MIPRO 2019, May 20-24, 2019, Opatija Croatia

Multitasking on Microcontrollers using Task Oriented Programming

Mart Lubbers, Pieter Koopman and Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
P.O. Box 9010, 6500 GL Nijmegen
Email: {mart,pieter,rinus}@cs.ru.nl

Abstract—**Microcontroller Units (MCUs) are all around us powering many of our so called *smart* devices. Most programs running on MCUs are control applications performing multiple jobs at the same time. Examples of these jobs are: blinking a status LED, reading button states, talking to sensors or communicating with the world. Often these jobs are dependent on each other and require communication between them. Small MCUs have no support for multiple threads, therefore the programmer needs to manually interleave the tasks. The job structure bears great similarities with tasks in Task Oriented Programming (TOP). Tasks representing work that needs to be done, can be interleaved and combined to form compound tasks. The embedded Domain Specific Language (eDSL) mTask is a TOP language that works on even the smallest of MCUs. This paper explains how to write multi-task control applications for MCUs using a TOP language such as mTask.**

I. Introduction

Smart devices are all around us and they are often powered by MCUs. They connect to the Internet of Things (IoT), sense the environment and act thereupon. Most of these computing devices are hidden in thermostats, smoke alarms, doorknobs, wearables, light bulbs and other household items. The MCUs powering them are cheap and energy efficient. As a consequence, they are weak in terms of computing power. A fair share of the programs for MCUs are control applications, doing many things at the same time. It is very intuitive to program these separate jobs as processes or threads. However, the severe hardware restrictions disallow the use of a feature rich OS. There are methods that allow for the specification of light-weight threads but they come with the cost of flexibility such as lack of local state or absence of possible dependencies between jobs. Without the support for threads, the programmer needs to divide all tasks in short pieces of work and manually interleave them. This is an error prone process and results in spaghetti code that is not very well maintainable.

TOP is a novel programming paradigm in which the basic building blocks are tasks. Tasks represent observable work that needs to be done by human beings or computer systems. They function as lightweight communicating threads and can be automatically interleaved because the work is inherently divided up into small rewrite steps. A programming language providing TOP for the web is iTasks [1], an eDSL hosted in Clean [2]. Data can be safely shared between tasks using Shared Data Sources (SDSs). Developing software using TOP gives the programmer a high separation of concerns.

The mTask eDSL is a TOP language for MCUs [3]. The language is shallowly embedded using class-based, or tagless [4], embedding. As a result, it supports multiple independent backends such as Arduino code generation, pretty printing, symbolic simulation and bytecode generation. The hardware requirements for mTask are very low which makes it suitable for very small MCUs.

Research Contribution: In this paper we show that many components in MCU development map directly on TOP components. Furthermore, we show by means of a case study how to write multithreaded control applications using a TOP language. This shows that TOP yields concise and maintainable MCU code. As a bonus it integrates well with TOP for the web.

II. Microcontroller Programming

This section shows some examples of traditional MCU programming and reveals a problem. The code examples will be given in the Arduino C++ dialect[1].

A. Blink

The first program one writes when trying a novel programming language often is the *Hello World!* program. Said program is unusable for MCUs since they do not have screens. Nevertheless, most MCUs do have a one-pixel screen, namely an LED. In the *blink* program, an LED is turned on and off repeatedly. Listing 1 shows a typical implementation of the blink program that blinks the built-in LED, connected to the digital General Purpose Input/Output (GPIO) pin 13 on the Arduino UNO, every 500 milliseconds.

```
void setup () {
   pinMode(13, OUTPUT);
}

void loop() {
   digitalWrite(13, HIGH);
   delay(500);
   digitalWrite(13, LOW);
   delay(500);
}
```

Listing 1. Blink in Arduino

[1]https://www.arduino.cc

An Arduino program requires two functions to be defined, namely `setup` and `loop`. The `setup` function is executed once at the boot of the device and usually contains code to initialize the global state and setup the peripherals. In this blink program the pin mode of the digital GPIO pin 13 is set to output so that it is put in low-impedance state and that it can provide substantial current. When the `setup` function is finished, the `loop` function is continuously called. It first sets the state of the digital pin to `true`, turning the LED on to be followed by waiting 500 milliseconds. After that the digital pin state is reverted to `false` just to wait another 500 milliseconds. Because the loop function is called continuously, the LED is blinking.

B. Threaded Blinking

Now say that we want to blink multiple blinking patterns on different LEDs concurrently. For example, blink three LEDs connected to pins $1, 2$ and 3 at intervals of $500, 300$ and 800 milliseconds. Intuitively you want to lift the blinking behaviour to a function and call this function three times with different parameters as done in Listing 2

```
void setup () { ... }

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

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

Listing 2. Naive approach to multiple blinking patterns in Arduino

Unfortunately, this does not work because the `delay` function blocks all further execution. The resulting program will blink the LEDs after each other instead of at the same time. To overcome this, it is necessary to slice up the blinking behaviour in very small fragments so it can be manually interleaved [5]. Listing 3 shows how three different blinking patterns might be achieved in Arduino using the slicing method. If we want the blink function to be a separate parametrizable function we need to explicitly provide all references to the required state. Furthermore, the `delay` function can not be used and polling `millis` is required. The `millis` function returns the number of milliseconds that have passed since the boot of the MCU. Some devices use very little energy when in `delay` or sleep state. Resulting in `millis` potentially affects power consumption since the processor is basically busy looping all the time. In the simple case of blinking three LEDs on fixed intervals, it might be possible to calculate the delays in advance using static analysis and generate the appropriate `delay` code. Unfortunately, this is very hard when for example the blinking patterns are determined at runtime.

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

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

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

Listing 3. Threading three blinking patterns in Arduino

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

III. Task Oriented Programming

TOP is a novel flavour of declarative and functional programming that describes work as tasks. In TOP, tasks represents an actual piece of *observable* work. The observability implies from the fact that other tasks can view the *task value* of the task in progress. Tasks are modelled as stateful rewrite engines. Single rewrite steps are very small and tasks can therefore be interleaved. Events triggering a rewrite can be anything ranging from user input, clocks or an event within the host system such as a peripheral. Tasks can be represented as trees that consist of basic or leaf tasks (e.g. reading input) and combinators, (e.g. a combinator running two tasks in parallel).

Every task emits a three state task value on every rewrite step representing either no value, an unstable value or a stable value. No value means that the task is unable to emit a complete value. E.g. a serial port without data. Unstable values represent complete values that may change in the future. These values in a basic task often represent a side effect. An example of a task emitting an unstable value is the task for reading an analog pin. A stable value never changes, often found in a finished side effect such as a finalized web form. Not all state transitions are legal, Figure 1 shows the state diagram.

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

Figure 1. State diagram for the legal transitions of task values

Tasks share data via their task values or via SDSs. SDSs form a read/write abstraction on, possibly impure, data. In iTasks there are SDSs for files, time, random data, database tables but also SDSs to provide introspection on the system such as for the list of running tasks. Tasks can register or watch an SDS, subscribing to events for the SDS. When the SDS is written to, the registered task is triggered for a rewrite. As well as tasks, SDSs can be transformed with SDS combinators and transformers.

IV. mTask

The mTask eDSL is a TOP language for MCUs [3]. It is shallowly embedded using a class-based, or tagless [4], embedding. The technique allows for multiple independent backends such as Arduino code generation, pretty printing, symbolic simulation and bytecode generation.

All constructs in mTask are expressed as type classes. A backend in mTask, e.g. a pretty printer for printing the code, is a type implementing *some* of the type classes. Adding functionality does not interfere with existing backends and is as easy as defining a new class and implement it with *some* backends. Analogously, adding a backend does not interfere with existing functionality, a new backend is just a new type implementing *some* of the classes. The eDSL functions are often the same function as in the original host language and therefore a disambiguating name is required. This is done by postfixing the function with a full stop. For example, addition in mTask is named `+.` to not clash with the existing + function in the host language.

An mTask program is always of the form `MTask v t` where `v` is the backend and `t` is the type of the task. Not all types are suitable for MCUs. Therefore, many eDSL functions have class constraints on the type of the task or expression. Amongst other things, these constraints restrict the use of functions as values, recursive data, make sure serialization is possible. Listing 4 shows an example function in mTask that *does* include this constraint together with some expression examples.

```
class arith v where
    lit  :: t → v t | type t
    (+.) :: (v t) (v t) → v t | type t & + t & zero t
    ...

fourtytwo = lit 37 +. lit 5
factorial =
    fun λfac=(λx→If (x ==. lit 0) (lit 1) (x *. fac (x -. lit 1)))
    In {main=fac (lit 5))}
```

Listing 4. Arithmetic function with class constraints

The mTask eDSL contains constructs for expressions, functions and tasks. All functions are defined on the top level, cannot be used in a curried fashion and are strict in their arguments. Functions can be defined recursively and full tail-call optimization is available. The `:: Main a = {main :: a}` is used to enforce the top-level definition restriction. To safeguard for curried use, the arguments are always represented as tuples instead of higher-arity host functions. For task support, mTask has basic tasks and task combinators. They will be explained on demand with the examples. All details regarding the exact types and the actual class definitions can be found in the repository[2].

A. Translating the Blink Program to mTask

Tasks are represented as trees and rewrite on every event that interests them. Nodes in this tree are called task combinators and leafs are called basic tasks. Rewriting is always done recursively, starting from the root node down to all the leafs. Since delays, and all other tasks, are non blocking, rewrite steps are quick. Different branches of the task tree executed are therefore highly interleaved.

While counter intuitive and not very maintainable, it is possible to literally translate the blink program to mTask as is done in Listing 5. This arises from the fact that most Arduino functions have a similar mTask counterpart. For example, sequencing tasks is done with the `>>|.` combinator. It first executes the left-hand side and when that side is stable, executes rewrites to the right-hand side. Nonetheless, the semantics of the translated program are very much different because of the different execution model. For instance, there is no notion of `setup` or `loop` code in mTask. `loop` like functionality can be simulated using the `ever` function. This task will rewrite the argument task until it is stable and when it is, it will start all over again with the original task. The `ever` function is just a regular task and therefore multiple `loop`s can be simulated with ease.

```
blink :: MTask v ()
blink = {main=ever (
            delay (lit 500)
        >>|. writeD D13 (lit True)
        >>|. delay (lit 500)
        >>|. writeD D13 (lit True)
    )}
```

Listing 5. A literal mTask translation of blink

B. Threaded Blinking

The `delay` *task* does not block the execution but *just* emits no value when the target waiting time has not yet passed and emits a stable value when the time is met. In contrast, the `delay` *function* on the Arduino is blocking which prohibits interleaving. To make code reuse possible and make the implementation more intuitive, the blinking behaviour is lifted to a recursive function instead of using the imperative `ever` construct. The function is parametrized with the current state, the pin to blink and the waiting time. Creating recursive functions like this is not possible in the Arduino language because the program would run out of stack in an instant and nothing can be interleaved. With a parallel combinator, tasks can be executed in an interleaved fashion. Therefore, blinking three different blinking patterns is as simple as combining the three calls to the `blink` function with their arguments as seen in Listing 6.

```
tblink :: MTask v ()
tblink = fun λblink=(λ(st, pin, wait)→
                delay wait
            >>|. writeD d13 st
            >>|. blink (Not st, pin, wait)) In
    {main =  blink (lit True, D1, lit 500)
        .||. blink (lit True, D2, lit 300)
        .||. blink (lit True, D3, lit 800)
    }
```

Listing 6. Threaded blinking in mTask

[2]https://gitlab.science.ru.nl/mlubbers/mtask/tags/4cows18

C. Combinators

In the examples, tasks are sequenced using the `>>|.` combinator. Introduced above, the `>>|.` combinator is a specialization of the step combinator (`>>*.`), the Swiss army knife of sequential combination (Listing 7). The right hand side is *observing* the task value of the left hand side. The value produced by the task on the left-hand side is matched against the continuations (`:: Step`) on the right-hand side. If a predicate matches, the task steps and rewrites to the task in the matched predicate.

The `>>=.` combinator is a specialization of the `>>*.` combinator and is similar to the monadic bind. If the left hand side is stable, it feeds the value to the function on the right hand side and rewrites to the resulting task. `>>|.` is similar but does not use the argument, i.e. monadic sequence.

```
(>>*.) infixl 1 :: (MTask v t) [Step v t u] → MTask v u

:: Step v t u
  = IfValue ((v t) → v Bool) ((v t) → MTask v u)
  | ...
  | Always (MTask v u)

(>>=.) l r = l >>*. [IfValue (λ_→lit True)     r]
(>>|.) l r = l >>=. [IfValue (λ_→lit True) λ_→r]
stepExample = readA A3
    >>* [IfValue (λx→x >. lit 100) λx→writeA A4 (x /. 2)]
```

Listing 7. Sequential task combination

There are two combinators in mTask to achieve parallel execution of tasks. Disjunction (`.||.`), first rewrites the left hand side and then the right hand side. Conjunction (`.&&.`), also combines two tasks but also combines the values into a tuple meaning that both sides must have a value. The stability is determined by the disjunction and conjunction respectively of the stability of the arguments. Since `blink` is not terminating, the behaviour of `.||.` and `.&&.` is identical in Listing 6.

D. Shared Data Sources

As of now, mTask supports SDSs representing global, well-typed variables but no SDS specific combinators or transformers yet. These SDSs may represent iTasks SDSs that are automatically synchronized with the server. SDSs are used to share data between tasks having no sequential relation. Listing 8 contains an example of a blink program for which the delay can be set using a button. It first defines an SDS containing the current blinking interval. This is followed by a modified blink function that first reads the number of milliseconds it needs to wait from the SDS before actually blinking. Two tasks run in parallel, the first task is the blinking itself. The second task is constantly checking the button state using the `pressed` task. If a button is pressed, the SDS containing the increment is updated.

```
sblink =
    sds λlag=500 In
    fun λblink=(λ (st, pin) →
                getSds lag
            >>=. λwait→delay wait
            >>|. writeD d13 st
            >>|. blink (Not st, pin)) In
    {main = blink (lit True, D13)
        .||. ever (
                    pressed upButton   >>* [IfValue id (rtrn (lit 100))]
              .||. pressed downButton >>* [IfValue id (rtrn (lit -100))
      ↪ ]
              >>=. λincr→getSds lag
              >>=. λoint→setSds lag (oint +. incr))}
```

Listing 8. SDSs interface

V. Thermostat

As a case study, this section will describe the development of a thermostat in mTask. The proposed thermostat has multiple jobs to take care of at the same time, namely the viewer, heater and updater. The viewer will read the temperature from the connected Digital Humidity and Temperature sensor (DHT) and show it on the connected LCD. The heater checks if the temperature is meeting the goal temperature. If the temperature is lower than the goal temperature, the heater turns on and conversely if the temperature is higher. The heating unit is connected via a relay to digital pin 0. The updater checks whether the up or down button is pressed and changes the goal temperature accordingly.

A. Defining the SDSs and Peripherals

An mTask program always starts with the global definitions such as peripherals, SDSs and functions. This preamble shows similarities with the `setup` function in Arduino but is completely declarative. There are two peripherals connected to the MCU, namely the LCD and the DHT. They are defined with a similar technique as SDSs are declared using host language lambdas so that they are used in a type-safe way. With the `dht` function, the DHT is declared, being connected to digital GPIO pin 1. The 2×16 character LCD is defined after that using the `lcd` function connected to the standard pins. There are two data sources that need to be shared between tasks and are represented as SDS. The current temperature (`temp`) and the target temperature (`goal`). Both SDSs store values of type `Int` and represent the temperature in degrees Celcius.

```
thermostat :: Main (MTask v ())
thermostat =
    DHT D1 DHT22 λdht→
    LCD 16 2 [] λlcd→
    sds λtemp=0 In
    sds λgoal=20 In
```

B. The Main Tasks

The main program is divided into four tasks that are connected using the parallel disjunction operator. The tasks very closely match the jobs that were described. All tasks are created using (recursive) functions to either have a reuse of code or to keep an intermediate state.

```
    fun λviewer=( ... ) In
    fun λheater=( ... ) In
    fun λupdate=( ... ) In
```

```
{main =  viewer (lit 0)
    .||. heater (lit False)
    .||. ever (update (lit UpButton, lit 1))
    .||. ever (update (lit DownButton, lit −1))}
```

Listing 9. Thermostat main tasks

C. Viewer

The viewer reads the temperature and communicates that to the user using the LCD. The `viewer` function is parametrized with the old temperature, which is 0 on the first call. The task it generates first reads the temperature from the DHT using the `temperature` task. This task reads the temperature continuously and emits it as an unstable task value. The `>>*.` combinator is used to only continue execution when the temperature is not the same as the old temperature. This makes sure the LCD and SDS are not continuously written. When the `>>*.` steps, the `temp` SDS is set and the value is written to the LCD. Finally a recursive call to itself is done with the new temperature so that it will continuously execute.

```
fun λviewer=(
    λoldtemp→temperature dht
        >>*. [IfValue (λt→t !=. oldtemp) (setSds temp)]
        >>=. λntemp→printAt lcd (lit 0) (lit 0) ntemp
        >>|. viewer ntemp) In
```

Listing 10. Measure and show the temperature

D. Heater

The second job is the control of the heater and this is written as a single task that calls the `heater` function. This function takes the current state as an argument and is initialized with the off state. It reads the `temp` and `goal` SDSs in conjunction instead of in sequence to make sure the latest values are always checked. If the temperature is over the goal and the heater is turned on, the heater is turned off. If the temperature is under the goal and the heater is off, the heater is turned on. After the step, the `heater` function is called recursively with the new heater state so that it will run forever.

```
fun λheater=(
    λon→getSds temp .&&. getSds goal
        >>*. [IfValue (tup λ(temp, goal)→temp >. goal &. on)
                    λ_→writeD d0 false
            ,IfValue (tup λ(temp, goal)→temp <. goal &. Not on)
                    λ_→writeD d0 true]
        >>=. λnst→heater nst) In
```

Listing 11. Control of the heater

E. Update

The third and last job is controlling the goal temperature and this is split up in two tasks that are both built using the `update` function. This task is lifted to a function solely to be reused with different arguments. The `update` function checks whether a button is pressed using the `pressed` task. This task returns an unstable `Bool` value representing the button state. If the given button is pressed, the `goal` SDS is updated with the given value. This step value is either 1 for the up button or -1 for the down button. There is no recursive call which means that the function only executes once. To make sure the buttons are always checked, the task is wrapped in an `ever`.

```
fun λupdate=(
    λ(button, step)→pressed button
        >>*. [IfValue id λ_→getSds goal]
        >>=. λnv→setSds goal (nv +. step)
        >>=. λnv→printAt lcd (lit 1) (lit 0)) nv In
{main =  viewer (lit 0)
    .||. heater (lit False)
    .||. ever (update (lit UpButton, lit 1))
    .||. ever (update (lit DownButton, lit −1))}
```

Listing 12. Monitoring the buttons

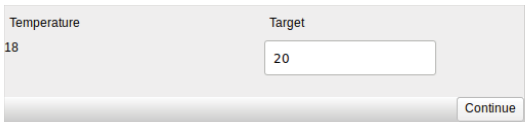
VI. Integration with an iTasks Server

Another strong point of the TOP approach is that the thermostat is programmed using an abstraction that also works for other domains. The iTasks framework is a TOP implementation for collaborative web applications and supports seamless integration with mTask [6]. SDSs can be shared between iTasks and mTask and mTask tasks can be lifted to iTasks tasks. Therefore, creating a web interface to set and view the thermostat can be done just by changing a few lines of code. The resulting program will show a web interface in which the user can change the target temperature and view the current temperature (Figure 2).

```
thermostat :: (Shared Int) (Shared Int) → Main (MTask v ())
thermostat tempref goalref =
    ...
    liftsds λtemp=tempref In
    liftsds λgoal=goalref In
    ...

webThermostat :: Task Int
webThermostat =
    withShared 0 λtemp→
    withShared 20 λgoal→
    withDevice {TTYSettings | defaultValue
            & devicePath = "/dev/ttyUSB0", baudrate = B115200} λdev→
        liftmTask dev (thermostat temp goal)
    −||− viewSharedInformation   "Temperature" [] temp
    −||− updateSharedInformation "Target"      [] goal
```

Listing 13. Integration with iTasks



18

20

Figure 2. Thermostat web interface

VII. Related Work

A. Functional Programming on Microcontrollers

Pure lazy functional programming is seldom used on MCUs because it has high memory requirements. Nevertheless, microscheme is a subset of scheme that offers a compiler for even the smallest of the Arduino family [7]. It does not support multithreading however. Functional Reactive Programming (FRP) [8] shows similarities with TOP and Juniper offers FRP for MCUs [9]. In FRP there are behaviours and events. Behaviours are declarative descriptions that hold on time and can reason over time. Events can come from within the system but can also be predicates on behaviours. TOP differs from FRP in the sense that tasks are not modelled explicitly after time but after progress.

B. EDSLs for Microcontrollers

In the eDSL field there are many solutions for providing a higher level of abstraction for programming MCUs. Grebe et al. created Haskino, a remote monad in Haskell executing imperative code on the Arduino [10]. It differs from mTask in the sense that the threads are imperative and have their own stack and thread-safe communication between them has to be done explicitly. In mTask the expressions are not interleaved, only the tasks are and therefore all communication is automatically safe because of the SDS semantics. Ivory is an eDSL written in a functional language that provides a safe C interface to program embedded systems [11]. It does not support threading.

C. Multi-threading on MCUs

Threading on MCUs is approached in multiple ways ranging in flexibility and memory requirements. Protothreads are stackless threads that provide an abstraction over state machine processes for sensor networks and can be interleaved [12]. Protothreads are similar to tasks in the sense that they can be interleaved but differ in the sense that they are imperative programs and share their stack with other threads but therefore cannot have local variables. Moreover, they must always be defined within the same function. There are also more general solutions available for doing multiple things at the same time on MCUs that generally do not work anymore on the very small ones. TinyOS is an OS designed for sensor systems that compiles to a static firmware for MCUs [13]. Threads in TinyOS differ from tasks in the sense that they are imperative and event driven using explicit atomicity annotations. Other solutions use a full-fledged Real-time Operating System (RTOS) such as ChibiOS, mbed or RTOS that just do not run on small MCUs. For example, QDuino that builds on the Quest kernel [14] or RT-Arduino, the Arduino extension offering an interface to the RTOS provided by the Erika kernel [15]. All these solutions have real threads and therefore require much faster MCUs than required for mTask.

D. Future Work

Stutterheim et al. described Task Oriented Software Development as an approach for developing TOP/iTasks applications in the multi-user web application domain [16]. The extension of mTask for integration with iTasks servers was hinted at in Section VI. This extension allows all layers of IoT to be programmed in one abstraction level, namely TOP. Different layers in TOP for web development match different stakes in software development. It might be the case that this holds for the MCU domain or both domains at the same time as well.

Gay et al. described how to implement several design patterns from Object Oriented Programming in TinyOS [17]. It would be interesting to see whether some of the design patterns have a dual in Functional and Task Oriented Programming as well by implementing them in mTask.

VIII. Conclusion

Creating multi tasking applications on MCUs is difficult with conventional means. The solutions either limits the flexibility or require much faster hardware. In TOP, tasks provide an abstraction on work that needs to be done. Task can be executed in parallel since they are automatically divided in atomic rewrite steps. The case study supports that creating multi-tasking applications for MCUs is very intuitive using a TOP language such as mTask. There is a high separation of concerns due to the fact that data, peripherals and tasks are modelled independently. The jobs that need to be done can often be literally translated to mTask tasks such that the program is very close to the design. Moreover, the seamless integration with iTasks allows entire systems to be programmed from one abstraction level.

References

[1] R. Plasmeijer, P. Achten, and P. Koopman, "iTasks: executable specifications of interactive work flow systems for the web," *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 141–152, 2007.

[2] R. Plasmeijer, M. van Eekelen, and J. van Groningen, "Clean language report version 2.2 (2011)," 2011. [Online]. Available: https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf

[3] P. Koopman, M. Lubbers, and R. Plasmeijer, "A Task-Based DSL for Microcomputers," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM Press, 2018, pp. 1–11.

[4] J. Carette, O. Kiselyov, and C. c. Shan, "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages," *Journal of Functional Programming*, vol. 19, no. 5, pp. 509–543, 2009.

[5] L. Feijs, "Multi-tasking and Arduino: why and how?" *Design and semantics of form and movement*, vol. 119, 2013.

[6] M. Lubbers, P. Koopman, and P. Rinus, "Task oriented programming and the internet of things," in *Proceedings of the 30th Symposium on Implementation and Application of Functional Programming Languages*. ACM, 2018, p. 12.

[7] R. Suchocki and S. Kalvala, "Microscheme: Functional programming for the Arduino," in *2014 SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, 2015.

[8] C. Elliott and P. Hudak, "Functional reactive animation," in *ACM SIGPLAN Notices*, vol. 32. ACM, 1997, pp. 263–273.

[9] C. Helbling and S. Z. Guyer, "Juniper: a functional reactive programming language for the Arduino," in *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, 2016, pp. 8–16.

[10] M. Grebe and A. Gill, "Threading the arduino with haskell," in *Post-Proceedings of Trends in Functional Programming*, 2017.

[11] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury, "Guilt free ivory," in *ACM SIGPLAN Notices*, vol. 50. ACM, 2015, pp. 189–200.

[12] A. Dunkels, O. Schmidt, and T. Voigt, "Using protothreads for sensor node programming," in *Proceedings of the REALWSN*, vol. 5, 2005.

[13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[14] Z. Cheng, Y. Li, and R. West, "Qduino: A multithreaded arduino system for embedded computing," in *Real-Time Systems Symposium, 2015 IEEE*. IEEE, 2015, pp. 261–272.

[15] P. Buonocunto, A. Biondi, and P. Lorefice, "Real-time multitasking in Arduino," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. Pisa: IEEE, Jun. 2014, pp. 1–4.

[16] J. Stutterheim, P. Achten, and R. Plasmeijer, "Maintaining Separation of Concerns Through Task Oriented Software Development," in *Trends in Functional Programming*, M. Wang and S. Owens, Eds. Cham: Springer International Publishing, 2018, vol. 10788, pp. 19–38.

[17] D. Gay, P. Levis, and D. Culler, "Software design patterns for TinyOS," *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 40–49, 2005.