Building Embedded Systems with Embedded DSLs

(Experience Report)

Patrick C. Hickey Lee Pike Trevor Elliott James Bielman John Launchbury

Galois, Inc.

{pat, leepike, trevor, jamesjb, john}@galois.com

Abstract

We report on our experiences in synthesizing a fully-featured autopilot from embedded domain-specific languages (EDSLs) hosted in Haskell. The autopilot is approximately 50k lines of C code generated from 10k lines of EDSL code and includes control laws, mode logic, encrypted communications system, and device drivers. The autopilot was built in less than two engineer years, thanks to the productivity gains of EDSLs. This is the story of how EDSLs provided the productivity and safety gains to do large-scale low-level embedded programming and lessons we learned in doing so.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords keyword1, keyword2

1. Introduction

Embedded programming involves the lowest levels of abstraction. Most development is in low-level languages, like C or assembly, and programs interact intimately with the hardware. Embedded domain-specific languages (EDSLs) are in some sense at the other end of the software spectrum: they are often embedded (in a different sense of the word!) in high-level programming languages such as Haskell or ML, and are used to lift the programmer's abstraction level.

That said, there is no reason in-principle why EDSLs cannot be used for embedded programming; this report is about our experience in building new EDSLs for embedded programming and the benefits and difficulties in using them. Our experiences are based on building an autopilot system called SMACCMPilot using our EDSLs. The breadth and scope of the project sets it apart. The autopilot software is a complete embedded system that includes not just the core flight control algorithms, but also device drivers, encrypted network stack, mode logic, and concurrency and task management. As far as we know, it is one of the largest (open-source) embedded systems projects developed using the EDSL approach.

Our story is a largely positive one: we developed the Ivory and Tower languages and their (EDSL) compilers from scratch in approximately 14 engineer-months then we used them to build the SMACCMPilot hardware support and application in another 22 engineer-months. We achieved a dramatic increase in productivity as well as code quality. By construction, the generated C excludes large classes of errors and undefined behaviors.

Our goal in this paper is to summarize some of our lessons-learned. While we use specific examples, the lessons apply more generally to large embedded system design in EDSLs. Our target audience includes both researchers developing new EDSLs for low-level programming as well as practitioners considering using EDSLs.

All of the EDSLs described herein, as well as SMACCMPilot itself, are open-source software. Further documentation and links to the sources are available at `smaccmpilot.org`.

2. Ivory: Safe C Programming

At face value, our approach sounds audacious if not ludicrous: faced with a deadline for developing a new high-assurance autopilot system in one-and-a-half years, start by designing a new programming language and compiler from the ground up.

Of course, developing an EDSL is not the same as developing a stand-alone compiler. Much of the typical compiler tool-chain, such as the front-end parser/lexer, is provided for free by the host language. It took approximately 6 engineer-months to create the Ivory language and compiler, which totals about 6k lines of Haskell code.

The language we developed for generating safe embedded C code is called *Ivory*. Ivory compiles to restricted C code suitable for embedded programming. Ivory shares the goal of other "safe-C" standalone languages and compilers like Cyclone [8] and Rust [13]. Our main motivation for not using those languages is our desire for a convenient, Turing-complete, type-safe macro-language (Haskell) to improve our productivity.

There have also been some "safe-C" EDSLs including Atom [7], Copilot [11], and FeldSpar [2]. The most significant difference between these languages and Ivory is that they are focused on pure computations (e.g., FeldSpar is a DSL for digital signal processing), and do not provide convenient support for defining in-memory data-structures and manipulating memory. Ivory is designed to be a EDSL that can be used for writing safe memory-manipulating embedded C.

Ivory also makes contributions from a programming language perspective, namely in its expressiveness and type-safety. We overview each, then present a small example, to give the reader a feel for the language.

Expressiveness Regarding the expressiveness, Ivory has a variety of useful features, including:

- *Memory-areas*: the ability to allocate stack-based memory and manipulate both local and global memory areas [3].

- *Product types*: C structs with well-typed accessors.
- *FFI*: typed interfaces for calling arbitrary C functions.
- *Bit-fields*: support for typed manipulation of bit-fields and registers [4].

We built Ivory with some limitations to simplify generating safe C programs. Ivory does not support heap-based dynamic memory allocation (but global variables can be defined). C arrays are fixed-length. There is no pointer arithmetic. Pointers are non-nullable. Union types are not supported. Unsafe casts are not supported: casts must be to a strictly more expressive type (e.g., from an unsigned 8-bit integer to an unsigned 16-bit integer) or a default value must be provided for when the cast is not valid. The most common unsafe C cast is not possible: no void-pointer type exists in Ivory.

In Ivory, these have not been limiting factors, particularly because of the power of using Haskell as a macro system. For example, while arrays must be of fixed size at C compile-time, we can define a single *Haskell* function that is polymorphic in the array size that becomes instantiated at a particular size at each use site.

Type-checking Ivory's domain-specific type checking focuses on guaranteeing memory safety and helping programmers reason about their programs' nonfunctional behaviors more easily.

In addition, Ivory programs have an *effects* type associated with them, implemented as a parameter to the Ivory monad. There are three kinds of effects tracked:

- *Allocation effects*: whether a program performs (stack-based) memory allocation as well as whether pointers point into global or stack memory.
- *Return effects*: whether a program contains a `return` statement.
- *Break effects*: whether a program contains a `break` statement.

Allocation effects allow memory allocation to be restricted and tracked at the type level. For example, from a program's type alone, we can determine whether it allocates memory on the stack, making stack usage easier to track. More importantly for memory-safety, allocation effects also ensure Ivory programs contain no dangling pointers: it is a type error to return a pointer to locally-allocated memory.

Return and break statements fundamentally affect control-flow and can result in unexpected behavior by breaking out of the current block or returning from a function. For example, in a top-level while loop implementing an real-time operating system task, there should be no break or return statements; we can enforce this with the type system. Tracking these effects is novel, we believe, and particularly important in the context of an EDSL in which programs are generated and manipulated heavily in the host language.

In an EDSL, we have at least two options for type checking: (1) write a domain-specific type-checker *in* Haskell (relying on Haskell's type-system just for macro-language type-checking), or (2) embed the domain-specific type checker into Haskell's type system.

We were motivated to pursue option (2) because it allows us to discover problems sooner in the development cycle. In the case of option (1), we only find out about problems in the program's AST during code generation. Option (2) ensures that all macro and library code is typed correctly, independent of its use in the generated code. We discuss the issues of finding errors early on in more detail in Section 5.

When we began developing Ivory, our hypothesis was that recent type-system extensions to the Glasgow Haskell Compiler make it feasible to embed the invariants necessary to ensure memory-safe C programming into the type-system [10]. From a practical standpoint, Ivory demonstrates just how far the type-system has come, allowing us to replicate the type safety of compilers like Cyclone, etc.

```
[ivory|
struct fooSstruct
  { bar :: Stored Uint8
  ; baz :: Array 10 (Stored Sint16)
  }
|]

setBaz :: Def ([Ref Global (Struct "fooStruct"), Sint16] :-> ())
setBaz = proc "setBaz" $ \ref val -> body (prgm ref val)

prgm :: Ref Global (Struct "fooStruct") -> Sint16 -> Ivory eff ()
prgm ref val = arrayMap $ \ix ->
                 store ((ref ~> baz) ! ix) val
```

```
// foo_source.c
#include "foo_module.h"

void setBaz(struct fooStruct* n_var0, int16_t n_var1) {
  for ( int32_t n_ix0 = (int32_t) 0
      ; n_ix0 <= (int32_t) 9
      ; n_ix0++ ) {
      n_var0->baz[n_ix0] = n_var1;
  }
}

// foo_module.h
struct fooStruct {
    uint8_t bar;
    int16_t baz[10U];
};

void setBaz(struct fooStruct* n_var0, int16_t n_var1);
```

Figure 1. Example Ivory module definition

We do not have space to adequately describe Ivory's type system; we leave that to a forthcoming paper. Here we will note that the embedding depends on the use of data kinds [17], type families [15], and rank-2 polymorphism [9].

Ivory example We present a small example of Ivory code. The example omits many features of the language, but should give the reader a feeling for it.

Consider Figure 1, in which an Ivory program is shown, as well as the corresponding generated C sources and headers (making a few syntactic changes to the C for readability, not relevant to the example).

First, we define a struct (or product type) using a quasiquoter that is part of the Ivory language. The generated Ivory code generated by Template Haskell [16] constructs a struct definition containing two fields consisting of an unsigned byte and an array of 10 signed 16-bit integers. Template Haskell also constructs a new type-level literal, `"fooStruct"`, that is unique to the defined struct. The `Stored` type constructor signifies that the value is allocated in-memory [3]. The `Array` type constructor takes a type-level natural number as a parameter (available as a Glasgow Haskell Compiler extension) to fix the size of an array.

A procedure, corresponding to a C function, has a type of the form

```
Def (params :-> out)
```

where `params` are the procedure's parameter types and `out` is its return type. The procedure `setBaz` takes two arguments and its return type is unit, corresponding to the `void` type in C. The types of the procedure's arguments are types in a type-level list: the first argument is a *reference*, a non-null pointer by construction, to a struct, and the second argument is a signed 16-bit integer. The `Ref` type constructor takes a *scope* type and a memory-area type.

The scope type denotes either stack-allocated scope, or global (and statically allocated) scope. In the example, we expect the reference to be to a global.

Procedures are defined with the `proc` operator that takes a string, corresponding to the name of the function that will be generated in C, and a function from the procedures arguments to its body. The body of the function is an Ivory program that sets each element in the `baz` field of the struct with the value `val` passed to it, leaving the `bar` field unchanged.

Following [3], Ivory guarantees memory-safe array access in the type system since array lengths are statically known. Ivory provides an `arrayMap` operator that applies a function to each valid index into the array. The function applied in this case is a `store` operation that takes a reference to a memory area, a value, and stores the value in the area. It is a type-error if the value's type and memory-area's type do not match.

The operation (`ref ~> baz`) takes the struct reference and returns a reference to the `baz` field. The bang (`!`) operator takes a reference to an array, an index, and returns a reference to the value at that index. The safety of indexing is maintained since the operator has the type

```
(!) :: Ref s (Array len area) -> Ix len -> Ref s area
```

tying the length of the array to the maximum index. For example, an index type (`Ix 10`) supports index values from 0 to 9.

The example only shows a small part of Ivory's language and does not exhibit some of its additional features to prevent unsafe programs. For example, if `setBaz` had allocated stack memory and created a reference to it, then tried to return the reference (creating a dangling pointer), it would result in a type error.

Additionally, for application-specific properties that cannot be type-checked, Ivory permits the insertion of assertions, assumptions on arguments, and requirements on return values. Ivory also automatically inserts checks for arithmetic underflow/overflow and division-by-zero. All these checks are useful during testing and we have used them to assist with static analysis and model-checking the generated C.

3. Tower: from Functions to Architectures

In many embedded systems, programmers produce an entire system of software that interacts with multiple input and output peripherals concurrently using a real-time operating system (RTOS). Typical RTOSes provide just a few low-level locking and signaling primitives for scheduling. Since microcontrollers do not have the virtual memory managment units (MMUs) found on larger processors, the RTOS kernel cannot protect any system memory against badly behaved user code. These restrictions put significant burden on programmers: they must ensure all tasks, and all communication between tasks, are implemented correctly.

During our initial development of SMACCMPilot, we found ourselves writing high-quality C functions in Ivory, which guarantees memory-safety of the generated code. But whenever we needed "glue code" to implement inter-process communication, initialize data-structures, read the system clock, lock the processor, etc., we were forced to abandon our well-typed world and tediously use C directly via Ivory's foreign function interface. Furthermore, the hand-written C is OS-specific, meaning it would have to be rewritten for any OS port.

Extending Ivory The hand-written glue code was ruining both our productivity and our assurance story. We wanted a language to describe the structure of the glue code that would generate it for us. Our key insight was that such an EDSL could be built as a macro over Ivory, using Ivory's code-generation facilities, without losing anything.

From these ideas, the Tower EDSL was born. You can think of Tower as an extension to the Ivory language, designed to deal with the specific concerns of multithreaded software architecture. Tower still allows the programmer to use all the low-level power of Ivory for general programming, but uses a separate language for describing tasks and the connections between them. This is one of the great productivity features of working with EDSLs: if you discover the language you are using is difficult, tricky, or unsafe for solving a particular problem, you can easily extend that language with a library without modifying the compiler.

In Tower, one specifies tasks and communication channels, and the Tower compiler generate correct Ivory implementations, as well as architecture description artifacts. Tower hides the dangerous low-level scheduling primitives from the user, and keeps type information for channels (i.e., the datatype of the channel message), expressed as Ivory types, in the Haskell type system.

Tower allows the programmer to describe a static graph of channels and tasks. For the intended use case in high assurance systems, a static configuration of channels and tasks makes it easy to reason about memory requirements, and permits the system to be analyzed for schedulability.

Multiple interpreters In the Tower front end, the programmer specifies a system that can be compiled to multiple artifacts.

Tower is designed to support different operating systems via a swappable backend. Since all code that touches operating system primitives is generated by Tower, it is easy for the user to specify a system and compile it for different operating systems. Tower supports both the open-soure FreeRTOS[6] as well as the formally-verified eChronos RTOS[5] developed by NICTA.

Tower also has a backend which generates a system description in the Architecture Analysis and Design Language (AADL) [14]. We also built a backend for the Graphviz dot language. These output formats make it possible to visualize, analyze, and automatically check properties about the system.

We were pleased by the productivity improvements and correctness guarantees the Tower language provided. In all, it took about 4 engineer-months to build Tower, and a total of about 3000 lines of Haskell code.

Tower example In Figure 2, we sketch a small Tower example that is representative of a device driver that blinks an LED. Small simplifications to Tower have been made in the code, eliding details relating to code generation and backend selection.

In the first column of the figure, the communication architecture is defined in the `Tower` monad. The program initializes a unidirectional channel between two tasks as well as the tasks themselves. A channel, or queue, consists of transmit (`tx`) and receive (`rx`) endpoints, respectively. The `blinkTask` task is an RTOS task that will send output to the `lightswitch` RTOS task via an RTOS-mediated channel. The `lightswitch` task toggles the LED based on the incoming Boolean values. (In the third column, a graph of the tower program is shown, generated from the Tower compiler's Graphviz dot output, showing the architectural structure of the two tasks as well as the queue between them.)

To conserve space, we only define `blinkTask`. The second column contains the definition of `blinkTask`, defined in the `Task` monad. The `blinkTask` task takes a channel source and returns a task. The task first initializes an *emitter* for the channel then creates a reference to allocated memory that is private to the task. Every 100 milliseconds, an Ivory action is taken. In this case, the action is to call Ivory function `blinkFromTime` that is executed whenever the task is enabled (we elide the implementation of `blinkFromTime` in this example). The boolean value `res` is then emitted on the channel.

```
blinkTower :: Tower ()
blinkTower = do
  (tx,rx) <- channel
  task "blink" (blinkTask tx)
  task "lightswitch" $
    onChannel rx $
      \lit -> do
        ifte_ lit (turnOn light)
                  (turnOff light)
```

```
blinkTask :: ChannelSource (Stored IBool)
      -> Task ()
blinkTask chan = do
  tx  <- withChannelEmitter chan
  res <- taskLocal
  onPeriod period $ \now -> do
    res <- call blinkFromTime now
    emit_ tx res
  where period = Milliseconds 100
```

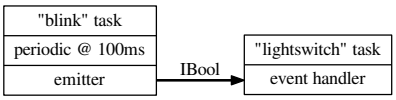

Figure 2. Tower (Col. 2), Task (Column. 1), Graphviz output (Col. 3)

4. SMACCMPilot: a High-Assurance Autopilot

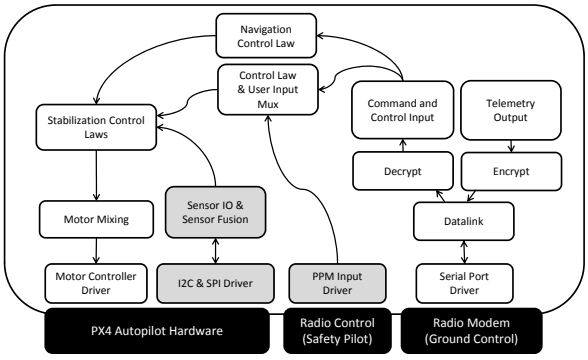

Figure 3. Simplified diagram of SMACCMPilot software architecture. Tasks written in Ivory are shown as white boxes, tasks implemented in legacy C++ code are gray boxes, channels are arrows, hardware components are black boxes.

Our main use for Ivory and Tower thus far has been in building a robust autopilot. The result is SMACCMPilot, an open-source (BSD licensed) autopilot system for quadcopter unmanned air vehicles (UAVs). It is complete embedded system that includes low-level IO peripheral drivers, an encrypted communication protocol stack, and several layers of control systems.

SMACCMPilot runs on open source flight controller hardware from the PX4 Autopilot project [12]. The hardware platform is a custom printed circuit board with an ARM Cortex M4 microcontroller and the accelerometer, magnetometer, gyroscope, and barometer sensors used to determine the orientation and altitude of the vehicle.

A simplified software architecture of SMACCMPilot is shown in Figure 3. The flight control software is primarily responsible for reading the sensors, estimating the vehicle's attitude and position using sensor fusion, calculating control outputs, and sending motor power commands to the motor controllers. Higher level controllers manage navigation, and an encrypted command, control, and telemetry link interprets ground station instructions and sends system state to the operator.

The result is a reasonably complex piece of embedded software. SMACCMPilot has 30 tasks connected by 47 channels, and 57 globally shared state variables. Most of those shared state variables are controller tuning parameters, which can be modified by commands sent over the telemetry link.

SMACCMPilot was developed alongside the Ivory/Tower tools. The complete system took approximately 22 engineer-months to develop. The low level drivers for the system were written first in C, then transliterated to Ivory as the language became mature enough to support them. We built a stack for command, control, and telemetry, encapsulated in an encrypted packet protocol. A few components from the ArduPilot open source project, the biggest of which is a 10kloc C++ library for inertial sensor fusion, are still used inside SMACCMPilot.

Complexity comparison The SMACCMPilot application code is 10kloc of Ivory, the board support code is 3kloc of Ivory (and Tower), and the telemetry link binary packing and unpacking code is a machine generated 10kloc of Ivory code. When compiled, the complete application is 48kloc of generated C code, and depends on some external C libraries to implement the operating system (4kloc) and other functions, such as sensor fusion. This compares favorably to existing open source flight controller systems.

We can compare this to two systems which have a similar feature set and run on similar hardware to SMACCMPilot. The ArduPilot project [1] and the PX4 project are popular open source autopilots, both of which have more high level navagation and autonomy capabilities than SMACCMPilot. Both implement all of the low level drivers to support similar (or identical) microcontroller based flight controller boards, comparable control laws, and implement the same MAVLink telemetry protocol.

The ArduPilot project is over 60kloc of C++, runs in three pseudo-threads, and supports at least four distinct autopilot hardware platforms. The PX4 Autopilot software stack has 25kloc of C/C++ application code, 25kloc of C/C++ platform support code, and depends on the large (50kloc+) NuttX operating system.

5. Lessons Learned

In this section, we discuss some of the benefits and challenges of using EDSLs for embedded programming, focusing on those that were surprising to us, despite our teams' previous experience in functional programming and embedded development.

Our experience using Ivory and Tower to build SMACCMPilot has been an extreme lesson in dog-fooding EDSLs. We had multiple developers writing the compilers and using them to build applications concurrently. We learned a few lessons that are relevant to any compiler development but particularly relevant to EDSL development.

Type-checking for embedded programming Build times are non-trivial for large software systems. At the time of writing, a fresh build of SMACCMPilot and associated test programs is over seven minutes of real time (and 12 minutes of CPU time since we have a multi-threaded build system). One reason the build time is so large is that it requires Cabal (the Haskell package manager) to discover library dependencies and install packages, compile the Haskell sources, and then compile the C sources. As well, some sources are compiled multiple times for different targets on multiple operating systems.

Then, to execute the software on the embedded device, we have to write the software to the device's memory via a JTAG

programmer or a serial bootloader, which takes on the order of ten seconds.

All this is to say that the end-to-end debug cycle might mean testing a small number of changes to Ivory or Tower per hour. Clearly, the debug cycle in embedded development particularly motivates us to make fewer bugs and to discover them early.

During development, it became apparent how useful Haskell type-checking is for embedded programming. As described in Section 2, we have embedded Ivory's type system in Haskell's. Thus, domain-specific type-errors are caught during Haskell type-checking. Type-checking, and other static warnings reported by GHC, are nearly instantaneous since it can be done on a module-by-module basis. The upshot is that Ivory programs that would generate unsafe C programs are caught immediately. The type system tracks the global or stack frame provenance of references, as well as structure accessors and array indicies, to ensure all well-typed Ivory programs generate memory-safe C.

In addition, we have found it useful to detect potential bugs even if the C compiler might also detect them. To take one example, consider unused variable declarations. While a C compiler can detect this, perhaps late in the compilation phase, we discover these warnings nearly instantaneously during type checking. Moreover, the more preprocessing we can do in Haskell, the more potential errors we may find, and with a better relation to the source.

Type-safe system plumbing The most tedious part of adding many new features to SMACCMPilot is the business logic in Tower: defining a new task, and then plumbing all the communication channels through the code. There is nothing conceptually difficult in doing so. It simply requires modifying the arguments to a Haskell function that generates a Tower task (or modifying the fields of a data-type if channels have been grouped together). Channels are typed, so type-checking detects most plausible inter-task communication errors.

Stepping back, the idea that plumbing arguments to Haskell functions is the hardest part of embedded development is amazing. We are not dealing with bugs in low-level OS interfaces, we are not making timing errors in communication, we are not dealing with type-errors like you might find in raw C (where data might be cast to `void*` or `char[]`).

Because plumbing is so easy, it encourages us to improve modularity in the system. Defining a new RTOS task is easy, so we might as well modularize functionality to improve isolation and security. For example, in the ground station communication subsystem, encryption and decryption are each executed in isolated tasks, simplifying the architectural analysis of the system. As we noted in Section 4, our system is significantly more modular than other autopilots.

Faking a module system In Ivory and Tower, top-level functions and structures are packaged into a Haskell data structure to provide to the Ivory compiler. The onus is on the programmer to package up all the necessary components.

On one hand, the approach provides the programmer control over how to modularize the generated C code, deciding which definitions to put in a C source or header file. On the other hand, we have found it to be verbose, tedious, and error-prone. Generally, we want the C files to have similar structure to the Haskell modules in which Ivory programs are written. From that respect, the Ivory module system simply duplicates the Haskell module system.

Worse, moreover, is when the programmer forgets to package a definition. The error only becomes apparent at C *link* time, near the end of a long build process. Missing definitions have plagued our builds.

We could move symbol resolution up the build cycle to the C-code generation phase. Ideally, we would move it up the build cycle even further. We are currently exploring the use of Template Haskell to generate Ivory modules at compile-time to assist the programmer.

```
data Cond eff a =
  Cond IBool (Ivory eff a)

(==>) :: IBool -> Ivory eff a
      -> Cond eff a
(==>) = Cond

cond_ :: [Cond eff ()]
      -> Ivory eff ()
cond_ [] = return ()
cond_ ((Cond b f):cs) =
  ifte_ b f (cond_ cs)
```

```
cond_
  [ x >? 100 ==> ret 10
  , x >? 50  ==> ret 5
  , true     ==> ret 0 ]

ifte_ (x >? 100)
 (ret 10)
 (ifte_ (x >? 50)
   (ret 5)
     (ret 0))
```

Figure 4. Conditional Ivory macro.

Control your compiler If we were writing our application in a typical compiled language, even a high-level one, and found a compiler bug, we would perhaps file a bug report with the developers... and wait. If we had access to the sources, we might try making a change, but doing so risks introducing new bugs or at the least, forking the compiler. Most likely, the compiler would not change, and we would either make some *ad-hoc* work-around or introduce regression tests to make sure that the specific bug found is not hit again. Such a situation is notorious in embedded cross-compilers that usually have a small support team and are themselves many revisions behind the main compiler tool-chain.

But with an EDSL the situation is different. With a small codebase implementing the compiler, it is easy to write new passes or inspect passes for errors. Rebuilding the compiler takes seconds.

More generally, we have a different mindset programming in an EDSL: if a class of bug occurs a few times—whether resulting in the compiler or not—we change the language/compiler to eliminate it (or at least to automatically insert assertions to check for it). Instead of a growing test-suite, we have a growing set of checks in the compiler, to help eliminate both our bugs and the bugs in *all* future Ivory programs.

Everything is a library With an EDSL, and particularly a Turing-complete macro language, everything is a library. The distinction between language developers and users becomes ambiguous. As an extreme example, one can think of Tower as "just" a library for Ivory. A smal example is defining a conditional operator in terms of Ivory's if-then-else primitive as shown in Figure 4. All types above were introduced in Section 2. With the `cond_` operator, we can replace nested if-then-else statements as shown in the figure with more convenient conditionals, without modifying the language.

Because macros are so easy to define and natural in EDSL development, our biggest challenge is ensuring developers on our team put useful ones in a standard library, to be shared.

Semantics To take advantage of legacy cross-compilers, we are forced to generate C code from our EDSL. A large focus in designing Ivory is to allow expressive but well-defined programs. We believe Ivory cannot produce memory-unsafe C programs. However, undefined C programs can be generated from Ivory; for example, signed integer overflow and division-by-zero are undefined. Guaranteeing programs are free from these behaviors is decidable (the arithmetic is on fixed-width integer types), but intractable to prove automatically.

To assist the programmer, the Ivory compiler automatically inserts predicates into the generated code to check for overflow, division-by-zero, etc. The user defines the behavior of the program

if a check fails. For example, during testing, we define the checks to insert a breakpoint for use with a debugger. Another option may be to do nothing and rely on the semantics provided by the C compiler. Still another option might be to trap to a user-defined exception-handler.

Currently, SMACCMPilot contains approximately 2500 compiler-inserted non-trivial checks that cannot be constant-folded away. In the future, we hope to *prove* these checks never fail. We have used model-checking in a limited fashion, but naively applied; it does not scale to the size of SMACCMPilot.

There are two other semantics categories to consider: defined behavior and implementation-defined behavior. In Ivory, we attempt to eliminate almost all implementation-defined behaviors. For example, only fixed-width size types, like `uint8_t` or `int32_t`, can be generated. Implementation-defined sizes, like `int` or `char` are not used. We have found these to be dangerous: programmers might assume properties about the size of a type that do not hold in a non-standard architecture (e.g., that an `int` is at least 32 bits or that `char` is unsigned; both are implementation-defined). Such assumptions are particularly dangerous when porting code between different embedded platforms. Indeed, when we ported portions of ArduPilot, initially built for an 8-bit AVR architecture to a 32-bit ARM, we found these sort of implicit assumptions.

Finally, even defined behavior is not necessarily intuitive behavior. For example, in C, the defined behavior for arithmetic on values that have a size-type smaller than `int` is to implicitly promote them `int`s before performing the arithmetic.

For example, given

```
uint8_t a = 10;
uint8_t b = 250;
bool    x = a-b > 0;
bool    y = (uint8_t)(a-b) > 0;
```

`x` evaluates to 0 and `y` to 1, provided that

```
sizeof(uint8_t) < sizeof(int)
```

This behavior is important to the embedded programmer because, across various embedded processors and C compilers, integer sizes are often defined differently.

In Ivory, arithmetic is at the size of the operands, which we believe is more intuitive. We force the generated C to respect this semantics by inserting casts into expressions. So the Ivory expression `a-b` results in the C expression `(uint8_t) (a-b)`.

6. Conclusions

We have described our use of the Ivory and Tower EDSLs for building a large embedded system.

Many of the advantages of EDSLs for embedded programming relate to type-checking in Haskell. Of course, some bugs cannot be caught statically. For the most part, once type-checking is complete, we are confident that the bug is a logical bug. We do not spend our time chasing segmentation faults or strange undefined or compiler-dependent behaviors but rather focus on the bugs result from our misunderstanding of the application, not the programming environment.

What is next? In the next few years, SMACCMPilot will continue to grow. It, along with the Ivory & Tower tools, are open source, in the hope of engaging a broader community. We will add new hardware, new sensors, and new controllers so that it is not only one of the highest-assurance autopilots in existence but is competitive with others in terms of functionality.

In addition, we are looking to improve the usability of Ivory and Tower. For example, we are working to integrate verification tools more closely into the language. We have also begun to define quasiquoters for the languages so that C programmers might feel more at home with the language but power (Haskell) users can still enjoy the benefits of EDSL programming.

In short, we believe EDSLs can be brought down from the ivory tower (pun intended) to the grungy world of embedded programming.

Acknowledgments

This work is supported by DARPA under contract no. FA8750-12-9-0169. Opinions expressed herein are our own. A number of people have provided input and adivce; we particularly thank Kathleen Fisher, Iavor Diatchki, and Andrew Tridgell. Joe Kiniry and Adam Foltzer provided helpful comments on earlier drafts of the paper.

References

[1] APM Project. APM multiplatform autopilot suite. Website `http://ardupilot.com/`. Retrieved Feb. 2014.

[2] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar - an embedded language for digital signal processing. In *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 121–136. Springer, 2011.

[3] I. S. Diatchki and M. P. Jones. Strongly typed memory areas programming systems-level data structures in a functional language. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 72–83. ACM, 2006.

[4] I. S. Diatchki, M. P. Jones, and R. Leslie. High-level views on low-level representations. In *Intl. Conference on Functional Programming*, pages 168–179. ACM, 2005.

[5] eChronos. eChronos. Website `http://ssrg.nicta.com.au/projects/TS/echronos`. Retrieved Feb. 2014.

[6] FreeRTOS. FreeRTOS. Website `http://freertos.org/`. Retrieved Feb. 2014.

[7] T. Hawkins. Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP), 2008. Available at `http://cufp.galois.com/2008/schedule.html`.

[8] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Conference*, Berkeley, CA, USA, 2002. USENIX.

[9] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. pages 24–35, June 1994.

[10] S. Lindley and C. McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Symposium on Haskell*, pages 81–92. ACM, 2013.

[11] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer, 2010.

[12] Pixhawk. PX4 autopilot project. Website `http://pixhawk.org/`. Retrieved Feb. 2014.

[13] Rust. Rust. Website `http://www.rust-lang.org/`. Retrieved Feb. 2014.

[14] SAE-AS5506. *Architecture Analysis and Design Language*. SAE, Nov 2004.

[15] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *Intl. Conference on Functional Programming*, pages 51–62, Sept. 2008. ISSN 0362-1340.

[16] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, Dec. 2002.

[17] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012.