Arduino programing of ML-style in ATS

Kiwamu Okabe
METASEPI DESIGN
kiwamu@debian.or.jp

Hongwei Xi
Boston University
hwxi@cs.bu.edu

Abstract

Functional programming languages often require a large run-time environment supported by the underlying OS to ensure various forms of safety during code execution. For instance, memory safety is usually achieved through systematic garbage collection (GC), which may not be available for embedded programming or should even be avoided on purpose due its adverse effect on predictability.

In this talk, we demonstrate that programming in ATS, a language with a functional core of ML-style, can be effectively employed to construct programs running on Arduino Uno (of 2KB SRAM and 8-bit CPU).

1. Introduction

Functional programming (FP) is gain popularity steadily. In general, FP is associated with the need for a big run-time environment supported by the underlying OS. For instance, functional programming languages (FPLs) like SML, OCaml, and Haskell fall into this category. This need makes it difficult to directly employ these languages in embedded programming where resources are far more limited and constraints (resource-wise and performance-wise) are stricter. As an example, memory safety in FP is usually achieved through systematic garbage collection (GC), which may not be available for embedded programming or should even be avoided on purpose due its adverse effect on predictability.

There have already been many attempts to address the issue of applying FP to embedded programming.

One proposed approach is to have the run-time needed for FP supported by a tiny OS (running on a VM) [1]. Of course, this approach is not applicable if the underlying device is still too limited for the tiny OS.

Another approach is to build a domain-specific language (DSL) based on a host language of functional style (e.g., Haskell) [2] and rely on the type system of the host language to ensure type-safety. However, a DSL often lacks flexibility, and can incur a high cost in terms of programming productivity.

Yet another approach is to directly employ a general purpose FPL in the construction of programs running on bare metal hardware [3], making trade-offs between safety with flexibility. This is the approach we take here as well.

In this talk, we introduce ATS [4] as a FPL for constructing programs running on Arduino Uno [5] (of 2 KB SRAM and 8-bit CPU core). We give a concrete example showing that higher-order functions (of ML-style) can be supported in this rather limited environment. We also present some measurement to show that binaries generated from ATS source are very close (in terms of size) to those generated from the C counterpart.

2. ATS programming language

ATS is a programming language equipped with a highly expressive type system rooted in the framework Applied Type System [4]. In particular, dependent types (of DML-style) and linear types are supported in ATS. For instance, we can use dependent types to ensure statically (that is, at compile-time) the absence of out-of-bounds array subscripting at run-time; we can use linear types to prevent resources from being leaked; etc.

Binaries generated from ATS source tend to be very compact and have minimal dependency on POSIX API. For programming Arduino boards, we can use ATS to construct code that does not rely on GC or any use of malloc/free. While this means that we have to stay away features that do need GC or malloc/free, we can still make use of higher-order functions (by stack-allocating closure-functions) and many other programming features of ML-style (e.g., pattern matching, loops based on tail-recursion).

3. Arduino Uno board

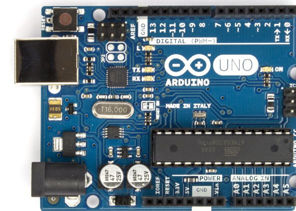

Figure 1. Arduino Uno board

Figure 1 shows an Arduino Uno board that has following summary of specification:

- Architecture: AVR (8-bit Harvard architecture)
- Flash Memory: 32 KB
- SRAM: 2 KB

Also the board has many pins that can connect the other function boards (that is, shields). Examples of shields include LCD screen, Ethernet, WiFi, Motor control. Clearly, it is difficult for any applications running on the board to use GC due to its tiny memory (2 KB). Using (customized) malloc/free may be possible but certainly requires great caution.

4. Compile flow for Arduino application

ATS can be readily used as a front-end to C. Figure 2 outlines a cross-compilation environment for the AVR architecture. The ATS compiler (`patsopt`) translates an ATS source file (e.g. `main.dats`) into an C source file (e.g. `main_dats.c`). The C code is then compiled by a C compiler (`avr-gcc`) into an object file (e.g. `main_dats.o`). Finally a linker (`avr-ld`) combines several object files (including those generated from ATS source) into a single executable (e.g. `main.elf`).

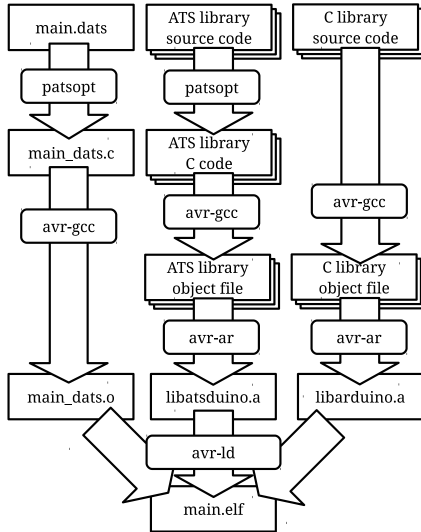

Figure 2. Compile flow for Arduino application

5. Arduino programming in ATS

In this section, we translate a sample Arduino program (written in C) into ATS. The following code is taken from Example 04 [6], which gradually fade in an LED:

Listing 1. C code to fade an LED in

```
1  #define LED 9
2  #define DELAY_MS 10
3
4  int main() {
5          int i;
6
7          init();
8          pinMode(LED, OUTPUT);
9
10         while (1) {
11                 for (i = 0; i < 255; i++) {
12                         analogWrite(LED, i);
13                         delay(DELAY_MS);
14                 }
15         }
16
17         return 0;
18 }
```

Listing 1 shows original code of C language. We can translate it into ATS as is given in listing 2.

Listing 2. ATS code to fade an LED in

```
1  #define LED 9
2  #define DELAY_MS 10.0
3
4  typedef analog_w_t = natLt(256)
5
6  implement main0 () = {
7    fun fadein() = let
8      var fwork = lam@ (n: analog_w_t) =>
9        (analogWrite(LED, n); delay_ms(DELAY_MS))
10   in
11     // type-error if 256 changes to 255 or 257
12     int_foreach_clo(256, fwork) // higher-order
13   end // end of [fadein]
14   val () = init ()
15   val () = pinMode (LED, OUTPUT)
16   val () = (fix f(): void => (fadein(); f()))()
17 }
```

The function `analogWrite` controls a given LED using pulse width modulation (PWM), taking as its second argument a natural number less than 256 (that indicates brightness).

The function `fadein` gradually increases the brightness of LED by calling `analogWrite`. It is implemented with a call to a higher-order function `int_foreach_clo`; given `256` and `fwork`, `int_foreach_clo` calls `fwork` on natural numbers from 0 until 255, inclusive. Note that `fwork` is a closure-function allocated in the frame of `fadein`. In particular, there is no dynamic memory allocation involved.

The fix-expression implements a non-terminating loop for calling `fadein(0)` repeatedly.

6. Binary size efficiency of the ATS code

Example	C	ATS
01	1145 byte	1203 byte
02	1135 byte	1173 byte
03C	1203 byte	1231 byte
04	1447 byte	1493 byte
05	1635 byte	1571 byte
06A	1431 byte	1477 byte
06B	1421 byte	1467 byte

Table 1. Binary size efficiency on examples of the Arduino book

In Table 1, we give a comparison between the binary size of the code generated from C programs and that of the code generated from the ATS counterparts of these C programs. In each of the presented case, the ATS version is only slightly larger (by fewer than 50 bytes) than the C version. This should be reasonable for practical use.

7. Conclusion

We have given a brief presentation in support of the claim that the ATS programming language can be employed effectively for constructing programs running on bare metal hardware such as 8-bit Arduino. It is shown that closure-functions can be stack-allocated to support the use of higher-order functions in the absence of GC. There are many other features in ATS for supporting safe and efficient low-level programming that cannot be presented here due to space limitation. For instance, safe manual memory management can be based on linear types; template-based programming allows for (extreme) late-binding of funtion calls; etc.

References

[1] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*.

[2] Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. Building embedded systems with embedded dsls. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*.

[3] Kiwamu Okabe and Takayuki Muranushi. Systems demonstration: Writing netbsd sound drivers in haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*.

[4] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*. Springer-Verlag LNCS 3085.

[5] Arduino Uno. `http://arduino.cc/en/Main/ArduinoBoardUno`.

[6] Massimo Banzi. *Getting Started with Arduino*. O'Reilly, 2 edition, 2011.