galois

Developing Tools for Formal Specification and Verification

Ryan Scott
AmeriHac

February 2026

galois

Developing Tools for Formal Specification and Verification *(with Haskell)*

Ryan Scott
AmeriHac
February 2026

Way-too brief history of Haskell use at Galois

Way-too brief history of Haskell use at Galois

- We have been using Haskell since company’s founding (1999).

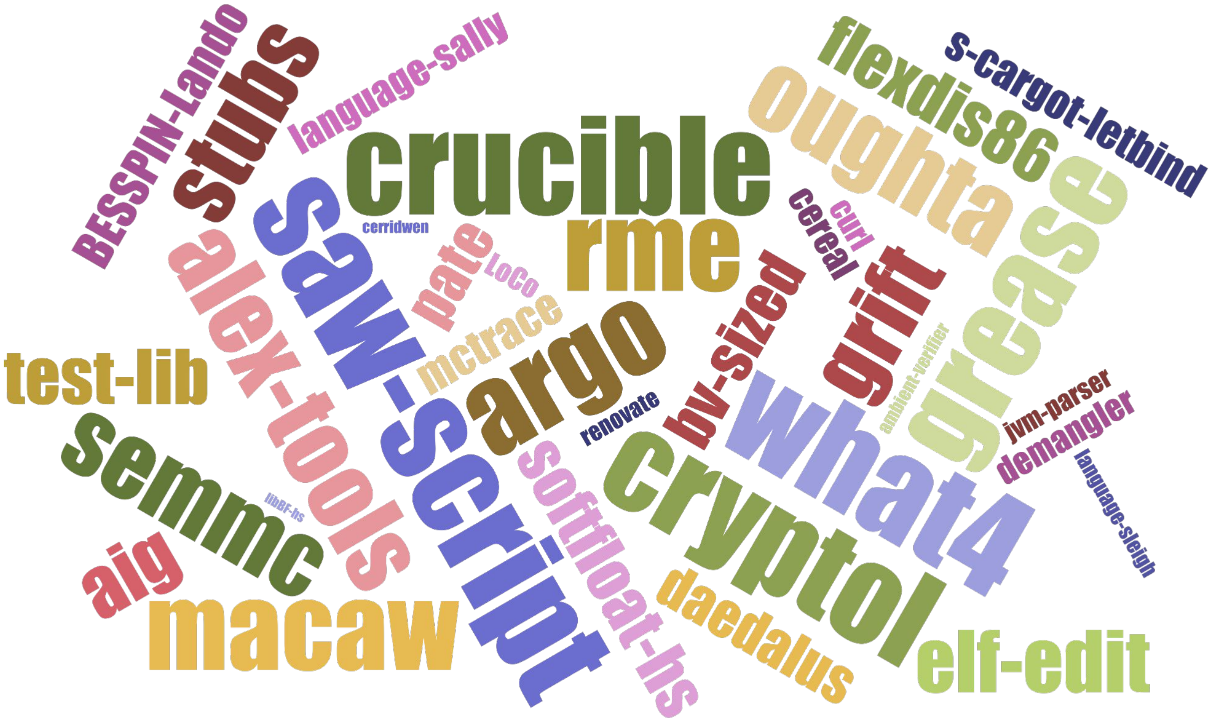
Way-too brief history of Haskell use at Galois

- We have been using Haskell since company's founding (1999).
- Cryptol (one of our first major tools) first publicly released in 2008, later released as open-source in 2014.

Way-too brief history of Haskell use at Galois

- We have been using Haskell since company's founding (1999).
- Cryptol (one of our first major tools) first publicly released in 2008, later released as open-source in 2014.
- Many formal verification tools and libraries, including SAW (2012), Crucible (2013), and What4 (2013), are written in Haskell.

Galois uses Haskell in *many* projects



Formal Specification and Verification

Formal Specification and Verification

- **Specification**: describing a system's design, features, requirements, and intended behavior (i.e., a blueprint)

Formal Specification and Verification

- **Specification**: describing a system's design, features, requirements, and intended behavior (i.e., a blueprint)

- **Formal** specification: mathematically rigorous design techniques (logic, type systems, etc.)

Formal Specification and Verification

- **Specification**: describing a system’s design, features, requirements, and intended behavior (i.e., a blueprint)

- **Formal** specification: mathematically rigorous design techniques (logic, type systems, etc.)

- **Formal verification**: mathematically proving that a system’s implementation conforms to its specification

Formal Specification

Specifying program behavior using Cryptol

Cryptol is a specification language, primarily intended for formally specifying the behavior of cryptographic algorithms.

Specifying program behavior using Cryptol

Cryptol is a specification language, primarily intended for formally specifying the behavior of cryptographic algorithms.

```
pairOfBitvectors : ([8], [16])
pairOfBitvectors = (255, 65535)
```

Specifying program behavior using Cryptol

Cryptol is a specification language, primarily intended for formally specifying the behavior of cryptographic algorithms.

```
pairOfBitvectors : ([8], [16])
pairOfBitvectors = (255, 65535)

flipAllBits : [8] -> [8]
flipAllBits bits = map complement bits
```

Executing Cryptol specifications

Executing Cryptol specifications

```
Cryptol> flipAllBits 0
255

Cryptol> flipAllBits 255
0

Cryptol> flipAllBits 127
128
```

Proving properties about Cryptol specifications

Proving properties about Cryptol specifications

```
Cryptol> :prove \(x : [8]) ->
    flipAllBits (flipAllBits x) == x
Q.E.D.
```

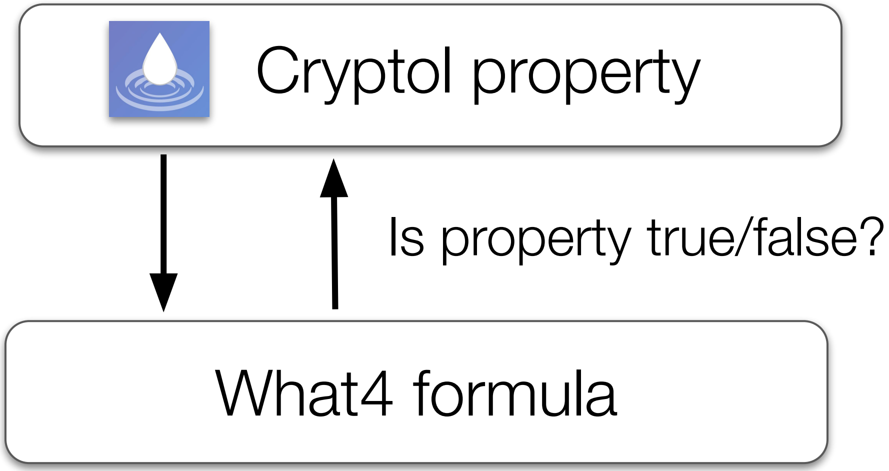
Proving properties about Cryptol specifications

```
Cryptol> :prove \(x : [8]) ->
    flipAllBits (flipAllBits x) == x
Q.E.D.

Cryptol> :prove \(x : [8]) ->
    flipAllBits x == x
Counterexample
(\(x : [8]) -> flipAllBits x == x) 0x00 = False
```

Talking to SMT solvers using What4

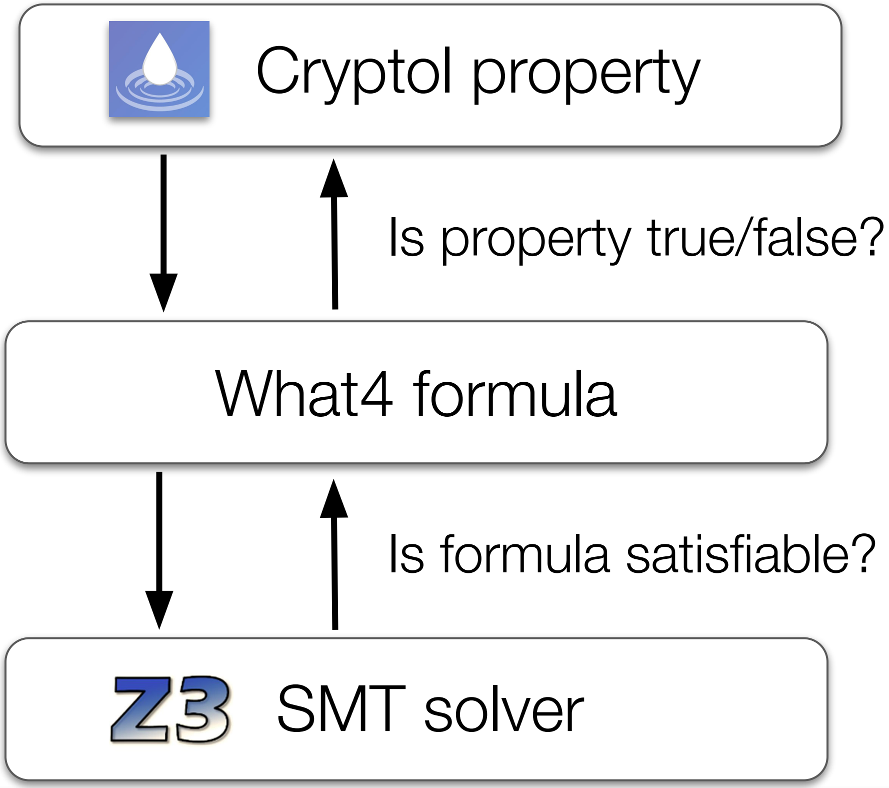
- To prove Cryptol properties, we translate Cryptol code into an intermediate language called What4



Talking to SMT solvers using What4

- To prove Cryptol properties, we translate Cryptol code into an intermediate language called What4

- What4 can easily be compiled into formulas that SMT solvers (e.g., Z3) can check for satisfiability



Design choice: *how* to talk to SMT solvers

Two competing options for how to communicate with SMT solvers:

1. Invoke SMT solver binaries as subprocesses (using Haskell's `process` library)
2. Use SMT solvers' C APIs (via Haskell's FFI)

Design choice: *how* to talk to SMT solvers

Two competing options for how to communicate with SMT solvers:

1. Invoke SMT solver binaries as subprocesses (using Haskell's `process` library)
2. Use SMT solvers' C APIs (via Haskell's FFI)

What4 picks option (1).

Design choice: when *not* to use Haskell

Cryptol and What4 both depend on an external C library (LibBF) to handle arbitrary-precision floating-point arithmetic.

Design choice: when *not* to use Haskell

Cryptol and What4 both depend on an external C library (LibBF) to handle arbitrary-precision floating-point arithmetic.

Why:

- It's a very mature library: not many updates required
- It's a very small library: easy to ship in a self-contained Haskell package without users needing to install external C libraries

Design choice: different libraries, different code styles

Design choice: different libraries, different code styles

- Cryptol, generally speaking, is written using Haskell2010 plus a mild number of GHC language extensions
- Mostly a product of the era in which Cryptol was first written (before most GHC language extensions were commonplace)

Design choice: different libraries, different code styles

- Cryptol, generally speaking, is written using Haskell2010 plus a mild number of GHC language extensions
- Mostly a product of the era in which Cryptol was first written (before most GHC language extensions were commonplace)

- What4 is written in a very different style of Haskell (lots of GADTs, type families, fancy type system features, etc.)
- Written with the goal of making SMT formulas type-correct by construction

Cryptol Haskell code

What4 Haskell code

Cryptol Haskell code

```
data Type
  = TCon !TCon ![Type]
  | TVar TVar
  | TUser !Name ![Type] !Type
  | TRec !(RecordMap Ident Type)
  | TNominal !NominalType ![Type]
```

What4 Haskell code

Cryptol Haskell code

```haskell
data Type
  = TCon !TCon ![Type]
  | TVar TVar
  | TUser !Name ![Type] !Type
  | TRec !(RecordMap Ident Type)
  | TNominal !NominalType ![Type]
```

What4 Haskell code

```haskell
data BaseTypeRepr (bt :: BaseType) :: Type where
   BaseBoolRepr :: BaseTypeRepr BaseBoolType
   BaseIntegerRepr :: BaseTypeRepr BaseIntegerType
   BaseRealRepr :: BaseTypeRepr BaseRealType
   BaseBVRepr ::
     (1 <= w) =>
     !(NatRepr w) ->
     BaseTypeRepr (BaseBVType w)
   BaseFloatRepr ::
     !(FloatPrecisionRepr fpp) ->
     BaseTypeRepr (BaseFloatType fpp)
   …
```

HFCS

HFCS (Haskell fancy code spectrum)

Simple
(Haskell2010)

Fancy
(many language extensions)

HFCS (Haskell fancy code spectrum)

Formal Reasoning

The challenge

How do we take popular imperative programming languages (C, Rust, Java, etc.) and reason about them *formally*?

Crucible

Crucible is a library for *symbolic execution* of imperative code:

Crucible

Crucible is a library for *symbolic execution* of imperative code:

- *Symbolic*: keeps program inputs abstract, enabling reasoning about multiple paths through a program simultaneously.

Crucible

Crucible is a library for *symbolic execution* of imperative code:

- *Symbolic*: keeps program inputs abstract, enabling reasoning about multiple paths through a program simultaneously.

- *Execution*: interprets (simulates) a program, producing mathematical representations (What4) of the program as output.

Typical Crucible workload

Imperative program

```
uint32_t f(uint32_t a[2], uint64_t idx) {
  return a[idx];
}
```

Typical Crucible workload

Imperative program

```
uint32_t f(uint32_t a[2], uint64_t idx) {
  return a[idx];
}
```

Crucible

What4

Typical Crucible workload

Imperative program

```
uint32_t f(uint32_t a[2], uint64_t idx) {
  return a[idx];
}
```

Crucible

What4

Program output

```
\(a : Vector Bv32) (idx : Bv64) ->
    arrayIndex a idx
```

Typical Crucible workload

Imperative program

```c
uint32_t f(uint32_t a[2], uint64_t idx) {
  return a[idx];
}
```

Crucible

What4

Program output

```
\(a : Vector Bv32) (idx : Bv64) ->
  arrayIndex a idx
```

Side conditions

```
0 <= idx && idx < 2
```

Crucible’s flavor of Haskell

Crucible’s Haskell style is largely inspired by What4 (lots of GADTs)

Crucible's flavor of Haskell

Crucible's Haskell style is largely inspired by What4 (lots of GADTs)

Simulating C using Crucible

Simulating C using Crucible

- C is a big, complicated language, so we first compile it to LLVM (using the Clang compiler).

Simulating C using Crucible

- C is a big, complicated language, so we first compile it to LLVM (using the Clang compiler).

- Crucible has an LLVM backend (Crucible-LLVM) that symbolically executes the LLVM code.

Simulating C using Crucible

- C is a big, complicated language, so we first compile it to LLVM (using the Clang compiler).

- Crucible has an LLVM backend (Crucible-LLVM) that symbolically executes the LLVM code.

- This means that we have to be able to ingest arbitrary LLVM code, which imposes some technical challenges.

Design choice: interfacing with LLVM in Haskell

One way to ingest LLVM's bitcode is to link against the LLVM API.

Design choice: interfacing with LLVM in Haskell

One way to ingest LLVM's bitcode is to link against the LLVM API.

- Pros: offloads the task off to LLVM itself.

Design choice: interfacing with LLVM in Haskell

One way to ingest LLVM’s bitcode is to link against the LLVM API.

- Pros: offloads the task off to LLVM itself.
- Cons: vastly complicates the packaging story (LLVM is a large dependency), and LLVM libraries for Haskell aren’t very well maintained.

Design choice: interfacing with LLVM in Haskell

Another way to ingest LLVM bitcode is to write a native Haskell library for parsing bitcode.

Design choice: interfacing with LLVM in Haskell

Another way to ingest LLVM bitcode is to write a native Haskell library for parsing bitcode.

- Pros: simpler packaging story (no external LLVM dependency).

Design choice: interfacing with LLVM in Haskell

Another way to ingest LLVM bitcode is to write a native Haskell library for parsing bitcode.

- Pros: simpler packaging story (no external LLVM dependency).
- Cons: LLVM's bitcode format changes often, which requires frequent maintenance to keep up to date with new LLVM versions.

Design choice: interfacing with LLVM in Haskell

Another way to ingest LLVM bitcode is to write a native Haskell library for parsing bitcode.

- Pros: simpler packaging story (no external LLVM dependency).
- Cons: LLVM’s bitcode format changes often, which requires frequent maintenance to keep up to date with new LLVM versions.

We chose this option and wrote our own library.

Simulating Rust using Crucible

To analyze Rust, we have multiple options:

Simulating Rust using Crucible

To analyze Rust, we have multiple options:

1. Simulate Rust code directly (but Rust is a big, complex language).

Simulating Rust using Crucible

To analyze Rust, we have multiple options:

1. Simulate Rust code directly (but Rust is a big, complex language).
2. Compile Rust to LLVM, then simulate LLVM (but the LLVM code is much, much more low-level than the Rust code).

Simulating Rust using Crucible

To analyze Rust, we have multiple options:

1. Simulate Rust code directly (but Rust is a big, complex language).
2. Compile Rust to LLVM, then simulate LLVM (but the LLVM code is much, much more low-level than the Rust code).
3. Compile Rust to a mid-level intermediate language (MIR) in between Rust and LLVM, then simulate that.

Simulating Rust using Crucible

To analyze Rust, we have multiple options:

1. Simulate Rust code directly (but Rust is a big, complex language).
2. Compile Rust to LLVM, then simulate LLVM (but the LLVM code is much, much more low-level than the Rust code).
3. Compile Rust to a mid-level intermediate language (MIR) in between Rust and LLVM, then simulate that.

We picked option (3).

Simulating Rust using Crucible

- Unlike LLVM's bitcode, Rust's MIR doesn't have a reliable on-disk representation that we can ingest.

Simulating Rust using Crucible

- Unlike LLVM's bitcode, Rust's MIR doesn't have a reliable on-disk representation that we can ingest.

- To work around this, we created our own Rust compiler plugin (`mir-json`) that dumps MIR in the middle of compilation to a custom, JSON-based format.

Simulating Rust using Crucible

- Unlike LLVM's bitcode, Rust's MIR doesn't have a reliable on-disk representation that we can ingest.

- To work around this, we created our own Rust compiler plugin (`mir-json`) that dumps MIR in the middle of compilation to a custom, JSON-based format.

- We then parse the JSON code into Crucible and symbolically execute it.

Design choice: ingesting LLVM vs. MIR

Design choice: ingesting LLVM vs. MIR

- Unlike with LLVM, where maintenance revolves around supporting new bitcode features, maintaining MIR support revolves around keeping a compiler plugin up to date.

Design choice: ingesting LLVM vs. MIR

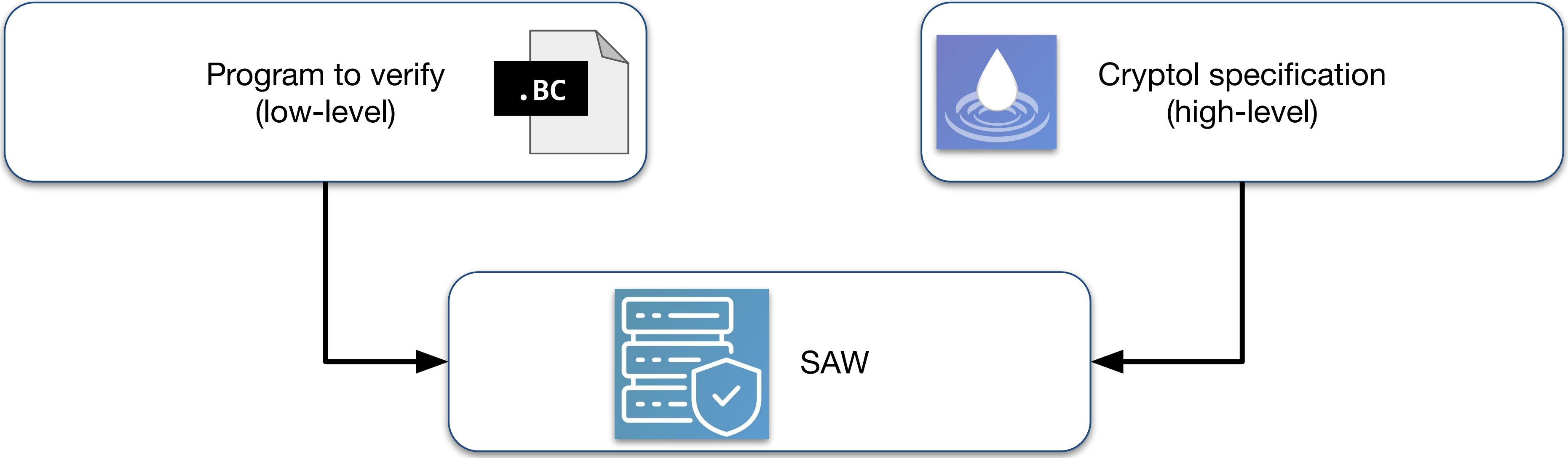
- Unlike with LLVM, where maintenance revolves around supporting new bitcode features, maintaining MIR support revolves around keeping a compiler plugin up to date.

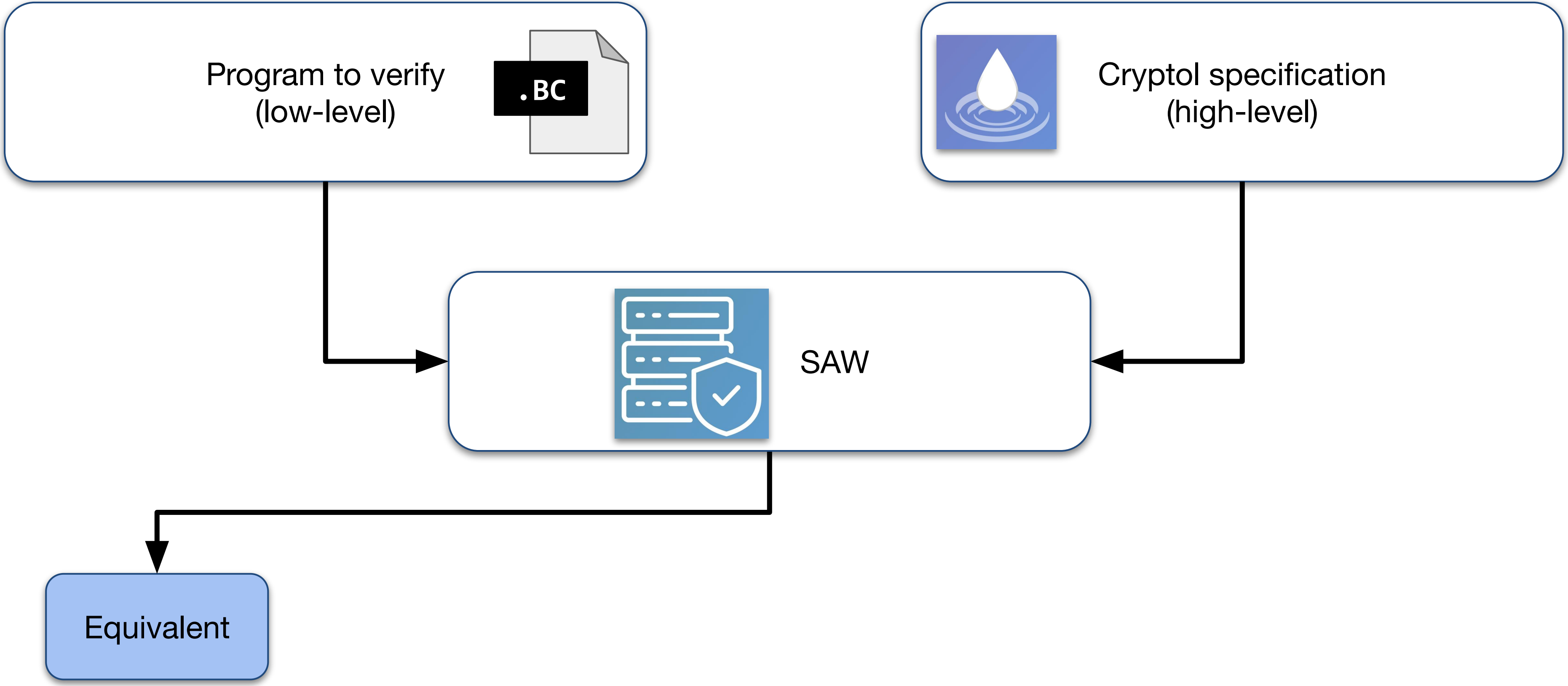
- We generally like to maintain Haskell code for ingesting other languages, but maintaining code in other languages (e.g., Rust) can also work.

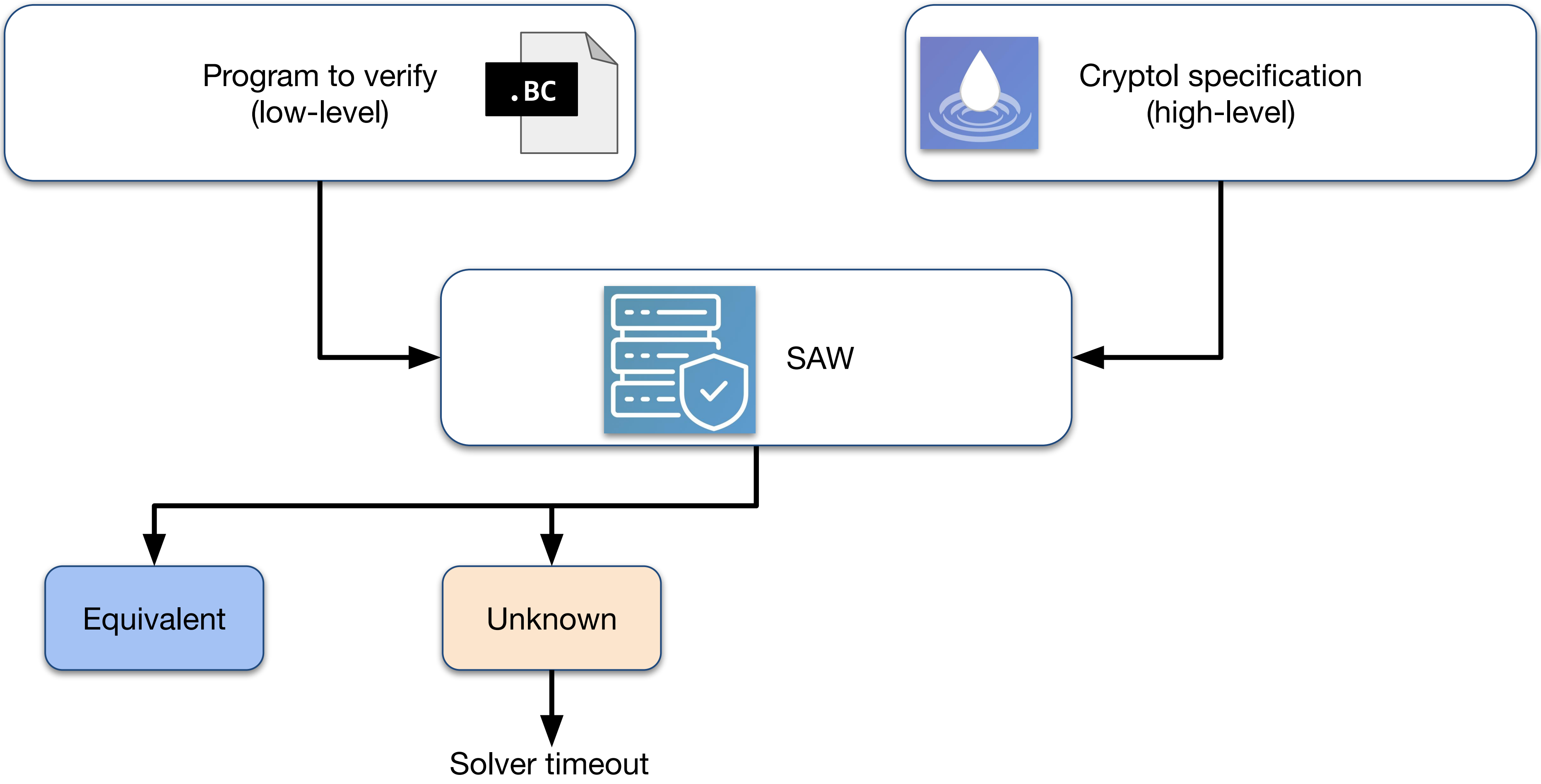
Formal Verification
(matching programs with specifications)

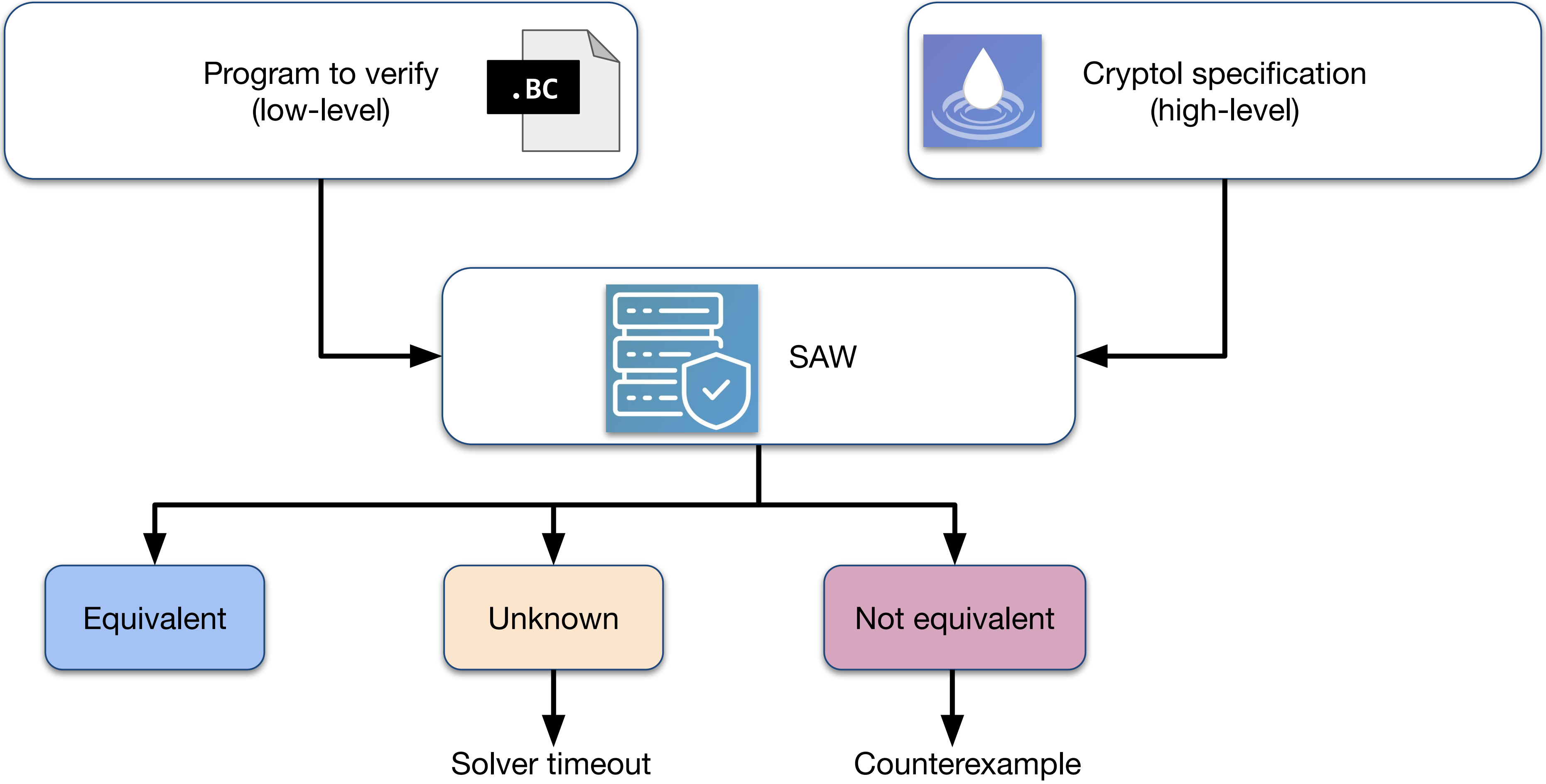
Verifying code against specs using SAW

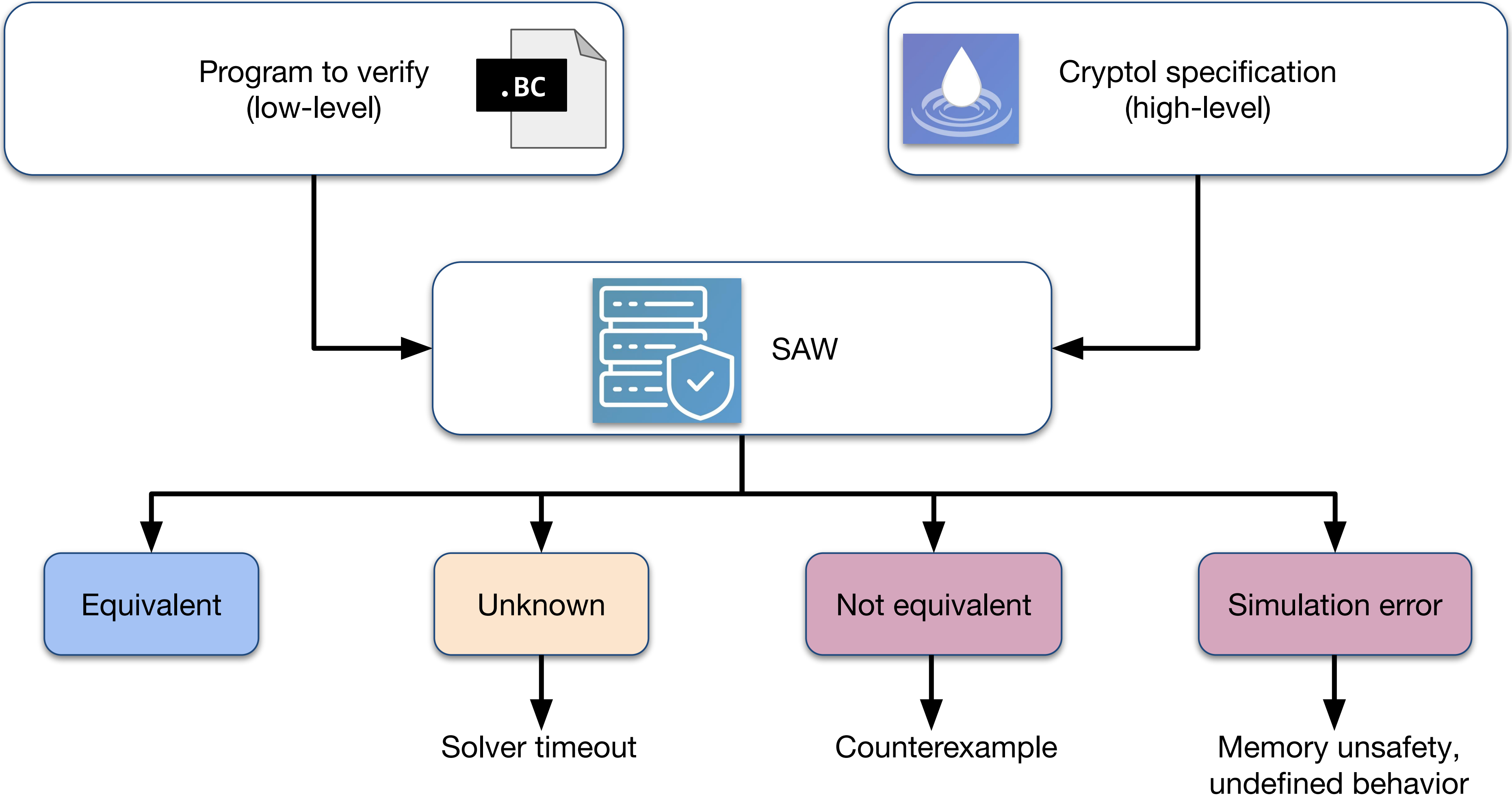
SAW (Software Analysis Workbench) is a tool for formally verifying properties of imperative code (using Crucible) against high-level Cryptol specifications.


Program to verify
(low-level)
.BC
Cryptol specification
(high-level)
SAW


Program to verify
(low-level)
.BC
Cryptol specification
(high-level)
SAW
Equivalent


Program to verify
(low-level)
.BC
Cryptol specification
(high-level)
SAW
Equivalent
Unknown
Solver timeout

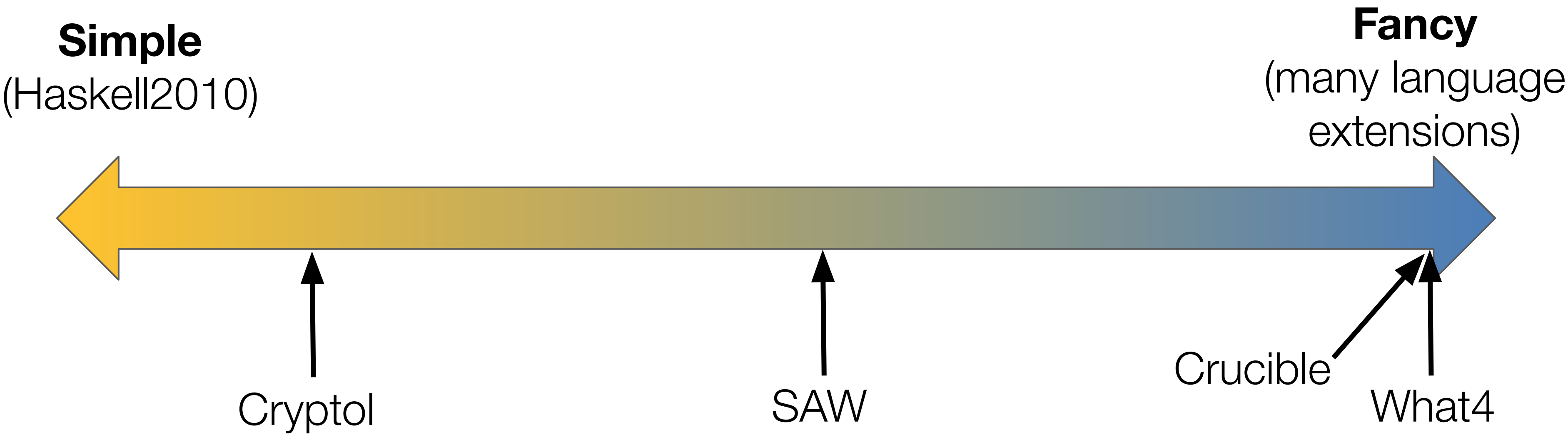

Program to verify
(low-level)
.BC
Cryptol specification
(high-level)
SAW
Equivalent
Unknown
Not equivalent
Solver timeout
Counterexample


Program to verify
(low-level)
.BC
Cryptol specification
(high-level)
SAW
Equivalent
Unknown
Not equivalent
Simulation error
Solver timeout
Counterexample
Memory unsafety,
undefined behavior

SAW's flavor of Haskell

SAW's flavor of Haskell

SAW's uses a mix of simple and fancy Haskell styles.



Reflections on industrial use of Haskell
(Here come the hot takes!)

Simple Haskell versus fancy Haskell

- We have used a variety of Haskell styles for different projects, ranging from simple (Cryptol) to fancy (What4, Crucible).

Simple Haskell versus fancy Haskell

- We have used a variety of Haskell styles for different projects, ranging from simple (Cryptol) to fancy (What4, Crucible).
- Much of the “fanciness” arises from trying to emulate features of dependent types in Haskell

Simple Haskell versus fancy Haskell

- We have used a variety of Haskell styles for different projects, ranging from simple (Cryptol) to fancy (What4, Crucible).
- Much of the "fancin
dependent types in

Dependently Typed Haskell in Industry (Experience Report)

DAVID THRANE CHRISTIANSEN, Galois, Inc., USA
IAVOR S. DIATCHKI, Galois, Inc., USA
ROBERT DOCKINS, Galois, Inc., USA
JOE HENDRIX, Galois, Inc., USA
TRISTAN RAVITCH, Galois, Inc., USA

Recent versions of the Haskell compiler GHC have a number of advanced features that allow many idioms from dependently typed programming to be encoded. We describe our experiences using this "dependently typed Haskell" to construct a performance-critical library that is a key component in a number of verification tools. We have discovered that it can be done, and it brings significant value, but also at a high cost. In this experience report, we describe the ways in which programming at the edge of what is expressible in Haskell's type system has brought us value, the difficulties that it has imposed, and some of the ways we coped with the difficulties.

Advantages of fancy Haskell code

Advantages of fancy Haskell code

- Encoding properties about SMT queries using Haskell's type system has caught certain bugs early, before they made it to production.

Advantages of fancy Haskell code

- Encoding properties about SMT queries using Haskell's type system has caught certain bugs early, before they made it to production.
- This greatly increases our confidence about the correctness of our code, even after performing large-scale refactors.

Disadvantages of fancy Haskell code

Disadvantages of fancy Haskell code

- Fancy code generally takes longer to train engineers to work with than simple code, which can increase development time.

Disadvantages of fancy Haskell code

- Fancy code generally takes longer to train engineers to work with than simple code, which can increase development time.
- Convincing GHC's typechecker of certain facts about type-level arithmetic can be surprisingly tricky.

Disadvantages of fancy Haskell code

- Fancy code generally takes longer to train engineers to work with than simple code, which can increase development time.
- Convincing GHC's typechecker of certain facts about type-level arithmetic can be surprisingly tricky.
- Heavy use of GADTs and type families results in very long compile times in certain cases.

Disadvantages of fancy Haskell code

- Fancy code generally takes longer to train engineers to work with than simple code, which can increase development time.
- Convincing GHC's typechecker of certain facts about type-level arithmetic can be surprisingly tricky.
- Heavy use of GADTs and type families results in very long compile times in certain cases.
- Fancy Haskell code is more likely to trigger GHC bugs!

Assorted thoughts on Haskell tooling

While Haskell tooling has improved quite a bit since I first started using the language (c. 2015), it is still lacking in the following areas:

Assorted thoughts on Haskell tooling

While Haskell tooling has improved quite a bit since I first started using the language (c. 2015), it is still lacking in the following areas:

- Code coverage (`hpc` is clunky, and achieving 100% code coverage is more difficult than it ought to be).

Assorted thoughts on Haskell tooling

While Haskell tooling has improved quite a bit since I first started using the language (c. 2015), it is still lacking in the following areas:

- Code coverage (`hpc` is clunky, and achieving 100% code coverage is more difficult than it ought to be).
- Documentation: Haddock is surprisingly slow on large projects, has some unintuitive default settings.

Assorted thoughts on Haskell tooling

While Haskell tooling has improved quite a bit since I first started using the language (c. 2015), it is still lacking in the following areas:

- Code coverage (`hpc` is clunky, and achieving 100% code coverage is more difficult than it ought to be).
- Documentation: Haddock is surprisingly slow on large projects, has some unintuitive default settings.
- Minimizing GHC bugs: I wish there was something like CReduce for GHC.

Not-so-hot takes

Not-so-hot takes

- Haskell is a great language for developing tooling for specification and verification!

Not-so-hot takes

- Haskell is a great language for developing tooling for specification and verification!

- We aren’t afraid to use other languages in our tech stack if it makes more sense to use them.

Not-so-hot takes

- Haskell is a great language for developing tooling for specification and verification!
- We aren’t afraid to use other languages in our tech stack if it makes more sense to use them.
- We err on the side of simple Haskell, but we may reach for fancy Haskell features if correctness is paramount.

Any questions?

Links to some Haskell-based tools we maintain:

`https://tools.galois.com/cryptol`

`https://tools.galois.com/saw`