Type-Safe Functions and Tasks in a Shallow Embedded DSL for Microprocessors

Pieter Koopman and Rinus Plasmeijer

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

Draft Research Paper

Abstract. This paper introduces a type-safe way for defining user-defined functions and tasks in a shallow embedded DSL. The programs specified in a single source are partly executes as ordinary iTask program on a host while the DSL parts can be executed on tiny microprocessor based systems like an Arduino. The clear boundary between the DSL and the host language ensures that it is possible to generate compact code for small microprocessors. We can use the type system of the host language to impose the restrictions needed for transforming tasks and functions in the DSL to C++. Our DSL is based on type constructor classes to allow multiple views of the shallow embedded DSL. Having multiple views, like code generation and simulation, of a DSL is usually achieved by a deep embedding. Our approach enables the addition of DSL language constructs as well as new views of the DSL without changing existing code.

Keywords: Domain Specific Language, Shallow Embedding, User-defined Functions, Embedded Systems, Internet of Things

1 Introduction

In this paper we introduce a Domain Specific Language, DSL, with type safe user-defined functions and tasks. This DSL is intended to enable Task Oriented Programming, TOP [20], on small embedded systems like an Arduino [1, 5]. Such microprocessor based systems will be omnipresent in the Internet of Things, IoT. The 8-bit 16 MHz ATmega328P microprocessor of the Arduino Uno R3 is very suited for simple control tasks, but it is not particular speedy. It provide just 32Kb of program store and 2Kb of RAM. Other microprocessors have more memory and a faster processor, but it are all very modest systems compared with desktops and laptops. Due to the limited capabilities of such systems it is unfeasible to implement a complete higher order and lazy functional programming language on these embedded systems.

Despite the limited processing power of such microprocessor based systems, these devices are very useful for simple control tasks. Usually these systems have

input/output ports under direct control of the microprocessor. In this paper we introduce task based programming for such systems. Microprocessors cannot execute full blown iTask programs [19]. We envision a solution were parts of an iTask program are executed on microprocessors and the main part is running on an ordinary server. To enable single source iTask programs that are partly executed on microcomputers we introduce a DSL for parts to be be executed on microcomputers. Using a DSL has as additional advantages that we can piggyback on the parser and a type checker of the host language. We use Clean [21], the implementation language of the iTask system as host language. We call our DSL for TOP of microprocessors mTask.

To achieve TOP on a tiny microprocessor we have to impose severe restrictions on the DSL used to specify tasks compared to the iTask system. The available memory makes it impossible to use a standard heap. This implies that we can neither have lazy evaluation nor higher order functions. It is possible to transform higher order functions to first a first order language by defunctionalization, [9, 23], but that introduces new functions. Due to the very limited amount of flash memory to store programs, generating additional functions is undesirable. Also with respect to datatypes we must be very reticent. Any data structure that uses large or uncontrolled amounts of memory will cause problems.

We require that the type system of the host language checks the types in the DSL. This is achieved by a shallow embedded DSL: the DSL is a set of functions. The types of these functions ensure that the expressions in the DSL are correctly typed. To allow different interpretations (like pretty printing, code generation and simulation) of the DSL we use type classes of functions instead of plain functions. Each interpretation, *view*, is a new instance of the type classes.

To prevent that we silently inherit the complete host language in our DSl we need a clearly bounded DSL. This is achieved by using type constructor classes instead of plain type classes.

The class based DSL can be extended by new language constructs or new views without changing any existing code. The type system of the host language checks that the required constructs and views are defined for an application.

In the translation view we compile mTask programs to C++ programs for the Arduino dialect. This has as advantages that code generation is relative simple and applicable to all microcomputers supported by the Arduino platform. Unfortunately, simple compilation to C++ imposes some additional restrictions on mTask.

In this paper we first give a brief introduction to Arduino programming in Section 2. The next Section, 3, introduces our type safe extendable shallow embedded DSL. Section 4 shows how we can define directly C++ code in a tailor-made view of our DSL. Section 5 shows how we can simulate programs in our DSL using a view that transforms DSL programs to expressions in its host language. A simple iTask program is used to interactively show the state and execute tasks. Finally we discuss related work in Section 6, and discuss the obtained results in Section 7.

2 Arduino Programming

An Arduino is typically programmed in its own dialect of C++. The Arduino IDE supports programming. It can translate such a C++ program to machine code for the selected board. This code is uploaded to the microprocessor via an USB cable and a tiny boat loader on the board. Apart from the various Arduino variants, the IDE also supports many other microprocessors.

An Arduino has no operating system. Therefore every Arduino program in C++ contains two basic functions. Each of these functions can be empty. The function **void** `setup()` is called once at startup of the microprocessor. After this initialization the function **void** `loop ()` is repeated forever.

The "hello world" example for the Arduino blinks the on board LED connected to digital pin 13. The pins of the Arduino can be used as input and output. It is required to set the pin in the right input/output mode before using it in that mode. In this example `setup()` configures pin 13 as an output. The `loop` switches on the LED, waits a second, switches of the LED, and waits another second. Since there is no operating system, there are no other threads or programs on the Arduino. By introducing some state variables and looking repeatedly at the clock with `millis()`, returning the milliseconds since startup, the use `delay` can easily be circumvented.

```
#define DELAY 1000
boolean ledOn = false;                  // status of LED
long lastTime = 0;                      // last status switch

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  if (millis() - DELAY > lastTime) {  // time to change?
    ledOn = not ledOn;
    digitalWrite(LED_BUILTIN, ledOn );
    lastTime += DELAY;
  }
}
```

3 The mTask DSL

As outlined in the introduction we want to create a task oriented programming language to program microprocessor. In the long run it should interoperate smoothly with the iTask system, such that parts of an iTask application can run on a microprocessor.

Due to the small amount of memory on microprocessor (e.g., 2KB RAM on an Arduino uno), it is impossible to port the entire iTask system to microprocessors. A typical iTask program requires 100MB of heap space. To run a task based

program in the limited memory of a microprocessor we reject the heap. Hence, we cannot have lazy evaluation and standard higher order functions. Transforming higher order functions to first order functions by defunctionalization introduces too many new functions for the limited flash memory available. Hence we restrict our language to first order functions. We obtain a first order, strict language. This strictness matches very well with the imperative nature of controlling the input/output ports of the microprocessor.

The examples in Section 2 use variables to store state information. In our DSL this information is stored in a state that is passed around in a monadic style. Program specific fields of the state are defined by need.

Our DSL should be equipped with the possibility to defined recursive functions. These functions will be used to express repetition. The functions also enable abstraction and reuse of code.

The actions of the program can be grouped in tasks. These tasks are parameterized, just like functions, to tune their actions. Unlike functions, tasks are not executed immediately when they are encountered during program execution, but somewhere in the future. It is perfectly possible to schedule different tasks or multiple instances of the same task for future execution. Since we have no heap the current program cannot wait for the result of an invoked task. Task invocation immediately returns a value of type `Task`.

Instead of the functions `setup` and `loop`, our programs execute a single `main` expression. After executing the main expression, the system repeatedly takes a task from the pool of scheduled tasks and executes it. Execution a task can initiate any number of new tasks.

The Arduino examples above show that it is very common to wait for some time in programs interacting with their environment. To facilitate this, every task definition specifies a custom delay. The task will not be executed before the given time is passed. The actual task invocation can have a customized delay.

The DSL is defined by a set of type classes. The functions in the classes define the allowed constructs in the mTask language. There is one instance of these classes for each view of the DSL. In this paper we will use code generation and translation to the host language Clean as views. This can be extended with views like pretty printing and partial evaluation.

3.1 Expressions

The basic class of our DSL is `arith`. It contains a function `lit` to lift a constant from the host language to mTask, and some basic operations. Like any class in the DSL, `arith` is based on a type constructor type `v t p`, where `v` is the view, `t` is the type of the mTask construct, and `p` indicates whether the expression is an updatable position, an arbitrary expression or a statement.

```
class arith v where
  lit :: t → v t Expr | toCode t
  (+.) infixl 6 :: (v t p) (v t q) → v t Expr
          | type, + t & isExpr p & isExpr q
```

Apart from the addition, `+.`, there are similar operators for subtraction, multiplication and division. The names of these operators are different from the usual operator names in Clean since we need different types. We do not redefine the operators in Clean since they coexists in programs with embedded DSL components. By convention we add a dot to the operator name. The class restrictions `toCode t` and `type t` guarantee that we can only lift types that can become code to the DSL level, and use DSL type of arguments for the operators respectively. Currently there are instances of `toCode` for the basic types and `String`. There are instances of the basic types for `type`. The class restriction `+ t` guarantees that we can apply addition in the DSL simulator.

These class members yields an expression, `Expr`. The mTask language has three kinds of constructs: updatable state elements, expressions, and statements. These kinds are identified by single constructor types.

```
:: Update = Update
:: Expr   = Expr
:: Stmt   = Stmt
```

The difference between expressions and statements is guided by the generated code. The arguments of the binary operations should be compilable to expression instead of statements. This implies that only updatable elements and expressions are allowed. This is checked at compile time by the class `isExpr`. The function `isExpr` is never used, its only purpose is to guarantee the contents restriction.

```
class isExpr a :: a → Int
instance isExpr Update where isExpr _ = 0
instance isExpr Expr   where isExpr _ = 1
```

The class `boolExpr` contains the Boolean operators, the overloaded equality and comparison operators. The new class shows that the DSL is extendable. More operators can be added by need without changing existing code.

```
class boolExpr v where
  (&.) infixr 3 :: (v Bool p) (v Bool q) → v Bool Expr |isExpr p&isExpr q
  (==.) infix 4 :: (v a p) (v a q) → v Bool Expr
        | ==, type a & isExpr p & isExpr q
```

3.2 Conditionals

The class `If` contains only the conditional expression of the DSL. The `If` has a DSL expression of the `Bool` as argument and two DSL expressions of type `t`. Any expression/statement type for the last two arguments is allowed. The result is a DSL component of type `t`. Whether the conditional yields an expression or a statement is determined by the kind of the argument using a functional dependency. The `~s` indicates that this class variable is dependent on the other class variables. There is also an infix conditional that has only a then part. It is a statement with an void result, `()`.

```
class If v q r ~s where
  If :: (v Bool p) (v t q) (v t r) → v t s | isExpr p
class IF v where
  (?) infix 1 :: (v Bool p) (v t q) → v () Stmt | isExpr p
```

3.3 Variables

As outlined above it is necessary to have user-defined fields in the state passed around. For an imperative language adding a field to the state is just a variable definition. For small embedded systems defining variables in the DSL is quite convenient, but it is not required by our approach to construct DSLs at all. In mTask the primary purpose of variables is the communication between tasks. However, the fields introduced by `var` are plain variables that can be used in any way the user wants.

Variables in mTask can be introduced by the class `var`. The first argument is the initial value of the variable. This value determines the type of the variable. The second argument is a function that takes the variable in the current view as argument and yields an arbitrary construct in the DSL. The introduced variable can be updated as indicated by the kind `Update`. There is also a `con` to define constants. Its kind is `Expr` instead of `Update` to indicate that a constant cannot be updated.

```
class var v where
  var :: t ((v t Update)→(Main (v c s))) → (Main (v c s)) | type t
  con :: t ((v t Expr)  →(Main (v c s))) → (Main (v c s)) | type t
```

The type `Main` packs a value in a record. Its only purpose is to make the main expression of mTask programs recognizable by the type system. The use of a record provides the desirable curly braces. Variables are introduced by a function. The argument represents the variable in the DSL program. Its type `v t Update` ensures that it is used only in a well-typed manner. There is no other way to specify variables; hence all variables in the DSL are properly defined.

We can assign a new value to such a variable in the DSL by the class `assign`. The first argument is the variable that is required to be updatable by `Update`. It is required that the new value given by the second argument has the same type `t` as the variable.

```
class assign v where
  (=.) infixr 2 :: (v t Update) (v t p) → v t Expr | type t
```

A minimal example is: `e1 = var 6 x.{main = x =. x *. lit 7}`.

3.4 Monadic Bind

There is a monadic bind operator, $\gg\!=.$, in mTask. In the variant $:.$ the second argument does not need the result of the first argument, this is just the semicolon from imperative programming.

```
class bind v where
  (>>=.) infixr 0 :: (v t p) ((v t Expr)→(v u q))→v u Stmt|type t&type u
```

There is no explicit `return`, all results are implicit returns.

3.5 Shield Control

To lift the C++ classes controlling a shield to mTask we introduce a class containing the shield manipulations in the host language. As example we show how to control an LCD. The LCD object is created by `liquidCrystal`. This object is introduced very similar to variables. The manipulation functions take this object as first argument. Hence they always act on a properly defined LCD object.

```
class lcd v where
  begin :: (v LCD Expr) (v Int p) (v Int q) → v () Expr
  print :: (v LCD Expr) (v t p) → v Int Expr|stringQuotes t
  setCursor :: (v LCD Expr) (v Int p) (v Int q) → v () Expr
  liquidCrystal :: [DigitalPin] ((v LCD Expr)→Main (v b q))
                   → Main (v b q)
```

The program that prints a message on a 16×2 LCD looks like:

```
helloLCD =
  liquidCrystal [] λlcd.
  {main = begin lcd (lit 16) (lit 2) :. print lcd (lit "Hello world")}
```

The list of digital pins in the argument of the constructor specifies to which pins the LCD is connected. An empty list is automatically converted to the most common list of pins.

3.6 Functions

Function definitions are implemented similar to variables. The main difference is that the function introducing the function name needs two arguments: the function body and the main expression. These elements are grouped by the infix constructor `In` of type `In`. The function body needs the name of the function in recursive functions. The function body itself is a function in the host language taking the function arguments in the host language as arguments. The class for function definitions in mTask is:

```
class fun v t where
  fun :: ((t→v s Expr)→In (t→v s p) (Main (v u q)))- Main (v u q)|type s
:: In body main = In infix 0 body main
```

Note that this class has two class arguments; the familiar view `v` and the type of the function argument `t`. By making this a type a class argument we can control the function arguments allowed in mTask by creating only the desired instances. Moreover, the type of the argument of the mTask function is available in Clean. Below we will see that this type is required in the views. The famous function and a main expression applying it to 5 are defined as:

```
e2 = fun λfac.(λn.If (n ≤. One) One (n *. fac (n -. One))) In
     {main = fac (lit 5)}
One = lit 1
```

Currently there are instances of the class `fun` allowing basic types, i.e. instance of the class `type`, and tuples with zero, two, three and four of these types as argument. Since there is no instance of `type` for functions the type system of Clean prevents higher order functions in mTask programs.

3.7 Task Definitions

Task definitions are very similar to function definitions. The result of such a task is of type `Task` instead of an instance of `type`. Remember that the operational semantics of a task is different than of a function. Functions are evaluated immediately in our imperative DSL. Task are evaluated somewhere in the future. The additional integer argument in task definitions is the minimal delay in milliseconds between task invocation and actual task execution.

```
class mtask v a where
  task :: Int ((a→v MTask Expr)→In (a→v u p) (Main (v t q)))
          → Main (v t q) | type t & type u
```

The task blink switches the LED on digital pin 13 every 500 ms. The argument `b` of the task indicates the new state of the LED. The main expression makes pin `D13` an output and starts the blink task.

```
blink = task 500 λblink. (λb. digitalWrite D13 b :. blink (Not b)) In
        {main = pinmode D13 OUTPUT :. blink (lit True)}
```

This example illustrates that, in contrast to the Arduino `loop ()`, tasks are not repeated by default. When repetition is needed a task can call any number of tasks in its body. Here, the task `blink` is called with the inverted Boolean value. Tasks can have the same type of arguments as functions.

The type `MTask` is different from the ordinary type `Task a` in the iTask system. Since these types coexist in programs, like the simulator in Section 5, we use different names. In our DSL the type `MTask` just carries the information needed by the simulator described in Section 5. This happens to be just the index in the list of task invocations. Hence, we define:

```
:: MTask = MTask Int
```

The clock example shows that it is perfectly possible to have multiple instance of the same task. Here the same task is used to update hours, minutes and seconds. The different delays and position are arguments of the task.

```
clock =
  liquidCrystal [] λlcd.
  fun λprintWithZero. (λn. n <. lit 10 ? print lcd Zero :. print lcd n) In
  task 0 λtick.
    (λ(n, max, pos, delay).setCursor lcd pos Zero :. printWithZero n :.
      setDelay (delay *. long (lit 1000))
```

```
      (tick (If (n =. max -. One) Zero (n +. One), max, pos, delay))) In
{main = begin lcd (lit 16) (lit 2) :. print lcd (lit "00:00:00") :.
        tick (lit 0, lit 24, lit 0, lit (60*60)) :.
        tick (lit 0, lit 60, lit 3, lit 60) :.
        tick (lit 0, lit 60, lit 6, lit 1) }
```

This concludes our description of the language mTask. There are few other classes in our DSL containing additional operators and convenience definitions. Due to the architecture that builds the language as a set of classes, it is easy to add classes without influencing existing code.

4 The Code Generation View of mTask

The most important view of mTask generates code of DSL programs that can be executed on a microprocessor. In this paper we will generate C++ code from the mTask constructs directly. Functions and expressions in our DSL will be mapped rather directly to corresponding C++ constructs. This is a rather extraordinary implementation route for functional programming languages. In general the need for heap manipulations, especially garbage collection, make a direct translation of a functional language to C++ impractical. In mTask we carefully avoided a heap due to memory size restrictions. The distinction between `Expr` and `Stmt` ensures that all mTask expressions of kind `Expr` can be translated to C++ expressions (without the need to introduce additional helper functions). Since there are only first-order functions with a proper name on the outermost level, the functions of our DSL can be directly mapped to functions in C++.

This architecture comes with some clear advantages. **1)** It is easier to generate C++ code than low-level machine code for a microprocessor. We profit from all optimizations of the existing compiler from C++ to microprocessor code. **2)** The code generation is to a large extent microprocessor independent. The Arduino system contains code generators for various microprocessors. **3)** It is obvious how the existing ecosystem of shields and associated C++ libraries can be used in mTask. We just have to extend our DSL with a one-to-one mapping of the relevant methods of the C++ class. The LCD library in Section 3.5 illustrates this. **4)** C gives enough low-level control to implement additional optimization in the future.

4.1 Data Type for Compilation

Any view `v` of our DSL requires two arguments: the type `t` and the kind `k`. The data type `Code` uses neither of these arguments. This type contains just a function changing the state, `CODE`, of the code generation. In this state we store the actual code in four different flavors: functions, variables, code for the `setup` function, and code for the `loop` function. In addition it stores book-keeping fields for indents, the kind of code currently generated, and whether we need to decorate the code with a `return` or a semicolon.

```
:: Code t k = C (CODE → CODE)
:: CODE =
  { fresh     :: Int      // to generate id's
  , freshTask :: Int      // to generate task id's
  , funs      :: [String] // code for functions
  , ifuns     :: Int      // indentation for functions
  , vars      :: [String] // code for variables
  , ivars     :: Int      // indentations for variables
  , setup     :: [String] // code for setup() function
  , isetup    :: Int      // indentation in setup()
  , loop      :: [String] // code for loop() function
  , iloop     :: Int      // indentation foor loop()
  , includes  :: [String] // names of included libraries
  , def       :: Def      // definition switch: where the code goes
  , mode      :: Mode     // decoration mode of translation
  }

:: Def  = Var | Fun | Setup | Loop
:: Mode = NoReturn | Return String | SubExp | Assign String
```

There is a set of manipulation functions. For instance the function `c` adds code to the current definition, and the operator `+.+` composes two code generation functions.

```
c :: a → Code b p | toCode a
c a =  C λc.case c.def of
    Fun   = {c & funs  = [toCode a: c.funs]}
    Var   = {c & vars  = [toCode a: c.vars]}
    Setup = {c & setup = [toCode a: c.setup]}
    Loop  = {c & loop  = [toCode a: c.loop]}

(+.+) infixl 5 :: (Code a p) (Code b q) → Code c r
(+.+) (C f) (C g) = C (g o f)
```

The mode of the code-state controls the generation of embedding C++ keywords like `return` and the semicolon. Subexpressions do not need any embedding. As everywhere else the definition switch controls to what part of the code the output goes.

```
embed :: (Code a p) → Code a p
embed e =
  getMode λm. case m of
    NoReturn = setMode SubExp +.+ e +.+ c ";"
    Return t = c "return " +.+ setMode SubExp +.+ e +.+ c ";"
    Assign s = c (s+" = ") +.+ setMode SubExp +.+ e +.+ c ";"
    SubExp   = e
```

The actual compilation applies the code generation function to the initial `CODE` state and composes all code strings in the right order. The simplified version without predefined variables and the `loop` function reads:

```
compile :: (Main (Code a p)) → [String]
```

```
compile {main=(C f)} =
  reverse c.vars ++ reverse c.funs ++
  ["void setup () {\n","  Serial.begin(9600);\n":reverse c.setup]++["\n}\n"]
where c = f newCode
```

4.2 Code Generation for Expressions

These preparations enable code generation for expressions as:

```
instance arith Code where
  lit a = embed (c a)
  (+.) x y = codeOp2 x " + " y

codeOp2 :: (Code a p) String (Code b q) → Code c r
codeOp2 x n y = embed (brac (x +.+ c n +.+ y))

brac :: (Code a p) → Code b q
brac e = c "(" +.+ e +.+ c ")"
```

4.3 Code Generation for Conditionals

Code generation for conditionals distinguishes two situations. We generate a conditional statement when one or more of the branches is a statement. Otherwise we generate the C++ conditional expression `c ? t : e`. The functional dependency in the class definition ensures that the entire conditional construct is a Stmt when one of the branches is a statement.

```
instance If Code Stmt Stmt Stmt where If c t e = IfStm c t e
instance If Code e    Stmt Stmt where If c t e = IfStm c t e
instance If Code Stmt e    Stmt where If c t e = IfStm c t e
instance If Code x    y    Expr where If c t e = IfExp c t e
IfExp b t e =
  embed (brac (b +.+ indent +.+ nl +.+ c " ? " +.+ t +.+ nl
               +.+ c " : " +.+ e +.+ unindent))
```

4.4 Code Generation for Variable Definitions

For variable and constant definitions we generate exactly the same code. The difference between those definitions in mTask is made due to the kind argument by the type system of the host language. It is possible to add keyword `const` to the generate C++ code of a constant definition, but this is not required. The actual work is done by `defCode`. It generates a fresh variable name, `name`. The actual definition of a global variable is added to the variable definition part of the code. In the value `embed name` is used as actual value for applied occurrences of this variable in this view. This is achieved by supplying this value to the function `f` that produces the main expression. The code for main is directed towards the body of the `setup` function.

```
instance var Code where
  var v f = defCode v f
  con v f = defCode v f

defCode :: t ((Code t p)→Main (Code u q)) → Main (Code u r) | type t
defCode v f =
  {main = fresh λn. let name = c ("v" + toCode n) in
   setCode Var +.+ c (type2string v + " ") +.+ name +.+  c (" = " +
   toCode v + ";\n") +.+ setCode Setup +.+ unMain (f (embed name))}
```

Code generation for an assignment generates the code for the variable, an equals sign and the code for the expression.

```
instance assign Code where
  (=.) v e = embed (setMode SubExp +.+ v +.+ c " = " +.+ e)
```

4.5 Code Generation for the Monadic Bind

For a monadic bind we define a variable to store the the result of the of the first argument. We use the name of this variable as argument to the function on the right-hand side. By design, the kind `Expr` instead of `Update`, it is not possible to update this variable. We have to juggle a little with modes to ensure that a potential return goes to the second statement and the first statement gets just a semicolon.

```
instance bind Code where
  (>>=.) x f =
    getMode λmode. fresh λn. let v = c ("b" + toCode n) in
    varType x +.+ c " " +.+ setMode NoReturn +.+ v +.+ c " = " +.+ x +.+
    nl +.+ setMode mode +.+ f (embed v)
```

4.6 Code Generation for Function Definitions

Our DSL is designed such that functions in mTask can be mapped directly to functions in C++. We designed the class `fun` such that we have to make an instance for every number of arguments allowed. Currently we allow zero to four arguments. As example we show the code generation for a two-argument function. Based on a fresh number we generate names for the function, `fname`, and its arguments, `aname` and `bname`. The class `argTypes` yields the types of the arguments needed in the C++ function definition. Applied occurrences of the function take a Clean tuple of mTask expressions as argument. The nameless function we supply as argument to `f` converts this to code for C++ arguments. In the function section of the code we generate the C++ version of our DSL function. In the obtained function body `g` we use the generated formal arguments as actual arguments.

```
instance fun Code (Code a p, Code b q) | type a & type b where
 fun f =
```

```
{main = getMode λmode. fresh λn.
let fname = c ("f" + toCode n + " ")
    aname = c ("a" + toCode n + " ")
    bname = c ("b" + toCode n + " ")
    (atype, btype) = argTypes f
    (g In main) = f (λ(x,y).embed (fname +.+ codeOp2 x ", " y)) in
 setCode Fun +.+ resType f +.+ fname +.+ codeOp2 (atype +.+ aname) ","
 (btype +.+ bname) +.+ funBody (setMode (Return (toCode (resType2 f))) +.+
 g (aname, bname)) +.+ setCode Setup +.+ setMode mode +.+ unMain main}
```

4.7 Code Generation for Task Definitions

Although the definition of tasks is very similar to functions, the operational behavior is quite different. Functions are evaluated strictly by compiling them directly to C++ functions. Tasks are scheduled for execution somewhere in the future.

Executing task in the future is implemented by a small array, `tasks`, of task activation records of C++ type `Task`. Each record contains the task-id (a small number that fits in a byte), a waiting time in milliseconds, and a array containing the arguments.

```
typedef union Arg {int i; bool b; char c; word w;} ARG;
typedef struct Task {byte id; long wait; ARG a[M_ARG];}TASK;
TASK tasks[MAX_TASKS];
```

The generated `loop()` function scans this array of tasks. When the next task in the array needs to wait it is copied to the end of the sequence of waiting tasks in a round robin arrangement. When the waiting time has expired the task is executed. The generated `loop` function contains a case construct to select the proper task body. The task-id is used to find the appropriated code fragment.

The compilation scheme for `mtask` is very similar to the plan for `fun`. Since we have also a `mtasks` class we distribute the code generation slightly different to reuse more code.

```
instance mtask Code a | taskImp a & types a where
  task i f =
   {main = freshTask λn.let (app, a) = taskImp n types;(b In main) = f (app i)
   in codeTaskBody (loopCode n (b a)) (unMain main)}
```

The class `taskImp` generates the task application and the formal arguments of the definition. A task application boils down to a call of `newTask` in the generated code with appropriate task-id, waiting time, and arguments. The `loopCode` generates the appropriate case in the switch statement for this task. It just puts the code generated for the task body between a **case** and a **break**.

```
loopCode :: Int (Code a b) → Code c d
loopCode n b =
  c "case " +.+ c n +.+ c ": {" +.+ indent +.+ nl +.+ setMode NoReturn
  +.+ b +.+ nl +.+ c "break;" +.+ unindent +.+ nl +.+ c "} "
```

5 The Simulation View

The next view translates mTask programs to plain Clean programs. This is very useful in a simulation of mTask programs. Since it is hard to debug programs running on an Arduino, simulation is an important tool to spot execution problems. In this section we outline how to translate our DSL to the host language and indicate how this can be simulated with an iTask program.

5.1 Data Type for Simulation

Similar to the code view, the evaluation view is a state transformer. The state in this view contains the relevant parts of the Arduino and the task queue. As Arduino parts we have the declared variables, an abstraction of the IO-pins, the timer and the serial output.

```
:: State
  { mtasks :: [(Int, State→State)]
  , store  :: [Dyn]
  , dpins  :: [(DigitalPin, Bool)]
  , apins  :: [(AnalogPin, Int)]
  , serial :: [String]
  , millis :: Int
  }
```

The state transition takes a read/write value of type `RW t` as argument. This value determines whether a variable occurs in a read, `R`, or write context, `W t`. The write context occurs on the left-hand side of assignments, everywhere else we have the read context. The function `F` value is used to update objects in the store.

```
:: Eval t p = E ((RW t) State → (t, State))
:: RW   t   = R | W t | F (t→t)
```

We define a tailor made monadic bind, `>>==`, and return, `rtrn`, for `Eval`.

```
(>>==) infixl 1 :: (Eval a p) (a→Eval b q) → Eval b r
(>>==) (E f) g = Eλr s.let (a,t) = f R s in unEval (g a) R t

rtrn :: a → Eval a q
rtrn a = E λr s → (a, s)

yield :: t (Eval s p) → Eval t Expr // effect of rtrn in Expr
yield a (E f) = E λr s.(a, snd (f R s))
```

5.2 Evaluating Expressions

The evaluation view of expression computes their value in the context of the monad introduced above.

```
instance arith Eval where
  lit a = rtrn a
  (+.) x y = x >>== λa. y >>== λb. rtrn (a + b)
```

5.3 Evaluation of Conditionals

Evaluation of conditionals is the usual evaluation of the conditional expression in the monad followed by the appropriate action. The `toExpr` in the `If` is necessary to ensure that the kind of the result is an expression. The `return ()` in the other conditionals produces the desired void result.

```
instance If Eval p q Expr where
  If  c t e = c >>== λb.if b (toExpr t) (toExpr e)

toExpr :: (Eval t p) → (Eval t Expr)
toExpr (E f) = E f
```

5.4 Evaluation of Variable Definitions

A variable becomes an element in the list store. Since we have different type of variable (e.g. `Int` and `Bool`), we store a dynamic representation of these values, a list of strings, instead of the values themselves. The element number in this list becomes the variable identifier.

```
instance var Eval where
  var v f = defEval v f
  con v f = defEval v f

defEval v f =
  {main = E (λr s.(length s.store, {s & store = s.store ++ [toDyn v]}))
  >>== λn.unMain (f (E (read n)))}
```

The function `read` selects the appropriate element form the `store`. The read/write context, of type `RW t`, indicates the desired action with this variable: `R` read, `W a` write value `a`, and `F f` apply function `f` to update the object

```
read :: Int (RW a) State → (a,State) | dyn a
read n R     s = (fromJust (fromDyn (s.store !! n)), s)
read n (W a) s = (a,{s&store=updateAt n (toDyn a) s.store})
read n (F f) s=:{store}
    # obj = f (fromJust (fromDyn (store !! n)))
    = (obj, {s & store = updateAt n (toDyn obj) store})
```

As outlined above, the read write context of variables is always read except in an assignment. Here we write the new value `a` with `W a`.

```
instance assign Eval where (=.) (E v) e = e >>== λa. E λr s.v (W a) s
```

5.5 Evaluation of the Monadic Bind

The bind operators `>>=.` and `:.` are directly translated to Clean.

```
instance bind Eval where
  (>>=.) x f = x >>== f o rtrn
  (:.)   x y = x >>== λ_. y
```

5.6 Evaluation of Functions Definitions

Also functions are directly converted to functions in Clean:

```
instance fun Eval x|arg x where fun f = e where (g In e) = f (λa.toExpr (g a))
```

5.7 Evaluation of Task Definitions

Task definitions are slightly more complicated since every invocation is stored as `State→State` function with it delay d in a separate field in the state. The function `toS2S` takes transforms an eval function to a plain state transformation.

```
instance mtask Eval x | arg x where task d f = e where
  (t In e) = f (λa.Eλr s.(MTask (length s.mtasks)
                ,{s & mtasks = s.mtasks ++ [(i, toS2S (t a))]}))
```

5.8 An mTask Simulator

The translation of a program in mTask by this view yields a state transformation function in Clean. By construction the tasks wait in the state to get fired. The function `step` collects the task from the state increments the time by the smallest delay of these tasks and apply the tasks in their creation order. In general this will create new tasks. This `step` function has the same effect as evaluating all currently available tasks in the generated `loop` of the compilation.

```
step :: State → State
step s =
  foldr appTask {s & millis = s.millis + delta, mtasks = []}
        [(w - delta, f) \\ (w, f) ← s.mtasks]
where delta = foldl1 min (map fst s.mtasks) // smallest wait
```

A simple iTask program can be used as an interactive simulator for mTask programs. The user can inspect and change the state between the task applications. By a push of the `loop` button we apply the `step` function, this executes all currently available tasks. The iTask defining the simulator is:

```
simulate :: (Main (Eval a p)) → Task ()
simulate {main=(E f)} = setup state0 where
  setup s =
    updateInformation "State" [] (toView s)
    >>* [ OnAction ActionFinish (always shutDown)
        , OnAction setUp (hasValue (λsi.simloop (snd (f R (mergeView s si)))))
        ]
  simloop s =
    updateInformation "State" [] (toView s)
    >>* [ OnAction ActionFinish (always shutDown)
        , OnAction ActionNew (always (setup state0))
        , OnAction (Action "loop" []) (hasValue λsi.simloop (step si))
        ]
```

A screenshot of this simulator is depicted in Figure 1.

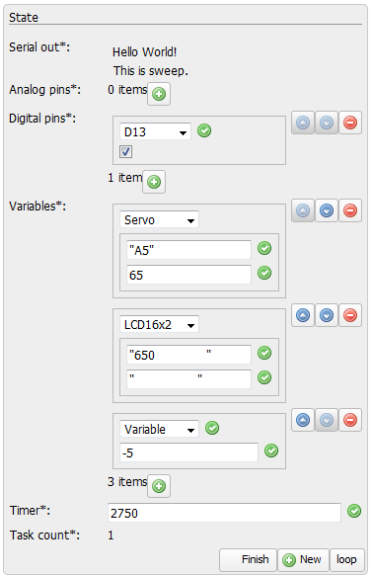

Fig. 1. Screenshot of the simulation.

6 Related Work

There are several groups of related work. We discuss approaches to control microprocessors with high-level languages, the generation of C code to implement function languages directly, and related representations of DSLs.

6.1 High-Level Languages for Microprocessors

There are many microprocessors with various capabilities. Many languages are ported in one way or another to some microprocessor. The Clean compiler is currently ported to the ARM-processor driving the Raspberry Pi. With the announcement of the Pi Zero [22] as the $5 computer this can become a serious alternative for many microprocessor applications, especially in the IoT.

The package hArduino allows Haskell programs to control Arduino boards and peripherals, using the Firmata [11] protocol. The Haskell program is not running on the Arduino itself.

Lua [12, 13] 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 is memory size and clock-speed.

The Espruino project projects provides a JavaScript interpreter on single chips microprocessor boards [25]. 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.

The SAPL approach compiles functional programming languages to small executables, typically executing in a browser [15, 14]. It is interesting to investigate whether it is possible to generate C code with a very small footprint to allow execution on a microprocessor in this way.

6.2 Generating C-Code

Generating C code from functional languages is quite common, e.g. [10, 24, 17]. Until version 6 GHC compiled Haskell to C code [18]. These implementation use C has a high-level assembly. Functions in the source language are not directly mapped to functions in C.

Filet-o-fish is tool to build DSLs to write operating systems [7]. Like mTask the DSL generates C/C++ code. The abstraction level of the DSLs constructed is typically lower as in our mTask system. The filet-o-fish approach uses standalone compiler instead of an embedded DSL.

6.3 Shallow Embedding of DSLs with Multiple Views

Carette et al. use a class based approach to construct a DSL with multiple views like we do [6]. One of the views is partial evaluation. Their language is basically λ-calculus. Their work is missing the fancy type system used here as well as C code generation.

6.4 Future Work

Although we are now able to execute task-oriented programs on a tiny microprocessor system like the Arduino, this work is not finished. The first thing to be done is making a connection between the mTask and the iTask system. The goal is that an iTask program can specify subtasks in mTask and delegate them to a microprocessor. The iTask system should be able to monitor and influence task execution on in the mTask system, similar to its own tasks.

The current mTask system is able to interact with other systems over the serial port by sending and receiving messages. To make it a better IoT language we will add communication over WiFi as well as Bluetooth.

The mTask system itself should be completed with data types. It is desirable to add at least strings and arrays. Currently the mTask system is lacking primitives for task management. Any task coordination must be implemented using state variables. It is desirable to introduce constructors to implement frequently occurring communication patterns between tasks.

7 Discussion

The goal of this work was to construct a system for task oriented programming on tiny microprocessors like the Arduino. We built a shallow embedded domain specific language based on type classes.

The advantage of shallow embedding is that a plain Hindley-Milner type system ensures well-typed DSL programs. Moreover, a shallow embedded DSL is easily extendable by adding new functions in the host language as elements of the DSL. This paper shows how one can introduce type-safe and well-defined variables, functions, and tasks in such an DSL by nameless functions in the host language. Guaranteeing well-defined identifiers at compile time is usually problematic in a DSL. This paper provides a simple and elegant solution.

Since we defined the DSL as a set of classes instead of a set of plain functions, it is easy to introduce new interpretations, called views, of the DSL. In this paper we showed how to compile our DSL to compact C++ code for the Arduino ecosphere. This makes the generated C++ programs portable to a family of microprocessors. By targeting on the smallest member (the Uno) of the family, the generated programs can be ported easily to similar microprocessors.

The second view translates programs from the DSL with state variables to to a monadic expression in the pure host language. A concise iTask program simulates the DSL programs interactively. The user can inspect and change the state of the DSL program between the execution of DSL tasks.

Finally we have briefly shown how to optimize DSL programs. The partial evaluation view optimizes programs within the DSL. The obtained program can be used in any of the views available. For view specific transformations a view of the DSL that yields an abstract syntax tree it is more appropriate. This combines best of both worlds; we have an extendable and type-safe DSL in the primary shallow embedding, while the generated data structure in the deep embedding is convenient for analysis and transformation.

Our example programs show that task oriented programming in the introduced mTask system is very suitable for programming microprocessor systems. Using parameterized tail-calls in the tasks the use of state variables can often be circumvented. This results in concise, well-typed, elegant, and portable high-level programs for microprocessor systems. The advantage of our DSL over C++ for microprocessor programming is that it provides high level task oriented programming.

References

1. Arduino.cc: (2015), https://www.arduino.cc
2. Arduino.cc: Arduino liquidcrystal library. (2015), https://www.arduino.cc/en/Reference/LiquidCrystal
3. Arduino.cc: Arduino newping library. (2015), http://playground.arduino.cc/Code/NewPing
4. Arduino.cc: Arduino servo library, (2015), https://www.arduino.cc/en/Reference/Servo
5. Arduino.org: (2015), http://www.arduino.org/
6. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19(5), 509–543 (Sep 2009), http://dx.doi.org/10.1017/S0956796809007205
7. Dagand, P.E., Baumann, A., Roscoe, T.: Filet-o-fish: Practical and dependable domain-specific languages for os development. SIGOPS Oper. Syst. Rev. 43(4), 35–39 (Jan 2010), http://doi.acm.org/10.1145/1713254.1713263

8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 207–212. POPL '82, ACM, New York, NY, USA (1982), http://doi.acm.org/10.1145/582153.582176
9. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 162–174. PPDP '01, ACM, New York, NY, USA (2001), http://doi.acm.org/10.1145/773184.773202
10. Feeley, M., Miller, J.S., Rozas, G.J., Wilson, J.A.: Compiling higher-order languages into fully tail-recursive portable c. Tech. rep. (1997)
11. Hoefs, J.: Firmata protocol (2014), http://firmata.org/wiki/Main_Page
12. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua – an extensible extension language. Softw. Pract. Exper. 26(6), 635–652 (Jun 1996)
13. Ierusalimschy, R., Figueiredo, L.H.d., Celes, W.: Lua 5.1 Reference Manual. Lua.Org (2006)
14. Jansen, J.M.: Programming in the λ-calculus: From Church to Scott and back. In: Achten, P., Koopman, P. (eds.) The Beauty of Functional Code, LNCS, vol. 8106, pp. 168–180. Springer (2013)
15. Jansen, J., Koopman, P., Plasmeijer, R.: From interpretation to compilation. In: Horváth, Z. (ed.) Proceedings of the 2nd CEFP'07. LNCS, vol. 5161, pp. 286–301. Springer, Cluj Napoca, Romania (2008)
16. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
17. Kameyama, Y., Kiselyov, O., Shan, C.c.: Combinators for impure yet hygienic code generation. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. pp. 3–14. PEPM '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2543728.2543740
18. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
19. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the ICFP'07. pp. 141–152. ACM, Freiburg, Germany (2007)
20. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. pp. 195–206. PPDP '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2370776.2370801
21. Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean language report (version 2.2) (2011), http://clean.cs.ru.nl/Documentation
22. Raspberrypi.org: Pi zero description. (2015), https://www.raspberrypi.org/products/pi-zero/
23. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference - Volume 2. pp. 717–740. ACM '72, ACM, New York, NY, USA (1972), http://doi.acm.org/10.1145/800194.805852
24. Sumii, E.: Mincaml: A simple and efficient compiler for a minimal functional language. In: Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education. pp. 27–38. FDPE '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1085114.1085122
25. Williams, G.: The espruino project (2015), http://www.espruino.com/