A Task-Based DSL for Microcomputers

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

Mart Lubbers
Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands
mart@martlubbers.net

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

Abstract

The Internet of Things, IoT, makes small connected computing devices almost omnipresent. These devices have typically very limited computing power and severe memory restrictions to make them cheap and power efficient. These devices can interact with the environment via special sensors and actuators. Since each device controls several peripherals running interleaved, the control software is quite complicated and hard to maintain.

Task Oriented Programming, TOP, offers lightweight communicating threads that can inspect each other's intermediate results. This makes it well suited for the IoT. In this paper presents a functional task-based domain specific language for these IoT devices. We show that it yields concise control programs. By restricting the datatypes and using strict evaluation these programs fit within the restrictions of microcontrollers.

CCS Concepts • **Software and its engineering** → **Domain specific languages**;

ACM Reference Format:
Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *RWDSL2018: Real World Domain Specific Languages Workshop 2018, February 24, 2018, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3183895.3183902

1 Introduction

Many devices are nowadays equipped with a simple microprocessor to control their behaviour. Typical examples are thermostats, light bulbs, electric sockets, fire alarms, door openers and so on. When these devices can communicate with each other, or some remote computer, they are said to be part of the Internet of Things, IoT. The microcomputers in these devices are very affordable and becoming omnipresent. Expensive devices like cars and apparatus with a very complex task are equipped with a full-fledged embedded computer and appropriate software. For most small and relatively cheap IoT devices such an embedded computer is too expensive or consumes too much energy; a simple and cheap microprocessor is used to execute the software. These systems have typically 30 KB to 4 MB flash memory to store the program. The life of this memory is restricted to 1000 write cycles. To store variables, the heap and the stack the systems have 2 to 40 KB of RAM.

The processor speed and memory limitations exclude the use of an operating system. The apparatus just executes the program controlling the device. Even these control programs consist of several tasks. For instance, to check the state of a button ten times a second, to update a display every second, to measure the temperature two times a minute, and to switch the heating after at least five minutes unless the button is pressed earlier. Due to the different time frames and the dependencies of these tasks, the control program tends to become rather messy, independent of the programming language used.

Task Oriented Programming, TOP, offers lightweight threads that can easily be composed to more complex tasks. Tasks are evaluated step-by-step and can inspect the current value of other tasks after such a step. TOP is first implemented in the iTask system [17, 18] embedded in Clean [19]. In the iTask system, primitive tasks are gathering input via automatically generated web-form or by collecting data from other programs and data stores. A powerful set of combinators is used to compose tasks to more complex tasks. In this paper, we show that TOP is very suited for programming IoT devices. Primitive tasks deliver the current value of inputs and sensors. Constructors very similar to the iTask system are used to combine tasks to more complex tasks.

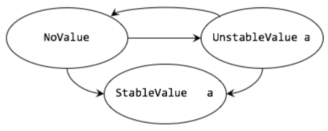
IoT devices typically have loosely dependent tasks that control the sensors, actuators and communication of the devices. Programming this in a TOP style offers concise programs. Executing these tasks within the constraints of small microcontrollers with very limited processing power and some KBs of RAM memory deserves some thought. Due to the severe limitations of the microcontrollers used we cannot port the iTask system to the IoT devices since a typical iTask program requires about 100 MB of heap space. We define an embedded Domain Specific Language, eDSL, called mTask for the IoT devices. This eDSL is embedded in the iTask system since we plan to make these TOP languages fully interoperable.

The contributions of this paper are:

- This paper introduces a task-based functional programming language for IoT devices. Compared with our previous language for microprocessor programming [16] the imperative peripheral control is replaced by referential transparent constructs.
- We demonstrate how make a *functional* extendable multi-view type-safe embedded DSL.
- The generated code runs on small and slow devices.
- Due to the use of Arduino C++ as the intermediate language, this functional eDSL runs on many different microcontrollers.

2 Task Oriented Programming

Tasks are pieces of work in a program that corresponds to the conceptual tasks to be executed by the systems and humans interacting with it. The behaviour of a task is specified by a function producing a value of type `TaskValue a` for some type `a`. Tasks will be evaluated in the updated system state over and over again until the task produces a stable value or its value becomes unused. Until a task has a stable value, it has either no value at all or an unstable value. Unstable task values can be different in subsequent evaluations. The task values and their transitions can be depicted as:



Task-oriented programming is first implemented in the iTask system implemented as a shallow embedded DSL in the strongly typed lazy functional programming language Clean. The task values are modelled in Clean by the algebraic datatype `TaskValue`.

```
:: TaskValue a  =   NoValue | Value a Stability
:: Stability    :== Bool
UnstableValue a :== Value a False
StableValue   a :== Value a True
```

A task is basically a state transformer of the task state. In a slightly simplified form this is a function that takes the current state as argument and produces a task value and an updated state.

```
:: Task a :== TaskState → (TaskValue a, TaskState)
```

Basic tasks interact with users or the environment. Typical examples are interaction with users via a webpage, read values from sensors, change the state of actuators, wait for timers, and watch the value of a Shared Data Source, SDS. Section 5.2 explains how tasks can communicate via such an SDS.

There are combinators for the parallel and sequential composition of tasks to larger tasks. The parallel composition of tasks offers a lightweight implementation of threads implementing concurrent jobs. The sequential composition models the state transitions of the (sub)system. This self-modifying task tree is evaluated over and over again until it produces a stable value.

3 DSL Design

The iTask system is a shallow DSL embedded in the lazy and pure functional programming language Clean. This implies that iTask is a library of functions and associated datatypes. For the TOP language for microcontrollers, we cannot use the same approach since we need multiple interpretations, often called views, of this DSL and the ability to extend it with abstractions of new peripherals without changing existing parts. In this paper, we define a new TOP version of the DSL mTask originally defined as an imperative DSL in [16].

In this Section, we briefly introduce the implementation technique of the DSL. This is basically a reinvention of tagless representation of Carette [5]. The advantages of this technique are that it combines strong typing, even of variables in the DSL, with multiple views and the possibility to extend the DSL with new constructs and views without touching existing code. The key idea is to use type constructor classes instead of ordinary functions to achieve the various views in a shallow embedded DSL.

As the first DSL component we define some illustrative expression components.

```
class expr v where
  lit :: t → v t | type t
  var :: ((v a)→In (v a) (v b)) → v b
  (&&.) infixr 3 :: (v Bool) (v Bool) → v Bool
  (==.) infix  4 :: (v t) (v t) → v Bool | type t

:: In a b = In infix 0 a b
```

The class variable `v` mimics the view[1]. The members of this class have type parameters such that the Hindley-Milner type system of the host language checks the types in the DSL. We do not need dependent types nor generalized algebraic datatypes. The `lit` lifts literals from Clean to mTask. By design, there is no way to convert values from mTask to plain Clean values. The class constraint `type t` ensures that the type `t` is element of the tailor made class `type`. This imposes the constraints needed in our

[1]Compared with the previous version of our mTask DSL the type parameter controlling the context in the generated code is eliminated.

DSL[2], it ensures for instance that there is an instance of `toString` and an equality. The `var` is used for typed variable introduction , e.g., `var λx = lit 3 * lit 7 In x + x`. We use `λx = e` as an alternative notation for `λx→e` in Clean, also `λx.e` is allowed to define nameless functions.

Since we need multiple interpretations of variables they have type `v t` instead of a plain value of type `t`. The infix constructor `In` makes a tuple of two values. For the Boolean AND-operator and the overloaded equality, we need new operators since the ordinary operators yield a Boolean instead of a view on such a Boolean. By convention, we will add a dot to the existing operator name in Clean. For the overloaded arithmetic operators like addition, we will use the existing overloaded operators in the required view.

3.1 Lambda Abstraction and Application

Next, we introduce a class for λ-abstraction, `L` and application, `@`. Since we need well-typed expressions there is no class member for variables. All variables are introduced by λ-abstraction [4].

```
class lambda v where
  L :: ((v a)→v b) → v (a→b)
  (@) infixl 9 :: (v (a→b)) (v a) → v b
```

This enables us to define the function `twice f x = f (f x)` as the λ-expression `L λf.L λx.f @ (f @ x)`.

These classes are sufficient to define an interesting DSL mimicking simply typed λ-calculus. By adding this class to the DSL we have shown how a DSL can be extended without touching the existing expressions.

3.2 Show View of Expressions

The simplest view transforms arithmetic expressions to a list of strings. The `ShowState` contains this list and a counter for fresh variables.

```
:: Show a = Show (ShowState → (a, ShowState))
:: ShowState = {vars :: Int, out :: [String]}
```

This state will be threaded in a monadic fashion through the expressions to be shown. For this purpose we define instances of the well know classes `Functor`, `Applicative` and `Monad` for `Show`.

```
instance Functor Show where fmap f g = pure f <*> g
instance Applicative Show where
  pure a = Show λs.(a,s)
  <*> f g = f ≫= λf. g ≫= λa. pure (f a)
instance Monad Show where
  bind (Show f) g = Show λs.let (a,t) = f s in unShow (g a) t
```

In addition we define a function `show` that adds a value to the list after applying `toString` to it.

```
show :: a → Show b | toString a
show a = Show λs.(undef, {s & out = [toString a: s.out]})
```

The function `fresh` yields a fresh variable of type `Show a`.

```
fresh :: Show (Show a)
fresh = Show λs.( show ("v" + toString s.vars)
                ,{s & vars = s.vars + 1})
```

With this tooling the instance `expr` for `Show` becomes straightforward. We use `binop` to show binary operators. This view of addition is also constructed with `binop`.

```
instance expr Show where
  lit a   = show a
  var f   = fresh ≫= λv. let (x In y) = f v in v >>|
            show " = " >>| x >>| show " in\\n" >>| y
  &&. x y = binop x "&&." y
  ==. x y = binop x "==." y

binop :: (Show a) String (Show b) → Show c
binop x f y = show "(" >>| x >>| show f >>| y >>| show ")"

instance + (Show a) | toString a where + x y = binop x "+" y
```

This transforms `var λx = lit 3 * lit 7 In x + x` to

```
  v0 = (3 * 7) in (v0 + v0)
```

Where we have glued all strings together. Note that the original name `x` is replaced by the generated name `v0`.

In the same style, we define an instance of λ-abstraction and application for `Show`.

```
instance lambda Show where
  L f   = fresh ≫= λv. show "(\\" >>| v >>|
          show "." >>| f v >>| show ")"
  @ f a = show "(" >>| f >>| show " @ " >>| a >>| show ")"
```

The λ-expression to compute twice the increment of zero

```
  var λinc = (L λx.x + lit 1) In
  var λtwice = (L λf.L λx.f @ (f @ x)) In
  twice @ inc @ lit 0
```

is shown as

```
  v0 = (λv1.(v1 + 1)) in
  v2 = (λv3.(λv4.(v3 @ (v3 @ v4)))) in
  ((v2 @ v0) @ 0)
```

3.3 Evaluation View of Expressions

For evaluating these expressions we use the `Maybe` monad as a view. Clean functions are used for variable substitutions and hence we do not need a real state. All values from the DSL are translated and evaluated to the corresponding values in Clean.

```
instance expr Maybe where
  lit a   = return a
  var f   = let (v In y) = f v in y
  &&. x y = (&&) <$> x <*> y
  ==. x y = (==) <$> x <*> y

instance + (Maybe a) | + a where + x y = (+) <$> x <*> y
```

As a consequence of this interpretation variables introduced by `var` are subject to the monomorphism restriction that gives all variables exactly one type. This makes it worthwhile to use Clean as the macro language for overloaded function in the DSL.

```
lInc   = L λx.x + lit 1
lTwice = L λf.L λx.f @ (f @ x)
main   = lTwice @ lTwice @ lTwice @ lTwice @ lInc @ lit 0
```

[2]For brevity we will often skip class constraints since the are omnipresent and makes the definitions verbose.

Evaluating `main` yields the value `65536` almost instantly.

Making these two views illustrates how to make independent interpretations of the DSL.

4 The mTask language

The language mTask is a task-based eDSL using Clean as the host language. It is strongly typed by the type system of the host language. Type safety is achieved by a shallow embedded DSL where the identifiers (like functions, function arguments and tasks) in the DSL are represented by typed identifiers in the host language. We use the expressions introduced in the previous Section and replace the `lambda`-expressions by named and parameterized functions.

Our mTask is an extendable DSL; we can add language constructs without changing any of the existing code. This is important to handle hardware extensions of the microcontrollers, like sensors and actuators. These extensions come with their tailor-made library. We want to add the interface to the library as primitives to our DSL, instead of re-implementing their functionality.

4.1 Types in mTask

A typical microprocessor contains 2 to 40 KBytes of RAM; all variables and the stack must be stored in this tiny memory. Hence, mTask currently only contains basic types and a predefined task-type. This is a deliberate choice and no inherent limitation of TOP, nor of the way we construct our DSL. To allow overloaded operations in mTask that are restricted to the types allowed we introduce two type classes. The class `type` is used for all types that exists in the eDSL mTask. This class ensures that the required operations of these types are available. Apart from the basic types, there is also an instance for task results as introduced in Section 2. The class `basicType` has only instances for the basic types: integers, Booleans, reals, characters and the void type.

```
class type t | toString, typeOf, value t
class basicType t | type t where basicType :: t

instance basicType Int, Bool, Real, Char, ()
```

4.2 Expressions

mTask is equipped with a complete set of operators on the basic types defined exactly as in the previous section. The conditionals are defined as:

```
class If v :: (v Bool) (v t) (v t) → v t | type t
class (?) infix 1 v :: (v Bool) (v t) → MTask v () | type t
```

4.3 Definitions in mTask programs

In mTask programs we allow (recursive) functions. instead of the λ-expressions from Section `lambda`. For convenience, we currently allow only functions at one level. To define this outermost level we introduce the single element record `main`.

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

This is used as `{main = lit 6 * lit 7}` in a main expression. For functions at arbitrary levels, we just have to implement lambda-lifting to transform locally defined function to the global level.

Functions with multiple views are defined by the type constructor class `fun`. To restrict the type and number of function arguments we allow only functions with exactly one argument, the class argument `a`. This argument can be void, a basic value, or any tuple of basic values we allow by defining an instance of the type class. In this way, we ensure that there are only first order functions with enables a memory efficient implementation. The function itself yields a value of type `s` in view `v`. The required function is provided by the view `v`. The result of the function definition given an appropriate view of the functions is the function itself and the main expression in which it is applied.

```
class fun a v :: ((a→v s)→In (a→v s) (Main v u))→Main v u
```

The factorial function can be defined as:

```
facDef :: Int → Main (v Int)
facDef x =
  fun λfac = (λn.If (n <. lit 2) one (n * fac (n - one)))
  In {main = fac (lit x)}
```

The scope of this function definition is the function definition and the main expression. The view has to provide an appropriate value for `fun`. Since `fun` yields a new main expression, we can introduce any number of functions needed. This is illustrated by the definition of the Fibonacci function using the auxiliary function `sub`.

```
fib =
  fun λsub = (λ(x, y).x - y) In
  fun λfib = (λn.If (n <. lit 2) one
             (fib (sub (n, one)) + fib (sub (n, lit 2)))) In
  {main = fib (lit 3)}
```

Since a function produces an object with the class restriction `type` rather than `basicType`, functions can very well be tasks.

5 Tasks in mTask

Tasks as introduced in Section 2, are a central modelling concept to structure programs in the mTask language. On one hand, tasks behave like lightweight threads; they will be executed over and over again until they are finished, or until their result is no longer needed. On the other hand, tasks behave like functions; after an evaluation step, they produce a function result of type `TaskValue a`. This function result can be used in other expressions like any other value. Tasks in mTask produce a value of the same type as tasks in the iTask system.

The state in a particular view is determined by that view. Hence, a task in mTask is a view of a task value.

```
:: MTask v a :== v (TaskValue a)
```

The `rtrn` lifts ordinary DSL values to mTask-values.

```
class rtrn v :: (v t) → MTask v t
```

5.1 Handling Input–Output Ports by Tasks

A distinguishing feature of microcomputers is the direct access to the input – output pins. As usual for these systems, we differentiate between digital pins that can be used to communicate Booleans values with the world and analogue pins. Reading from an analogue pin yields an integer generated by the analogue to digital converter of that port. Writing an integer value to such an analogue port generates a pulse width modulated signal. On most systems we can also write Boolean values to analogues ports and read Boolean values from these ports; that is, the analogue ports can be used as digital ports.

We label these pins by the members of tailor-made data types instead of using integers as their identifiers. The advantage is that we cannot use non-existing pins, but we have to adapt these types to the number of input-output pins of the microprocessor used.

```
:: DPin = D0 | D1 | D2 | D3 | D4 | D5 | ..  // digital pins
:: APin = A0 | A1 | A2 | A3 | A4 | A5       // analog pins

class aio v where           // reading and writing to analogue pins
  readA  :: (v APin) → MTask v Int
  writeA :: (v APin) (v Int) → MTask v Int

class dio p v  | pin p where // reading and writing to digital pins
  readD  :: (v p) → MTask v Bool
  writeD :: (v p) (v Bool) → MTask v Bool
```

5.2 Task Combinators

Since tasks produce a value that can be inspected, we can define operators that act based on the current value of a task. These operators can compose tasks sequentially and in parallel. The step combinator facilitates the inspection of the current value of a task.

5.2.1 Delaying Tasks

Tasks in microprocessor systems often have to wait. For instance, to prevent that they are repeated too quickly, or for signals to become stable. In the previous version of our DSL, every task had a delay before it was started. The main reason for this design was that its ease of implementation within the limits of the microcontrollers.

In the current mTask language we have a `delay` task that wait the given number of milliseconds. While the waiting time is not passed it yields the remaining time as an unstable value. When the waiting time has passed it yields the surplus waiting time as a stable value.

```
class delay v :: (v n) → MTask v n | number n
```

Many microprocessor systems work with 8 or 16-bit integers. These numbers are too small for long delays. Hence, `delay` can also have a `Long` integer of 32-bits as argument; to make the maximum delay 49 days.

5.2.2 Parallel Task Composition

We present two combinators for parallel task composition. The and-operator for tasks `.&&.` yields a stable result when both of its arguments yield a stable value. The or-operator yields a result when one of its arguments yields a stable value. These operators have both a left to right bias.

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

Whenever necessary, new operators can be added without affecting the existing code.

5.2.3 Task Result Inspection

The purpose of producing intermediate values after an evaluation step of a task is that other tasks can inspect these values and act based on these results. For this purpose, we introduce a step operator that mirrors the step operator in the iTask system. The infix operator `>>*.` has as its righthand-side argument a list of possible steps. The first steps that matches is applied and yields the specified result. When none of the steps is applicable, it produces `NoValue`, the next evaluation of the tasks re-evaluates the lefthand-side and checks all steps again.

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

:: 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)
```

The names of the steps are supposed to be self explanatory. For instance the step `IfValue (λx.x >. lit 60) (λx. x * x)` is taken if the task on the lefthand-side produces an integer value (stable or unstable) and this value is bigger than `60`; in that situation the result is `x * x` where `x` is the result of the first task. In addition to the step operator there is the combinator `ever` that will repeat the given task indefinitely.

```
class ever v :: (MTask v a) → (MTask v ())
```

Apart from the resulting type, an expression of the form `ever t` is equivalent to `t >>*. []` for any task `t`.

5.2.4 Sequential Task Composition

Although any sequential composition can be constructed with the step combinator and appropriate steps, there are additional combinators for the concise notation of frequently used consecutive compositions of tasks. The operator >>=. is similar to the monadic bind >>=; as soon as the lefthand-side produces a stable value, this value is given as the argument to the function on the righthand-side.

The operator `>>|.` switches to the task on the righthand-side as soon as the task on the lefthand-side has a stable result. The variants `>>~.` and `>>..` are the equivalents that take a step an any value, even if they are unstable.

Using these definitions we can write the 'Hello World' program for microcontrollers; the LED connected to pin `D13` blinks once every second.

```
blinkEver :: Main (MTask v ()) | mtask v ()
blinkEver =
    {main = ever (
        delay (lit 500) >>|.
        readD d13 ≫=. λstate.
        writeD d13 (Not state)
    )}
```

Instead of the `ever` combinator we can achieve repetition by a recursive task. The class `mtask` is just ensures that all basic classes defining mTask are defined for view `v`. The next example uses a function argument to store the state of the blinking LED.

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

5.3 Task Communication

Since tasks are functions we can pass the result of one task in another task by passing it as an argument to the latter task. Due to the repetition of tasks, this is often quite inconvenient. To achieve communication we need a task that terminates itself, pass the result to the place where it is needed and start a new version of the first task to execute the remainder of the work.

In TOP, tasks can also communicate via a Shared Data Source, SDS. An SDS is a named and typed object in the task state. This state is passed around during task evaluation. Tasks can communicate via an SDS by writing and reading the value of such an SDS. Since each SDS lives in the state which is passed around, communication by an SDS is referentially transparant.

A SDS can be introduced by the class `sds` of our DSL.

```
class sds v :: ((v (SDS t))→In t (Main (MTask v u)))
               → Main (MTask v u)
```

The `sds` definition yields an initial value and the main expression. We use the main expression again to enforce that SDS definitions can occur only at the outermost level. The type of the SDS is determined by its initial value. Just like the variables in Section 3 and the definitions in Section 4.3.

The value of an SDS can be obtained and modified by their getters and setters defined in the class `sdsRW`.

```
class sdsRW   v where
  getSDS :: (v (SDS t)) → MTask v t
  setSDS :: (v (SDS t)) (v t) → MTask v t
```

The mTask program `switchedBlink` illustrates the use of a SDS. The SDS `key` holds the value of a toggle key. The task `switch` repeatedly checks if the key is pressed and flips the SDS accordingly. The task `led` blinks the LED on `d13` if the SDS `key` allows this.

```
switchedBlink =
  sds λkey = True In
  fun λswitch = (λ(). mTask () (
    readA a0 ≫=. λa.
    getSds key ≫=. λk.
    a <. lit 1000 ?
      setSds key (Not k) >>|.
    delay (lit 25) >>|.
    switch ())) In
  fun λled = (λstate. mTask () (
    getSds key ≫=. λk.
    rtrn (k &. Not state) ≫=. λstate2.
    writeD d13 state2 >>|.
    delay (lit 1000) >>|.
    led state2)) In
  {main = switch () .&&. led false }
```

Note that both tasks have their own delay.

5.3.1 Indicating Task Types

The example above illustrates that task definitions in mTask are just function definitions yielding a task result. In order to type check this properly the host language must be able to solve the overloading. It is fine if to have an overloaded expression, but the compiler must be able to determine for instance if all class restrictions are fulfilled. For nonterminating recursive functions like `switch` and `led` in the example above we need to help the compiler. With the annotation `mTask` we indicate that this is a task with the given result type.

```
mTask :: a (MTask v a) → MTask v a | mtask v & fun (v a) v
mTask f t = t
```

6 Language Extensions

There is a huge variety of peripherals for microcontrollers systems available. Examples are sensors measuring temperature or distance; actuators controlling lights, motors, or displays, and units for more advanced tasks like WiFi communication, or GPS location detection.

On the hardware side, these accessories are stacked as a shield on an Arduino, or connected with a few wires to the ports of the microprocessor. To achieve useful collaboration between the microprocessor and these add-ons the desired communication protocol must be executed by the microprocessor. For this purpose, there is a tailor-made library in C++ for each peripheral type that implements the required control protocol. A program using such a device creates a control object and controls it by calling methods of this control object.

In the mTask system, we want to reuse these control libraries as first-class citizens. We neither want to redo all implementation work of these libraries in mTask nor want to introduce some foreign function interface to

C++ objects in mTask. We achieve this by introducing a language extension mimicking the API of the control objects. The construction of mTask by a set of type constructor classes facilitates this excellently.

To demonstrate this we show the classes controlling temperature sensors and Liquid-Crystal Displays. The general pattern used is that there is a constructor like a method to create the control object. The definition of these objects is very similar to the definition of functions and shares. Just like shares, we define a set tailor-made manipulation functions for the object at hand.

6.1 Temperature Sensor

The DHT11, DHT21 and DHT22 are members of a family of cheap temperature and humidity sensors [6]. Apart from the power lines, there is just a single line to control them. This line is connected to a single pin of the microprocessor. The microprocessor and the sensor use a serial communication protocol over this connection. We use the Adafruit C++ library to control the sensor [1]. This library implements this protocol and provides methods to read the temperature and humidity.

The class `dht` interfaces this library in mTask. The constructor of the DHT objects requires a pin number and the DHT-type used. The DHT-type is an enumeration type listing the available types.

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

:: DHT        = {temperature :: Real, humidity :: Real}
:: DHTtype    = DHT11 | DHT21 | DHT22
```

The actual `DHT` type is not used in most views. We have chosen to use a record type that is convenient for the simulation view.

6.2 Liquid Crystal Display

In the very same style, we define a class to interface the library controlling LCDs [2]. Here the constructor gets the dimensions of the LCD and the list of pins used in the connection as arguments. When this list is empty the standard pins of are used.

Most displays provide five buttons connected with a resistor network to pin A0. The function `pressed` checks whether a button is pressed.

```
class lcd v where
  LCD   :: Int Int [DPin] ((v LCD)→Main (v b)) → Main (v b)
  print :: (v LCD) (v t) → MTask v Int // returns bytes written
  setCursor :: (v LCD) (v Int) (v Int) → MTask v ()
  pressed   :: (v Button) → MTask v Bool

:: Button = RightButton | UpButton | DownButton | ..

printAt lcd x y z :== setCursor lcd x y >>|. print lcd z
```

The macro definition of `printAt` shows that the host language can be used to generate mTask expressions.

6.3 Example

The mTask version of a thermostat is a slightly more realistic example. The thermostat is equipped with a DHT22 temperature sensor and a two-line LCD. The first line of the display shows the actual temperature, the second line shows the goal temperature. The heating is controlled by a digital pin named heating.

The program contains three tasks running at their own speed. These tasks communicate via an SDS for the actual temperature and an SDS for the goal temperature. The task `measure` reads the actual temperate every 5 seconds from the DHT sensor and update the display and SDS named `temp`. The task `keys` checks the up- and down-button 10 times a second. This speed is a compromise between autorepeat and preventing contact dender. The `goal` temperature and the display are updated when a key is pressed. The task `control` takes care of the actual on- and off-switching of the heating. The argument of this recursive task contains the current state of the heating. When the heating is switched on it waits for a `minOnTime`. There is a similar `minOffTime` of the system when the state switches from on to off. Without a state change, this task checks 4 times a second if a state change is required.

```
thermostat =
  DHT D0 DHT22 λdht =
  LCD 16 2 [] λlcd =
  sds λgoal = 20.0 In
  sds λtemp = 0.0 In
  fun λmeasure = (λ(). mTask Int (
    temperature dht >>~. λact.
    setSds temp act >>|.
    printAt lcd Zero Zero act >>|.
    delay (lit 500))) In
  fun λnewGoal = (λg.
    setSds goal g >>|.
    printAt lcd Zero One g) In
  fun λkeys =
    (λ().mTask Int (
    getSds goal ≫=. λg.
    buttonPressed >>*.
      [IfValue ((==.) upButton)   (λ_.newGoal (g +. step))
      ,IfValue ((==.) downButton) (λ_.newGoal (g -. step))
      ] >>|. delay (lit 100))) In
  fun λcontrol = (λrunning.mTask () (
    getSds goal ≫=. λg.
    getSds temp >>*.
      [IfValue (λt.g >. t &. Not running) (λt.
        writeD heating true >>|.
        delay minOnTime >>|.
        control true)
      ,IfValue (λt.g <. t &. running) (λt.
        writeD heating false >>|.
        delay minOffTime >>|.
        control false)
      ,Always (delay (lit 250) >>|. control running)
      ])) In
  {main = control false .&&. ever (measure () .&&. keys ())}
```

This example shows clearly the nature of task based programming of microprocessor systems. It is easy to imagine and implement improvements to this thermostat. For instance, a more informative display, time-based changes of the goal temperature, using a web-based interface in addition to the LCD. After the construction of the appropriate interface definitions in mTask, these are simple extensions of the given program.

7 Views

Like in Section 3 we define views as instances of the type classes defining the DSL. Useful views include:

1. Code generation. Without actual code generation, the mTask programs cannot be executed by a microprocessor. Hence, this view is essential.
2. Pretty printing. The mTask programs are embedded in a high-level programming language. All constructs in the host language can be used to construct a mTask program dynamically. Pretty printing of the mTask program is a convenient way to show what program is actually used.
3. Simulation. It is perfectly possible to transform the mTask program to host language code by an appropriate view. When also the appropriate state is generated the mTask program can be simulated, for instance with an interactive iTask program. Such a simulation is very convenient to observe the detailed behaviour of the mTask program. In a real microprocessor, it is hard to observe the initial state and to control the sensors in such a way that the behaviour of interest can be observed.
4. Program Optimization. Just like programs in any other language it might be useful or even required to optimize mTask programs for speed or memory usage. When we implement this as a transformation from mTask programs to mTask programs all other views can be applied to the transformed program.

We only discuss the essential code generation view. It differs most from the previous version.

7.1 Code Generation View

There are very many different types of microcontrollers. These devices have a number of different instruction sets. Hence, the direct generation of machine code for a number of different platforms would be a tremendous amount of work. Fortunately, the Arduino C++ dialect is implemented for the vast majority of these devices. By generating C++ code for mTask programs and using the Arduino infrastructure our programs can be executed on many different microprocessor platforms.

In the previous version of mTask, a direct mapping from the imperative tasks without result to C-code was possible. In the current version, a more complex transformation is required. The idea is to transform the mTask programs such that a function is available for each part of the program that might be re-evaluated as a task. This implies for instance that we need tailor-made functions for the arguments of the task combinators. To make these functions high-level functions that contain all necessary information they are all lifted to the global level. This required that all function arguments used in the body of a new function must be added as arguments to the function definition and its application. Such transformations are much easier on a traditional deep-embedded language, a data structure, than on a shallow embedded representation of the language. Hence, we first define a view of mTask that produces such a data structure.

7.1.1 Syntax Tree generation

The syntax tree is a set of simple algebraic datatypes representing expressions and definitions. The tree is decorated with the type information available in the mTask version of the language. For example, expressions are represented by the type `Expr`.

```
:: Expr
  = Lit      Type String      // literal
  | Var      Type Name        // function argument
  | Sds      Type Name        // use of a SDS
  | App      Type Name [Expr] // function application
  | TaskExpr Expr             // a task expression
  | Object   Type Name        // an object like LCD
  | BindExpr Expr [StepExpr]  // step combinator »*.
```

The view `AST` will produce the abstract syntax tree, or more accurately a record with a all relevant syntax tree information.

```
:: AST a = AST (ASTstate → (a, ASTstate))
:: ASTstate =
  {expr :: Expr          // representing of last mTask expression
  ,defs :: [FunDef]      // function definitions are gathered here
  ,sdss :: [SDSDef]      // the SDS definitions of encountered
  ,objs :: [ObjectDef]   // object definitions; sensors, displays etc.
  ,libs :: [Name]        // C++ libraries needed for the objects
  ,ids2 :: [String]      // list of fresh variable names
  }
```

Since the transformation always produces an expression of type `Expr`, even when the `AST` monad requires another type, the state record contains the field `expr`. There are functions to set and get this expression as well as for the manipulation of the fields of the state record.

```
getExpr :: AST Expr
setExpr :: Expr → AST x

(>>|=) infixl 1 :: (AST x) (Expr → AST y) → AST y
(>>|=) x f = x >>| getExpr >>= f
```

For `AST a` we define the usual monadic operations; bind ≫=, Functor `fmap`, and Applicative `<*>` and `pure`.

With these tools, the transformation of arithmetic expressions from mTask to a syntax tree is very compact.

All operators become an application of the corresponding function to the transformed arguments.

```
instance arith AST where
  lit x = setExpr (Lit (typeOf x) (toString x))
  // etc.
```

Also combinator become applications of special functions. This implies that we have to be a little careful in the second phase of the compilation; not all expressions that are represented by a function application are simple strict functions.

```
instance rtrn AST where
  rtrn x = x >>|= λxt.setExpr (App (MTaskType (typeOf xt))
                                   "return" [xt])
```

For the bind operator, we immediately do the required program transformation that introduces a fresh function that is lifted to the global level and stored as function definition in the state. In this way, the task corresponding to the right-hand side of the bind becomes a function application that can be re-evaluated when this appears necessary at runtime. As actual argument for the righthand-side `f` of this combinator we use `setExpr var`; this will set the expression in the state to the newly generated variable expression.

```
instance seq AST where
  (>>=.) x f
   = x >>|= λxt.
     ((λ n."v" + n) <$> freshId) >>= λ name1.
     return (Var (deTask xt) name1) >>= λ var.
     (f (setExpr var)) >>|= λ body.
     return (collectVars name1 body) >>= λ addedArgs.
     ((λ n."f" + n) <$> freshId) >>= λ name2.
     return (App (typeOf body) name2 addedArgs) >>= λ fun.
     storeDEF (FunDef (typeOf body) name2 (addedArgs ++
                                          [var]) body) >>|
     setExpr (App (MTaskType (typeOf body)) "bind" [xt,fun])
```

The definition of an SDS, a function and objects follow a similar pattern. The appropriate definition is stored in the state. As the actual argument for the definition in mTask, we use code that generates a reference to the appropriate object within the AST monad. The simples example is the generation of the AST for an SDS.

```
instance sds AST where
  sds def =
    {main =
      ((λ n."s" + n) <$> freshId) >>= λsName.
      return (let x = value in
        K (typeOf x) (def (return (Sds x)))) >>= λsType.
      return (SdsExpr sType sName) >>= λs.
      let (g In {main = m}) = def (setExpr s) in
        return (Sds g) >>= λval.
        storeSDS (SDSDef sType sName
                   (Lit sType (toString val))) >>|
      m
    }
```

All other classes of mTask are provided with a view for the type constructor `AST`.

7.1.2 Runtime Model

The runtime model runs a task-based program under very strong memory constraints. The typical microprocessor has 2 to 40 KB of random access memory. This memory is used to store variables, the heap and the stack.

We avoid the need of a heap by restricting ourselves to basic datatypes and using a strict evaluation mechanism. The `main`-expression in mTask programs consists of a single task expression. Each task expression is either a function application or some task combinator with task expressions as arguments. This implies that a task expression is, in fact, a task tree. Nodes in the tree are either; a task returning a basic value; a function application producing a new tree node, or one of the task combinators `.&&.` or `.||.`).

Function arguments are always basic types. These values are copied when they are needed elsewhere. This implies that task nodes cannot be shared in the task expression; the task expression is a tree without sharing. This enables a simple reference counting mechanism for expression nodes. At runtime, a small array of expression nodes is allocated. The elements of this array are either free or filled with one of the possible values listed above. In all proper task programs, there is at each moment a very limited number of subtasks active. This implies that even a small array of task nodes is enough to support decent mTask programs.

7.1.3 Function Calls

In contrast to the previous implementation of mTask, functions in this version of the language are not transformed to functions in C++. Since mTask is a proper functional language, there are many recursive functions. When we transform them into C++ functions, we get the C++ runtime behaviour of these functions. Tail-call optimization is essential to run recursive functions in a small stack space.

To circumvent these problems we use a mixed approach. For C++ function from libraries, we use the ordinary C++ function calls. For function calls of mTask functions, we use our own stack. This stack contains elements of type `ARG`. This idea is based on implementation techniques from Ertl [8] and Jansen [15].

```
typedef void* Addr; // pointer to C-code
typedef union Arg {
  int i;
  bool b;
  char c;
  word w;
  Addr a;
} ARG;
```

The `stack` is just an array of these arguments and a stack pointer `sp` indicating the first free position.

```
ARG stack[STACK_SIZE];
int sp = -1;              // last used position on stack
```

The stack layout for a function call consists of

1. One `Arg` to store the function result.
2. The return address in the C-program.
3. The function arguments of this function.

For a tail recursive call we just replace the function arguments by the arguments of the new function and jump to the corresponding code. As an example, we consider the factorial program in Clean. By using an accumulator the factorial becomes a tail recursive function.

```
factorial n = fact n 1
fact n a = if (n < 2) 1 (fact (n - 1) (n * a))
Start = fac 3
```

The corresponding C-code for an Arduino is[3]:

```
void setup() {
  sp++;                          // make space for return value
  stack[++sp].a = &&next;        // push return address
  stack[++sp].i = 3;             // push function argument
  goto factorial;                // call function
next:
  Serial.println(stack[sp].i);   // print result
  return;

factorial:
  stack[++sp].i = 1;             // make accumulator
  goto fact;                     // tail call
fact:
  if (stack[sp-1].i < 2) {       // can we terminate?
    stack[sp-3] = stack[sp];     // yes; accumulator to result
    sp -= 2;                     // pop accumulator and arg
    goto *stack[sp--].a;         // return
  } else {                       // no; recursive call needed:
    stack[sp].i *= stack[sp-1].i; // update accumulator
    stack[sp-1].i -= 1;          // update n
    goto fact;                   // tail call
  }
}
```

8 Related Work

This work is a direct successor of [16]. In that paper, we introduced an extendable imperative eDSL to program microcontrollers. The control of the sensors and actuators had an imperative nature; any expression can have a side-effect on these peripherals. This language has a primitive notion of tasks; a task is basically a procedure that is executed after a user-specified delay. Since those tasks are basically a sequence of operations without a proper function result we cannot define combinators to compose these tasks. Apart from the side-effect on its actuators, a task can spawn any number of new tasks, including recursive invocations of itself. The current paper replaces the imperative tasks by referential transparent tasks with a result. The required control of peripherals is achieved in a monadic way. Since tasks have a proper function result, they can be composed by task combinators.

There is much work to port programming languages to microcomputers. The Arduino version of C++ is available for most microprocessor systems. We use it as the implementation platform for our virtual machine.

The package hArduino allows Haskell programs to control Arduino boards and peripherals, using the Firmata [12] protocol. Recent variants of this package called Haskino contain a version of Arduino-C lifted to Haskell [10]. This DSL is compiled to the corresponding C-code and loaded in the microcomputer. The Haskell program is not running on the Arduino itself. Juniper is a Haskell package for Functional Reactive Programming, FRP, for the Arduino [11]. It is implemented as a deeply embedded DSL that compiles to C. Our task-based approach is more flexible than the FRP paradigm since tasks can be created and deleted dynamically, inspect each others state, and communicate via shares.

The Ivory Language is an eDSL, embedded in Haskell, for safe systems programming made by Galois [7]. One can consider Ivory as a safer C for embedded programming. Just like our mTask system, it is an eDSL that guarantees type safety and memory safety. Differences are that Ivory is imperative, it is deeply embedded and its type system requires dependent types.

Lua [13, 14] is a powerful, fast, lightweight, embeddable scripting language ported to the ESP8266 microprocessor. The ESP8266 is far more powerful than the ATmega328P driving the Arduino, both in memory size and clock-speed. This very interesting platform costs only a few dollars and has WiFi support.

Micropython [9] is an implementation of Python 3 and parts of the standard library for microcontrollers. It is developed primary for the pyboard, a dedicated microcomputer, but ported to microcontrollers like the ESP8266 and boards based on it like the Nodemcu.

Microscheme is an implementation of a subset of Scheme for the Arduino [20]. This implementation uses a simple heap in the 2K of RAM of the Arduino. It implements proper tail calls and it offers the exception handling required by Scheme's dynamic nature. Microscheme contains a last-resort primitive for memory recovery of the form `(free! ...)`, instead of a garbage collector.

The Espruino project provides a JavaScript interpreter on single chips microprocessor boards [21]. This JavaScript interpreter is also ported to the ESP8266. The interpreter is originally designed for 128kb of Flash and 8kb of RAM. This is small in JavaScript terms, about 1000 times smaller than an ordinary interpreter, but still a factor 4 bigger than an Arduino Uno.

Feldspar is a DSL for digital signal processing embedded in Haskell [3]. Like mTask it controls low-level digital signals, but it is not task-based. None of these languages is an extendable multi-view DSL.

[3]The labels and some peephole optimizations are hand-crafted to clarify the idea and compress the code.

9 Conclusion and Future Work

Task Oriented Programming, TOP, consists of lightweight threads that produce intermediate results after each evaluation step. Tasks are repeated until they produce a stable value, or their value is no longer needed. Since there is a well-defined evaluation order of these tasks, the common problems with threads and accessing shared resources are avoided. These tasks can communicate via Shared Data Sources, this is much more convenient than communication via task results.

In this paper, we demonstrated that TOP is well suited to model the concurrent tasks executed on microcontrollers. We introduced an extendable multi-view eDSL for TOP on microcontrollers. It yields concise purely functional programs. Since more and more devices are equipped with a small processor to control their behaviour, this is a very promising way to write and maintain these programs in a cost-effective way.

Since microcontrollers have severe restrictions in memory and processing power, it is important that the task-based programs can be compiled with a small memory footprint. We have shown how a good compromise between expressibility and the required constraints is achieved by imposing restrictions on the mTask eDSL.

Although the current system is very well usable, there are many improvements possible. More complex programs require more complicated datatypes; it seems very well possible to define composed datatypes in the same way as shared data sources. Another solution is to move the more powerful processors. The booming internet of things makes these devices more and more affordable. Another desire is the communication with tasks in the iTask system. These tasks have the same notion of shares and tasks. By a proper integration task in iTask and mTask can really cooperate. Finally, there are much more peripherals and communication protocols that can and should be supported. Since mTask is designed as an extendable system this is very well possible by design.

References

[1] Adafruit. 2016. DHT-sensor-library. (2016). https://github.com/adafruit/DHT-sensor-library

[2] Arduino.cc. 2015. Arduino LiquidCrystal Library. (2015). https://www.arduino.cc/en/Reference/LiquidCrystal

[3] Emil Axelsson, Koen Claessen, Gergely DŐvai, ZoltĞn HorvĞth, Karin Keijzer, Bo LyckegÑrd, Anders Persson, Mary Sheeran, Josef Svenningsson, and AndrĞs Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms.. In *MEMOCODE*. IEEE, 169–178.

[4] H. Barendregt, W. Dekkers, and R. Statman. 2013. *Lambda Calculus with Types*. Cambridge University Press.

[5] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009). https://doi.org/10.1017/S0956796809007205

[6] D-Robotics. 2010. DHT11 Humidity & Temperature Sensor. (2010). http://www.micro4you.com/files/sensor/DHT11.pdf

[7] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt Free Ivory. *SIGPLAN Not.* 50, 12 (Aug. 2015), 189–200. https://doi.org/10.1145/2887747.2804318

[8] M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *J. Instruction-Level Parallelism* 5 (2003). http://www.jilp.org/vol5/v5paper12.pdf

[9] Damien George. 2017. (2017). http://micropython.org/

[10] Mark Grebe and Andy Gill. 2016. Haskino: A Remote Monad for Programming the Arduino. In *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings (Lecture Notes in Computer Science)*, Marco Gavanelli and John H. Reppy (Eds.), Vol. 9585. Springer, 153–168. https://doi.org/10.1007/978-3-319-28228-2_10

[11] 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 (FARM 2016)*. ACM, New York, NY, USA. https://doi.org/10.1145/2975980.2975982

[12] Jeff Hoefs. 2014. Firmata protocol. (2014). http://firmata.org/wiki/Main_Page

[13] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua – an Extensible Extension Language. *Softw. Pract. Exper.* 26, 6 (June 1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2006. *Lua 5.1 Reference Manual*. Lua.Org.

[15] Jan Martin Jansen and John van Groningen. 2016. A Portable VM-based Implementation Platform for Non-strict Functional Programming Languages. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages (IFL 2016)*. ACM, New York, NY, USA. https://doi.org/10.1145/3064899.3064903

[16] Pieter Koopman and Rinus Plasmeijer. 2016. A Shallow Embedded Type Safe Extendable DSL for the Arduino. In *Revised Selected Papers of the 16th International Symposium on Trends in Functional Programming - Volume 9547 (TFP 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 104–123. https://doi.org/10.1007/978-3-319-39110-6_6

[17] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the ICFP'07*, Ralf Hinze and Norman Ramsey (Eds.). ACM, Freiburg, Germany.

[18] 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 (PPDP '12)*. ACM, New York, NY, USA, 195–206. https://doi.org/10.1145/2370776.2370801

[19] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2011. Clean language report (version 2.2). (2011). http://clean.cs.ru.nl/Documentation.

[20] Ryan Suchocki and Dr. Sara Kalvala. 2014. Microscheme: Functional programming for the Arduino. In *Proceedings of the 2014 Scheme and functional programming workshop*, Jason Hemann and John Clements (Eds.). California Polytechnic State University, 21–29. http://www.schemeworkshop.org/2014/IndianaCSTR718.pdf

[21] Gordon Williams. 2015. The Espruino project. (2015). http://www.espruino.com/