL code into efficient machine language or C but then the system would lose its dynamic properties since the MCU then would have to be reprogrammed every time a new task is sent to the device.

EDSLs have often been used to generate C code for MCU environments. This work uses parts of the existing mTask-eDSL which generates C code to run a TOP-like system on MCUs [Koopman et al. 2018; Koopman and Plasmeijer 2016]. Again, this requires a reprogramming cycle every time the task-specification is changed and there is no interaction with the server. The nature of the embedding technique allows additional backends to be written without touching existing ones. Hence, the eDSL is used for this solution but with a novel backend.

Another eDSL designed to generate low-level programs is called Ivory and uses Haskell as a host language [Elliott et al. 2015]. The language uses the Haskell type-system to make unsafe languages type safe. For example, Ivory has been used in the automotive industry to program parts of an autopilot [Hickey et al. 2014; Pike et al. 2014]. Ivory's syntax is deeply embedded but the type system is shallowly embedded. This requires several Haskell extensions that offer dependent type constructions. The process of compiling an Ivory program happens in two stages. The embedded code is transformed into an Abstract Syntax Tree (AST) that is sent to a chosen backend. In our system, the eDSL is transformed directly into functions and there is no intermediate AST. Moreover, Ivory generates static programs and thus it is necessary to reprogram the devices when they need to be repurposed.

Not all IoT devices run solely compiled code, e.g. the ESP8266 powered NodeMCU is able to run interpreted Lua code. Moreover, there is a variation of Python called micropython that is suitable for running on MCUs. However, the overhead of the interpreter for such rich languages often results into limitations on the program size. It would not be possible to repurpose an IoT device because implementing this extensibility in the interpreted language leaves no room for the actual programs. Also, some devices only have $2K$ of ram, which is not enough for this.

9 CONCLUSION

The IoT is growing and gaining popularity very fast. However, programming the IoT is cumbersome. The devices in the IoT are programmed individually using a plethora of programming languages and communication protocols. Previously we introduced an eDSL to program the IoT devices from a single source. In this work we modify the approach to support creating dynamic IoT applications.

First, an iTasks extension is shown that allows us to dynamically connect devices to a running program. The communication is physical connection method and protocol agnostic. The devices need to be programmed once with an appropriate RTS to function in this system. The RTS is available for several MCUs. The device specific part of the RTS is small and modular. This makes porting it to a new architecture easy and keeps the footprint small. Adding peripherals is easy. It does not even require recompilation of clients due to the modular setup of the RTS code.

Secondly, we add high level communication between the eDSL on the device and the iTasks server via SDSs. The iTasks SDSs on the server can be shared with specific devices. These devices publish SDS changes on command to limit the amount of communication.

Lastly, we tackle the maintenance problem of programs on the devices in the field. A new backend for the mTask eDSL is created that compiles to tailor-made bytecode. This bytecode is shipped at runtime to the device for execution. The RTS interprets the bytecode using a stack machine. The IoT tasks are then dynamically loaded to a device for execution. This dynamic allocation of tasks to devices makes the system very flexible. For example, when an IoT task is assigned to a device and that device becomes unusable, the iTasks system can reassign it to another device automatically to add redundancy to a system.

We think that these contributions make it suitable for real world applications. We are currently testing this. Nevertheless, the technique can be improved in umpteen ways.

[7] https://leventerkok.github.io/h\gls{Arduino}

10 FUTURE WORK

10.1 Extensions

An additional simulation view to the IoT eDSL can be added that works similar to the existing C-backend simulation. The first option is to simulate a device completely, the simulator is an instance of the `Duplex` class. Secondly, it can simulate symbolically by implementing the eDSL classes. At the time of writing, work is done on an iTasks simulator device of the former kind.

True multitasking can be added to the client software. IoT tasks get slices of execution time and have their own stack, this allows IoT tasks to run truly concurrent. Multitasking allows tasks to be truly interruptible by other tasks. Furthermore, this allows for more fine-grained timing control of tasks. However, it influences memory requirements.

Many research topics can be explored in the field of resource management and analysis, both statically at compile time and dynamically at runtime for both peripheral requirements and memory requirements.

10.2 Further Improvements

The current implementation offers a subset of the TOP combinators in IoT. The subset can be extended to allow for more fine-grained control flow between IoT tasks. Furthermore, more logic can be moved to the device instead of it residing on the server reducing the communication overhead.

At the moment, all data is sent as plain text over the wire and the device cannot know whether it talks to a legitimate server or an attacker. Due to the nature of the system, namely sending code that is executed, security needs to be investigated. Only bytecode for very specific IoT tasks can be sent to the device at the moment which mitigates the risk somewhat. As the language will become more expressive, the security risk increases.

Finally, the robustness of the system can be improved. Tasks residing on a device that disconnects should be kept on the server to allow a swift reconnect and restoration of the tasks. Moreover, an extra specialization of the shutdown can be added that drops the connection but keeps the tasks in memory. This can be extended by allowing devices to send their tasks back to the server. In this way devices can even connect to different servers. Tasks can be stored in EEPROM or on external memory to be able to access them even after a reboot or to save memory. EEPROM can be written about ten to a hundred times more often than flash memory.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments that helped improve this paper.

REFERENCES

Tom Brus, Marko van Eekelen, Maarten Van Leer, and Marinus Plasmeijer. 1987. Clean – a language for functional graph rewriting. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 364–384.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

Li Da Xu, Wu He, and Shancang Li. 2014. Internet of things in industries: a survey. *Industrial Informatics, IEEE Transactions on* 10, 4 (2014), 2233–2243.

László Domoszlai, Eddy Bruel, and Jan Martin Jansen. 2011. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae* 3 (2011), 76–98.

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

László Domoszlai and Rinus Plasmeijer. 2012. Compiling Haskell to JavaScript through Clean's core. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica* 36 (2012), 117–142.

Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt free ivory. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 189–200.

Atis Elsts, Farshid Hassani Bijarbooneh, Martin Jacobsson, and Konstantinos Sagonas. 2015. ProFuN TG: A tool for programming and managing performance-aware sensor network applications. In *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th*. IEEE, 751–759.

Jeremy Gibbons. 2015. Functional programming for domain-specific languages. In *Central European Functional Programming School*. Springer, 1–28.

Mark Grebe and Andy Gill. 2016. Haskino: A remote monad for programming the arduino. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 153–168.

Mark Grebe and Andy Gill. 2017. Threading the Arduino with Haskell. In *Post-Proceedings of Trends in Functional Programming*.

Patrick Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. 2014. Building embedded systems with embedded DSLs. In *ACM SIGPLAN Notices*. ACM Press, 3–9. https://doi.org/10.1145/2628136.2628146

Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. 2007. Efficient Interpretation by Transforming Data Types and Patterns to Functions. *Trends in Functional Programming* 7 (2007), 73.

Mark Jones. 1995. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*. Springer, 97–136.

Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM Press, 1–11. https://doi.org/10.1145/3183895.3183902

Pieter Koopman and Rinus Plasmeijer. 2016. A Shallow Embedded Type Safe Extendable DSL for the Arduino. In *Trends in Functional Programming*. Lecture Notes in Computer Science, Vol. 9547. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-39110-6.

Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

Arjan Oortgiese, John van Groningen, Achter Peter, and Plasmeijer Rinus. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29nd 2017 International Symposium on Implementation and Application of Functional Languages (IFL '17)*. ACM, New York, NY, USA, 12.

Lee Pike, Patrick Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. 2014. Programming languages for high-assurance autonomous vehicles: extended abstract. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages meets Program Verification*. ACM Press, 1–2. https://doi.org/10.1145/2541568.2541570

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

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*. ACM, Leuven, Belgium, 195–206.

Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2011. Clean language report version 2.2 (2011). https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf

Josef Svenningsson and Emil Axelsson. 2012. Combining deep and shallow embedding for EDSL. In *International Symposium on Trends in Functional Programming*. Springer, 21–36.