

A Formalization of Core Why3 in Coq

Joshua M. Cohen ¹ Philip Johnson-Freyd ²

January 19, 2024

¹Princeton University

²Sandia National Laboratories

Building a Deductive Verifier

```
int insertionSort(int [] arr)
requires (arr != null)
ensures (sorted(arr))
{ ... }
```

VC Generator

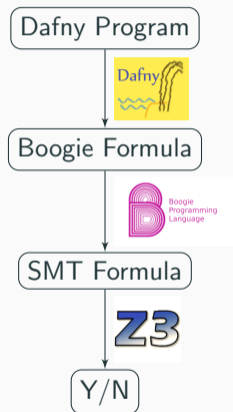
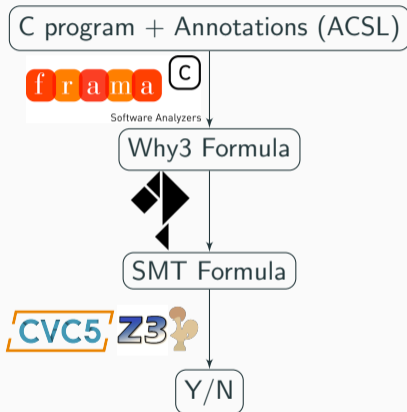
$x < y \wedge x + 2 < 5 \implies y > z \wedge \dots$

SMT Solver

- Translate the program + spec into verification conditions in SMT format
- Use heuristics and simplifications to ensure that the SMT formula is tractable
- Difficult - big gap between source and target logic

A Better Approach to Deductive Verifiers

Use an *intermediate verification language*:



Much easier to build verifier (need only to compile to intermediate language) and support multiple backend solvers

How can we be sure such a tool is correct?

- Translation from program logic/specification language to SMT formulas is highly nontrivial
- Toolchain spread across multiple large, complex software projects, large TCB (ex: Frama-C+Why3+CVC5 is $\approx 650k$ LOC, Z3 adds another 500k)
- Even if each tool is individually correct, differences in assumptions at the boundaries can compromise soundness
- Even stating a correctness theorem is not possible - most components do not have formal descriptions

Alternate approach: *foundational verification* (VST, Iris, CakeML, Bedrock2, etc)

- Formalize language semantics and program logic in proof assistant
- Develop custom proof automation to enable proofs in this logic
- Small TCB (only proof assistant kernel + semantics), typically very expressive
- But these tools are hard to build and use
 1. Requires knowledge of proof assistant and custom tactics, inaccessible to non-experts
 2. Cannot take advantage of SMT automation
 3. No intermediate languages - need to re-invent whole toolchain and automation each time

How can we retain automation while achieving foundational guarantees?

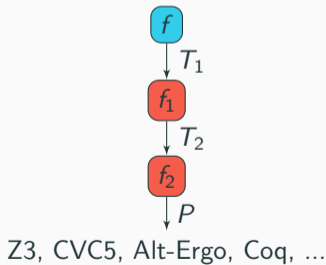
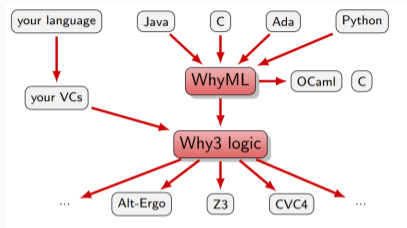
- Verifying entire toolchain not practical
- Crucial ingredient: verifying intermediate verification language (Why3)
- With this, writing verified deductive verifier reduced to problem of verified source \rightarrow Why3 compilation (VC generation)
- Verified intermediate language would do for foundational verifiers what Why3 and Boogie have done for verifiers in general

In order to do this, we need a *formal semantics* for Why3

Boogie (simpler logic) has been formalized [2], Why3 has not

Why3

- 2 parts: core logic and ML-like language with annotations built on top
- Logic has external API — some tools use logic directly
- Outputs to ≈ 20 backend solvers (SMT, proof assistants, other specialized solvers)
- Translate from higher-level logic to solver via series of transformations



Why3 in Practice

Why3 is used in many active verification tools and projects, including:



Frama C: C



EasyCrypt: Cryptography



SPARK 2014: Ada



Archetype: Smart Contracts



Creusot: Rust



Qbricks: Quantum Programs

Why3's Logic

Classical First-Order logic with

- Polymorphism
- Let and if-expressions
- (Mutually Recursive) Algebraic data types
- Pattern matching
- (Mutually) Inductive predicates
- (Mutually) Recursive functions and predicates
- Hilbert's epsilon operator

Goal: as expressive as possible while allowing efficient SMT-based automation

Similar to logics for other verification languages (Dafny, VeriFast, etc)

Denotational semantics: interpret Why3 terms/formulas as objects in Coq's logic

Start with Coq function $\text{dom}: \text{ty} \rightarrow \text{Type}$

Under (type/term) variable valuation, define interpretation:

- $\llbracket f \rrbracket_v : \text{Prop}$ for formula f
- $\llbracket t \rrbracket_v : \text{dom}(v(\tau))$ for term t of type τ

Core FOL is encoded as we expect:

- $\llbracket \text{Tvar } x \rrbracket_v = v(x)$
- $\llbracket f_1 \wedge f_2 \rrbracket_v = \llbracket f_1 \rrbracket_v \wedge \llbracket f_2 \rrbracket_v$
- $\llbracket \forall(x : \tau), f \rrbracket_v = \forall(d : \text{dom}(\tau)), \llbracket f \rrbracket_{x \rightarrow d, v}$

Functions and predicates interpreted as Coq functions of the appropriate type

- Requires some care to handle polymorphism correctly
- Use *heterogenous lists* to express well-typed function application

- Existing pen-and-paper description [1] of Why3 logic specifies conditions each structure must obey
 - E.g. ADT constructors injective and disjoint
- We give explicit, generic constructions satisfying these properties:
 - ADTs → **W-types**
 - Inductive predicates → **impredicative encoding**
 - Recursive functions → **well-founded recursion**

Recursive Structures - Design Decisions

- Highly layered approach: user of semantics only needs higher-level properties, not complex encodings
- Encode as generically as possible: ADTs and inductive predicates should be reusable in other developments
- Structures interact subtly: to define recursive functions, need to prove that pattern matching produces “smaller” variables according to well-founded relation on W-type encoding

Our Why3 Formalization

Define validity of Why3 formulas in context: formula is true under all interpretations consistent with type/function/predicate definitions

```
Definition valid ( $\Gamma$ : context) (f: formula) (f_ty: formula_typed  $\Gamma$  f)
(f_closed: closed_formula f) : Prop :=
 $\forall$  (p: pre_interp) (p_full: full_interp p) (v: valuation),  $\llbracket f \rrbracket_v$ .
```

We prove metatheorems about the logic:

- Consistency
- Existence of consistent interpretations (i.e. well-typed definitions can be constructed in Coq; no vacuous definitions)

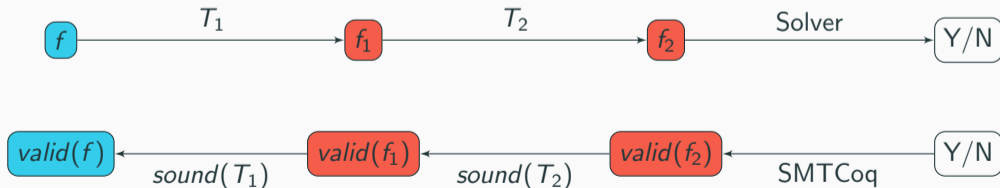
Applications: Why3 Transformations

Eventual goal: verified Why3 implementation

Want to prove correct Why3 transformations, can compose for full soundness

Demonstrate with 2 transformations from Why3:

- Eliminate let bindings - has nontrivial termination argument
- Axiomatize inductive predicates - replace inductive predicate with constructor axioms and inversion principle



Applications: Proving Why3 Goals

To show that our semantics are correct and useful, we also build a sound-by-construction proof system on top

- We prove sound all familiar natural deduction rules
- For convenience, build tactic library on top of proof system and wrote versions of many basic Coq tactics
 - Interesting ones: induction, unfold, simplify pattern match
- Prove lemmas from Why3's standard library about lists and trees using these tactics, for example:

```
lemma inorder_length:  $\forall t: \text{tree } 'a, \text{length } (\text{inorder } t) = \text{size } t$ 
```



- We can define tactics and prove goals in the way we expect - confidence that our semantics match the intended ones

Conclusion and Future Work

- We gave a formal semantics in Coq for the logic fragment of Why3, an extension of first-order logic with polymorphism and recursive structures
- Future work:
 1. Continue to verify transformations to produce a verified version of Why3 (with SMTCoq)
 2. Extend to features of Why3 not yet included (e.g. function types)
 3. With MetaCoq, allow idiomatic but foundational Why3 proof goals in Coq
- Key idea: verifying intermediate verification languages is useful and we have the tools to do it, this can help us build foundationally verified tools without sacrificing automation

Our formalization is available at github.com/joscoh/why3-semantic

Thanks for listening!

-  J.-C. Filliâtre.
One logic to use them all.
In M. P. Bonacina, editor, *Automated Deduction – CADE-24*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
-  G. Parthasarathy, P. Müller, and A. J. Summers.
Formally validating a practical verification condition generator.
In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification*, pages 704–727, Cham, 2021. Springer International Publishing.

Impredicative Encoding of Inductive Predicates

```
Inductive even : nat → Prop :=  
  | ev_0 : even 0  
  | ev_SS: ∀ n, even n → even (S (S n)).
```

```
Definition even' : nat → Prop :=  
  fun m ⇒ ∀ (P: nat → Prop),  
  P 0 → (∀ n, P n → P(S (S n))) → P m.
```

```
Lemma even_least : ∀ (P: nat → Prop), (*easy part*)  
  P 0 → (∀ n, P n → P(S (S n))) →  
  ∀ m, even' m → P m.
```

```
Lemma even_constrs: even' 0 ∧ (*hard part*)  
  ∀ n, even' n → even' (S (S n)).
```

```
Lemma even_equiv: ∀ n, even n ↔ even' n.
```