Programming Languages for High-Assurance Autonomous Vehicles

Extended Abstract

Lee Pike Patrick Hickey James Bielman Trevor Elliott
Thomas DuBuisson John Launchbury

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

Abstract

We briefly describe the use of embedded domain-specific languages to improve programmer productivity and increase software assurance in the context of building a fully-featured autopilot for unpiloted aircraft.

Categories and Subject Descriptors D.3.2 [*Specialized application languages*]

Keywords high-assurance, programming language design

1. Introduction

Domain-specific languages (DSLs) improve programmer productivity. In the "*embedded* DSL (EDSL) approach", a DSL is embedded within a general-purpose programming language. EDSLs simplify developing new languages and compilers since the developer can reuse infrastructure from the host language and does not need to implement a full parser, lexer, type-checker, etc.

While EDSLs improve productivity and are useful for rapid prototyping, can they improve the safety and security of software?

Yes. Over the past year and a half, we have shown you can have your cake and eat it too, using EDSLs to increase programmer productivity while ensuring generated embedded software does not suffer common bugs. As a case-study, we have developed a higher-assurance, open-source autopilot system for small autonomous vehicles called `SMACCMPilot`.[1]

A secure autopilot system encompasses the full spectrum of embedded system development including

- device drivers
- hard and soft real-time multi-tasking
- wireless networking (with radio controllers and a ground-control station)
- network cryptography
- multiple interacting control loops
- fail-safe subsystems and diagnostics
- hardware-in-the-loop testing

To address the complexity and heterogenity of autopilot programming, we have developed a new EDSL called *Ivory*. Ivory generates memory safe C code specialized for embedded systems programming, and its host language is Haskell, a pure, strong and statically-typed functional programming language [4]. As an EDSL, the Ivory compiler is under 4k LOCs.

To make memory safety easier to guarantee at compile time, Ivory restricts certain constructs. For example, Ivory restricts the following:

- No heap allocation. All memory allocation is on the control stack. All memory allocation is of statically-known sizes.
- User-code loop iterations are bounded by a constant. (There is one special `forever` loop combinator for implementing operating system tasks.)
- Size types are machine-independent (with the exception of floating point sizes).
- Expressions have no side-effects.
- No pointer arithmetic.
- Type casts are limited to information-preserving casts or casts that require a default value if truncation may occur (e.g., casting from a signed to unsigned value).

These restrictions are similar to Jet Propulsion Laboratory's "Power of 10" rules for safety-critical software design [3] except in Ivory, the rules are enforced by the compiler, not convention.

Ivory's type system is embedded in Haskell's parametric polymorphic type system and builds on powerful extensions including data kinds [7] (e.g., to distinguish memory areas [2]), type-level naturals [6] (for array lengths and loops), type-level strings (for struct definitions), and type families [5] (for controlling effects like memory allocation and `return` statements).

With the restrictions described above, Ivory takes a lot of flexibility from the programmer. But it adds something too: because Ivory is embedded in Haskell, the Ivory macro system is a type-safe, Turing-complete programming language, *for free*. As an EDSL, many of these restrictions are transparent to the programmer through the use of compile macros.

[1] The acronym 'SMACCM' stands for *Secure Mathematically-Assured Composition of Control Models*. The compilers and autopilot are available at `smaccmpilot.org`, licenced as BSD3.

PLPV '14, January 21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2541568.2541570

For example, consider the case of loops. At C code generation time, loop bounds become fixed, but in Haskell, the macro language, we can write a *polymorphic* loop function, such as the following:

```
loop :: Ix n -> Ivory ...
loop bound = ...
```

`loop` is an Ivory function, where its type is given after the double colons and its implementation is given on the following line (irrelevant details are elided with ellipses). The function takes an index (`Ix n`), where `n` is a type-level natural number, and produces an Ivory computation that performs an action. The loop combinator will invoke this Ivory computation with an index counting up from 0 to `(bound - 1)`—in C, this generates a `for` loop. The function `loop` can be used with any bound, but at compile time, `loop` is specialized for each context it is called.

Macros are used in everything from generating safe bit-data manipulations (e.g., for device drivers) to architectural coordination.

The case of architectural coordination macros is noteworthy. Early in our project, we found ourselves generating memory-safe C programs from Ivory for individual real-time operating system (RTOS) tasks, but having to write the "glue code" that manages the tasks' setup, initialization, synchronization, and communication in C. Rather then write a new EDSL to generate glue code, we reuse Ivory for this purpose as well by building a set of macros over Ivory called *Tower*.

For example, to define an RTOS task called `fooTask` we might write a Tower/Haskell function as follows:

```
fooTask :: ChannelSource 10 (Struct "state")
        -> Task ...
fooTask src = do
    emitter <- withChannelEmitter src "src"
    onPeriod 250 ( \now -> do
      ...
      emit emitter ... )
```

The function takes as an argument a queue (channel) on which it broadcasts typed values that are `state` structures, which we leave undefined for this example. The queue can contain up to 10 values. In the task's definition, we extract the queue from the `Task` monad, and in this example, write a body that executes every 250 milliseconds. After performing some actions, the task emits a `state` on its channel.

In a sense, Tower is a separate language with a different syntax and semantics from Ivory, but it does not even have its own abstract syntax. Tower macros evaluate directly to Ivory code, using only a few RTOS-specific system calls, imported in Ivory as foreign C functions.

Tower does more than just simplify access to RTOS system calls. Channels and shared memory are statically typed in Haskell, and a graph describing communication channels between tasks is captured as a Haskell value during code generation. With this information, we can make security-critical guarantees, such as "all out-bound messages sent to the serial driver (sending data to the radio) pass through the encrypter first."

Because the Tower macro system generates the bulk of the code for composing Ivory components into the autopilot application, adding new functionality often has the feeling of writing a state-machine in Ivory then "plumbing" the new component into the system with Tower.

Moreover, a developer who writes macros that generate Ivory programs knows that no matter how complex her Haskell functions are, compiled C code is guaranteed to be memory safe. In addition, she can write macros over her DSL using general-purpose libraries without porting them or requiring a foreign-function interface. For example, we use QuickCheck, a library for automated test-case generation [1], to automatically generate Ivory programs which test the user's Ivory program.

At the time of writing, our autopilot application has taken about two engineer-years of effort, and generates approximately 50k lines of code (LOCs) of C code from approximately 5k LOCS of Ivory/Tower EDSL code. Comparing our work to similar open-source projects, we estimate that the autopilot application would be about 25k LOCs if hand-written in C.

Ivory's generated C code is typically more verbose and contains more duplicate statements than hand-written C. We've found the Ivory programmer often uses host-language (Haskell) macros to duplicate code where a C programmer would factor duplication using function calls. The Ivory language retains the flexibility to factor into functions if required for code size or performance, but often, Ivory's verbose code generation makes no difference to the modern optimizing C compiler.

Additionally, the generated C code has just under 2500 assertions that are automatically inserted by the Ivory compiler. Most of the assertions are checks on arithmetic underflow/overflow and division by zero. The assertions simplify testing and verification.

Conclusions EDSLs are not a panacea for building high assurance systems. We still write logical bugs, we still have to tune control systems, we still have to deal with hardware failure and the intricacies of cross-compilers, linker scripts, poor hardware documentation. We are far from formally specifying the requirements of an autopilot; doing so would be a research topic of its own.

However, we do not spend our time hunting down segmentation faults or buffer overflows. We do not make mistakes in using RTOS primitives to set up inter-task communication. We spend more of our time implementing and debugging core functionality. We write what appear to be high-level functional programs despite efficiently executing on a small embedded microprocessor.

In summary, EDSLs allow you to have your productivity and your assurance, too.

Acknowledgments

This work is sponsored by DARPA High-Assurance Cyber-Military Systems (HACMS) program. The SMACCMPilot autopilot project draws on the work of the ArduPilot open source project, and we thank the ArduPilot developer community and 3D Robotics for their time and support.

References

[1] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000.

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

[3] G. J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, 2006.

[4] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.

[5] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the International Conference on Functional Programming*, ICFP '08. ACM, 2008.

[6] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257. ACM, 1998.

[7] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.