Type Theory in the Software Analysis Workbench

Aaron Tomb
Galois, Inc.

Type Theory Based Tools
2017-01-15

© 2017 Galois, Inc. All Rights Reserved.

- Describe an existing system: the Software Analysis Workbench (SAW)
    - Tool for constructing functional models of imperative programs
    - Shared intermediate language nominally based on type theory
    - Heavy use of automated reasoning
- Solicit input on future directions
    - Type theory has much more promise
    - How best to use it?

© 2017 Galois, Inc. All Rights Reserved.

galois

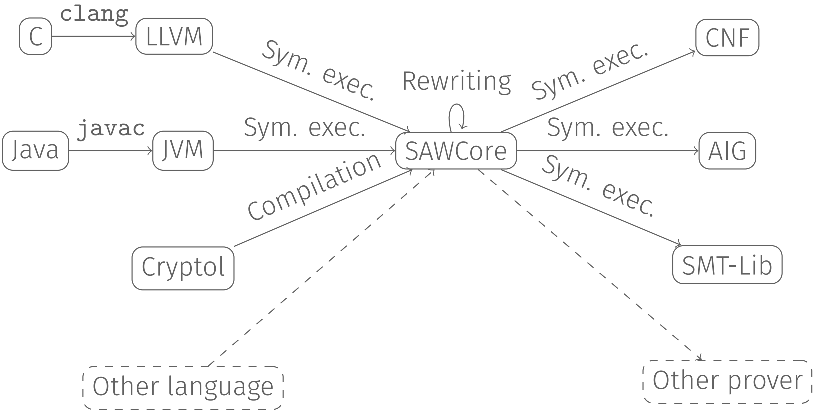
What is SAW?

- SAW = Software Analysis Workbench
  - Software: many languages
  - Analysis: many types of analysis, focused on functionality
  - Workbench: flexible interface, supporting many goals
- Intended as a flexible tool for software analysis
- What separates it from other systems?
  - One view: compiler :: imperative code $\longrightarrow$ functional code
  - Captures all functional behavior, simplifying later if necessary
  - Uses efficient internal representations tuned to equivalence checking
  - Strong bit vector reasoning support
  - Focus on practicality over novelty
- Open source (BSD3) and available now

© 2017 Galois, Inc. All Rights Reserved.

galois

What's Behind This?



© 2017 Galois, Inc. All Rights Reserved.

|galois|

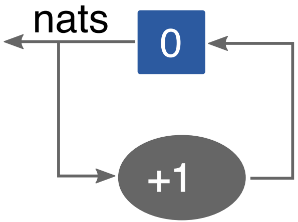
A single, high-level specification for (cryptographic) algorithms

- Cryptol goals
  - Appropriate for cryptography
  - Natural
  - Concise
  - Similar to existing notation
  - Appropriate for execution and verification
- Language features
  - Statically-typed functional language
  - Sized bit vectors (type level naturals)
  - Stream comprehensions (stream diagrams)

© 2017 Galois, Inc. All Rights Reserved.

- Functions and sequences are key notions
- Both can be recursive
- To compute the sequence of all natural numbers

```
nats = [0] # [ n + 1 | n <- nats ]
```



© 2017 Galois, Inc. All Rights Reserved.

Relationship Between Cryptol and SAW

- Cryptol is essentially the expression language of SAWScript
- Built-in support for Cryptol syntax
  - Translated automatically into `Term` objects with `{{ ... }}`
- Emerged as an evolution of Cryptol REPL commands
  - Generalizes more constrained `:prove` and `:sat`
  - More complete language
  - Beyond automated proofs
- Supports proofs purely on Cryptol
- Allows proofs comparing Cryptol to real-world implementations

© 2017 Galois, Inc. All Rights Reserved.

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
```

© 2017 Galois, Inc. All Rights Reserved.

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
Time:     3.433s
Valid
```

© 2017 Galois, Inc. All Rights Reserved.

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
Time:     3.433s
Valid
sawscript> time (prove z3  {{ p }})
```

© 2017 Galois, Inc. All Rights Reserved.

galois

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
Time:     3.433s
Valid
sawscript> time (prove z3  {{ p }})
Time:     0.006s
Valid
```

© 2017 Galois, Inc. All Rights Reserved.

galois

```
import "DES.cry";
let {{ enc = DES.encrypt }};
let {{ dec = DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

© 2017 Galois, Inc. All Rights Reserved.

```
import "DES.cry";
let {{ enc = DES.encrypt }};
let {{ dec = DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

© 2017 Galois, Inc. All Rights Reserved.

galois

```
import "DES.cry";
let {{ enc = DES.encrypt }};
let {{ dec = DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

```
Valid
Time:     4.694s
Valid
Time:     4.718s
Valid
Time:     0.003s
```

© 2017 Galois, Inc. All Rights Reserved.

galois

More Complex Proof: 3DES

```
import "DES.cry";
let {{ enc = DES.encrypt }};
let {{ dec = DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

```
Valid
Time:     4.694s
Valid
Time:     4.718s
Valid
Time:     0.003s
```

© 2017 Galois, Inc. All Rights Reserved.

galois

```
static int ffs_ref(int word) {
    if(word == 0) return 0;
    for(int cnt = 0, i = 0; cnt < 32; cnt++)
        if(((1 << i++) & word) != 0) return i;
    return 0;
}
static int ffs_imp(int i) {
    byte n = 1;
    if ((i & 0xffff) == 0) { n += 16; i >>= 16; }
    if ((i & 0x00ff) == 0) { n +=  8; i >>=  8; }
    if ((i & 0x000f) == 0) { n +=  4; i >>=  4; }
    if ((i & 0x0003) == 0) { n +=  2; i >>=  2; }
    if (i != 0) { return (n+((i+1) & 0x01)); } else { return 0; }
}
```

```
ffs_cls <- java_load_class "FFS";
ffs_ref <- java_extract ffs_cls "ffs_ref" java_pure;
ffs_imp <- java_extract ffs_cls "ffs_imp" java_pure;
prove abc {{ \x -> ffs_ref x == ffs_imp x }}; // Valid: 0.014s
```

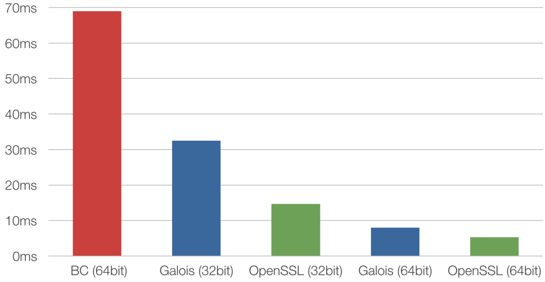
© 2017 Galois, Inc. All Rights Reserved.

galois

- Proved correctness of many implementations using SAW
- Proof is automated but slow: 5m – 2.5h
- Script to prove OpenSSL C implementation in place
  - Likely to be merged into official OpenSSL source tree
  - AES-128 and AES-256, encryption and decryption
  - Slowest equivalence checking (at least 1h for each proof)
- ~ 1300 C LOC
- ~ 230 spec LOC
- ~ 5 script lines per proof (all plumbing)

© 2017 Galois, Inc. All Rights Reserved.

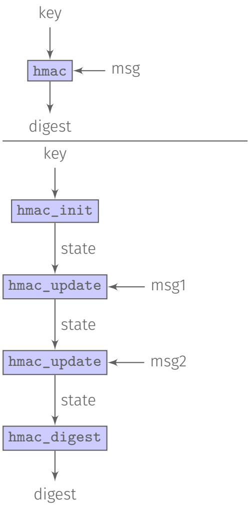
- Elliptic Curve Digital Signature Algorithm (ECDSA)
- In-house Java code, tuned for speed and verifiability
  - Available with SAW distribution
- ~ 2400 Java LOC
- ~ 1600 spec LOC
- ~ 1500 proof script LOC (largely plumbing)
- Proof completes in < 5m



galois

© 2017 Galois, Inc. All Rights Reserved.

- Amazon TLS implementation
- Code from official `s2n` repository
- ~ 15 (top-level) spec LOC (monolithic function)
- ~ 300 C LOC (iterative code)
- ~ 400 script LOC (all plumbing)
- Proofs for various fixed message sizes
  - <1m per proof



© 2017 Galois, Inc. All Rights Reserved.

|galois|

Constructing Models with Symbolic Execution

- Imperative $\longrightarrow$ functional via symbolic execution
- For straight-line code, symbolic value of any variable at end is a pure function of symbolic inputs
- Model memory ephemerally
- For branches, merge symbolic states at post-dominators
  - A nested application of the if-then-else function
- Unroll loops
  - So they're just a case of sequential branching
  - Can terminate more frequently by SAT-checking branch conditions
- Have also experimented with using fixpoint combinator

© 2017 Galois, Inc. All Rights Reserved.

|galois|

The SAWCore Language

- Dependently typed core calculus
- Takes some inspiration from CiC, some from MLTT
  - More on specifics, future later
- Represented efficiently with hash-consed DAGs
- Large number of primitives
  - Covering, e.g., the SMT `QF_AUFBV` theory
  - Even though these can be (and have been!) defined in SAWCore, too
- Two type checkers
  - One from surface syntax to explicitly type terms
  - One on explicitly typed terms (incomplete)
  - No guarantee that they agree!

© 2017 Galois, Inc. All Rights Reserved.

- As a type theory, two notions of proof in SAWCore
  - Showing inhabitant of equality type
  - Showing a Boolean term equivalent to `True`
- Proofs can be performed by SAT and SMT solvers
  - Several tactics for transformation in advance
  - Solvers use classical logic!
- Hand-constructed proof objects are more powerful
  - But no tactics at this level
- Terms of type `Eq a b` are theorems
- Terms of structure `a == b` can be theorems
  - If shown valid by external prover

© 2017 Galois, Inc. All Rights Reserved.

galois

Rewriting in SAWCore

- Rewriting the main proof tactic available
- Both `Eq a b` and `a == b` can be used as rules
  - The latter normally proved before use, but optional
  - Function definitions are collection of rewrite rules
- Symbolic execution can be thought of as an instance of rewriting
- Some limitations:
  - No conditional rewriting (so far)
  - No auto-simplification for associativity, commutativity, etc.
- Other interactive provers are more flexible
  - Though in some cases less efficient (we routinely process multi-GB terms)
  - And not as integrated with automated provers or model extractors

© 2017 Galois, Inc. All Rights Reserved.

|galois|

Open Question: Semantics of SAWCore

- Currently: a somewhat unsound, ad-hoc bag of features
- Ideally: choose an existing, well-studied core calculus and implement it faithfully
- Maybe adapt to semantics of Lean?
  - Lean could be directly linked in
  - Haskell bindings to core API already exist
  - Core language is simple
  - Any interactive proof could use Lean tactics
- Coq export for definitions would also be valuable
  - Proofs would probably be prover-specific, though

© 2017 Galois, Inc. All Rights Reserved.

Open Question: Representing Non-Termination

- SAW's main goal: representing program semantics
- Many real programs don't terminate
  - Or at least are hard to prove to be terminating
- What's the most effective way to represent them?
- Various possibilities, none ideal
  - Distinguish between type and non-type terms at a sort level, a la Zombie (complex)
  - Use co-inductive reasoning (but induction is more straightforward when possible)
  - Use deep embeddings with a flexible interpreter (slow!)
  - Require variants (simple, but more user burden)

© 2017 Galois, Inc. All Rights Reserved.

galois

Open Question: Interactive Proof in SAW

- Some interactive proofs already possible
- Mostly: unconditional rewriting followed by automated tools
- Limited to a single proof goal
  - So case splitting is out
  - Induction even farther away
- Considering the possibility for multiple goals
- Also considering integration with existing interactive provers
  - Lean is a prime candidate

© 2017 Galois, Inc. All Rights Reserved.

Open Question: Proofs About Complex Memory Models

- Currently, memory “erased” from denotations
- Very efficient and powerful when it works
- Limits the class of programs we can handle
- Explicit memory objects in denotations would help
- How to best represent them?
  - SMT array theory probably too impoverished
  - Maybe a different “primitive” type?
  - Something encoded directly in the logic?

© 2017 Galois, Inc. All Rights Reserved.

galois

SAW: efficient proofs about imperative programs via translation to functional programs + SAT/SMT

- Practical system, used to verify real-world code, such as:
  - AES from OpenSSL
  - HMAC, DRBG from s2n
  - ECDSA from Galois
  - Portions of several Curve25519 implementations
- Use of types gives structuring principles, helps detect mistakes
- Type theory provides power and flexibility
  - and an explicit form okay, since terms are automatically constructed
- But what possibilities have we yet to take advantage of?

© 2017 Galois, Inc. All Rights Reserved.

galois

Aaron Tomb, Adam Foltzer, Adam Wick, Andrey Chudnov, Andy Gill, Benjamin Barenblat, Ben Jones, Brian Huffman, Brian Ledger, David Lazar, Dylan McNamee, Edward Yang, Fergus Henderson, Iavor Diatchki, Jeff Lewis, Jim Teisher, Joe Hendrix, Joe Hurd, Joe Kiniry, Joel Stanley, John Launchbury, John Matthews, Jonathan Daugherty, Kenneth Foner, Kyle Carter, Ledah Casburn, Lee Pike, Levent Erkök, Magnus Carlsson, Mark Shields, Mark Tullsen, Matt Sottile, Nathan Collins, Philip Weaver, Robert Dockins, Sally Browning, Sam Anklesaria, Sigbjørn Finne, Thomas Nordin, Trevor Elliott, Tristan Ravitch

© 2017 Galois, Inc. All Rights Reserved.

Resources

- Cryptol
  - Web: `http://cryptol.net`
  - GitHub: `https://github.com/GaloisInc/cryptol`
- Software Analysis Workbench
  - Web: `https://saw.galois.com`
  - GitHub: `https://github.com/GaloisInc/saw-script`
- HMAC verification blog post:
  - `https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/`

© 2017 Galois, Inc. All Rights Reserved.