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

Mark P. Jones, James Hook, Thomas Hallgren
OGI School of Science & Engineering at OHSU
Beaverton, Oregon
Flashback to HCSS in 2001:

- We had assembled a team ... 

- ... but the Programatica Project had not officially started ...

- I presented our vision of what Programatica might become ...
The Programatica Vision:

- Build a program development environment that supports and encourages its users in thinking about, stating, and validating key properties.
- Enable programming and validation to proceed hand in hand, using properties to link the two.
- Allow users to realize benefits gradually by choosing between varying levels of assurance.
Type checking

Execute & test

Random test generator

Instrumenting compiler

Interactive proof editor

Model checker

Theorem proving

Haskell Programs + Properties + Certificates

User supplied, domain-specific toolsets...

Vision: Integrating Evidence from Multiple Sources
Back then: Mockups

```haskell
-- reading from memory is just function application:
read :: Memory -> Addr -> Word
read mem addr = mem addr

property ReadEmpty = All (a::Addr).
                    read empty a = a

-- writing to memory is function extension:
write :: Memory -> Addr -> Word -> Memory
write mem addr val = 
                    if addr==a
                      then mem addr
                      else mem addr

property ReadWrite = All (a::Addr).
                    All (w::Word).
                    All (m::Memory).
                    read (write m a w) a == w

property WriteWrite = All (a::Addr).
                     All (w,w'::Word).
                     All (m::Memory).
                     write (write m a w') a w = write m a w
```
Today: Real, Working Tools
In it’s time, Programatica was the most sophisticated program development environment on the market;

“It scares me to think that we nearly ended up in a world dominated by Java technology ... Programatica was a godsend; we couldn’t have made the transition to Haskell without it ...”

James Gosling, Microsoft CEO, eComdex 2007
Today: The view from 2003

We’re on track to have a public release of the tools early in the summer ...

We’re preparing materials for a short course on the Programatica approach to software development, and on the toolset, to coincide with the release ...
Building High-assurance Software:

There are many ways to increase assurance:

- Test programs on specific cases
- Test programs on randomly generated test cases derived from expected properties
- Peer review
- Use algorithms from published papers
- Reason about equational properties
- Reason about meta-properties (e.g., using types)
- Use theorem provers to validate (translated) code
- ...

Each one can contribute significantly to increased reliability, security, and trustworthiness
Evidence: A Unifying Feature

There are significant differences in the applicability, assurance, and technical details of each of these techniques.

But there is a common feature:

- Each one results in some tangible form of evidence that provides a basis for trust.
Examples of Evidence:

There are many kinds of evidence:
- An (input, expected output) pair for a test case
- A property statement, and heuristics for guiding the selection of “interesting” random test cases
- A record of a code review meeting
- A citation/URL for a published paper or result
- An equational proof
- A type and the associated derived property
- A translation of the source program into a suitable theory and a user-specified proof tactic
- ...

In Programatica, each different kind of evidence is stored with the program as a certificate
Evidence and Certificates:

The certificate abstraction is designed to support:

- **Capture** of evidence of validity (in many different forms) and **Collation** with source materials
- **Combination** of evidence
- **Tracking** dependencies and **detecting** when evidence needs to be revalidated because of changes in the source code
- **Management** of evidence by analyzing and reporting on what has been established, identifying weaknesses, guiding further effort, etc...
Programatica Components:

- A semantically rich, formal modeling language (Haskell)
- An expressive programming logic that can be used to capture critical program properties (P-logic)
- A toolset for creating, maintaining, and auditing the supporting evidence (pfe,cert,...)
Example: Modeling a Crypto-Chip

An example based (very loosely) on the General Dynamics AIM crypto-chip

Conceptual view:

One chip, multiple channels
Channels may use different algorithms
GUARANTEED separation between channels
High-level Model:

\[ \text{chip} :: \text{Algs} \rightarrow ([\text{Packet}] \rightarrow [\text{Packet}]) \]

\[ \text{type Packet} = (\text{ChannelId}, \text{Payload}) \]
The Separation Property:

assert Separation =
All algs :: Algs.
All select :: (ChannelId → Bool).
{ filter (select . fst) . chip algs } ===
{ chip algs . filter (select . fst) }
The Separation Property:

This law guarantees that:

- Outputs do not depend on inputs to other channels.
- Channels do not generate spurious outputs.
Putting Programatica to Work:

- Our goal is to build tools that will help to establish and automate validation of properties like this.

- We have described the non-interference property at a high-level.

- But we want to model the chip at a level that is closer to its implementation on silicon.
Basic architecture:

- **Upper Engine**
- **Shared Memory**
- **Lower Engine**
- **Algorithm**
- **Registers**
  - RegF
  - Alg
Basic architecture:

- Receive packets, save in shared memory.

Diagram:

- Upper Engine
- Shared Memory
- Registers
- Lower Engine
- Algorithm
- RegF
- Alg
Basic architecture:

Load saved registers & algorithm for channel.
Invoke lower engine to process packet.
Basic architecture:

Upper Engine

Shared Memory  Registers

Lower Engine

Algorithm

Save register set, if lower engine completes successfully.
Basic architecture:

Zero out shared register set.

Diagram:
- Upper Engine
- Shared Memory
- Registers
- Lower Engine
- Algorithm
- RegF
- Alg
- RegF
- Alg
- RegF
- Alg
- RegF
- Alg
Basic architecture:

- Pass processed packet data to output.

Diagram:
- Upper Engine
  - Shared Memory
  - Registers
  - Lower Engine
  - Algorithm
  - RegF
  - Alg

Building the Model:

We developed an executable model of the chip as a Haskell program: (260 LOC)
Execution (upper engine):

Processing of a single packet in the upper engine is described by a function:

\[
\text{onePacket} :: \\
\text{Algs} \rightarrow \\
\text{Packet} \rightarrow \text{State (Memory, Regs)} \\
\quad \text{(Maybe Packet)}
\]

Processing of packet streams:

\[
\text{chip} :: \text{Algs} \rightarrow [\text{Packet}] \rightarrow [\text{Packet}] \\
\text{chip algs} = \text{catMaybes} \cdot \\
\quad \text{loop (onePacket algs)} \\
\quad \text{(initMem, initRegs)}
\]
Execution (upper engine):

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))
Why Haskell?

One reason: no hidden side-effects

- **Purity**: if \( f \) is a function of type \( A \rightarrow B \), the result of \( f \, x \) will depend only on \( x \)

- **Monads**: using abstract datatypes to encapsulate and control the scope of effects explicitly:

  \[
  \text{inFst (runProtected valid code)}
  \]

  Language semantics enforces protection, without lower level OS/API wrapper.

See Peter White’s talk for more ...
Programmatica Haskell Browser: Memory

- Standard menus, navigation, browsing, options & certificate management actions are here
- Each of the files in the model appears here
- Syntax colored source text appears here
- Context sensitive messages show up here
The Programatica Front End:

- The GUI, pfebrowser, usually provides the most convenient interface for working with Programatica.

- A command line tool, pfe, is also available.

- Both are useful tools in their own right for Haskell programmers.

Usage: pfe [options] <command>

where <command> is one of:
- new <files> -- create a new project containing <files>
- add <files> -- add files to the project
- remove <files> -- remove files from the project
- files -- list files in the project
- options -- show options in effect
- modules <modules> -- list modules in the project
- graph <modules> -- show module dependency (sub)graph
- dotgraph <modules> -- dot format module dependency graph
- revgraph <modules> -- show reverse module dependency (sub)graph
- unused modules -- show unimported and unreachable modules
- prune modules -- remove unreachable modules from the project
- file <modules> -- which file is the module in
- module <files> -- which module does the file contain
- defined modules -- list entities defined in the module
- free modules -- list names referenced but not defined in the module
- pragmas modules -- extract pragmas from modules
- lex <files> -- show the result of lexical analysis
- exports modules -- list entities exported by the modules
- find <identifiers> -- find exported entities with the given names
- inscope modules -- list entities in modules' top-level environment
- pe modules -- parse and pretty-print modules
- tc modules -- type check and display decorated modules
- tcpb modules -- remove pattern bindings, then tc
- tclc modules -- remove list comprehensions, then tc
- types modules -- show types/kinds of top-level entities
- instances modules -- list instances defined in a module
- iface modules -- show the interfaces of modules

and more options related to dependency analysis, pretty-printing, etc.
A Development Environment:

- Standard Haskell compilers and interpreters are used to compile and execute code.
- pfebrowser provides sophisticated browsing capabilities with hyperlinking, integrated type checking, ...

- Programatica is a program development environment.
Using Properties:

We annotated the model with properties ...

... and quickly spotted bugs in our code!
Programatica: “Programming as if Properties Matter”

Properties are
- written
- parsed
- analyzed
- type-checked
as an integral part of the source text

Goals:
- Maintain consistency between code and properties
- Capture programmer expectations/intentions as part of the programming process
- Just writing down properties heightens thinking about correctness
Extreme Programming

Testing and Programming proceed hand in hand

- Testing reveals errors in the program
- Programming reveals errors in the test cases
“Extreme Formal Methods”

Programming and Validation proceed hand in hand

- Validation reveals errors in the program
- Programming reveals errors in the specification
Generating Evidence:

We began the process of validation, using QuickCheck to generate random test cases for asserted properties...

... and found bugs in our specification!...
... and bugs in our code!
Gathering Evidence:

Source materials are stored with related evidence and dependency information.

A “hidden information” directory is shared between the files in a package.
Using QuickCheck:

QuickCheck is an independently developed random testing tool (Hughes and Claessen, Chalmers University, Sweden)

Haskell developer’s perspective:

Haskell program + property annotations → QuickCheck Library → Executable Code

Passed n tests; or Failed with counterexample

rng
Using QuickCheck with pfe:

Programatica implementer's perspective:

Programatica source → Slicing → QuickCheck Library → Haskell program + property annotations → Executable Code → rng

(Slicing is a reusable transformation that reduces the size of the code that is passed to QuickCheck, and eliminates spurious dependencies)

Passed n tests; or Failed with counterexample
Using QuickCheck with pfe:

Programatica user’s perspective:

- **Programatica source**
- **The QuickCheck Server**
  - Passed n tests; or
  - Failed with counterexample
Servers and Certificates:

- Use of a **registry** enables a flexible, extensible system.

- Use of **servers** and **certificates** permits a generic interface that automates/hides the translation between Programatica and any external tools.
The Registry:

Once again we are exploiting the existing filesystem in a design for the registry that is extensible, language neutral, and portable:
Servers in pfe:

Current implementation includes:

- “Paper and Pencil” (I say so!)
- QuickCheck
- Alfa (a proof assistant based on constructive type theory)

Others currently in progress/under consideration include:

- Free theorem generator
- Regression testing
- Isabelle (HOL theorem prover)
- Bounded model checker
Certificates in pfebrowswer:
Stronger Evidence:

We began to construct a formal (hand) proof of Separation ... 

The overall structure is modular:

- Properties of Memory
- Properties of State
- Properties of FM
- Properties of MemMonad
- Properties of onePacket
- Separation for the chip
Combining Evidence:

We began to construct a formal (hand) proof of Separation ...

The overall structure is modular:
Validation and Combination:

We want to validate and combine evidence from different sources:

- Certificates carry **sequents** “Assume $\vdash$ Conclude” that act as an interface/contract between Programatica and any external tools.

- Servers for external tools are used to test **validity** (i.e., to check that a certificate’s sequent is consistent with its evidence)

- Built-in servers use sequents of existing certificates to guide the construction of new, composite certificates.
Integrating Evidence:

Multiple tools can play a role in validating or assuring behavior of the system as a whole:
Separation Fails!: We uncovered two bugs in our attempt to prove separation:

- Separation fails if an algorithm can fail to terminate

![Diagram showing separation failure](image)

- Separation fails because the algorithm for a channel sees the absolute address of packets in shared memory.
  - Is this a bug in the code or the specification?
  - Is this a security loophole?
  - Several fixes are available: relative addressing, zeroing out memory, etc...

This is useful feedback for the designer/developer to discuss!
Dealing with Change:

- Our model, our specification, or both must be revised to complete the task in hand.
- Whatever happens, some of the evidence we have collected may no longer be valid.
- Some evidence can be reconstructed automatically, but some will be quite expensive to reconstruct.
- In software development, change is the norm, not the exception, so we need to handle change as efficiently as possible.
Hashing to Detect Change:

- When we parse a source file, we calculate a cryptographically robust hash (e.g., MD5) over the abstract syntax of each definition.

- These hashes are cached as hidden information:
  - 0cc175b9c0f1b6a831c399e269772661
  - 92eb5fffee6ae2fec3ad71c777531578f
  - 81a5fe3d544359af13848e6192ece475
  - 445a4ca24e10824e03ef42e2e1d755d9
  - 987dd8f5f1293857dc7932c14c7f3d80
  - 8b3ee2a3933b9c01878bcddc298ff9e2
  - bb53046df3ef7793ee7c37aec0d090d0
  - ad797e6f29cf558f7aeb8200563ecd3a
  - 8959f36e873441e58dcc9222777b6d47
  - 84de7ff93b201e8c5b4cf0e006dfe848
  - 7a5acfc765e1875a49daffd8561ae025

- If we find a definition whose hash is not listed, then it must be new/modified.
Using a Dependency Graph:

Properties
Definitions
Primitives
Using a Dependency Graph:

- `<m>` to `<l>` to `<k`
- `<f>`
- `<c` to `<d`
- `<a` to `<j`
- `<b` to `<c`
- `<a` to `<e`
- `<a` to `<i`
- `<e` to `<i`
- `<i` to `<g`

Legend:
- Yellow: Properties
- Blue: Definitions
- Gray: Primitives

New Definition!
Using a Dependency Graph:

Potential change

Properties
Definitions
Primitives
Benefits of Hashing:

- Fine-grained dependency analysis reduces the cost of reconstructing evidence after the program has been modified.

- By hashing over abstract syntax, we do not flag any changes if the source text is reformatted, if comments are changed, etc...
Management Activities:

Evidence management tools let users ask (and answer) questions like the following:

- What properties have I verified (or not)?
- What tools did I use?
- Is the evidence up to date & consistent with the code?
- What other verification strategies should I pursue?
- Where am I most vulnerable?
- What should I do next?

Scoring & prioritization mechanisms required
Summary:

Solid foundations:
- Precise, formal semantics for Haskell
- A sound & expressive programming logic, P-logic

Extensible tools:
- A flexible infrastructure for certification
- A small but growing collection of servers

A vision for high-assurance development:
- Extends & integrates current methodologies
- An evolution path for applying formal methods