Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell

The Programatica Team

OGI School of Science & Engineering at OHSU
20000 NW Walker Road, Beaverton, Oregon OR 97006, USA

Abstract. The development of high-assurance computational systems requires sophisticated engineering, comprehensive analysis, and careful management to ensure that the required levels of functionality and security are achieved. In this paper, we use the example of developing a high-assurance implementation of a crypto-chip to show how these needs are met using the results of the Programatica project at OGI. In particular, our approach combines three significant components: the use of Haskell as a semantically rich, formal modeling language; the use of an expressive programming logic to capture high-level security properties; and a toolset for managing, maintaining, and auditing the supporting evidence that is obtained from techniques such as code reviews, testing, and formal validation. The result is a powerful and new kind of development environment that integrates and builds on existing best practices to support the certification of high-assurance systems.

1 Vision

Programatica is a system for the development of high-confidence software systems and executable systems specifications. It is inspired by experience using the language Haskell [6] to model, refine, and implement computational systems. The goal of the Programatica environment is to support a coherent, but diverse set of methods for assuring confidence in Haskell programs.

As we have used Haskell, we have used several distinct modes to assure the fitness of an implementation or model:

1. We test program fragments on ad hoc test cases;
2. We test program fragments on random test cases derived from expected program properties;
3. We submit code to peer review;
4. We implement algorithms based on published papers;
5. We develop combinator libraries that allow us to program at a very high level (embedded domain-specific languages); the embedded DSL often more closely resembles a specification language than an implementation;
6. We reason about equational properties of functional programs;
7. We reason about meta-properties that derive from the type system;

8. We use monadic abstraction to encapsulate effects; and
9. We use Haskell to prototype formal definitions that we embed into other formal systems, such as HOLCF or SpecWare.

In different high-assurance domains we expect different standards of validation. In some applications, simply formulating the expected properties and supporting them with test cases will be a significant threshold of confidence. In others, static analysis, theorem proving and provably exhaustive testing may be required. Programatica is designed to accommodate all of these modes individually or in combination. Programatica provides an open interface for asserting properties and providing evidence for those properties. Evidence is bound into a program with a certificate, and certified properties can be combined in a simple logic. Certificates can be examined to identify the types of evidence that were used to establish the property, details of how the certificate was validated, the dependence of the certificate on the program or other certificates, and information about the user who validated the certificate.

Programatica is also intended to support an iterative development environment in which the program, its properties, and evidence are simultaneously developed and improved. We describe this as *Extreme Formal Methods*: we expect properties to be developed in parallel with the code, just as test cases are developed in extreme programming [1]. As the code matures (and the architecture stabilizes) the methods used for supporting the properties will be refined.

1.1 Why Haskell?

Haskell is at the core of Programatica. It is the language in which the computational models are expressed. What are the essential features of the computational modeling language?

- Haskell is a pure language that supports equational reasoning. One key premise of Programatica is that programs should behave "like math." It should be straightforward to calculate with programs. It should be possible to substitute values into program fragments with predictable results.
- Haskell has a very expressive type system that assists in guaranteeing program safety, and in supporting the natural expression of algorithms. It is one of the most expressive type systems in use.
- Haskell can naturally and precisely model virtually any side effect in a programming language. The type system of Haskell supports monadic abstraction. Monads let Haskell implement mathematical models of side effects and their propagation. Monads can be used to model mutable state, concurrency, backtracking, non-determinism, or any combination of these features. Monads are explained in more detail in Section 3.
- Haskell is an expressive, mature language in which experts can program effectively. While not the most popular language today, Haskell is arguably one of the most elegant. It grew out of the programming language research community and has been a testbed for advanced features in type systems, class based abstractions, and features supporting modular program development.

1.2 From Haskell to Programatica

To realize the Programatica vision of simultaneous development of a program and its properties we have developed a very expressive logic for Haskell, called P-logic. We have extended Haskell to support property definitions and assertions in P-logic. P-logic remains an active area of research in the project.

In this paper, we show how these pieces fit together with the Programatica toolset using a case study about a simple programmable crypto-chip. We demonstrate the use of Haskell as a modeling language to describe the functional behavior of the chip (Section 3); we show how key, high-level security properties—such as channel separation—are expressed as P-logic assertions (Section 2); and we explain how the Programatica toolset can be used to support the task of gathering and maintaining evidence (Section 4). For reasons of space, we will not be able to discuss all of the details in this development. We will aim instead to give an impression of the role that the different components play, and an insight into the potential that they offer together for high-assurance software development.

In a companion paper in this volume we present the status of the Oregon Separation Kernel (OSKer), a much larger Programatica artifact that models a multi-level secure operating system. In the supplementary materials we present details of the Programatica logic and a paper with a detailed proof of separation for an OSKer-like system.

2 Channel Separation for a Simple Crypto-Chip

The diagrams in Figure 1 provide an overview of the chip design that we study in this paper. The overall structure is loosely based on the General Dynamics AIM (Advanced Infosec Machine) crypto-chip. The chip processes packets of

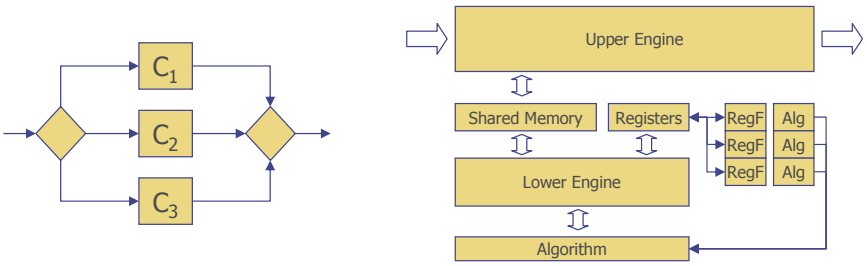

Fig. 1. The conceptual view (left) and concrete block structure (right) for a programmable crypto-chip.

data, each of which is tagged with a channel id. The conceptual view in Figure 1 suggests how these packets are multiplexed and demultiplexed, processing the packets on each channel independently of the packets on other channels. (To

keep the diagram simple, we show only three channels in these illustrations.) It is easy to model this view of the chip in Haskell as a function:

```
chip :: Algs -> ([Packet] -> [Packet])
```

The `Algs` parameter here specifies a mapping of algorithms (e.g., for encryption, filtering, etc.) to channels; this is the *programmable* component of the chip. The result is a packet filter—a function of type `[Packet] -> [Packet]`—that takes a list of packets as inputs and returns a processed list as output. Each such packet contains a channel identifier and a payload (i.e., the data to be processed):

```
type Packet = (ChannelId, Payload)
```

The block diagram in Figure 1 is much closer to the physical implementation of the chip on silicon. From this viewpoint, it is clear that a much greater portion of the hardware is actually shared between all of the channels, including two processing units (the upper and lower engines), a shared memory that connects them, and the register file and algorithm storage unit for the lower engine. Only one channel is active at any given time; the register file and algorithm for each of the other channels are (at least conceptually) swapped out to the banks of storage shown on the right of the block diagram.

The challenge now is to ensure that the greater level of sharing in the implementation does not compromise the logical separation of the channels in the conceptual model. More specifically, we need to check that there is no possibility for the data or algorithms associated with one channel to interfere with, monitor, or affect the data and algorithms on another channel. This high-level property of the chip can be formulated concisely and elegantly as a P-logic property:

```
assert Separation = All algs :: (ChannelId -> Alg).
                    All select :: (ChannelId -> Bool).
                     {filter (select . fst) . chip algs}
                      ===
                     {chip algs . filter (select . fst)}
```

The intuition here is that, if we install a filter on the output of the chip that removes all packets from some specified set of channels, then we get the same effect as if we had placed the filter instead on the input to the chip. This property can also be depicted graphically, as shown in Figure 2. In the P-logic formulation, the `All` keyword is a universal quantifier indicating that, for all possible mappings `algs` of algorithms to channels, and for all possible sets of channels (which we represented by a predicate `select`), the result of placing the filter on the output is the same as the result obtained when the filter is on the input. If this property holds, then we can be sure that the desired separation between channels is achieved:

– *No channel can produce outputs that depend on the inputs passed to other channels.* To see this, note that we could install a filter on the input to remove the packets for all but one channel. The `Separation` property guarantees that the chip will produce exactly the same output in this situation as it would if we had not installed the filter.

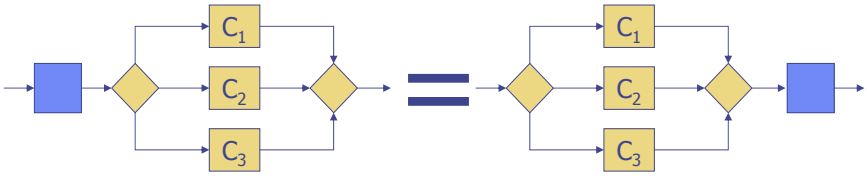

Fig. 2. A graphical representation of the high-level channel separation property. The square component (on the input of the left hand side and again on the output of the right hand side) is a filter that removes all packets associated with some (arbitrary) set of channels.

– *No channel can generate spurious outputs or generate output on another channel.* To see this, note that we could install a filter on the output to prevent such output, but the `Separation` property guarantees that the result will be the same as if no such filter were present.

There are many different ways that we might attempt to verify this property, including some combination of developer testing, team code-reviews, or formal methods such as model checking and theorem proving. Of course, in the process, we might instead uncover some bug or logical flaw in the design that causes the property to fail. In that situation, we would also expect the discovery to provide information suggesting how the problem might be fixed—perhaps adding extra logic, for example, to zero out a section of memory. Once this is done, we would need to go back and complete our attempt to verify the separation property, redoing those parts that are impacted by the change to ensure that no new problems have been introduced, but not wasting time or effort on places where the changes have no effect. These are exactly the kinds of tasks that the Programatica toolset (Section 4) is designed to support.

3 Modeling the Crypto-Chip in Haskell

In the previous section, we described the general structure of the chip, together with a critical `Separation` property. Importantly, we were able to do both of these things at a high-level—that is, in a way that is meaningful and useful to anyone who is interested in using the chip, even if they do not know anything about its implementation. To establish the `Separation` property, however, we must first model it in detail, reflecting the way that it is implemented in silicon.

At just under 260 lines of Haskell code, the model of the crypto-chip that we have built is quite concise, but still too long to include in full in this short paper. Instead, we will restrict the code in this paper to short but illustrative fragments. One important aspect of the model is its modular construction from six smaller components, as shown in Figure 3. Three of those components (accounting for just over 50% of the code) provide fairly general purpose libraries for common datatypes such as finite maps, state transformers, and an abstract datatype of

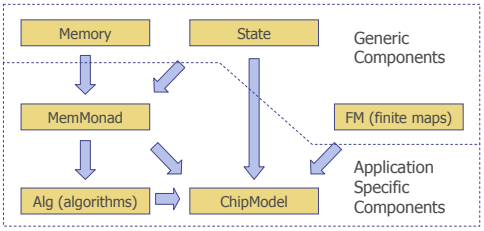

Fig. 3. Our formal model of the crypto-chip is captured in six Haskell modules.

memories (which the model uses for both the register files and shared memory components of the chip). Of course, like all Haskell programs, the model also makes use of standard functions and operators provided in the Haskell standard prelude. Any effort that we invest in documenting and validating the behavior of these parts of the system, will, like the code itself, be a resource that we can reuse and build on in other projects.

The remaining three components (`MemMonad`, `Alg`, and `ChipModel`) are more application specific, tailored to the particular needs of modeling the crypto-chip. The top level entry point is the function `chip` mentioned previously:

```
chip :: Algs -> [Packet] -> [Packet]
chip algs
      = catMaybes . loop (onePacket algs) (initMem, initRegs)
```

In fact the bulk of the work is done by a function `onePacket`, which describes the steps involved in processing a single input packet:

```
onePacket :: Algs -> Packet -> State (Mem, Regs) (Maybe Packet)
onePacket algs (chan, ws)
 = do regs <- inSnd readState
      rng  <- inFst (malloc ws)
      let alg     = algs `at` chan
          regfile = regs `at` chan
          valid   = includes rng
          code    = runAlg (alg (fst rng) regfile)
      res  <- inFst (runProtected valid code)
      case res of
        Nothing       -> return Nothing
        Just regfile' -> let regs' = extend chan regfile' regs
                         in do inSnd (setState regs')
                               packet <- inFst (readPacket rng)
                               return (Just (chan, packet))
```

This code makes essential use of *state monads* to structure and control the use of side effects. In fact, the support that Haskell provides for programming with

monads is one of the features that makes it particularly attractive for use in security related applications. The term 'monad' carries little insight, and newcomers to Haskell are sometimes intimidated by the origins of monadic programming in abstract mathematics. But, in fact, monads can be understood as an example of something that is very familiar to programmers: an abstract datatype (ADT).

Programmers have long understood the benefits of abstract datatypes (ADTs) for traditional data structures such as stacks and queues. A typical ADT provides a well-defined interface while also hiding details its implementation. In particular, in a programming language that supports and enforces the abstraction of an ADT, the implementor can be sure that internal data structures will not be visible and hence cannot be corrupted or misused by clients of their ADT.

From this perspective, monads are just ADTs for describing computations. The `chip` and `onePacket` functions use a monad called `State (Mem, Regs)` to describe stateful computations that can access and modify the global memory of the chip (`Mem`), the bank of saved register files (`Regs`), ... *and nothing else.* Inside the definition of `onePacket`, we see computations in which access to the state of the chip is even more restricted. For example, in a statement of the form `inFst c`, the command `c` has access only to the first (`Mem`) component of the global state, while, in a statement of the form `inSnd c`, the command `c` has access only to the second (`Regs`) component. Further restrictions on a command's ability to access and modify the state are imposed in the command `runProtected valid code`, which aborts the execution of `code` if it makes any attempt to access read from or write to any location that is not accepted by the `valid` predicate. For example, in the code for `onePacket`, the command `inFst (malloc ws)` is used to allocate space in shared memory for the data portion `ws` of the input packet. The range `rng` of addresses in which that packet is stored is then used to restrict access to those locations alone in shared memory when the `code` associated with that channel is executed.

These examples only begin to demonstrate the facility that monads provide for encapsulating and controlling the impact of side-effects. In the particular case of restricted access to memory, the monads that we have used provide a more fine-grained and robust level of control than the hardware based memory protection mechanisms that are available on some hardware platforms. The latter is typically provided to programmers via an operating system API, that is not itself part of the programming language, and relies on careful coding to maintain some distinction between kernel- and user-mode code. Furthermore, a language like Haskell allows programmers to define and use new monads, possibly customized to suit the needs of a particular application, and with the potential to control a much wider range of computational effects than just state, including exception handling, I/O, read-only memory, concurrency, and so on.

4 Programatica Tools for Managing Evidence

Researchers in industry and academia have attacked the challenges of high assurance software development in many ways, demonstrating concretely how the use

of systematic design processes, rigorous testing, or formal methods can each contribute significantly to increased reliability, security, and trustworthiness. There are obviously some significant differences between these techniques, but there is also a unifying feature: each one results in some tangible form of *evidence* that provides a basis for trust. Examples of such evidence might include a record of the meeting in which a code review was conducted, the set of test cases to which the code was subjected, or a formal proof that establishes the validity of a critical system property. In practice, however, it can be difficult to manage and maintain this collection of evidence as the project that it refers to continues to grow and evolve. To do this effectively, for example, we must track dependencies between code and evidence, identifying situations where a change in the program might invalidate, and so trigger a review of some portions of the evidence.

The Programatica toolset provides a program development environment for Haskell and includes features to support understanding, construction, and maintenance of software. Critically, however, the Programatica tools also facilitate and support effective use of evidence throughout a project's life. In particular, these tools support high assurance software development with Haskell, allowing users to integrate program code with statements of key system properties in P-logic; to capture and collate a wide range of evidence with source materials; to track dependencies and maintain consistency; to automate the construction, combination, and reuse of evidence; and to understand, manage, and guide further development and validation efforts. Programatica supports this wide range of activities by adopting a *certificate* abstraction as a mechanism for encapsulating evidence. This results in a flexible and extensible architecture in the style of some component-based programming systems that allows many different types of evidence to be accessed and manipulated using the same generic interface.

The screenshot in Figure 4, for example, shows the PFE (Programatica front-end) browser being used to explore evidence of the `Separation` property of the `ChipModel` example from the previous section. The browser has many of the features of current integrated development environments. On the left, for example, a tree-based project navigator allows the user to move between different sections of the model. On the right, the source code for the `ChipModel` is displayed with the benefits of syntax-coloring, hyperlinks, and mouse-overs that provide extra information to facilitate browsing of code, properties, and certificates. Notice that the source code includes both definitions of executable code like `chip` and assertions of properties like `Separation`. The icon that appears next to the statement of `Separation` in Figure 4 indicates that a certificate has been provided for this particular property; further details about the certificate are provided in the pop-up window, which is brought up by clicking on the icon. The toolset provides many other important features not shown here, including facilities for creating and editing different types of certificate, for maintaining and inspecting dependency information, and for reporting on project status.

Each different type of certificate has an associated *server* component that acts as a plug-in to the Programatica toolset. In particular, servers are responsible for translating between the Haskell and P-logic notation used in source

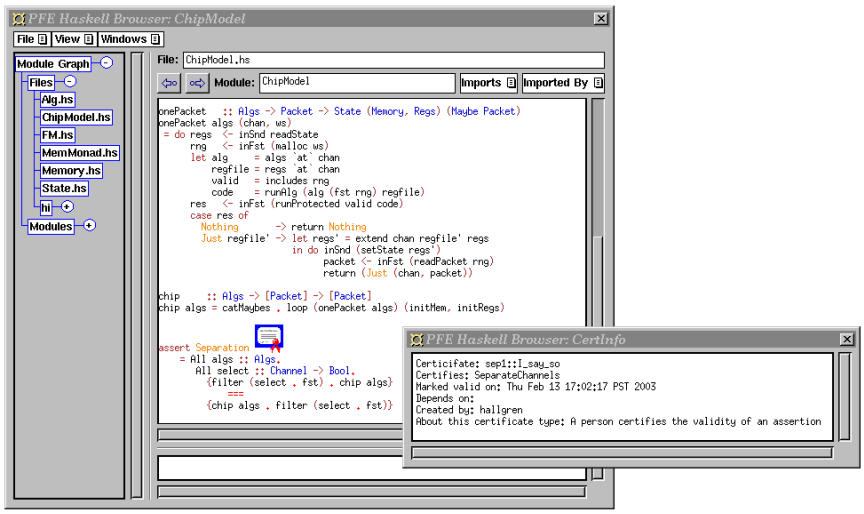

Fig. 4. Using the PFE browser to examine evidence for the `Separation` property.

documents and the syntax of an external validation tool such as a test-case generator, model checker, or theorem prover. Servers are also responsible for capturing and packaging context from source documents so that it can be used by an external tool. In the case of an external theorem prover, for example, we refer to this as 'theory formation' because it requires assembling a theory that includes any facts and definitions that are needed to complete the proof of a particular theorem. To date, we have built servers that deal with several kinds of evidence, including

- Information provided by textbooks, component specifications, and other resources. We describe this as "*I say so*" evidence because the trust that we place in it is ultimately determined by the trust that we place in the source. Certificates of this kind may, for example, include a pointer to a technical paper where a particular property is established, or perhaps include documents that record the outcome of a code review.
- Individual test cases. A certificate of this type includes input data for a specific test case scenario together with the output that is expected. When the program is modified, the test case can be executed automatically and the results compared with the expected output to check that existing functionality has not been broken by the changes. This is a key technique in extreme programming, but shows up here as a special case of certificate management.
- Random testing. This server translates the portions of the main program that are needed in the definition of a particular property to a script for use with the QuickCheck library [2]. Here, the benefits that we obtain by testing with many different inputs must be weighed against the quality/relevance of the random test cases that are generated by QuickCheck.

– Formal proof. This server translates Haskell programs into the notation of the Alfa proof editor [4], which allows the construction of formal proofs in a logical framework based on constructive type theory.

These examples cover a fairly wide range of evidence types, and so demonstrate the generality of the Programatica certificate abstraction. In current work, we are also investigating the possibility of adding additional server types to interface with other external tools including a free-theorem generator [7], a bounded model-checker, and the Isabelle theorem prover [5]. It is also our intention to provide documentation that will enable independent users of Programatica to develop their own servers and so interface to other external validation tools.

5 The Future of High-Assurance Software Development

International initiatives such as the Common Criteria [3] provide important standards for evaluation and assurance of software supporting IT security by building on current best practices in industry, government, and academia. The Common Criteria also recognizes the potential for formally verified design and testing in critical applications where the highest levels of assurance are required (i.e., at EAL5 to EAL7), but its evaluation methodology does not yet extend to these cases.

Programatica offers a powerful new vision for the future of high-assurance software development. Its design extends current evaluation methodologies, readily supporting and integrating the different kinds of evidence that they require. Moreover, Programatica offers an evolution path for introducing and applying formal methods to document and validate essential functional properties of critical software and hardware systems at the highest assurance levels.

References

1. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
2. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000.
3. Common Criteria: The Standard for Information Security. `http://www.commoncriteria.org/`.
4. Thomas Hallgren et al. The Alfa proof editor. `http://www.cs.chalmers.se/~hallgren/Alfa/`.
5. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
6. Simon Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, 1999. Available from `http://www.haskell.org/definition/`.
7. Philip Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.