A Formalization of Core Why3 in Coq

JOSHUA M. COHEN, Princeton University, USA
PHILIP JOHNSON-FREYD, Sandia National Laboratories, USA

Intermediate verification languages like Why3 and Boogie have made it much easier to build program verifiers, transforming the process into a logic compilation problem rather than a proof automation one. Why3 in particular implements a rich logic for program specification with polymorphism, algebraic data types, recursive functions and predicates, and inductive predicates; it translates this logic to over a dozen solvers and proof assistants. Accordingly, it serves as a backend for many tools, including Frama-C, EasyCrypt, and GNATProve for Ada SPARK. But how can we be sure that these tools are correct? The alternate foundational approach, taken by tools like VST and CakeML, provides strong guarantees by implementing the entire toolchain in a proof assistant, but these tools are harder to build and cannot directly take advantage of SMT solver automation. As a first step toward enabling automated tools with similar foundational guarantees, we give a formal semantics in Coq for the logic fragment of Why3. We show that our semantics are useful by giving a correct-by-construction natural deduction proof system for this logic, using this proof system to verify parts of Why3's standard library, and proving sound two of Why3's transformations used to convert terms and formulas into the simpler logics supported by the backend solvers.

CCS Concepts: • **Software and its engineering → Software verification**; **Semantics**; • **Theory of computation → Logic and verification**; **Denotational semantics**.

Additional Key Words and Phrases: Formal Semantics, Why3, Coq, First-Order Logic

ACM Reference Format:
Joshua M. Cohen and Philip Johnson-Freyd. 2024. A Formalization of Core Why3 in Coq. *Proc. ACM Program. Lang.* 8, POPL, Article 60 (January 2024), 30 pages. https://doi.org/10.1145/3632902

1 INTRODUCTION

Building a program verifier is hard: a good approach involves producing verification conditions from an input program and specification, translating these conditions to a simpler logic supported by an automated solver (such as an SMT solver), and using heuristics and simplifications to ensure that the resulting goal is tractable. Richer specification languages result in a larger gap between the source and target logics, increasing the difficulty further. Intermediate verification languages like Why3 [Bobot et al. 2011] and Boogie [Barnett et al. 2006] make this process much easier by providing a higher-level logical interface to backend solvers and automatically simplifying and transforming formulas into formats supported by these solvers. With these tools, building a program verifier becomes a compiler problem rather than a proof automation problem. All the developer of a new source-language verifier needs to do is compile their source-level goals into the intermediate logic; the rest of the translation and solving is handled automatically. This approach has been widely adopted in many practical, state-of-the-art tools including Dafny [Leino 2010] and Frama-C [Cuoq et al. 2012].

Authors' addresses: Joshua M. Cohen, jmc16@princeton.edu, Princeton University, Princeton, NJ, USA; Philip Johnson-Freyd, pajohn@sandia.gov, Sandia National Laboratories, Livermore, CA, USA.

ACM 2475-1421/2024/1-ART60
https://doi.org/10.1145/3632902

For such an intermediate language to be useful, it should provide a powerful logic that permits expressive specifications, while still allowing automation via translation to simpler logics supported by automated solvers. Why3's logic has been carefully designed to achieve this middle ground. To the user it provides a rich feature set including multi-sorted first-order theories, polymorphic lemmas, inductive types and predicates, recursive functions, and several useful built-in theories. At the same time, it allows automation by targeting a variety of backend solvers including SMT solvers, automated first-order logic provers, and proof assistants. The translation from these higher-level features to SMT encodings is highly nontrivial, but this must only be done once in the Why3 tool itself. Using it alleviates a substantial portion of the challenge of building verification tooling.

How can we be sure that such a deductive verification tool is correct? That is, how do we know that claims "proven" by the source language verifier really hold? There are many places in the toolchain where errors could have crept in — the source verifier's translation of source goals to intermediate goals, the backend solver's decision of satisfiability or validity, and the intermediate language framework's non-trivial translation can all introduce errors. Each of these parts is a large, complex piece of software; the total amount of trusted code can easily exceed hundreds of thousands of lines. Additionally, even if all parts are individually correct, differences in assumptions or semantics between layers can still produce unsoundness.

An alternative approach to building verifiers is the *foundational* one taken by VST [Appel et al. 2014], Iris [Jung et al. 2018], CakeML [Kumar et al. 2014], and Bedrock2 [Erbsen et al. 2021] where the source program semantics are fully mechanized in an interactive proof assistant and a program logic is then defined and proved sound against that semantics in the same proof assistant. In some of these examples, the formal semantics of the source language are used to verify a compiler for the language, resulting in an end-to-end verified system with strong guarantees and a very small trusted computing base (only the proof assistant kernel and a formalization of the semantics of the target architecture), at least an order of magnitude smaller than automated toolchains.[1] However, this approach is challenging to implement in practice and has a high barrier to entry for users, both because these tools rely on proof assistants and specialized tactics, and because they lack access to the rich automation provided by the intermediate-language-based architecture.

We would like a way to have our cake and eat it too, retaining the automation provided by a first-order intermediate language while simultaneously achieving foundational guarantees. The crucial ingredient is to verify the intermediate language implementation; in other words, we would like to do for foundational program verifiers what Why3 and Boogie have done for program verifiers in general. With a verified Why3 implementation connected to a trustworthy backend solver (or validated results from an untrusted solver, as in SMTCoq [Ekici et al. 2017]), creating a foundational program verifier is reduced to the problem of proving the correctness of the compilation of source language goals to intermediate goals (i.e., verification condition generation).

To illustrate this further, Figure 1 shows the current pipeline for Frama-C (most deductive verifiers using an intermediate language will have a similar pipeline). A C program and its specification via annotations (in this case in the Frama-C specification language ACSL) are given as input to Frama-C, which generates Why3 formulas. The Why3 tool applies a series of transformations to produce formulas in a format supported by a particular SMT solver. The solver evaluates the validity of these formulas and returns a yes/no answer, which is presented to the user. On the right hand side of Figure 1, we show a hypothetical verified version of this same pipeline. At the lowest level, with a tool like SMTCoq, we can prove that the SMT solver output is correct for the input

[1]For instance, verifying C code with VST involves a TCB of around 20K LOC, while Frama-C+Why3+CVC4 comprise around 500K LOC; Z3 adds another 500K. This does not account for the compilers, OS, etc. For a more thorough examination of CompCert's TCB, see [Monniaux and Boulmé 2022].

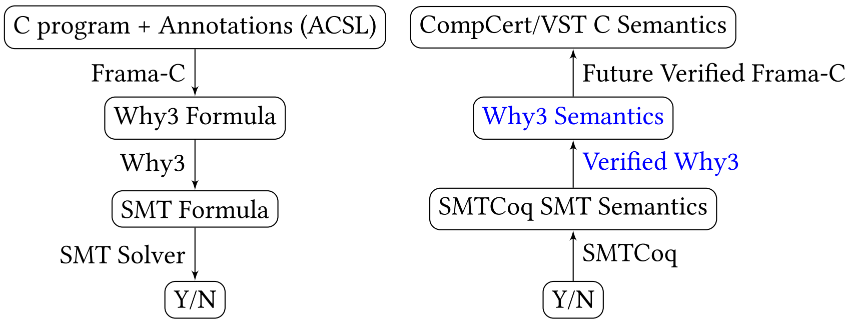

Fig. 1. Current (left) and hypothetical verified (right) Frama-C pipeline; our work aims to enable the blue components

SMT formula. Next, a verified Why3 implementation would prove that validity is preserved by Why3's transformations for this input (either via verification or translation validation) and thus that the original Why3 formula was valid. Finally, a verified Frama-C implementation (i.e. a verified verification condition generator) would prove that the validity of the generated Why3 formulas implies that the annotations for the original program hold.

To prove such a pipeline correct, we need theorems establishing correctness of each stage of translation; that is, a definition of validity and a proof that validity is preserved in each component. There are Coq formalizations of SMT validity [Ekici et al. 2017] and C semantics and a program logic [Appel et al. 2014; Leroy 2009]. However, there is no existing formalization of Why3 validity. To that end, we describe such a formal semantics of core Why3 in the Coq proof assistant. We also demonstrate that these semantics make it possible to state and prove the correctness of the syntactic transformations performed within Why3 and therefore to create a verified implementation of Why3.

An alternate approach to building a verified deductive verifier is translation validation – checking that the output of a verification tool is correct on specific inputs. For example, one would check that the SMT formulas produced by Frama-C imply the correctness of the original program. However, while this may be easier for simple source languages, such a task would be extremely difficult for a tool like Frama-C due to the vast differences between the source and target semantics. Moreover, if one wanted to use a different backend solver or source language, the entire process would need to be repeated. Hence, we believe that the intermediate-language-based approach is the correct one for verifying, just as for building, program verifiers. Our work represents a necessary first step towards a verified implementation of Why3 and therefore towards a world in which building provably correct program verifiers is significantly more practical.

The semantics we give is a denotational (Tarski style) one, closely inspired by Why3's pen and paper semantics [Filliâtre 2013]. Tarski style semantics are ideal for reasoning about validity across logics, and are superficially simple. Nonetheless, some of the features of Why3 — such as polymorphism, inductive types, and recursive functions — induce significant complexity. To manage those features, we turn to the rich literature base of programming language semantics, for instance using W-types to encode inductive types and using an impredicative encoding of inductive predicates. A user of our semantics, when translating from a source language, no longer needs to worry about defining these tricky features correctly, as the semantics already does so.

Equivalently, our semantics gives a *model* of Why3's logic in Coq – we explicitly construct a Coq term for each well-typed Why3 term, formula, and recursive structure with the required properties. This goes beyond existing pen-and-paper proofs, which describe the conditions that such a model should satisfy [Filliâtre 2013] but do not explicitly construct one. Without explicit constructions one may fear pathological behavior such as the possibility that introducing a new inductive type was not "conservative" and changed the validity of theorems which did not mention it–our approach ensures this can not happen: all logic features are handled once and for all. Explicit construction requires additional reasoning about the subtle interactions between different parts of the logic (§5.3).

1.1 Contributions

Our contributions are as follows:

(1) Using the Coq proof assistant, we provide the first formalization of the logic fragment of Why3, including its syntax, type system, and a verified typechecker.
(2) We give a denotational semantics for Why3 in Coq, encoding algebraic data types, pattern matching, recursive functions, and inductive predicates. We believe that this is the first formalization of such a logic that lies between standard first-order logic and full higher-order logic/dependent type systems.
(3) We show that our semantics are useful and allow us to effectively reason about the different parts of the logic by demonstrating several applications:
   (a) We verify the soundness of two of Why3's transformations, including the axiomatization of inductive predicates.
   (b) We give a sound-by-construction, natural-deduction-style proof system for Why3's logic.
   (c) We build a tactic library on top of this proof system and demonstrate the practicality of our design by using these tactics to prove theorems about lists and trees from the Why3 standard library.

Section §2 gives a brief introduction to Why3 and Coq. In §3 we discuss our deep embedding of Why3 in Coq, while in §4 we present the semantics for the core first-order logic. §5 discusses the semantics for the more complicated parts of the logic — recursive types and functions, inductive predicates, and pattern matching. §6 gives the full semantics and extends with Why3 concepts such as tasks and theories; we also prove some meta-theorems about Why3's logic. In §7, we evaluate our semantics through applications and discuss which parts of our development could be reusable and applicable beyond Why3. Finally, §8 presents related work.

2 BACKGROUND - WHY3 AND COQ

Why3 consists of two parts: a rich logic for program specification (our focus) and an ML-like language WhyML with a built-in verification condition generator (generating formulas in the underlying logic) [Filliâtre and Paskevich 2013]. Though Why3 can be used to directly write verified software, as WhyML can be extracted to OCaml, its primary purpose is as a backend for other verification tools. It outputs to at least 19 provers, including SMT solvers (Z3, CVC4, and Alt-Ergo, among others), specialized numeric and first-order logic solvers (Vampire, Gappa, etc.), and proof assistants (Coq, Isabelle, and PVS). Accordingly, Why3 has been used in many verification tools and projects, including Frama-C [Cuoq et al. 2012] for C, EasyCrypt [Barthe et al. 2011] for cryptography, SPARK 2014 [AdaCore and Ltd 2018] for Ada, as well as tools for verifying Rust [Denis et al. 2022], OCaml [Pereira and Ravara 2021], quantum circuits [Chareton et al. 2021], and distributed systems [Lourenço and Pinto 2022]. Some of these tools construct WhyML programs, while others directly create terms and formulas in Why3's logic, which is accessible via an external OCaml API.

Why3's logic is fairly complex; it is a typed, classical first-order logic with rank-1 polymorphism, algebraic data types (ADTs), pattern matching, recursive functions and predicates, inductive predicates, and Hilbert's epsilon operator. All above structures can be mutually recursive. We note that other unrelated verification tools, such as Dafny and VeriFast [Jacobs et al. 2011], have broadly similar specification languages (supporting ADTs, recursive functions, etc), suggesting that this is indeed a sensible middle ground that enables one to write rich specifications while remaining efficient in practice. This logic is described in more detail by Filliâtre [2013]; we base our formalization heavily on this pen-and-paper description.

Why3's logic language is structured into *theories*, consisting of abstract or concrete definitions for type, function, and predicate symbols, as well as lemmas, goals, axioms, and imports of previously defined theories. Using a theory (use t) makes all definitions, lemmas, and axioms in t (and its dependencies) available to use, while cloning (clone t as T **with** ...) creates a new copy of a theory, possibly instantiating some of the abstract type, function, and predicate symbols with concrete values. Verifying a theory reduces to verifying a series of *tasks*, which consist of a context with the declared symbols and definitions, a set of assumptions, and the goal to be proved. Why3 proves these goals by applying a series of *transformations* to the tasks based on the specific features supported by the solver in use — for example, axiomatizing inductive types or inlining function definitions.

We use the Coq proof assistant for our formalization. Based on a modern variant of the Calculus of Inductive Constructions, Coq implements a higher-order logic through a dependently typed programming language named Gallina. To prove theorems in Coq, one can either proceed interactively using tactics that modify the proof state or directly write proof terms in Gallina; we utilize both approaches as needed. In our formalization, we make extensive use of dependent types, dependent pattern matching, and several dependently typed data structures, described in §4 and §5. Coq is a particularly good choice for our formalization; we need powerful inductive types (capable of encoding and reasoning about W-types), induction over arbitrary well-founded predicates, and an impredicative Prop, among other features. Additionally, for a future verified version of Why3 proved correct against our semantics, Coq's ability to extract Gallina code to OCaml would be critical. Other, simpler systems like LF require less trust, but would not allow us to express many of these complex patterns.

Since Why3 is a classical logic and Coq implements an intuitionistic logic, our semantics uses 3 axioms: classical logic (law of the excluded middle), indefinite description (Hilbert's epsilon operator), and functional extensionality; all are included in Coq's standard library and are known to be consistent with Coq and each other (see §7.5 for more details).

Why3 and Coq already interact in several ways. Coq is one of the backend proof assistants for Why3, providing support for goals outside the scope of SMT solvers. Additionally, one can *realize* Why3 theories in Coq (and PVS); that is, give a model for the theory axioms and definitions. For example, one can realize the Why3 int theory with Coq's Z library. Then, future Coq proofs of Why3 goals can use the instantiated datatypes and definitions directly. Finally, Why3 provides a Coq tactic to call external provers as oracles and admit the goal; this allows one to use Coq as a backend while still using SMT solvers for simpler goals.

3 SYNTAX AND TYPING

Syntax. We deeply embed Why3's syntax in Coq, representing Why3 types, patterns, terms, and formulas as Coq inductive types. Here, we give a brief overview of the syntax, see [Filliâtre 2013] for more details. For clarity, we present the syntax in standard mathematical notation; our Coq datatypes match this representation very closely.

Figure 2 shows the syntax for types, patterns, terms, and formulas. Why3 types (τ in informal notation, vty in our Coq formalization) consist of built-in types int and real, type variables (α, typevar), and the application of a previously declared type symbol (t, typesym) to a list of type arguments. We denote x_τ to be a variable x of type τ (in Coq, vsymbol := string * vty). Patterns (p, pattern) consist

$$
\begin{array}{rcl}
\tau \in \text{Types} & := & \mathit{int} \mid \mathit{real} \mid \alpha \mid \mathrm{t}(\tau, \dots, \tau) \\
p \in \text{Patterns} & := & x_\tau \mid \mathrm{f}(\tau, \dots, \tau)(p, \dots, p) \mid \_ \mid (p \mid p) \mid p \;\mathit{as}\; x_\tau \\
t \in \text{Terms} & := & c_{int} \mid c_{real} \mid x_\tau \mid \mathrm{f}(\tau, \dots, \tau)(t, \dots, t) \mid \textbf{let}\; x_\tau := t \;\textbf{in}\; t \mid \epsilon x_\tau . f \\
& \mid & \textbf{if}\; f \;\textbf{then}\; t \;\textbf{else}\; t \mid (\textbf{match}\; t \;\textbf{with} \mid p \to t \mid \dots \mid p \to t \;\textbf{end}) \\
f \in \text{Formulas} & := & \mathrm{p}(\tau, \dots, \tau)(t, \dots, t) \mid \forall x_\tau, f \mid \exists x_\tau, f \mid t = t \mid f \wedge f \mid f \vee f \mid f \implies f \\
& \mid & f \iff f \mid \neg f \mid \top \mid \bot \mid \textbf{let}\; x_\tau := t \;\textbf{in}\; f \mid \textbf{if}\; f \;\textbf{then}\; f \;\textbf{else}\; f \\
& \mid & (\textbf{match}\; t \;\textbf{with} \mid p \to f \mid \dots \mid p \to f \;\textbf{end})
\end{array}
$$

Fig. 2. Syntax of types, patterns, terms, and formulas

$$
\begin{array}{rcl}
d \in \text{Def} & := & \textbf{datatype}\; a \;\textbf{with}\; \dots \;\textbf{with}\; a \mid \textbf{recursive}\; \delta \;\textbf{with}\; \dots \;\textbf{with}\; \delta \\
& \mid & \textbf{inductive}\; i \;\textbf{with}\; \dots \;\textbf{with}\; i \\
a \in \text{ADT} & := & \mathrm{t}(\alpha, \dots, \alpha) = \mathrm{f}(\alpha, \dots, \alpha)(\tau, \dots, \tau) : \tau \mid \dots \mid \mathrm{f}(\alpha, \dots, \alpha)(\tau, \dots, \tau) : \tau \\
\delta \in \text{RecFun} & := & \textbf{function}\; \mathrm{f}(\alpha, \dots, \alpha)(x_\tau, \dots, x_\tau) : \tau = t \\
& \mid & \textbf{predicate}\; \mathrm{p}(\alpha, \dots, \alpha)(x_\tau, \dots, x_\tau) = f \\
i \in \text{IndPred} & := & \mathrm{p}(\alpha, \dots, \alpha)(\tau, \dots, \tau) = f \mid \dots \mid f
\end{array}
$$

Fig. 3. Syntax of concrete definitions

of variables, constructor application, wildcards, or patterns, and binding (where p as x matches p and binds x to the result). Terms (t, term) consist of constant integer and rational literals, variables, function symbol (f, funsym) application, let binding, epsilon, conditionals, and pattern matching. Formulas (f, formula) consist of predicate symbol (p, predsym) application, quantifiers, equality, binary operators (and, or, implies, and iff), negation, true/false, let binding, conditionals, and pattern matching. Note that terms and formulas are mutually recursive, as a term conditional involves a formula. Also note that this logic is first-order: only terms can be quantified over and bound in let and match expressions.

Finally we have definitions for symbols: algebraic data types, recursive functions and predicates, and inductive predicates (Figure 3). We can additionally have abstract definitions for type, function, and predicate symbols. A context Γ is a list of definitions (both concrete and abstract), which will be needed in our semantics as validity is with respect to a context.

Typing. Why3's logic is typed: the judgments are that, in context Γ, a type τ is valid (consisting of declared type symbols), that a term t has type τ (term_has_type Γ t τ) and that a formula f is well-typed (formula_typed Γ f). The typing rules for types, patterns, terms, and formulas are standard and we largely omit them here, see [Filliâtre 2013] for the full typing rules (our formalization makes a few minor changes to the typing rules, mainly because we handle contexts differently). We make a few remarks: first, for function and predicate application, the applied types are substituted for the symbol's type parameters. For instance, given function symbol $\mathrm{reverse}(\alpha)(\mathrm{list}(\alpha)) : \mathrm{list}(\alpha)$ and term $\mathrm{reverse}(\mathrm{int})([1; 2])$, int is substituted for α when typechecking. Second, the typing rules place restrictions on free variables in patterns: in a constructor pattern, no two argument patterns can have overlapping free variables (e.g., Why3 does not allow $x :: x :: t$ as a pattern), and the free variables in an "or" pattern must be identical. Finally, pattern matches in terms and formulas require that the type being matched on is indeed an ADT and that matching is exhaustive.

Typechecking definitions is more complicated. Why3 includes numerous checks on the declared definitions: ADTs require that the constructors all have the same parameters and correct return

$$f_0 \quad := \quad p(\alpha_1, \ldots, \alpha_n) \mid f \implies f_0 \mid \forall x_\tau, f_0 \mid \textbf{let } x_\tau := t \textbf{ in } f_0$$

Fig. 4. Grammar for inductive predicates

type. Furthermore, Why3 checks that all data types are inhabited by finding a constructor whose types can be proved to be inhabited (assuming all abstract types are inhabited). Recursive functions and predicates require that all free variables in the body are declared in the function arguments and that all functions terminate according to a syntactic check: there is a lexicographic order of arguments that guarantees a structural descent over ADTs. Finally, inductive predicates have a special syntactic form; for inductive predicate $\mathrm{p}(\alpha_1, \ldots, \alpha_n)(\tau_1, \ldots \tau_n)$, a clause f_i must be closed and belong to the grammar of Figure 4. Additionally, all predicates from the mutually recursive block must occur only in strictly positive positions in the constructors.

In our formalization, we implement versions of many of these checks: for inhabited types, recursive function termination, inductive predicate strict positivity, and inductive predicate grammar. However, we are more restrictive in some cases (see §7.4 for details and discussion of how this affects soundness): we currently require a structurally decreasing argument for termination (unlike Why3's more general lexicographic ordering), we require that all recursive definitions are uniform — the type parameters of recursive instances must be the same as those of the input — and we do not (yet) implement a check for exhaustive pattern matching.

These checks are critical in defining the semantics for recursive definitions (§5). In fact, our encoding serves as a proof that these syntactic checks truly suffice to define the objects that we are interested in. For example, we show that the syntactic termination check for recursive functions implies that our Coq representation of the function only ever calls itself on arguments that are "smaller" according to a particular well-founded relation (based on structural inclusion of the underlying ADT representation); this is required to define a function in Coq.

Finally, we define what it means for an entire context to be well-typed: no type, function, or predicate symbol can be defined multiple times, all individual definitions must be well-typed as defined above, and each definition can only refer to earlier definitions except within mutually recursive blocks. Then, we write verified typecheckers for types, patterns, terms, formulas, definitions, and contexts. Most of this is fairly straightforward; the most complicated is the termination check for recursive functions: we must identify an index for each function in the mutually recursive block such that the argument at this index is an ADT and all recursive calls occur on syntactically smaller arguments at this index.

4 SEMANTICS FOR THE FIRST-ORDER LOGIC

In this section we present the semantics for terms and formulas without pattern matching or ADTs (we consider these in §5.1 and §5.2). We give a denotational semantics, where we represent Why3 terms and formulas as objects in Coq's logic (formulas, for instance, will be represented as a Coq bool[2]). In order to represent a term or formula containing type, function, or predicate symbols, as well as free type or term variables, we need to first define interpretations of these objects.

We define a *sort* as a monomorphic type (e.g., list int is a sort, list a is not). A *pre-interpretation* assigns a meaning to all sorts, function, and predicate symbols. Specifically, we first assign each sort a nonempty Coq **Set**, setting ints and reals to $\mathbb{Z}$ and $\mathbb{R}$, respectively:

```
Definition domain (domain_aux: sort → Set) (s: sort) : Set :=
  match sort_to_ty s with | vty_int ⇒ Z | vty_real ⇒ R | _ ⇒ domain_aux s end.
```

[2]The axioms we use (classical logic and indefinite description), allow bool and Prop to be used interchangeably.

```
Inductive base := | bint | bbool.
Definition base_to_ty (b: base) : Type := match b with | bint ⇒ Z | bbool ⇒ bool end.
Definition example : hlist base_to_ty [bint; bbool; bint] := HL_cons 1 (HL_cons false (HL_cons 3 HL_nil)).
```

Fig. 5. Example use of heterogeneous lists

```
Record pi_dom := {
  dom_aux: sort → Set;
  domain_ne: ∀ s, domain_nonempty (domain dom_aux) s; }
```

We denote domain (dom_aux pd) s as domain pd s or as $[\![s]\!]$ when pd is clear from the context.

Next, we need to assign an interpretation for function and predicate symbols. This is trickier: for function symbol $\mathrm{f}(\boldsymbol{\alpha})(\tau_1, \ldots, \tau_n) : \tau$ and a list of sorts $\boldsymbol{s}$, we want $\mathrm{f}(\boldsymbol{s})$ to be a function of type $[\![\sigma(\tau_1)]\!] \times \ldots \times [\![\sigma(\tau_n)]\!] \longrightarrow [\![\sigma(\tau)]\!]$, where σ is the map that sends each α in $\boldsymbol{\alpha}$ to the corresponding s in $\boldsymbol{s}$. Predicate symbols are similar, but return a bool.

Representing this in Coq is not trivial, as the function argument types may differ. We encode this using *heterogeneous lists*, a dependently typed data structure in which, given f: A → Type and l: list A, the ith element has type f (nth i l) (for a more detailed presentation of heterogeneous lists, see Chlipala [2013]). We can define this as follows:

```
Inductive hlist {A: Type} (f: A → Type) : list A → Type :=
  | HL_nil: hlist f nil
  | HL_cons: ∀ x tl, f x → hlist f tl → hlist f (x :: tl).
```

Figure 5 shows an example of how to use an hlist to construct a 3-element list where the first and last elements are integers and the middle one is a boolean. This simple example can be implemented with a tuple, but if the length is not known statically (as in our semantics), we need an hlist. We provide a generic library for hlists and various operations (inversion, length, indexing, filtering, etc) that we use in numerous parts of the semantics.

With hlists, we can represent the function and predicate symbol pre-interpretation as follows, where sym_sigma_args and sym_sigma_ret represent the map σ above acting on the function arguments and return type, respectively:

```
Record pi_funpred (pd: pi_dom) := {
  funs: ∀ (f:funsym) (srts: list sort), hlist (domain pd) (sym_sigma_args f srts) →
    domain pd (funsym_sigma_ret f srts);
  preds: ∀ (p:predsym) (srts: list sort), hlist (domain pd) (sym_sigma_args p srts) → bool; }
```

We will similarly denote funs pd pf f s as $[\![\mathrm{f}(s)]\!]$ when the pre-interpretation is clear from the context, and likewise for predicate symbols.

With this pre-interpretation (which for now ignores ADTs, but we will return to this in §5.1), we can define a *valuation*, which describes how we should interpret type and term variables in a term or formula. A type variable valuation vt maps each type variable to a sort. We can easily extend this to map arbitrary types to sorts by replacing all type variables (we call this function v_subst). With an abuse of notation, we will denote v_subst vt ty as $vt(ty)$. Then, a term variable valuation maps each variable x_τ to an element of $[\![vt(\tau)]\!]$:

```
Definition val_typevar := typevar → sort.
Definition v_subst (v: typevar → sort) (t: vty) : sort := ...
Definition val_vars (pd: pi_dom) (vt: val_typevar) := ∀ (x: vsymbol), domain pd (v_subst vt (snd x)).
```

$$
\begin{array}{rcl}
[\![c_{int,\ real}]\!]_v & = & c \\
[\![x_\tau]\!]_v & = & v(x_\tau) \\
[\![\textbf{let } x_\tau := t_1 \textbf{ in } t_2]\!]_v & = & [\![t_2]\!]_{v[x_\tau \to [\![t_1]\!]_v]} \\
[\![\textbf{if } f \textbf{ then } t_1 \textbf{ else } t_2]\!]_v & = & \textbf{if } [\![f]\!]_v \textbf{ then } [\![t_1]\!]_v \textbf{ else } [\![t_2]\!]_v \\
[\![\epsilon x_\tau . f]\!]_v & = & \epsilon(\lambda y \ \to \ [\![f]\!]_{v[x_\tau \to y]}) \\
\\
[\![\top]\!]_v & = & \text{true} \\
[\![\bot]\!]_v & = & \text{false} \\
[\![\forall x_\tau, f]\!]_v & = & \forall d, [\![f]\!]_{v[x_\tau \to d]} \\
[\![\exists x_\tau, f]\!]_v & = & \exists d, [\![f]\!]_{v[x_\tau \to d]} \\
[\![t_1 = t_2]\!]_v & = & [\![t_1]\!]_v = [\![t_2]\!]_v \\
[\![f_1 \otimes f_2]\!]_v & = & [\![f_1]\!]_v \otimes [\![f_2]\!]_v, \otimes \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \\
[\![\neg f]\!]_v & = & \neg [\![f]\!]_v \\
[\![\textbf{let } x_\tau := t \textbf{ in } f]\!]_v & = & [\![f]\!]_{v[x_\tau \to [\![t]\!]_v]} \\
[\![\textbf{if } f_1 \textbf{ then } f_2 \textbf{ else } f_3]\!]_v & = & \textbf{if } [\![f_1]\!]_v \textbf{ then } [\![f_2]\!]_v \textbf{ else } [\![f_3]\!]_v
\end{array}
$$

Fig. 6. Definitions of term_rep and formula_rep

We can now define mutually recursive Coq functions term_rep and formula_rep that give the meaning of a (well-typed) Why3 term or formula under context Γ, pre-interpretation (pd, pf), and type variable valuation vt:

term_rep: ∀ (v: val_vars pd vt) (t: term) (ty: vty) (Hty: term_has_type Γ t ty), domain (v_subst vt ty)
with formula_rep: ∀ (v: val_vars pd vt) (f: formula) (Hty: formula_typed Γ f), bool

The typing proof is essential for several typecasts in the functions (for instance, if x_{τ_1} has type τ_2, then $\tau_1 = \tau_2$). To help with the dependent pattern matching, we use Coq's Equations package [Sozeau and Mangin 2019]. Informally, we denote term_rep v t ty Hty as $[\![t]\!]_v : [\![vt(ty)]\!]$ and formula_rep v f Hty as $[\![f]\!]_v : bool$, omitting the typing proofs.

Figure 6 shows the definitions of term_rep and formula_rep, omitting function and predicate application for now. Note that $v[x \longrightarrow y]$ denotes a valuation v' identical to v except that x is sent to y. We encode epsilon using Coq's indefinite description axiom and ClassicalEpsilon library. The classical axioms we use imply a strong version of the law of the excluded middle (∀ (P: Prop), {P}+ {~P}), allowing us to use Prop and bool interchangeably.

Function and Predicate Application. We would like to say the following (the predicate case is similar):

$$[\![\mathrm{f}(\tau_1, \dots, \tau_m)(t_1, \dots, t_n)]\!]_v \quad = \quad [\![\mathrm{f}(vt(\tau_1), \dots, vt(\tau_m))]\!]([\![t_1]\!]_v, \dots, [\![t_n]\!]_v)$$

This is more challenging than it appears. $[\![t_1]\!]_v, \dots, [\![t_n]\!]_v$ hides a large amount of complexity: term_rep must be recursively called on a list of terms to construct an hlist of the appropriate type. This necessitates a nested fixpoint or separate recursive function, which we implement as fun_arg_list, with the type:

```
Definition fun_arg_list {ty: vty} (vt: val_typevar) (f: funsym) (vs: list vty) (ts: list term)
  (reps: ∀ (t: term) (ty: vty), term_has_type Γ t ty → domain pd (v_subst vt ty))
  (Hty: term_has_type Γ (Tfun f vs ts) ty):
hlist (domain pd) (sym_sigma_args f (map (v_subst vt) vs)).
```

Conceptually, this function is fairly simple: it calls the function reps on each element of ts, bundling the results into an hlist. Since reps is instantiated with term_rep (recursively), this function must additionally be written in such a way that Coq can tell that all calls to reps occur on elements of ts, which are structurally smaller than the original input to term_rep. To create the hlist, we define the function using tactics (in fact, we define a more general function that allows us to write fun_arg_list and pred_arg_list as simple applications).

But this doesn't quite work; the types do not match. In particular, if we have function symbol f with parameters $\boldsymbol{\alpha}$ and argument types $\boldsymbol{t}$ applied to types $\boldsymbol{\tau}$ and arguments $\boldsymbol{x}$, the typing rules dictate that argument x_i must have type $\sigma(t_i)$, where σ sends $\boldsymbol{\alpha} \rightarrow \boldsymbol{\tau}$. Thus, $[\![x_i]\!]_v$ has type $vt(\sigma(t_i))$. However, in order to apply the function interpretation $[\![\mathrm{f}(vt(\tau_1), \ldots, vt(\tau_m))]\!]$, we need $[\![x_i]\!]_v$ to have type $\sigma'(t_i)$, where σ' sends $\boldsymbol{\alpha}$ to $vt(\boldsymbol{\tau})$. Therefore, we must show that, for any type t, $vt(\sigma(t)) = \sigma'(t)$; in other words, σ and vt commute as type substitutions (though they are very different: σ sends a small set of variables to specific types, vt maps every variable to a sort). Then, we can include a typecast in fun_arg_list when creating each element of the hlist. We note that the pen-and-paper description of this logic [Filliâtre 2013] omits this subtlety; this is one of several places where our formalization discovers small gaps in the informal semantics.

5 EXTENDING WITH RECURSIVE STRUCTURES

So far, we have defined a fairly standard classical multi-sorted first-order logic; the real complexity comes from adding recursive structures — types, functions, and predicates. These structures are fundamentally higher-order; we cannot completely axiomatize them in first-order logic and therefore we need a stronger logic like Coq or ZFC to define our semantics.

Filliâtre [2013] handles these features by imposing additional conditions on the interpretation (for instance, that the interpretation of an ADT's constructor must be injective, and so on). But this approach has two problems for our purposes. First, if the conditions are contradictory or our typing rules are not strong enough, these conditions might be unsatisfiable and then our semantics become trivial. Second, we cannot tell if these conditions are sufficient. In fact, Filliâtre describes ADT representations as the free algebra generated by the constructors, but gives an incomplete set of conditions; it does not include an induction principle, which we will need in order to define recursive functions over these types and to prove Why3 goals that require induction (see §5.1 for more discussion).

Instead, we take a different approach: we take Filliâtre's set of conditions on the interpretation (adding conditions as needed) and interpret this as an API that any encoding of these structures must satisfy. Then, we construct objects in Coq satisfying these conditions and thus prove that, under the typing rules, consistent interpretations do exist for any possible assignment to uninterpreted symbols. In other words, we construct an explicit model of Why3's logic. In the following sections, we describe our construction of generic recursive types, recursive functions, and inductive predicates in Coq, as well as our implementation of generic pattern matching. We take a highly layered approach, where each construction has 3 parts: the complex, dependently typed core encoding, a simpler representation layer encapsulating the core, and the API, which we prove satisfied by this simpler representation. In later parts of the semantics and applications, only the API is needed; the core encoding's complexity is completely hidden.

5.1 Algebraic Data Types

Core Encoding. To generate arbitrary algebraic data types (lists, trees, etc) from a syntactic description, we use W-types [Martin-Löf 1982], which provide a generic way to represent a variety of inductive types. We give a brief description before describing our encoding.

```
Variable (vars: typevar → Set) (abstract: typesym → list vty → Set).
Definition get_nonind_vtys (l: list vty) : list vty := ... (*Keep only the non-recursive types*)
Definition vty_to_set (v: vty) : Set := match v with (*Convert Why3 type to Coq Set*)
  | vty_int ⇒ Z | vty_real ⇒ R | vty_var x ⇒ vars x | vty_cons ts vs ⇒ abstract ts vs end.
Fixpoint big_sprod (l: list Set) : Set := ... (*Iterated tuple*)
(*Build the base type for a single constructor*)
Definition build_constr_base (c: funsym) : Set := big_sprod (map vty_to_set (get_nonind_vtys (s_args c)).
(*Build the base type for a nonempty list of constructors*)
Fixpoint build_base (constrs: ne_list funsym) : Set :=
  match constrs with
  | ne_hd hd ⇒ build_constr_base hd
  | ne_cons hd tl ⇒ Either (build_constr_base hd) (build_base tl)
  end.

Definition build_constr_rec (ts: typesym) (c: funsym) : Set := finite (count_rec_occ ts c).
Fixpoint build_rec (ts: typesym) (constrs: ne_list funsym) : (build_base constrs → Set) :=
  match constrs with
  | ne_hd f ⇒ fun _ ⇒ build_constr_rec ts f
  | ne_cons f fs ⇒ fun o ⇒ match o with
    | Left _ ⇒ build_constr_rec ts f
    | Right y ⇒ build_rec ts fs y
    end
  end.
```

Fig. 7. Construction of non-recursive (build_base) and recursive (build_rec) data in W-type

As a running example, we consider a type that represents a binary tree implementing a map from int to string: data tree = | Leaf | Node **of** (tree * int * string * tree). In the Node case, we can bundle the non-recursive elements in a tuple (int * string), and we can represent the recursive ones as the single argument bool → tree. This gives an equivalent constructor Node: (int * string) → (bool → tree) → tree. In general, for every constructor, we can separate the non-recursive and recursive arguments in this way; if there are n recursive arguments, we encode the recursion as C → t, where $|C| = n$.

To generalize this, we can construct a type A bundling the non-recursive data for all constructors. In our example, this can be Either unit (int * string), which says that the first constructor has no data, and the second has an int and a string. Then, we encode the recursive arguments by defining a type B giving the type C for each constructor; thus, we want B : A → **Set**. This tells us how many recursive instances we have for the constructor to which the argument A corresponds. In our example, B (**Left** _) = empty and B (**Right** _) = bool, representing 0 and 2 recursive calls, respectively. To extend this to mutually recursive types, we include an additional index I: **Set**, where I identifies each ADT in the block; then A: I → **Set** (the non-recursive data for each ADT in the block), and B: ∀ (i: I), A i → **Set** (the number of recursive instances for an ADT in the block and for a constructor in that ADT). With this intuition, we can define W-types in Coq.

```
Variable (I: Set) (A: I → Set)(B: ∀ (i: I) (j: I), A i → Set).
Inductive W : I → Set :=
  | mkW : ∀ (i: I) (a: A i) (f: ∀ j, B i j a → W j), W i.
```

To encode Why3 mutual ADTs as W-types, we must define I, A, and B. I is a type with exactly n

```
Definition adt_rep (m: mut_adt) (srts: list sort) (dom: sort → Set) (a: alg_datatype)
  (a_in: adt_in_mut a m) : Set := mk_adts ...
Definition constr_rep {gamma: context} (gamma_valid: valid_context gamma)
  (m: mut_adt) (m_in: mut_in_ctx m gamma) (srts: list sort)
  (srts_len: length srts = length (m_params m)) (dom: sort → Set) (t: alg_datatype)
  (t_in: adt_in_mut t m) (c: funsym) (c_in: constr_in_adt c t)
  (*All ADTs must be mapped appropriately – same condition as in [pi_dom]*)
  (adts: (∀ (a : alg_datatype) (Hin : adt_in_mut a m),
        domain dom (typesym_to_sort (adt_name a) srts) = adt_rep m srts dom a Hin))
  (a: hlist (domain dom) (sym_sigma_args c srts)):
  adt_rep m srts dom t t_in := make_constr ...
```

Fig. 8. Abstraction layer for ADTs

elements, where n is the number of ADTs in the mutual block. Figure 7 shows the construction of A (build_base) and B (build_rec). A is an iterated Either over all constructors, where the type for each constructor is given by an iterated tuple of the non-recursive argument types. We assume that we have a meaning for type variables and non-recursive type symbols, which we will define later. This involves some slightly awkward mechanisms, including a separate nonempty list type; this avoids littering the resulting types with Either _ empty. B uses a type corresponding to the number of recursive instances of the ith mutual ADT for a given constructor (we call this build_constr_rec), and then matches on the A i argument to determine which constructor case it is in; calling build_constr_rec.

The full encoding for mutual block m as a W-type is thus:

```
Definition mk_adts : finite (length m) → Set :=
  W (finite (length m)) (fun n ⇒ build_base (adt_constrs (fin_nth m n)))
     (fun this i ⇒ build_rec (adt_name (fin_nth m i)) (adt_constrs (fin_nth m this))).
```

Encoding the constructors is more complicated. We need a function that, given a constructor symbol f for the nth ADT in mutual block m, an instance of build_constr_base (the bundled non-recursive arguments) and an instance of the recursive arguments (expressed as a function from |m| to length-indexed lists of recursive instances), gives an instance of mk_adts n. The function has the signature:

```
Definition make_constr (n: finite (length m)) (f: funsym)
  (Hin: ... (*f is a constructor for the nth mutual type*))
  (recs: ∀ (x: finite (length m)), (count_rec_occ (adt_name (fin_nth m x)) f).-tuple (mk_adts x))
  (c: build_constr_base f) : mk_adts n := mkW (finite (length m)) ...
```

where count_rec_occ gives the number of recursive instances of the given type symbol in a function symbol's arguments, and n-tuple is a length-indexed list from the SSReflect [Gonthier and Mahboubi 2010] library.

Abstraction Layer. These definitions encode Why3 ADT descriptions as inductive types, but they are not very convenient to use; the user should not need to manually construct finite types, build_constr_base instances, or the complicated dependent function describing the recursive instances for a constructor. Thus, we implement a simpler abstraction that hides the low-level details of the W-type implementation. We consider a mutual ADT applied to a list sort, as well as a pre-interpretation for type symbols. The resulting representation of ADTs (adt_rep) is much simpler

```
Theorem constr_rep_disjoint: ∀ {f1 f2: funsym} {f1_in: constr_in_adt f1 t} {f2_in: constr_in_adt f2 t}
  (a1: arg_list domain (sym_sigma_args f1 srts)) (a2: arg_list domain (sym_sigma_args f2 srts)),
  f1 ≠ f2 →
  constr_rep f1 f1_in dom_adts a1 ≠ constr_rep f2 f2_in dom_adts a2.
Theorem constr_rep_inj: ∀ {f: funsym} {f_in: constr_in_adt f t}
  (a1 a2: arg_list domain (sym_sigma_args f srts)),
  constr_rep f f_in dom_adts a1 = constr_rep f f_in dom_adts a2 →
  a1 = a2.
Definition find_constr_rep (x: adt_rep t t_in) :
  {f: funsym & {Hf: constr_in_adt f t * arg_list (domain) (sym_sigma_args f srts) |
  x = constr_rep f (fst Hf) dom_adts (snd Hf)}}.
```

Fig. 9. API for ADTs

(Figure 8). We then add to our pre-interpretation for types (§4) the condition that all ADTs must be mapped to their corresponding adt_rep:

```
Record pi_dom := { ...; (*previous components*)
  adts: ∀ (m: mut_adt) (srts: list sort) (a: alg_datatype) (Hin: adt_in_mut a m),
  domain dom_aux (typesym_to_sort (adt_name a) srts) = adt_rep m srts dom_aux a Hin; }
```

For the constructors to match our interpretation of function symbols, we need a function that takes in an hlist of instances of the constructor's arguments, and outputs an element of the appropriate adt_rep. Therefore, we filter out recursive and non-recursive arguments from the hlist and bundle them appropriately (as a build_constr_base and as the dependent function for recs, respectively). Bundling the recursive arguments is conceptually simple, but the dependent types in the hlist and the tuple make this harder. For instance, we need to treat an hlist with elements of the same type as an ordinary Coq list. Nevertheless, the resulting representation is clean, with hypotheses and types that are easy to work with (Figure 8).

With this, we can add to our pre-interpretation for function and predicate symbols the requirement that all constructor symbols map to constr_rep (with a typecast to the appropriate domain):

```
Record pi_funpred := { ...;
  constrs: ∀ m t c m_in t_in c_in srts srts_len a, funs c srts a =
    constr_rep_dom gamma_valid m m_in srts srts_len (dom_aux pd) t t_in c c_in (adts pd m srts) a}.
```

An API for adt_reps. Filliâtre [2013] handles ADTs by stating that any pre-interpretation must satisfy 3 conditions: the constructor representations must be disjoint and injective, and that every instance of an ADT interpretation must have a constructor representation and arguments that produce it. Figure 9 gives these theorems for our representation. Note that we need a stronger version of the 3rd property: it is not enough that such a constructor exists, we must have a Coq function to find it.

This property is particularly difficult to prove; we need an inverse function for constr_rep, allowing us to create an hlist from the the bundled non-recursive arguments and bundled tuple-map for the recursive arguments of make_constr. We can build the hlist inductively, but we must update the tuple map appropriately at each step; the dependent types make our induction hypotheses tricky to state and difficult to use. We partially alleviate this by converting to an alternate representation that does not use length-indexed tuples; instead it uses lists and maintains separate proofs about length, with lemmas to convert back and forth. Ultimately, we define this inverse function as a sigma type

```
Theorem adt_rep_ind m m_in srts (Hlen: length srts = length (m_params m))
  (P: ∀t t_in, adt_rep m srts (dom_aux pd) t t_in → Prop):
  (*For any ADT t in mutual block m, constructor c, instance x = c(a) of t*)
  (∀ t t_in (x: adt_rep m srts (dom_aux pd) t t_in) (c: funsym)
    (Hc: constr_in_adt c t) (a: hlist domain (sym_sigma_args c srts)) (Hx: x = constr_rep ... c a),
    (*If, whenever P holds of all recursive instances in a*)
    (∀ i t' t_in' Hithrec, i < length (s_args c) → P t' t_in' (cast ... (hnth i a s_int ...))) →
    (*Then P holds of x*)
    P t t_in x) →
  (*Then P holds for all instances of t*)
  ∀t t_in (x: adt_rep m srts (dom_aux pd) t t_in), P t t_in x.
```

Fig. 10. Generalized induction principle for adt_rep

with the hlist and a proof that it is the inverse of the original conversion function; the definition and proof are long and complex, but this is hidden; we only need to know that such a computable inverse function exists. For similar reasons, find_constr_rep is also defined as a sigma type.

As noted, these 3 conditions are not quite enough; they do not fully characterize ADTs and would allow us to write nonterminating recursive functions. ADTs not only satisfy these 3 conditions, they must also be the smallest object satisfying them; to encode this, we prove an induction theorem. This property is not first-order; accordingly, our theorem is higher-order and relies on the additional strength of Coq's logic. We need this induction principle in particular to prove that these ADT representations are well-founded thereby enabling us to define (terminating) recursive functions over them (§5.3). Figure 10 gives this theorem, which ultimately (but nontrivially) follows from the well-foundedness of the underlying W-type in Coq. With these theorems, adt_rep and constr_rep can effectively be considered opaque; only these 4 theorems are ever needed by clients.

We remark that our encoding has some limitations: it cannot handle non-uniform inductive types (where the type parameters change in recursive calls[3]) or nested inductive types (e.g., rose trees). The latter can be encoded using mutually recursive types; however, our representation makes critical assumptions about uniformity, and nonuniform types would require a substantially revised encoding (and it is unclear how often these are used in practice in Why3 — see §7.4). Nevertheless, our semantics can handle arbitrary list- and tree-like mutually recursive data structures sufficient for a wide variety of real-world specifications.

5.2 Pattern Matching

With our representation of ADTs and updated pre-interpretation definitions, we can complete the definition of term_rep and formula_rep by including pattern matching. Filliâtre [2013] interprets patterns by compiling them into elementary tests; this is useful for implementation but the algorithms for doing so are non-trivial and difficult to specify cleanly. Instead, we take a simpler, semantics-oriented approach against which such compilation could be proved correct. In particular, we encode pattern matching by describing how a successful pattern match affects the variable valuation val_vars, adding new bindings for the variables bound in the pattern. To that end, we define a function match_val_single that matches an element of domain pd (v_subst vt ty) against a pattern p of type ty, returning an optional list that includes all the new variable bindings if the term matches the pattern, and None otherwise. This function has the signature:

[3]For instance type B a = | One a | Two (B (a, a)) is a datatype for perfect binary trees - those with 2^n nodes; see Okasaki [1996] for more discussion and uses of non-uniform types.

$$
\begin{array}{rcl}
[\![x_\tau, ty, d]\!] & = & Some[(x, \{vt(ty), d\})] \\
[\![\_, ty, d]\!] & = & Some[] \\
[\![p1 \mid p2, ty, d]\!] & = & \text{if isSome}[\![p1, ty, d]\!] \text{ then } [\![p1, ty, d]\!] \text{ else } [\![p2, ty, d]\!] \\
[\![p\ as\ x_\tau, ty, d]\!] & = & l \leftarrow [\![p, ty, d]\!]; \text{return } (x, \{vt(ty), d\}) :: l
\end{array}
$$

Fig. 11. Definition of match_val_single

```
Fixpoint match_val_single (v: val_typevar) (ty: vty) (p: pattern) (Hp: pattern_has_type gamma p ty)
  (d: domain pd (v_subst v ty)) : option (list (vsymbol * {s: sort & domain (dom_aux pd) s}))
```

Since we don't know yet which variables will be bound, the sigma type lets us hide the specific value while avoiding extra dependent type obligations that would make the definition unwieldy. We will later prove that if (x, y) appears in the outputted list of match_val_single, then the s component of y is v_subst vt (snd x) - in other words, we really do add valid valuation pairs. We will also prove that the output list consists of a permutation of the pattern's free variables. But it is simpler to define our function without proving these facts immediately.

Figure 11 gives the definition of match_val_single for the non-constructor cases. We let $[\![p, ty, d]\!]$ represent the result of match_val_single on pattern p of type ty matching against d: domain pd (v_subst vt ty), where pd : pi_dom and vt: val_typevar are implicit (we use a monadic notation for the bind case).

The last and most interesting case is constructor application (f tys ps). We check that the given type is an ADT; if so, the argument matched on must be of type adt_rep. Then, the constructor-finding function find_constr_rep is used to identify the constructor and arguments. If the resulting constructor is f and all of the arguments recursively match ps, then the overall valuation is formed by concatenating all of the recursively constructed lists. The trickiest part of this is dealing with the constructor's arguments - just as in the function case for term_rep, we again need nested recursion over hlists as well as a typecast to ensure that the elements of this hlist have the correct types.

Finally, we extend term_rep and formula_rep with pattern matching. We do so in the natural way: for **match** t **with** ps **end**, we take $[\![t]\!]_v$, and iterate through ps (a list of pattern * term or pattern * formula), calling match_val_single for each one. Whenever we reach Some l for the first time (say, on input (p_i, t_i) in ps), we return $[\![t_i]\!]_{l@v}$, where $l@v$ is the valuation v with the bindings in l added. We use a nested fixpoint to implement this (unlike with fun_arg_list, Coq cannot tell that the recursion terminates otherwise). Pattern matching thus completes our definitions of term_rep and formula_rep; we now have a full, formal definition of the truth of a term or formula under an interpretation and valuation.

Properties of the Denotational Semantics. term_rep and formula_rep are at the core of our semantics; for them to be useful, we need various invariance lemmas that allow us to change parts of an interpretation, valuation, or context without changing the result. For example, the functions are invariant under changes to the variable valuation vv that agree on all free variables in the term or formula. Likewise, we can change the function and predicate symbol interpretation pf as long as the two interpretations agree on all function and predicate symbols appearing in the term (this is crucial for inductive predicates and recursive functions). We can also change the type variable valuation if it agrees on all free type variables in the term or formula, but this changes the return type of term_rep, so we need a typecast.

Additionally, we define syntactic substitution over type and term variables and prove that these coincide with the semantic definitions: e.g., $[\![t[t_1/x]]\!]_v = [\![t]\!]_{v[x \rightarrow [\![t_1]\!]_v]}$ if no variable from t_1 becomes bound in t (we generalize this to the multi-variable case as well). Again, the type substitution case is similar but trickier because it changes the return type of term_rep. We also define α-equivalence and show that α-equivalent terms have identical denotations, allowing us to give a capture-avoiding substitution function that also coincides with the semantic one. For most of these proofs, we prove the result recursively for fun_arg_list and pred_arg_list, prove a result describing how match_val_single changes under e.g., changing vt, and finally prove the theorem for term_rep and formula_rep. This process is quite involved, but a few general results suffice for a variety of applications (see §7).

5.3 Recursive Functions

The specification for (mutually) recursive functions and predicates over ADTs is simple: for an interpretation to be consistent with a function definition $\mathrm{f}(\alpha_1, \dots \alpha_m)(x_1, \dots, x_n) = b$, $[\![\mathrm{f}(\mathbf{s})]\!](\mathbf{a}) = [\![b]\!]_v$ for any sorts $\boldsymbol{s}$ and arguments $\boldsymbol{a}$, where v maps $\boldsymbol{\alpha}$ to $\boldsymbol{s}$ and $\boldsymbol{x}$ to $\boldsymbol{a}$ (the predicate case is similar). This specification is subtle; $[\![b]\!]$ is evaluated in the context where $[\![\mathrm{f}]\!]$ is interpreted correctly. This circularity requires us to give an explicit definition of $[\![\mathrm{f}]\!]$ as a recursive function in Coq. Since all functions must be provably terminating in Coq (as in Why3), we encode these using *well-founded recursion.*[4]

Core Encoding. Broadly, our approach is as follows: we define a relation on adt_rep representing structural inclusion and prove that this is well-founded. We define the recursive function via well-founded induction on (a lifted version of) this relation; the function body consists of modified versions of term_rep and formula_rep that make recursive calls whenever evaluating a function or predicate call to a symbol in the mutually recursive block. To prove that every recursive call occurs on a "smaller" value, we keep track of several invariants about the relationship between syntactically smaller variables (from the termination check in the typing rules) and their semantically smaller valuations.

The syntactic termination check is largely straightforward; it maintains a list small of known smaller variables and a optional variable hd of the input (which becomes None if this value is shadowed). Whenever checking a function call, if the call is recursive (i.e., in the current mutual block), it ensures that the element at the terminating index is a variable in small. New small variables are added in a pattern match: matching on hd or a variable in small adds all variables arising (transitively) in the free variables of a constructor pattern. The overall check requires that all functions in the mutual block have a terminating index when small is initially empty.

Our core encoding for the function assumes that we have all needed information (e.g., the mutual ADT m and sorts srts on which we recurse and the decreasing index for each function in the mutual block). One can define general recursive functions in Coq on well-founded relations (a relation in which there are no infinite chains of "smaller" elements) using the **Fix** operator. This requires a binary relation, a proof that the relation is well-founded, and proofs that all recursive calls occur on instances smaller than the input.

Thus, we first define a relation adt_smaller : {s: sort & domain s} $\rightarrow$ {s: sort & domain s} $\rightarrow$ Prop encoding structural inclusion. In particular (ignoring typecasts), adt_smaller {s1, d1} {s2, d2} holds exactly when s1 and s2 are instances of the same mutual ADT m(vs) and when, if d2=c(args) for constructor c and arguments args, then d1 appears in args. We can prove that this relation is well-founded using adt_rep_ind, our generalized induction principle over adt_reps; we use the transitive closure, which is still well-founded.

Before defining the function, we first need to construct the input and return types; all needed arguments must be packed into a single type. The full recursive function must include the function

[4]Non-recursive functions and predicates satisfy the same specification but no longer have any circularity; we directly interpret them as their function bodies.

or predicate symbol and a proof that the symbol is in the mutual block (this allows us to define a family of functions and thus encode mutual recursion), a list srts, an hlist of function inputs of the appropriate type, a variable valuation, and a proof that the type variable valuation associates each function parameter with the corresponding element of srts. We call the fully packed type packed_args2; we lift the adt_smaller relation to this type first by giving a relation on hlists (checking that the elements at the decreasing index of each satisfy adt_smaller), and then lifting it through several additional packed arguments; we prove that this lifted relation is still well-founded. Our function's return type depends on the input: if given a function symbol, it returns an element of the corresponding domain of the function's return type; on a predicate symbol, it returns a bool.

Our function (funcs_rep_aux) works by defining nested versions of term_rep and formula_rep called term_rep_aux and formula_rep_aux; they describe how to interpret the term or formula using appropriate recursive calls to funcs_rep_aux. In the following, we discuss term_rep_aux, formula_rep_aux is very similar, but with a simpler return type.

The types of these functions are significantly more complex than term_rep — they must maintain many more invariants beyond typing. In particular, they keep track of small and hd (from the termination check), the proof that the term satisfies the check (decrease_fun), the fact that small and hd consist of ADTs, and the crucial invariants about small and hd valuations: the valuation of every variable in small is actually smaller than the original input, according to the transitive closure of adt_smaller, and hd's valuation is equal to the input. In the following, d is the input at the decreasing index.

```
Fixpoint term_rep_aux ... (Hty: term_has_type gamma t ty) (Hdec: decrease_fun fs ps small hd m vs t)
(Hsmall: ∀ x, In x small → vty_in_m m vs (snd x) ∧ adt_smaller_trans (hide_ty (v x)) d)
(Hhd: ∀ h, hd = Some h → vty_in_m m vs (snd h) ∧ hide_ty (v h) = d) ... := ...
```

This invariant is the key to defining this function: we establish a link between syntactic smallness (which involves keeping track of a list of variables) and the semantic notion based on structural inclusion of adt_rep (and thus of the underlying W-types).

The body of term_rep_aux is broadly similar to term_rep, with 2 main exceptions: the function application case and the proofs of invariant preservation. The invariant preservation is largely straightforward; the interesting case occurs when we add new variables to small when pattern matching. To show that the invariant is preserved, we must show that all of these variables evaluate to semantically smaller values (according to the transitive closure of adt_smaller):

```
Theorem match_val_single_smaller (vt: val_typevar) (v: val_vars pd vt) (ty: vty) (p: pattern)
  (Hp: pattern_has_type gamma p ty) (Hty: vty_in_m m vs ty) (d: domain (v_subst vt ty))
  (l: list (vsymbol * {s: sort & domain s})):
  match_val_single gamma_valid pd vt ty p Hp d = Some l →
  ∀ x y, In (x, y) l → In x (pat_constr_vars m vs p) → adt_smaller_trans y (hide_ty d)).
```

This theorem is difficult to prove, but it demonstrates that our syntactic check indeed aligns with the semantic notion that we intended; it also ties together our representations for ADTs, pattern matching, and the well-founded relation we need for our recursive functions. Thus, these features cannot be considered in isolation; for a complete semantics, we need to reason about the subtle interactions between these structures to ensure everything is well-defined.

The last and most important piece of the puzzle is to handle the actual call to funcs_rep_aux in the function application case of term_rep_aux. Namely, we must show that the arguments to this call are indeed smaller than the original input. The hlist argument comes from fun_arg_list: it is the result of recursively calling term_rep_aux on the list of arguments ts to the function call. From the termination condition, we know that the element of ts at the decreasing index i is a variable

```
Theorem funs_rep_spec (pf: pi_funpred gamma_valid pd) (l: list funpred_def)
  (l_in: In l (mutfuns_of_context gamma)) (f: funsym) (args: list vsymbol) (body: term)
  (f_in: In (fun_def f args body) l) (srts: list sort) (srts_len: length srts = length (s_params f))
  (a: hlist domain (sym_sigma_args f srts)) (vt: val_typevar) (vv: val_vars pd vt),
  funs_rep pf f l (fun_in_mutfun f_in) l_in srts srts_len a = cast ... (
  (*The function is the same as:*)
  term_rep gamma_valid pd
  (vt_with_args vt (s_params f) srts) (*setting the function params to srts,*)
  (pf_with_funpred pf l l_in) (*recursively using [funs_rep] and [preds_rep],*)
  (val_with_args _ _ (upd_vv_args pd vt vv (s_params f) srts (eq_sym srts_len)
    (s_params_Nodup _)) args a) (*setting the function arguments to a,*)
  body (f_ret f) (f_body_type l_in f_in)). (*and evaluating the body*)
```

Fig. 12. Recursive function API

x in small. By our invariant Hsmall, x's valuation is indeed smaller than the input; we can lift this through our well-founded relations to get the result we need.

Formalizing this argument in Coq is very tricky. First, we need a new version of fun_arg_list which accounts for the additional invariants; we call this get_arg_list_recfun. To use the Hsmall invariant we need to know that the ith element of get_arg_list_recfun evaluates to v x. This involves two pieces: showing that the ith element get_arg_list_recfun is term_rep_aux applied to the ith element of ts and that term_rep_aux evaluated on a variable x returns v x. But within the definition of term_rep_aux, we do not know this second fact; therefore we need to encode this information in the return type of term_rep_aux.

The first fact is even trickier, as a simple (transparent) proof fails to satisfy Coq's termination checker; Coq cannot tell that the call to term_rep_aux *within the termination proof* occurs on decreasing terms. Even inlining these proofs does not solve the issue; instead, we encode this proof into the return type of get_arg_list_recfun so that it returns both an hlist and a proof that, for every i within the correct bounds, the ith element is formed by term_rep_aux on the ith element of the input term list. At last, this allows Coq to prove that our function terminates.

The rest of the definition of funcs_rep_aux is not too complicated; it creates a valuation that sets the type and term variables to the srts and function body free variables, respectively, and then calls term_rep_aux and formula_rep_aux, typecasting the result as needed.

Abstraction Layer. The abstraction layer is significantly simpler than for ADTs; we need a little translation to pack the arguments appropriately and to use the assumptions from the well-typed context to populate all of the information needed, then a call to funcs_rep_aux. To find the decreasing indices we use our verified typechecker, as the typing rules only guarantee that such indices exist. With this, we have a complete definition for recursive functions, which has exactly the type we need for our pre-interpretation:

```
Definition funs_rep (pf: pi_funpred gamma_valid pd) (f: funsym) (l: list funpred_def)
  (f_in: funsym_in_mutfun f l) (l_in: In l (mutfuns_of_context gamma)) (srts: list sort)
  (srts_len: length srts = length (s_params f)) (a: hlist domain (sym_sigma_args f srts)):
  domain (funsym_sigma_ret f srts).
```

Recursive Function API. Figure 12 gives the only property required of our function representations: if we interpret all recursive functions as their representations, the function is equivalent to

```
Inductive even : nat → Prop :=
  | ev_0 : even 0
  | ev_SS: ∀ n, even n → even (S (S n)).
Lemma even_ind: ∀ (P : nat → Prop),
  P 0 → (∀  n : nat, even n → P n → P (S (S n))) → ∀ n : nat, even n → P 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),
  P 0 → (∀  n, P n → P(S (S n))) → ∀ m, even' m → P m.
Lemma even_constrs: even' 0 ∧ (∀  n, even' n → even' (S (S n)).
Lemma even_equiv: ∀ n, even n ↔even' n.
```

Fig. 13. Inductive predicate and impredicative representation for even

evaluating the body on the arguments. To show this, we prove that term_rep_aux and term_rep are equivalent as long as the pre-interpretation assigns recursive functions to funcs_rep_aux and the valuation is set appropriately. This theorem seems problematic: to use our function representation, we need to construct a pre-interpretation that already requires this representation to be defined; in order words, funcs_rep_aux would be defined in terms of itself. But we can break the circularity by proving another result: term_rep_aux is invariant under changes to the pre-interpretation that agree on all function and predicate symbols which are *not* part of the mutual block. Thus, we can define funcs_rep_aux in terms of an arbitrary pf, and then we can change pf to set the function symbol interpretations correctly without affecting our recursive definition. Once again, with this API, we never need to unfold the complicated underlying representation.

5.4 Inductive Predicates

Inductive predicates are significantly simpler to define than recursive types or functions, as we can take advantage of the impredicativity of Coq's Prop to use functions instead of inductive types. The specification we want is: for inductive predicate $\mathrm{p}(\alpha_1, \ldots, \alpha_k)(\tau_1, \ldots, \tau_m) = \mid f_1 \mid \ldots \mid f_n$ and sorts $\mathbf{s}$, $[\![p(\mathbf{s})]\!]$ is the least predicate such that $[\![f_1]\!]_v, \ldots, [\![f_n]\!]_v$ hold, where v maps $\boldsymbol{\alpha} \rightarrow \boldsymbol{s}$. We need to define an interpretation of $[\![p(\mathbf{s})]\!]$ satisfying two properties: under this interpretation, all $[\![f_i]\!]_v$ should hold and for any other predicate q that satisfies the constructors, for any arguments a, $[\![p(\mathbf{s})]\!](a) \rightarrow q(\mathbf{s})(a)$.

To encode this, we use an impredicative approach, similar to a Böhm-Berarducci [1985] encoding for types, where we represent an inductive predicate as a function in Coq encoding its induction principle. Figure 13 demonstrates this technique with the even property on nat. Defining even inductively generates the even_ind induction principle. Alternatively, we can encode even' in the following way: even' m holds whenever all propositions P : nat → Prop that satisfy the even constructors satisfy m. Note that this crucially relies on the impredicativity of Prop; this would not allow us to encode ADTs, since Coq's **Set** is not impredicative.[5] We can show that even' is correctly defined: we can prove the least predicate properties, and can prove it equivalent to even (the latter is not needed for the generic encoding, but it demonstrates that this approach defines the predicate we expect).

We generalize this approach to define arbitrary inductive predicates; Figure 14 shows the non-mutual case. The mutual case is very similar, but the P argument becomes an hlist of

[5]Impredicative **Set** is inconsistent with classical logic and indefinite description, which we use.

```
Definition indpred_rep_single (pf: pi_funpred gamma_valid pd) (p: predsym) (fs: list formula)
  (Hform: Forall (formula_typed gamma) fs) (srts: list sort)
  (a: hlist (domain pd) (sym_sigma_args p srts)) : bool :=
  all_dec (*Definition: For any possible P*)
  (∀ (P: ∀(srts: list sort), arg_list (domain pd) (sym_sigma_args p srts) → bool),
    (*If all of the constructors hold when p is interpreted as P*)
    iter_and (map is_true (dep_map (@formula_rep _ gamma_valid
      pd (mk_vt (s_params p) srts) (interp_with_P pf p P) (mk_vv _))
      fs Hform)) → (*Then P holds of a*) P srts a).
```

Fig. 14. Representation of (non-mutual) inductive predicates

∀(p: predsym) (srts: list sort), hlist (domain pd (sym_sigma_args p srts)) → bool over the list of predicates; i.e., an arbitrary property of each predicate in the block.

Proving the least predicate property is trivial; it follows immediately from our definition. However, showing that the constructors are satisfied is much harder. We give a proof sketch:

We denote $[\![f]\!]_{(p \to P; v)}$ as the interpretation of f under valuation v when p is interpreted as P. We let I represent indpred_rep p. First, we prove that every constructor f_i of predicate p can be rewritten into a special form: $\forall \mathbf{x},\ \text{let } \mathbf{y} = \mathbf{t} \text{ in } (g_1 \wedge \ldots \wedge g_k) \to p(\mathbf{z})$ — this follows from the special grammar in the typing rules and the definition of term_rep and formula_rep. With this form, we need to prove $[\![p(\mathbf{z})]\!]_{(p \to I; v)}$, assuming $[\![g_j]\!]_{(p \to I; v)}$, where v is formed by assigning correct values to $\boldsymbol{x}$ and $\boldsymbol{y}$. By the definition of indpred_rep, we need to prove that for any P, if $[\![f_j]\!]_{p \to P}$ for all j, then P holds of $\mathbf{z}$. In particular, we have assumed $[\![f_i]\!]_{p \to P}$, so by again rewriting f_i into its special form, we see that it suffices to prove $[\![g_j]\!]_{p \to P; v}$ for all j. We already assumed $[\![g_j]\!]_{p \to I; v}$. We can prove that, for any formula g, if p appears strictly positively in g, then $[\![g]\!]_{p \to I}$ implies that $[\![g]\!]_{p \to P}$ for any P. This completes the proof.

Unsurprisingly, positivity is crucial in ensuring that the least predicates exist for a given definition. We again prove that our syntactic typing rules are sufficient to construct objects satisfying the intended properties. To complete our proofs of the API properties, we need a result that lets us change the interpretation for each predicate symbol p in the mutual block; just as with recursive functions, this allows us to reason about indpred_rep in the context in which all inductive predicates are already assigned to their representations.

6 PUTTING IT ALL TOGETHER: THE LOGIC OF WHY3

We now define a *full interpretation* — a pre-interpretation that is consistent with the APIs for recursive functions and predicates and inductive predicates. We prove that for any possible assignment to uninterpreted type, function, and predicate symbols, there is a full interpretation agreeing with this initial assignment.

Then we define the standard logical notions: a closed formula f is satisfied by a full interpretation ($I \vDash f$) if for any possible valuation, $[\![f]\!]_v$ holds (note that the context Γ is implicit; we will write it when needed). For a set Δ, we denote $I \vDash \Delta$ to mean that $I \vDash d$ for all $d \in \Delta$. A formula is *valid* ($\vDash f$) if every full interpretation satisfies it, and a set of formulas Δ *logically imply* f ($\Delta \vDash f$) if, for any full interpretation I, whenever $I \vDash \Delta$, $I \vDash f$. We can then prove metatheorems about the logic:

Theorem 6.1 (log_conseq_equiv). *If f is monomorphic (for instance, by replacing type variables with fresh type constants), then $\Delta \vDash f$ iff the set $\{\neg f\} \cup \Delta$ is unsatisfiable.*

Theorem 6.2 (consistent). *For any full interpretation I and formula f, it is not the case that both $I \vDash f$ and $I \vDash \neg f$.*

Theorem 6.3 (semantic_lem). *For any full interpretation I and monomorphic formula f, $I \vDash f$ or $I \vDash \neg f$.*

Theorem 6.4 (semantic_deduction). *For any set of formulas Δ and monomorphic formulas f and g, $\Delta \vDash f \rightarrow g$ iff $\{f\} \cup \Delta \vDash g$.*

These metatheorems check that we defined our logic appropriately (for example, that we did not allow a formula and its negation to be true) and will be useful for our proof system.

Next, we formalize Why3 *tasks* and *theories* (§2). A task consists of a context Γ, a set of local assumptions Δ, and a goal f; a task is *well-formed* if Γ is a valid context, Δ consists of well-typed formulas, and f is a closed, monomophric, well-typed formula. A task is *valid* if $\Gamma, \Delta \vDash f$.

Recall that Why3 translates formulas to formats supported by solvers by applying a series of *transformations*, functions of type task $\rightarrow$ list task. A transformation is *sound* if, whenever all of the output tasks are valid, so is the input task. This formalizes the notion that if we transform task t to tasks $t_1, \ldots, t_n$, it suffices to prove these output goals correct. In §7.3, we prove that two of Why3's transformations are indeed sound according to our semantics.

Additionally, we provide a limited implementation of Why3 theories; we require all clones to be exported, and we view theories as a preprocessing step to create the context Γ and local definitions Δ; we do not keep track of theory ancestry as Why3 does. Nevertheless, this allows us to handle the abstraction provided by theories appropriately. Defining and writing functions over theories is a bit tricky: the natural definition does not lead to structurally recursive functions, so we use Equations. The preprocessing is also fairly complex: for each theory, we need its internal and external context (as not all used theories are exported); this depends on the external contexts of previously-defined theories. To define the contexts, we need to qualify names and instantiate abstract parameters. The last function we need finds the tasks for a given theory — for every lemma or goal, there is an associated task consisting of the previously-defined context (included the exported contexts of any imported theories), the local lemmas and axioms (and the exported lemmas and axioms from imported theories), and the monomorphized goal. Then, we can prove a theory valid by proving each of its tasks valid.

7 EVALUATION

We want to validate that our formal semantics are *correct* and *useful.* To do this, we apply the semantics in several ways: we give a natural-deduction-style proof system and prove soundness, use this proof system (and a derived tactic library) to prove goals from Why3's standard library, and prove the soundness of two transformations Why3 uses when translating to simpler logics.

To evaluate the correctness of our semantics, we first note that we closely match the intended semantics [Filliâtre 2013]. Additionally, the fact that we can prove the standard natural deduction rules and can prove Why3 goals using this proof system (with proofs that follow the structure we expect) provides strong evidence that our semantics closely align with the intended meaning of the various connectives and features in the logic.

These applications also show that the semantics are useful; they admit a standard proof system and a more usable tactic-based interface, they allow us to prove naturally occurring Why3 goals, and, most importantly, we can prove Why3 transformations sound; this would allow us to write a version of Why3 proved correct according to these semantics.

7.1 A Sound Proof System in Coq

Proving goals directly from the semantics is tedious and unintuitive; we want a proof system with which to prove validity indirectly. We will say that a task (Γ, Δ, f) is derivable $(\Gamma, \Delta \vdash f)$ if we can prove f using assumptions Δ in context Γ. To define when a task is derivable, we could give all of the standard natural deduction rules - introduction and elimination rules for each connective, a rule to define classical logic (e.g. double negation elimination), rules for handling equality, and rules for working with the recursive definitions. Instead, we take a simpler approach; we give a single rule: if transformation tr is sound whenever a task satisfies P, if t satisfies P, and if all the outputs of tr t are derivable, then so is t:

```
Inductive derives: task → Prop :=
  | D_trans: ∀ (tr: trans) (t: task) (l: list task) (P: task → Prop) (Hp: P t), (*t satisfies P*)
    task_wf t → (*If t is well-formed*)
    soundif_trans P tr → (*If tr is sound on all tasks satisfying P*)
    (∀ x, In x (tr t) → derives x) → (*And all outputs of tr are derivable*)
    derives t. (*Then t is derivable*)
Theorem soundness (t: task): derives t → task_valid t.
```

The soundness of this proof system is immediate. With this single definition, we can prove all of the standard natural deduction rules. For instance, suppose we want to prove the and-introduction rule. We define the transformation (Γ, Δ, f ∧ g) ⇒ [(Γ, Δ, f); (Γ, Δ, g)]. Proving the soundness of this transformation requires us to show that if $\Gamma, \Delta \vDash f$ and $\Gamma, \Delta \vDash g$, then $\Gamma, \Delta \vDash f \wedge g$, which is easy. Then, we can prove the and-intro rule:

```
Lemma D_andI gamma delta f g:
  derives (gamma, delta, f) → derives (gamma, delta, g) → derives (gamma, delta, f ∧ g).
```

We prove the introduction and elimination rules for all connectives, as well as properties of equality (that it is an equivalence relation and a congruence), the hypothesis rule, and some context-manipulating rules (weakening, reordering and renaming). Finally, we give rules dealing with function definitions, ADTs and pattern matching, including induction on ADTs (see §7.2).

We note that these transformations should be considered distinct from the Why3 transformations we discuss in §7.3. Though some of these have Why3 analogues, here we focus on writing simple transformations to prove our proof rules; we do not attempt to be faithful to Why3 or to do any other simplification. Nevertheless, our derivation rule allows to replace these with more complicated transformations in the future, enabling use of complex Why3 transformations in our proof system.

7.2 Proving Why3 Goals in Coq

We would like to ensure that our semantics are capable of proving real Why3 goals. To that end, we translated parts of Why3's standard library into Coq. Our implementation of theories automatically produces the associated proof goals; we use the soundness theorem of our proof system to give a derivation instead of using the semantics directly. Since proofs in pure natural deduction can be tedious, we designed a small tactic system on top of the proof system; we include version of Coq's intros, rewrite, apply, assumption, specialize, assert, reflexivity, symmetry, f_equal, clear, destruct, split, left/right, and exfalso. In addition, we include 3 tactics based on recursive structures: an unfold tactic to replace a function or predicate application with the function body with substituted type and term arguments, a simpl_match tactic to simplify pattern matches by identifying the matching case and substituting in the bound terms, and an induction tactic to perform induction over (non-mutual) ADTs. The correctness of each of these follows from our recursive structure APIs: unfold follows from our spec for recursive function and predicate definitions, we prove simpl_match correct by showing

```
lemma append_assoc: ∀ l1 l2 l3: list 'a, l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
lemma append_length: ∀ l1 l2: list 'a, length (l1 ++ l2) = length l1 + length l2
lemma mem_append: ∀ x: 'a, l1 l2: list 'a, mem x (l1 ++ l2) ↔mem x l1 ∨ mem x l2
lemma mem_decomp: ∀ x: 'a, l: list 'a, mem x l → ∃ l1 l2, l = l1 ++ Cons x l2
lemma reverse_reverse: ∀ l: list 'a, reverse (reverse l) = l
lemma reverse_mem: ∀ l: list 'a, x : 'a, mem x l ↔mem x (reverse l)
lemma reverse_length: ∀ l: list 'a, length (reverse l) = length l
lemma inorder_length: ∀ t: tree 'a, length (inorder t) = size t
```

Fig. 15. Selection of theorems from Why3 Standard Library proved correct in Coq

that a (conservative) syntactic pattern match simplification is consistent with match_val_single, and induction follows from adt_rep_ind.

We then use these tactics to prove Why3 goals; these proofs end up quite similar (albeit with less automation) to the corresponding proof in Coq.[6] In particular, we translated parts of Why3's relations, algebra, int, option, list, and bintree libraries,[7] proved that these theories are all well-typed, and proved the validity of the Append and Reverse theories for lists and the InorderLength theory for binary trees. See Figure 15 for a selection of theorems proved. These theorems, and the definitions they depend on, involve extensive use of theory using and cloning, polymorphism, induction over ADTs, recursive functions, and pattern matching. This gives us additional confidence in the correctness of our semantics; we can prove these results in the way we expect, suggesting that our semantics truly capture the intended meaning of the connectives and recursive structures.

7.3 Why3 Transformations

These semantics are a crucial first step in producing a verified version of Why3; the next step consists of verifying the transformations that Why3 performs on tasks to produce simpler goals amenable to automated solvers. Thus, we demonstrate that our semantics are indeed capable of proving real Why3 transformations sound. In particular, we translate the Why3 transformations eliminate_let and eliminate_inductive from OCaml to Coq as faithfully as possible[8] and prove soundness.

Soundness of eliminate_let. This is an extremely simple function; the only interesting case is for let-binding (sub_t t1 x t2 safely substitutes t1 for x in t2):

```
eliminate_let (Tlet t1 x t2) = eliminate_let (sub_t (eliminate_let t1) x t2)
```

Proving this sound is almost trivial; it follows from the correctness of substitution (§5.2). But *defining* it in Coq is challenging, as this is not structurally recursive — substitution can cause terms to blow up in size (as with β reduction in the lambda calculus). In fact, there is no obvious decreasing measure — instead, we use a lexicographic measure consisting of the number of let-bindings in the term or formula, followed by the size of the term, and we use the Equations package to handle the well-founded definitions. This measure is sufficient; in the Tlet or Flet case, the number of let-bindings decreases (even as the size gets bigger); in all other cases the number of let-bindings is the same or smaller while the size of the argument to the recursive call decreases. We note that this argument critically relies on the evaluation order: we need the fact that the term we substitute in

[6]Note that one typically proves Why3 goals with automated solvers, this functions primarily as a test for our semantics rather than a primary interface for a future verified Why3.
[7]https://why3.lri.fr/stdlib/
[8]In eliminate_let, the only difference is in our substitution function; we handle variable binding differently. In eliminate_inductive, we additionally do not implement a separate simplification for quantified goals.

(eliminate_let t1) has no let-bindings. Accordingly, our function needs a dependent return type; for the termination proof we need to know that the recursive result has no let-bindings. But we prove that our final transformation is equal to a transcribed Why3 version. This complication suggests that verifying even simple transformations is worthwhile: slight changes in this function could lead to nontermination.

Soundness of eliminate_inductive. In contrast, eliminate_inductive is relatively easy to define but much harder to prove sound. This transformation removes all inductive predicate definitions from the context, replacing them with uninterpreted predicate symbols and adding axioms asserting that the constructors hold and that inversion holds. For example, the even predicate (§5.4) produces 3 axioms: even(0), ∀ n, even(n) → even (S (S n)), and ∀ x, even(x) → x = 0 ∨ ∃ n, even(n) ∧ x=S(S(n)).

To prove this sound, we split it into two transformations: one that adds the axioms, and one that changes the context. Then, we show that it is sufficient to prove each sound. The second is straightforward; for the first, we must show that all axioms are true in the original context. We give a version of the proof specialized to the even predicate; the full proof is a complex generalization.

The constructor axioms are easy; we already proved that they hold of any full interpretation. The inversion axioms are the difficult part. The least predicate property of even says that, for any P: nat → Prop, if P 0 and ∀ n, P n → P (S(S n)) hold, then, ∀ x, even x → P x. Thus, we let P(x) = (x = 0 ∨ ∃ n, even n ∧ x = S (S n)), and we must show that the constructors are satisfied. Clearly P 0 holds. The second case is more interesting; we must show that for any n, assuming P n holds, then P (S(S n)) holds. We use the second case of P, giving S(S n) to instantiate the quantifier; we must prove that even S (S n) and S (S n) = S (S n). The second is trivial; to prove the first, we use the fact that all constructors for even hold in any full interpretation; thus, we use the second constructor and need to show that even n holds. Therefore, we must show that P n implies even n. Since P n holds, either n=0 or ∃ m, even m ∧ n = S (S m). In each case, substituting appropriately, the result follows by the fact that the constructors for even hold.

This transformation provides a good demonstration that our semantics can indeed handle the complex reasoning involved in proving many of Why3's transformations correct. Separately, we note that between our earlier encoding of inductive predicates and our proofs of the inversion lemmas, we have completed an impredicative construction of inductive predicates in higher-order logic. While the formula_rep is specialized to Why3, these results could be extended fairly easily to work in other contexts (see §7.5).

More broadly, we have now demonstrated applications of our semantics for recursive types, pattern matching, recursive functions, and inductive predicates, giving us confidence that our encodings are correct; the APIs with which we proved them against are powerful enough to use these structures in the way that we expect.

7.4 Comparing Why3 to Our Semantics

Our formalization aims to capture "core" Why3, which roughly aligns with Filliâtre [2013]. Nevertheless, there are several features that we do not include or that we handle significantly differently:

- We do not include function types or lambdas, which are not part of core Why3 (lambdas are encoded using the epsilon operator), but are handled using special cases in the Why3 tool. Extending our semantics with function types would be possible; the primary challenge would be in generalizing our encoding of ADTs.
- We do not include coinductive predicates. We expect they could be added with a similar, but dual, encoding to that used for inductive predicates.
- As discussed in §5.1, we include only a subset of allowed ADTs. In particular, we do not support nested or non-uniform ADTs.

- Our formalization of Why3's type system does not include a check for pattern matching exhaustiveness. Since all Why3 types are inhabited, this does not introduce soundness problems; we can just return a default element.
- Our recursive functions use a different termination metric (simple structural recursion rather than lexicographic ordering) and we only support uniform functions and (recursive and inductive) predicates.

We also do not include components that are outside of core Why3, though they still lie in the logic language: tuples, records, type aliases (all of which are derived from ADTs), range types, and float types (these could be added and treated similarly to int and real). Finally, in Why3, one can mix logical specifications and executable code, as pure WhyML functions can be used in specifications, while logical functions and predicates can be used in ghost code. These pure functions can include annotations with decreasing arguments (both ADTs and integer measures). Since we do not model WhyML, we do not (yet) include any of these features.

However, our semantics does capture a large fragment of Why3's logic language; we note that most of Why3's standard library (if pure program functions are translated to their logical function counterparts) fits comfortably within this fragment. In particular, all 9 inductive predicates and 12/13 ADTs in the library are within this fragment (the remaining ADT is a nested type; we could encode this as a mutual ADT instead). We also estimate that we could formalize approximately 85% of the pure function and predicate definitions in the standard library; the others use function types and/or pure WhyML code with an integer termination measure. Thus, our formalization captures a very practical subset of Why3, and it is unclear to what extent some excluded features like non-uniform types and coinduction are used in practice.

Nevertheless, the parts of Why3's logic language that we did not include could compromise soundness and violate our proofs of consistency (§6) and the fact that interpretations consistent with the recursive structures exist. In particular, we note the potential for issues in the more general inductive types supported by Why3 (including non-uniform types and those with function arguments that require a check for strict positivity) and the more sophisticated lexicographic termination check. We plan to formalize much of this in future work, see §9. Beyond core Why3, there is the potential for unsoundness in the interaction between logical and WhyML code, as the syntax and semantics of (pure) WhyML functions are different from those of logical functions (for example, the integer termination measures in WhyML functions have no direct analogue in the logic fragment).

7.5 Reusability and Axioms

Finally, we evaluate the potential utility of our semantics in verifying the semantics of non-Why3 logics with similar expressivity (Dafny, VeriFast, Cryptol, etc). In particular, our W-type and ADT implementation is independent of Why3 definitions and could be reused more generally. In principle, our inductive predicate encoding could be refactored fairly simply to eliminate dependence on Why3; the proofs about the least predicate and inversion properties would hold. Our recursive function encoding, on the other hand, is unlikely to be useful for other projects, as it is tightly wedded to details of pattern matching and Why3 typing rules.

Separately, one could imagine performing a similar formalization for a stronger logic like the Calculus of Inductive Constructions. MetaCoq [Sozeau et al. 2020] (see §8) provides syntax and typing rules, but building a denotational model for PCUIC (the logic formalized for MetaCoq) would require additional effort and axioms beyond our work. Gödel's incompleteness theorem ensures that one cannot write a model of CIC inside Coq without additional axioms. MetaCoq also needs additional axioms, but it is telling as to the difference in goals to see what axiom they assume:

strong normalization for the theory they model – an axiom about the reduction theory. Instead, a denotational model would need a model-theoretic axiom, such as the existence of a suitable large cardinal.

The axioms we use, which also affect reusability, are based primarily on the classical nature of Why3; we use classical logic (LEM + Hilbert choice) because Why3's logic has these features, but our proofs do not rely on classical reasoning beyond this. Under the above axioms, Prop and bool are essentially equivalent, so we use them interchangeably, but if one wanted a semantics for an intuitionistic version of Why3 (or some other logic), we could construct a very similar formalization using only Prop. To that end, we consciously restricted the use of classical axioms in 3 ways:

(1) We did not assume classical axioms in the W-type and ADT encoding (though we do need UIP and functional extensionality).
(2) We proved (axiom-free) decidability for many predicates (e.g. typechecking, finding the decreasing index for recursive functions), even when only used in contexts where Hilbert's epsilon is assumed.
(3) When possible, we used boolean predicates rather than Prop (in contexts where the two are not equivalent) to get axiom-free proof irrelevance.

8 RELATED WORK

Verification of Deductive Verifiers. Our work aims to bridge the gap between automated and foundational program verification tools; that is, to enable automated but provably-sound tools. There is a long line of work in this space, beginning with verifying verification condition (VC) generators for small languages. Homeier and Martin [1995] write a verified VC generator in HOL for a simple while-loop-based language. Vogels et al. [2010] prove correct a more efficient VC generator for a Boogie-like language in Coq and verify some source-to-source transformations on the input program. Frade and Sousa Pinto [2023] verify a VC generator for dynamic logic based on KeY. They write and verify this generator using Why3 (using the logic fragment to specify their logic and WhyML to implement the program); our work could enable this approach to have the same guarantees as efforts using Coq or Isabelle. More extensive is Featherweight VeriFast [Jacobs et al. 2015], which formalizes and proves sound in Coq a core subset of the VeriFast [Jacobs et al. 2011] tool for verifying C and Java programs. It operates at a different level of the abstraction hierarchy than our work, focusing on memory safety conditions and separation logic predicates at the source language level. Featherweight VeriFast omits recursive functions and inductive datatypes (both of which are included in VeriFast's specification language) and does not verify function termination. In principle, it would be possible to compile VeriFast to Why3 rather than SMT directly and therefore to integrate Featherweight VeriFast (for separation logic and memory safety properties) with our semantics (for the recursive structures). Finally, Parthasarathy et al. [2021] enable sound use of Boogie, an alternative to Why3, creating a tool that generates certificates in Isabelle to validate successful runs of some Boogie transformations. Along the way, they formalize Boogie in Isabelle; the logic of this language is much simpler than Why3's.

Combining Automated and Foundational Verifiers. Other recent efforts have combined automated reasoning with foundational guarantees beyond VC generation. SMTCoq [Ekici et al. 2017] allows one to call SMT solvers within Coq to prove goals; it checks the solver's certificate to ensure that the process is sound. RefinedC [Sammler et al. 2021] provides automated but foundational reasoning about C programs in Coq; it does not use SMT solvers but rather a fragment of separation logic that enables automatic search without backtracking. Similarly, VST-A [Zhou et al. 2024] augments VST with annotations. It decomposes the verification problems into symbolic execution of straightline Hoare triples, reducing the burden on the user while retaining foundational guarantees.

Formalized Logic and Proof Assistants. Some work on formalizing logic in proof assistants aims at formalizing deep metatheorems of first-order logic, such as completeness [Forster et al. 2020] and incompleteness [O'Connor 2005]. Other recent work has focused on formalizing the much richer logics used in proof assistants. Barras [2010] builds a denotational model for much of the Calculus of Inductive Constructions inside the Coq proof assistant while [Anand and Rahli 2014] builds a PER model for Nuprl also in Coq. Nipkow and Roßkopf [2021] formalize the higher-order logic used in the Isabelle proof assistant and give a verified proof checker, while Candle [Abrahamsson et al. 2022] is a fully verified HOL Light implementation written in CakeML.

Arguably the most sophisticated of these efforts is MetaCoq [Sozeau et al. 2020], which formalizes Coq's syntax and type system within Coq itself. MetaCoq includes a deep embedding of the syntax, typing rules, and operational semantics (reduction) for an extended version of the Calculus of Inductive Constructions. In many ways, our syntax and typing rules can be seen as simplified versions of MetaCoq's, with similar conditions for objects like inductive types (e.g. positivity). However, the MetaCoq project's goals are orthogonal to ours. It takes a syntactic approach and proves metatheoretic results about the (much more complicated) type system (e.g. subject reduction, consistency assuming strong normalization). Meanwhile, we give a model of Why3's logic (our denotational semantics) and aim to enable reasoning about semantics-preserving transformations on Why3 terms and formulas. Since MetaCoq focuses on typing and reduction rules, it does not construct recursive structures (types, inductive predicates, pattern matching) as we do. Moreover, one of the trickiest parts of our work, dealing with termination of recursive functions, is omitted from MetaCoq completely; it leaves the termination check abstract, since strong normalization is assumed (rather than in our work, where we have to prove that the Why3 termination check leads to a well-typed Coq term). We plan to use MetaCoq in future extensions of our work, see §9.

Verification of Why3. Herms et al. [2012] develop a verified implementation of an older version of Why3 (called Why) in Coq, including a verified VC generator for a subset of what is now WhyML. Their semantics are similar to ours in §4. However, their formalization only includes the core first-order logic; it does not include ADTs, pattern matching, recursive functions, inductive predicates, or the epsilon operator. It focuses on proving the VC generation correct and it does not reason about lower-level transformations on the terms and formulas to enable automated solving. More recently, Garchery [2021] develops a certificate-based approach to validate Why3 transformations and writes a proved-correct certificate checker in the Lambdapi/Dedukti proof assistant. The certificates are based only on a polymorphic first-order logic, similarly without pattern matching or recursive structures, and the semantics are based on a shallow embedding in Dedukti. In principle, these certificates could be useful in developing a formalized Why3 implementation, as we could use similar certificates and checkers, proving correctness against our semantics.

9 CONCLUSION AND FUTURE WORK

We have presented a formal semantics for the logic fragment of Why3, a high-level logic for program specification that augments first-order logic with polymorphism, algebraic data types, pattern matching, recursive functions and predicates, and inductive predicates. We validated our semantics by giving a proof system, proving Why3 goals from the standard library correct according to these semantics, and proving two Why3 transformations sound. These semantics are a crucial first step in enabling foundational tools that can take advantage of solver automation, and can be extended in a number of directions. First, we would like to add currently missing features from Why3 to our semantics, including function types and lexicographic orderings for termination. Second, we would like to generalize our semantics to allow different ADT representations and then to use MetaCoq to automatically prove that a provided ADT implementation (e.g. Coq list) satisfies the properties required by our semantics (injectivity of constructors, induction, etc). This would allow idiomatic Coq proofs of Why3 goals rather than using our deeply embedded proof system and W-type encoding; this would be faster and would generate much smaller proof terms. Finally, we would like to prove more Why3 transformations correct, producing a complete implementation that is connected to a tool such as SMTCoq to enable fully automated yet verified verification of Why3 goals. Such an implementation would make it much easier to build foundationally verified, automated program verifiers, much as Why3 itself has made it easier to build verifiers in general.

DATA AVAILABILITY STATEMENT

Our Coq formalization and proofs are available at github.com/joscoh/why3-semantics/tree/popl-24 as well as in our artifact [Cohen and Johnson-Freyd 2023].

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and feedback. This research was supported in part by National Science Foundation grant CCF-2219757.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES

Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. 2022. Candle: A Verified Implementation of HOL Light. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:17. https://doi.org/10.4230/LIPIcs.ITP.2022.3

AdaCore and Altran UK Ltd. 2018. *SPARK 2014 User's Guide: Release 19.0w.*

Abhishek Anand and Vincent Rahli. 2014. Towards a Formally Verified Proof Assistant. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 27–44. https://doi.org/10.1007/978-3-319-08970-6_3

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press. https://doi.org/10.1017/CBO9781107256552

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

Bruno Barras. 2010. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning* 3, 1 (Jan. 2010), 29–48. https://doi.org/10.6092/issn.1972-5787/1695

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO 2011*, Phillip Rogaway (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wroclaw, Poland, 53–64. https://inria.hal.science/hal-00790310

Corrado Böhm and Alessandro Berarducci. 1985. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science* 39 (1985), 135–154. https://doi.org/10.1016/0304-3975(85)90135-5

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

Joshua M. Cohen and Philip Johnson-Freyd. 2023. *POPL Artifact for "A Formalization of Core Why3 in Coq"*. https://doi.org/10.5281/zenodo.8417774

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 126–133. https://doi.org/10.1007/978-3-319-63390-9_7

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 604–619. https://doi.org/10.1145/3453483.3454065

Jean-Christophe Filliâtre. 2013. One Logic to Use Them All. In *Automated Deduction – CADE-24*, Maria Paola Bonacina (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/978-3-642-38574-2_1

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Yannick Forster, Dominik Kirst, and Dominik Wehr. 2020. Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory. In *Logical Foundations of Computer Science*, Sergei Artemov and Anil Nerode (Eds.). Springer International Publishing, Cham, 47–74. https://doi.org/10.1007/978-3-030-36755-8_4

Maria João Frade and Jorge Sousa Pinto. 2023. A verified VCGen based on dynamic logic: An exercise in meta-verification with Why3. *Journal of Logical and Algebraic Methods in Programming* 133 (2023), 100871. https://doi.org/10.1016/j.jlamp.2023.100871

Quentin Garchery. 2021. A Framework for Proof-carrying Logical Transformations. In *Proof eXchange for Theorem Proving (Electronic Proceedings in Theoretical Computer Science, Vol. 336)*. EPTCS, Virtual, United States, 5–23. https://doi.org/10.4204/EPTCS.336.2

Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (Jan. 2010), 95–152. https://doi.org/10.6092/issn.1972-5787/1979

Paolo Herms, Claude Marché, and Benjamin Monate. 2012. A Certified Multi-prover Verification Condition Generator. In *Verified Software: Theories, Tools, Experiments*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–17. https://doi.org/10.1007/978-3-642-27705-4_2

P. V. Homeier and D. F. Martin. 1995. A mechanically verified verification condition generator. *Comput. J.* 38, 2 (01 1995), 131–141. https://doi.org/10.1093/comjnl/38.2.131

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* Volume 11, Issue 3 (Sept. 2015). https://doi.org/10.2168/LMCS-11(3:19)2015

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Cláudio Belo Lourenço and Jorge Sousa Pinto. 2022. Why3-do: The Way of Harmonious Distributed System Proofs. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 114–142. https://doi.org/10.1007/978-3-030-99336-8_5

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*, L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2

David Monniaux and Sylvain Boulmé. 2022. The Trusted Computing Base of the CompCert Verified Compiler. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 204–233. https://doi.org/10.1007/978-3-030-99336-8_8

Tobias Nipkow and Simon Roßkopf. 2021. Isabelle's Metalogic: Formalization and Proof Checker. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 93–110. https://doi.org/10.1007/978-3-030-79876-5_6

Russell O'Connor. 2005. Essential Incompleteness of Arithmetic Verified by Coq. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 245–260. https://doi.org/10.1007/11541868_16

Chris Okasaki. 1996. *Purely functional data structures.* Ph. D. Dissertation. Carnegie Mellon University.

Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 704–727. https://doi.org/10.1007/978-3-030-81688-9_33

Mário Pereira and António Ravara. 2021. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 677–689. https://doi.org/10.1007/978-3-030-81688-9_31

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64, 5 (jun 2020), 947–999. https://doi.org/10.1007/s10817-019-09540-0

Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (jul 2019), 29 pages. https://doi.org/10.1145/3341690

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2010. A Machine-Checked Soundness Proof for an Efficient Verification Condition Generator. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 2517–2522. https://doi.org/10.1145/1774088.1774610

Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proceedings of the ACM on Programming Languages* 8, POPL (2024).

Received 2023-07-11; accepted 2023-11-07