

Final assignment - Advanced programming

Lucas Franceschino

15/01/2017

Contents

1	Language	2
1.1	Internal representation	2
1.1.1	Task construction	2
1.1.2	If construction	3
2	Anatomy of a Task	3
3	Combinators	3
4	Simulation with iTask	3
4.1	GUI	3
4.2	Eval of Expr	4
4.2.1	Calling task and affecting inputs	4
5	Translation of a Task to ardunio-side	4
5.1	Needs	4
5.2	Tree of calls and callbacks	5
5.3	Task representation	5
5.4	Data storage	5
6	Implemented programs	5
6.1	Timer	6
6.2	Team counter	6
7	Current state	6
7.1	C++	6
7.1.1	How to test generated C++ code	7
7.2	iTask simulation	7
7.3	Lack in my task design	7
8	Notes	7

1 Language

The language is implemented in the module `Expr`.

1.1 Internal representation

The DSL is defined in the same module `Expr`, and uses GADT. The ADT `Expr` represents the language. Here is a sum up of what is used in `Expr` (this table might be incomplete, see `expr.dcl` for more precise information):

Expression	Description
Integer Int	lift an Int
Boolean Bool	lift a Bool
Str String	lift a String
Float Real	lift a Real
VoidExpr	lift a Void
Mul a a	the arithmetical operation \times
Div a a	the arithmetical operation $/$
Plus a a	the arithmetical operation $+$
Minus a a	the arithmetical operation $-$
Le Bool Bool	the boolean operation $<$
Eq Bool Bool	the boolean operation $==$
Not Bool Bool	the boolean operation <i>not</i>
Or Bool Bool	the boolean operation <i>or</i>
And Bool Bool	the boolean operation <i>and</i>
Task name params body	create a task, name is a unique string
RunTasksKeepLeft ...	combinator: take two tasks and inputs
RunTasksKeepBoth ...	combinator: take two tasks and inputs
RunTasksKeepAny ...	combinator: take two tasks and inputs
RunTasksSequence ...	combinator: take two tasks and inputs
RunTask task inputs	call a given task
TupleN tuple	$N \in [[2, 5]]$, lift tuples of dimension N
IfExpr* Bool A B	represent if cond A B
IfRetV*	
Variable String	fetch content of variable by name
GetTimestamp	return current timestamp
Clear	empty the screen
Print	print text
WaitForButton	wait for a button to be in a specific state
GetButtonPressed	
ToString	
...	<i>See <code>Expr.dcl</code></i>

1.1.1 Task construction

A task is represented by the type `:: ATask tIn tOut = ATask tOut`. For example, a task taking integer and returning a tuple of strings would be `ATask Int (String, String)`.

To construct a task, you need to provide a body. This body has to be of type `ReturnedValue tOut`. A simple inline expression operation, like `12 * 3` returns some data. But (mainly in the C++ part) `RunTask` take the current context of a task and replaces it by the execution of another task: it was very

important to forbid any "inline" operation like `*`. If you want to write `(RunTask A) + (RunTask B)`, you need to combine `A` and `B` (using `A -&&- B`) and then sequence the result (using `>>=`) to another task taking a tuple of inputs `(v,w)` and outputting `v+w`.

When we want to return a value directly, without any `RunTask`, we need to use the operator `:=`. For example: `task "id" [] := (integer 2)`

1.1.2 If construction

`if` is a keyword we want to apply both on `Expr a` and on `ReturnedValue (Expr a)`. Indeed, if we want to branch two different tasks with different parameters, then we need to write `if condition (CallTask A ...) (CallTask B ...)`.

That's why `if` is a class with instances on `Expr a` and `ReturnedValue (Expr a)`.

2 Anatomy of a Task

A `Task` has a specific signature: an input type `in` and an output type `out`. Also, a `Task` has a body. The body (of type `out`) is represented internally as a `ReturnedValue out`. A `Task` combinator is just a normal `Task`.

A task has an unique `id`, its name, a string. That can be an issue, see part 7.3

Internally, a task is represented with `ATask inputType outputType`.

3 Combinators

We can combine tasks using 5 different operators:

- wait for any: `[(ATask in out) (ATask in out) -> (ATask in out)]`
- wait for left: `[(ATask in out) (ATask in _) -> (ATask in out)]`
- wait for both: `[(ATask in outL) (ATask in outR) -> (ATask in (outL, outR))]`
- sequence: `[(ATask in m) (ATask m out) -> (ATask in out)]`
- sequence redirect: `[(ATask inA outA) (ATask inB outB) (ATask (y, outA) inB) -> ...]`. This combinator is special: it takes two tasks, an input for the first task, a value of any type, and a task combining these two last values into a last one of type `inB`. This combinator allows to make data "jump" over a task: we can execute `A` then `B` with independant data connected by a task `C`.

4 Simulation with iTask

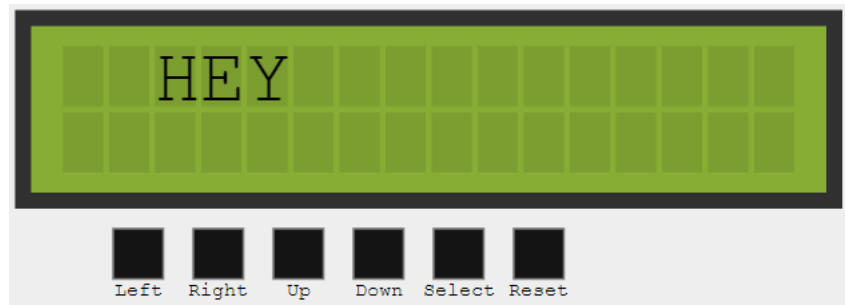
4.1 GUI

In the module `guiArduinoSimulator`, I defined some functions to design a SVG interface with `iTask`. It displays a `LcdDescriptor`, which consists of two fields that represent lines (two strings) to display, and a field representing the current pressed button.

When we want to print something on the screen, at some specific positions, we just replace the corresponding sub-string of one of the two lines in the `LcdDescriptor`.

Each button can handle a click and changes the current pressed button. A button is unpressed after being caught. (this behaviour results in a small bug: if the program does not listen to a specific button, that same button will not be released).

In order to make the visual simulation nicer, I used the same colors as usual LCD screens, here is the result:



4.2 Eval of Expr

4.2.1 Calling task and affecting inputs

When a task is called, the evaluator binds input values to parameters. We have a list of names for the input arguments of the task, and a data of a specific type for the input. For example, if the task takes two integers, then the input type will be `(Int, Int)`.

Using dynamic type checking and dynamic pattern match, I map the first level tuples to the different input variables names. If the input is not a tuple, then it is considered as a singleton. That might be a not wise choice, indeed, we can't pass one two-values tuple as a tuple to a task: it will always be passed as two arguments. But if we nest tuples, then, nested tuples will be mapped directly.

The good point with this approach is that it provides a natural way of extracting tuples.

5 Translation of a Task to arduino-side

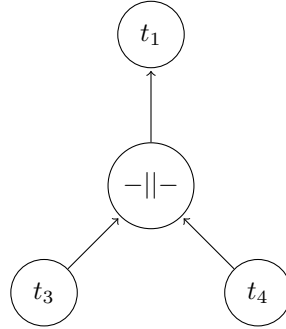
To be able to re-write task oriented programs in C++ style, I decided to write a little runtime to make the arduino able to handle task structures.

5.1 Needs

A task is a bunch of code taking some inputs and outputing some formatted data. The type of the inputs and outputs are known and not dynamic. Hence, we need a way to store any kind of data.

Since a task combinator is an usual task, we need to be able to call tasks in a task. Task can return data, and we can sequence tasks. Then, data need to be able to flow from a task to another one.

Callbacks To make all of that working, we need to represent all tasks as a sort of tree of dependences. For example, if we have the following tasks:



With t_3 and t_4 returning some integer values, t_2 being a combinator waiting for any value, and t_1 a `ATask Int Void` (i.e. a task displaying on screen its input).

We need a structure to store returned values, and to redirect these values.

5.2 Tree of calls and callbacks

The vector `queueTaskCall` handles calls of tasks. Each entry can either:

- Call a task with some data as input: this is used with sequence combinator.
- Transfer some data to the parent task: this is used with parallel combinators.

5.3 Task representation

On the arduino side, the body of each task is written directly in the `loop` function. These bunches of code are executed only when the active (executed) task is the current task body.

A task has a callback identity, which consist of an `id` and of a boolean, saying if the returning data is `left` or `right`. For example, if we execute `x -&&- y`, x and y being tasks, when x finishes returning some data, x will indicate its callback identity: an `id` and `left` (because x is on the left of the combinator `-&&-`). Then, the callback specified by the `id` will have a specific behaviour:

- wait for any: as soon as the callback is called, we return the data to the parent callback
- wait for left: as soon as the callback is called with a left value, we call the parent callback
- wait for both: as soon as the callback is called twice, we call the parent callback
- sequence: as soon as the callback is called, we call a specified task

5.4 Data storage

When a task calls another one, the call is added to a stack. A call consists of a structure with field of type `void*`. That's the way I store any data.

When a task, in its body, wants to recover a variable v , it calls `getArg_[int|string|float|...](v)`. Actually, v is not a string, but a number: when the C++ code is written, we read the name of the variable and we recover the index in the task call. So, the task recovers the v th parameter.

6 Implemented programs

I implemented the two asked programs: a timer and a two-teams counter. These ones are available as iTask simulation (you can select the one you want to test by selecting it on the main screen).

The generated C++ code is not working due to some last minutes problems handling tuples in C++. The system of combinators has some trouble to redirect values inside tuples.

6.1 Timer

There are two modes: setting time or active timer. When you want to set the time, the button Left increments the minutes while Right increments the seconds. When the desired time is set, just press Select and the timer starts.

The other mode is when the timer is running. If you press reset or you wait until the countdown is over, the timer goes to the setting time mode, with the last value used.

Note The timer can be quite slow. It is because each time I lookup for a timestamp value, I make iTask wait 1 second. (that's why there is like 3-4 seconds of delay after pressing Select)

6.2 Team counter

The buttons up and down respectively increment left and right teams' scores. The reset button makes both scores be zero.

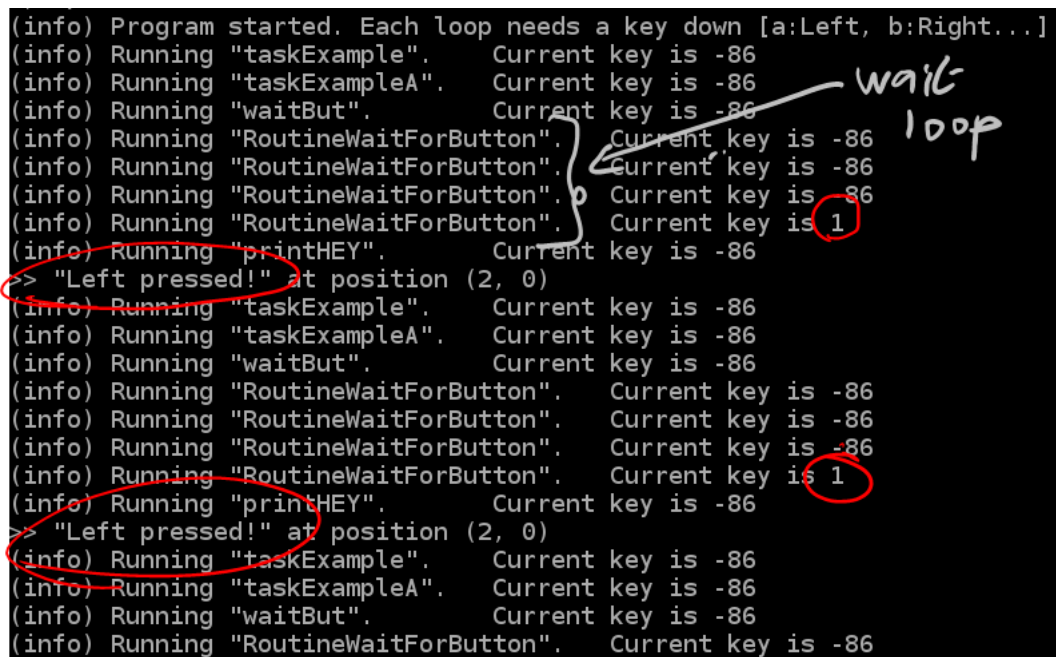
7 Current state

7.1 C++

The C++ part is not entirely working: everything using tuples but simple argument passing is not working. In fact, when tuples are nested (which is a requirement in this DSL when we combine tasks partially), my code is buggy with types, some stuff are missing to make it work correctly.

In order to debug and test my C++ outputs, I made a basic C++ code which compiles on a normal gcc (the file `manager.cpp`). It has a loop which asks for a keyboard input (for a char). This input would correspond to the buttons inputs. `a` is Left, `b` is Right, etc.

For example, a simple program displaying something after a specific key is down, works. Here is an example of the output:



```
(info) Program started. Each loop needs a key down [a:Left, b:Right...]  
(info) Running "taskExample". Current key is -86  
(info) Running "taskExampleA". Current key is -86  
(info) Running "waitBut". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is 1  
(info) Running "printHEY". Current key is -86  
> "Left pressed!" at position (2, 0)  
(info) Running "taskExample". Current key is -86  
(info) Running "taskExampleA". Current key is -86  
(info) Running "waitBut". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is 1  
(info) Running "printHEY". Current key is -86  
> "Left pressed!" at position (2, 0)  
(info) Running "taskExample". Current key is -86  
(info) Running "taskExampleA". Current key is -86  
(info) Running "waitBut". Current key is -86  
(info) Running "RoutineWaitForButton". Current key is -86
```

A handwritten annotation "wait loop" with an arrow points to the sequence of "RoutineWaitForButton" tasks. The value "1" in the "Current key is 1" lines is circled in red.

We can see the program handles multiple tasks.

Due to the conception of the DSL, to pass an information from a task A to a task B executing in the middle a separate task (i.e. no data come from that task), we need to use `RunTasksSeqRedirect`. This instruction needs two tasks (the first one takes type `in1` and output `out1`, while the second one takes type `in2` and output `out2`), an input for the task `in1`, an input of any type `t`, and a unifying task taking `out1` and `t`, outputting `in2`. This construction is a bit complex, and I didn't have time to write the C++ writer for that. Unfortunately, each time we want to listen to a button and then do something with some previous data, we need such a structure.

7.1.1 How to test generated C++ code

The generator just generates the "middle" of the code: all the fixed parts are not generated. To test the generated code, you have to paste it inside the comments `[tasks-definition]` and `[/tasks-definition]` in the `loop` function of the C++ file `manager.cpp`.

Also, you need to change the `CALL` line of the function `setup`: it tells the program which task and input parameters to start with.

7.2 iTask simulation

The iTask simulation is working, but I did not have time to try every features. There is one little bug I noticed, I've already talked about it in part 4.

Also, the instructions relative to the extraction of tuples (`gFst` and `gSnd`) are not working. Since we can get around this problem calling a task (values of an input tuple will be mapped to the different input parameters of a task), I didn't dig into that issue more.

Compile To compile the iTask simulation, just run `cpm index.prj`.

7.3 Lack in my task design

In order to handle recursion of tasks, I needed to identify a task uniquely: otherwise, it would result in some infinite C++ code generation.

I thought about comparing tasks by inspecting its body and signature, but I was unable to find something convincing: even if two tasks have the exact same body and signature, even same calls, it would not guarantee any equality.

Hence, I needed some easy way to identify a task. I thought about attributing automatically an unique integer, but I didn't find any way to do so (that problem of identifying a task always reduces to finding a way of establishing equality between tasks).

I eventually decided to let the programmer decide of this id, by specifying an unique `string` name. That leads to possible (big) problems: if the programmer copy/paste or simply give a same name for two tasks, then the C++ generator will act as if only the first one exists.

8 Notes

I think my DSL and the way I handled the C++ generation is too complicated, I wanted to make something very general and usable, and I ended up with something quite heavy and not very funny to use: there are too many little things to take into account, and then the result is not intuitive to use.

I wanted to introduce a nice way of handling shared data, with some possibility to resume a task when a data is edited, with some events, but I did not have time to do so.

I also thought of attributing a timestamp to every task call: either 0 and the task would be executed as soon as possible, either a (future) timestamp, in order to delay the execution of a task. Finally, I handle time easily, by checking timestamp at each loop (which is not as nice).

About the code, there are not many comments, I'm really sorry about that, I spent a lot of time on writing it, I wished I could make it clearer, but I didn't have any time left to comment properly.

Anyway, that assignment was interesting and instructing, it has been quite challenging for me!