Embedded Domain Specific Languages, part 4/4

Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

(based on slides by Pieter Koopman)

What this lecture is about

- Shallowly Embedded Domain Specific Languages
- + multiple views
- + case study: mTasks

DSL embedding techniques so far

```
:: AEx = Int Int | Add AEx AEx | V Var
:: BEx = Eq AEx AEx | Not BEx
```

+ type safe, apart from variables
+ simple
+ multiple views
- no overloading, hard to add types
- unchecked variables

```
lit :: a -> Ex a
add :: (Ex Int) (Ex Int) -> Ex Int
eq  :: (Ex b)   (Ex b)   -> Ex Bool | == b
def :: ((Var a) -> In a (Ex b)) -> Ex b
```

+ type safe
+ simple
- **single view**
+ overloading, easy to add types and operators
+ **checked variables**

```
:: Ex a = Lit a | V Var
        | Add     (BM a Int)  (Ex Int) (Ex Int)
        | E.b: Eq (BM a Bool) (Ex b) (Ex b) & == b
```

+ type safe, apart from variables

? less simple (GADTs remove the burden of adding bi-maps)

+ multiple views
+ **overloading**, easier to add types
- unchecked variables

Combining the best DSL properties

- We need functions with different interpretations
  - + type safe
  - + multiple views
  - + overloading
  - + checked variables
- Type classes offer just that!

```
class zero a :: a
instance zero Int    where zero = 0
instance zero String where zero = ""
instance zero Char   where zero = '0'
```

- Actually we need type constructor classes

```
class unit v :: a -> v a
instance unit [] where unit a = [a]
instance unit {} where unit a = {a}
instance unit ?  where unit a = ?Just a
```

From shallow to class-based embedding

```
import Data.Error
import qualified Data.Map
import Text => qualified join
```

- Common monadic state handling

```
:: Eval  a   =: Eval (State -> (MaybeError String a, State))
:: State    :== 'Data.Map'.Map Int Dynamic
:: Print a   =: P    (PS -> (a, PS))
:: PS         = {i :: Int, ind :: Int, out :: [String]}

runE (Eval f) = fst (f 'Data.Map'.newMap)
runP (P    f) = 'Text'.join "" (reverse (snd (f {i=0, ind=0, out=[]})).out)
```

- no reification
- views are independent

Shallow embedding

```
lit :: a -> Eval a
lit i = pure i
```

Class-based embedding

```
class    aexpr v     where lit :: a -> v a | toString a
instance aexpr Eval  where lit a = pure  a
instance aexpr Print where lit a = print a
```

Eval tooling, part 1/2

```
:: Eval a =: Eval (State -> (MaybeError String a, State))
```

```
instance Monad Eval
   where bind


instance Functor Eval
   where fmap

instance <*> Eval
   where (<*>)

instance pure Eval
   where pure

instance MonadFail Eval
   where fail
```

```
import Data.Functor
import Control.Applicative
import Control.Monad
import Control.Monad.Fail
```

Eval tooling, part 1/2

```
:: Eval a =: Eval (State -> (MaybeError String a, State))

instance Monad Eval
   where bind (Eval f) g = Eval \s = case f s of
                                     (Ok a, t) = let (Eval h) = g a in h t
                                     (Error e, t) = (Error e, t)

instance Functor Eval
   where fmap f (Eval g) = Eval \s = let (a, t) = g s in (fmap f a, t)

instance <*> Eval
   where (<*>) f x = f >>= \g = x >>= \a = pure (g a)

instance pure Eval
   where pure a = Eval \s = (Ok a, s)

instance MonadFail Eval
   where fail m = Eval \s = (Error m, s)
```

```
import Data.Functor
import Control.Applicative
import Control.Monad
import Control.Monad.Fail
```

Eval tooling, part 2/2

```
:: Eval a =: Eval (State -> (MaybeError String a, State))
```

```
read :: (Var a) -> Eval a | TC a
read
```

```
write :: (Var a) a -> Eval a | TC a
write
```

```
fresh :: Eval (Var a)
fresh
```

Eval tooling, part 2/2

```
:: Eval a =: Eval (State -> (MaybeError String a, State))
```

```
read :: (Var a) -> Eval a | TC a
read v
  = Eval \s = case 'Data.Map'.get v s of
          ?Just (x::a^) = (Ok x, s)
          ?Just x = (Error ("Var " <+ v <+ " of wrong type"), s)
          ?None   = (Error ("Var " <+ v <+ " undefined"), s)

write :: (Var a) a -> Eval a | TC a
write v x = Eval \s = (Ok x, 'Data.Map'.put v (dynamic x) s)

fresh :: Eval (Var a)
fresh = Eval \s = (Ok ('Data.Map'.mapSize s), s)
```

Print tooling, part 1/2

```
:: Print a =: P (PS -> (a, PS))
:: PS       = {i :: Int, ind :: Int, out :: [String]}
```

```
instance Monad Print
   where bind
instance Functor Print
   where fmap
instance <*> Print
   where (<*>)
instance pure Print
   where pure
instance MonadFail Print
   where fail
```

Print tooling, part 1/2

```
:: Print a =: P (PS -> (a, PS))
:: PS      = {i :: Int, ind :: Int, out :: [String]}
```

```
instance Monad Print
   where bind (P f) p = P \s = let (x, t) = f s; (P g) = p x in g t
instance Functor Print
   where fmap f p = p >>= pure o f
instance <*> Print
   where (<*>) pf px = pf >>= \g = px >>= pure o g
instance pure Print
   where pure a = P \s = (a, s)
instance MonadFail Print
   where fail s = print ("(fail \"" <+ s <+ "\")")
```

Print tooling, part 2/2

```
:: Print a =: P (PS -> (a, PS))
:: PS       = {i :: Int, ind :: Int, out :: [String]}
```

```
print     :: a -> Print b | toString a
print
printNL   :: Print a
printNL
freshVar  :: Print (Var a)
freshVar
inc       :: Print a
inc
dec       :: Print a
dec
```

Print tooling, part 2/2

```
:: Print a =: P (PS -> (a, PS))
:: PS        = {i :: Int, ind :: Int, out :: [String]}
```

```
print     :: a -> Print b | toString a
print a   = P \s = (undef, {s & out = [toString a : s.out]})
printNL   :: Print a
printNL   = P \s = (undef, {s & out = ["\n" <+ repeatn (s.ind*2) ' ' : s.out]})
freshVar :: Print (Var a)
freshVar = P \s = (s.i, {s & i = s.i + 1})
inc       :: Print a
inc       = P \s = (undef, {s & ind = s.ind + 1})
dec       :: Print a
dec       = P \s = (undef, {s & ind = s.ind - 1})
```

Arithmetic operations: reuse Clean operators

```
instance + (Eval a) | + a where (+) x y = (+) <$> x <*> y
instance - (Eval a) | - a where (-) x y = (-) <$> x <*> y
instance * (Eval a) | * a where (*) x y = (*) <$> x <*> y
instance / (Eval a) | /, zero, == a where
 (/) x y = (/) <$> x <*>
           (y >>= \m = if (m==zero) (fail "divide by 0") (pure m))

instance  + (Print a) | + a
   where (+) x y = print "(" >>| x >>| print "+" >>| y >>| print ")"
instance  - (Print a) | - a
   where (-) x y = print "(" >>| x >>| print "-" >>| y >>| print ")"
instance  * (Print a) | * a
   where (*) x y = print "(" >>| x >>| print "*" >>| y >>| print ")"
instance  / (Print a) | / a
   where (/) x y = print "(" >>| x >>| print "/" >>| y >>| print ")"
```

Boolean expressions

this can also be 3 separate classes

```
class bexpr v where
  (&.) infixr 3 :: (v Bool) (v Bool) -> v Bool
  (|.) infixr 2 :: (v Bool) (v Bool) -> v Bool
  ~.            :: (v Bool) -> v Bool

instance bexpr Eval where
  (&.) x y = x >>= \b = if b y (pure b)
  (|.) x y = x >>= \b = if b   (pure b) y
  ~.   x   = not <$> x

instance bexpr Print where
  (&.) x y = print "(" >>| x >>| print "&." >>| y >>| print ")"
  (|.) x y = print "(" >>| x >>| print "|." >>| y >>| print ")"
  ~.   x   = print "(~. " >>| x >>| print ")"
```

Comparisons

this can also be added to bexpr

```
class eexpr v where
  (==.) infix 4 :: (v a) (v a) -> v Bool | toString, == a
  (<.)  infix 4 :: (v a) (v a) -> v Bool | toString, <  a
  // for free
  (!=.) infix 4 :: (v a) (v a) -> v Bool | toString, == a & bexpr v
  (!=.) x y = ~. (x ==. y)
  (>.)  infix 4 :: (v a) (v a) -> v Bool | toString, <  a & bexpr v
  (>.) x y = y <. x
  (<=.) infix 4 :: (v a) (v a) -> v Bool | toString, <  a & bexpr v
  (<=.) x y = ~. (y <. x)
  (>=.) infix 4 :: (v a) (v a) -> v Bool | toString, <  a & bexpr v
  (>=.) x y = ~. (x <. y)
```

Comparisons implementations

```
class eexpr v where
  (==.) infix 4 :: (v a) (v a) -> v Bool | toString, == a
  (<.)  infix 4 :: (v a) (v a) -> v Bool | toString, <  a

instance eexpr Eval where
  (==.) x y = (==) <$> x <*> y
  (<.)  x y = (<)  <$> x <*> y

instance eexpr Print where
  (==.) x y = print "(" >>| x >>| print "==." >>| y >>| print ")"
  (<.)  x y = print "(" >>| x >>| print "<."  >>| y >>| print ")"
  (>.)  x y = print "(" >>| x >>| print ">."  >>| y >>| print ")"
```

Statements

```
class stmt v where
  skip            ::  v ()
  (:.) infixr 1 :: (v a)    (v b)         -> v b
  If              :: (v Bool) (v a) (v a) -> v a
  While           :: (v Bool) (v a)       -> v ()
```

Statements implementations

```
class stmt v where
  skip              ::  v ()
  (:.) infixr 1 :: (v a)     (v b)         -> v b
  If                :: (v Bool) (v a) (v a) -> v a
  While             :: (v Bool) (v a)       -> v ()
instance stmt Eval where
  skip          =
  (:.)  s t     =
  If    c t e =
  While c b     =
instance stmt Print where
  skip          =
  (:.)  s t     =
  If    c t e =
  While c b     =
```

Statements implementations

```
class stmt v where
  skip              ::  v ()
  (:.) infixr 1 :: (v a)     (v b)        -> v b
  If                :: (v Bool) (v a) (v a) -> v a
  While             :: (v Bool) (v a)       -> v ()
instance stmt Eval where
  skip          = pure ()
  (:.)   s t    = s >>| t
  If     c t e = c >>= \b = if b t e
  While c b    = If c (b :. While c b) skip
instance stmt Print where
  skip          = print "skip"
  (:.)   s t    = s >>| print ":." >>| printNL >>| t
  If     c t e = print "If " >>| c >>| print " " >>| t >>| print " " >>| e
  While c b    = print "While "  >>| c >>|
                 inc >>| printNL >>| b >>| dec
```

Variables

```
class vars v a where
  var            ::  (Var a) -> v a
  def            :: ((Var a) -> In a (v b)) -> v b
  (=.) infixr 2 ::  (Var a) (v a) -> v a

:: In a b = In infix 0 a b

instance vars Eval a | TC a where
  var v          = read v
  def f          = fresh >>= \v = let (a In e) = f v in write v a >>| e
  (=.) v e       = e >>= write v
```

Printing variables

```
instance vars Print a | toString a where
  var v     = printVar v
  def f     = freshVar >>= \v =
                 let (i In e) = f v
                  in print "def " >>| printVar v >>| print " = " >>|
                     print i >>| print " In " >>| printNL >>| e
  (=.) v e = printVar v >>| print " =. " >>| e

printVar :: (Var a) -> Print a
printVar i = P \s = (undef, {s & out = ["v" <+ i : s.out]})
```

Example

```
fac :: Int -> v Int | aexpr, bexpr, eexpr, stmt v
                    & *, - (v Int)
                    & vars  v Int

fac x = def \r = 1 In
        def \n = x In
        While (lit 1 <. var n) (
          r =. var r * var n :.
          n =. var n - lit 1
        ) :.
        var r
```

like previous lecture; Clean lambda bindings fix the required scoping rules

```
def v0 = 1 In
def v1 = 5 In
While (1<.v1)
  v0 =. (v0*v1):.
  v1 =. (v1-1):.
v0
```

```
Start = runP (fac 5)
Start = runE (fac 5)
```

```
Ok 120
```

All wonderful, where is the problem?

- Inspecting arguments is nasty, they are functions
- Program transformations like optimization are tricky

```
:: Opt v a = Val a | Exp (v a)

opt :: (Opt v a) -> v a | aexpr v & toString a
opt (Val t) = lit t
opt (Exp e) = e

instance aexpr (Opt v) | aexpr v
   where lit a = Val a

instance * (Opt v a) | aexpr v & * (v a) & toString a
   where (*) x y = Exp (opt x * opt y)
```

Optimization of addition and subtraction

```
instance  + (Opt v a) | zero, ==, +, toString a & aexpr v & + (v a)
    where (+) (Val n) (Val m) = Val (n + m)
          (+) (Val n) (Exp m)
              | n == zero       = Exp m
              | otherwise       = Exp (lit n + m)
          (+) (Exp n) (Val m)
              | m == zero       = Exp n
              | otherwise       = Exp (n + lit m)
          (+) (Exp n) (Exp m) = Exp (n + m)
instance  - (Opt v a) | zero, ==, -, toString a & aexpr v & - (v a)
    where (-) (Val n) (Val m) = Val (n - m)
          (-) n (Val m)
              | m == zero       = n
              | otherwise       = Exp (opt n - lit m)
          (-) n m               = Exp (opt n - opt m)
```

A functional DSL

class-based embedding

Functional DSL

- We do not need a state in the evaluation

```
:: Val a = OK a | Err String // or import from Data.Error

instance Monad     Val where bind


instance Functor   Val where fmap
instance <*>       Val where (<*>)
instance pure      Val where pure
instance MonadFail Val where fail
```

Functional DSL

- We do not need a state in the evaluation

```
:: Val a = OK a | Err String // or import from Data.Error

instance Monad     Val where bind f g = case f of
                                         (OK a) = g   a
                                         Err s  = Err s
instance Functor   Val where fmap  f p = p >>= pure o f
instance <*>       Val where (<*>) f x = f >>= \g = x >>= pure o g
instance pure      Val where pure a = OK a
instance MonadFail Val where fail s = Err s

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

…

Functional DSL

```
class    If m :: (m Bool) (m a) (m a) -> m a
instance If Val where If c t e = c >>= \b = if b t e
```

- Definitions very similar to shallow embedding

```
class    def m a :: (a -> In a (m b)) -> m b
instance def Val a where def f = let (b In e) = f b in e
```

- Example (identical to shallow embedding, apart from the type)

```
fac n
  = def \f = (\n = If (n ==. lit 0) (lit 1) (n * f (n - lit 1)))
    In f (lit n)

Start :: Val Int
Start = fac 5
```

```
OK 120
```

Printing the functional DSL

- Reuse the Print machinery from above

```
instance + (Print a) | + a where
  (+) x y = print "(" >>| x >>| print "+" >>| y >>| print ")"
instance If Print where
  If c t e = print "(If " >>| c >>| print " " >>| t >>|
                print " " >>| e >>| print ")"
```

Printing function definitions

```
e2 = def \f = (\x = x + lit 1) In f (lit 0)
```

fresh name (freshVar)

generate arguments here (app)

use actual arguments here (body)

```
instance def Print a | app, body a where
 def f
   = freshVar >>= \n =
     let (e1 In e2) = f (app (print "(" >>| printvar n)) in
         print "\ndef " >>| printvar n >>| print " = " >>|
         body e1 >>| print " In " >>| e2

Start = runP (fac 5)
```

```
def v0 = \v1 = (If (v1==.0) 1 (v1*(v0 (v1-1)))) In (v0 5)
```

Printing function definitions – the helpers

- Print variable name

```
printvar i = print ("v" <+ i)
```

- Print the function arguments

```
class    app a :: (Print c) -> a
instance app (Print a)
   where app s = s >>| print ")"
instance app ((Print a) -> b) | app b
   where app s = \x = app (s >>| print " " >>| x)
```

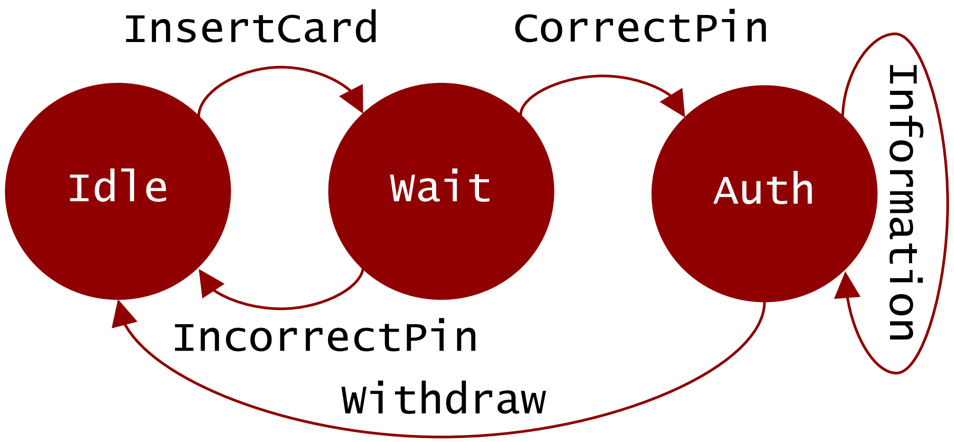
- Print arguments in the body

```
class    body a :: a -> (Print c)
instance body (Print a)
   where body s = s >>| print ""
instance body ((Print a) -> b) | body b
   where body f = freshVar >>= \v =
                  print "\\" >>| printvar v >>| print " = " >>|
                  body (f (printvar v))
```

no-op, needed for type checking

Tagless state machine DSL

the ATM example



same type check of transitions as last week

```
class trans v where
   insertCard     :: v Idle Wait
   correctPin     :: v Wait Auth
   incorrectPin   :: v Wait Idle
   information    :: v Auth Auth
   withdraw       :: Int -> v Auth Idle
   (:.) infixl 1 :: (v a c) (v c b) -> v a b
```

```
t1 :: v Idle Idle | trans v
t1 =
 insertCard  :. correctPin :.
 information :. withdraw 42
```

```
:: Print a b =: P [String]
instance trans Print where
   insertCard    = P ["insert card"]
   correctPin    = P ["correct pin"]
   incorrectPin = P ["incorrect pin"]
   information   = P ["information"]
   withdraw n    = P ["withdraw",toString n]
   (:.) (P x) (P y) = P (x ++ y)
```

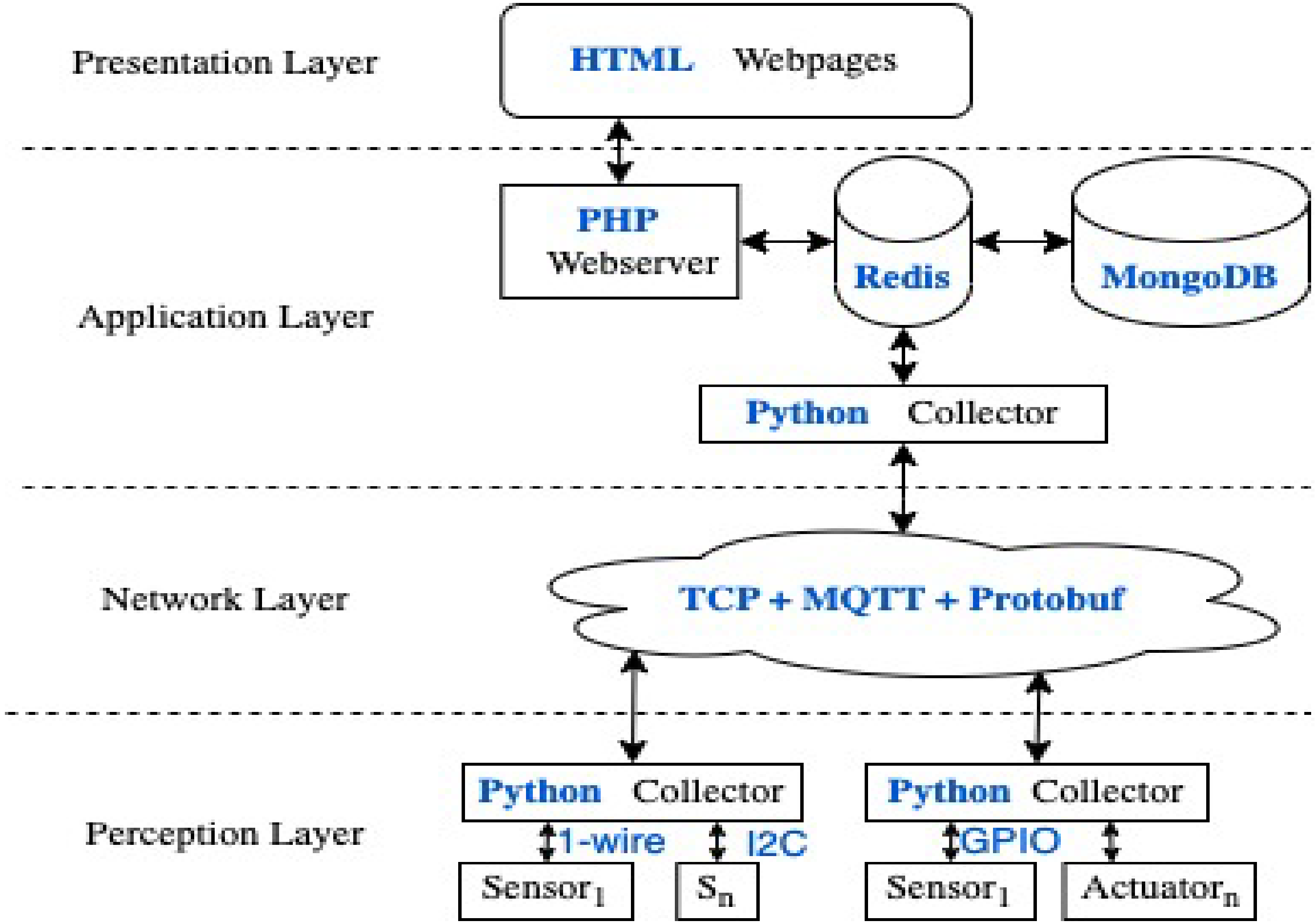
```
Start = print t1
print (P l) = l
```

```
["insert card","correct pin",
"information","withdraw","42"]
```

mTask: TOP for the IoT

a real-world application

The IoT Development Grief



- **distributed heterogeneous** system
- many languages and protocols
  - Python, PHP
  - TCP, MQTT, Protobuf
  - HTML, JSON,
  - Redis, MongoDB
  - I2C, 1-Wire, GPIO

\+ flexible

\- complex

\- semantic friction

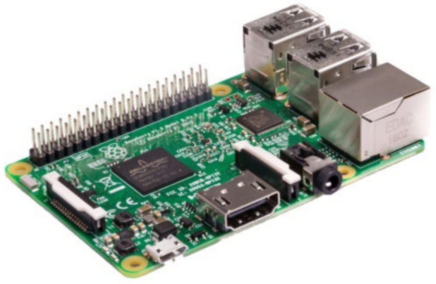
\- problems detected at runtime

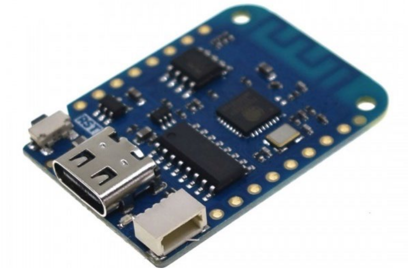
\- maintenance is very hard

iTask offers a single source solution

IoT devices: single-board computers vs. microcontrollers

	Raspberry Pi 3	Wemos D1 mini
price	60 €	6 €
energy	4 W	0.4 W
volatile fast memory	2,000 MB	0.05 MB
flash memory (wears)	32,000 MB	4 MB
CPU speed	1,400 MHz	80 MHz
word	64 bits	32 bits
WiFi	✗	✓
operating system	✓ Pi OS	✗ *





* we can use FreeRTOS

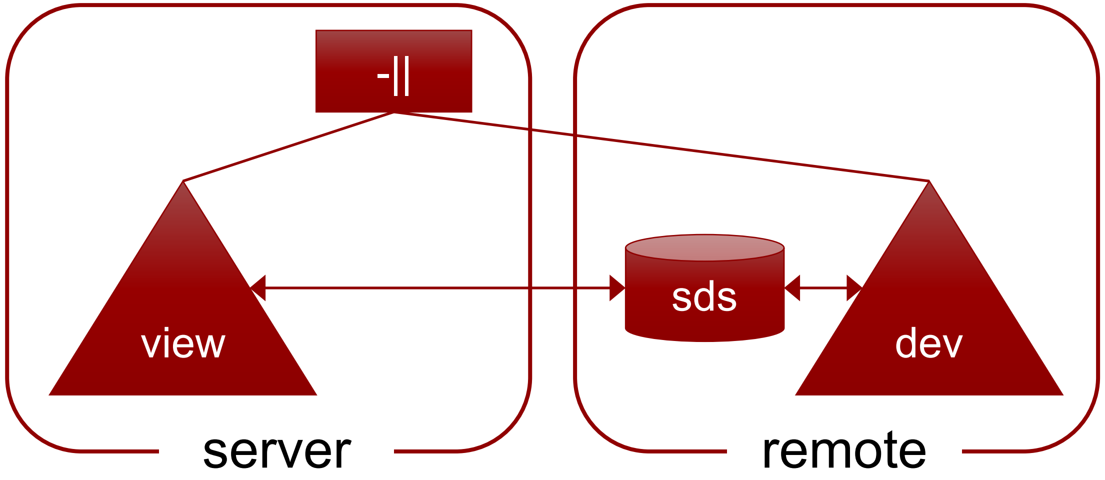
- Microcontrollers are fine IoT edge devices

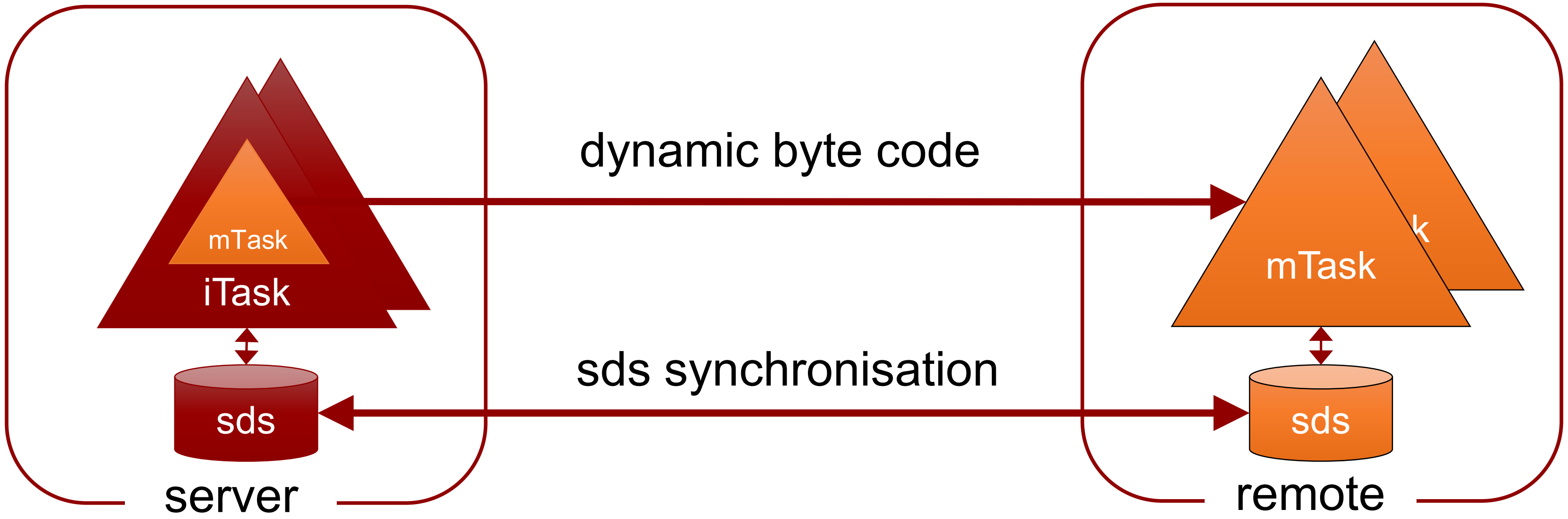
+ Price and energy consumption are excellent, Wi-Fi included

- Memory and speed are limited, which **has an impact on the software**

The need for mTask

- Remote task on a device like the Wemos D1
- Challenge: limited resources
  - processor is too slow
  - memory is too small (4 MB flash and 50 KB RAM)
  - tasks are too dynamic to store in flash (wear)
- Solution: mTask: restricted version of iTask

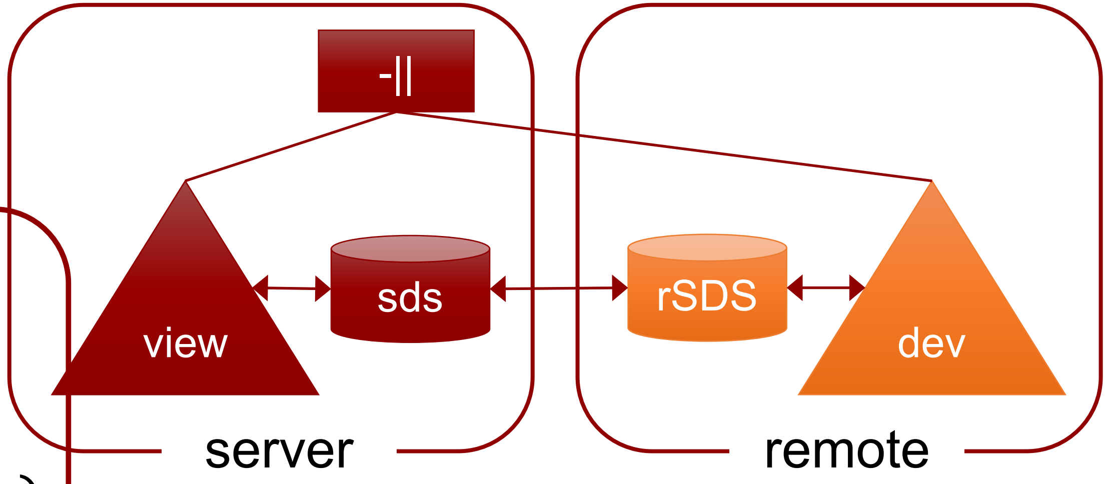


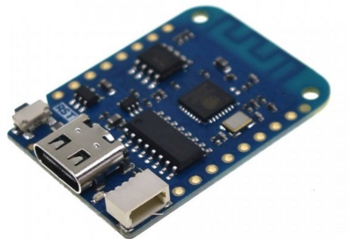


byte code interpreter + tiny OS

TOP by example: mTask temperature sensor

```
devTask :: Main (MTask v Bool) | mtask v
devTask
  = liftsds \rSDS = tempSDS In
    DHT DHT_I2C \dht =
    fun \measure = \old =
      temperature dht >>~. \new =
        If (new  !=. old)
          (setSds rSDS new >>|. measure new)
          (measure old) In
    {main = getSds rSDS >>~. measure}
```

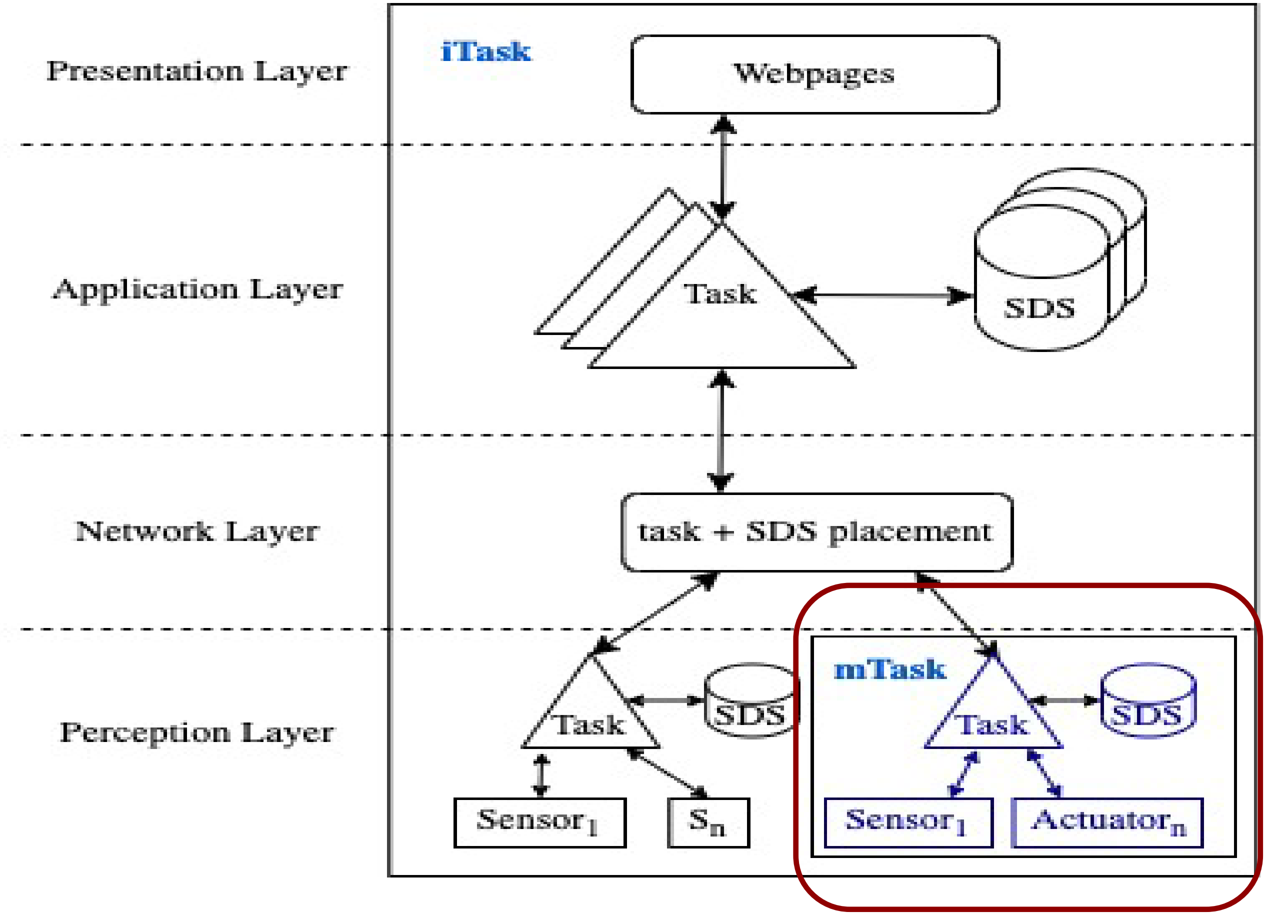




- code is dynamically compiled to byte code
- shipped to selected device
- run by the mTask OS

this is a tagless DSL for multiple views

Task-Oriented Programming for the IoT



mTask architecture

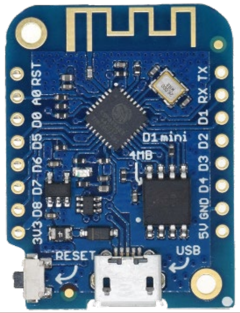
- single source for all code
  + typed: no runtime errors
  + no version problems
  + no semantic friction
- a separate part for edge node
  - runtime compiled to bytecode
  - runtime shipment to device
  - bytecode interpreter on device featherlight **domain-specific OS**
  - tasks are stored in RAM, prevents wear of flash memory

Sleeping modes Wemos D1 mini (ESP8266 µc)

item	active	modem sleep	light sleep	deep sleep
WiFi radio	on	**off**	off	off
CPU	on	on	**pending**	**off**
RAM	on	on	on	**off**
power used	300-700mW	50mW	1.5mW	0.005mW

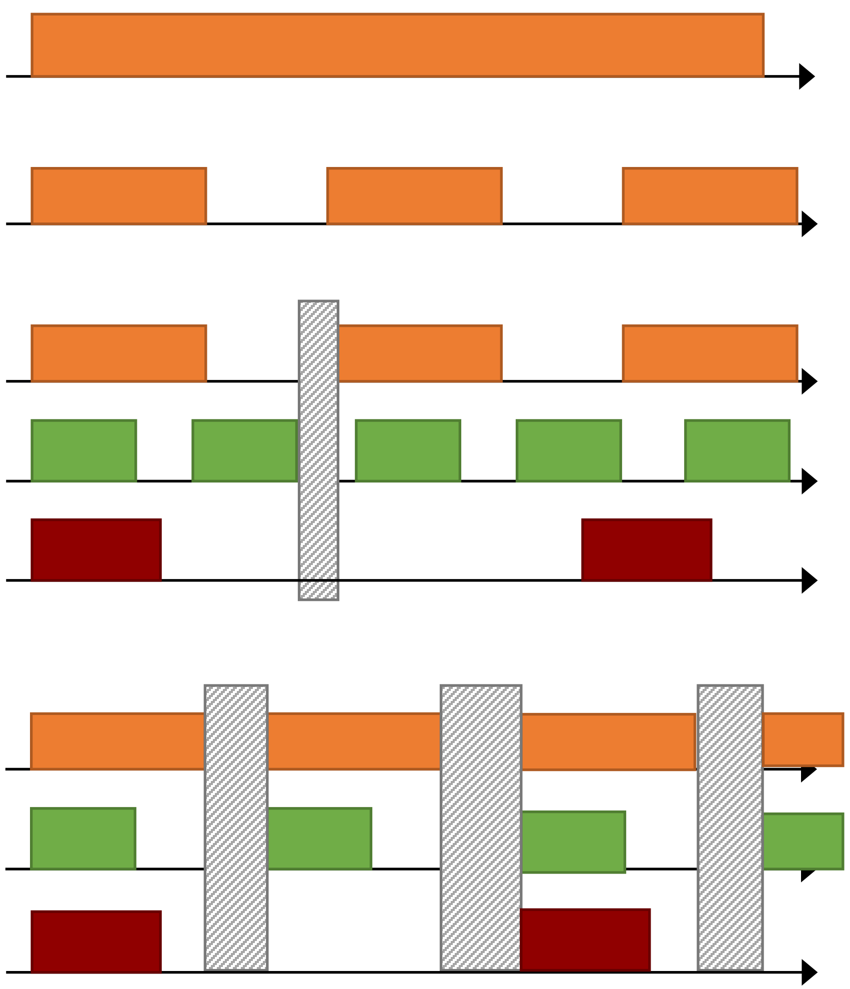
deep sleep erases the current state of the RAM on some devices

- All you need is sleep to save power
  - taking a nap can save energy
- Most tasks do not require 100% active µc
- Efficient µc requires only 1 mW during operation, even less during their sleep



Synchronised sleeping

- Repeating tasks at full speed is often not needed
- Adding delays enables sleep of tasks
  - e.g. measure the temperature once every 2 seconds
  - programmer can add delays
- The device can only sleep when all tasks are idle
  - this is at least difficult to program
  - impossible to predict for unrelated tasks
- Solution: execution regions
  - e.g. measure the temperature between 0 and 2 seconds
  - mTask adds execution regions to all tasks
  - delay task as long as possible/useful, but not longer
  - the mTask OS does this automatically



Assign a refresh rate to each task

task	interval in ms
temperature	<0, 2000>
gesture sensor	<0, 1000>
sound, light	<0, 100>
read SDS	<0, 2000>
read GPIO	<0, 100>
write SDS/GPIO	<0, 0>

based on expected change rates

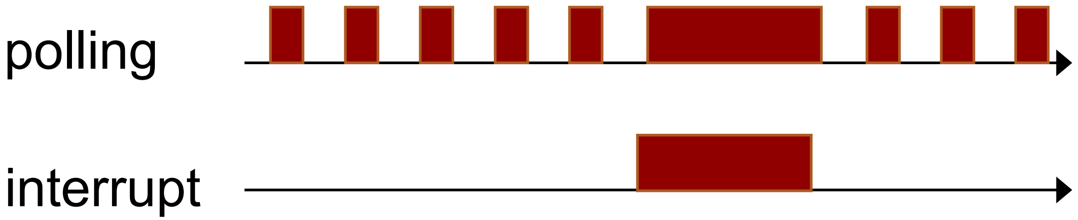
$$\mathcal{R} :: (MTask\ v\ a) \rightarrow \langle Int, Int\rangle$$

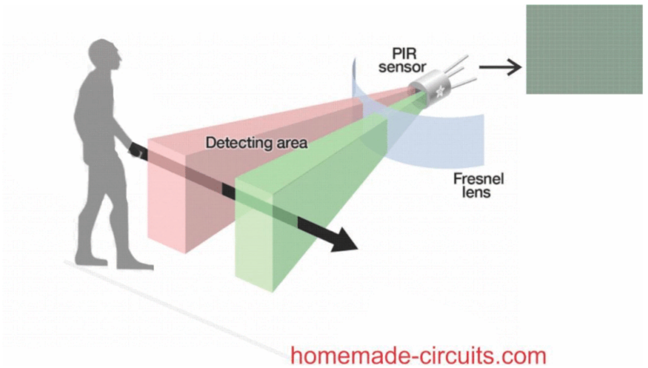
$$\begin{aligned}
\mathcal{R}(t_1\ .||.\ t_2) &= \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \\
\mathcal{R}(t_1\ .\&\&.\ t_2) &= \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \\
\mathcal{R}(t_1 >> |.\ t_2) &= \mathcal{R}(t_1) \\
\mathcal{R}(t >>= .\ f) &= \mathcal{R}(t) \\
\mathcal{R}(t >> *.\ [a_1 \ldots a_n]) &= \mathcal{R}(t) \\
\mathcal{R}(rpeat\ t) &= \langle 0, 0\rangle \\
\mathcal{R}(rpeatEvery\ d\ t) &= \langle 0, 0\rangle \\
\mathcal{R}(delay\ d) &= \langle d, d\rangle \\
\mathcal{R}(t) &= \begin{cases} \langle \infty, \infty\rangle & \text{if } t \text{ is Stable} \\ \langle r_l, r_u\rangle & \text{otherwise} \end{cases}
\end{aligned}$$

Interrupts = event-driven computation

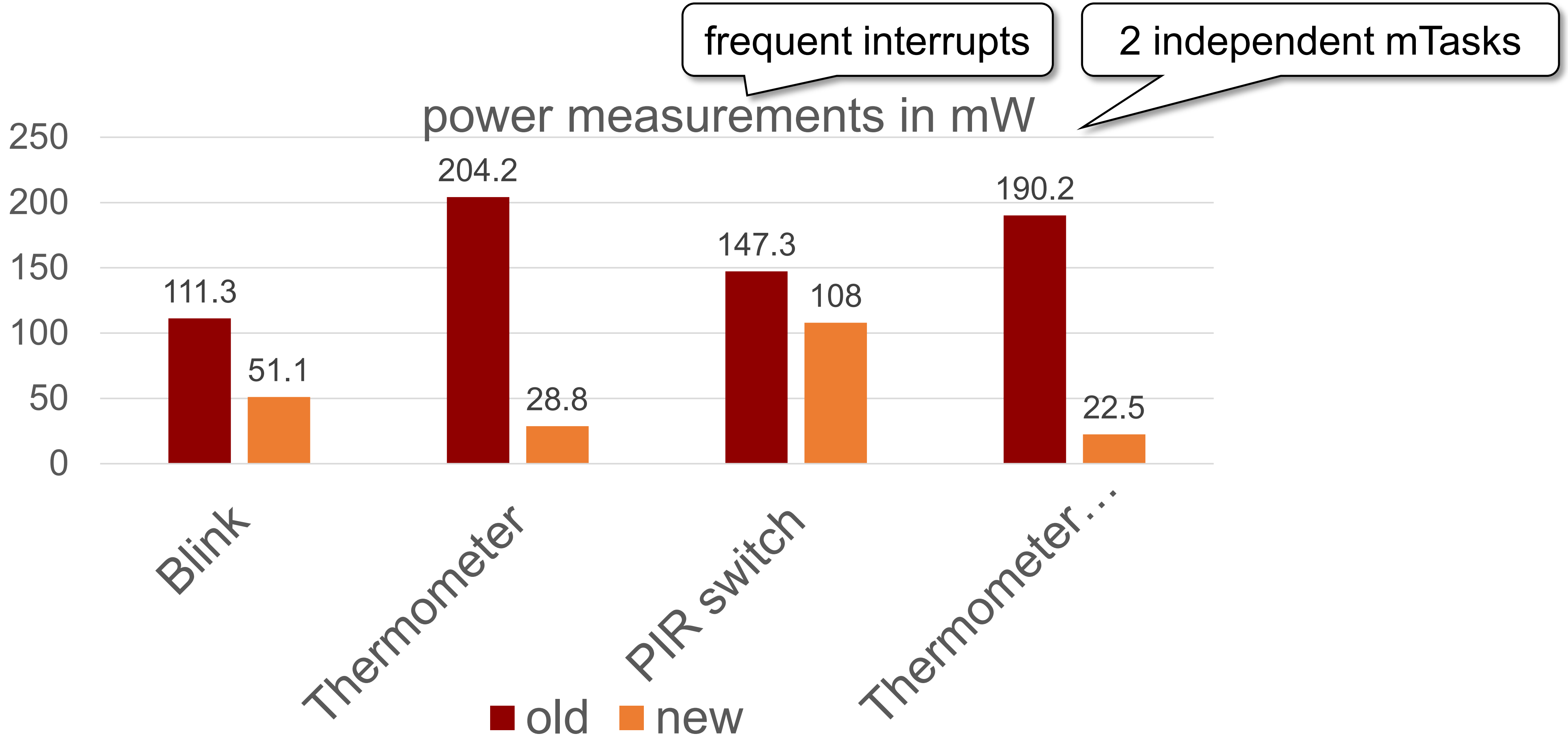
- Some sensors can wake up the microcontroller
  - restrictions apply to code executed after an interrupt
- Better than polling
  - less energy needed
  - fewer events missed
  - mTask has task-level support for interrupts

```
{ main = rpeat
  (    interrupt high pirpin
  >>|. writeD ledpin false
  >>|. delay interval
  >>|. writeD ledpin true) }
```





Automatic sleep of edge node



Sustainable IoT programming

- iTask+mTask makes IoT programming and maintenance easier
  - single concise source
  - all program fragments are automatically up to date
  - checked by strongly-typed compiler
  - 70-90% less code
- Restricted nodes make the IoT greener
  - mTask needed for single source programming
  - code for remote devices is dynamically generated
- Sustainable IoT programming
  1. single concise source suited for maintenance
  2. restricted hardware saves energy and natural resources
  3. feather-light OS for tasks offers automatic sleep to save energy

Interested in internship or master project on IoT?
Contact Mart Lubbers

Or in combination with sustainability?
Contact Bernard van Gastel

eDSL designs used in this course

	Deep ADT	Deep GADT	Shallow	Tagless
multiple views	++	++	-	**++**
optimization	++	+	- -	-/+
static type checking	+	++	++	++
overloading in DSL + static type checking	- -	++	++	++
checked DSL variables	--	-	++	**++**
extending the DSL	-	-	++	++
new datatypes in DSL	-	**+**	+	++
interpretation overhead	yes	yes	no	no

- Tagless: class based shallow embedding
  - multiple views + checked variables + no interpretation overhead

but the type errors can be horrible ☹

Conclusion

- DSLs are very useful
- Embedded DSL inherits much from their host language
  - this saves lots of implementation effort
- Various options to construct embedded DSLs
  - ADT
  - GADT (or faking it by bimaps)
  - shallow (= functions)
  - tagless (= type constructor classes)
- Each option has advantages and disadvantages