Interpreting Task Oriented Programs on Tiny Computers

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

Small Microcontroller Units (MCUs) drive the omnipresent Internet of Things (IoT). These devices are small, cheap, and energy efficient. However, they are not very powerful and lack an Operating System. Hence it is difficult to apply high level abstractions and write software that stays close to the design.

Task Oriented Programming (TOP) is a paradigm for creating multi-user collaborative systems. A program consists of tasks—descriptions of what needs to be done. The tasks represent the actual work and a task value is observable during execution. Furthermore, tasks can be combined and transformed using combinators.

mTask is an embedded Domain Specific Language (eDSL) to program MCUs following the TOP paradigm. Previous work has described the mTask language, a static C code generator, and how to integrate mTask with TOP servers. This paper shows that for dynamic IOT applications, tasks must be sent at runtime to the devices for interpretation. It describes in detail *how* to compile specialized IOT TOP tasks to bytecode and *how* to interpret them on devices with very little memory. These additions allow the creation of complete, dynamic IOT applications arising from a single source using a mix of iTasks and mTask tasks. Details such as serialization and communication are captured in simple abstractions.

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

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2020. Interpreting Task Oriented Programs on Tiny Computers. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, Singapore, Singapore, USA, 12 pages. https://doi.org/10.1145/3412932.3412936

1 INTRODUCTION

IOT consists of tiny devices that sense, act, and communicate with each other and with the world. The IOT is often visualized as a layered system [5]. An example of an IOT application that touches every layer is a smart-home hub. A smart-home hub usually consists of a coordinating server and several clients. This server coordinates the devices and offers an interface to the user, e.g. via a display and buttons or a web interface. Typical clients have hard-coded tasks or they receive them from a server on demand. Some clients contain interfaces to interact with the user directly and start tasks as well.

In IOT applications, clients are often heterogeneous collections of microcontrollers all with their own peculiarities, language of choice and hardware interfaces. The hardware needs to be cheap, small and energy efficient. As a result, the MCUs used to power these devices have little computational power, a tiny amount of memory, and little communication bandwidth. Typically the devices do not run a full-fledged OS but statically compiled firmware. This firmware is often written in an imperative language and needs to be flashed to the program memory. This greatly reduces the flexibility for dynamic systems where tasks are created on the fly and executed on demand. While devices are getting a bit faster, smaller, and cheaper, they keep these properties to an extent. In this paper we show how these problems can be solved by dynamically sending interpreted TOP code to the MCU.

The TOP paradigm is a declarative paradigm where a program consists of tasks. Tasks represent collaborative work that needs to be done by people and computer systems. The iTasks framework is the reference implementation for TOP [20]. Given the task specification, iTasks generates a multi-user web interface to, guide the work based on the current state of affairs.

Tasks are generated on the fly and can be combined to form compound tasks. Moreover, it is possible to tailor-make tasks at runtime for specific work that needs to be done. A task's execution is modelled as an interactive rewriting system. As a consequence, work is automatically divided into short slices and when interleaving the work is seemingly executed in parallel. When designing a program for an IOT device, the jobs that need to be performed on the device fit the TOP paradigm very well [18]. Tasks that devices need to perform are frequently parallel and adapted to the current needs of the system. An example of this adaptability and parallel nature for our smart-home hub is a multi-room thermostat. Different tasks are required to be executed at the same time, e.g. measuring the temperature in the room, controlling the heater and reacting to user input both on the client *and* on the server, for example when devices in the rooms contain buttons to locally set the target temperature.

Unfortunately, straightforwardly compiling iTasks to code suitable for IOT devices is not possible since the devices are simply not powerful enough. Furthermore, writing multi-threaded applications using actual threads is strenuous because of the restricted memory capacity and lack of an OS. There are ways of simulating threads in imperative languages on MCUs, but they all have downsides such as spaghetti code, the lack of local state, and compile-time fixed thread timings (see Section 7).

Previous work described the TOP language mTask [14]. The mTask language is an EDSL hosted in Clean that aides writing the device part of an IOT application using TOP. The mTask language contains an important subset of TOP such as tasks, task combinators and Shared Data Sources (SDSs). It allows interaction with peripherals following the TOP paradigm. However, this version only supported generating static C code and there was no built-in communication with a server.

In real-life TOP applications, tasks are generated on the fly and combined at will using combinators. The program memory for an MCU is not suitable for rapid reprogramming when a task needs to be executed. We therefore resort to interpretation. With interpretation, tasks are sent at runtime to the device from a server to be executed on demand. In previous work we have shown how a simplified imperative version of mTask could be integrated with iTasks [17]. Small, imperative tasks could be lifted to iTasks tasks and could access iTasks SDSs. They were compiled to bytecode and interpreted on the device. However, there was no support for task oriented programming on the client.

1.1 Research contribution

For dynamic applications, generating code at runtime for interpretation on the device is necessary. In this paper we show in detail how to compile mTask tasks to bytecode. This bytecode can be interpreted on MCUs with very little memory and processing power. The programmer does not have to worry about details such as serialization, communication, or rapidly changing programs using precious write cycles of the program memory. With these additions, complete, dynamic IOT applications can be constructed from a single source using a mix of iTasks and mTask tasks.

Our solution accomplishes this by (1) describing mTask's semantics more formally, (2) providing a concrete compilation scheme for compiling mTask tasks to bytecode for an abstract machine, and (3) describing a concrete implementation of this abstract machine.

Section 2 presents an overview of TOP in general while Section 3 introduces the mTask language. Section 4 presents the bytecode backend infrastructure—i.e. the integration with iTasks, the compiler and the Runtime System, and presents a demonstrative example. The compilation rules are given in Section 5 followed by the rewriting rules in Section 6. The paper concludes with related work, conclusions and future work (Section 7, 8 and 9).

2 TASK ORIENTED PROGRAMMING

TOP is a programming paradigm for specifying distributed systems modelling collaborations between people and machines. *Tasks* are the basic building blocks of the program and behave a bit like communicating threads. They represent the work that needs to be done and they can be combined and transformed to form compound tasks. The full set of combinators available in the reference implementation and their semantics can be found in [21]. Tasks are event-driven, stateful rewrite functions that yield, after each step, a three-state *task value*. Task values are observable by other tasks and may change over time. The task value is either *no value*, *unstable* or a *stable* value. The allowed task value transitions are shown in Figure 1. The events that drive the execution of a task can be anything ranging from user input to clocks, hardware events or even the task itself requesting a subsequent execution step.

$$\begin{tikzcd} NoValue \arrow[r, leftrightarrow] \arrow[rr, bend left] & Unstable \arrow[r] & Stable \end{tikzcd}$$

Figure 1: A state transition diagram for task values.

Task values can be observed and acted upon by other tasks if they are combined sequentially or in parallel. Furthermore, data can be shared using SDSs. SDSs provide a general abstraction over data. They can be accessed through three tasks that retrieve, store or modify the data (`get`, `set`, and `upd` respectively). While tasks in parallel are interleaved, access to SDSs is always atomic, i.e. exactly one rewrite step. SDSs can be combined and transformed using combinators similar to those used for tasks.

3 MTASK LANGUAGE

The core of the mTask system is the mTask language—a multi-backend class-based EDSL. The classes are type-constructor classes and therefore a backend implementing a class is a type of the form `v t` where `v` is the actual backend. The phantom type `t` represents the type of the construction. Not all types are suitable for MCUs: they have to be serializable and bounded. To enforce this, the type `t` must have instances for the `type` class collection containing these constraints.

The classes for expressions—i.e. arithmetic functions, conditional expressions and tuples—are given in Listing 1. Some of the functions are oddly named (e.g. `+.`) to avoid name conflicts with existing functions. There is no need for loop control due to support for tail-call optimized recursive functions and tasks. The `lit` function fulfills a special role of the language: it allows lifting host language values to the mTask domain. For tuples there is a useful default instance (`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
    ...
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

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

Listing 1: The mTask classes for simple expressions.

3.1 Functions

Functions are supported in the EDSL, albeit with some limitations. All user-defined mTask functions are typed by Clean expressions to ensure type safety. All functions are defined using the multi-parameter typeclass `fun` (Listing 2). The first parameter (`a`) of the typeclass is the shape of the argument and the second parameter (`v`) is the backend. Functions may only be defined at the top level and to constrain this, the `Main` type is introduced to box a program.

One implementation for the `fun` class is defined for every arity. So for a function from `a` to `b`, the **instance** `fun (T a) T | type a` is used. The listing gives example instances for arities zero to two for backend `T`. Defining the different arities as tuples of arguments forbids the use of curried functions. To illustrate the use, the factorial function is given as an example. The `type` constraint on the function arguments forbids the use of higher order functions. This Clean function will construct the program that will calculate the factorial of the given argument.

All mTask constructions used in the `factorial` function must be defined as class constraints on the backend type variable. This creates quite a bit of clutter as the number of class constraints increases rapidly. The class collection `mtask` can be used to avoid long lists of constraints.

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

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

class mtask v | arith v & cond v & ... & fun () v & fun (v Int) v & ...

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 x)}
```

Listing 2: The mTask classes for function definitions.

3.2 Tasks

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

Task values in mTask are represented by the same type as in iTasks (Listing 3). To lift a value in the expression domain to the task domain, the basic task `rtrn` is used. The resulting task will yield the given value as a stable task value.

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

Listing 3: The mTask classes for basic tasks.

Interaction with General Purpose Input/Output (GPIO) pins, and other peripherals for that matter, is also captured in basic tasks. For each type of pin, there is a read and a write task that, given the pin, executes the action. The class for analogue GPIO pin access is shown in Listing 4. The `readA` task constantly yields the value of the analogue pin as an unstable task value. The `writeA` task writes the given value to the given pin once and returns the written value as a stable task value.

```
:: APin = A0 | A1 | A2 | A3 | ...
:: DPin = D0 | D1 | D2 | D3 | ...
class aio v where
    readA  :: (v APin)         → MTask v Int
    writeA :: (v APin) (v Int) → MTask v Int
```

Listing 4: The mTask classes for GPIO access.

3.3 Task combinators

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

The step combinator (`>>*.`) is the Swiss army knife of sequential combination (Listing 5). The value that the left-hand side of the combinator yields is matched against the task continuations (`Step v t u`) on the right-hand side, i.e. the right-hand side tasks *observe* the task value. If one of the continuations yields a new task, the combined task continues with it, pruning the left-hand side. All other sequential combinators are derived from the step combinator as default instances but their implementation can be overridden to provide a more efficient implementation. The `>>=.` combinator is very similar to the monadic bind: it continues if and only if a stable value is yielded. The `>>~.` combinator continues when any value, stable or unstable, is yielded. The `>>|.` and `>>..` combinators are variants of the aforementioned combinators that do not take the value into account.

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

    (>>=.) infixl 1 :: (MTask v t) ((v t) → MTask v u) → MTask v u | ...
    (>>=.) m f = m >>*. [IfStable (λ_→lit True) f]
    (>>~.) infixl 1 :: (MTask v t) ((v t) → MTask v u) → MTask v u | ...
    (>>~.) m f = m >>*. [IfValue (λ_→lit True) f]
    (>>|.) infixl 1 :: (MTask v t) (MTask v u) → MTask v u | ...
    (>>|.) m f = m >>=. λ_→f
    (>>..) infixl 1 :: (MTask v t) (MTask v u) → MTask v u | ...
    (>>..) m f = m >>~. λ_→f

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

Listing 5: The mTask classes for sequential task combinators.

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

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

Listing 6: An example task using sequential combinators.

In contrast to iTasks—that has one super combinator for all parallel combinations—there are only two parallel combinators (Listing 7). The conjunction combinator `.&&.` combines the task values to a tuple. The disjunction combinator `.||.` combines them into a single task value, giving preference to the *most stable* value (see Subsection 6.4). The listing shows an example of querying two pins at the same time, returning the one with the highest value.

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

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

Listing 7: The mTask classes for parallel task combinators.

Finally there are some miscellaneous combinators. For example, the `rpeat` function forever executes the argument task. When the argument task is stable, it starts all over again. The `delay` task waits for the specified amount of time. This task yields no value until the given time has elapsed, then it will yield the number of milliseconds it overshot as a stable task value. To demonstrate them, the `blink` program is given that toggles the given pin every 500 milliseconds. The functionality of `rpeat` can also be simulated using a recursive function as shown in the `blinkFun` task.

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

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

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

Listing 8: The mTask classes for repeat and delay.

3.4 Shared Data Sources

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

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

Listing 9 provides the definitions and an artificial example of a task mirroring a pin value to another pin using an SDS.

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

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

Listing 9: The mTask classes for SDS tasks.

The `liftsds` class below is used to allow iTasks SDSs to be accessed from within mTask tasks and vice versa. The function has a similar type as `sds` and creates an mTask SDS from an iTasks SDS so that it can be accessed using the class functions from the `sds` class. Listing 10 shows an example of this where an iTasks SDS is used to control an LED on a device. During task execution, the device gets notified when the SDS is modified on the server.

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

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

Listing 10: The mTask class for lifting iTasks SDSs.

4 BYTECODE COMPILER INFRASTRUCTURE

This section presents the infrastructure surrounding the bytecode backend. This is where tasks are compiled, sent, and executed during runtime of the iTasks server to a specific device. The supporting Clean functions for this are given in Listing 11.

The `withDevice` function offers access to the device. The first argument of the function, which is a type implementing the `channelSync` class, contains the information for maintaining a connection with the device. At present, there are instances for `channelSync` for types representing a TCP or serial connection and a simulator. The resulting task connects the device and ascertains that the connection is set up, kept up and closed down on completion. After the connection is set up, the second argument—the task doing something with a device—is executed.

Within the argument task—besides executing iTasks tasks—the `liftmTask` task can be used. This function lifts an mTask task to an iTasks task using the specified device so that the mTask task can be fully utilized within the iTasks system. The `BCInterpret` type houses the backend and therefore implements the mTask classes. It adheres to the compilation scheme given in Section 5. The mTask constructions have the form `Main (MTask BCInterpret u)`. Under the hood, `liftmTask` creates a task that executes the compiler, sends the generated bytecode, listens for messages from the device and watches the lifted SDSs. The task value of the mTask task is observable from iTasks because the task is now a regular iTasks task. Furthermore, lifted SDSs can be accessed and used for communication. In a traditional setting, all these things—such as communication, data sharing, task scheduling—would have to be done by hand. In contrast, `liftmTask` automatically does it all.

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

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

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

Listing 11: Functions integrating mTask with iTasks.

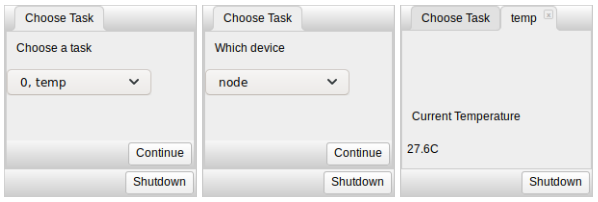
4.1 Example

This subsection presents a toy home automation program (Listing 12) to illustrate the language and its integration with an iTasks server. It consists of a web interface for the user to control, which tasks may be executed on either of two connected devices: an Arduino UNO, connected via a serial port and an ESP8266 based prototyping board called NodeMCU, connected via TCP over WiFi.

Lines 1–2 show the specification for the devices followed by lines 4–8 containing the actual task. This task first connects the devices (lines 5–6) followed by a `parallel` task that is visualized as a tabbed window with a shutdown button to terminate the program (lines 7–8). The `chooseTask` task (lines 10–20) allows the user to pick a task, sending it to the specified device. Tasks are picked from the `tasks` list (lines 22–39). For example, the `temperature` task might show the current temperature to the user. When the temperature changes, the Digital Humidity and Temperature sensor reports it and the task value for the `temperature` task changes. This change in task value is reflected in the iTasks server and the task value of the `liftmTask` task changes accordingly. The task is lifted to an iTasks task and the `>&>` iTasks combinator transforms the task value into an SDS that is displayed to the user using `viewSharedInformation`. A screenshot of the temperature task is given in Figure 2.

```
1   arduino = {TTYSettings | zero & devicePath="/dev/ttyACM0"}
2   nodeMCU = {TCPSettings | host="192.168.0.1", port=8123}
3
4   autoHome :: Task ()
5   autoHome = withDevice arduino λdev1→
6              withDevice nodeMCU λdev2→
7       parallel [(Embedded, chooseTask dev1 dev2)] [] <<@ ArrangeWithTabs True
8       >>* [OnAction (Action "Shutdown") (always (shutDown 0))]
9
10  chooseTask :: MTDevice MTDevice (SharedTaskList ()) → Task ()
11  chooseTask dev1 dev2 stl = tune (Title "Choose Task") $ forever
12      $           enterChoice "Choose a task" [] (zip2 [0..] (map fst tasks))
13      >>= λ(i, n)→enterChoice "Which device"  [] ["arduino", "node"]
14      >>= λdevice→appendTask Embedded (mkTask n i device) stl
15      >>| chooseTask dev1 dev2 stl
16  where
17      mkTask n i device _
18          # dev = if (device == "node") dev2 dev1
19          = ((snd (tasks !! i) $ dev)
20              >>* [OnAction (Action "Close") $ always $ return ()]) <<@ Title n
21
22  tasks :: [(String, MTDevice → Task ())]
23  tasks =
24      [ ("temp", λdev→
25          liftmTask (DHT D6 DHT22 λdht→{main=temperature dht}) dev
26          >&> viewSharedInformation "Current Temperature"
27              [ViewAs λi→toString (toReal (fromMaybe 0 i) / 10.0) +++ "C"]
28          @! ())
29      , ("lightswitch", λdev→
30          withShared False λsh→
31              liftmTask (lightswitch sh) dev
32              -|| updateSharedInformation "Switch" [] sh
33      , ("factorial", λdev→
34              updateInformation "Factorial of what" [] 5
35              >>= λi→liftmTask (factorial i) dev
36              >>= viewInformation "result" []
37              @! ())
38      ]
39      , ... ]
```

Listing 12: An example of a home automation program.



(a) Select task (b) Select device (c) View result

Figure 2: Screenshots of the example program in action.

4.2 Runtime system

The RTS is designed to run on systems with as little as 2kB of RAM. Aggressive memory management is therefore vital. Not all firmwares for MCUs support heaps and—when they do—allocation often leaves holes when not used in a Last In First Out strategy. Therefore the RTS uses a chunk of memory in the global data segment with its own memory manager tailored to the needs of mTask. The size of this block can be changed in the configuration of the RTS if necessary. On an Arduino UNO —equipped with 2kB of RAM— this size can be about 1500 bytes.

In memory, task data grows from the bottom up and an interpreter stack is located directly on top of it growing in the same direction. As a consequence, the stack moves when a new task is received. This never happens within execution because communication is always processed before execution. Values in the interpreter are always stored on the stack, even tuples. Task trees grow from the top down as in a heap. This approach allows for flexible ratios, i.e. many tasks and small trees or few tasks and big trees.

The event loop of the RTS is executed repeatedly and consists of three distinct phases.

4.2.1 Communication. In the first phase, the communication channels are processed. The messages announcing SDS updates are applied immediately, the initialization of new tasks is delayed until the next phase.

4.2.2 Execution. The second phase consists of executing tasks. The RTS executes all tasks in a round robin fashion. If a task is not initialized yet, the bytecode of the main function is interpreted to produce the initial task tree. The rewriting engine uses the interpreter when needed, e.g. to calculate the step continuations. The rewriter and the interpreter use the same stack to store intermediate values. Rewriting steps are small so that interleaving results in seemingly parallel execution. In this phase new task tree nodes may be allocated. Both rewriting and initialization are atomic operations in the sense that no processing on SDSs is done other than SDS operations from the task itself. The host is notified if a task value is changed after a rewrite step.

4.2.3 Memory management. The third and final phase is memory management. Stable tasks, and unreachable task tree nodes are removed. If a task is to be removed, tasks with higher memory addresses are moved down. For task trees—stored in the heap—the RTS already marks tasks and task trees as trash during rewriting so the heap can be compacted in a single pass. This is possible because there is no sharing or cycles in task trees and nodes contain pointers pointers to their parent.

4.3 Instruction set

The instruction set is a fairly standard stack machine instruction set. Listing 13 shows the Clean type representing the instruction set of which Table 1 gives detailed semantics. Type synonyms are used to provide insight on the arguments of the instructions. One notable instruction is the `MkTask` instruction, it constructs a task tree node and pushes a pointer to it on the stack.

```
:: ArgWidth  :== UInt8            :: ReturnWidth :== UInt8
:: Depth     :== UInt8            :: Num         :== UInt8
:: SdsId     :== UInt8            :: PerId       :== UInt8
:: JumpLabel :== UInt16

:: BCInstr
    = BCJumpF JumpLabel | BCJump JumpLabel | BCJumpSR ArgWidth JumpLabel
    | BCLabel JumpLabel
    | BCReturn ReturnWidth ArgWidth | BCTailcall ArgWidth ArgWidth JumpLabel
    | BCArg ArgWidth | BCStepArg UInt16 UInt8
    | BCIsStable | BCIsUnstable | BCIsNoValue | BCIsValue
    | BCPush String | BCPop Num | BCRot Depth Num | BCDup | BCPushPtrs
    | BCAdd | BCSub | BCMult | BCDiv | BCAnd | BCOr | BCNot
    | BCEq | BCNeq | BCLe | BCGe | BCLeq | BCGeq
    | BCMkTask BCTaskType

:: BCTaskType
    = BCStable1 | BCStable2 | ... | BCUnstable1 | BCUnstable2 | ...
    | BCRepeat | BCDelay | BCTAnd | BCTOr
    | BCSdsGet SdsId | BCSdsSet SdsId
    | BCStep ArgWidth JumpLabel
    //Peripherals
    | BCReadD | BCWriteD | BCReadA | BCWriteA
    | BCDHTTemp PerId | BCDHTHumid PerId
    | ...
```

Listing 13: The type housing the instruction set.

4.4 Compiler

The bytecode compiler backend for the mTask language is a stateful writer monad. The state contains the bytecode for the main expression, the context of arguments, a function dictionary, streams for fresh labels and SDS identifiers, an SDS dictionary containing both local (*Left*) and lifted (*Right*) SDSs (see Subsection 3.4) and a list of peripherals. Executing the bytecode compiler does not result in usable bytecode immediately. After execution of the monad the following processing steps are applied: all tail call `BCReturn` instructions are optimized to `BCTailcall`; the functions are concatenated before the main expression[1]; redundant instructions are removed; the labels are resolved to actual relative memory addresses; all lifted SDSs are queried for their initial value. The result—bytecode, SDS specifications and peripherals specification—can then be sent to the device for execution.

[1]In this way the labels are still correct when removing the bytecode for the main function to save space

```
:: BCInterpret a :== StateT BCState (WriterT [BCInstr] Identity) a
:: BCState =
    { bcs_mainexpr    :: [BCInstr]
    , bcs_context     :: [BCInstr]
    , bcs_functions   :: Map JumpLabel BCFunction
    , bcs_freshlabel  :: JumpLabel
    , bcs_freshsds    :: UInt8
    , bcs_sdses       :: Map UInt8 (Either String255 (Shared String255))
    , bcs_hardware    :: [BCPeripheral]
    }
:: BCFunction =
    { bcf_instructions :: [BCInstr]
    , bcf_argwidth     :: UInt8
    , bcf_returnwidth  :: UInt8
    }
```

Listing 14: The type for the bytecode backend.

5 COMPILATION RULES

The compilation scheme is divided in three schemes. When something is surrounded by $\|$, e.g. $\|a_i\|$, it denotes the number of stack cells required to store it. Some schemes have a *context r* as an argument which contains information about the location of the arguments in scope. More information is given in the schemes requiring such arguments.

Scheme	Description
$\mathcal{E}[\![e]\!]\ r$	Produces the value of expression e given the context r and pushes it on the stack. The result can be a basic value or a pointer to a task.
$\mathcal{F}[\![e]\!]$	Generates the bytecode for functions.
$\mathcal{S}[\![e]\!]\ r\ w$	Generates the function for the step continuation given the context r and the width w of the left-hand side task value.

5.1 Expressions

Almost all expression constructions are compiled using $\mathcal{E}$. The argument of $\mathcal{E}$ is the context (see Subsection 5.2). Values are always placed on the stack; tuples are unpacked. Function calls, function arguments and tasks are also compiled using $\mathcal{E}$ but their compilations are explained later.

$$
\begin{aligned}
\mathcal{E}[\![\texttt{lit}\ e]\!]\ r &= \texttt{BCPush (bytecode e)};\\
\mathcal{E}[\![e_1\texttt{+.}\ e_2]\!]\ r &= \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \texttt{BCAdd};\\
&\quad \textit{Similar for other binary operators}\\
\mathcal{E}[\![\texttt{Not}\ e]\!]\ r &= \mathcal{E}[\![e]\!]\ r; \texttt{BCNot};\\
&\quad \textit{Similar for other unary operators}\\
\mathcal{E}[\![\texttt{If}\ e_1\ e_2\ e_3]\!]\ r &= \mathcal{E}[\![e_1]\!]\ r; \texttt{BCJmpF}\ l_1;\\
&\qquad \mathcal{E}[\![e_2]\!]\ r; \texttt{BCJmp}\ l_2;\\
&\qquad \texttt{BCLabel}\ l_1; \mathcal{E}[\![e_3]\!]\ r;\\
&\qquad \texttt{BCLabel}\ l_2;\\
&\quad \textit{Where } l_1 \textit{ and } l_2 \textit{ are fresh labels}\\
\mathcal{E}[\![\texttt{tupl}\ e_1\ e_2]\!]\ r &= \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r;\\
\mathcal{E}[\![\texttt{first}\ e]\!]\ r &= \mathcal{E}[\![e]\!]\ r; \texttt{BCPop}\ w;\\
&\quad \textit{Where } w \textit{ is the width of the left value}\\
\mathcal{E}[\![\texttt{second}\ e]\!]\ r &= \mathcal{E}[\![e]\!]\ r; \texttt{BCRot}\ w_1\ (w_1+w_2); \texttt{BCPop}\ w_2;\\
&\quad \textit{Where } w_1 \textit{ is the width of the left and}
\end{aligned}
$$

w_2 of the right value

Translating $\mathcal{E}$ is very straightforward, it basically means executing the monad. Almost always, the type of the backend is not used, i.e. it is a phantom type. To still have the functions return the correct type, the `tell`` helper is used. This function is similar to the writer monad's `tell` function but is casted to the correct type. Listing 15 shows the implementation for the arithmetic and conditional expressions. Note that r, the context, is not an explicit argument but stored in the state.

```
instance arith BCInterpret where
    lit   t   = tell` [BCPush $ toByteCode{|*|} t]
    (+.)  a b = a >>| b >>| tell` [BCAdd]
    ...
instance cond BCInterpret where
    If c t e = freshlabel >>= λelselabel→freshlabel >>= λendiflabel→
        c >>| tell` [BCJumpF elselabel] >>|
        t >>| tell` [BCJump endiflabel,BCLabel elselabel] >>|
        e >>| tell` [BCLabel endiflabel]
```

Listing 15: Backend implementation for the arithmetic and conditional classes.

5.2 Functions

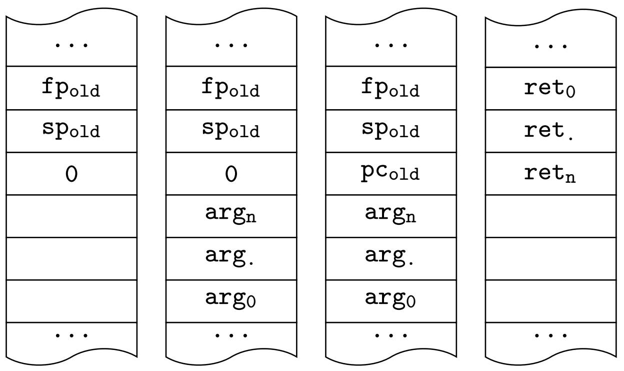
Compiling functions occurs in $\mathcal{F}$, which generates bytecode for the complete program by iterating over the functions and ending with the main expression. When compiling the body of the function, the arguments of the function are added to the context so that the addresses can be determined when referencing arguments. The main expression is a special case of $\mathcal{F}$ since it neither has arguments nor something to continue. Therefore, it is just compiled using $\mathcal{E}$.

$$
\begin{aligned}
\mathcal{F}[\![main = m]\!] &= \mathcal{E}[\![m]\!]\ \ [];\\
\mathcal{F}[\![f\ \ a_0 \ldots a_n = b\ \ \texttt{In}\ \ m]\!] &= \texttt{BCLabel}\ \ f;\\
&\quad \mathcal{E}[\![b]\!]\ \ [\langle f, i\rangle, i \in \{(\Sigma_{i=0}^{n}\|a_i\|)..0\}];\\
&\quad \texttt{BCReturn}\ \ \|b\|\ \ n;\\
&\quad \mathcal{F}[\![m]\!];
\end{aligned}
$$

A function call starts by pushing the stack and frame pointer, and making space for the program counter (a) followed by evaluating the arguments in reverse order (b). On executing `BCJumpSR`, the program counter is set and the interpreter jumps to the function (c). When the function returns, the return value overwrites the old pointers and the arguments. This occurs right after a `BCReturn` (d). Putting the arguments on top of pointers and not reserving space for the return value uses little space and facilitates tail call optimization.

Calling a function and referencing function arguments are an extension to $\mathcal{E}$ as shown below. Arguments may be at different places on the stack at different times (see Subsection 5.4) and therefore the exact location always has to be determined from the context using `findarg`[2]. Compiling argument a_{f^i}, the ith argument in function f, consists of traversing all positions in the current context. Arguments wider than one stack cell are fetched in reverse to preserve the order.

[2]`findarg [l':r] l = if (l == l`) 0 (1 + findarg r l)`



(a) BCPushPtrs (b) Arguments (c) BCJumpSR (d) BCReturn

Figure 3: The stack layout during function calls.

$$
\begin{aligned}
\mathcal{E}[\![f(a_0, \ldots, a_n)]\!]\ \ r &= \texttt{BCPushPtrs};\\
&\quad \mathcal{E}[\![a_n]\!]\ \ r; \mathcal{E}[\![a_{\ldots}]\!]\ \ r; \mathcal{E}[\![a_0]\!]\ \ r;\\
&\quad \texttt{BCJumpSR}\ \ n\ \ f;\\
\mathcal{E}[\![a_{f^i}]\!]\ \ r &= \texttt{BCArg}\ \mathrm{findarg}(r, f, i)\ \ \text{for all}\ \ i \in \{w \ldots v\};\\
v &= \Sigma_{j=0}^{i-1}\|a_{f^j}\|\\
w &= v + \|a_{f^i}\|
\end{aligned}
$$

Translating the compilation schemes for functions to Clean is not as straightforward as other schemes due to the nature of shallow embedding. The `fun` class has a single function with a single argument. This argument is a Clean function that—when given a callable Clean function representing the mTask function—will produce `main` and a callable function. To compile this, the argument must be called with a function representing a function call in mTask. Listing 16 shows the implementation for this as Clean code. To uniquely identify the function, a fresh label is generated. The function is then called with the `callFunction` helper function that generates the instructions that correspond to calling the function. That is, it pushes the pointers, compiles the arguments, and writes the `JumpSR` instruction. The resulting structure (`g In m`) contains a function representing the mTask function (`g`) and the `main` structure to continue with. To get the actual function, `g` must be called with representations for the argument, i.e. using `findarg` for all arguments. The arguments are added to the context and `liftFunction` is called with the label, the argument width and the compiler. This function executes the compiler, decorates the instructions with a label and places them in the function dictionary together with the metadata such as the argument width. After lifting the function, the context is cleared again and compilation continues with the rest of the program.

```
instance fun (BCInterpret a) BCInterpret | type a where
    fun def = {main=freshlabel >>= λfunlabel→
        let (g In m) = def λa→callFunction funlabel (byteWidth a) [a]
        in  addToCtx funlabel zero (argwidth def)
        >>| liftFunction funlabel (argwidth def)
                (g (findArgs funlabel zero (argwidth def))) Nothing
        >>| clearCtx >>| m.main
}

callFunction :: JumpLabel UInt8 [BCInterpret b] → BCInterpret c | ...
liftFunction :: JumpLabel UInt8 (BCInterpret a) (Maybe UInt8) → BCInterpret ()
```

Listing 16: The backend implementation for functions.

5.3 Tasks

Task trees are created with the `BCMkTask` instruction that allocates a node and pushes it to the stack. It pops arguments from the stack according to the given task type. The following extension of $\mathcal{E}$ shows this compilation scheme (except for the step combinator, explained in Subsection 5.4).

$$
\begin{aligned}
\mathcal{E}\llbracket \texttt{rtrn}\ e \rrbracket\ r &= \mathcal{E}\llbracket e \rrbracket\ r; \texttt{BCMkTask BCStable}_{\|e\|};\\
\mathcal{E}\llbracket \texttt{unstable}\ e \rrbracket\ r &= \mathcal{E}\llbracket e \rrbracket\ r; \texttt{BCMkTask BCUnstable}_{\|e\|};\\
\mathcal{E}\llbracket \texttt{readA}\ e \rrbracket\ r &= \mathcal{E}\llbracket e \rrbracket\ r; \texttt{BCMkTask BCReadA};\\
\mathcal{E}\llbracket \texttt{writeA}\ e_1\ e_2 \rrbracket\ r &= \mathcal{E}\llbracket e_1 \rrbracket\ r; \mathcal{E}\llbracket e_2 \rrbracket\ r; \texttt{BCMkTask BCWriteA};\\
\mathcal{E}\llbracket \texttt{readD}\ e \rrbracket\ r &= \mathcal{E}\llbracket e \rrbracket\ r; \texttt{BCMkTask BCReadD};\\
\mathcal{E}\llbracket \texttt{writeD}\ e_1\ e_2 \rrbracket\ r &= \mathcal{E}\llbracket e_1 \rrbracket\ r; \mathcal{E}\llbracket e_2 \rrbracket\ r; \texttt{BCMkTask BCWriteD};\\
\mathcal{E}\llbracket \texttt{delay}\ e \rrbracket\ r &= \mathcal{E}\llbracket e \rrbracket\ r; \texttt{BCMkTask BCDelay};\\
\mathcal{E}\llbracket \texttt{rpeat}\ e \rrbracket\ r &= \mathcal{E}\llbracket e \rrbracket\ r; \texttt{BCMkTask BCRepeat};\\
\mathcal{E}\llbracket e_1 \texttt{.||.} e_2 \rrbracket\ r &= \mathcal{E}\llbracket e_1 \rrbracket\ r; \mathcal{E}\llbracket e_2 \rrbracket\ r; \texttt{BCMkTask BCOr};\\
\mathcal{E}\llbracket e_1 \texttt{.\&\&.} e_2 \rrbracket\ r &= \mathcal{E}\llbracket e_1 \rrbracket\ r; \mathcal{E}\llbracket e_2 \rrbracket\ r; \texttt{BCMkTask BCAnd};
\end{aligned}
$$

This simply translates to Clean code by writing the correct `BCMkTask` instruction as exemplified in Listing 17.

```
instance rtrn BCInterpret where rtrn m = m >>| tell` [BCMkTask (bcstable m)]
```

Listing 17: The backend implementation for `rtrn`.

5.4 Step combinator

The `step` construct is a special type of task because the task value of the left-hand side may change over time. Therefore, the continuation tasks on the right-hand side are *observing* this task value and acting upon it. In the compilation scheme, all continuations are first converted to a single function that has two arguments: the stability of the task and its value. This function either returns a pointer to a task tree or fails (denoted by ⊥). It is special because in the generated function, the task value of a task can actually be inspected. Furthermore, it is a lazy node in the task tree: the right-hand side may yield a new task tree after several rewrite steps (i.e. it is allowed to create infinite task trees using step combinators). The function is generated using the $\mathcal{S}$ scheme that requires two arguments: the context r and the width of the left-hand side so that it can determine the position of the stability which is added as an argument to the function. The resulting function is basically a list of if-then-else constructions to check all predicates one by one. Some optimization is possible here but has currently not been implemented.

$$
\begin{aligned}
\mathcal{E}\llbracket t_1 \texttt{>>*.} t_2 \rrbracket\ r = {} & \mathcal{E}\llbracket a_{fi} \rrbracket\ r, \langle f, i\rangle \in r; \texttt{BCMkTask BCStable}_{\|r\|};\\
& \mathcal{E}\llbracket t_1 \rrbracket\ r;\\
& \texttt{BCMkTask BCAnd};\\
& \texttt{BCMkTask}\\
& \quad (\texttt{BCStep}\ (\mathcal{S}\llbracket t_2 \rrbracket\ (r + [\langle l_s, i\rangle])\ \|t_1\|));
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}\llbracket [] \rrbracket\ r\ w &= \texttt{BCPush}\ \bot;\\
\mathcal{S}\llbracket \texttt{IfValue}\ f\ t : cs \rrbracket\ r\ w &= \texttt{BCArg}(\|r\| + w); \texttt{BCIsNoValue};\\
&\quad \mathcal{E}\llbracket f \rrbracket\ r; \texttt{BCAnd};\\
&\quad \texttt{BCJmpF}\ l_1;\\
&\quad \mathcal{E}\llbracket t \rrbracket\ r; \texttt{BCJmp}\ l_2;\\
&\quad \texttt{BCLabel}\ l_1; \mathcal{S}\llbracket cs \rrbracket\ r\ w;\\
&\quad \texttt{BCLabel}\ l_2;\\
&\quad \textit{Where } l_1 \textit{ and } l_2 \textit{ are fresh labels}\\
&\quad \textit{Similar for } \texttt{IfStable} \textit{ and } \texttt{IfUnstable}\\
\mathcal{S}\llbracket \texttt{IfNoValue}\ t : cs \rrbracket\ r\ w &= \texttt{BCArg}(\|r\| + w); \texttt{BCIsNoValue};\\
&\quad \texttt{BCJmpF}\ l_1;\\
&\quad \mathcal{E}\llbracket t \rrbracket\ r; \texttt{BCJmp}\ l_2;\\
&\quad \texttt{BCLabel}\ l_1; \mathcal{S}\llbracket cs \rrbracket\ r\ w;\\
&\quad \texttt{BCLabel}\ l_2;\\
&\quad \textit{Where } l_1 \textit{ and } l_2 \textit{ are fresh labels}\\
\mathcal{S}\llbracket \texttt{Always}\ f : cs \rrbracket\ r\ w &= \mathcal{E}\llbracket f \rrbracket\ r;
\end{aligned}
$$

First the context is evaluated. The context contains arguments from functions and steps that need to be preserved after rewriting. The evaluated context is combined with the left-hand side task value by means of a `.&&.` combinator to store it in the task tree so that it is available after a rewrite. This means that the task tree is be transformed as follows:

```
t1 >>= λv1→t2 >>= λv2→t3 >>= ...
//is transformed to
t1 >>= λv1→rtrn v1 .&&. t2 >>= λv2→rtrn (v1, v2) .&&. t3 >>= ...
```

The translation to Clean is given in Listing 18.

```
instance step BCInterpret where
    (>>*.) lhs cont
        //Fetch a fresh label and fetch the context
        =   freshlabel >>= λfunlab→gets (λs→s.bcs_context)
        //Generate code for lhs
        >>= λctx→lhs
        //Possibly add the context
        >>| tell` (if (ctx =: []) []
                //The context is just the arguments up till now in reverse
                (  [BCArg (UInt8 i)\\i←reverse (indexList ctx)]
                ++ map BCMkTask (bcstable (UInt8 (length ctx)))
                ++ [BCMkTask BCTAnd]
                ))
        //Increase the context
        >>| addToCtx funlab zero lhswidth
        //Lift the step function
        >>| liftFunction funlab
                //Width of the arguments is the width of the lhs plus the
                //stability plus the context
                (one + lhswidth + (UInt8 (length ctx)))
                //Body label ctx width continuations
                (contfun funlab (UInt8 (length ctx)))
                //Return width (always 1, a task pointer)
                (Just one)
        >>| modify (λs→{s & bcs_context=ctx})
        >>| tell` [BCMkTask $ instr rhswidth funlab]

toContFun :: JumpLabel UInt8 → BCInterpret a
toContFun steplabel contextwidth
    = foldr tcf (tell` [BCPush fail]) cont
where
    tcf (IfStable f t)
        = If ((stability >>| tell` [BCIsStable]) &. f val)
            (t val >>| tell` [])
    ...
    stability = tell` [BCArg $ lhswidth + contextwidth]
    val = retrieveArgs steplabel zero lhswidth
```

Listing 18: Backend implementation for the step class.

5.5 Shared Data Sources

The compilation scheme for SDS definitions is a trivial extension to $\mathcal{F}$ since there is no code generated as seen below.

$$\begin{aligned} \mathcal{F}[\![\texttt{sds}\ \ x = i\ \ \texttt{In}\ \ m]\!] &= \mathcal{F}[\![m]\!]; \\ \mathcal{F}[\![\texttt{liftsds}\ \ x = i\ \ \texttt{In}\ \ m]\!] &= \mathcal{F}[\![m]\!]; \end{aligned}$$

The SDS access tasks have a compilation scheme similar to other tasks (see Subsection 5.3). The `getSds` task just pushes a task tree node with the SDS identifier embedded. The `setSds` task evaluates the value, lifts that value to a task tree node and creates an SDS set node.

$$\begin{aligned} \mathcal{E}[\![\texttt{getSds}\ \ s]\!]\ \ r &= \texttt{BCMkTask}(\texttt{BCSdsGet}s); \\ \mathcal{E}[\![\texttt{setSds}\ \ s\ \ e]\!]\ \ r &= \mathcal{E}[\![e]\!]\ \ r; \texttt{BCMkTask BCStable}_{\|e\|}; \\ &\quad \texttt{BCMkTask}(\texttt{BCSdsSet}s); \end{aligned}$$

While there is no code generated in the definition, the bytecode compiler is storing the SDS data in the `bcs_sdses` field in the compilation state. The SDSs are typed as functions in the host language so an argument for this function must be created that represents the SDS on evaluation. For this, an `BCInterpret` is created that emits this identifier. When passing it to the function, the initial value of the SDS is returned. This initial value is stored as a bytecode encoded value in the state and the compiler continues with the rest of the program.

Compiling `getSds` is a matter of executing the `BCInterpret` representing the SDS, which yields the identifier that can be embedded in the instruction. Setting the SDS is similar: the identifier is retrieved and the value is written to put in a task tree so that the resulting task can remember the value it has written. Lifted SDSs are compiled in a very similar way. The only difference is that there is no initial value but an iTasks SDS when executing the Clean function. A lens on this SDS converting `a` from the `Shared a` to a `String255`—a bytecode encoded version—is stored in the state. The encoding and decoding itself is unsafe when used directly but the type system of the language and the abstractions make it safe. Upon sending the mTask task to the device, the initial values of the lifted SDSs are fetched to complete the SDS specification.

```
:: Sds a = Sds Int
instance sds BCInterpret where
    sds def = {main = freshsds >>= λsdsi→
            let sds = modify (λs→{s & bcs_sdses=put sdsi
                        (Left (toByteCode t)) s.bcs_sdses})
                    >>| pure (Sds sdsi)
                (t In e) = def sds
            in e.main}
    getSds f   = f >>= λ(Sds i)→ tell` [BCMkTask (BCSdsGet (fromInt i))]
    setSds f v = f >>= λ(Sds i)→v >>| tell`
        (  map BCMkTask (bcstable (byteWidth v))
        ++ [BCMkTask (BCSdsSet (fromInt i))])
```

Listing 19: Backend implementation for the SDS classes.

6 TASK REWRITING

Tasks are rewritten every event loop iteration and one rewrite cycle is generally very fast. This results in seemingly parallel execution of the tasks because the rewrite steps are interleaved. Rewriting is a destructive process that actually modifies the task tree nodes in memory and marks nodes that become garbage. The task value is stored on the stack and therefore only available during rewriting.

6.1 Basic tasks

The `rtrn` and `unstable` tasks always rewrite to themselves and have no side effects. The GPIO interaction tasks do have side effects. The `readA` and `readD` tasks will query the given pin every rewrite cycle and emit it as an unstable task value. The `writeA` and `writeD` tasks write the given value to the given pin and immediately rewrite to a stable task of the written value.

6.2 Delay and repetition

The `delay` task stabilizes once a certain amount of time has been passed by storing the finish time on initialization. In every rewrite step it checks whether the current time is bigger than the finish time and if so, it rewrites to a `rtrn` task containing the number of milliseconds that it overshot the target. The `rpeat` task combinator rewrites the argument until it becomes stable. Rewriting is a destructive process and therefore the original task tree must be saved. As a consequence, on installation, the argument is cloned and the task rewrites the clone.

6.3 Sequential combination

First the left-hand side of the step task is rewritten. The resulting value is passed to the continuation function. If the continuation function returns a pointer to a task tree, the task tree rewrites to that task tree and marks the original left-hand side as trash. If the function returns $\perp$, the step is kept unchanged. The step itself never fields a value.

6.4 Parallel combination

There are two parallel task combinators available. A `.&&.` task only becomes stable when both sides are stable. A `.||.` task becomes stable when one of the sides is stable. The combinators first rewrite both sides and then merge the task values according to the semantics given in Listing 20.

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

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

Listing 20: Task value semantics for the parallel combinators.

6.5 Shared Data Source tasks

The `BCSdsGet` node always rewrites to itself. It will read the actual SDS embedded and emit the value as an unstable task value.

Setting an SDS is a bit more involved because after writing, it emits the value written as a stable task value. The `BCSdsSet` node contains the identifier for the SDS and a task tree that, when rewritten, emits the value to be set as a stable task value. The naive approach would be to just rewrite the `BCSdsSet` to a node similar to the `BCSdsGet` but only with a stable value. However, after writing the SDS, its value might have changed due to other tasks writing it, and then the `setSDS`'s stable value may change. Therefore, the `BCSdsSet` node is rewritten to the argument task tree which always represents constant stable value. In future rewrites, the constant value node emits the value that was originally written.

The client only knows whether an SDS is a lifted SDS, not to which iTasks SDS it is connected. If the SDS is modified on the device, it sends an update to the server.

7 RELATED WORK

7.1 Functional Programming on IoT devices

Haenisch et al. showed that Functional Programming (FP) for IOT leads to less code complexity and better maintainability [12]. For example, Microscheme is a purely functional programming language for the Arduino that runs on the little Arduino UNO [23]. To mitigate memory issues, direct compilation is used and some elements like garbage collection and first class closures are omitted. Furthermore it only supports a single thread of execution.

Functional Reactive Programming (FRP) [7] bears similarities to TOP. Juniper, FRP for MCUs, translates to C++ [13]. In this way even the small Arduino UNO can be programmed with parallel tasks. It differs from the mTask approach in the sense that there is no automatic communication and they do not allow dynamic tasks.

Others have approached FRP on MCUs without the computing power constraint in mind. For example, many frameworks such as *Kafka* and *NodeJS*, can be used to program sensor networks without having to know the details of the communication protocols [19]. Also, de Troyer et al. used the FRP language *Elixir* that runs on the *Erlang* virtual machine to create distributed IOT applications with ease [24]. These approaches allow the creation of multithreaded IOT applications with automatic communication similar to TOP. However, a lot more computing power and memory is required to run them and no approach supports dynamic task sending. Smaller devices can participate but they become simpler data relay devices instead.

7.2 Interpretation on IOT devices

Typical MCUs have limited memory which makes interpretation difficult. However, many solutions for interpreted or tethered control of the device have been made.

Lightweight scripting languages for MCUs are amply available, for example micropython, LUA and *EveryLite* [16]. All these solutions require either a more substantial amount of memory or only support single threaded operation.

Another example of this is the Firmata protocol that allows remote control of peripherals via a host machine [22]. Implementations for this in a functional language are also available such as hArduino [8]. Grebe et al. created Haskino that —using a remote monad— allows the interpretation of C++ like code on the Arduino [10]. Later they extended this model to support multi tasking [11]. The language is still imperative and communication between the threads has to be defined explicitly. In mTask, the interleaving only happens on the task level, making SDS communication automatically thread-safe.

Reprogramming MCUs is also possible without interpretation and even wirelessly using Over-the-Air programming. However, this wears out the programming memory. For example Bachelli et al. described such a system in which code can be updated on demand by writing it in the program memory OTA [1, 2].

7.3 Multitasking on IOT Devices

Multitasking and multithreading generally is difficult on MCUs because of the limited resources. Solutions using (real time) OSs are available to support multithreading on MCUs. Examples are TinyOS [15], Qduino [4] or ERIKA [3]. They all have relatively high hardware requirements (e.g. 32kB RAM) and often do not support dynamic tasks or automatic communication with a server.

Multitasking on MCUs is also possible using manual interleaving [9]. It often results in explicitly simulating state machines, is work-intensive, error-prone and results in hard to maintain code when using it for more than simple tasks. To mitigate the issues, protothreads are available [6]. Protothreads allow, by clever use of macros, automatic interleaving of stackless threads that are defined in a single function. However, they cannot span multiple functions.

7.4 Multitasking in a functional language

Suspending and resuming calculations in FP programs in general is possible by using call/CC techniques [25]. The programmer defines where the code can be suspended and with some wrapper code, multiple tasks can be interleaved. Tasks in TOP, and hence mTask, are modelled as rewriting systems and are automatically interleaved by interleaving the rewriting steps. Furthermore, thread-safe communication is provided out of the box. Many other methods of multitasking in functional languages rely on the thread support of the OS which makes them unsuitable for memory-scarce invironments such as IoT devices.

8 CONCLUSION

The mTask system is a programming environment for developing all layers of dynamic IOT applications from a single source following the TOP paradigm. The language is implemented as a multi-backend, class-based shallowly embedded DSL. Its bytecode backend offers run-time compilation for tailor-made IOT tasks that can be executed on tiny microcontrollers. The given compilation scheme describes exactly how tasks are compiled to bytecode. Devices are prepared with an RTS that takes care of the communication between the server and the client as it schedules and executes tasks. It runs on devices with as little as 2kB memory since the RTS only uses about 500 bytes of memory, leaving the rest for tasks, the stack and the heap. Due to the nature of rewriting, parallel execution can be simulated by interleaving the rewrite steps of tasks and subtasks. Furthermore, mTask tasks can be lifted to iTasks tasks to create applications with tasks both on the server and the device. An mTask task can access iTasks SDSs with regular SDS constructs from the mTask language. Programmers can achieve this without knowledge about the underlying protocols and communication techniques, resulting in very compact code. The iTasks task combinator set can then be used on mTask tasks mixed with iTasks tasks, albeit via the server, to create applications spanning all IOT layers. With the addition of the bytecode backend, the mTask system allows the creation of complete, dynamic IOT applications. In this type of application, tasks change rapidly and are constructed using runtime information while not wearing out the program memory.

9 FUTURE WORK

Executing arbitrary code received from a server is a security concern. While the language is quite general, it is still restricted to operate solely on the device itself. Communication with the peripherals is not spelled out using dynamically sent code which mitigates the risk a bit. In the current system, the device cannot determine if it is communicating with a legitimate server. Some IOT communication protocols such as ZigBee have checks for this built in. Security by design is often argued to be a much stronger starting point. Therefore it would be fruitful to investigate the security issues to identify the weak points and improve thereupon.

Tasks are constructed at runtime, but always with Clean as the host language. Follow-up research could be to see whether it is possible to create a type-safe iTasks editor for mTask tasks so that at run-time truly any task can be created out of thin air.

In terms of language features, SDSs are very powerful in TOP. In iTasks, lenses can be applied and SDSs can be transformed. The mTask system would benefit from this as well. It is worth investigating to see which type of lenses and transformations are possible in the mTask ecosystem. Furthermore, the combinators could be enriched with more direct interaction with the peripherals. For example, the step combinator in iTasks contains options for buttons.

The execution model could also be enriched. The iTasks system only rewrites tasks when an event occurred that is of interest to it. It would be interesting to incorporate this as well in the mTask RTS. Events might arise from interrupts, but also timers or triggers from the host server. As an extension, interrupts might be caught using an `IfInterrupt` continuation in the `step` combinator. Furthermore, we would like to investigate how these interrupts can be modeled in mTask and what the behaviour in the rewriting system would be. It might be that the semantics for interrupts are similar to those of exceptions in iTasks.

ACKNOWLEDGMENTS

This research is partly funded by the Royal Netherlands Navy. Furthermore, we would like to thank the reviewers for their invaluable comments.

REFERENCES

[1] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. 2018. Reprogramming Low-end IoT Devices from the Cloud. In *2018 3rd Cloudification of the Internet of Things (CIoT)*. IEEE, 1–6.

[2] Emmanuel Baccelli, Joerg Doerr, Shinji Kikuchi, Francisco Padilla, Kaspar Schleiser, and Ian Thomas. 2018. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE PerCom 2018*.

[3] Pasquale Buonocunto, Alessandro Biondi, and Pietro Lorefice. 2014. Real-time multitasking in Arduino. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE, Pisa, 1–4. https://doi.org/10.1109/SIES.2014.7087331

[4] Zhuoqun Cheng, Ye Li, and Richard West. 2015. Qduino: A multithreaded arduino system for embedded computing. In *Real-Time Systems Symposium, 2015 IEEE*. IEEE, 261–272.

[5] 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.

[6] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. 2005. Using protothreads for sensor node programming. In *Proceedings of the REALWSN*, Vol. 5.

[7] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 263–273.

[8] Levent Erkok. 2016. hArduino by LeventErkok. https://leventerkok.github.io/hArduino/

[9] Loe Feijs. 2013. Multi-tasking and Arduino : why and how?. In *Design and semantics of form and movement. 8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013)*, L. L. Chen, T. Djajadiningrat, L. M. G. Feijs, S. Fraser, J. Hu, S. Kyffin, and D. Steffen (Eds.). Wuxi, China, 119–127.

[10] 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.

[11] Mark Grebe and Andy Gill. 2019. Threading the Arduino with Haskell. In *Trends in Functional Programming*, David Van Horn and John Hughes (Eds.). Springer International Publishing, Cham, 135–154.

[12] Till Haenisch. 2016. A case study on using functional programming for internet of things applications. *Athens Journal of Technology & Engineering* 3, 1 (2016).

[13] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, 8–16.

[14] 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 on - RWDSL2018*. ACM Press, Vienna, Austria, 1–11. https://doi.org/10.1145/3183895.3183902

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

[16] Zhenying Li, Xiaohui Peng, Lu Chao, and Zhiwei Xu. 2018. EveryLite: A Lightweight Scripting Language for Micro Tasks in IoT Systems. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 381–386.

[17] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2018. Task Oriented Programming and the Internet of Things. In *Proceedings of the 30th Symposium on the Implementation and Application of Functional Programming Languages*. ACM, Lowell, MA, 83–94. https://doi.org/10.1145/3310232.3310239

[18] M. Lubbers, P. Koopman, and R. Plasmeijer. 2019. Multitasking on Microcontrollers using Task Oriented Programming. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija, Croatia, 1587–1592. https://doi.org/10.23919/MIPRO.2019.8756711

[19] Haidong Lv, Xiaolong Ge, Hongzhi Zhu, Zhiwei Yuan, Zhen Wang, and Yongkang Zhu. 2018. Designing of IoT Platform Based on Functional Reactive Pattern. In *2018 International Conference on Computer Science, Electronics and Communication Engineering (CSECE 2018)*. Atlantis Press.

[20] 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.

[21] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*. ACM, 195–206.

[22] Hans-Christoph Steiner. 2009. Firmata: Towards Making Microcontrollers Act Like Extensions of the Computer.. In *NIME*. 125–130.

[23] Ryan Suchocki and Sara Kalvala. 2015. Microscheme: Functional programming for the Arduino. In *2014 SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*.

[24] Christophe Troyer, de, Jens Nicolay, and Wolfgang Meuter, de. 2018. Building IoT Systems Using Distributed First-Class Reactive Programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 185–192. https://doi.org/10.1109/CloudCom2018.2018.00045

[25] Mitchell Wand. 1999. Continuation-based multiprocessing revisited. *Higher-Order and Symbolic Computation* 12, 3 (1999), 283–283.

Instr.	Args	Semantics	sp	pc

Return	rw aw	st[fp-aw-3+i] = st[fp+i]
		for all i∈{0..rw}
		fp = st[fp-aw-2]
			st[fp-aw-3+rw]	st[fp-aw-1]

JumpF	jl		sp-1	st[sp-1] ? pc+1 : jl
Jump	jl			jl

JumpSR	aw jl	st[sp-i-1] = pc+2
		fp = sp
				jl

Tailcall	w_1 w_2 jl	rotate (w_1+3+w_2,w_2) fp = fp-w_1+w_2 **where** w_1 is the width of the caller, w_2 the width of the callee	fp	jl
Arg	a	st[sp] = st[fp-1-a]	sp+1	pc+2
Push	n s	st[sp+i] = s[i] **for all** i∈{0..n}	sp+n	pc+2+n
Pop	n		sp-n	pc+2
Rot	d n	rotate (d, n) **for all** i∈{0..n}	sp-n	pc+3
Dup		st[sp] = st[sp-1]	sp-1	pc+1
PushPtrs		st[sp] = sp st[sp+1] = fp st[sp+2] = 0	sp+3	pc+1
UnOp		st[sp-1] = ⋄ st[sp-1]		pc+1
BinOp		st[sp-2] = st[sp-2] ⊕ st[sp-1]	sp-1	pc+1
MkTask	Stable_n	st[sp-n-1] = node (stable, st[sp-1], …, st[sp-n-1])	sp-n+1	pc+2
	Unstable_n	st[sp-n-1] = node (unstable, st[sp-1], …, st[sp-n-1])	sp-n+1	pc+2
	ReadD	st[sp-1] = node (readd, st[sp-1])		pc+2
	ReadA	st[sp-1] = node (reada, st[sp-1])		pc+2
	Repeat	st[sp-1] = node (repeat, st[sp-1])		pc+2
	Delay	st[sp-1] = node (delay, st[sp-1])		pc+2
	WriteD	st[sp-2] = node (writed, st[sp-1], st[sp-2])	sp-1	pc+2
	WriteA	st[sp-2] = node (writea, st[sp-1], st[sp-2])	sp-1	pc+2
	And	st[sp-2] = node (and, st[sp-1], st[sp-2])	sp-1	pc+2
	Or	st[sp-2] = node (or, st[sp-1], st[sp-2])	sp-1	pc+2
	SdsSet i	st[sp-1] = node (sdsset,i, st[sp-1])		pc+3
	SdsGet i	st[sp] = node (sdsget, i)	sp+1	pc+3
	DHTTemp i	st[sp-1] = node (dhttemp, i)	sp+1	pc+3
	DHTHumid i	st[sp-1] = node (dhthumid, i)	sp+1	pc+3
	Step aw jl	st[sp-1] = node (step, aw, jl, st[sp-1])	sp-1	pc+5

Table 1: Semantics for all instructions