A Foundationally Verified Intermediate Verification Language

Joshua M. Cohen

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Andrew W. Appel

May 2025

Abstract

Annotation-based deductive verifiers have emerged in recent years as practical, capable, and increasingly scalable tools for verifying programs in languages including C, Rust, Go, OCaml, and Dafny. Intermediate verification languages (IVLs) like Why3, Boogie, and Viper have made it significantly easier to build such verifiers by performing part of the translation from high-level, expressive program logics to SMT formulas once and for all. But these verifiers are difficult to fully trust: their toolchains comprise hundreds of thousands of lines of code and bugs in any part can result in unsoundness. An alternate approach to building program verifiers is the foundational one, where the language semantics, program logic, and user-supplied proofs are defined in a proof assistant, giving an end-to-end soundness theorem. These tools, including VST, CakeML, and Bedrock2, provide very strong guarantees but require manual proof and are inaccessible to non-experts. The two approaches have largely remained separate. IVL-based tools are not themselves verified, while interactive tools cannot take advantage of the automation provided by the IVL and SMT-based pipelines.

In this thesis, we take steps towards bridging this gap by providing an implementation of the Why3 IVL in the Coq proof assistant, giving the first foundationally verified IVL. First, we give a novel formal semantics for a core subset of the logic implemented by Why3 – polymorphic first-order logic with recursive types, functions, predicates, and pattern matching. We prove sound a compiler from this logic to first-order logic, giving the first machine-checked soundness proofs for a sophisticated pattern matching compiler and a first-order axiomatization of algebraic data types. We then develop a new framework to idiomatically implement OCaml APIs in Coq and use this to produce a Coq implementation of the Why3 logic API. Our IVL implementation is fully executable both within Coq and via extraction to OCaml. Using the latter, we demonstrate our tool's practicality by running existing Why3-based tools and test suites against it, while the former allows our tool to serve as a future back-end for foundational verifiers such as VST.

Acknowledgments

I could never have completed this PhD alone, and I am grateful to everyone who helped me in my journey.

First, I want to thank my adviser Andrew Appel for all his advice and assistance along the way. His door was always open, and I greatly enjoyed our conversations and the opportunity to learn about all the interesting problems he was working on, from Coq proofs to voting machines. I also truly appreciate his flexibility as my PhD evolved in directions that neither of us could have expected.

I could not have completed this project without the help of Philip Johnson-Freyd, my mentor and collaborator at Sandia National Laboratories. Our weekly meetings were always illuminating, and I learned so much about different areas of PL theory, logic, and real-world program verification.

I would also like to thank the members of the Princeton PL faculty: Aarti Gupta for her support throughout my PhD, including when I attended my first conference, Zachary Kincaid and David Walker for serving on my committee, and Lennart Beringer for his mentorship and advice.

I am very fortunate to have met so many great fellow students during my time in grad school. I want to thank Andrew Johnson for keeping me sane through our many conversations about football, Costco, and everything else. I am also grateful to the other Princeton PL group members for making my PhD experience collaborative and enjoyable, especially Nick, Nikhil, Joomy, Akash, Shaowei, and Qinshi. Beyond the PL group, I want to thank Calvin, Dario, and Megan for helping me survive the early COVID years and for our subsequent game and movie nights afterwards.

My time at Princeton, and my life a whole, was enormously improved after I met Claire in my second year. I am very grateful that we could embark on and finish our PhD journeys together, and I could not have completed mine without her. She always believed in and supported me, especially during the stressful final months of my PhD, and she continually motivated me to finish my thesis even when the hurdles seemed insurmountable.

Lastly, I would like to thank my parents Lisa and Jeff, my sisters Lauren and Lily, my grandparents, and the rest of my family for their support and assistance throughout my academic journey. I am especially thankful that my parents always emphasized my education and encouraged me to explore my academic interests.

This research was supported in part by National Science Foundation grant CCF-2219757.

Contents

Chapter 1

Introduction

Writing correct software is a critical but very difficult task. Program bugs are common, leading to crashes, incorrect results, and security vulnerabilities. Accordingly, researchers have developed a variety of techniques to detect and reduce bugs, including type systems, static analysis, and abstract interpretation. In many settings, even these tools are not sufficient; we want a guarantee of *functional correctness*: that a program satisfies a particular specification, written in a formal logic, for any possible input. This allows one to prove strong correctness properties about programs, especially important in safety-critical settings.

Formal verification tools allow users to specify and prove such correctness properties. Rice's Theorem implies that program correctness is undecidable, so fully automated verification for arbitrary programs is impossible. Nevertheless, the increasing power of Satisfiability Modulo Theories (SMT) solvers [16] and proof assistants has kickstarted the development of practical program verifiers for real-world languages. These tools fall into two categories. *Interactive* verifiers are built within proof assistants; the user writes down a specification in the theorem prover and manually proves their program correct. Meanwhile, *semi-automated* verifiers require some input from the user – generally in the form of function pre- and post-conditions, loop invariants, and assertions – and generate *verification conditions* (VCs) from the annotated program, sending these logical formulas to a solver (generally an SMT solver) to determine validity.

In these semi-automated tools, the source program logic should be expressive enough to permit natural, rich program specifications, but this leads to a large gap between the source logic and the much simpler logic supported by SMT solvers. Much of this translation is tricky but not language- or solver-specific. *Intermediate verification languages* (IVLs) provide a solution: they handle part of this translation once and for all, enabling multiple source-language verifiers to target a higher-level interface than SMT formulas and thus greatly reduce the burden of creating such *translational* verifiers. Widely used IVLs include Why3 [24], Boogie [13], and Viper [83]. In recent years, this approach has proved extremely fruitful and has led to a proliferation of translational verifiers for almost every mainstream programming language, including Frama-C for C [61], Creusot for Rust [44], Nagini for Python [45], Gobra for Go [112], and the verification-aware language Dafny [72], which compiles to C#, Java, JavaScript, Go, and Python.

The essential difference between interactive and semi-automated tools is their contrasting approach to the fundamental tradeoff between expressivity and automation. Today's interactive tools (e.g. VST [7], CakeML [66], and Bedrock2 [47]) have access to the full power of higher-order proof assistants and thus have essentially unbounded expressivity; this enables one to define very powerful program logics capable of reasoning about memory, concurrency, recursion, randomness, and more. Semi-automated tools, meanwhile, are limited to first-order, decidable theories supported by SMT solvers. However, as their names suggest, the user burden is significantly higher for interactive tools. Not only are the systems too expressive for general-purpose automation, but even writing specifications requires significant expertise in using proof assistants and the program logic in question; this makes such tools inaccessible to non-experts. In contrast, the limited expressivity, SMT-based automation, and friendly syntax of semi-automated verifiers (where specifications are written as comments similar to the programming language in use) makes such tools significantly more accessible. These tools have seen increasing use in industry, including by developers with no formal-methods experience (e.g. the AWS Encryption SDK for Dafny [1]). Similarly, for tool developers, IVLs have greatly reduced the burden of building semi-automated verifiers, reducing the hard problem of translating an expressive program logic to first-order formulas into the easier one of encoding the source-language-specific reasoning into a high-level language or logic. Interactive tools are much more difficult to build, requiring very large proof and development effort to handle new languages with new custom automation required for each one.

Despite these differences, both classes of tools are used in similar ways to verify imperative programs. Such a task is tricky, often requiring program logics based on separation logic [94] for reasoning about heap-manipulating programs, frequently with extensions for concurrency, nondeterminism, randomness, and more. Even proving weaker properties like memory safety is often nontrivial, and proofs can easily become unmanageable when combined with functional-correctness reasoning. A widely adopted solution is to separate such functional-correctness proofs into three steps: (1) defining a *functional model* of an imperative program, (2) proving that the imperative program refines the functional model, and (3) proving desired high-level properties of the model. Thus, the low-level details – proving validity of pointers, absence of overflow, etc – are kept separate from the high-level mathematical reasoning about the domain of interest.

This technique has been successfully applied at scale in many projects. C programs in the domains of cryptography [8], error-correcting codes [37], numerical methods [105], and a networked server [62] were verified in the Coq proof assistant; the imperative reasoning uses the Verified Software Toolchain (VST) [7] program logic for C (§1.1.1), while the functional correctness proofs use pure Coq reasoning and libraries such as Mathematical Components [54] for linear algebra, and the Flocq [27] formalization of floating-point arithmetic. In the semi-automated world, similar layered approaches were used in Dafny for verifying distributed systems in the IronFleet project [56] and for verifying a large-scale authorization engine used in production at Amazon Web Services [31]. Other efforts involve a combination of tools: in the VerifiedSCION project [92], the imperative and functional model proofs were completed in Gobra and the Isabelle proof assistant, respectively.

This paradigm has influenced the design of IVLs, which automate part of this process once and for all. Boogie is a simple imperative language with first-order specification focused on efficient VC generation; it aims to automate Hoare-style reasoning about imperative programs. Viper is an IVL for verifying programs with separation logic; it is well-suited to reasoning about heap-manipulating programs and to serve as a target more powerful program logics (e.g. those based on Concurrent Separation Logic [87]). However, it is less capable of dealing with functional models (hence the functional model reasoning in VerifiedSCION used Isabelle rather than Gobra, which is built on Viper). Why3, meanwhile, is more generic and includes many facilities for reasoning about functional models, including first-class notions of recursive types, functions, and predicates, while providing back-ends for both automated and interactive solvers as well as mechanisms for users to guide the verifier (e.g. for induction). However, it does not have a built-in notion of separation logic and it includes a more complex ML-like language as opposed to Boogie's simple imperative one.

This automation is useful, but how do we know that these program verifiers are themselves bug-free? Particularly dangerous are *soundness* bugs, where the verifier reports that the input program is verified, but in fact the specification does not hold. Today's interactive verifiers are designed with very strong soundness guarantees: the typical approach includes a formal semantics for the programming language of interest and a program logic proved sound with respect to this semantics; the user carries out proofs in that program logic within the proof assistant. Thus, these tools are *foundational*: one proves a single, machine-checked theorem that the specification indeed holds according to the language's formal semantics. If the tool is connected to a verified compiler, the guarantees may extend further to the assembly language layer or below. For example, VST is a foundational program logic for C verification; it is connected to the CompCert [74] verified C compiler. In the entire toolchain, the trusted computing base (TCB) is quite small, comprising the implementation of the proof assistant[1] and the formalization of the language semantics. Bedrock2 [47] follows a similar approach, extending the soundness guarantees further to the hardware layer.

Semi-automated verifiers, on the other hand, have no such guarantees. 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 back-end solver's decision of satisfiability or validity, and the IVL framework's non-trivial translation can all introduce errors. Each of these parts is a large, complex piece of software, and the total amount of trusted code can easily exceed hundreds of thousands of lines.[2] Additionally, even if all parts are individually correct, differences in assumptions or semantics between layers can still produce unsoundness. Even worse, these components often do not have well-defined semantics; one could not even state a theorem that the toolchain, or individual components, is sound.

This thesis addresses the question of how to bridge this gap and retain the benefits of the IVL-based approach to building verifiers while achieving foundational soundness guarantees. The crucial element is to *verify the IVL translation*; this would significantly reduce the burden of building foundationally verified program verifiers (such as a future first-order version of VST built on an IVL), much like IVLs have made it easier to build verifiers in general. We can prove the soundness of a key part of the pipeline once and for all and retain the benefits of the IVL-based approach, providing the user with foundational soundness guarantees without any additional proof effort or required expertise.

We target the Why3 IVL, which provides an expressive logic for writing and reasoning about functional models, including polymorphism, algebraic data types (ADTs), pattern matching, recursive functions, and inductive predicates. We call the logic implemented by Why3 **P-FOLDR** (**P**olymorphic **F**irst-**O**rder **L**ogic with **D**ataypes and **R**ecursion); it is

[1]The *de Bruijn criterion* asserts that proof assistants should have small proof checkers; this is the component that needs to be trusted. Coq satisfies the de Bruijn criterion. For a more thorough examination of CompCert's TCB, see [82].

[2]For example, verifying C code with VST involves a TCB (semantics+Coq kernel) of approximately 20K LOC, while the Frama-C+Why3+SMT toolchain comprises 500K-1M LOC, depending on the back-end solver(s) used. These estimates do not include lower level components (e.g. OCaml compiler, hardware, etc).

similar to the logics implemented by virtually every semi-automated verifier that permits functional model reasoning (e.g. Dafny, VeriFast [59] – see §1.1.2). Transforming this logic into the much simpler first-order logic used by SMT solvers is highly nontrivial, and we need an appropriate formal semantics to prove this transformation sound and remove this compilation from the TCB of Why3 and its clients. To that end, this thesis makes the following contributions:

1. We introduce P-FOLDR – similar to the pen-and-paper description given by Filliâtre [48] – in Chapter 2 and give a novel formalization of its semantics in the Coq proof assistant in Chapter 3. Our formalization constructs an explicit model in Coq, allowing us to prove consistency and rule out pathological behavior including non-well-founded recursive definitions. We give a sound proof system for P-FOLDR and use it to prove Why3 goals from the standard library.

2. We prove sound several of the transformations Why3 uses to encode recursive structures as first-order formulas. We give the first formally verified sophisticated pattern matching compiler (Chapter 4) and the first machine-checked proof of a first-order axiomatization of algebraic datatypes (Chapter 5).

3. We propose a lightweight design principle and small framework to write stateful programs in Coq while extracting to idiomatic OCaml. We use this to implement parts of the Why3 API in Coq, producing code executable both within Coq and compatible with the existing Why3 OCaml toolchain. We show how such an implementation can compose with our soundness proofs to achieve end-to-end guarantees and demonstrate that it is practical by testing on real-world examples (Chapters 6 and 7).

While other IVLs have formally *validated* implementations (see §1.2) – ones that produce certificates that, if checked successfully, prove soundness on a particular input – this thesis provides the first foundationally *verified* implementation of a practical, real-world IVL, the first IVL implemented within a proof assistant, and the first formal semantics and proved-

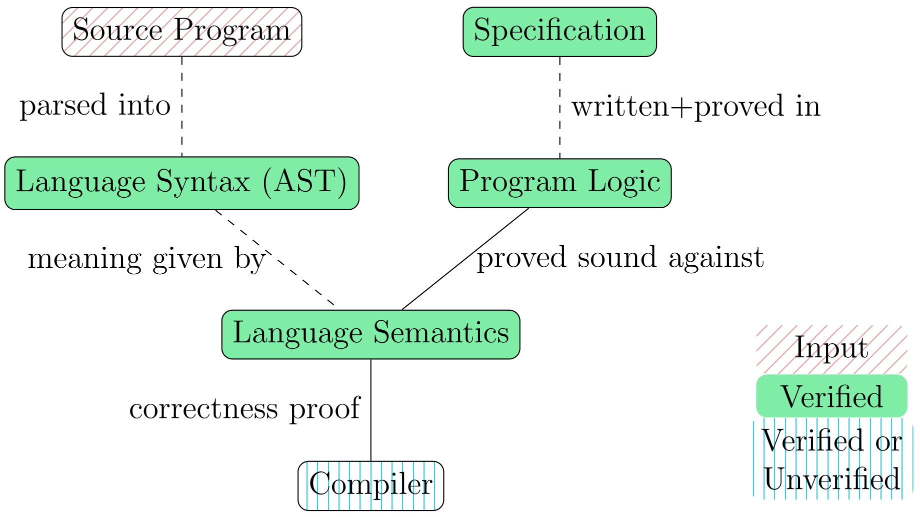

Figure 1.1: Foundational Verification

sound compilation for an IVL with features such as ADTs, pattern matching, recursive functions, and inductive predicates.

The Coq system described in this thesis is available at `https://github.com/joscoh/why3-semantics/tree/thesis`. Parts of this thesis (particularly Chapter 3 and parts of Chapter 7) were previously published [36]. The semantics described in this thesis has been improved since the paper's publication. Differences include the addition of pattern matching exhaustiveness checking, a more sophisticated termination checker, an improved implementation of α-equivalence and α-conversion, and the use of efficient sets and maps from the std++ library [65]. The ideas in §6.2 were presented at the CoqPL 2025 workshop [35].

1.1 Background

1.1.1 Foundational Verifiers

Figure 1.1 shows the typical pipeline for a foundational, interactive verifier. A tool parses the user's source code into an abstract syntax tree (AST). Written in a proof assistant, the verifier includes a formal semantics for the programming language the AST represents, a program logic (generally some form of separation logic), and a proof that the program logic rules are sound according to the semantics. The user then writes down their specification as a predicate in the (deeply embedded) program logic and proves their program correct using the proof rules of the program logic (generally assisted by custom tactics/proof automation). The soundness theorem ensures that if this proof succeeds, the specification indeed holds according to the defined semantics. In many cases, these verifiers are connected to verified compilers – compilers that generate a proof that the translation from the source to the target language (e.g. C and assembly) preserves the semantics. This composes with the program logic's soundness proof and extends the guarantees down to the target. Examples of foundational verifiers include VST [7] for C, CakeML [66], a verified bootstrapping compiler for ML, Bedrock2 [47], which compiles from a C-like language to hardware, and Verifiable P4 [110] for the P4 network packet programming language.

```
int minimum(int a[ ], int n) {
  int i, min;
  min=a[0];
  for (i=0; i<n; i++) {
    int j = a[i];
    if (j<min) min=j;
  }
  return min;
}

Definition minimum_spec :=
DECLARE _minimum
  WITH a: val, n: Z, al: list Z
PRE [ tptr tint , tint ]
  PROP (1 ≤ n ≤ Int.max_signed; Forall repable_signed al)
  PARAMS (a; Vint (Int.repr n))
  SEP (data_at Ews (tarray tint n)(map Vint (map Int.repr al)) a)
POST [ tint ]
  PROP ()
  RETURN (Vint (Int.repr (fold_right Z.min (hd 0 al) al)))
  SEP (data_at Ews (tarray tint n)(map Vint (map Int.repr al)) a)
```

Figure 1.2: A function to compute the minimum value of an array in C and its VST specification

To demonstrate how such tools work, we show an example program verified in VST. Figure 1.2 shows a C program that computes the minimum value of a nonempty array and a possible function specification in VST, which takes the form of a precondition (marked PRE) and a postcondition (marked POST). In the precondition, one can specify a pure Coq proposition in PROP (in this case that n is bounded), express the C function arguments as Coq variables in PARAMS, and specify the contents of memory in the SEP clause. In this example, the precondition asserts that there is a list of integers at pointer a whose contents are given by Coq list al. In the postcondition, one can similarly specify a pure proposition (PROP), the return value (RET - in this case the result of folding the Coq Z.min function over the list al), and the contents of memory (unchanged). A crucial part of VST's power is that *Coq itself is the assertion language* – one can specify propositions, return values, and memory contents as Coq functions. This makes the logic powerful and higher-order but difficult to automate effectively with general-purpose tools like SMT solvers. Here, the functional model is that of Coq's lists, while the function specification relates the return value with a particular operation on the model (folding Z.min over the list). One can use VST's custom Floyd [30] proof automation – tactics that perform symbolic execution and apply the rules of the VST program logic – to prove that the AST generated by minimum satisfies minimum_spec. Then, proofs about the functional model (e.g. that the result appears in the list if the list is non-empty) use ordinary Coq reasoning; the total proof burden is about 100 LOC.

1.1.2 Semi-Automated Verifiers and IVLs

Figure 1.3 shows the same minimum function implemented in the Dafny verification-aware programming language. The annotations include a precondition (**requires**) that the input array is nonempty, a postcondition (**ensures**) that the returned element is smaller than everything in the array (note that this is weaker than the VST spec above), and loop invariants (**invariant**) necessary to prove the specification. Note that the annotation burden

```
method minimum (a: array<int>) returns (m: int)
requires a.Length > 0
ensures ∀ i : int :: 0 ≤ i < a.Length ⟹ m ≤ a[i]
{
    m := a[0];
    var i := 1;
    while (i < a.Length)
    invariant 1 ≤ i ≤ a.Length
    invariant ∀ j : int :: 0 ≤ j < i ⟹ m ≤ a[j]
    {
        if (a[i] < m) {
            m := a[i];
        }
        i := i+1;
    }
    return m;
}
```

Figure 1.3: The minimum function in Dafny

is quite light for this simple example and that the specifications themselves are written in a combination of first-order logic and program syntax (e.g. for indexing into arrays). In this example, we do not relate the imperative **method** to a functional model (which in Dafny can be represented as a pure **function**), but it would be straightforward to do so.

These verifiers work by translating the program and the annotations into a series of verification conditions, often by using weakest-precondition-style reasoning. For example, the loop body can be represented as (using primes for updated variables):

$$L := (a[i] < m \to m' = a[i]) \land (\neg(a[i] < m) \to m' = m) \land i' = i + 1$$

To prove that loop invariant I is truly an invariant, the tool could check the validity of the formula $I \to L \land I'$ by negating the formula and proving unsatisfiability with an SMT solver. The precise language of such formulas depends on the verifier. For example, Dafny uses the Boogie IVL as its back-end, outputting Boogie programs and first-order logic formulas. Frama-C uses Why3 as its back-end and thus outputs (richer) Why3 logic definitions.

Semi-automated verifiers with expressive specification languages need to compile the

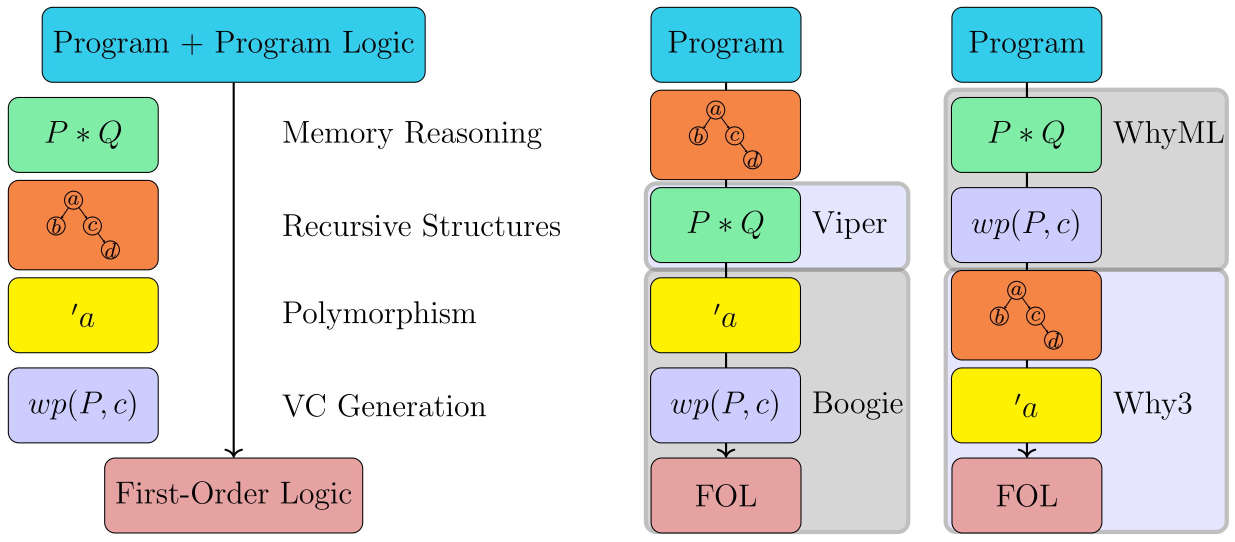

Figure 1.4: IVL-based verifier pipeline instantiated for Boogie, Viper, and Why3

following features not directly supported by SMT solvers:

- Any source-language-specific reasoning (e.g. for object-oriented programming)
- Some form of reasoning about heaps and memory (e.g. Separation Logic)
- Recursive structures that allow one to define functional models (e.g. ADTs, pattern matching, recursive functions)
- Polymorphism
- VC generation (i.e. transforming an imperative program into logical formulas)

These tasks do not have a defined order, and different IVLs choose different parts of the pipeline to compile to first-order logic (Figure 1.4). Boogie compiles the last two tasks, providing a small imperative language with polymorphic first-order specifications. Viper supports separation logic (more precisely, Implicit Dynamic Frames [101], a variant), providing a back-end that targets Boogie (and an SMT-based symbolic execution back-end). Why3 consists of two IVLs (§1.1.3): the Why3 logic (our focus) includes polymorphism and recursive structures, while ML-like WhyML, built atop the logic, includes VC generation and reasoning about non-aliasing memory.

Thus, the choice of IVL determines what compilation is done in the front-end and what can be left to the back-end. Dafny, for example, must compile recursive structures and heap reasoning in the front-end. Frama-C uses the Why3 logic as a back-end, and therefore does its own VC generation. Creusot translates to WhyML, and therefore only needs Rust-specific reasoning in the front-end (though it does need weakest-precondition reasoning to reconstruct a structured program from a control flow graph). Gobra uses Viper as a back-end; its front-end reasons about concurrency and other Go-specific features, as well as about recursive structures [10].

1.1.3 Why3 and Coq

As we have seen, Why3 consists of both the Why3 logic and the WhyML language (whose VC generator produces formulas in the Why3 logic) [50]. Though Why3 can be used directly to write verified software, as WhyML can be extracted to OCaml, it mainly serves as a back-end for other verification tools. Why3 supports about 20 provers, including SMT solvers (Z3 [43], CVC5 [12], and Alt-Ergo [38], among others), other numeric and first-order logic solvers (Vampire [95], Gappa [42], etc.), and proof assistants (Coq [106], Isabelle [85], and PVS [89]). Accordingly, Why3 is used in many verification tools and projects, including Frama-C for C, Creusot for Rust, EasyCrypt [17] for cryptography, SPARK 2014 for Ada, as well as tools for verifying OCaml [93], quantum circuits [32], and distributed systems [76]. 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 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. Verifying a theory reduces to verifying a series of *proof tasks*, consisting of a context of 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 user input and the specific features supported by the solver in use — for example, axiomatizing inductive types or inlining function definitions.

We use the Coq proof assistant [106] for our formalization. Based on a modern variant of the Calculus of Inductive Constructions (CIC), Coq implements a higher-order logic through a dependently typed programming language 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 §3.1 and §3.2. Coq is a particularly good choice for our formalization; we need powerful inductive types capable of encoding and reasoning about W-types (see §3.2.1), induction over arbitrary well-founded predicates, and an impredicative Prop (see §3.2.5), among other features. We also make heavy use of Coq's ability to extract Gallina code to OCaml to connect our IVL implementation with the existing Why3 toolchain (Chapter 6). Other, simpler systems like LF [55] have smaller kernels but lack many of these features.

Since Why3 is a classical logic and Coq implements an intuitionistic logic, our proof development assumes 3 axioms: classical logic (law of the excluded middle), indefinite description (Hilbert's ϵ 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).

Why3 and Coq already interact in several ways. Coq is one of the back-end proof assistants for Why3, providing support for goals outside the scope of SMT solvers. Additionally, one can *realize* Why3 theories in Coq (and PVS) by giving 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.

1.2 Related Work

In this section we discuss existing work related to verifying program verifiers, combining automated and foundational approaches, and verifying IVLs. In subsequent chapters, we present additional relevant related work.

1.2.1 Improving Automation of Interactive Verifiers

There are many recent efforts to improve the automation of interactive verification tools. One group of such efforts focuses on using SMT solvers within proof assistants. Early work included adding SMT solvers to Isabelle's famous Sledgehammer tactic [19], which invokes various external provers to automatically find proofs. Recent extensions include proof reconstruction in Isabelle [99].

Meanwhile, in Coq, SMTCoq [46] is a plugin allowing one to call external SMT solvers within Coq to prove goals; it checks the solver's certificate to ensure that the process is sound. Sniper [23, 22] extends this to first preprocess Coq goals to eliminate/axiomatize recursive structures, generating proof tactic scripts to assert that the procedure is sound (we discuss Sniper in more detail in §5.6). Itauto [18] uses a SAT solver implemented and proved correct in Coq. CoqHammer [40] is a Coq version of Sledgehammer. In this thesis, we focus on the compilation of P-FOLDR to first-order logic; to soundly connect this to SMT solvers within Coq, such efforts would be critical.

1.2.2 Other Automated and Foundational Tools

Other efforts have focused on improving the automation of foundational tools outside the IVL- and SMT-based pipeline. RefinedC [98] 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 proof search without backtracking. This has been further extended to Rust programs with RefinedRust [52]. VST-A [114] augments VST with annotations. It decomposes the verification problems into symbolic execution of straight-line Hoare triples, reducing the burden on the user while retaining foundational guarantees. In both cases, one must still prove residual goals in Coq, though the goals are expected to be simple.

1.2.3 Verification of IVL-Based Verifiers

More directly related to this thesis is the substantial work on verifying IVL-based verifiers. In this section, we focus on work related to VeriFast, Boogie, and Viper; in the next section we discuss Why3. Figure 1.5 uses the notation of Figure 1.4 to show which parts of the IVL pipeline each piece of related work (and this thesis) encompass.

VeriFast [59] is a separation-logic-based verifier for C and Java. While it is not quite an IVL, it serves a similar purpose when supporting multiple front-end languages. Featherweight VeriFast [60] formalizes and proves sound in Coq a core subset of VeriFast. Recent work [111] extends VeriFast with a proof-of-concept to generate Coq proof certificates to validate soundness against the CompCert C semantics. Both of these efforts operate 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. Both omit recursive functions and inductive datatypes (which are included in VeriFast's specification language) and do not verify function termination.

Vogels et al. [108] prove sound a VC generator for a Boogie-like language in Coq; they also prove sound a translation from a toy object-oriented language to Boogie. The same authors [109] then prove correct a more efficient VC generator for the Boogie-like language that avoids exponential blowups. More recently, Parthasarathy et al. [91] develop a certifying version of Boogie that generates Isabelle proof certificates to validate successful runs of Boogie's VC generation and compilation. Parthasarathy et al. [90] extend this to develop a certifying implementation of Viper's Boogie back-end. Dardinier et al. [41] extend this work further to provide a general framework for certifying the soundness of separation-logic-based IVL

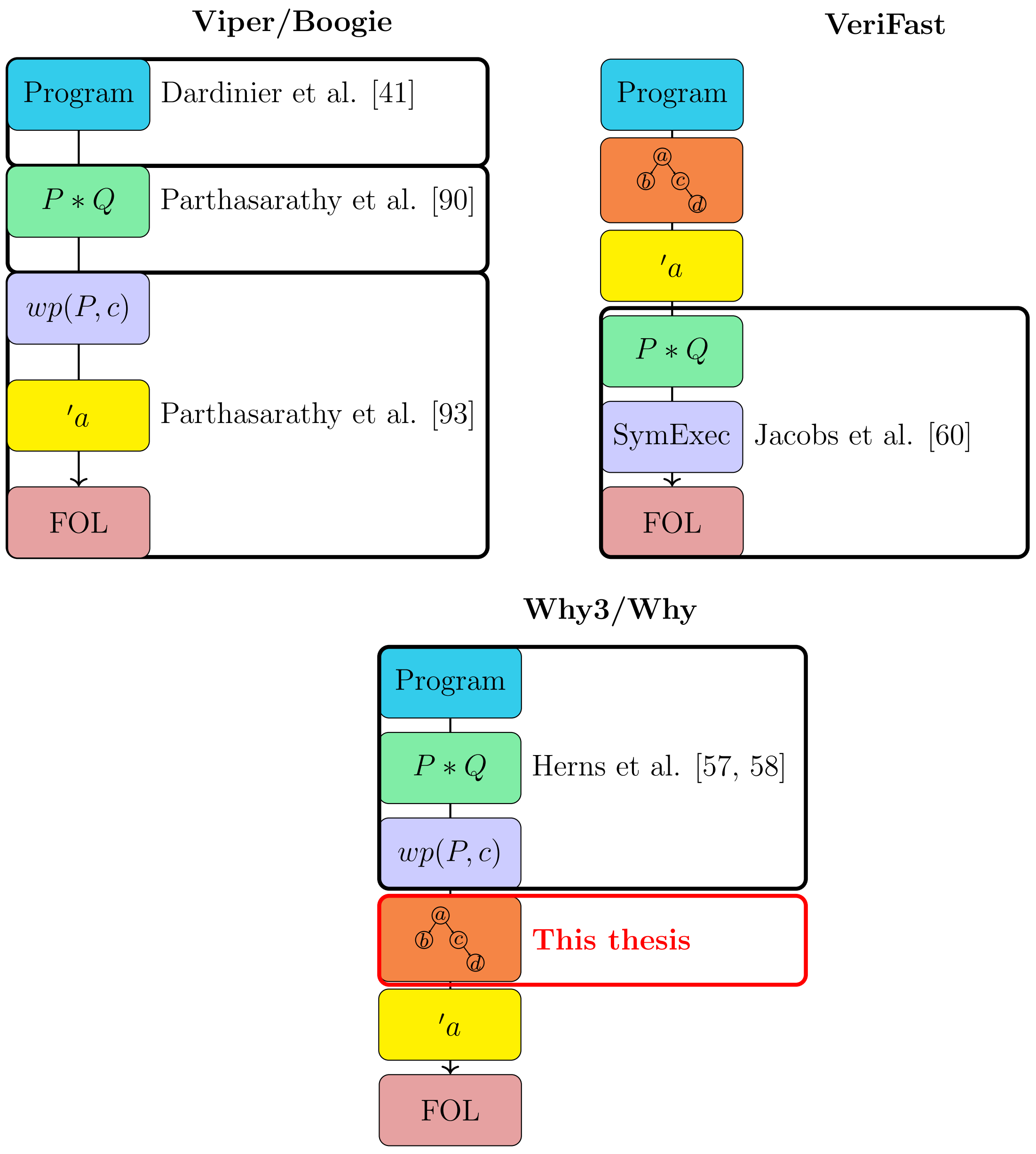

Figure 1.5: Formally Verified (Why3) and Validating (Viper/Boogie/VeriFast) IVLs

toolchains, instantiating this for Viper and a simple concurrent front-end language.

Though these efforts have similar goals as our work, they are ultimately somewhat orthogonal, and they differ in several key ways. First, they provide certifying, rather than certified, implementations. That is, they produce proof scripts that, if successfully checked, show that the tool was indeed sound on a particular input. Our approach proves sound the relevant compilation steps once and for all – we prove the stronger property that the compiler will succeed and give a sound result on all well-typed inputs. Second, these efforts, like Featherweight VeriFast, focus on different steps in the verifier pipeline: VC generation and separation logic. None handle recursive types, pattern matching, inductive predicates, or pure recursive functions (though recursive separation-logic predicates are included in the Viper formalization, and polymorphism is included in the Boogie one). It is instructive to note the differences between the semantics: the Viper semantics are operational and axiomatic, reasoning about heaps, program states, and separation logic operators. Meanwhile, we are focused on pure logic rather than stateful programs; accordingly, our semantics is denotational, constructing a model of Why3's logic in Coq that correctly interprets algebraic datatypes, recursive functions, and inductive predicates. The key proof steps also differ: while Boogie's certifying implementation must reason about control flow graph transformations and Viper's reasons about heaps and separation logic predicates, we must prove the soundness of axiomatizing recursive structures as first-order formulas. Finally, our verified IVL runs fully within a proof assistant (rather than as a certificate-producing extension to an existing tool); this would allow it to seamlessly serve as a back-end for other foundational tools.

1.2.4 Verification of Why3

Herms et al. [57, 58] develop a verified implementation of an older version of Why3 (then called Why) in Coq, including a verified VC generator for a subset of what is now WhyML. Their semantics are similar, though not identical, to ours for the basic first-order logic

(§3.1). However, their formalization only includes the core first- order logic; it does not include polymorphism, ADTs, pattern matching, recursive functions, inductive predicates, or the ϵ 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 [53] 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 (though the system includes an induction principle over integers), and the semantics are based on a shallow embedding in Dedukti. In principle, these certificates could be combined with our semantics to validate, rather than verify, certain transformations.

Chapter 2

P-FOLDR – The Logic of Why3

In this chapter, we describe P-FOLDR, the logic of core Why3. We describe the syntax, typing rules, and semantics of this logic in informal notation. We will briefly describe our Coq formalization of the syntax and typing rules; in the next chapter, we detail our novel Coq formalization of the recursive semantics. This logic was previously described by Filliâtre [48]; our presentation is largely similar, with several exceptions:

- Our typing rules for patterns and pattern matches are slightly more permissive, allowing pattern matches on any type (§2.2).
- We give an explicit algorithm for checking termination of recursive functions and predicates (§2.2.2).
- Our semantics for pattern matching differs significantly from Filliâtre's compilation-based approach (§2.3).
- We explicitly require that algebraic data types are inductive, a condition missing from Filliâtre's description (§2.3).

P-FOLDR extends classical first-order logic with the following constructs:

- Rank-1 polymorphism
- let- and if-expressions
- Hilbert's ϵ choice operator

- Algebraic data types
- Pattern matching
- Recursive functions and predicates
- Inductive predicates

Recursive types, functions, and predicates can be mutually recursive. Figure 2.1 shows an example of some definitions and statements in this logic. It defines the type of polymorphic lists as a recursive data type, defines the recursive predicate mem by pattern matching over this type, defines an abstract type and predicate symbol, defines an inductive predicate denoting sortedness of lists, and finally states a lemma that determines when adding an element to the front of a list preserves sortedness.

Why3 is explicitly intended to provide "a common specification language that aims at maximal expressiveness without sacrificing efficiency of automated proof search" [24]. In other words, P-FOLDR aims to be expressive enough to write rich, functional program specifications while still allowing efficient automation via translation to SMT and other solvers. Of course, Why3 is not the only tool with such a goal, and virtually all semi-automated verifiers today that aim to enable functional-model-level reasoning implement a broadly similar logic. For example, VeriFast (not built on an IVL) includes ADTs, pattern matching, and recursive functions, while Dafny includes all of P-FOLDR as well as other features like built-in polymorphic maps. Thus, no matter the IVL or toolchain, semi-automated verifiers have converged to a roughly common set of features achieving this tradeoff between expressivity and automation.

2.1 Syntax

Figure 2.2 shows the syntax of types, patterns, terms, and formulas in P-FOLDR. Our Coq formalization is a deep embedding; each part of the logic corresponds to a Coq inductive type. Types (τ in informal notation, vty in our Coq formalization) consist of built-in types int

```
type list 'a = Nil | Cons 'a (list 'a)

predicate mem (x: 'a) (l: list 'a) =
  match l with
  | Nil        → false
  | Cons y r → x = y ∨ mem x r
  end

type t
predicate le t t
axiom le_trans : ∀ x y z:t. le x y→ le y z→ le x z

inductive sorted (l: list t) =
  | Sorted_Nil:
      sorted Nil
  | Sorted_One:
     ∀ x: t. sorted (Cons x Nil)
  | Sorted_Two:
     ∀ x y: t, l: list t.
      le x y→ sorted (Cons y l)→ sorted (Cons x (Cons y l))

lemma sorted_mem:
∀ x: t, l: list t.
(∀ y: t. mem y l→ le x y) ∧ sorted l ↔ sorted (Cons x l)
```

Figure 2.1: A Why3 ADT, recursive predicate, inductive predicate, and lemma

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

Figure 2.2: Syntax of types, patterns, terms, and formulas

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

Figure 2.3: Syntax of concrete definitions

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 of variables, constructor application, wildcards, disjunctions, 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, ϵ choice, 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: quantifiers, let-bindings, match expressions, and ϵ only bind terms, not formulas.

Figure 2.3 shows the syntax for concrete definitions: mutually recursive ADTs, recursive functions and predicates, and inductive predicates. An ADT consists of a nonempty list of constructors, a recursive function or predicate contains a list of variable arguments and a body, and an inductive predicate consists of type arguments and a list of constructors. In addition to these concrete definitions, we can additionally have abstract definitions for type, function, and predicate symbols (our Coq formalization separates recursive and non-recursive functions/predicates but is otherwise identical to the syntax shown). A context Γ is a list of concrete and abstract definitions; we will ultimately define typing, truth, and validity with respect to a context.

2.2 Typing

P-FOLDR is typed: the judgments are that, in context Γ, a type τ is valid ($\mathsf{valid\_type}\ \Gamma\ \tau$), a term t has type τ ($\mathsf{term\_has\_type}\ \Gamma\ t\ \tau$), and a formula f is well-typed ($\mathsf{formula\_typed}\ \Gamma\ f$). We show the typing rules for types, patterns, terms, and formulas in Figures 2.4, 2.5, 2.6, and 2.7, respectively. Our type system is broadly similar to the pen-and-paper version [48], though it differs in a few places. The primary change concerns pattern matching and ADTs: we remove the requirement that a term/formula pattern match must be performed on an ADT and replace this with the condition that a constructor in a pattern must truly be an ADT constructor. This makes the type system more permissive; one is free to match, for instance, an integer against a variable or wildcard, but any constructor pattern enforces the ADT condition when needed. Because we need to know the declared ADTs, our typing judgments are defined in a context, whereas Filliâtre's type system requires only a signature (which simply lists the declared function/predicate/type symbols), checking the ADT conditions separately. We also include the exhaustiveness check on patterns in the typing rules (we will describe this check in detail in §4.2.2). Finally, we include a missing condition that the return type for pattern constructors and function applications is valid and we add the ϵ rule.

We make a few further remarks. First, note that for function and predicate application, the applied types are substituted for the symbol's type parameters. For instance, given function symbol $\mathsf{reverse}(\alpha)(\mathsf{list}(\alpha))\mathsf{:list}(\alpha)$ and term $\mathsf{reverse}(\mathsf{int})([1;2])$, 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 a disjunction pattern must be identical.

Typing definitions (ADTs, recursive functions, etc) is more complicated. Each has general well-formedness assumptions (e.g. the return types of ADT constructors are correct) as well as particular structural checks: ADTs must be inhabited, pattern matches must be exhaustive, recursive functions must be (syntactically) terminating, and inductive predicates

$$\frac{}{\Gamma \vdash \text{int}} \qquad \frac{}{\Gamma \vdash \text{real}} \qquad \frac{}{\Gamma \vdash \alpha} \qquad \frac{\text{t}(\alpha_1, \ldots, \alpha_n) \in \Gamma \quad \forall i \leq n, \Gamma \vdash \tau_i}{\Gamma \vdash \text{t}(\tau_1, \ldots \tau_n)}$$

Figure 2.4: Typing rules for types

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash x_\tau : \tau} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \_ : \tau} \qquad \frac{\Gamma \vdash p_1 : \tau \quad \Gamma \vdash p_2 : \tau \quad fv(p_1) = fv(p_2)}{\Gamma \vdash (p_1 \mid p_2) : \tau} \qquad \frac{\Gamma \vdash p : \tau \quad x_\tau \notin fv(p)}{\Gamma \vdash p \; as \; x_\tau : \tau}$$

$$\frac{\begin{array}{c}\text{f}(\alpha_1, \ldots, \alpha_m)(\tau'_1, \ldots \tau'_n) : \tau \in \Gamma \quad \forall i \leq m, \Gamma \vdash \tau_i \quad \Gamma \vdash \tau \quad \sigma = \{\alpha_1 \rightarrow \tau_1, \ldots, \alpha_m \rightarrow \tau_m\} \\ \forall i, j \leq n, i \neq j \implies fv(p_i) \cap fv(p_j) = \emptyset \quad \forall i \leq n, \Gamma \vdash p_i : \sigma(\tau'_i) \qquad \exists a, a \in \Gamma \wedge ADT(a) \wedge constr(\text{f}, a)\end{array}}{\Gamma \vdash \text{f}(\tau_1, \ldots, \tau_m)(p_1, \ldots, p_n) : \sigma(\tau)}$$

Figure 2.5: Typing rules for patterns

$$\frac{}{\Gamma \vdash c_{\text{int}} : \text{int}} \qquad \frac{}{\Gamma \vdash c_{\text{real}} : \text{real}} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash x_\tau : \tau}$$

$$\frac{\begin{array}{c}\text{f}(\alpha_1, \ldots, \alpha_m)(\tau'_1, \ldots \tau'_n) : \tau \in \Gamma \quad \forall i \leq m, \Gamma \vdash \tau_i \quad \Gamma \vdash \tau \quad \sigma = \{\alpha_1 \rightarrow \tau_1, \ldots, \alpha_m \rightarrow \tau_m\} \\ \forall i \leq n, \Gamma \vdash t_i : \sigma(\tau'_i)\end{array}}{\Gamma \vdash \text{f}(\tau_1, \ldots, \tau_m)(t_1, \ldots, t_n) : \sigma(\tau)}$$

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \textbf{let } x_\tau := t_1 \textbf{ in } t_2 : \tau_2} \qquad \frac{\Gamma \vdash f \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \textbf{if } f \textbf{ then } t_1 \textbf{ else } t_2 : \tau} \qquad \frac{\Gamma \vdash f \quad \Gamma \vdash \tau}{\Gamma \vdash \epsilon x_\tau, f : \tau}$$

$$\frac{\Gamma \vdash t : \tau_1 \quad \forall i \leq n, \Gamma \vdash p_i : \tau_1 \quad \forall i \leq n, \Gamma \vdash t_i : \tau_2 \quad exhaustive(t, p_1, \ldots, p_n)}{\Gamma \vdash \textbf{match } t \textbf{ with} \mid p_1 \rightarrow t_1 \mid \ldots \mid p_n \rightarrow t_n \textbf{ end} : \tau_2}$$

Figure 2.6: Typing rules for terms

$$\frac{}{\Gamma \vdash \top} \qquad \frac{}{\Gamma \vdash \bot} \qquad \frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2 \quad \circ \in \{\wedge, \vee, \implies, \iff\}}{\Gamma \vdash f_1 \circ f_2} \qquad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 = t_2}$$

$$\frac{\text{p}(\alpha_1, \ldots, \alpha_m)(\tau'_1, \ldots \tau'_n) \in \Gamma \quad \forall i \leq m, \Gamma \vdash \tau_i \quad \sigma = \{\alpha_1 \rightarrow \tau_1, \ldots, \alpha_m \rightarrow \tau_m\} \qquad \forall i \leq n, \Gamma \vdash t_i : \sigma(\tau'_i)}{\Gamma \vdash \text{p}(\tau_1, \ldots, \tau_m)(t_1, \ldots, t_n)}$$

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash f \quad q \in \{\forall, \exists\}}{\Gamma \vdash q \; x_\tau, f} \qquad \frac{\Gamma \vdash f}{\Gamma \vdash \neg f} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash f}{\Gamma \vdash \textbf{let } x_\tau := t \textbf{ in } f} \qquad \frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2 \quad \Gamma \vdash f_3}{\Gamma \vdash \textbf{if } f_1 \textbf{ then } f_2 \textbf{ else } f_3}$$

$$\frac{\Gamma \vdash t : \tau \quad \forall i \leq n, \Gamma \vdash p_i : \tau \quad \forall i \leq n, \Gamma \vdash f_i \quad exhaustive(t, p_1, \ldots, p_n)}{\Gamma \vdash \textbf{match } t \textbf{ with} \mid p_1 \rightarrow f_1 \mid \ldots \mid p_n \rightarrow f_n \textbf{ end}}$$

Figure 2.7: Typing rules for formulas

```
let rec check_ts tss ts =
    (* recursive data type, abandon *)
    if Sts.mem ts tss then false else
    let cl = find_constructors kn ts in
    cl != [] &&
    (* an algebraic type is inhabited iff
        we can build a value of this type *)
    List.exists (check_constr (Sts.add ts tss)) cl
and check_constr tss (ls,_) =
    (* we can construct a value iff every
        argument is of an inhabited type *)
    List.for_all (check_type tss) ls.ls_args
and check_type tss ty = match ty.ty_node with
    | ts(tl) ->
        List.for_all (check_type tl) &&
        check_ts tss tvs ts
    | _ -> true
```

Figure 2.8: Algorithm for determining if ADTs are inhabited

must belong to a special grammar and be strictly positive. These checks can be nontrivial to describe and formalize, but they will be crucial in allowing us to define the semantics appropriately in Chapter 3 – at key points throughout our formalization we will rely on these typing conditions to prove in Coq that the structures we want to define indeed exist.

2.2.1 Algebraic Data Types and Pattern Matching

The well-formedness assumptions for ADTs include the requirement that the constructors all have the same parameters and the correct return type. We also include metadata indicating whether a function is a constructor and if so, how many other constructors are in its type; we require that this information is consistent with the context. Why3 includes a check for strict positivity; we do not include this because we do not yet include function types (§7.1). The final check is to ensure that all data types are inhabited by searching for a constructor whose arguments are provably inhabited.

Figure 2.8 shows an OCaml-like pseudocode version of the algorithm for checking that ADTs are inhabited that we implement and verify. The algorithm works by checking each type symbol (for instance, list). To do this, it ensures that the type symbol is not in the process of being checked already (without this, **type** foo = | a **of** foo would be considered inhabited). It then gets the constructors for the type and searches for a constructor whose types all exist, according to the check check_type. This check considers all primitive types and variables inhabited; for every type symbol application, it checks both that the type symbol is inhabited (adding the original type symbol to the set of in-process-symbols) and that all arguments are inhabited. This is slightly more restrictive than Why3's check; Why3 does not assume all type variables are inhabited, but rather keeps track of which variables correspond to types not known to be inhabited.

This function is quite difficult to reason about effectively. The primary complication arises from the highly non-structural nature of the recursion: in the course of checking a type symbol, we could end up checking any other type symbol that occurs (transitively) in the arguments of a constructor for this symbol. Thus, there is no obvious way to convince Coq that this function terminates. Instead, we use the common technique of *fuel* – we add an additional nat-valued input to the function, and the call to check_ts in check_type occurs with a smaller fuel value. If the fuel ever reaches zero, the function terminates and returns false. Since each type symbol in the context will be checked at most once, our fuel bound is the number of type symbols in the context. Such a fuel parameter is also useful for induction as the theorems must reason about the entire context at once.

Pattern matching is required to be exhaustive. The check that Why3 (and P-FOLDR) uses is based on the compilation of pattern matches to simple patterns (patterns consisting of a constructor applied to variables or a wildcard). We defer the explanation of this algorithm and the exhaustiveness check to Chapter 4.

2.2.2 Recursive Functions and Termination Checking

For declared functions and predicates $f(\alpha_1, \ldots, \alpha_n)(x_{\tau_1}, \ldots, x_{\tau_m}) : \tau = b$ to be well-typed, their bodies must be well-typed, the free variables of b must belong to $\{x_{\tau_1}, \ldots, x_{\tau_m}\}$, and the type variables of b must belong to $\alpha_1, \ldots, \alpha_n$. Additionally, non-recursive definitions cannot include the declared function or predicate symbol in their body. Recursive functions must be terminating; accordingly, Why3 includes a termination checker based on structural inclusion of algebraic data types (for example, t is smaller than x :: t). P-FOLDR (and our Coq formalization) includes a version of such a termination checker, based in part on the Why3 and Coq termination checkers, but different from each. Here, we detail the differences and discuss our implementation.

Our termination checker starts by determining, for a given pattern match, the set of smaller variables – those that appear (transitively) in constructor patterns. We call this function pat_constr_vars. Crucially, this transitivity requires all intermediate variables to be of the same mutual type – in other words, we do not allow the following, since list and tree are different mutual types:

```
type list 'a = | Nil | Cons 'a (list a)
type tree 'a = | Leaf 'a | Node (list (tree 'a))
function size(t: tree 'a) : int =
  match t with
  | Leaf x → x
  | Node Nil → 0
  | Node (Cons x tl) → size x
  end
```

As we will see, our formalization does not support such "nested" ADTs at all, though this can always be encoded using mutual recursion, which we do support.

Then, we define a termination test that checks if the function/predicate bodies in a mutually recursive block are decreasing on a given set of indices. We show the most interesting rules in Figure 2.9. The function is parameterized by the mutual block containing functions fs and predicates ps, as well as the claimed decreasing indices for each such function (dec_idx(f)). It then determines if a function body b decreases with known-smaller-variable set s and a singleton-or-empty set h representing the original variable ($(s, h) \Downarrow b$). When evaluating a function call, if the function is in the mutual block, it checks that the argument at the claimed decreasing index is a variable in s (rule DEC_FUN_IN). The trickiest parts are

$$\frac{f \in fs \quad \mathbf{t}_{\mathsf{dec\_idx}(f)} = x_{\tau'} \quad \mathsf{uniform}(\boldsymbol{\tau}) \qquad x_{\tau'} \in s \quad \forall t \in \mathbf{t}, (s,h) \Downarrow t}{(s,h) \Downarrow f(\boldsymbol{\tau})(\mathbf{t})} \textsc{Dec\_fun\_in}$$

$$\frac{f \notin fs \quad \forall t \in \mathbf{t}, (s,h) \Downarrow t}{(s,h) \Downarrow f(\boldsymbol{\tau})(\mathbf{t})} \textsc{Dec\_fun\_notin}$$

$$\frac{\mathsf{var\_case}(h, s, x_\tau) \quad \forall (p,t) \in ps, (\mathsf{pat\_constr\_vars}\ p \cup (s \setminus fv(p)), h \setminus fv(p)) \Downarrow t}{(s,h) \Downarrow \textbf{match}\ x_\tau\ \textbf{with}\ ps\ \textbf{end}} \textsc{Dec\_match\_var}$$

$$\frac{\begin{array}{c} \forall ps', t, (c(ps'), t) \in ps \to \\ \left(\left(\bigcup_{\substack{i \leq |ps'| \\ \mathsf{var\_case}(s,h,\mathbf{t}_i)}} (\mathsf{pat\_constr\_vars}\ ps'_i) \right) \cup (s \setminus fv(c(ps'))), h \setminus fv(c(ps')) \right) \Downarrow t \\ \forall (p,t) \in ps, p \neq c(ps') \to (s \setminus fv(p), h \setminus fv(p)) \Downarrow t \end{array}}{(s,h) \Downarrow \textbf{match}\ c(\mathbf{t})\ \textbf{with}\ ps\ \textbf{end}} \textsc{Dec\_match\_constr}$$

$$\frac{(t = x_\tau \to \neg \mathsf{var\_case}(h, s, x_\tau)) \quad t \neq c(\mathbf{t}) \quad \forall (p,t) \in ps, (s \setminus fv(p), h \setminus fv(p)) \Downarrow t}{(s,h) \Downarrow \textbf{match}\ t\ \textbf{with}\ ps\ \textbf{end}} \textsc{Dec\_match\_rec}$$

Figure 2.9: Termination checking rules for function application and pattern matching

for pattern matches:

1. If the match occurs on a variable in s or h (a condition we call var_case), we check that each branch in the match terminates after removing the pattern's free variables from s and h and then adding pat_constr_vars to s (DEC_MATCH_VAR).
2. If the match occurs on a constructor application $c(\mathbf{t})$, we check that each branch in the match terminates under the following condition (DEC_MATCH_CONSTR):
   - If the pattern is $c(ps')$, we add to s all pat_constr_vars from the inner patterns corresponding to smaller elements (according to var_case) of $\mathbf{t}$ (as before, after removing the pattern free variables).[1]
   - Otherwise, we simply remove the pattern free variables from s and h.

There are also simpler recursive cases for matches (DEC_MATCH_REC) and function appli-

[1]Why3 implements simultaneous pattern matching by matching on tuples (see §4.2.3 for more details). Thus, this case lets us show termination when recursive functions are defined by matching on multiple arguments (e.g. zip for lists).

cations (DEC_FUN_NOTIN) that do not fall into one of the above categories. The remaining cases are straightforward, removing any bound variables from s and h before recursion. We require that all recursive function bodies satisfy this termination check with an initially empty s and with h containing the claimed decreasing variable. We then check that the arguments at all claimed indices have consistent types – all must belong to a single mutual ADT. Finally, we search over all possible index sets to find one that satisfies these conditions, if one exists. This last step takes potentially exponential time (though we use some heuristics, first filtering by argument types, to make it fast in practice); both Coq and Why3's termination checkers are worst-case exponential as well.

Termination Checking in P-FOLDR, Coq, and Why3 Our termination checker differs from those of Coq and Why3. Coq's termination checker is based on strictly decreasing subterms, where the subterm relation for a particular mutually recursive type is defined when the type is declared. Our checker is essentially a simpler version of Coq's termination checker, omitting facilities for nested recursion and dependent types. Furthermore, since P-FOLDR distinguishes between terms and propositions (unlike Coq) and only allows recursion on terms, it cannot express general well-founded recursion.[2] Why3's termination checker is quite different than ours:

1. Rather than using simple structural inclusion, it allows a lexicographic ordering of arguments that are structurally decreasing.
2. It does *not* rely on strict subterms of a single mutual ADT; instead, it is context-free and requires no knowledge of ADTs.
3. In some sense, it is lazy – it finds all paths in the call graph involving a recursive call and checks these rather than checking every function individually.

Adding lexicographic termination to our typechecker would be possible. Ultimately, we

[2]Why3 does allow well-founded recursion in WhyML; one proves that their relation is well-founded and then is allowed to use it as an explicit termination metric (variant) for a WhyML function or lemma. Since we do not model WhyML, we do not prove anything about this.

will prove (see §3.2.4) that, semantically, the notion of "smaller" variables is well-founded in Coq. Using a lexicographic ordering would make the relation more complex but poses no significant theoretical obstacles. The latter two differences are more problematic. Why3's context-free termination checker merely checks that the variables being recursed on appear guarded by *some* constructor, regardless of the type. Thus, Why3 can prove the function size at the beginning of the section terminating. In Coq, allowing such definitions would lead to a contradiction due to impredicativity:

```
Inductive Foo : Prop := | foo : (∀ A : Prop, A → A) → Foo.
Fixpoint oops (x : Foo) : False :=
  match x with foo f ⇒ oops (f Foo (foo f)) end.
```

However, neither the Why3 tool nor P-FOLDR can define oops, since one cannot write functions over inductive predicates (unlike in Coq, there is a sharp distinction between types and propositions).[3] This is a very subtle point; no previous published work identified that the soundness of the termination checker in Why3 relies on the inability to define functions over inductive predicates.[4] We note that a context-free termination checker would be difficult to reason about; our correctness theorem (§3.2.4) relies on induction over particular mutual ADTs, while a context-free version would require simultaneous induction over all defined ADTs.

Finally, the existing Why3 typechecker's laziness means that we can write other types of functions that do not pass Coq's termination checker:

```
function a (x: list int) (y: list int) : list int = b x y
with b (p: list int) (q: list int) : list int =
  match q with
  | Cons q qs → a p qs
  | Nil → Nil
  end
```

This is a particularly interesting example: a is decreasing on the second argument, but this is only evident when the calls are composed (resulting in a recursive call a p qs). Neither Coq's

[3]The Why3 tool automatically converts between booleans and propositions in some cases; it is possible to define something close to Foo but not oops.

[4]Though other consistency issues requiring restrictions on inductive predicates are known: https://gitlab.inria.fr/why3/why3/-/issues/546.

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

Figure 2.10: Grammar for inductive predicates

nor our termination checker can see this, as a does not decrease by itself. Such behavior would also be difficult to reason about, as this requires explicit description of mutually recursive function call graphs.

2.2.3 Inductive Predicates

Inductive predicates are significantly simpler. First, they have a special syntactic form: given a predicate $\mathrm{p}(\alpha_1, \ldots, \alpha_n)(\tau_1, \ldots \tau_m)$ with constructors $f_1, \ldots, f_n$, each clause f_i must be closed and belong to the grammar of Figure 2.10 (which depends on p). Additionally, all predicates from the mutually recursive block must occur only in strictly positive positions in the constructors. Both conditions are quite straightforward to check.

2.2.4 Typechecking

Now we define well-typing of entire contexts (valid_context Γ). In addition to requiring that each definition satisfies the above checks, we need to ensure that each function, predicate, and type symbol is defined only once. Finally, we require that the context is inductively well-typed: each definition is well-typed with respect to only the context defined before it. This ensures that each definition only uses previously defined symbols, and allows us to iteratively construct our models of well-typed contexts by adding each definition in sequence. However, it introduces complications when proving that context-modifying transformations are well-typed: we must prove both that all definitions are well-typed and also that the transformation preserves well-ordering.

With the full type system defined, we write a verified typechecker for types, patterns, terms, formulas, definitions, and contexts; this lets us typecheck concrete P-FOLDR contexts (useful in our proof system in §3.4.2) and prove some useful corollaries, such as the fact that terms are uniquely well-typed.[5]

2.3 Semantics

In this section, we describe a Tarski-style semantics for P-FOLDR, which is based on interpreting terms as objects in a model. In Chapter 3, we will explicitly construct such a model in Coq.

Interpretations and Valuations 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. A *sort* is a monomorphic type (e.g., list int is a sort, list 'a is not). A *pre-interpretation* assigns a meaning to all sorts and all function and predicate symbols. Each sort is assigned a nonempty domain (in Coq, a Set) $[\![s]\!]^\tau$ such that int is interpreted as $\mathbb{Z}$ and real is interpreted as $\mathbb{R}$.

Next, we need to assign an interpretation for function and predicate symbols. Polymorphism makes this trickier: for function symbol $\mathrm{f}(\boldsymbol{\alpha})(\tau_1, \ldots, \tau_n) : \tau$ and a list of sorts $\boldsymbol{s}$, we want the interpretation of $\mathrm{f}(\boldsymbol{s})$ to be a function $[\![\mathrm{f}(\boldsymbol{s})]\!]^\lambda$ of type $[\![\sigma(\tau_1)]\!]^\tau \times \ldots \times [\![\sigma(\tau_n)]\!]^\tau \rightarrow [\![\sigma(\tau)]\!]^\tau$, 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.

Finally, we require that the interpretations for ADTs satisfy several properties. For ADT a in mutual block m and sorts $\mathbf{s}$ we must have that:

1. Constructors are injective: for any constructor c of a, if $[\![c(\mathbf{s})]\!]^\lambda(\boldsymbol{t_1}) = [\![c(\mathbf{s})]\!]^\lambda(\boldsymbol{t_2})$, then $\boldsymbol{t_1} = \boldsymbol{t_2}$.
2. Constructors are disjoint: for any constructors c_1 and c_2 of a, if $[\![c_1(\mathbf{s})]\!]^\lambda(\boldsymbol{t_1}) = [\![c_2(\mathbf{s})]\!]^\lambda(\boldsymbol{t_2})$, then $c_1 = c_2$.

[5]This system has no type inference, and some intermediate terms and formulas are annotated with types (e.g. for the formula $t_1 = t_2$, we need to know the type of t_1 and t_2).

3. There is a function $\mathbf{find}(x)$ that, given x of type $[\![a(\mathbf{s})]\!]^{\tau}$, returns constructor c of a and arguments $\boldsymbol{t_1}$ such that $x = [\![c(\mathbf{s})]\!]^{\lambda}(\boldsymbol{t_1})$.
4. All ADTs in m must be *inductive*; represented by the following generalized induction principle: Let P be a property of any ADT a in m and any instance of $[\![a(\mathbf{s})]\!]^{\tau}$.[6] The following holds:

$$\begin{aligned}
&(\forall a \in m, c \in constrs(a), \mathbf{t},\\
&\qquad (\forall t \in \mathbf{t}, a' \in m, t \in [\![a'(\mathbf{s})]\!]^{\tau} \to \ P\ a'\ t) \to\\
&\quad P\ a\ \left([\![c(\mathbf{s})]\!]^{\lambda}(\boldsymbol{t})\right)) \to\\
&\forall a \in m, x \in [\![a(\mathbf{s})]\!]^{\tau}, P\ a\ x
\end{aligned}$$

The induction principle can be read as follows: suppose that for all ADTs a of m, all constructors c of a, and all $\mathbf{t}$, when P holds of all components of $\mathbf{t}$ that have ADT type for *any* ADT a' in m, then P holds of $[\![c(\mathbf{s})]\!]^{\lambda}(\boldsymbol{t})$. Then, for any a in m and x in $[\![a(\mathbf{s})]\!]^{\tau}$, P holds of x. This generalizes the usual induction principles on lists and trees.

These conditions differ from Filliâtre's in two ways. For the third property, we require that the **find** function is constructive; mere existence is not sufficient. More importantly, Filliâtre correctly describes ADT representations as the free algebra generated by the constructors, but gives an incomplete set of conditions by omitting an induction principle. This is not obviously a problem in the pen-and-paper semantics, but induction will be critical in proving the well-foundedness of recursive function models in Coq.

With this pre-interpretation, we can define a *valuation*, which describes the meaning of free 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. Then, a term variable valuation maps each variable x_{τ} to an element of $[\![\mathsf{vt}(\tau)]\!]^{\tau}$. Informally, we will refer to both valuations as v, and in most cases, the

[6]In Coq, P has type $\forall a, a \in m \to [\![a(\mathbf{s})]\!]^{\tau} \to \mathsf{Prop}$.

$$\begin{array}{rcl}
[\![c_{\text{int, real}}]\!]^t_v & = & c \\
[\![x_\tau]\!]^t_v & = & v(x_\tau) \\
[\![\mathrm{f}(\tau_1, \ldots, \tau_m)(t_1, \ldots, t_n)]\!]^t_v & = & [\![\mathrm{f}(v(\tau_1), \ldots, v(\tau_m))]\!]^\lambda([\![t_1]\!]^t_v, \ldots, [\![t_n]\!]^t_v) \\
[\![\textbf{let } x_\tau := t_1 \textbf{ in } t_2]\!]^t_v & = & [\![t_2]\!]^t_{v[x_\tau \to [\![t_1]\!]^t_v]} \\
[\![\textbf{if } f \textbf{ then } t_1 \textbf{ else } t_2]\!]^t_v & = & \textbf{if } [\![f]\!]^f_v \textbf{ then } [\![t_1]\!]^t_v \textbf{ else } [\![t_2]\!]^t_v \\
[\![\epsilon x_\tau, f]\!]^t_v & = & \epsilon(\lambda y \ \to \ [\![f]\!]^f_{v[x_\tau \to y]}) \\
\\
[\![\mathrm{p}(\tau_1, \ldots, \tau_m)(t_1, \ldots, t_n)]\!]^t_v & = & [\![\mathrm{p}(v(\tau_1), \ldots, v(\tau_m))]\!]^\lambda([\![t_1]\!]^t_v, \ldots, [\![t_n]\!]^t_v) \\
[\![\top]\!]^f_v & = & \text{true} \\
[\![\bot]\!]^f_v & = & \text{false} \\
[\![\forall x_\tau, f]\!]^f_v & = & \forall d, [\![f]\!]^f_{v[x_\tau \to d]} \\
[\![\exists x_\tau, f]\!]^f_v & = & \exists d, [\![f]\!]^f_{v[x_\tau \to d]} \\
[\![t_1 = t_2]\!]^f_v & = & [\![t_1]\!]^t_v = [\![t_2]\!]^t_v \\
[\![f_1 \circ f_2]\!]^f_v & = & [\![f_1]\!]^f_v \circ [\![f_2]\!]^f_v, \ \circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \\
[\![\neg f]\!]^f_v & = & \neg [\![f]\!]^f_v \\
[\![\textbf{let } x_\tau := t \textbf{ in } f]\!]^f_v & = & [\![f]\!]^f_{v[x_\tau \to [\![t]\!]^t_v]} \\
[\![\textbf{if } f_1 \textbf{ then } f_2 \textbf{ else } f_3]\!]^f_v & = & \textbf{if } [\![f_1]\!]^f_v \textbf{ then } [\![f_2]\!]^f_v \textbf{ else } [\![f_3]\!]^f_v
\end{array}$$

Figure 2.11: Semantics of terms and formulas without pattern matching

type valuation is fixed.

Semantics for Terms and Formulas Figure 2.11 defines the semantics of (well-typed) terms ($[\![t]\!]^t_v \in [\![v(\tau)]\!]^\tau$ if t has type τ) and formulas ($[\![f]\!]^f_v \in$ bool), for now omitting pattern matching. Almost all cases directly map the Why3 logical construct to the corresponding meta-logic (eventually, Coq) construct; the exception is for variable binding, which extends the valuation ($v[x \to y]$ denotes a valuation v' identical to v except that x is sent to y).

Pattern Matching Semantics Filliâtre interprets patterns by compiling them into elementary tests. In particular, posited functions isf and projf_i denote whether a term is an instance of constructor f and get the ith argument of such an application, respectively. Interpreting a pattern match is treated as compilation, replacing a match with a series of isf, projf_i, and let-bindings. We take an alternate semantics-oriented approach against which we later prove a more sophisticated version of such a compilation scheme sound (§5.5).

We encode pattern matching by describing how a successful pattern match affects the

$$
\begin{array}{rcl}
[\![x_\tau, \tau, d]\!]^p & = & \mathsf{Some}\{x \to d\} \\
[\![\_, \tau, d]\!]^p & = & \mathsf{Some}\ \emptyset \\
[\![p_1 \ \mid \ p_2, \tau, d]\!]^p & = & \textbf{if } \mathsf{isSome}[\![p_1, \tau, d]\!]^p \textbf{ then } [\![p_1, \tau, d]\!]^p \textbf{ else } [\![p_2, \tau, d]\!]^p \\
[\![p \ as \ x_\tau, \tau, d]\!]^p & = & m_1 \leftarrow [\![p, \tau, d]\!]^p; \textbf{return } (m_1 \cup \{x \to d\}) \\
[\![c(ps), \tau, d]\!]^p & = & \textbf{if } \tau = a(vs) \ \text{ for ADT } a \\
 & & \textbf{then let } (c_1, a_1) \ = \ \textbf{find}(d) \text{ in} \\
 & & \quad \textbf{if } c = c_1 \textbf{ then } [\![ps, tys, a_1]\!]^R \textbf{ else } \mathsf{None} \\
 & & \textbf{else } \mathsf{None} \\
 & & \\
[\![[], [], []]\!]^R & = & \mathsf{Some}\ \emptyset \\
[\![p :: ps, \tau :: tys, d :: ds]\!]^R & = & m_1 \leftarrow [\![p, \tau, d]\!]^p; m_2 \leftarrow [\![ps, tys, ds]\!]^R; \textbf{return } (m_1 \cup m_2)
\end{array}
$$

Figure 2.12: Definition of $[\![p, \tau, d]\!]^p$ – matching element d against pattern p of type τ

variable valuation v by adding new bindings for the variables bound in the pattern. To that end, we define the interpretation of a single match of pattern p of type τ against element d of type $[\![v(\tau)]\!]^\tau$; we denote this as $[\![p, \tau, d]\!]^p$. This interpretation produces a map of new variable bindings if the match succeeds; the definition is shown in Figure 2.12. The function works as we might expect: variables are always matched and bound to the matchee, wildcards always match but do not bind anything, disjunctions attempt to match the first pattern, then the second, and as-patterns match the pattern and add an additional variable binding. The constructor case is the most interesting: we first check if the type is an ADT; if so, we can call **find** to retrieve the constructor and arguments a_1 that produce d. If the constructors match, then we recursively check the arguments in a_1 against the corresponding patterns in ps (using the row matching $[\![ps, tys, ds]\!]^R$). If all matches succeed, we join the resulting maps (by typing, there are no free variables in common, so the maps have no overlap). The type τ is only needed to determine if the value d is an ADT; subsequently, we will not write the τ argument.

With this, the function that interprets pattern matches ($[\![t, ps]\!]^{ps}_v$, Figure 2.13) is simple: under a term variable valuation v, given a term t to match on and a list of pattern-term (or formula) pairs, iterate through the pattern list until a single pattern matches, then interpret

$$\mathsf{orep}\ v\ a\ o_1\ o_2\ = \textbf{match}\ o_1\ \textbf{with}\ |\ \mathsf{Some}\ m \to \mathsf{Some}\ [\![a]\!]_{m\cup v} |\ \mathsf{None} \to o_2\ \textbf{end}$$

$$\begin{aligned} [\![t, []]\!]_v^{ps} &= \text{default} \\ [\![t, (p, t_1) :: ps]\!]_v^{ps} &= \mathsf{orep}\ v\ t_1\ [\![p, [\![t]\!]_v^t]\!]^p\ [\![t, ps]\!]_v^{ps} \end{aligned}$$

Figure 2.13: Definition of pattern match semantics

the corresponding term under the valuation extended with the newly bound variables (note that the union operator prioritizes bindings in the first map). If no such pattern is found, the interpretation function returns a default value.[7] Then we extend the interpretations of terms and formulas with pattern matching (we show the term case; the formula case is similar):

$$[\![\textbf{match}\ t\ \textbf{with}\ ps\ \textbf{end}]\!]_v^t = [\![t, ps]\!]_v^{ps}$$

Recursive Functions and Inductive Predicates An *interpretation* (or a *full interpretation* in our Coq development) is a pre-interpretation that is consistent with the defined recursive functions/predicates and inductive predicates.

The specification for (mutually) recursive functions and predicates over ADTs is simple: for an interpretation to be consistent with $\mathrm{f}(\alpha_1, \dots, \alpha_m)(x_1, \dots, x_n) = b$, it must be the case that $[\![\mathrm{f}(\mathbf{s})]\!]^\lambda(\mathbf{a}) = [\![b]\!]_v^t$ 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).

Meanwhile, for inductive predicate $\mathrm{p}(\alpha_1, \dots, \alpha_k)(\tau_1, \dots, \tau_m) =\ \ |\ f_1\ |\ \dots\ |\ f_n$ and sorts $\mathbf{s}$, it must the case that $[\![p(\mathbf{s})]\!]^\lambda$ is the least predicate such that $[\![f_1]\!]_v^f, \dots, [\![f_n]\!]_v^f$ hold, where v maps $\boldsymbol{\alpha} \to \boldsymbol{s}$. This means that $[\![p(\mathbf{s})]\!]^\lambda$ satisfies the two properties that define the least predicate: under this interpretation, all $[\![f_i]\!]_v^f$ should hold and for any other predicate q that satisfies the constructors, for any arguments $\mathbf{a}$, $[\![p(\mathbf{s})]\!]^\lambda(\mathbf{a}) \to q(\mathbf{s})(\mathbf{a})$.

Validity, Tasks, and Transformations 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,

[7]All types are inhabited, so this is well-defined. In any case, we prove that exhaustiveness ensures that such a default value will never be reached.

$[\![f]\!]_v^f$ holds (note that the context Γ is implicit). 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 will frequently show the context explicitly for logical implication ($\Gamma, \Delta \vDash f$).

Why3 *proof tasks* and *transformations* are the crucial ingredients in our verified compiler. A task consists of a context Γ, a set of local assumptions Δ, and a goal f; a task is *well-typed* if Γ is a well-typed context, Δ consists of closed, well-typed formulas, and f is a closed, well-typed formula. A task is *valid* if $\Gamma, \Delta \vDash f$. Recall that Why3 translates tasks to simpler logics by applying a series of *transformations*, functions that take a proof task and produce some new tasks. A transformation T is *sound* if, whenever task t is well-typed, if all the output tasks $T(t)$ are valid, then so was t. This captures the notion that if we transform task t to tasks $t_1, \dots, t_n$, it suffices to prove these output goals.

Chapter 3

Why3Sem: Formalizing P-FOLDR in Coq

In this chapter, we present Why3Sem, a formalization of P-FOLDR in Coq. P-FOLDR occupies a middle ground between two types of logics most commonly formalized in proof assistants: First-Order Logic (FOL) and dependent type systems that implement higher-order logics such as the Calculus of Inductive Constructions (CIC), the type theory implemented by Coq.[1] P-FOLDR is clearly a strict extension of FOL (which has neither recursion nor polymorphism), and, while its functional programming features are inspired by logics like CIC, it differs in several crucial respects. First, it enforces clear distinctions between types and terms as well as between terms and formulas. For instance, ADTs and inductive predicates are completely separate, where in Coq, both are instances of general inductive types. Similarly, since everything is first-order, quantifiers and other binders only bind term variables (and not formulas), and there is no explicit quantification over types. Another crucial difference lies in semantics: while CIC is a programming language whose semantics can be expressed operationally as reduction rules (as in MetaCoq [102]), P-FOLDR is not

[1]Other proof assistants (e.g. Isabelle/HOL, HOL Light, etc) implement higher-order logic without dependent types. In such systems, packages compile recursive types and definitions to more primitive structures (e.g. [64]). Since P-FOLDR extends FOL with built-in recursive structures, we restrict our comparison to FOL and dependent type systems.

a programming language and has no built-in notion of evaluation. Indeed, it does not even contain lambda terms as first-class values (in Why3, these are encoded using the ϵ operator). Instead, as we saw in Chapter 2, its semantics can be defined denotationally, via interpretations (models).

However, it is unclear that the semantics in Chapter 2 are well-defined. That is, how do we know that there exist interpretations satisfying e.g. the ADT conditions of §2.3? If not, there are no models and thus the logic becomes useless: every formula is valid. To rule out such pathological behavior, we construct an explicit model of P-FOLDR in Coq, interpreting terms and formulas as objects in Coq's logic and showing that we can always construct interpretations consistent with well-typed definitions. This approach is common for formalizing and proving properties of FOL [51] and is similar to forms of logical relations used to formalize Martin-Löf type theory [4], but none of these account for recursive types, functions, and predicates. As we will see, constructing such a model is difficult, and relies crucially on P-FOLDR's typing rules. In the end, we give an explicit construction of a model for any well-typed set of definitions. This can be viewed as a type soundness result; rather than show that well-typed terms always reduce to values (as in CIC), we show that well-typed contexts have models.

3.1 Polymorphic First-Order Logic

We first briefly discuss our formalization of pre-interpretations (without ADTs), valuations, and the semantics of terms and formulas. Most follow the pen-and-paper description of Chapter 2 closely; we focus on formalization-specific aspects.

We encode the definition of type pre-interpretations ($[\![\cdot]\!]^\tau$ – for now without ADTs) as `pi_dom` in Coq:

```
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)).
```

Figure 3.1: Example use of heterogeneous lists

```
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.

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 for readability.[2] Function and predicate pre-interpretations ($[\![\cdot]\!]^\lambda$) are trickier, as the function argument types may differ. From §2.3, we know that if we have a function foo with e.g. int and list real arguments that returns an int, the interpretation must be a Coq function of type $[\![\text{int}]\!]^\tau \times [\![\text{list real}]\!]^\tau \rightarrow [\![\text{int}]\!]^\tau$. To encode this generically, we use *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 [34]). 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 3.1 shows an example of a 3-element hlist where the first and last elements are integers and the middle 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 Why3Sem.

[2]One may wonder why pi_dom does not simply require dom_aux vty_int = Z. Making this equality *definitional* rather than *propositional* results in fewer type casts in proofs.

With hlists, we can represent the function and predicate symbol pre-interpretation as follows, where arg_list is an hlist specialized to pre-interpretations and sym_sigma_args and sym_sigma_ret represent the map σ (which maps type variables to the corresponding sort) acting on the function arguments and return type, respectively:

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

We separate type and term variable valuations. First, a type variable valuation vt maps each type variable to a sort – v_subst extends this to map arbitrary types to sorts by replacing all type variables. With an abuse of notation, we will denote v_subst vt ty as $vt(ty)$. Then we define term variable valuations:

```
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)).
```

We can now define mutually recursive Coq functions term_rep and formula_rep that implement $[\![\cdot]\!]_v^t$ and $[\![\cdot]\!]_v^f$ (§2.3) – that is, they give the meaning of a (well-typed) Why3 term or formula under context Γ, pre-interpretation (pd, pf), and type variable valuation vt:

```
Fixpoint 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
```

Restricting our input to well-typed terms and formulas is crucial; we need the typing proofs for several typecasts in the functions. For example, if the variable x_{τ_1} has type τ_2, we know by typing that $\tau_1 = \tau_2$. However, adding the typing proof necessitates dependent pattern matching; to help with this, we use Coq's Equations [103] package. Almost all term_rep and formula_rep cases follow the definition in Figure 2.11 very closely; each operator in the meta-logic is the corresponding Coq one. We encode ϵ using Coq's indefinite description axiom and ClassicalEpsilon library. The exceptions are function/predicate application and pattern matching, which take more work.

Function and Predicate Application Recall that function application has the following interpretation (the predicate case is similar):

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

Encoding this in Coq is more challenging than it appears. $[\![t_1]\!]_v^t,\dots,[\![t_n]\!]_v^t$ hides a large amount of complexity: it represents the result of calling term_rep recursively 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 ty, term_has_type Γ t ty →domain pd (v_subst vt ty))
  (Hty: term_has_type Γ (Tfun f vs ts) ty):
arg_list (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. This is not hard to do using tactics; in fact, we define a more general function of which fun_arg_list and pred_arg_list are special cases.

But this alone doesn't quite work: the types are not correct. In particular, given 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 $\boldsymbol{\alpha} \to \boldsymbol{\tau}$. Thus, $[\![x_i]\!]_v^t$ has type $vt(\sigma(t_i))$. However, in order to apply the function interpretation $[\![\mathrm{f}(vt(\tau_1),\dots,vt(\tau_m))]\!]^\lambda$, we need $[\![x_i]\!]_v^t$ to have type $\sigma'(t_i)$, where σ' sends $\boldsymbol{\alpha}$ to $vt(\boldsymbol{\tau})$. Therefore, we must prove that σ and vt commute as type substitutions. This is not obvious, as the two substitutions are defined very differently: σ sends a small set of variables to specific types, while vt maps every variable to a sort. Nevertheless, the following lemma allows us to cast the types in fun_arg_list when adding each element to the hlist:

Lemma 3.1.1 (funsym_subst_eq)**.** *Define* $\sigma_{\boldsymbol{a},\boldsymbol{b}}(\tau)$ *as the map that replaces each type variable* $a \in \boldsymbol{a}$ *in* τ *with the corresponding type* $b \in \boldsymbol{b}$*.*

Let $\boldsymbol{\alpha}$ *be a unique list of type variables, let* $\boldsymbol{\tau}$ *be a list of types of the same length, and let* v *be a type variable valuation (e.g. a map from type variables to* sorts*). Then for any type* τ_1*,* $v(\sigma_{\boldsymbol{\alpha},\boldsymbol{\tau}}(\tau_1)) = \sigma_{\boldsymbol{\alpha},v(\boldsymbol{\tau})}(\tau_1)$*.*

We note that the pen-and-paper description of Why3's logic omits this subtlety; this is another place where our formalization discovers small gaps in the informal semantics.

3.2 Semantics for Recursive Structures

So far, we have formalized a classical polymorphic first-order logic, similar to that mechanized by Parthasarathy et al [91] for Boogie expressions. 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 the additional power of a stronger logic like Coq or ZFC to define our semantics.

In Chapter 2 and in Filliâtre's original pen-and-paper description, these features impose 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 the typing rules are not strong enough, these conditions might be unsatisfiable and then the logic becomes useless (every formula is valid). Second, we cannot tell if these conditions are sufficient. As we discussed in §2.3, Filliâtre's original presentation omits an induction principle, which we need in order to define models of recursive functions over ADTs and to prove Why3 goals that require induction.

Instead, we take a different approach: we consider the conditions on interpretations of Chapter 2 as a *specification* that our encoding of these structures must satisfy. Then, we construct objects in Coq satisfying these conditions and thus prove that, under the typing

```
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 pd) (sym_sigma_args f1 srts))
  (a2: arg_list (domain pd) (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 pd) (sym_sigma_args f srts) |
    x = constr_rep f (fst Hf) dom_adts (snd Hf)}}.
```

Figure 3.2: Specification for ADTs

rules, consistent interpretations exist for any possible assignment to uninterpreted symbols. In other words, we construct an explicit *model* of any well-typed P-FOLDR context. 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 spec, which we prove satisfied by this simpler representation. In later parts of the semantics and applications, only the spec (i.e. the properties of Chapter 2) is needed; the core encoding's complexity is completely hidden from the user.

3.2.1 Algebraic Data Types

Specification We show the Coq versions of the properties required of ADTs (§2.3) in Figures 3.2 and 3.3. Note that find_constr_rep is the Coq version of the **find** function. As discussed, we will need the (higher-order) induction principle to prove that these ADT representations are well-founded thereby enabling us to define terminating recursive functions over them (§3.2.4). In the following, we will construct the needed type (pre-)interpretation

```
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 m, constructor c, x = c(a)*)
  (∀ t t_in (x: adt_rep m srts (dom_aux pd) t t_in) (c: funsym)
    (Hc: constr_in_adt c t)
    (a: arg_list (domain pd) (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.
```

Figure 3.3: Generalized induction principle for ADTs

for ADTs adt_rep and constructor (pre-)interpretation constr_rep via an encoding based on W-types and show how we ultimately prove that such constructions satisfy this specification.

Core Encoding To generate arbitrary algebraic data types (lists, trees, etc) from a syntactic description, we use W-types [81], which provide a generic way to represent a variety of inductive types. We give a brief description before describing our encoding.

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 give an alternate representation. First, we 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 $|\mathsf{C}| = n$.

To generalize this, we construct a type A bundling the non-recursive data for all constructors. In our example, this is Either unit (int * string), denoting 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. The cardinality of B(x) denotes the number of recursive instances in the constructor to which x belongs. 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 P-FOLDR mutual ADTs as W-types, we must define I, A, and B. I is a type with exactly n elements, where n is the number of ADTs in the mutual block. Figure 3.4 shows the construction of A (build_base), while Figure 3.5 shows 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 encoding 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

```
Variable (vars: typevar → Set)
(abstract: typesym → list vty → Set).
(*Keep only the non-recursive types:*)
Definition get_nonind_vtys (l: list vty) : list vty := ...
(*Convert Why3 type to Coq Set*)
Definition vty_to_set (v: vty) : Set := match v with
  | 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.
```

Figure 3.4: Construction of non-recursive W-type data

```
(*[count_rec_occ] gives number of recursive typesym instances*)
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.
```

Figure 3.5: Construction of recursive W-type data

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 [54] library.

Abstraction Layer These definitions encode Why3 ADT descriptions as inductive types, but they do not have the correct types to satisfy the specification or the properties in §2.3. Furthermore, a user of the semantics should not need to manually construct finite types, build_constr_base instances, or the complicated dependent function describing the recursive instances for a constructor. Instead, we need simpler abstractions adt_rep and constr_rep which hide 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 is much simpler (Figure 3.6). We then require on our pre-interpretation for types (§3.1) the condition that all ADTs must be mapped to their corresponding adt_rep:

```
Record pi_dom_full (pd: pi_dom) := {
  adts: ∀ (m: mut_adt) (srts: list sort) (a: alg_datatype)
  (m_in: mut_in_ctx m gamma) (a_in: adt_in_mut a m),
    domain (dom_aux pd) (typesym_to_sort (adt_name a) srts) =
    adt_rep m srts (dom_aux pd) a a_in; }.
```

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 n-tuple make this difficult. For instance, we

```
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 ...
```

Figure 3.6: Abstraction layer for ADTs

```
Definition constr_rep {Γ: context} (gamma_valid: valid_context Γ)
  (m: mut_adt) (m_in: mut_in_ctx m Γ) (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 mapped appropriately ([pi_dom] condition)*)
  (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: arg_list (domain dom) (sym_sigma_args c srts)):
  adt_rep m srts dom t t_in := make_constr ...
```

Figure 3.7: Abstraction layer for constructors

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 3.7).

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}.
```

Proving the Specification Proving the first two properties (Figure 3.2) is fairly straightforward; each follows directly from properties of the underlying W-type. However, the last property is quite difficult to prove; we need an inverse function for constr_rep, allowing us to create an hlist from the bundled non-recursive arguments and bundled tuple-map for the recursive arguments of make_constr. We can build the hlist inductively but 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 Σ-type (dependent pair) of 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 Σ-type. The induction principle (Figure 3.3) ultimately (but nontrivially) follows from the well-foundedness of the underlying W-type in Coq. We note again that adt_rep and constr_rep can effectively be considered opaque; only the properties in the specification are ever needed by clients (including in our proof system in §3.4.2 and our compiler soundness proofs in Chapters 4 and 5).

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 are rarely used in practice, see §7.2). Nevertheless, our semantics can handle arbitrary list- and tree-like mutually recursive data structures sufficient for a wide variety of real-world specifications.

3.2.2 Pattern Matching

Our Coq formalization handles pattern matching almost exactly as described in §2.3 (which, as we noted, is a departure from Filliâtre). Our function that implements $[\![p, ty, d]\!]^p$ is called match_val_single; it matches an element of type domain pd (v_subst vt ty) against a pattern p of type ty, returning an optional map including all the new variable bindings if the term matches the pattern, and None otherwise. It has the signature:

[3]For instance type B a = | One of a | Two of (B (a, a)) is a datatype for perfect binary trees - those with 2^n nodes; see Okasaki [88] for more discussion and uses of non-uniform types.

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

There are two complications beyond §2.3 due to the dependent types. First, since we don't know yet which variables will be bound, the Σ-type in the result lets us hide the specific value while avoiding extra dependent type obligations that would make the definition unwieldy. We prove that if (x, y) appears in the outputted map 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 also prove that the output map contains exactly the free variables of the pattern. But it is simpler to define our function without proving these facts immediately. Second, in the constructor case of match_val_single, the arguments found by find_constr_rep (**find** in §2.3) are themselves a heterogeneous list; just as with function and predicate interpretations, there are several additional proof obligations to ensure that the dependent types are correct and the definition is well-formed. Our function that implements $[\![t, ps]\!]_v^{ps}$ is called match_rep; it is relatively straightforward to define as long as we are careful to write it in a way allowing Coq to prove termination of the nested term_reps.

3.2.3 An Ergonomic Semantics

The functions term_rep and formula_rep are at the core of Why3Sem; any higher-level reasoning (including in the definitions of recursive functions and inductive predicates) relies on their definitions. Though defined in a largely intuitive way that matches the pen-and-paper semantics, they are very tricky to reason about on their own, not least because they rely on many structures – the context (and its typing proof), the type and function symbol interpretation, the type and term variable valuation, and the term or formula's own typing proof. In many cases, some or all of these structures might change. For instance, the context changes in certain transformations that axiomatize recursive structures and the term variable valuation changes during substitution. However, if these structures change in certain predictable ways, we can still reason about the denotations. In particular, we prove that term_rep and formula_rep are equivalent under:

- Any two typing proofs – even without directly assuming proof irrelevance (term_rep_irrel and fmla_rep_irrel).
- Any two contexts and any two function and predicate symbol interpretations that agree on all function and predicate symbols present in the term or formula (term_change_gamma_pf). As we will see, this is crucial for defining inductive predicates and recursive functions.
- Any two type variable valuations that agree on all type symbols present in the term (tm_change_vt). This is tricky because the return type of term_rep depends on the type variable valuation; we need some typecasting.
- Any two term variable valuations that agree on all free variables in the term (tm_change_vv).

These theorems, proved as corollaries of two larger meta-theorems (tm_fmla_change_vt and term_fmla_change_gamma_pf), allow us to freely change irrelevant parts of the inputs and reason effectively about the semantics.

Additionally, we define several forms of syntactic substitution and prove these correct. In particular, we define simultaneous substitution of multiple terms for variables and derive the simpler single-term-for-variable and variable-for-variable versions. For each of these, we prove correctness by showing that these coincide with the semantic definitions (changing valuations): for example, $[\![t[t_1/x]]\!]_v^t = [\![t]\!]_{v[x \to [\![t_1]\!]_v^t]}^t$ if no variable from t_1 becomes bound in t (the general case is similar). With this, we define α-equivalence, show that α-equivalent terms have identical denotations, give an α-conversion function to rename bound variables (we give more details in §6.3.3), and finally give a capture-avoiding substitution function that also provably coincides with the semantic definitions. We additionally define type substitution, which is broadly similar but tricker because its specification involves reasoning about changes to the type variable valuation, changing the return type of term_rep.

For most of these theorems, we prove the result recursively for fun_arg_list and pred_arg_list,

```
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: arg_list (domain pd) (sym_sigma_args f srts))
  (vt: val_typevar) (vv: val_vars pd vt),
  (*The interpretation of f(srts) is the same as:*)
  funs_rep pf f l (fun_in_mutfun f_in) l_in srts srts_len a =
    cast ... (
    term_rep gamma_valid pd
    (*setting the function params to srts,*)
    (vt_with_args vt (s_params f) srts)
    (*recursively using [funs_rep] and [preds_rep],*)
    (pf_with_funpred pf l l_in)
    (*setting the function arguments to a,*)
    (val_with_args _ _ (upd_vv_args pd vt vv (s_params f) srts
      (eq_sym srts_len) (s_params_Nodup _)) args a)
    (*and evaluating the body*)
    body (f_ret f) (f_body_type l_in f_in)).
```

Figure 3.8: Recursive function specification

prove a result describing how match_val_single changes under, for instance, 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 all of our needs: defining recursive functions and predicates (§3.2.4, §3.2.5), giving a sound proof system (§3.4), and compiling P-FOLDR to polymorphic FOL (Chapters 4 and 5).

3.2.4 Recursive Functions

Specification Figure 3.8 gives the only property required of our function representations (§2.3): if we interpret all recursive functions as their representations, applying the function is equivalent to evaluating the body on the arguments. Once again, we will show how to construct the interpretations funs_rep satisfying this specification. Note that this specification is subtle: the function body is interpreted in a context in which the function itself is interpreted correctly. This circularity requires us to give an explicit definition of the function's interpretation 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 give 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 application of 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.

Our termination checker is described in §2.2.2 – recall that it maintains a set s of known smaller variables (called small in the Coq formalization) and a set h (hd – implemented as an optional variable) of the input, updating these sets as variables are bound and when additional smaller variables are found within pattern matches.

Our core encoding for the recursive function assumes that we have all needed information – 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 binary relation adt_smaller on the dependent pair {s : sort & domain s} 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

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

```
Fixpoint term_rep_aux ... (Hty: term_has_type gamma t ty)
(Hdec: decrease_fun fs ps small hd m vs t)
(Hsmall: ∀ x, 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) ... := ...
```

Figure 3.9: The invariants for term_rep_aux

adt_reps; we then give the transitive closure adt_smaller_trans, which is still well-founded.

Before defining the recursive function, we first need to give the input and return types; for the Fix operator, all needed arguments must be packed into a single type. The full recursive function must include the function or predicate symbol and a proof that the symbol is in the mutual block (we can view this as defining a family of functions to encode mutual recursion), a list of sorts srts, an hlist of function inputs of the appropriate type, a variable valuation, and a proof that the type variable valuation associates each function type 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 claimed decreasing index of each satisfy adt_smaller_trans), and then lifting it through the packed arguments; this lifted relation is still well-founded. Our function's return type depends on the input: when 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. Note that this is flexible enough to permit mutual blocks with a mix of functions and predicates.

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 similar, but has a simpler return type.

The types of these functions are significantly more complex than term_rep — they must maintain many more invariants beyond typing (Figure 3.9). In particular, they keep track of small and hd (from the termination check), the proof that the term terminates (decrease_fun), the fact that small and hd consist of ADTs (vty_in_m), 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 adt_smaller_trans, and hd's valuation is equal to that of the input at the decreasing index (d). This invariant is the key to defining this function: we establish a link between syntactic smallness (which involves keeping track of a set of variables) and the semantic notion based on structural inclusion of adt_reps (and thus of the underlying W-types).

The body of term_rep_aux is broadly similar to term_rep, except for the function application case and the proofs of invariant preservation. The invariant preservation is mostly straightforward, but the interesting case occurs when we add the pat_constr_vars to small when pattern matching. To show that the invariant is preserved, we must show that all such variables have semantically smaller values (according to adt_smaller_trans). We show a slightly simplified form of this theorem (pat_constr_vars finds the variables known to be smaller inside a pattern as described in §2.3):

Theorem 3.2.1 (match_val_single_smaller)**.** *For any pattern p of type τ, $d \in [\![v(\tau)]\!]^{\tau}$, and map m, if $[\![p, \tau, d]\!]^{p} =$ Some m, then for any x and y such that $m[x] =$ Some y, if $x \in$ pat_constr_vars p, then adt_smaller_trans y d holds.*

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 (note that this is not the case when describing conditions on interpretations in Chapter 2).

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 function call's list of arguments ts. From the termination condition, we know that the element of ts at the decreasing index i is a variable 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 not trivial. First, we need a new version of fun_arg_list (which we call get_arg_list_recfun) that accounts for the additional invariants. 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 of get_arg_list_recfun is term_rep_aux applied to the ith element of ts and showing 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.[5]

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. We omit the full definition here.

[5]In fact, we need to encode yet another pieces of information in the return type of term_rep_aux: the fact that if term_rep_aux is evaluated on a constructor f applied to term list tms, then we can find the arg_list a such that the result is $[\![f]\!]^{\lambda}(a)$, and that if the ith element of tms is a variable, then the ith element of a is $v(x)$. We need this condition for our additional termination case allowing us to match on a constructor to find smaller variables. But this does not complicate the proof significantly beyond the above variable case.

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 the information needed, followed by 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: arg_list (domain pd) (sym_sigma_args f srts)):
  domain (funsym_sigma_ret f srts) := ... funs_rep_aux ...
```

Proving the Specification Finally, we must prove the specification of Figure 3.8. To show this, we prove that term_rep_aux and term_rep are equivalent as long as the pre-interpretation assigns recursive functions to funs_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 other words, funcs_rep_aux would be defined in terms of itself. But we can break the circularity by proving another result: similarly to what we showed for term_rep in §3.2.3, we prove that term_rep_aux is invariant under changes to the pre-interpretation that agree on all function and predicate symbols that 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 recursive function symbol interpretations correctly after the fact without changing the semantics.

3.2.5 Inductive Predicates

Recall that we must give an interpretation for inductive predicate $p(\boldsymbol{\alpha})$ such that all constructors for p hold and for any other predicate q that satisfies the constructors, for any arguments a, $[\![p(\mathbf{s})]\!]^\lambda(a) \to q(\mathbf{s})(a)$ (§2.3). Here, we do not show the Coq specification.

```
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.
```

Figure 3.10: Inductive predicate and impredicative representation for even

Inductive predicates are significantly simpler to define than recursive types or functions, as we can take advantage of the impredicativity of Coq's Prop (where we can quantify over Prop and still produce a term in Prop) to use functions instead of inductive types. We thus use an representation similar to a Böhm-Berarducci [26] encoding for types, where we represent an inductive predicate as a function in Coq encoding its induction principle. Figure 3.10 demonstrates this technique with the even property on natural numbers. 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.[6] 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 3.11 shows the non-mutual case. The mutual case is similar, but the P argument becomes an hlist of ∀(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.

[6]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: arg_list (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 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).
```

Figure 3.11: Representation of (non-mutual) inductive predicates

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 $[\![f]\!]^f_{(p\to P;v)}$ 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 inductive predicate p can be rewritten into a special form: $\forall \mathbf{x}$, let $\mathbf{y} = \mathbf{t}$ in $(g_1 \wedge \ldots \wedge g_k) \to p(\mathbf{z})$ — this follows from the inductive predicate grammar in the typing rules (Figure 2.10) and the definitions of term_rep and formula_rep. With this form, we need to prove $[\![p(\mathbf{z})]\!]^f_{(p\to I;v)}$, assuming $[\![g_j]\!]^f_{(p\to I;v)}$, 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 $[\![f_j]\!]^f_{p\to P}$ for all j, then P holds of $\mathbf{z}$. In particular, this implies that $[\![f_i]\!]^f_{p\to P}$, so by again rewriting f_i into its special form, we see that it suffices to prove $[\![g_j]\!]^f_{(p\to P;v)}$ for all j. We already assumed $[\![g_j]\!]^f_{(p\to I;v)}$. We now prove separately that, for any formula g, if p appears strictly positively in g, then $[\![g]\!]^f_{(p\to I,v)}$ implies that $[\![g]\!]^f_{(p\to P,v)}$ for any P and v. This completes the proof.

Unsurprisingly, positivity is crucial in ensuring that the least predicates exist for a given definition. We again rely on the typing rules to construct objects satisfying the intended properties. To complete our proofs of the inductive predicate 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.

3.3 P-FOLDR as a Logic

We now define a *full interpretation* — a pre-interpretation that is consistent with the specifications 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; i.e. any well-typed set of Why3 definitions has a model:

```
Theorem full_interp_ex: ∀ funi predi,
{pf: pi_funpred gamma_valid pd pdf |
  full_interp gamma_valid pd pf ∧
  (∀ f srts a, In (abs_fun f) gamma →
    (funs gamma_valid pd pf ) f srts a = funi f srts a) ∧
  (∀ p srts a, In (abs_pred p) gamma →
    (preds gamma_valid pd pf) p srts a = predi p srts a)}.
```

We can define the notions of validity, satisfiability, etc (§2.3) and prove metatheorems about P-FOLDR:

Theorem 3.3.1 (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 3.3.2 (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 3.3.3 (semantic_lem). *For any full interpretation I and monomorphic formula f, $I \vDash f$ or $I \vDash \neg f$.*

Theorem 3.3.4 (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.

Theorem 3.3.2 justifies the negate-and-prove-UNSAT approach to proving validity using an SMT solver.

We then give definitions for Why3 proof tasks (context, hypotheses, and goal) and transformations (§2.3). Recall that a task is *valid* if $\Gamma, \Delta \vDash f$ and a transformation T is sound if, whenever task t is well-typed, if all the output tasks $T(t)$ are valid, then so was t. In Chapter 5, we prove that the main Why3 transformations eliminating recursive structures are indeed sound according to Why3Sem.

3.4 A Sound Proof System for P-FOLDR

We want to validate that Why3Sem is *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 (later) prove the soundness of the main transformations Why3 uses when translating to simpler logics (Chapter 5).

We argue for the correctness of Why3Sem by first noting that we closely match the intended semantics (Chapter 2 and [48]). 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 Why3Sem closely aligns with the intended meaning of the various connectives and features in the logic. These applications also show that Why3Sem is useful; it admits a standard proof system and a more usable tactic-based interface, it allows us to prove naturally occurring Why3 goals, and, most importantly, we can prove Why3 transformations sound and ultimately produce a proved-correct P-FOLDR to FOL compiler.

3.4.1 Natural Deduction

Proving goals directly from the semantics is tedious and unintuitive; we want a proof system with which to prove validity syntactically. 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 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, giving 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 tasks satisfying P*)
  (∀ x, In x (tr t) → derives x) → (*tr outputs 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 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)]. To prove the soundness of this transformation, we show that if $\Gamma, \Delta \vDash f$ and $\Gamma, \Delta \vDash g$, then $\Gamma, \Delta \vDash f \wedge g$, which is easy. The and-intro rule follows immediately:

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

We use the same idea to give 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. Some of these rules are significantly trickier to prove sound:

Quantifiers Particularly challenging are the ∀-introduction and ∃-elimination rules – in both cases, we must modify the context to provide a fresh constant symbol and reason about how interpretations for the two contexts relate, being careful to map the fresh symbol to the valuation of the quantified variable. Modifying the context is particularly tricky because everything depends on it – the typing rules, interpretations, valuations, and semantics.

Rewriting To enable rewriting of equal terms or equivalent formulas, we give a term-for-term substitution function replace_tm that we prove preserves typing[7] and semantics (assuming that the terms/formulas being substituted have equivalent denotations). As usual, we must be careful to avoid variable capture; here, we are conservative and avoid substituting under any binders clashing with a free variable (rather than α-converting).

Type Substitution Given a polymorphic hypothesis, we want to substitute types; we provide a generic type substitution function ty_subst to do this. We want to prove a similar correctness theorem as for term substitution (§3.2.3), but this is extremely challenging. Not only is the theorem difficult to state since the type substitution changes the return type of term_rep, but because the variables are typed, under a type substitution, two unequal variables can become equal (ex: x_a and x_{int} are equivalent under the substitution $\alpha \rightarrow \mathsf{int}$). Reasoning about the uniqueness of variable names inductively is very tricky; we ultimately need several stronger intermediate notions about unique names that disappear from the final theorem.

We also have 3 transformations/proof rules derived from our recursive structure semantics:

Unfolding Recursive Functions We provide a proof rule to replace a (possibly) recursive function or predicate instance with its body (i.e. unfolding). The soundness follows relatively directly from our semantics for recursive functions (§3.2.4), though our implementation has

[7]We found a bug where such rewriting is *not* well-typed in the Why3 tool, see §4.2.3.

2 passes: finding recursive function instances and then rewriting, allowing us to control how many instances should be unfolded as well as to reuse the rewriting proofs.

Simplifying Pattern Matches We also want a way to simplify "obvious" pattern matches – for example, **match** nil **with** | nil → a | cons x y → b should simplify to a. To do this, we provide a simplifier matches that determines whether a given pattern matches (and if so, gives the matched variable-term pairs), does not match, or if we cannot tell (e.g. if the term matched on is uninterpreted). We then prove that if match_val_single (§3.2.2) gives None, then matches gives NoMatch or DontKnow, and if match_val_single gives Some l, then matches gives DontKnow or Matches l', where l and l' agree on variables, and l represents the interpretation of the terms in l'. We finally prove that simplifying the Matches cases via simultaneous substitution of the discovered terms preserves the semantics.

Induction over ADTs Finally, we prove an induction rule for non-mutual ADTs. Suppose we have an ADT a with constructors $f_1, \ldots, f_n$ such that for constructor i, the recursive arguments are $a_{i,1}, \ldots, a_{i,k}$ and the non-recursive arguments are $b_{i,1}, \ldots, b_{i,j}$. Then, to prove goal $\forall (x : a), p(x)$, we would like instead to prove n goals, where the ith goal has the form:

$$\forall \boldsymbol{a}, \boldsymbol{b}, p(a_{i,1}) \wedge \ldots \wedge p(a_{i,k}) \rightarrow p(f_i(\boldsymbol{a}, \boldsymbol{b}))$$

Note that $p(a_{i,m})$ is really the substitution $p[a_{i,m}/x]$. The soundness of this transformation ultimately, though highly nontrivially, follows from adt_rep_ind (§3.3); we establish a correspondence between the syntactic induction expressed by this transformation (in terms of constructor function symbols and arguments) and the semantic induction principle (expressed via the constr_rep functions applied to hlists).

Note that we show the soundness of these proof rules by providing transformations as Coq functions. However, these should be considered distinct from the Why3 transformations we discuss in Chapter 5. Though some of the above proof rules have Why3 analogues, here

Tactic	Purpose
intros	Introduce quantified variables and hypotheses
assert	Prove an intermediate result
f_equal	Turn goal $f\ x_1 \dots x_n = f\ y_1 \dots y_n$ into goals $x_1 = y_1, \dots, x_n = y_n$
reflexivity	Solve $x = x$
symmetry	Turn $x = y$ into $y = x$
apply H	If $H : P \to Q$ in context, turn goal Q into goal P
clear H	Remove hypothesis H
split	Turn goal $P \land Q$ into goals P and Q
exists	Instantiate existential quantifier
destruct H	Depending on hypothesis: 1. Split hypothesis H: $P \land Q$ into hypotheses P and Q 2. Split hypothesis H: $P \lor Q$, results in 2 goals, one assuming P, the other assuming Q 3. Change hypothesis H: $\exists x, P(x)$ to get x and $P(x)$ for fresh x Note: unlike Coq, not defined for arbitrary inductive types
left/right	Turn goal $P \lor Q$ into P or Q
exfalso	Turn goal into $False$
apply I	Solve goal $True$

Figure 3.12: Basic tactics implemented in our Why3 proof system

we focus on writing simple transformations to prove the soundness of the proof rules; we are neither faithful to Why3 nor concerned with any optimizations. Nevertheless, our generic and extensible derivation rule allows us to replace these with more complex transformations, enabling the use of the compiler transformations (e.g. axiomatization of ADTs) in the proof system with no issue.

3.4.2 Proving Why3 Goals in Coq

With these proof rules, we then developed a small tactic library for applying the proof system (just as with Coq, it is tedious to reason in pure natural deduction). Our tactic library is based on Coq's. Figure 3.12 shows the basic Coq tactics we implemented; almost all map fairly straightforwardly to a natural deduction introduction or elimination rule. The more complex tactics are shown in Figure 3.13; they correspond to the more sophisticated proof rules we detailed above.

Tactic	Purpose
specialize H y	Turn hypothesis $H : \forall x, P(x)$ into $P(y)$
specialize_ty H a	Specialize polymorphic hypothesis H with type a
rewrite H	Rewrite term equality or logical equivalence H; we include forward and backward variants for both goals and hypotheses
unfold f	Unfold (recursive) function or predicate f
unfold_at f n	Unfold nth occurrence of f
simpl_match	Simplify pattern matches
induction	Given goal of form $\forall (x : a), P(x)$, perform induction over non-mutual ADT a

Figure 3.13: More complex tactics implemented in our Why3 proof system

```
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
```

Figure 3.14: Selection of theorems from Why3 standard library proved correct in Coq

We then use these tactics (and some others, for instance to apply the typechecker) to prove the validity of Why3 goals; these proofs end up quite similar (albeit with less automation) to the corresponding proof in Coq. Before we can do this, we first provide a limited implementation of Why3 theories, which we view as a preprocessing step to create the context Γ and local definitions Δ; we do not keep track of theory ancestry as Why3 does.[8] We handle two types of imports: 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. Our implementation requires all clones to be exported. 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 imported 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.

We translated parts of Why3's relations, algebra, int, option, list, and bintree libraries,[9] 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 3.14 for a selection of lemmas proved. These lemmas, and the definitions they depend on, involve extensive use of theory using and cloning, polymorphism, induction over ADTs, recursive

[8]For our verified compiler, we do not prove anything about theories. We are principally interested in reasoning once the theory has been split into its constituent tasks.

[9]`https://why3.lri.fr/stdlib/`

functions, and pattern matching. This gives us additional confidence in the correctness of Why3Sem; we can prove these results in the way we expect, suggesting that the semantics truly capture the intended meaning of the connectives and recursive structures.

This proof system has several limitations: since it is deeply embedded in Coq, and each tactic is running a Coq function over the AST of the deeply embedded P-FOLDR terms, it is both slow and unable to take advantage of regular Coq tactics. Similarly, we cannot reason about types such as lists and trees as Coq inductive types, but rather only using our effectively opaque encoding. Our main focus is a verified compiler from P-FOLDR to polymorphic FOL; however, if we wanted a practical verified Coq back-end for Why3 as well (which would be valuable, as the current Why3 Coq back-end can easily produce ill-typed Coq code - see §7.3), we could build this system into a less deeply embedded one, likely by relating our semantics to idiomatic Coq terms via MetaCoq. We discuss this further in §7.6.

3.5 Related Work

Some work on mechanizing logic in proof assistants aims at formalizing deep metatheorems of first-order logic and its proof systems, such as completeness [51] and incompleteness [86]. Other recent work has focused on formalizing the much richer logics used in proof assistants. Barras [14] builds a denotational model for much of the Calculus of Inductive Constructions inside the Coq proof assistant while Anand et al. [6] build a PER model for Nuprl in Coq. Roßkopf et al. [96] formalize the higher-order logic used in the Isabelle proof assistant and give a verified proof checker, while Candle [3] is a fully verified HOL Light implementation written in CakeML.

Arguably the most sophisticated of these efforts is MetaCoq [102], 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 predicates (e.g. positivity). However, the MetaCoq project's goals are orthogonal to ours. It 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. MetaCoq also omits exhaustiveness checking (see §4.3). 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. The normalization limitation has been overcome for a simpler type theory without general inductive types [4]; we discuss possible ways to use ideas from our semantics in MetaCoq in §7.5.

Chapter 4

A Verified Pattern Matching Compiler

In this chapter, we present the first formalization and mechanized proof of a sophisticated pattern matching compiler. The problem of compiling pattern matches to simpler constructs like decision trees is extremely well-studied (§4.3), largely with the goal of developing efficient compilation techniques for ML and Haskell. Why3's compiler is based on pattern matrix decomposition and is very similar to the techniques of Le Fessant and Maranget [71] and Maranget [80], which also form the basis of the algorithm used in the OCaml compiler. We describe Why3's algorithm in detail, prove its soundness, detail how it is used to implement exhaustiveness checking (which differs somewhat from existing techniques in the literature [79]), formulate a robustness property, and describe an exhaustiveness bug we discovered in Why3. In the next chapter, we will additionally prove the soundness of the Why3 transformation that compiles all pattern matches before ADT axiomatization. Though we implement the exact algorithm found in Why3, the technique is quite general and many of our proofs could be reused in other contexts.

```
match t_1 , ... , t_n with
| p_{1,1} , ... , p_{1,n} → a_1
| p_{2,1} , ... , p_{2,n} → a_2
| ...
| p_{m,1} , ... , p_{m,n} → a_m
end
```

$$\begin{pmatrix} p_{1,1} & p_{1,2} & \dots & p_{1,n} & a_1 \\ p_{2,1} & p_{2,2} & \dots & p_{2,n} & a_2 \\ & & \vdots & & \\ p_{m,1} & p_{m,2} & \dots & p_{m,n} & a_m \end{pmatrix}$$

Figure 4.1: Simultaneous pattern matching and the resulting pattern matrix

4.1 An Algorithm for Compiling Pattern Matches

As is standard [71, 80], we view the (simultaneous) pattern match as a matrix P (Figure 4.1). Note that we will refer to the last column in the matrix as "actions"; they can be terms or formulas. We next define several matrix decompositions. First, specialization for a constructor c, denoted $S(c, P)$, gives the remaining matrix assuming that the first term we match on (t_1) is an instance of constructor c. Intuitively, we can remove all rows beginning with a non-c constructor, replace c with its k arguments, replace a wildcard with k wildcards, and split a disjunction into two further cases (where $\oplus$ denotes concatenation, i.e. gluing one matrix above the other):

Pattern $p_{j,1}$	Row(s) of $S(c,P)$
$c(q_1, \dots, q_k)$	$\begin{pmatrix} q_1 & \dots & q_k & p_{j,2} & \dots & p_{j,n} & a_j \end{pmatrix}$
$c'(ps)$, $c \neq c'$	None
$\_$	$\begin{pmatrix} \_ & \overset{k}{\dots} & \_ & p_{j,2} & \dots & p_{j,n} & a_j \end{pmatrix}$
$(q1 \mid q2)$	$S\left(c, \begin{pmatrix} q_1 & p_{j,2} & \dots & p_{j,n} & a_j \end{pmatrix}\right) \oplus S\left(c, \begin{pmatrix} q_2 & p_{j,2} & \dots & p_{j,n} & a_j \end{pmatrix}\right)$
x	$\begin{pmatrix} p_{j,2} & \dots & p_{j,n} & \textbf{let } x = t_1 \textbf{ in } a_j \end{pmatrix}$
$p \ as \ x$	$S\left(c, \begin{pmatrix} p & p_{j,2} & \dots & p_{j,n} & \textbf{let } x = t_1 \textbf{ in } a_j \end{pmatrix}\right)$

This is a bit messy; we will show later that we can first eliminate disjunctions, variables, and as-patterns in the first column before computing $S(c, P)$, reducing the definition to the following:

compile(tl, P)

1. If P is empty (no rows), return "Non-exhaustive".
2. Otherwise, if tl is empty, then return the first action in P.
3. Otherwise, let $t :: tl$ be the term list. Simplify the matrix P so that the first column only consists of constructors and wildcards. There are 3 cases:
   (a) If there are no constructors in the first column, return compile(tl, $D(P)$).
   (b) Otherwise, if $t = cs(al)$ for constructor cs, if cs is in the first column, return compile($al \oplus tl, S(cs, P)$), [1] else return compile($tl, D(P)$).
   (c) Otherwise, let $base$ be $[\ ]$ if all constructors for the ADT occur in the first column and $[\_ \rightarrow$ compile$(tl, D(P))]$ otherwise. Then construct list ps as follows: for each constructor cs in the first column, add $c(vs) \rightarrow$compile($vs \oplus tl, S(cs, P)$), where vs are fresh variables. Finally, construct the pattern match **match** t **with** $ps \oplus base$ **end**.

Figure 4.2: Compiling pattern matches, given pattern matrix P and term list tl

Pattern $p_{j,1}$	Row of $S(c, P)$
$c(q_1, \ldots, q_k)$	$\begin{pmatrix} q_1 & \ldots & q_k & p_{j,2} & \ldots & p_{j,n} & a_j \end{pmatrix}$
$c'(ps)$, $c \neq c'$	None
$\_$	$\begin{pmatrix} \_ & \overset{k}{\ldots} & \_ & p_{j,2} & \ldots & p_{j,n} & a_j \end{pmatrix}$

The second matrix we need is the default (D) matrix: this gives the result assuming that t does not match *any* of the constructors appearing in the first column. We show only the definition for the simplified first column (i.e. only constructors/wilds), which removes all constructors:

Pattern $p_{j,1}$	Row(s) of $D(P)$
$c(ps)$	None
$\_$	$\begin{pmatrix} p_{j,2} & \ldots & p_{j,n} & a_j \end{pmatrix}$

With these decompositions, we can define the compile algorithm (Figure 4.2). We demonstrate the algorithm on the following example that demonstrates both nested and simultaneous matching:

```
match l₁ , l₂ with
| [] , [] → x₁
| [_] , _ → x₂
| _ :: _ , _ → x₃
| [] , _ :: _ → x₄
end
```

$$\begin{pmatrix} \mathsf{nil} & \mathsf{nil} & x_1 \\ \mathsf{cons}(\_, \mathsf{nil}) & \mathsf{nil} & x_2 \\ \mathsf{cons}(\_, \_) & \_ & x_3 \\ \mathsf{nil} & \mathsf{cons}(\_, \_) & x_4 \end{pmatrix}$$

The first column contains both constructors; the specialized matrices are:

$$P_1 := S(\mathsf{nil}, P) = \begin{pmatrix} \mathsf{nil} & x_1 \\ \mathsf{cons}(\_, \_) & x_4 \end{pmatrix} \qquad P_2 := S(\mathsf{cons}, P) = \begin{pmatrix} \_ & \mathsf{nil} & \mathsf{nil} & x_2 \\ \_ & \_ & \_ & x_3 \end{pmatrix}$$

l_1's constructor is unknown, so applying Step 3c results in the following partial match:

```
match l₁ with
| [] → compile(l₂ , P₁)
| y₁ :: y₂ → compile ([y₁; y₂; l₂], P₂)
end
```

We focus on the first case. P_1 again contains both constructors; the specialization matrices are $P_3 := S(\mathsf{nil}, P_1) = \begin{pmatrix} x_1 \end{pmatrix}$ and $P_4 := S(\mathsf{cons}, P_1) = \begin{pmatrix} \_ & \_ & x_4 \end{pmatrix}$. Step 3c results in the match:

```
match l₂ with
| [] → compile([] , (x₁))
| y₃ :: y₄ → compile ([y₃; y₄], (_ _ x₄))
end
```

Each call to compile can be simplified easily. The first evaluates to x_1 by Step 2, while the second evaluates to x_4 by repeated applications of Step 3a. The second case (on P_2) is broadly similar; we omit the details but show the full compiled pattern match in Figure 4.3. Note that the column-ordered approach of the algorithm ensures that case splitting occurs in the order of the arguments. Different orders could result in different sized (but semantically equivalent)

[1]The Why3 implementation reverses al (and likewise vs in Step 3c and the $q_1 \ldots q_k$ in the constructor case for $S(c, P)$). In the Coq development, there is some additional complexity in reasoning about the reversals. For simplicity of presentation, we show the non-reversed version.

```
match l1 with
| [] → match l2 with
        | [] → x1
        | y3 :: y4 → x4
        end
| y1 :: y2 → match y2 with
        | [] → x2
        | y5 :: y6 → x3
        end
end
```

Figure 4.3: Compiled pattern match

matches, and there has been significant research (e.g. [80]) in developing techniques and heuristics to give small outputs.

We also demonstrate how the algorithm checks exhaustiveness. Suppose we had not included the last (x_4) case in the original pattern match. Then, P_1 (i.e. $S(\mathsf{nil}, P)$) is $\begin{pmatrix}\mathsf{nil} & x_1\end{pmatrix}$, and the first match again results in $\mathsf{compile}(l_2, P_1)$. Here we are again in Step 3c, but $S(\mathsf{cons}, P_1)$ is empty, triggering Step 1.

Now we fill in several key implementation details missing from Figure 4.2. First, we describe how to remove all non-constructor, non-wildcard patterns from the first column. We use the following simplify transformation (when matching on term t_1). Note that $\overline{p}$ represents the rest of the columns in the matrix and $\oplus$ again represents concatenation.

Row of P	Row(s) of $\mathsf{simplify}(P)$
$\begin{pmatrix}c(al) & \overline{p} & a\end{pmatrix}$	$\begin{pmatrix}c(al) & \overline{p} & a\end{pmatrix}$
$\begin{pmatrix}\_ & \overline{p} & a\end{pmatrix}$	$\begin{pmatrix}\_ & \overline{p} & a\end{pmatrix}$
$\begin{pmatrix}(q_1 \vert q_2) & \overline{p} & a\end{pmatrix}$	$\mathsf{simplify}\begin{pmatrix}q_1 & \overline{p} & a\end{pmatrix} \oplus \mathsf{simplify}\begin{pmatrix}q_2 & \overline{p} & a\end{pmatrix}$
$\begin{pmatrix}x & \overline{p} & a\end{pmatrix}$	$\begin{pmatrix}\_ & \overline{p} & \textbf{let } x = t_1 \textbf{ in } a\end{pmatrix}$
$\begin{pmatrix}p_1 \; as \; x & \overline{p} & a\end{pmatrix}$	$\mathsf{simplify}\begin{pmatrix}p_1 & \overline{p} & \textbf{let } x = t_1 \textbf{ in } a\end{pmatrix}$

In effect, this transformation expands the disjunctions into two separate rows and replaces variables occurring at the outer level with let-bindings in the action column (note that it does not recurse inside constructor patterns). Of course, it is inefficient to scan over the pattern matrix separately to simplify the matrix, again to construct $D(P)$, and once again per constructor c to compute $S(c, P)$. The Why3 implementation combines these into a single scan; we prove that this is equivalent to the decomposition presented. All our proofs operate on the decomposed algorithm; it is much easier to reason about the matrices S and D when assuming that disjunctions and as-patterns (which induce recursion within patterns) do not appear.[2]

The only remaining implementation detail concerns how to check in Step 3c that all constructors for the ADT appear. The Why3 version does this in 2 ways: either the caller provides a function to retrieve the list of constructors for an ADT or compile uses metadata from the constructor function symbols (the number of siblings of the constructor). The former requires context information but is useful for reporting unmatched cases (which we do not consider), as it can identify all remaining constructors. Why3Sem includes the constructor metadata and the appropriate typing rules; we prove that the two approaches are equivalent in any well-typed context.

4.1.1 Termination

The first difficulty in formalizing compile is proving termination – the function recurses on matrices $S(c, \mathsf{simplify}\ \mathsf{P})$ and $D(\mathsf{simplify}\ \mathsf{P})$, which are not structurally smaller than input P. We first consider several natural measure candidates that fail:

- We might hope that the algorithm decreases the total size of P in each step (i.e., the sum of the sizes of all patterns in the matrix). Unfortunately, simplify expands disjunctions, resulting in a larger matrix.
- Instead, we could try to construct a measure that decreases enough when breaking apart a disjunction to offset the increase due to the new rows. For example. consider

[2]Another reason our decomposition is helpful is that the single-pass version is not structurally recursive, in contrast to the definitions of S, D, and simplify presented. We can use Equations and a simple size metric to define the function, but Coq reasoning is often simpler over structurally recursive functions.

the measure $\sum_{r \in P} 2^{|r|}$ – i.e. raise 2 to the power of the size of each row. Since $|(p_1|p_2)| = 1 + |p_1| + |p_2|$, the size of the row decreases from $2^{1+|p_1|+|p_2|} * rest$ to $(2^{|p_1|} + 2^{|p_2|}) * rest$. However, while this ensures that $|\mathsf{simplify}(P)| < |P|$, it makes $S(c, P)$ potentially larger. Specifically, each wildcard becomes k wildcards, increasing the size of the row by a factor of 2^{kc}, where $|\_| = c$.

- We might then hope to offset this by similarly decreasing our measure more when breaking apart constructor applications (since $S(c, P)$ is only computed if c appears in the first column). However, for any additive decrease in measure (i.e. any k such that $|c(al)| = k + \sum_{a \in al} |a|$), it is easy to show that the multiplicative increase from the wildcards can be larger.
- Finally, we note that if $|\_| = 0$, then $2^{kc} = 1$, so the size of S does not increase. But then $|D(P)| = |P|$ if the first column has no constructors.

We note that the difficult part of finding a termination metric is the interleaving of expansion (in simplify and the wildcard case in $S(c, P)$) and contraction (in the constructor case of $S(c, P)$ and in $D(P)$). If there were no disjunctions, the algorithm would clearly terminate – the pattern matrix size is a decreasing measure as long as the constructor measure decreases by enough to offset the increase from the additional wildcards (this can be statically bounded). So a (much less efficient) variant of compile terminates: first expand all disjunctions, then compile. While it is not completely obvious that this upper bounds the number of steps of compile, we use this as intuition to design a measure: the sum of the pattern sizes of the fully expanded matrix (parameterized by a constant representing the constructor decrease).

More formally, we give the definition of the full expansion of a pattern matrix in Figure 4.4. The difficult case is for constructor patterns: we must expand the entire row of arguments, which involves expanding all patterns in the row and concatenating all the resulting expansions. We then define a size function $| \cdot |_n$ on patterns, pattern-lists, and pattern-matrices, where n is a parameter describing the extra fuel added to the constructor case:

$$
\begin{aligned}
E^p &: \textsf{pat} \to \textsf{list pat} \\
E^p(\_) &= [\_] \\
E^p(x) &= [\_] \\
E^p(p|q) &= E^p(p) \oplus E^p(q) \\
E^p(p \; as \; x) &= E^p(p) \\
E^p(c(ps)) &= \oplus_{ps' \in E^r(ps)} [c(ps')]
\end{aligned}
$$

$$
\begin{aligned}
E^r &: \textsf{list pat} \to \textsf{list (list pat)} \\
E^r([]) &= [[]] \\
E^r(p :: ps) &= \bigoplus_{p' \in E^p(p), ps' \in E^r(ps)} [p' :: ps']
\end{aligned}
$$

$$
\begin{aligned}
E^P &: \textsf{list (list pat)} \to \textsf{list (list pat)} \\
E^P(P) &= \bigoplus_{r \in P} [E^r(r)]
\end{aligned}
$$

Figure 4.4: Expansion of patterns (E^p), rows (E^r), and pattern matrices (E^P)

$|c(ps)|_n = n + \sum_{p \in ps} |p|_n$. To give a suitable upper bound, we let m be the largest number of arguments that *any* constructor in the matrix P takes. The total amount of added measure is bounded by $\text{len}(P') * m$, where $\text{len}(P')$ is the length (i.e. number of rows) of the current matrix P'. Since P' might be larger than P (due to simplify), we can upper bound $\text{len}(P')$ by $\text{len}(E^P(P))$. We let $b(P) = \text{len}(E^P(P)) * m$. Thus, the full termination metric is $|E^P(P)|_{b(P)+1}$. Proving that compile terminates according to this measure involves 4 key steps:

1. First, we show that simplifying the matrix does not change the full expansion: $E^P(\textsf{simplify}(P)) = E^P(P)$.
2. Then, we show that the measure decreases in the default case: for any n, $|E^P(D(P))|_n < |E^P(P)|_n$. This is not too hard to show, as either constructors or wildcards are removed when constructing D.
3. The most difficult step is to show that the measure decreases in the specialization case. Formally, we prove that if constructor c appears in the first column of P and takes k arguments, $|E^P(S(c, P))|_n + n \le |E^P(P)|_n + \text{len}(E_P(P)) * k$. The proof is quite tricky, as the definition of E_p for constructors necessitates reasoning about all possible (nested) expansions. Nevertheless, this allows us to instantiate n with a large enough value for a total decrease in potential.
4. Finally, since the concrete instantiation of n is $(b(P) + 1)$ which depends on P (and

thus changes throughout the algorithm), we prove a monotonicity property: if $n \leq m$, $|P|_n \leq |P|_m$. We also show that b does not decrease: $b(D(P)) \leq b(P)$ and $b(S(c, P)) \leq b(P)$.

We combine these results to prove that compile terminates. However, none of this was specific to compile – we effectively proved that *any* algorithm based on pattern matrix decomposition terminates via this metric. More sophisticated algorithms [80, 71] follow the same basic structure but with more optimizations and heuristics (e.g, not always examining the first column, swapping columns, data structure sharing, etc). Our termination metric should suffice in these settings with almost identical proof.

4.1.2 Defining compile in Coq

We can define compile with well-founded recursion using the Equations framework. The Coq definition follows Figure 4.2 closely and has type list (term * vty) $\rightarrow$ list (list pattern * A) $\rightarrow$ option A, where A is the type of actions (in our case terms or formulas). compile is also parameterized by functions indicating how to find ADT constructors, how to construct a pattern match, and how to construct a let-binding for A. The return type represents either an ill-typed or non-exhaustive pattern match (None) or a successful compilation (Some a). As discussed above, we include an additional parameter indicating which method to use to determine if all constructors are present, but we prove the two equivalent for well-typed arguments. Finally, we include an additional parameter governing the behavior of Step 3b; see §4.2.3.

To prove properties of such a function, we define several custom induction principles (the ones generated by Equations are insufficient) that allow us to prove separately that the property of interest is preserved under simplify, and then assume the matrix is simplified when proving the cases involving $D(P)$ and $S(c, P)$ – this avoids nested induction. In our approach, proving properties about compile can be decomposed into (1) proving the property is preserved by simplify (2) proving the needed result for $D(P)$ for simplified (and well-typed)

$$\llbracket al, [] \rrbracket_v^M = \textsf{None}$$
$$\llbracket al, (r,t) :: rs \rrbracket_v^M = \textsf{orep}\ v\ t\ \llbracket al, r \rrbracket^R\ \llbracket al, rs \rrbracket_v^M$$

Figure 4.5: Semantics of pattern-matrix matching ($\llbracket \cdot \rrbracket_{M,v}$)

P, (3) proving the needed result for $S(c, P)$ for simplified (and well-typed) P, and (4) proving each step (1, 2, 3a, 3b, 3c) separately assuming all S and D cases hold; typically 3c is the only interesting one.

4.2 Properties of Pattern Matching Compilation

4.2.1 Soundness of compile

We want to prove the semantic correctness of compile – a theorem of the form: if compile(tms, P) gives term (or formula) t, then t is semantically equal to the original pattern match. To state this theorem, we first need to define the semantics of matching against a pattern matrix. We already defined the semantics for pattern matching a single term against a pattern (match_val_single or $\llbracket p, d \rrbracket^p$) and for determining if a pattern-row matches (matches_row or $\llbracket ps, ds \rrbracket^R$) in §3.2.2. We extend this to a pattern matrix (Figure 4.5) in matches_matrix ($\llbracket al, P \rrbracket_v^M$), which iterates through each row until it finds a match, then evaluates the matched action under the valuation v extended with the new bindings (note that this reduces to match_rep for a single-column matrix). Finally, we define $\llbracket ts \rrbracket_v^T$ to be the (heterogeneous) list generated by $\llbracket t \rrbracket_v^t$ for each t in ts (recall that $\llbracket t \rrbracket_v^t$ (term_rep – §3.1) is the interpretation for terms under valuation v). Then the correctness theorem is:

Theorem 4.2.1 (compile_correct)**.** *Let ts be a list of term * vty, let P be a pattern matrix, and let v be a valuation. Suppose that ts and P are well-typed in context Γ (subsequently, Γ will be implicit). Also suppose that there are no variable names in common between the free variables of ts and the pattern variables of P. Then, if compile ts P = Some tm, then*

$[\![[\![ts]\!]_v^T, P]\!]_v^M = \textsf{Some}\ [\![tm]\!]_v^t$.

This theorem says that if compile succeeds and produces a term t, then the pattern match succeeds and results in a term semantically equivalent to t. The proof is quite complex; we give a sketch in some detail.

First, we prove that simplify preserves the semantics. This is not trivial, first because a row that begins with a disjunction becomes multiple rows in the resulting matrix; we must prove that the semantics of the resulting matrix is equivalent to that of the original row. Additionally, since simplify transforms variable patterns into let-bindings, this changes a simultaneous binding into an iterated one, causing problems if variable names overlap. Consider the following example:

```
match y, z with
| x, y → f x y
end
```

The result should be equivalent to f y z. However, since simplify is an iterated let-binding, it results in **let** y = z **in** (**let** x = y **in** f y x), which is semantically equivalent to f z z. To avoid this, we require the condition on variables names in Theorem 4.2.1 (which is not a huge burden, as we can always α-convert; see §5.4). Under this condition, we can prove that simplification does not change the semantics:

Lemma 4.2.1 (simplify_match_eq)**.** *Assume that $t :: ts$ and P have no variable names in common. Then* $[\![[\![t :: ts]\!]_v^T, \textsf{simplify}\ t\ P]\!]_v^M = [\![[\![t :: ts]\!]_v^T, P]\!]_v^M$

Next, we must reason about the matrices $D(P)$ and $S(c, P)$. It will be helpful to define the notion that "term t is semantically equivalent to $c(al)$" (that is, $[\![t]\!]_v^t = [\![c]\!]^\lambda(al)$), where c is a constructor in ADT a; we call this predicate tm_semantic_constr. Then we prove that, given any term of ADT type a, we can find the constructor c and arg_list al such that tm_semantic_constr $t\ c\ al$. This is a straightforward application of find_constr_rep (§3.2.1), but it gives a nicer abstraction with which we prove several intermediate results that help us reason about the matrix decompositions. In all the following, we assume the input matrix is simplified and well-typed. First, we show that if term t is semantically equal to $c(al)$, matching t against pattern $c(ps)$ is the same as matching al against ps:

Lemma 4.2.2 (match_val_single_constr_row)**.** *Suppose* tm_semantic_constr *t c al. Then* $[\![c(ps), [\![t]\!]^t_v]\!]^p = [\![al, ps]\!]^R$.

The proof is straightforward, relying on injectivity and disjointness properties of constructor interpretations. Similarly, we then prove that if tm_semantic_constr holds of a different constructor, then the term does not match:

Lemma 4.2.3 (match_val_single_constr_nomatch)**.** *Suppose $c \neq c$' and* tm_semantic_constr *t c al. Then* $[\![c'(ps), [\![t]\!]^t_v]\!]^P =$ *None.*

Along with some additional lemmas about concatenation of pattern-rows, we now have all the pieces to prove the D and S cases; we start with the simpler D result. Recall that $D(P)$ is intended to represent the remaining pattern match if the term t does not match any of the constructors in the first column of P. We prove this property in 2 parts: either t has ADT type and matches a constructor that does not appear in the first column, or t does not have ADT type. The following lemmas state each such case:

Lemma 4.2.4 (default_match_eq)**.** *Suppose that* tm_semantic_constr *t c al but c does not appear in the first column of pattern matrix P. Then* $[\![[\![t :: ts]\!]^T_v, P]\!]^M_v = [\![[\![ts]\!]^T_v, D(P)]\!]^M_v$.

Lemma 4.2.5 (default_match_eq_nonadt)**.** *Suppose t does not have ADT type. Then* $[\![[\![t :: ts]\!]^T_v, P]\!]^M_v = [\![[\![ts]\!]^T_v, D(P)]\!]^M_v$.

The proofs are easy; the first follows by induction over the pattern matrix and Lemma 4.2.3, while the second uses the fact that there can be no constructors in the first column of P by typing.

We prove a similar lemma for the S matrix. Recall that $S(c, P)$ is intended to represent the remaining pattern match if the term t matches constructor c. Unlike D, the remaining match is not just the rest of the rows; rather, we must first match the arguments of the c-patterns. The formal statement is:

Lemma 4.2.6 (spec_match_eq)**.** *Suppose* tm_semantic_constr *t c al holds. Then* $[\![[\![t :: ts]\!]^T_v, P]\!]^M_v = [\![al \oplus [\![ts]\!]^T_v, S(c, P)]\!]^M_v$.

We reason by induction on P. In the case where the first row starts with $c(ps)$, we decompose the pattern-row matching concatenation, using Lemma 4.2.2 to reason about the $c(ps)$ match. The case where the first row starts with $c'(ps)$ ($c \neq c'$) is similar; we use Lemma 4.2.3. Lastly, in the wildcard case, we prove that matching a row against all wildcards gives Some $\emptyset$.

Finally, we have all the pieces needed for the proof of Theorem 4.2.1. The full proof is quite complex; we give a sketch of an interesting case.

Proof. We focus on the last inductive case (Step 3 of the algorithm). We want to prove that, for any v, $[\![[\![t :: ts]\!]^T_v, P]\!]^M_v =$ Some $[\![tm]\!]^t_v$ (it is important that the valuation v is generalized for induction).

In Step 3a, the first column consists only of wildcards. Thus, we can use Lemmas 4.2.4 and 4.2.5 depending on whether term t has ADT type. We only prove Step 3c; 3b is simpler and uses many of the same ideas. Since there must be at least one constructor in the first column of P, t (the first term in the term list) has ADT type. Thus, we can find a constructor c and arguments al such that tm_semantic_constr t c al holds. Furthermore, by the definition of compile, we know that $[\![tm]\!]^t_v =$ match_rep t $(ps \oplus base)$, where ps and $base$ are defined as in Figure 4.2. We consider 2 possible cases:

1. Assume c appears in the first column of P. Then, we can split ps into $ps_1 \oplus (c(vs) \to$ compile$(vs \oplus ts)) \oplus ps_2$ such that c does not appear in the patterns in ps_1 and ps_2.[3] Recall that match_rep works by iterating over the pattern list until a match succeeds.

[3]Note that the return type of compile is option A, not A. In reality, there is a monadic bind everywhere, and if any compile case fails, so does the result. Since we assume the overall result succeeds, all recursive calls also give Some, so via abuse of notation, we ignore the option type.

By Lemma 4.2.3, no pattern in ps_1 matches t. The first successful match is therefore against the $c(vs)$ term: we simplify first by invoking Lemma 4.2.2 and then by noting that since vs are variables, each inner match succeeds. Furthermore, we know that the resulting valuation is m, where each v in vs maps to the corresponding element of al. Thus, letting tm_1 be such that $\mathsf{compile}(vs \oplus ts) = \mathsf{Some}\ tm_1$ (see previous footnote), we have that $[\![tm]\!]^t_v = [\![tm_1]\!]^t_{m\cup v}$. By the induction hypothesis, we have that $[\![[\![vs \oplus ts]\!]^T_{m\cup v}, S(c,P)]\!]^M_{m\cup v} = \mathsf{Some}[\![tm_1]\!]^t_{m\cup v}$. Finally, we note (after splitting the concatenation in $[\![\cdot]\!]^T$) that $[\![vs]\!]^T_{m\cup v} = al$, since m maps $vs \to al$. The vs variables are fresh (and hence do not appear in ts or $S(c,P)$); therefore $[\![ts]\!]^T_{m\cup v} = [\![ts]\!]^T_v$ and so the desired equality holds by Lemma 4.2.6.

2. If c does not appear in the first column of P, things are much simpler. None of the ps can match by Lemma 4.2.3; therefore $[\![tm]\!]_v = \mathsf{match\_rep}\ t\ base$. Crucially, $base$ cannot be empty since there is at least one constructor not present in the first column of P, so there must be a wildcard to match – the complete result follows from Lemma 4.2.4 and the induction hypothesis.

□

Our work is the first formal, machine-checked proof of such a compilation scheme based on pattern matrix decomposition. Maranget [80] gives a brief argument for correctness but omits many details and proves the algorithm correct against a more restrictive syntactic definition of matching, which we discuss in the following section.

4.2.2 Exhaustiveness Checking

An exhaustiveness check follows as an immediate corollary of Theorem 4.2.1.

Corollary 4.2.1. *Under the assumptions of Theorem 4.2.1, if* $[\![[\![ts]\!]^T_v, P]\!]^M_v = \mathsf{None}$, *then* $\mathsf{compile}\ ts\ P = \mathsf{None}$.

In other words, if there is an interpretation such that the match is semantically non-exhaustive, then compile will correctly fail (return None). However, this is a fairly weak specification: compile could *always* return None and satisfy Theorem 4.2.1 (and hence Corollary 4.2.1).

It is worthwhile to compare and contrast this with existing pen-and-paper proofs of a virtual identical scheme to prove non-exhaustivness by Maranget [79]. Maranget discusses two versions of exhaustiveness checking: for ML-based call-by-value languages and for Haskell-like lazy languages. Both theorems are very similar, and can be summarized as, "the exhaustiveness check on pattern matrix P of types tys returns 'non-exhaustive' iff there is a value list vs of type tys such that vs does not match P" (where matching is defined as a syntactic relation on value-vectors). The difference is that lazy values also include an unknown value Ω and the exhaustiveness check must be generalized by a *disambiguating predicate*.

Such a theorem differs from Corollary 4.2.1 in several ways. First, compile checks exhaustiveness with respect to a particular input term list; it can prove that **match** Nil **with** | Nil → _ **end** is exhaustive (though, as we will see in §4.2.3, this turns out to be too permissive in other contexts). Meanwhile, Maranget's theorems reason about pattern matrices that match *any possible value list* of the correct types, a stricter condition.

Our setting is more general than that of call-by-value matching: Maranget's simple semantics for ML-style pattern matches are restricted to *values* whereas compile (and its theorems) deal with general *terms*. In a logic like P-FOLDR with no notion of reduction or operational semantics, there is similarly no notion of values. Our theorems must allow open terms, terms containing uninterpreted symbols, and other non-value terms. For example, suppose we are given a pattern match over uninterpreted function foo(x) : list int:

```
match (foo x) with
| Nil → e1
| Cons x1 x2 → e2
end
```

We don't know which case will match, but we know (by the ADT properties of §3.2.1) that under any inteteptation and valuation, $[\![\mathsf{foo}]\!]^{\lambda}[\![x]\!]_v^t = c(al)$ for some constructor c – thus, we can prove that the match is semantically exhaustive. However, Maranget's ML match relation is purely syntactic, defined by a constructor pattern matching a constructor value iff the constructors are equal and all elements pairwise match. To relate our setting to the call-by-value one, we must (1) reason about matching semantically rather than syntactically and therefore (2) quantify over interpretations, without which semantic matching is undefined. When we equip terms with a fixed interpretation/valuation, this removes the ambguity of uninterpreted symbols and free variables, allowing us to reason very similarly as in Maranger's strict matching proofs. But there is still a crucial difference due to the additional quantification: there exist interpretations under which foo is interpreted as *any* constructor; such ambiguity is not present for values.

We can view lazy pattern matching and its relation to our setting similarly. Here, Maranget's proofs require a monotonicity property so that a failing match will continue to fail as values are partially evaluated; this relies on an ordering relation on values such that Ω is smaller than all others. In our context, we do not immediately have such an ordering, evaluation, or monotonicity notion. Once again, if we reason semantically and quantify over interpretations, we can recover similar concepts. For example, we can re-interpret the ordering relation as denoting the existence of an interpretation that produces semantic equality (e.g. $v \leq w \iff \exists I, [\![v]\!]_I^t = [\![w]\!]_I^t$). This satisfies similar properties – all non-constructor terms are "smaller" than all constructors while different constructors are incomparable – but it would be very inconvenient to reason about. Therefore, we can view our proofs as broadly similar to (both of) Maranget's but significantly simpler when reasoning in a purely logical setting where the meaning of terms is given by a particular interpretation rather than by (full or partial) evaluation.

It would be possible to state and prove the reverse direction of Maranget's theorem translated to our setting: if compile returns None, there is an interpretation and term that does not match:

Theorem 4.2.2 (unproved)**.** *compile* $P =$ *None* $\implies \exists I, \exists ts, [\![[\![ts]\!]_I^T, P]\!]_I^M =$ *None.*

Equivalently, this theorem states that if matching succeeds *for all interpretations and for all terms of the correct type*, then compile should return Some. We do not formally prove such a result (soundness is sufficient for our applications), though we do need a weaker version – if the pattern match is simple (consisting only of wildcards and constructors applied to variables) and “obviously” exhaustive (either all constructors in an ADT are present or there is a wildcard), then compile always succeeds. Note that many programs in practice fall into this category (e.g. many standard functions on lists and trees).

4.2.3 Robustness of compile

As we saw in §2.2, exhaustiveness checking is included in the P-FOLDR type system. We can now give the precise condition: for any term or formula **match** t **with** ps **end**, compile [t] ps is Some.[4] This induces one additional obligation: compile produces a pattern match in Step 3c, so we must prove that this match itself passes the exhaustiveness checker. We prove this by showing that compile produces simple, obviously exhaustive patterns.

Including exhaustiveness checking in the type system causes a larger difficulty: we must show that every type-preserving function also preserves the exhaustiveness check. Writing such a theorem is surprisingly difficult; in such functions we modify:

- The term list (e.g. for substitution and rewriting in the proof system)
- The term and pattern types (e.g. for type substitution)
- The pattern match return type (e.g. in eliminate_definition, described in §5.2)
- The variables in the pattern matrix (e.g. in α-conversion)

In the end, we need a generic *robustness* theorem to prove that exhaustiveness checking still succeeds under any such changes. Of course the conditions cannot be too permissive: changing the pattern matrix arbitrarily clearly does not preserve exhaustiveness; neither does changing the types so that a constructor match no longer succeeds. In the end, the

[4]ps becomes a single-column pattern matrix.

```
theory ExhaustTest
    use export list.List

    function foo () : int =
        match (Cons 1 Nil) with
        | Cons x Nil → x
        end

    constant y : list int

    axiom y_eq: y = Cons 1 Nil

    goal foo_spec: match (Cons 1 Nil) with
                   | Cons x Nil → x
                   end = 1
end
```

Figure 4.6: Why3 violation of exhaustiveness robustness

latter turns out to be the trickier condition to formalize: we need to ensure that an ADT type cannot be transformed into a non-ADT type – this would cause a matching constructor pattern to fail. But we cannot just assume the types (or even ADT types) are the same, as type substitution can turn a type variable into an ADT type. We define an asymmetric relation ty_rel to encode this – it states that a non-ADT type can be related to an ADT type, but not the other way around; this relation holds of type substitution. To avoid the first problem (changing the patterns in the matrix), we require that the two pattern matrices have the same "shape" - i.e. they are equal up to changing variable names.[5] There is a slight subtlety: the constructor case includes the types for polymorphic type substitution: we require these types to be pairwise related by ty_rel for induction.

It is possible to state such a robustness theorem, and we will do so shortly, but unfortunately the theorem does not hold of compile. The problem occurs in Step 3b, where compile tests whether the term is a constructor application or not. If we allow the term list to change arbitrarily, it is possible that an exhaustiveness check that formerly succeeded now

[5]This is weaker than α-equivalence of patterns (§6.3.3): we ignore variable names entirely rather than requiring that relations between names (i.e. equality and inequality) are preserved.

fails. Figure 4.6 shows an example Why3 input that passes the exhaustiveness checker. In the Why3 IDE, one can interactively apply transformations. If one applies rewrite $\leftarrow$ y_eq, Why3 crashes with a fatal error, indicating that the resulting term fails the exhaustiveness check. While rewriting in such a "backwards" manner is unlikely in practice, this non-robustness also rules out potential well-typed optimizations and compiler passes like constant subexpression elimination. This is a bug in Why3 (transformations should not result in ill-typed terms), which we identified and reported.[6]

One possible fix is simply to omit Step 3b of compile. In fact, this is the common approach to exhaustiveness checking (e.g. in Coq and OCaml) and avoids this rewriting issue. However, in compile_match, the transformation that compiles all pattern matches to simple patterns before axiomatizing ADTs (§5.4), Step 3b is very important. For instance, since simultaneous pattern matching is implemented by matching on tuples, Step 3b is needed to reduce such matches. Thus, our solution is to parameterize compile by another flag simpl_constr that indicates whether Step 3b is used. In our Coq development, we prove the soundness of compile for both versions. For exhaustiveness checking, we set simpl_constr to false, so that a robustness theorem (Theorem 4.2.3) holds under all needed settings; meanwhile, in compile_match, simpl_constr should be true to make the resulting patterns as simple as possible for the eventual SMT queries. We discuss the connection between the two versions of compile in §5.4. Here, we give the full robustness theorem. t_fun_equiv is a relation that holds on two function application terms when the functions are equal, the types are pairwise related by ty_rel, and t_fun_equiv holds recursively pairwise on the argument lists; otherwise, it holds if two terms are formed by the same constructor – it rules out the backwards rewriting case above.

Theorem 4.2.3 (compile_change_tm_ps)**.** *For any pattern matrices P_1 and P_2, any term lists tms_1 and tms_2, and any type lists tys_1 and tys_2, suppose the following conditions hold:*

- *P_1 and P_2 have the same dimensions and each corresponding pattern pair has equivalent*

[6] `https://gitlab.inria.fr/why3/why3/-/issues/903`

"shapes" (but P_1 and P_2 need not have the same action type),

- *tms_1 and tms_2 have the same length (which equals the length of tys_1 and tys_2),*
- *If simpl_constr holds, then each corresponding pair of terms in tms_1 and tms_2 are related by t_fun_equiv, and*
- *Each corresponding pair of types in tys_1 and tys_2 satisfies ty_rel.*

Then if exhaustiveness checking succeeds (i.e. compile returns Some) on P_1, tms_1 and tys_1, then exhaustiveness checking succeeds on P_2, tms_2, and tys_2.

Such a theorem suffices to prove the well-typedness of the needed functions in Why3Sem, including rewriting (arbitrarily), α-conversion, and type and term substitution. The proof proceeds much like that of Theorem 4.2.1; we prove the cases for simplify, S, D, and finally compile.

4.3 Related Work

Pattern matching compilation is a very well-studied problem. Augustsson [11] presents a simple compilation scheme, introducing (implicity) some of the ideas of the matrix-decomposition approach though not yet in full generality. Laville [70] and Maranget [78] study lazy pattern matching; the latter introduces the S and D matrices and gives pen-and-paper proofs of the main compilation steps from pattern matrices to decision trees. These techniques are extended by Le Fessant and Maranget [71] and Maranget [80] to develop further heuristics and optimizations for efficient matching. Maranget [79] studies the problem of exhaustiveness checking (and useless clause identification) and proves the matrix-decomposition-based exhaustiveness check correct.

By contrast, there is very little prior work about verifying such compilation schemes. Tuerk et al. [107] implement a pattern matching compiler for the HOL proof assistant (as HOL does not natively include pattern matching) using a generic approach aiming to produce patterns smaller than the standard decision tree method and closer to the hand-written versions. The authors do not prove the compilation correct but extend CakeML's [66] proof-producing code generator to compile HOL pattern matches to ML; this results in a certificate that the compiled pattern match (in CakeML) matches the semantics of the compiled HOL match. CakeML itself provides a simple verified pattern matching compiler and exhaustiveness checker that is much more restrictive. For example, CakeML cannot prove the following exhaustive:

```
match l with
| [] → _
| [x] → _
| x :: t → _
end
```

The CertiCoq [5] verified compiler from Coq to C does not prove anything about pattern compilation. It assumes that all patterns are already simple, relying on Coq's front-end to compile patterns before reification. It is based on the MetaCoq [102] formalization of Coq in Coq, which makes the same assumption.

Chapter 5

Verifying a Compiler from P-FOLDR to First-Order Logic

The bulk of Why3's translation to FOL consists of five main transformations, each eliminating one of the complex structures – recursive functions, inductive predicates, pattern matching, ADTs, and polymorphism. Why3 also includes additional transformations (e.g. propositional simplification), which are necessary for improving SMT performance. We do not consider these additional transformations for this thesis, though they could be proved sound using similar (though simpler) techniques.

Verifying these transformations is tricky; for each one, we prove soundness and well-typing – necessary for composing transformations but nontrivial to prove under significant changes to the context. To the best of our knowledge, none of these transformations has been previously formally proved sound (other than for polymorphism; see below), but most are fairly straightforward. The exception, and the main contribution of this chapter, is our formalization and proof of correctness for the first-order ADT axiomatization, the first such mechanized proof.

We do not prove the soundness of the elimination of polymorphism (i.e. monomorphization). Thus, we compile P-FOLDR to polymorphic first-order logic, which is supported by the Alt-Ergo SMT solver. To be compatible with Z3 and CVC5, which support only many-sorted first-order logic, we would need this additional step. We note that similar monomorphization functions have been verified in Isabelle [21] and validated in Isabelle for Boogie [91]. The Why3 monomorphization procedure is based on a technique of Blanchette et al. [20] and it would be possible to prove its soundness in Coq using our methods.

5.1 A Framework for Proving Soundness

While the transformations vary, the soundness proofs broadly follow a similar structure. Given a transformation, we want to prove that if $\Gamma', \Delta' \vDash g'$, then $\Gamma, \Delta \vDash g$ (where primes represent the transformation result). That is, we want to prove that for every full interpretation I over Γ, if $I \vDash \Delta$, then $I \vDash g$. Then, we construct interpretation I' over Γ', and we must show the following:

Property 5.1.1. *I' is a full interpretation for Γ'.*

Property 5.1.2. *If $I \vDash \Delta$, then $I' \vDash \Delta'$.*

Property 5.1.3. *If $I' \vDash g'$, then $I \vDash g$.*

From this framework, different types of transformations induce different obligations:

- If the only change to Γ is replacing concrete definitions with abstract ones (e.g. eliminating recursive functions and inductive predicates), $I' = I$ and showing that I is a full interpretation in the new context is not hard.
- If the only change to Δ is adding new axioms (e.g. also for eliminating recursive functions and inductive predicates), then it suffices to prove these axioms valid to prove Property 5.1.2.
- If the transformation is a rewrite over terms and formulas (e.g. compiling pattern matches, parts of ADT axiomatization and monomorphization), then $\Gamma' = \Gamma$ and

```
function fact (n : nat)
  : nat =
  match n with
  | O→ S O
  | S n' → n * fact n'
  end
```

```
function fact nat : nat

axiom fact'def : ∀ n : nat.
  match n with
  | O→ fact n = S O
  | S n' →
    fact n = n * fact n'
  end
```

Figure 5.1: Why3 definition and result of eliminate_definition on the factorial function

$I' = I$. It suffices to prove semantic equivalence of the rewrite. Equivalence is crucial; Properties 5.1.2 and 5.1.3 require both directions of the implication to hold.

- Otherwise, under more involved changes to the context (e.g. ADT axiomatization, monomorphization), we must prove these 3 steps in full generality.

The other main task is to prove typing. Generally, the most difficult part is proving Γ' well-typed. We must also prove any added (or transformed) axioms in Δ' well-typed, and if the transformation is a rewrite, we must prove that the rewrite preserves typing, free variables, type variables, and function/predicate symbols appearing in a term – these are necessary to satisfy the typing rules for recursive functions and predicates.

5.2 Eliminating Recursive Functions

Recursive (and non-recursive) functions are replaced with an unfolding axiom in the eliminate_definition transformation. Figure 5.1 shows the axiom produced for the factorial function over a Coq-style nat. Note that this axiom is not written in the obvious way (simply setting the function equal to the body), but rather the equality is pushed through match, if, and let. When the pattern matches are eliminated, this gives terms less likely to lead to matching loops in SMT solvers, as there is more "guidance" on how to instantiate the quantifier (e.g. fact (S n') requires the instantiated term to be a successor, while fact n allows any natural number) – see [72] for more discussion.

This is a very simple transformation; indeed it is almost equivalent to the specification we proved about recursive functions (§3.2.4). We prove that this is equivalent to the "natural" axiom (without the equality pushed inside match), showing completeness as well as soundness. There is a subtle corner case: pattern matching exhaustiveness (the fact that match_rep never reaches the default case) is required to prove completeness. Thus, though pattern matching exhaustiveness is not needed for the soundness of P-FOLDR or the transformations (since all types are inhabited, it is OK to return a default element if the matching fails), it is still useful in showing stronger properties.

As discussed in §5.1, since nothing was added to the context, we set $I' = I$ and show that I is still a full interpretation for Γ'; this follows from the fact that concrete definitions have been removed but not added. Showing Property 5.1.2 requires us first to prove that the added axioms are valid; this follows from the equivalence discussed. To prove the remainder of Properties 5.1.2 and 5.1.3, we must show that the meaning of terms and formulas does not change under Γ' – this follows from the fact that no ADTs were added (or else match_rep is not preserved) and the generic equivalence lemmas of §3.2.3. Of course this would not work if we added, rather than removed, concrete definitions.

Finally, we must show that the resulting context is well-typed. If we remove all the concrete definitions and replace them with identical abstract symbols, this is not hard. However, things are not so simple because we are allowed to axiomatize a subset of recursive functions in a mutual block. To ensure that the result is still well-typed, we need to prove termination, which we do by finding the new set of decreasing indices and proving that such a calculation always succeeds on a subset of a terminating mutual block.

5.3 Axiomatizing Inductive Predicates

Inductive predicates are replaced with axioms asserting that the constructors hold and that the inversion principle is true (eliminate_inductive in Why3). Figure 5.2 shows the

```
inductive even (n: nat) =
| EvenO: even O
| EvenS: ∀ m: nat.
    even m→ even (S (S m))
```

```
predicate even nat

axiom EvenO : even(O)
axiom EvenS : ∀ m:nat.
  even(m) → even(S(S(m)))
axiom even_inversion :
  ∀ z:nat. even(z) →
  (z = O ∨ ∃ m:nat. even(m) ∧ z
     = S(S(m)))
```

Figure 5.2: even inductive predicate and result of eliminate_inductive

example for the even inductive predicate (again over a Coq-style nat).

Here, proving Γ' well-typed is easy since all inductive predicates are removed. The rest of the proof is similar to that of eliminate_definition, but the difficulty lies in proving the axioms valid. The constructor axioms essentially follow from the specification of inductive predicates (§3.2.5). But the inversion axiom is surprisingly difficult to prove sound. Here, we give a proof sketch for even; the general case is broadly similar.

By the least predicate property of even, for any P, if (1) $P(0)$ and (2) $\forall n, P(n) \rightarrow P(S(S(n)))$ hold, then $\forall x, \mathsf{even}\ x \rightarrow P(x)$ holds. We choose $P(x)$ to be:

$$P(x) := (x = 0 \vee \exists n, \mathsf{even}\ n \wedge x = S(S(n)))$$

and we must show that (1) and (2) hold. Clearly $P(0)$ holds. (2) is more interesting. Given n, assume that $P(n)$ holds. We want to show that $P(S(S(n)))$ holds. It suffices to find an n' such that even n' and $S(S(n)) = S(S(n'))$; thus, let n' be n. To prove that $\mathsf{even}(n)$ holds, we use the fact that, since even is an inductive predicate, its constructors are true.[1] Recall that by assumption, $P(n)$ holds; thus, we must show that $P(n)$ implies $\mathsf{even}(n)$.[2] We have two cases: if $n = 0$, the result follows from the first even constructor; if $n = S(S(y))$ and $\mathsf{even}(y)$ holds, then we substitute and use the second even constructor.

[1]In the general case, we use the fact that under a full interpretation, the constructors of all inductive predicates are true.

[2]In the general case, we prove this implication holds by positivity.

5.4 Compiling Pattern Matches

The transformation that eliminates complex and nested patterns, compile_match, walks over the given term or formula and calls compile (Chapter 4) on each pattern match. It is a rewrite transformation; as we discussed in §5.1, we must prove the semantics equivalent and prove several typing properties (including preservation of free variables, type variables, and used function/predicate symbols). The most interesting proof is for the semantics. The result follows from Theorem 4.2.1, the compile correctness theorem, but recall that this theorem says that if compile gives Some, then the result is semantically equivalent. Thus, we must show that all calls to compile succeed (give Some). We know by typing that all pattern matches are exhaustive, but after changing the exhaustiveness checker to satisfy robustness (§4.2.3), the exhaustiveness check and the version of compile used in compile_match differ on the value of simpl_constr. Therefore, we show a correspondence between the two versions of compile, which proves that our new exhaustiveness check (with simpl_constr set to false) is strictly more restrictive than the old one:

Theorem 5.4.1 (compile_bare_simpl_constr)**.** *Given pattern matrix P and term lists tms_1 and tms_2 such that everything is well-typed, if the exhaustiveness check succeeds on tms_1 and P when simpl_constr is false, then the exhaustiveness check succeeds on tms_2 and P when simpl_constr is true.*

The proof follows by induction, where the interesting case unsurprisingly occurs when the matched term is a constructor and thus one check is in Step 3b (of Figure 4.2), while the other is in Step 3c.[3] Here, we use the stronger exhaustiveness check to know that every constructor compilation (using $S(c, P)$) must succeed; therefore, the particular one needed also succeeds. This is not obvious: the two checks operate over different arguments – adding fresh variables and the constructor application arguments, respectively. For this reason, Theorem 5.4.1 must be general enough to allow different (but still well-typed) term lists.

[3]The simplify case is also surprisingly nontrivial and relies on robustness properties – this is the reason we need the t_fun_equiv condition in Theorem 4.2.3.

The full semantic equivalence result for compile_match requires an additional α-conversion to satisfy the unique-name hypothesis of Theorem 4.2.1.

For our purposes, semantic equivalence is not quite enough: we need more information about the result of compile_match for the axiomatization of ADTs (§5.5). Namely, we show that the resulting pattern matches are simple, obviously exhaustive, and organized in a particular way: each pattern match consists of a nonempty list of unique constructors applied to variables, optionally followed by a wildcard. This is a strong condition, and gives us a precise characterization of the resulting structure of the terms. As an implicit corollary, this proves that every (well-typed) P-FOLDR term, no matter how sophisticated the pattern match, can be reduced to an equivalent term consisting only of these very restricted matches.

5.5 A Sound First-Order Axiomatization of ADTs

Many SMT solvers do not support algebraic data types with an inductive set of constructors. When translating a language (Why3) or logic (P-FOLDR) with ADTs to SMT formulas, one must eliminate the ADTs and replace them with axiomatized abstract functions.[4] Here we prove sound the particular axiomatization that Why3 uses (eliminate_algebraic), though we will note that our approach would extend to other first-order representations.

5.5.1 Axiomatizing ADTs

Figure 5.3 shows an example of an ADT-related goal in Why3: it defines the list datatype, the length function and a proof goal involving these definitions. §5.2 showed how to axiomatize recursive definitions; the next step is to eliminate the recursive types and pattern matching, a significantly more complex transformation. Once compile_match has already reduced everything to simple patterns, the elimination of an ADT can be viewed as a 2-step process (though both happen simultaneously): first, the type and constructors are replaced

[4]Some SMT solvers do support ADT theories, but internally, they perform a similar first-order axiomatization.

```
type list 'a = Nil | Cons 'a (list 'a)

function length (l: list 'a) : int =
  match l with
  | Nil      → 0
  | Cons _ r → 1 + length r
  end

goal foo: ∀ l: list 'a.
  length (match l with | Nil → Nil | Cons x t → t end) ≤ length l
```

Figure 5.3: ADT example in Why3

by abstract symbols while new abstract function symbols and axioms are introduced; second, all pattern matches are replaced with expressions using the newly introduced function symbols.

Figure 5.4 shows the possible outputs to the transformation; we describe each part in turn. Projections describe how to extract the arguments of a constructor; there is a projection symbol and axiom for each argument of each constructor. The selector function axiomatizes (simple) pattern matches, returning the argument corresponding to the matched constructor. The indexer function describes which constructor an ADT instance belongs to. The disjointness axiom asserts that constructors are distinct, and the inversion axiom states that all elements of ADT type are equal to a constructor applied to the corresponding arguments (expressed via projections). We note that Why3 does not generate all of these axioms every time, and many are user-configurable or only required in certain situations. For example, the disjointness axiom follows from the indexer axioms, so it is only generated if the indexers are not. Nevertheless, we will prove all the generated axioms sound to cover all possible cases.

The second part of the transformation, the elimination of pattern matches, is less standard. Here, there are two cases: if the pattern match occurs in a term (as with the goal foo), the match is transformed into an expression using the selector function, with projections retrieving the appropriate constructor arguments (i.e. the variables in the pattern match). If the pattern match occurs in a formula (as with length_def), the expression **match** t **with**

```
type list 'a

function Nil : list 'a
function Cons 'a (list 'a) : list 'a

(*Projections*)
function cons_proj_1 : list 'a → 'a
function cons_proj_2 : list 'a → list 'a
axiom cons_proj_1_def: ∀ u1 u2. cons_proj_1 (Cons u1 u2) = u1
axiom cons_proj_2_def: ∀ u1 u2. cons_proj_2 (Cons u1 u2) = u2

(*Selector*)
function match_list : list 'a → 'b → 'b → 'b
axiom match_list_cons: ∀ z1 z2 u1 u2.
  match_list (Cons u1 u2) z1 z2 = z1
axiom match_list_nil: ∀ z1 z2. match_list Nil z1 z2 = z2

(*Indexer*)
function index_list : list 'a → int
axiom index_list_cons: ∀ u1 u2. index_list (Cons u1 u2) = 0
axiom index_list_nil: index_list Nil = 1

(*Disjointness*)
axiom cons_nil: ∀ u1 u2. Cons u1 u2 <> Nil

(*Inversion*)
axiom list_inversion: ∀ u.
  u = Cons (cons_proj_1 u) (cons_proj_2 u) ∨ u = Nil

(*length axiom*)
axiom length_def: ∀ l. (l = Nil → length l = 0) ∧
  (∀ u1 u2. l = Cons u1 u2 → length l = 1 + length u2)

goal foo: ∀ l.
  length (match_list l Nil (cons_proj_2 l)) ≤ length l
```

Figure 5.4: Result of axiomatizing ADTs and eliminating pattern matching

| c(a) → f ... **end**, becomes either ((∀ a. t = c(a) → f) ∧ ...) or ((∃ a, t = c(a) ∧ f) ∨ ...) depending on the polarity of the formula in which the pattern match appears.[5]

Of course, this is not the only possible ADT axiomatization, and different tools make slightly different choices. Dafny has a broadly similar encoding [72], with projections and indexers, as well as axioms defining an order on ADTs.[6] Sniper [23, 22], a tool to transform Coq goals into first-order SMT goals (see §5.6), generates injectivity, disjointness, and inversion axioms. Other first-order axiomatizations of ADTs for SMT solvers [15, 100] provide isf functions indicating to which constructor the element belongs (replacing the indexers) and may also include non-circularity axioms (e.g. that l ≠ Cons x l) .

Methods for eliminating pattern matching differ more widely. Why3, as we have seen, uses a selector axiom in some cases and directly generates formulas in the others. Dafny turns pattern matches into nested if-expressions, with one case per constructor (in some sense this can be seen as an "eager" approach compared to Why3's "lazy" approach). Sniper also does not include a selector axiom; it uses Coq to automatically simplify the pattern match per constructor.

5.5.2 Proving Soundness

We again follow the framework of §5.1. This time, the modified interpretation I' is quite difficult to define, as we must decide how to interpret the newly added function symbols.[7] Meanwhile, unlike in previous transformations, the added axioms are not necessarily true in every interpretation, but we must show that they are true under I'. It is also tricky to prove that pattern matching elimination, a rewriting step, preserves the semantics of term_rep and formula_rep – unlike with compile_match, the context Γ and the defined ADTs change,

[5]The two versions are logically equivalent, as we show, but the polarity-dependent representations mean that goals are universally quantified while hypotheses are existentially quantified.

[6]Dafny needs an explicit ordering to guard recursive function calls. In Why3's core logic, the only recursion is through (lexicographic) structural inclusion, which is checked statically. In Dafny, some termination conditions are only checked at verification time (see §7.5).

[7]Our version of eliminate_algebraic is parameterized by a predicate denoting which mutual ADTs to keep (for instance, one may want to preserve monomorphic enumeration types). In this chapter, we assume all types are being eliminated to simplify the presentation.

```
Definition proj_interp {m: mut_adt} {a: alg_datatype}
(c: funsym) (p: funsym) (i: nat)
(Hn: i <? length (s_args c))
(f_nth: nth i (projection_syms badnames c) id_fs = p)
(al: arg_list p ...) :=
(*Cast head of al to [adt_rep]*)
let x := proj_args_eq p al ... in
let (c1, al1) := find_constr_rep m a x in
match funsym_eq_dec c c1 with (*check if c = c1*)
| left Heq ⇒ dom_cast (...) (hnth i al1) (*nth elt of al1*)
| right Hneq ⇒ funs gamma_valid pd pf f srts args (*default*)
end.
```

Figure 5.5: Interpretation of projection symbols

requiring specialized reasoning. In the following sections, we show the construction of I', briefly discuss the proof that I' satisfies the axioms, and sketch the semantic preservation proof for the rewriting step; we complete the soundness proof as described in §5.1.

Defining the Interpretation

We are given a full interpretation I and ADT a. Since I is full, it correctly interprets a as the underlying W-type (though we only need the properties of §3.2.1).

Projection symbols are intended to represent extracting a component from the constructor arguments. We can give an interpretation as follows: given the ith projection symbol p of constructor c and semantic arguments al (i.e. a heterogeneous list of elements of the interpretations of the argument types of p), we know that the first type argument of p must be a, so the first (only) element of al has type $[\![a]\!]^\tau$ (here we ignore the polymorphic type arguments for simplicity) – call this element x. Therefore, find_constr_rep gives c_1 and al_1 such that $x = [\![c_1]\!]^\lambda(al_1)$. If $c = c_1$, then the interpretation returns the ith argument of al_1; otherwise, it returns some default argument. The Coq definition follows this reasoning exactly, with some additional dependent typing obligations. We show a simplified version in Figure 5.5, omitting some full definitions and arguments.

Interpreting the indexer axioms is similar. Given indexer semantic arguments al, we

```
Definition indexer_interp {m a} (al: arg_list ...):=
(*Cast head of al to [adt_rep]*)
let x := indexer_args_eq a al ... in
let (c1, _) := find_constr_rep m a x in
(*Find index of c1 in a's constructor list*)
dom_cast ... (Z.of_nat (index c1 (adt_constr_list a))).
```

Figure 5.6: Interpretation of indexer symbols

```
Definition selector_interp {m a} (al: arg_list ...) :=
let csl := (adt_constr_list a) in
(*Get [adt_rep] and remaining [arg_list]*)
let (x, al2) := selector_args_eq a al ... in
let (c1, _) := find_constr_rep m a x in
(*Find index of c1 in a's constructor list*)
let idx := index c1 csl in
(*Find corresponding elt of al2*)
dom_cast ... (hnth idx al2).
```

Figure 5.7: Interpretation of selector symbols

again know that the first (only) element of al is an ADT type, find_constr_rep again gives the corresponding constructor c, and the interpretation simply returns the index of c in the list of a's constructors. We show the simplified Coq representation in Figure 5.6.

The selector is a bit more complicated. This time, the selector symbol for ADT a takes in $n+1$ arguments, where n is the number of constructors of a (there are additional complications due to the fresh type constants; again we omit this for simplicity). Given selector semantic arguments al, we can again show that the first element of al is an ADT type and use find_constr_rep to get the constructor c (the arguments are irrelevant here). Then, we find the index i of c within a's constructor argument list and return the ith argument of al – this encodes the idea that the selector should return its $(i+1)$st argument when called on the ith constructor. Figure 5.7 shows the simplified definition.

We note that this approach extends to any first-order axiomatization we might want to give. For example, interpretations for isf predicates which determine if an ADT element belongs to a given constructor would be almost identical to those of indexers but would return a boolean rather than the integer index. More interestingly, we could use a similar approach to interpret Dafny's well-foundedness axioms (e.g. `l < x :: l`). Rather than `find_constr_rep`, the interpretation would use the well-founded relation `adt_smaller` (§3.2.4) that denotes semantic structural inclusion and which we used to prove the well-foundedness of recursive functions.

We now construct I' as the interpretation that uses the appropriate definitions for all newly added function symbols, keeping everything else the same (including the type interpretation). It is somewhat tedious to show that this is unambiguous; we rely on certain syntactic constraints to ensure that the newly generated function symbols do not overlap with any others, for example by ending the name of each class of added axiom with a different suffix.

Proving that the added axioms are satisfied by I' is quite straightforward (with some occasional complications due to dependent types) and only relies on properties of `constr_rep`. For example, to prove the inversion axiom satisfied by I', `find_constr_rep` identifies the input's constructor c, then we prove the clause in the disjunction corresponding to c by unfolding the definitions of the projection interpretations and relying on injectivity and disjointness of `constr_rep`. The disjointness axiom is even easier; it follows almost immediately from the disjointness of `constr_rep`.

Eliminating Pattern Matches

Proving the correctness of the pattern matching elimination (called `rewriteT`/`rewriteF` in Why3) is quite difficult, requiring simultaneous reasoning about two contexts (the old context Γ, and the new, ADT-less context Γ') and two interpretations (I over Γ and I' over Γ').

We give a brief sketch of an interesting case in the proof. We consider the term pattern matching case, where `match t with | ... c(vs) → e ... | _ → d end` is rewritten into `( match_foo t ... (let vs := projs t in e) ... d ... )` where the $(i+1)$st argument of `match_foo` is an iterated let expression binding vs to the projections if the ith constructor c appears in the match as $c(vs)$ and is d otherwise. Note that this rewriting relies on the fact that compile_match has already been run and thus all patterns are simple. This is extremely helpful for the proof; we can reason about pattern matching largely syntactically rather than by unfolding the (recursive) definition of match_val_single. In particular, on matched term t, if tm_semantic_constr $t\ c\ al$ holds (§4.2.1), we prove that if $c(vs) \to e$ appears in the pattern match, then match_rep evaluates to $[\![e]\!]^t_{vs \to al}$ (and otherwise match_rep evaluates to $[\![d]\!]^t$). These results crucially rely on properties of compile_match: each constructor appears at most once in the match, constructors are only applied to variables, and all matches are exhaustive.

Therefore, there are two cases. If tm_semantic_constr $t\ c\ al$ holds for $c(vs) \to e$ in the pattern match, then the match expression evaluates to $[\![e]\!]^t_{vs \to al}$. The match_foo expression, according to selector_interp, returns the argument corresponding to constructor c: the iterated let-binding `let v1 = proj1 t in ... let vn = projn t in e`, where `vs=[v1; ... ; vn]`. The equivalence follows by unfolding proj_interp and by the fact that the vs are fresh variables. Otherwise, if tm_semantic_constr $t\ c'\ al$ holds for c' not in the match, the reasoning is similar: we prove that both expressions evaluate to $[\![d]\!]^t$.

As expected, we also need a variety of results about the typing of rewriteT/F. These are trickier than for compile_match, since we must again reason in two different contexts with different datatype definitions and declared function symbols. We prove the preservation of well-typing, free variables, type variables, and function/predicate symbol occurrences. A key complication is that even the purely syntactic results (e.g. free variables) rely on well-typing, since as we have seen, rewriteT relies on exhaustiveness and simple-pattern assumptions. For example, if there is a constructor not appearing in the pattern match, rewriteT assumes a wildcard pattern is present.[8]

Proving that I' is a full interpretation is now straightforward. We note that eliminate_algebraic can only be run after eliminate_definition (enforced by Why3) or else the resulting terms are ill-typed (we can no longer prove that recursive functions terminate). We

[8]If not, the Why3 version throws an exception; our version returns some default value.

also require that eliminate_inductive has already removed all inductive predicates; this is not strictly necessary for the proofs, but it prevents us from having to reason about how eliminating pattern matches affects positivity checks. This ordering of transformations is consistent among the drivers[9] for all standard SMT solvers. However, it is not the case that all definitions are eliminated: non-recursive functions and predicates may still be in the context, and rewriteT and rewriteF will also eliminate their pattern matches. Thus, to show that I' is full for Γ', we show that the rewritten non-recursive functions and predicates still satisfy their semantic property (i.e. they are equal to their unfolding) – this follows easily from the semantic equivalence of rewriteT/F.

Finally, we complete the proof of soundness as described in §5.1. We proved Properties 5.1.1 and 5.1.3 (a straightforward application of the semantic equivalence of rewriteF). We almost proved Property 5.1.2; however, rewriteF is also run on the newly added axioms. The last piece is to show that on formulas without pattern matches or constructor applications, rewriteF has no semantic or typing effect.[10] Note again that we needed the semantic equivalence (not just soundness) of rewriteF, which is applied in both the hypotheses and the goal.

5.6 Related Work

ADT Axiomatization First-order axiomatizations of ADTs are common to support SMT theories based on ADTs [15, 100] as well as in SMT-based verification tools that include higher-level reasoning support (e.g. Dafny [72]). Other tools (e.g. Viper[77] and Gobra [112]) include ADTs as essentially syntactic sugar for the derived axioms; these tools do not include any complex pattern matching.

However, there is very little work on proving soundness of such axiomatizations. Sniper [23, 22] is a tool that turns certain Coq goals into FOL formulas to enable use of SMT solvers.

[9]Why3 specifies which transformations are run for a solver via *driver files*.
[10]Unfortunately, they are not syntactically equal because of the polarity map.

It has a very similar set of transformations: eliminating recursive functions, axiomatizing ADTs, and eliminating pattern matches (note that patterns are already simple from Coq's built-in pattern matching compilation). This approach is *certifying*: it consists of tactics that generate the resulting Coq definitions and assertions (by reifying to MetaCoq) and then proof scripts that prove the assertions hold. Such proofs are generally quite simple, since one can use Coq's built-in mechanisms for simplifying pattern matches, performing case analysis on inductive types, and doing induction. Such an approach is better suited for Coq but would not extend to SMT-based systems like Why3 or Dafny that cannot reason about ADTs except via axiomatization and translation to SMT. Our methods also prove once and for all that the ADT axiomatization is sound and succeeds on well-typed inputs; tactic-based approaches like Sniper cannot provide formal guarantees that the translation or generated proof scripts succeed.

Compilation to Many-Sorted Logic Though we do not prove anything about monomorphization, we note that unlike the other transformations, there has been significant work on both translating polymorphic logic to many-sorted FOL to build program verifiers as well as on proving such schemes correct (formally and informally). Early work included the encoding of Couchot and Lescuyer [39] in an older version of Why3 as well as Boogie's encoding of higher-rank polymorphism (more expressive than Why3) into a guard-based approach and an approach based on passing type arguments to functions [73]. Bobot and Paskevich [25] report on a subsequent version of Why3's encoding, focusing on the case where one wants to preserve some specific sorts (e.g. ints) for SMT solvers. The current Why3 implementation is based on the "featherweight tags" encoding of Blanchette et al. [20]; this is just one of many encodings presented and proved sound in the process of developing Sledgehammer. Verified encodings include those of Blanchette et al. [21] and Parthasarathy et al. [91] in their formal validation of Boogie.

Chapter 6

Foundational Why3: Towards a Why3 Coq API

Why3 is implemented in OCaml. It connects to front-ends such as Frama-C and EasyCrypt through an OCaml API and connects to back-end solvers – SMT solvers such as Alt-Ergo and Z3 as well as proof assistants like Coq – by printing the transformed Why3 proof tasks to a file to pass to the appropriate solver. In this chapter we present Foundational Why3, a Coq-based alternative that can connect to our formalization of P-FOLDR and its compilation to FOL to provide formal soundness guarantees. Foundational Why3 can run via Coq's extraction to OCaml, connecting to Frama-C and EasyCrypt and to SMT solvers through (almost) exactly the same OCaml APIs as original Why3. But it can also run inside Coq using the vm_compute mechanism. This both provides a seamless connection to Coq for subgoals to be proved interactively and, if extended with a method to call SMT solvers within Coq, would enable creation of IVL-based verifiers run entirely within a proof assistant.

In the previous chapters, we explored and mechanized the theoretical foundations of P-FOLDR and its compilation to FOL; here, we focus on the problem of creating a practical IVL implementation within a proof assistant. We describe the system we have built, including the many design decisions we made. We present a lightweight design pattern enabling us to implement and verify stateful, exception-throwing OCaml APIs in Coq and use this to implement parts of the Why3 OCaml API in Coq. Finally, we discuss how to connect this implementation to our semantics and compiler (though we do not complete all the proofs; we demonstrate on stateful substitution and the eliminate_let transformation and describe how this could be done for the rest of the transformations).

The compilation we discussed in Chapter 5 is fully computable within Coq, but it is not suitable for use as a real-world IVL. Our core syntax ignores any data without logical meaning. This is ideal for proofs, as our soundness reasoning is not polluted with unnecessary information; however, the existing Why3 OCaml implementation (which we will refer to as Why3-O) contains a significantly richer structure, including metadata for error messages, user-customizable behavior, and SMT-related data (e.g. triggers for quantifiers). Moreover, there are several key differences between our purely functional core semantics and compiler and the mostly functional (but impure) Why3 implementation; we cannot directly test our compiler against Why3 test suites and clients.

6.1 Design Goals and Decisions

First, we must determine what exactly a verified Why3 implementation should entail and how it should relate to the existing unverified tool. Figure 6.1 presents an overview of Why3-O's architecture. An external client starts by calling the Why3 logic API – this client could be the parser (if the user writes a Why3 file by hand), a tool using Why3 as a back-end (e.g. EasyCrypt or Frama-C), or the WhyML verification condition generator. This API includes facilities for constructing identifiers, types, patterns, terms, formulas, definitions, tasks, and theories, ensuring that only well-typed AST nodes can be created. If the typechecking succeeds, the API has generated a theory AST node, which split_theory will split into the constituent tasks (we implemented a simpler version of this in §3.4). Each task

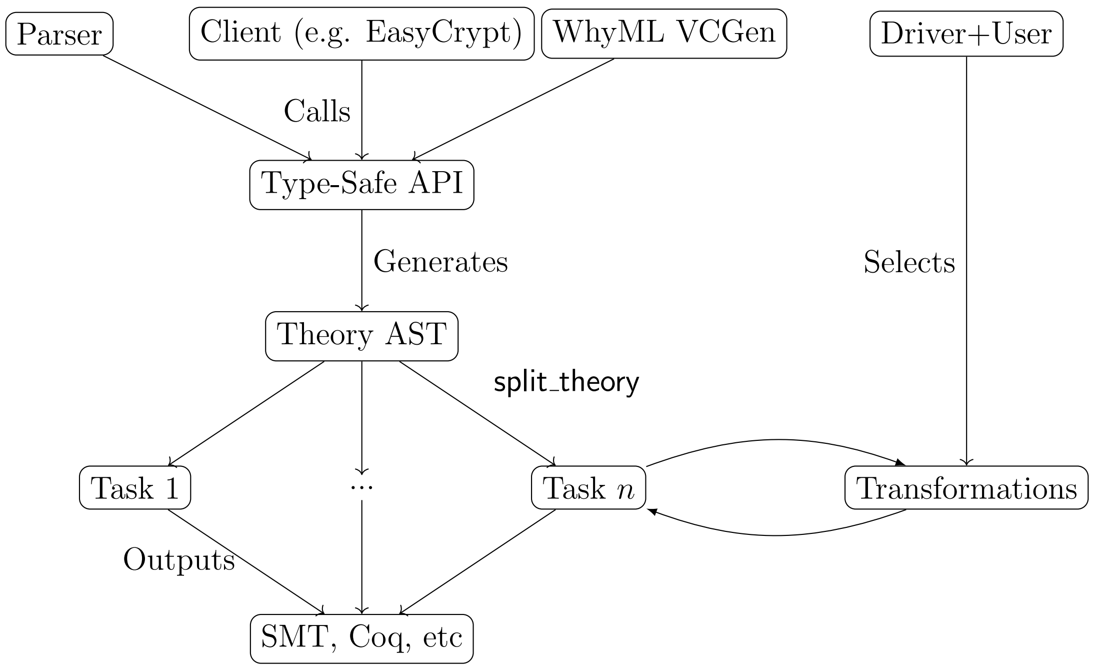

Figure 6.1: Simplified overview of Why3 IVL architecture

then undergoes a series of transformations as selected by a driver file for a particular solver[1] until it lies in the subset of P-FOLDR supported by that solver. Finally, Why3 prints the AST to a file, which it gives to the solver in question.

Verifying all of Why3 is infeasible and would include reasoning about many components that do not directly relate to the soundness of the logic compilation. We aim to produce a verified yet realistic IVL with the following design goals:

- Our priority is to develop a practical IVL implementation executable within Coq that can connect to our soundness proofs to enable IVL-based verification for foundational tools (e.g. VST).
- As a secondary goal, we would like our implementation to extract to OCaml and interoperate with existing Why3 tools and test suites. This serves as evidence that our verified subset is extensive enough to encompass real proof goals and practical

[1]This is a simplification; the user can also specify transformations to be applied, and Why3 allows proofs to be *replayed* to apply a specified series of transformations – for our purposes, it is enough that *some* input specifies which transformations should be run.

for real-world use. It would allow users of tools like EasyCrypt and Frama-C to use Foundational Why3 without changing their workflow

- When possible, we want both implementations to be efficient and idiomatic. However, we prioritize the Coq version when these goals conflict.

We identify two possible paths to these goals. First, we could extend our compiler implementation with some additional data (e.g. triggers for quantifiers) and generate an OCaml version using Coq's native extraction functionality. We could then link this extracted OCaml code with a hand-written parser in the front-end and a method to output SMT formulas in the back-end, producing an executable that would compile Why3-like tasks (or theories, if we write a split_theory function) into SMT formulas. This is the same basic approach that CompCert takes for compiling C programs.

This would give us the freedom to implement the AST and transformations however we want, and it would allow us to directly use our existing compiler implementation and proofs (with a suitable parser generating our core syntax). However, there are a number of drawbacks. This would require significant duplication of Why3's front-end and back-end. Additionally, our formalization was defined with verification in mind, and thus it makes different tradeoffs between efficiency and ease of reasoning than a real-world implementation would. Finally, this fails to satisfy the above goals, as it would not be compatible with existing Why3 tools and testing frameworks.

We choose an alternative approach: we identify certain parts of the Why3 pipeline and implement those in Coq, making an effort to remain as compatible with Why3-O as possible. We focus only on the soundness-critical path – the type-safe API and the transformations, using existing Why3 OCaml code for the rest of the pipeline.

This design has numerous potential benefits, including the ability to verify parts of Why3 *incrementally*, but we must resolve the many differences between Why3-O and our core syntax and compiler. These differences fall into two categories: some are simply design choices while other involve fundamental incompatibilities between Coq and OCaml. In §6.2,

```
type pattern = {
  pat_node : pattern_node;
  pat_vars : Svs.t;
  pat_ty   : ty;
}
and pattern_node =
| Pwild
| Pvar of vsymbol
| Papp of lsymbol * pattern list
| Por  of pattern * pattern
| Pas  of pattern * vsymbol
```

Figure 6.2: Definition of patterns in Why3

we show how we can partially resolve these latter differences.

6.1.1 Why3 vs Core Why3

AST Differences The Why3-O ASTs are richer than our core ASTs, incorporating more metadata. Beyond this, there are two key differences. First, our core syntax distinguishes between terms and formulas, whereas Why3-O does not. Rather, it has a single term datatype that includes type information; a term has type Some ty, while a formula has type None. This means that our core datatypes are slightly more restrictive, though the difference disappears for well-typed terms. Second, the Why3-O ASTs are all mutually recursive, following a particular structure; we show the pattern type (Figure 6.2) as an example. Some metadata – in this case the list of free variables and the pattern's type – is bundled together into a record alongside the "node", which has the expected recursive structure.

Typing By Design Why3-O is type-safe: the AST datatypes are opaque[2] and can only be created by smart constructors that perform typechecking (e.g. ensuring that the claimed type of a formula is None or checking termination of recursive functions). These constructors throw informative exceptions if the typechecking fails. This has several implications for the

[2]In OCaml, they are private, which exposes the constructors to enable pattern matching but does not allow the user to create an instance of the type by using a constructor.

API design. First, functions can *assume* that their inputs are well-typed. Usually this is not a problem (Why3Sem requires well-typing for transformations), but in some core functions, we do not want to assume well-typing (e.g. α-equivalence; see §6.3.3). Second, the API is used internally (e.g. in transformations), so the typechecker is called every time new AST nodes are created. This turns many bugs (e.g. the bug we found in §4.2.3) into ill-typed terms that cause Why3 to fail with an exception. Our typing proofs for transformations can be viewed as a proof that these repeated checks are unnecessary.

Unique Names and Identifiers Why3-O contains a fundamental operation to generate unique identifiers. This task is critical, as it is used to disambiguate repeated symbol names (e.g. when new hypotheses are created in transformations), to generate new variable names for α-conversion, and for hash-consing (see next paragraph). To accomplish this, the API uses mutable global counters; this ensures that the generated identifiers are short. Our core compiler also needs to generate unique names in similar places. Since Coq does not have mutable state, we implemented this as an inefficient function that finds an element not present in a set by enumeration. As we will see, this is one of the trickiest differences to resolve.

Hash-Consing And Equality The second major use of global mutable state in Why3-O is for hash-consing, an optimization in which each AST node is created with a unique tag and stored in a global hash table. Then, to create another instance of this node, the constructor first looks in the hash table and returns the stored element if present. Such an optimization reduces space usage and makes checking equality very fast – since only one instance of a given AST node can ever be created, reference equality (equality of pointers) is sufficient. By contrast, our core AST has no such notion; we only have decidable structural equality, which is much slower over large AST nodes. More broadly, hash-consing enforces a global invariant on the generated API terms – that they are uniquely determined by their tags.

Data Structures As with any large software system, Why3's performance depends on well-chosen data structures. Why3-O makes heavy use of sets, maps, and hash tables, the former implemented using balanced binary search trees. In our Coq compiler, we use extensional binary tries as implemented in the std++ library [65] – these are more efficient in Coq than BSTs [9] and have nicer properties for proofs. Luckily, this difference is smaller than it appears; while binary tries require mappings from keys to positive integers, Why3-O uses such functions anyway to implement the order relation for BSTs. However, these tags are not injective (a requirement of extensional binary tries in Coq), both because in Coq we can reason about AST nodes not created by hash-consing constructors and because in some cases different AST nodes can have identical tags (terms are hashed modulo α-equivalence).

Integers Why3-O uses a mix of machine-length integers (OCaml's `int` type) and arbitrary length integers (implemented in OCaml in the `Zarith` library). In Coq, numbers are typically implemented using unary natural numbers (for proofs) and a datatype representing unbounded binary numbers (`Z` or `positive`) for computations.[3] These are two very different kinds of data: OCaml `ints` are fast but require reasoning about overflow, while Coq datatypes are easy to reason about but are heap-allocated and require many pointer indirections, reducing performance.

6.1.2 Designing a Coq Why3 API

The above differences demonstrate fundamentally incompatible designs between Why3-O and our core syntax and verified compiler. Some of these incompatibilities are not hard to resolve – for instance, we can give an augmented AST for types, terms, etc., adding the needed metadata and defining the semantics via translation to our existing core syntax (see §6.3.1). It is more difficult to handle the features not present in Coq – exceptions, mutable state, reference equality, etc. We have two choices: we could attempt to stay close to Why3-O

[3]Recent versions of Coq add *primitive integers* that are axiomatized and built in to the kernel.

or we could make Foundational Why3 purely functional (ideally in a client-indistinguishable way).

To demonstrate the problems with the second approach, we imagine two potential modifications of the API: one that generates identifiers functionally rather than statefully and one that avoids hash-consing and uses slower equality checks. The latter would work, though it would be costly, both in severely degrading performance (see §7.2) and in requiring significant change even to unrelated parts of Why3 that assume hash-consing invariants.

The former is more problematic. It is possible to generate *canonical* identifiers for the types we need (i.e. functions that inject into the positive numbers). Then, it is trivial that two AST nodes with identical tags are identical, and we could still use hash-consing. But this approach does not scale. One example of such an injection (for numbers, strings, tuples, lists, arbitrary trees, etc) is given in the std++ Coq library; we use this in Why3Sem and the compiler. But for types with even moderate amounts of nesting (e.g. a term contains function symbols that contain types that contain strings), these values quickly grow too large for real-world computation.[4] Even strings alone are impractical: while 100,000 values can be stored within 17 bits, a 15-character string (smaller than many used in Why3) requires 120 bits under the natural (and std++) encoding.

Separately, a functional approach does not fully suffice: for α-conversion, we need fresh identifiers for the same variable. It is not obvious how to generate them efficiently without some kind of state. One possible solution is to store additional information in the terms representing the largest identifier, enabling us to quickly find a fresh one, but this would change types (visible to external clients) and make most term functions stateful. We note that this is not a problem in systems like VST, since the tool knows in advance all needed identifiers (by parsing the input C file) and can assign canonical identifiers as needed. Because we are implementing an API, we do not know anything about how many or which identifiers are

[4]For instance, using the standard std++ encoding, the encoding of the type `list a` already has over 150 binary digits. In our proof system and typechecker, we occasionally had to use association lists to avoid encoding AST nodes with heavy nesting, which caused stack overflows in Coq.

needed, forcing us to rely either on impractical functions with nice mathematical properties (as in the core semantics) or efficient stateful generation of identifiers which complicates reasoning.

Thus, to ensure that Foundational Why3 is both compatible with Why3-O and practical, it is necessary for the API to be stateful. But Gallina is a total, purely functional language and thus does not have mutable state. While we can represent stateful computation in Coq using a monad, this adds explicit monadic effects to the API types, breaking compatibility with Why3-O.

6.2 Implementing OCaml APIs in Coq

The standard way to generate OCaml code from Coq is via *extraction*, a process that erases Coq proofs and mechanically converts Gallina to purely functional OCaml code. Typically, one combines this extracted OCaml with hand-written code to handle I/O and other functions impossible or impractical to implement in Coq, then links everything together to produce a single executable. This is the approach of CompCert, the FSCQ verified file system [33], and CertiCoq.

However, most programs are not only executables but additionally provide APIs to allow clients more fine-grained control. This is especially prevalent in the OCaml ecosystem; for instance, formal methods tools like Why3, Frama-C, Alt-Ergo, and Coq all provide external APIs for users to interact with the tool programmatically, e.g. to develop plugins. These programs are themselves built atop more basic libraries providing APIs, e.g. Stdlib, Batteries, Core, and ocamlgraph. We aim to enable verified implementations of such APIs so that a user can implement the desired function in Coq, extract to OCaml, and satisfy the existing API signature; an OCaml client would get the benefits of verification for free. This would permit incremental verification – one could verify parts of the API as needed.

But such APIs cannot in general be implemented in Coq. For example, consider the hd

```
val create_param_decl : lsymbol → decl

let create_param_decl ls =
  if ls.ls_constr <> 0 || ls.ls_proj then
    raise (UnexpectedProjOrConstr ls);
  let news = Sid.singleton ls.ls_name in
  mk_decl (Dparam ls) news
```

Figure 6.3: A Why3 API function not representable in Coq

function in OCaml's List library, which throws an exception if the list is empty:

```
val hd : 'a list → 'a

let hd = function
  [] → failwith "hd"
  | a::_ → a
```

It is impossible to write an axiom-free function with this type in Coq (for good reason: hd (@nil False) is a proof of False). However, OCaml clients may rely on this exception-throwing behavior, so we cannot just change the function to give an option.

Why3 has many such functions in its API. For example, create_param_decl (Figure 6.3), creates an abstract logical symbol (i.e. function or predicate symbol) declaration. It performs some basic well-formedness checks and raises an exception if the checks fail. Then, it constructs the declaration with the hash-consing constructor mk_decl. The type in OCaml is lsymbol → decl; interpreted as a Coq type, such a function must always produce a decl given an input lsymbol. In reality, this function may throw an exception or may produce an lsymbol after modifying some global state.

In this section, we propose a lightweight design principle enabling us to implement such a function in Coq while (1) extracting to OCaml code with the correct types and behavior, (2) remaining executable and axiom-free within Coq, and (3) permitting us to prove the resulting Coq code correct without heavy-duty machinery (e.g. separation logic). The main idea is to *modify extraction to represent features differently in Coq and OCaml.* For each OCaml feature, we give a computable model of this feature in Coq, linking the two with special extraction modification directives. This increases the TCB of the extracted OCaml

```
Record errtype : Type :=
  { errname : string; errargs: Type; errdata : errargs}.
Definition mk_errtype name {A} (x: A) :=
  {|errname := name; errargs := A; errdata := x|}.
Definition Failure (msg: string) : errtype :=
  mk_errtype "Failure" msg.
```

Figure 6.4: Step 1: modelling exceptions in Coq

implementation (§7.4) but results in computable and reasonably idiomatic code on both sides.

The method shown in this section was presented at the CoqPL 2025 workshop [35]. For that purpose, we developed a library demonstrating this design pattern that also includes some smaller and self-contained examples; it is available at `https://github.com/joscoh/coq-ocaml-api`. The underlying ideas are not new, but existing approaches (§6.4) are far more heavyweight and largely fail to satisfy our desired properties, especially computability within Coq.

6.2.1 Error Handling

We first demonstrate the technique for implementing OCaml code with exceptions, using the running example of the List.hd function. We model error handling in Coq using an error monad (included in the coq-ext-lib library). To do this, we first model OCaml exceptions with a record errtype as shown in Figure 6.4. Each errtype instance includes a name and some arguments; we give the Failure exception (i.e. OCaml's failwith) as an example. Note that OCaml's exn type for exceptions is an *extensible variant*, an ADT allowing new constructors to be added at any time. Such types do not exist in Coq; a record allows us to add new exceptions but disallows pattern matching (which we can partially recover by matching on the errtype name).

Next, we define the monadic error interface (Figure 6.5). The error monad is simply a sum type representing either a successful return or an errtype. We define the standard monad

```
Definition errorM A : Type := (errtype + A)%type.
Definition err_ret {A} (x: A) : errorM A := ret x.
Definition err_bnd {A B} (f: A→ errorM B)
  (x: errorM A) : errorM B := bind x f.
Definition throw : ∀ {A} (e: errtype),
  errorM A := fun A e⇒ raise e.
```

Figure 6.5: Step 2: Coq error interface

```
Extract Constant errorM "'a" ⇒ "'a".
Extract Inductive errtype ⇒ exn [""].
Extract Inlined Constant err_ret ⇒ "(fun x→ x)".
Extract Inlined Constant err_bnd ⇒ "(@@)".
Extract Inlined Constant throw ⇒ "raise".
Extract Inlined Constant Failure ⇒ "Failure".
```

Figure 6.6: Step 3: Extracting the error interface to OCaml

functions return and bind and the error-producing monadic throw via coq-ext-lib. The key step is to extract this to the associated interface (i.e. using exceptions). For the OCaml types to be consistent, we must erase the monad when extracting. Therefore, errorM A gets extracted to type A and errtype extracts to exn. Pushing the changes through, we see that the type of err_ret becomes 'a → 'a; this is the identity function (consistent with the erased monad). Likewise, err_bnd has type ('a → 'b) → 'a → 'b when extracted; this is function application. Finally, throw has type exn → 'a; unsurprisingly we use OCaml's raise for this. Individual models of exceptions are extracted to their OCaml counterparts. Figure 6.6 shows the extraction modification commands implementing this transformation.

All the above steps must be done only once (we have done so in our Coq development and self-contained library). Then, we can implement an exception-throwing OCaml program in Coq by writing it in this error monad:

```
Definition hd {A: Type} (l: list A) : errorM A :=
match l with
| nil ⇒ throw (Failure "hd")
| x :: _⇒ err_ret x
end.
```

Extraction then produces the expected function with exactly the same types and effectful behavior as desired:

```
let hd = function
| [] → raise (Failure "hd")
| x::_ → x
```

The Coq function is fully computable and very similar to the function we would have written without extraction. We can prove properties about hd using standard Coq reasoning, without any additional axioms or program logics. Additionally, note that this method works as long as the Coq implementation uses *only* the functions in the interface in Figure 6.5. If one tries to pattern match on an error monad instance, for example, the resulting OCaml code will be ill-typed. We discuss this more in §6.2.4.

6.2.2 Mutable State

As we have seen, Why3-O uses mutable state for generating unique identifiers and for hash-consing (other instances of mutable state can be replaced with equivalent purely functional implementations at less cost; we ignore these), which is critical for performance. We follow the same basic approach as for error handling, this time using a state monad in Coq and extracting to OCaml mutable references. We highlight several key aspects and subtleties.

The Monadic Interface A state monad is a computation taking in an initial state and producing a new state and an output – st S A := S → S * A, where S is the type of the state and A is the return type. The basic operations include bind and return as well as get and set – the former has type st S S and allows one to retrieve the value of the state; the latter has type S → st S unit and updates the state. runState allows one to run a stateful computation on an initial state. Composing state with bind runs the first computation and threads the resulting state into the second one.

We extract this to mutable references very similarly to the error case. The monad is erased, bind and return are identical to the error case, get gives the value of the reference, and set updates the value in the reference. To know which mutable reference to query, we encapsulate the state implementation in a module, providing facilities for clients to construct state (i.e. mutable references) of any non-polymorphic type.

Soundness However, the naive approach above is not sound. The problem is runState, which has Coq type st S A → S → A. After the monads are erased, the extracted runState should have type 'a → 's → 'a; the only (pure) function with this type simply returns the first element. Essentially, this means that the OCaml implementation of runState would ignore the initial state and return the input value. In fact, this makes sense: in the OCaml version, the initial state was already fixed when the mutable reference was first created. Changing the initial state would be like replaying the computation assuming we had instead instantiated the mutable reference differently, an impossible task.

A simple example shows that this is unsound. Suppose we have a mutable counter with an incr function (of Coq type st Int Int). In Coq, it is true that runState incr 0 = runState incr 0 (both are 1), but in the extracted OCaml, that is false – the left-hand side evaluates to 1, while the right-hand side evaluates to 2 since incr has been called twice. A solution is to *fix the initial value*: our generic state module is parameterized by both the state type and the initial value. Then, we provide a runState function of type st S A → A. In Coq, this function runs the stateful computation on the fixed initial value; in OCaml, it returns the first argument and then resets the value of the mutable reference to the initial value. In this way, it is safe to run the computation as many times as desired; but it can only be run (in Coq) on the value the OCaml implementation truly started with.

Composing State and Combining with Errors There are several pieces of mutable state in the API: an integer for the identifier counter and state for hash-consing the types, declarations, theory declarations, and tasks. In general, we design our Coq API with types including the smallest amount of state necessary for each function, even though the eventual transformations need the full state. We provide various lift functions to turn e.g. st S1 A into st (S1 * S2) A and prove numerous identities to combine and reorder state; these have

```
(*In CoqBigInteger.v*)
Definition t : Type := Z.
Definition mul : t → t → t := Z.mul.
(*In Extract.v*)
Extract Inlined Constant CoqBigInt.t ⇒ "BigInt.t".
Extract Inlined Constant CoqBigInt.mul ⇒ "BigInt.mul".
```

Figure 6.7: Implementing big integer operations

no computational effect (and all are erased during extraction) but ensure that the types are consistent. This design is not ideal: it requires significant code overhead and a fixed ordering on the full state, which we decide arbitrarily, but it works reasonably in practice.[5]

Finally, most functions involve both state and error handling, so we combine these two features into a single monad via *monad transformers*. This error-and-state monad represents a computation that takes an initial state and produces a new state along with a possibly-error-producing result. The extraction of the combined error-and-state monad is almost identical to those we have seen, and we provide the appropriate lift operations to embed each individual monad into the combined one.

6.2.3 Other OCaml Features Not Representable In Coq

Mutable state and exceptions are the trickiest components to deal with, but there are several other OCaml features not supported in Coq for which we take a similar approach of modifying extraction.

Integers Why3-O uses both machine-length and arbitrary-length integers. It is possible to model fixed-length integers in Coq as an unbounded mathematical integer (Z) and a proof of the bound (e.g. CompCert's Int library does this); then, we could extract this integer model to OCaml fixed-size integers. We do this in a few cases, but this requires many tedious proof obligations about bounds, which are unnecessary in most cases (e.g. computing the

[5]Haskell-style lenses would alleviate some of these issues (reducing boilerplate in getting and setting the composed states), but not all – they do not permit arbitrary composition of states.

```
(*decidable equality*)
Definition eqb x y := (tag x =? tag y) && ...
Definition eqb_fast x y := eqb x y.
(*In Extract.v*)
Extract Inlined Constant eqb_fast ⇒
  "(fun x y→ x == y || eqb x y)".
```

Figure 6.8: Implementing fast equality

length of a list). Instead, we make small changes to the API in several places and replace ints with OCaml's Zarith library for unbounded integers.[6] Most such integers are small, and thus Zarith is very efficient. Then, we implement an integer interface in Coq with all the required operations we need; we extract these to the appropriate Zarith functions. Figure 6.7 shows an example, where BigInt is the existing wrapper around Zarith in Why3.

Reference Equality Reference equality (==) compares memory addresses; it is much faster than standard (structural) equality (=). Why3-O frequently uses reference equality to compare AST nodes quickly (thanks to hash-consing, this is safe). No Coq function can represent reference equality, as the following example shows:

```
"p" == "p";; (*false in OCaml*)
let x = "p" in x == x ;; (*true in OCaml*)

Lemma let_eq {A B : Type} (foo: A→ A→ B) y:
(let x := y in foo x x) = foo y y. (*equal in Coq*)
Proof. reflexivity. Qed.
```

We cannot abandon reference equality entirely; it is far too slow to traverse each AST node every time. Meanwhile, in Coq, we need decidable structural equality for proofs. To resolve this, we make use of two crucial properties. First, if x == y, then x = y in OCaml.[7] Second, because the hash-consed elements have unique tags by definition, two elements should be equal iff their tags are equal. We implement "fast" versions of equality as in Figure 6.8.

[6]There are not very many such functions we need to change: other than hashing, integers are mainly used for size computations (e.g. the number of term free variables), which are rare.

[7]This property could be axiomatized in Coq as follows: First, we model reference equality as a relation ref_eq : A → A → bool → Prop (a relation allows reference equality to "return" both true and false for equivalent Coq values). The axiom states that if ref_eq x y true, then x = y.

First, we define standard structural equality and prove that it accurately decides Leibniz equality. However, we check tags first before checking any other fields. Then, we define a fast version in Coq identical to decidable equality; extracting this to the function shown.[8] If the two nodes are equal, by the hash-consing property, `x == y` will give `true`; thus, the computation short circuits and does not need traversal. Meanwhile, if the two nodes are not equal, reference equality gives false, but the tags are unequal; thus, `eqb` returns `false` without examining the rest of either structure. This resulted in significant speedup compared with using pure structural equality in the OCaml implementation (see §7.2).

Mixed Record-Inductive Types The last feature for which we use our modified extraction mechanism is mixed record-inductive types, mutually recursive types where at least one type in the mutual block is a record (see Figure 6.2 for an example). Coq does not support such types; unlike the previous features, there is no fundamental reason why this must be the case (indeed, Coq records are really just regular inductive types with additional notation). We have two possible choices for how to implement such types. First, we could just use a mutually recursive type, replacing the record with a single-constructor ADT. This is straightforward to reason about in Coq. However, it is not compatible with the existing OCaml API, and clients of Why3 depend on dot-notation to access record fields.

There is an alternative that allows us to trick Coq into accepting a mixed record-inductive type. Namely, we make one type parametric, and later instantiate it with itself, as Figure 6.9 shows. Careful examination will reveal that, from the point of view of an external client, this representation is equivalent to the `pattern` and `pattern_node` shown in Figure 6.2: the recursive structure is identical and `pattern` is a record with the correct fields.[9] However, there is a downside: Coq cannot generate an induction principle for such a type and cannot tell that recursive functions defined over the structure of this type terminate. It is possible

[8]If we need fast computation within Coq, we can compare tags, as long as we are in a context where all nodes are hash-consed. We later (§6.3.2) discuss global state invariants expressing this.

[9]It appears that a user can create a `pattern_o` instantiated with *different* types, but since the types are marked `private` to external users, this is not possible.

```
Record pattern_o (A: Type) :=
{pat_node: A; pat_vars: Svs.t; pat_ty : ty_c}.

Inductive pattern_node :=
  | Pwild
  | Pvar : vsymbol → pattern_node
  | Papp: lsymbol → list (pattern_o pattern_node) → pattern_node
  | Por: (pattern_o pattern_node) → pattern_c → pattern_node
  | Pas : (pattern_o pattern_node) → vsymbol → pattern_node.

Definition pattern := (pattern_o pattern_node).
```

Figure 6.9: Implementing mixed record-inductive type in Coq

to get around this by defining a size function and performing well-founded induction, but this is tedious and repetitive. Instead, we include both types, using the mutually recursive one for computation and proofs in Coq, while extracting to the mixed record-inductive type. Again, we provide a carefully chosen interface to ensure that the OCaml code typechecks, including constructors and functions to retrieve the components of the Coq mutual ADT. We extract to the corresponding OCaml functions on the mixed record-inductive type.

We have consistently chosen representations for the unsupported OCaml features that make our Coq reasoning and computation simple and reasonably efficient: monads, Z, decidable equality, and mutually recursive types. This aligns with our design goals: our Coq implementation assumes no axioms and is reasonably close to what we would write if we were not extracting to OCaml at all. The OCaml implementation requires more trust, but we provide a relatively small set of primitives for which we modify extraction, giving us confidence that both versions of our API compute the same functions.

6.2.4 Limitations

Though our approach is lightweight and satisfies our design properties, it has several drawbacks. These limitations do not prevent us from implementing our desired subset of Why3-O in Coq, but they shift some additional burden on the user and restrict other possible uses.

Limits to OCaml's Type System While we can extract the concrete monads to their non-monadic counterparts, there are limits to this. Namely, we define new functions per monad (e.g. err_ret and st_ret rather than a single return); similarly, we do not write and extract functions polymorphic over monads. The problem is that OCaml's type system is not nearly as strong as Coq's, lacking higher-kinded types. Therefore, the resulting code would be littered with Obj.magic (unsafe typecast), inserted by Coq's extraction whenever OCaml cannot express a particular type, making the code extremely poor quality and difficult to read. Therefore, we write separate definitions, using Coq's notation scopes to write similar-looking code for each monad without ambiguity. Note that in the end, our functions are evaluating concrete monad instances; therefore, if Coq could perform partial evaluation before extracting to OCaml, this would not be a problem.

Compiling the Extracted Code There is some additional practical work needed to correctly compile the resulting OCaml programs. We use Why3's existing .mli files, which define the external interface and the information visible to clients[10] instead of the extracted ones. There is a dependency problem: the type-safe constructors throw exceptions which themselves have AST arguments defined in the same file. In other words, we cannot declare the exceptions either before or after the extracted code. Instead, we split the API files into 4 parts: (1) defining the AST definitions (in Coq, extracted) (2) defining the exceptions (in OCaml) (3) defining the needed API functions, including type-safe constructors (in Coq) (4) defining the rest of the API, which we need purely for compatibility with existing Why3 OCaml code. We then use dune to concatenate the files together between extraction and OCaml compilation.

Coq and Opacity Why3-O crucially relies on information hiding: external users (even within other parts of the API) can only construct and use AST nodes as prescribed in the

[10]With minor modifications, e.g. for the alternate mixed record-inductive types and arbitrary-length integers.

API. This results in type safety and other global invariants (e.g. hash-consing properties). To represent such reasoning, we would need to make certain definitions opaque. Additionally, as we saw, our method for implementing OCaml APIs relies on a disciplined Coq user – if the user calls functions outside the given interface, the resulting OCaml code will be incorrect and will likely fail to typecheck.[11] Ideally, we would enforce this restriction within Coq itself through opacity.

Unfortunately, opacity is not well supported in Coq. For instance, some types cannot be made opaque or else Coq cannot tell that recursive functions terminate, while other forms of opacity are ignored by Coq simplification.[12] There are only two ways to make something truly opaque in Coq: axiomatize it (and sacrifice computability) or hide it behind a module type. The latter complicates reasoning and violates our design goals. First, every Coq object relying on this opaque definition must then be defined in a module parameterized by the module type. For instance, the Term API would become a module functor taking in a State module type. This completely violates our design goal of compatibility with Why3-O, and OCaml users should not need to reason about any state monad implementation in their code, even an opaque one. Coq modules are also difficult to reason about, as they are not first-class and interact poorly with inductive type definitions. Some, but not all, of these issues would be alleviated if Coq had an .mli-like construct; then one could specify exactly the exposed interface without needing an explicit module functor (indeed, .mli files in OCaml are module types guaranteed to have exactly one implementation; the compiler can infer the module functor argument). But at present, we cannot have computability, compatibility, and information hiding at once in our Coq API.

Limitations on Mutable State Lastly, our mutable state implementation is restrictive – one cannot directly define an array or any piece of state that depends on a more complicated mutable structure (e.g. a tree or graph). Additionally, while this method does

[11]Despite our modified runState implementation, since the St type is not opaque, it is still ultimately a function and could be applied; this violates the soundness of our extraction method.

[12]For example, Coq's compute respects opacity in some cases, while vm_compute ignores it.

support multiple instances of the same type of state, the user must be careful to keep the two separate – in Coq, these could be indistinguishable (e.g. there is no difference between a stateful computation over a single state of type `A * A` and one over two separate states of type `A`, though the OCaml representation is quite different). However, this simpler representation suffices for our purposes: the states we need are integer- and hash-table-valued (our "mutable" hash table is a mutable reference storing an immutable binary trie). It would be possible to extend this implementation to handle more kinds of state; see §6.4.

6.3 Connecting Foundational Why3 to Why3Sem

In this section we describe how we can connect Foundational Why3 and our proved-correct compiler for P-FOLDR. Our basic approach is as follows: first, we define the semantics of Foundational Why3 via translation to P-FOLDR through functions that strip away the extra information and produce a core AST node. Second, we define reasoning principles on stateful, exception-throwing code and define soundness for Foundational Why3 transformations. Next, to facilitate reasoning, we prove a decomposition theorem that allows us to completely separate the proofs of soundness (completed in Chapter 5) and the equivalence between Foundational Why3 and core tasks (recall, a task consists of the context, hypothesis, and goal). This equivalence, a generalized form of α-equivalence, permits purely syntactic and compositional reasoning. Finally, we give an end-to-end proof of the equivalence between stateful and stateless safe substitution and show how this can be used to prove the soundness of `eliminate_let`, the transformation that eliminates let-bindings by substitution.

6.3.1 From Foundational Why3 to P-FOLDR

We define Coq functions `eval_*` (`eval_term`, `eval_ty`, `eval_task`, etc) that remove the extra metadata (unique tags, free variable sets, etc) and produce core nodes, with semantics given by Why3Sem. Most of these are straightforward, with two exceptions.

First, polymorphic function application is handled differently: the Why3-O AST requires a function symbol and a list of terms annotated with their types. The typechecker automatically infers the type substitution (if one exists) for the type parameters of the function. By contrast, in the core AST, we explicitly give the type substitution; to resolve this, we write a version of this inference for core terms and use this in eval_term. Second, some Why3-O AST nodes are implemented differently: while the core syntax distinguishes between terms and formulas, Why3-O does not. Similarly, while core proof tasks distinguish between definitions, hypotheses, and the goal, Why3-O includes all in one data structure (decl). To handle these differences, many of the eval functions return an option; a None result represents some ill-typedness (e.g. if one tries to quantify over a term instead of a formula). The semantics are only defined on terms that evaluate to Some. Eventually, we would like to extend this by giving a similar type system for Foundational Why3 as for P-FOLDR and proving that well-typed AST nodes always evaluate to Some x, where x is a well-typed core AST node. Nevertheless, the eval_task function transforms a Foundational Why3 task into a core task, enabling us to define typing and validity:

```
Definition typed_task (t: task) : Prop :=
  ∃ t', eval_task t = Some t' ∧ core_task_typed t'

Definition valid_task (t: task) : Prop :=
  ∃ t', eval_task t = Some t' ∧ core_task_valid t'.
```

6.3.2 Relating Stateful and Stateless Code

Now we define soundness for Foundational Why3 transformations. While core transformations are functions of type task → list task, Foundational Why3 transformations have type task → errState full_st (list task), where full_st is the global state. This state consists of a counter for identifiers and hash-cons information (counter and hash table) for types, declarations, theory declarations, and tasks. Therefore, our soundness definition will involve the error+state monad and the specific invariants that we need on full_st.

To enable such reasoning, we take inspiration from the State Hoare Monad [104], which

```
Definition st_spec {A B: Type} (Pre: A→ Prop) (s: st A B)
(Post: A→ B→ A→ Prop) : Prop :=
∀ i, Pre i→ Post i (fst (runState s i)) (snd (runState s i)).
```

Figure 6.10: Hoare-style reasoning for the state monad

```
Definition errst_spec {A B: Type} (Pre: A→ Prop)
(s: errState A B) (Post: A→ B→ A→ Prop) : Prop :=
∀ i b, Pre i→ fst (run_errState s i) = inr b→
  Post i b (snd (run_errState s i)).
```

Figure 6.11: Hoare-style reasoning for the error+state monad

augments state monads with (dependently typed) Hoare-style reasoning principles. Our approach is not dependently typed; instead, we separately define the meaning of Hoare triples via a shallow embedding. Figure 6.10 shows the definition; it says that a specification holds if, whenever the precondition holds on the initial state, then the postcondition holds on the initial state, final state, and return value. Note that most Hoare logics do not explicitly reason about the initial state in the postcondition; this generality is useful but makes some of the reasoning principles a bit trickier. For example, the bind rule is similar to the classic Hoare rule for sequential composition but must explicitly quantify over the intermediate state. We prove reasoning principles for bind, return, get, and set, along with weakening and consequence rules and several convenient rules for proofs (e.g. for combining separate specifications). Note that unlike with a deeply embedded program logic, none of this is strictly necessary: we could always unroll all the state monads and definitions and reason using ordinary Coq tactics. However, this provides a more convenient and compositional way to structure proofs.

We lift this idea to the error+state monad in the natural way (Figure 6.11); the same compositionality theorems hold, with additional results for lift operations – if a st A B is lifted to errState A B, we can lift the corresponding st_spec to errst_spec and if an errorM B is lifted to errState A B, in the non-error case, we simply prove the implication for the stateless computation, and in the error case, we can prove any spec. Note that we prove *partial* *correctness* – if the computation succeeds, then the postcondition must hold. This aligns with our goal of proving soundness: an implementation that fails with an exception may be incomplete, but it is not unsound. However, we would eventually like to extend this to total correctness and prove that for well-typed Foundational Why3 inputs, the transformations succeed.

We next define when a given task is consistent with the global state (st_wf). This requires that (1) all variables occurring in the state must have ID smaller than the current counter and (2) all hash-consed AST nodes have tags smaller than the corresponding counter and are in the corresponding hash table.[13]

We can now define soundness for transformations in the error+state monad: if the input task is well-typed and consistent with the initial state, then if the transformation executes successfully and the output task is valid, so was the input (there is a corresponding version for lists):

```
Definition trans_errst_sound (trans: trans_errst task) : Prop :=
∀ (t: task), errst_spec
(fun s ⇒ st_wf t s ∧ typed_task t)
  (trans t)
(fun _ (r: task) _ ⇒ valid_task r → valid_task t).
```

In principle, we could now use this definition to prove transformations sound. We must reason both about hash-consing and the generation of unique names (for variables and symbols). Dealing with the former is not hard since eval removes the tag information. However, unique names pose a larger challenge – trans_errst_sound is very restrictive, as it requires reasoning about the exact results of eval_task. Instead, we would like to relate the stateful implementation to the corresponding transformation in our compiler - the two are equivalent, but only modulo variable and symbol names. Moreover, we would like to separate the proof relating the stateful and stateless versions from the soundness proof, carried out entirely in Why3Sem. Thus, we need an appropriate abstraction that lets us reason about equivalence between Foundational Why3 and core tasks that is (1) purely syntactic, (2) compositional,

[13]We will eventually additionally require that the hash tables contain unique AST nodes, but our proofs do not yet need this.

and (3) invariant to changes in variable names. We now provide such an abstraction: a generalized notion of α-equivalence.

6.3.3 Generalized α-Equivalence

α-Equivalence of Terms, Formulas, and Patterns Up to now, we have largely omitted discussion of α-equivalence, which has been crucial in several places throughout Why3Sem and the compiler, e.g. to implement safe substitution, to prove the special syntactic form for inductive predicates (§3.2.5), and to prove compile_match sound. We include both the definition of α-equivalence as well as two functions for α-conversion: one that makes all bound variable names unique (and which we show corresponds to a stateful version in §6.3.4) and a faster, more readable renaming scheme for safe substitution in the proof system. The definition of α-equivalence for terms and formulas is largely standard. We first define α-related terms and formulas under a given pair of finite maps m_1 and m_2 storing the mapping of bound variables between terms. Formally, variables x and y are related iff $m_1(x) = y$ and $m_2(y) = x$ (the bindings are consistent) or x is not in m_1, y is not in m_2, and $x = y$ (for free variables). Two terms or formulas are α-equivalent if they are α-related under empty maps, and bindings add the corresponding pairs to the maps.

The difficult case concerns patterns and pattern matching. Checking α-equivalence of patterns amounts to constructing maps $r_1 : fv(p_1) \to fv(p_2)$ and $r_2 : fv(p_2) \to fv(p_1)$ such that the two patterns have the same "shape" (e.g. equivalence ignoring variables) and the variables map correspondingly. An explicit construction is necessary to use these maps to recursively check the α-equivalence of each branch in a pattern match. While these maps have nice properties (e.g. bijectivity), the algorithm to construct them is stateful and tricky to reason about.

Showing that α-equivalent patterns are well-typed is nontrivial. Recall that well-typed patterns have a number of additional restrictions on their variables (§2.2) – disjunctions have the same free variables, constructor pattern arguments all have disjoint variables, and in p as x, x does not occur in p. Therefore, we must show that the constructed maps preserve variable equality and inequality. Additionally, the Why3-O implementation assumes that the input patterns are well-typed (justified by the API design), while we define α-equivalence over potentially ill-typed patterns (for instance, we want to know that α-equivalence is an equivalence relation regardless of typing). Our implementation is slightly different from that of Why3-O and assumes nothing about typing; we prove the two versions equivalent under typing assumptions. With all this, we can prove desired properties of α-equivalence:

Theorem 6.3.1. *Let t_1 and t_2 be α-equivalent terms (formulas are similar). Then the following hold:*

- *If $\Gamma \vdash t_1 : \tau$, then $\Gamma \vdash t_2 : \tau$ (a_equiv_t_type).*
- *If both terms are well-typed, then for all interpretations and valuations, $[\![t_1]\!]_v^t = [\![t_2]\!]_v^t$ (a_equiv_t_rep).*
- *The free type and term variables of t_1 and t_2 are equal (a_equiv_t_type_vars and a_equiv_t_fv).*
- *t_1 and t_2 have the same shape (i.e. structure ignoring variables) (alpha_shape_t).*

In all cases, these arise as corollaries of more general results about α-related terms and formulas.

Generalizing α-Equivalence to Tasks We lift α-equivalence to larger structures (definitions, contexts, and tasks) in the natural way: each corresponding term or formula is α-equivalent and in general, non-term/formula data is equal.[14] But this is not sufficient for recursive functions and predicates. Recall that recursive functions consist of a name f, a list of variable arguments vs, and a body b. We cannot simply require the that bodies are α-equivalent, as the arguments may differ. Instead, for definitions (f_1, vs_1, b_1) and (f_2, vs_2, b_2), b_1 and b_2 must be α-related under the maps $vs_1 \to vs_2$ and $vs_2 \to vs_1$. We also need the condition that vs_1 and vs_2 either both have unique names or neither does.[15]

[14]In the future, this would be extended with a map between the symbol names; unlike with α-equivalence, this is constant, bijective, and straightforward to reason about.

[15]For well-typed contexts, they both do; the both-or-neither condition allows us to prove that if one is well-typed, so is the other. We cannot just require both or else this relation is not reflexive.

For the desired decomposition theorem, we want to prove that generalized α-equivalence preserves typing and validity of tasks. Proving semantic equivalence is tedious but not too difficult, and the crucial part is to prove that the spec for recursive functions (§3.2.4) still holds. This ultimately follows from results about the semantics of α-related terms (more general than the property in Theorem 6.3.1) and the fact that all free variables correspond appropriately in the α-maps. Typing is much more difficult: we must show that the termination check (§2.2.2) is preserved by this α-relation. Recall that the termination check keeps track of sets of variables s and h. With α-equivalence, we now need detailed assumptions about how the variables in the maps m_1 and m_2 relate to s and h. Specifically, we prove the following:

Lemma 6.3.1 (a_equiv_decrease_fun)**.** *Let t_1 and t_2 be terms α-related by maps m_1 and m_2. Let s_1 and s_2 be sets of variables, and let h_1 and h_2 be empty or singleton sets of variables (i.e. options). Suppose the following conditions hold:*

1. *If $m_1(x) = y$ and $m_2(y) = x$, then if $x \in s_1$, then $y \in s_2$.*
2. *If $m_1(x) = y$ and $m_2(y) = x$, then $x \in h_1$ iff $y \in h_2$.*
3. *All variables in s_1 and h_1 occur in m_1.*

Then if $(s_1, h_1) \Downarrow t_1$, then $(s_2, h_2) \Downarrow t_2$.

The proof is nontrivial; it is *not* the case that each termination case is exactly equivalent, since it is possible that one pattern match occurs on a smaller variable (i.e. case DEC_MATCH_VAR) and the other does not (DEC_MATCH_REC). However, we relate distinct cases through a weakening lemma allowing us to add variables to the smaller set without changing termination: if $s_1 \subseteq s_2$ and $h_1 \subseteq h_2$, then $(s_1, h_1) \Downarrow t \implies (s_2, h_2) \Downarrow t$.

A Decomposition Theorem With all this, we prove that α-equivalence on tasks preserves typing and semantics. Then, we define relations on Foundational Why3 tasks and core tasks (and terms, types, etc) generalizing the equality from the eval_* functions:

```
Definition task_related (t1: task) (t2: core_task) : Prop :=
∃ t, eval_task t1 = Some t ∧ a_equiv_task t t2.
```

Finally, we provide a decomposition theorem allowing us to prove a Foundational Why3 transformation tr_1 sound in 2 steps: (1) Give a core transformation tr_2 and prove it sound according to Why3Sem (2) Prove that under a well-formed state, tr_1 and tr_2 take related inputs to related outputs:

```
Theorem prove_trans_errst_decompose (tr1: trans_errst task)
(tr2: Task.task → Task.task):
sound_trans tr2 →
(∀ tsk1 tsk2, errst_spec
  (fun s ⇒ st_wf tsk1 s ∧ task_related tsk1 tsk2)
    (tr1 tsk1)
  (fun _ r _ ⇒ task_related r (tr2 tsk2))) →
trans_errst_sound tr1.
```

This enables a clean split between the semantic reasoning (done entirely in Why3Sem) and the syntactic reasoning between the stateful and stateless transformations. Note that this kind of decomposition is very similar to the functional model-implementation separation common in large-scale verification efforts. One typically wants to reason separately about the state-manipulating imperative program and a purely functional, stateless model with nicer mathematical properties. The connection between the two is often handled by the verification framework. For example, in VST, one specifies the result of a memory update as a Coq function on the bytes stored in that location. In our case, state is shallowly embedded in a proof assistant, but the above decomposition theorem allows us to perform a similar separation of concerns for this particular setting (unique name generation).

6.3.4 Case Study: Substitution and eliminate_let

Here, we discuss the partial proof of soundness of the eliminate_let transformation, which eliminates let-bindings via substitution. The version of the transformation we consider[16] has

[16]Note that this is not exactly the one used by Why3, which is not structurally recursive and has a nontrivial termination argument. We do implement and prove sound the Why3 version as a core transformation [36] but do not implement a stateful version or discuss further here.

only one interesting case; the others just apply the function recursively:

$$\mathsf{eliminate\_let}(\mathbf{let}\ x := t_1\ \mathbf{in}\ t_2) = (\mathsf{eliminate\_let}\ t_2)[(\mathsf{eliminate\_let}\ t_1)/x]$$

Thus, the key step is to relate stateful and stateless substitution. We prove this relation for a subset of terms (where the only binding is let-binding) and show how we could use this to verify the soundness of eliminate_let.

Defining eliminate_let with Term Traversal In Why3-O, the non-let cases of eliminate_let are defined using a generic t_map function that maps over terms. t_map is not explicitly recursive, but its function argument is instantiated with eliminate_let recursively. However, Coq cannot tell that such a scheme terminates. Instead, we give a generic recursive traversal function over terms (which we call term_traverse). We need this traversal to be binding-safe, so to open binders we use the API function t_open_bound, which (statefully) produces a fresh variable and substitutes this for the bound variable.[17]

Defining term_traverse is quite tricky. It is *not* structurally recursive due to binders. Given formula $\forall x, f$, the function t_open_bound opens the binder, incrementing the counter and resulting in fresh variable y and formula $f[y/x]$, which is not a subterm of the original one. To show termination, we define a standard size function, but even this size function is tricky to reason about thanks to the mutually recursive model of mixed record-inductive types for terms (§6.2.3) – the size function is defined over terms, but the recursion occurs on term-nodes. We prove an alternate induction principle for terms allowing us to ignore non-node information and bypass mutual recursion, using this to prove that variable substitution preserves the size.

However, this is still not sufficient because the function body of the state monad bind does not have any information about the state it is called on. A recursive call within the body

[17]There are also similar functions t_open_branch for pattern binders and t_open_quant for quantifiers, both of which generate and substitute multiple fresh variables at once. We focus on t_open_bound for simplicity.

of bind knows *nothing* about the term and its size and therefore cannot prove termination. To handle this, we define a dependently typed version of bind for the error+state monad that remembers the state it was called on, and we prove the appropriate Hoare State Monad reasoning principle. Putting this all together to define term_traverse is still tricky: we cannot use Equations as it does not support mutual well-founded recursion (and has other problems with monadic binders), so we use a manual technique known to produce good quality OCaml code [75]. Finally, we prove an induction principle for term_traverse, simplifying reasoning about arbitrary traversals.

We complete the definition of eliminate_let by implementing a version of stateful substitution: first, we do a trivial term_traverse to make all binders unique (we call this t_mk_wf), then we perform a regular (stateless) substitution – this ensures that no variables can be captured. Note that this is different from Why3's implementation, which uses a deferred substitution mechanism (see §7.1).

Invariants The correctness theorems for substitution and eliminate_let rely on the state invariants – this lets us show that the newly generated variables are truly fresh. Unfortunately this induces many tedious proof obligations, as we must show that every stateful function produces AST nodes consistent with the state and preserves the consistency of existing nodes. We prove that any function that only increases the global counter and hash table counters and that only grows the hash tables[18] cannot make nodes inconsistent. Then we show that this monotonicity property holds of substitution and eliminate_let and separately prove their outputs consistent.

Though we do not give a full type system, we define an invariant types_wf encoding some well-formedness conditions on types (e.g. that the branches of an if-expression have the same type and that the claimed free variables in a binding are correct). This is needed for the substitution proof of correctness, which relies on type information and free variable sets.

[18]In Why3, hash-consing is implemented using *weak hash tables*, which can shrink if an element is garbage-collected. The above monotonicity property does not hold in the presence of garbage collection, and our implementation uses non-weak data structures.

Specifications for Substitution We prove that each component of substitution is related to its stateless version, assuming all inputs are consistent with the given state. First, we can show that, given term t that evaluates to core term e, opening the binder (v, t) results in a fresh variable v_2 and a term t_2 that evaluates to $e[v_2/v]$:

```
Theorem t_open_bound_res_tm v b t1 e1 (Hwf: types_wf t1):
  eval_term t1 = Some e1 →
  errst_spec (fun _ ⇒ True)
    (errst_tup1 (errst_lift1 (t_open_bound (v, b, t1))))
  (fun _ (v2, t2) _ ⇒ eval_term t2 =
    Some (sub_var_t (eval_vsym v) (eval_vsym v2) e1)).
```

We separately prove that v_2 is equal to v except that its ID is the old state s_1's counter; this ensures that v_2 is fresh as long as t_1 and v are consistent with s_1 (e.g. all IDs were smaller than s_1).

Next, we prove that substitution is related to stateless substitution. This is the core of the correctness argument, and the abstraction of α-equivalence lets us relate these two versions of substitution syntactically. The crucial part of the proof concerns t_make_wf; we prove that the output evaluates to the α-conversion of the input, with the new variable names given by the combination of the old names and the new states (which can be determined functionally). Then, we use this to show that if t_1 is related to core term e_1 and t_2 is related to e_2, then the result of stateful substitution is related to the (stateless) safe substitution on core terms $e_2[e_1/x]$.

```
Theorem t_subst_single_tm_spec v t1 t2 e1 e2
(Hlet: only_let t2) (Hwf: types_wf t2):
term_related t1 e1 →
term_related t2 e2 →
errst_spec
(fun s1 ⇒ term_st_wf t1 s1 ∧ term_st_wf t2 s1 ∧ vsym_st_wf v s1)
  (t_subst_single v t1 t2)
(fun _ t3 _ ⇒ term_related t3 (safe_sub_t e1 (eval_vsym v) e2)).
```

Note that the hypothesis and postcondition involve *relations* (term_related) rather than direct evaluation (eval_term) like in t_open_bound and t_make_wf. Here, we only know that the resulting term is α-equivalent to the one we want, since the bound variables may change arbitrarily (the function is deterministic, but this specification allows for different α-conversion mechanisms or traversal orders). Since we only assume the inputs are related, we must show that no matter what α-equivalent terms we choose, after running the desired function (in this case substitution), the result is equivalent (we show the case for α-equivalence, the general α-related case is similar):

Lemma 6.3.2 (alpha_equiv_t_sub_not_bnd)**.** *Suppose t_1 and t_2 are α-equivalent, t_3 and t_4 are α-related under the map $x \leftrightarrow y$, and neither x nor any variable free in t_1 is bound in t_3 (and likewise for y, t_2, and t_4). Then $t_3[t_1/x]$ and $t_4[t_2/y]$ are α-equivalent.*

Proving eliminate_let Sound We do not carry out the full eliminate_let proof in Coq, but it follows from the correctness of substitution. We show the main proof steps. First, we can prove the term-rewrite rule equivalent to the stateless one via the induction principle on term_traverse:

```
Theorem elim_let_rewrite_related f1 g1
(Hlet: only_let f1) (Hwf: types_wf f1)
eval_fmla f1 = Some g1 →
errst_spec (term_st_wf f1)
  (elim_let_rewrite f1)
(fun _ f2 _ ⇒ fmla_related f2 (elim_let_f g1))
```

The interesting cases are handled by the substitution-related results.

Then, we can lift the result to the full task transformation. Since the above theorem has an eval- precondition, we prove a similar result as Lemma 6.3.2 to show that α-equivalent inputs result in α-equivalent outputs to elim_let_f. Finally, we can use the decomposition theorem (§6.3.3) to combine this with the soundness proof of the stateless eliminate_let for overall soundness.

6.4 Related Work

There is a very long line of work about the broad problem of connecting functional programs written in proof assistants to efficient imperative implementations. Ynot [84] is a framework for Coq that supports writing, reasoning about, and extracting imperative programs in Coq

– it is based on axiomatizing the external effects (e.g. mutable state, nontermination, and exceptions). Ynot is much more powerful than our method but is more heavyweight and requires axioms; the resulting programs are not directly computable within Coq. Itrees [113] are a coinductive data structure used to represent denotational semantics for imperative and impure languages; they can be extracted to executable implementations. However, since they are coinductive and may represent infinite computation, Itrees also cannot be computed in Coq.

Other efforts have focused on connecting Coq with external imperative code. VeriFFI [63] is a verified foreign function interface between Coq and C; it allows one to use C functions in Coq and Coq functions in C, with the meaning of C functions given by VST function specs (which the user must prove satisfied). It provides the example of mutable arrays using a monadic interface as the Coq functional model, but without complete proofs. VeriFFI does not suffer from the opacity limitations we discussed; the external functions are axiomatized with appropriate isomorphisms and rewrite rules (this prohibits computation in Coq). Formally Verified Defensive Programming (FVDP) [28] is a paradigm for allowing Coq programs to call OCaml programs as oracles (when extracted), using a nontermination monad to encapsulate the (possibly non-pure) result. It includes an embedding of OCaml pointer equality, implements a hash-consing example, and decomposes proofs into stateful and non-stateful reasoning. Combining this approach with ours could permit Coq implementation of APIs that depend on both Coq and OCaml code (though the resulting API functions would not be executable within Coq directly).

Similar ideas are also used in the Isabelle Refinement Framework [67], which automatically refines algorithms to efficient implementations (e.g. a map to a red-black tree). The user gives and proves a *refinement relation* between the two representations. This has been extended to imperative refinement [68, 69]. Here, the user writes a monadic function (in the nontermination monad) and uses separation logic defined over a shallow embedding of heap-manipulating programs; automation including a VC generator assists the process.

Perhaps the most directly similar ideas – implementing monadic functions and extracting to imperative code – were implemented by Abrahamsson et al. in CakeML [2] and by Sakaguchi in Coq [97]. Abrahamsson et al. generate imperative CakeML programs from monadic HOL programs using the error-and-state monad and produce a certificate that validates that the translation is sound. They combine the various pieces of state into a single record. Notably, because they implement whole programs and not APIs, they set the state appropriately at the beginning of each computation, avoiding the soundness issue we encountered (our solution, resetting to the initial state, is a generalization of this approach). Their approach is proved correct, and similar ideas would be useful in allowing us to verify the soundness of our modified extraction, assuming a suitable program logic for OCaml. Sakaguchi implements array-manipulating programs in Coq, using a state monad specialized to arrays and modifying extraction in a very similar way as we do, dealing with a similar soundness issue by copying arrays when needed. The approach is limited to arrays (to avoid aliasing analysis) and produces OCaml code with lots of `Obj.magic`, but similar ideas could be useful to extend our method to handle arrays.

There is additionally significant prior work on hash-consing. As mentioned, FVDP includes hash-consing as an example. Why3's hash-consing implementation is based on the technique of Filliâtre and Conchon [49]. Braibant et al. [29] study the implementation of hash-consing in Coq, providing 3 different implementations. First, they implement everything inside Coq – they use a state monad and define similar invariants on the global state, though they reason in pure Coq rather than with a Hoare State Monad. They next give an implementation that avoids hash-consing reasoning in Coq but extracts to Filliâtre and Conchon's OCaml hash-consing library, extracting Coq decidable equality to OCaml physical equality. They lastly give a completely axiomatized version. Our approach is a combination of the first two: we allow full computability and reasoning in Coq, extract to more efficient (but not the most efficient) OCaml code, and take care to avoid unsound physical equality extraction.

Chapter 7

Discussion

In this chapter, we consider several practical and theoretical properties of our verified IVL and its relation to Why3. In particular, we discuss the differences between Why3 and our semantics, compiler, and API implementation, run Foundational Why3 on the Why3 and EasyCrypt test suites, list the bugs and issues we discovered in Why3 as a result of this work, estimate the TCB of our development, mention possible remaining soundness gaps, and discuss which parts of our semantics could be reused in other contexts. Finally, we explore extensions and future work.

7.1 Comparing Why3Sem to Why3

Why3Sem aims to capture core Why3, which roughly aligns with the pen-and-paper semantics [48]. 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 ϵ 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 mentioned in §3.2.1, we include only a subset of allowed ADTs. In particular, we do not support nested or non-uniform ADTs.
- Our recursive functions use a different termination metric (simple structural recursion rather than lexicographic ordering) and we only support uniform functions and 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, Why3Sem does capture a large subset 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 of 13 ADTs in the library are supported by Why3Sem (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 integer termination measures. Thus, our formalization captures a very practical subset of Why3, and some excluded features like non-uniform types and coinduction are rarely used in practice (§7.2).

Our compiler transformations aim to represent their Why3 counterparts faithfully. There are a few minor differences; for instance, our eliminate_algebraic transformation requires all types in a mutual block to be axiomatized or kept. The largest difference is in rewriteF for eliminating pattern matches– Why3-O includes an optimization that generates let-bindings rather than implications in some cases (e.g., length (Cons h t) = 1 + length t rather than x = Cons h t $\rightarrow$ length x = 1 + length t). This is surprisingly subtle; these are *not* semantically equivalent under any valuation, but rather one must generalize the variable valuation appropriately.[1] In any case, we found that this optimization has no performance impact on the comprehensive Why3 test suite and thus did not implement it.

Finally, Foundational Why3 aims to be compatible with Why3-O, and we attempt to mimic the existing functionality as closely as possible. The differences are:

- We use big integers rather than machine-length ones.
- We use binary tries to implement sets, maps, and hash tables rather than binary search trees and ordinary hash tables.
- Our variable substitution mechanism differs substantially. Why3-O uses *deferred* (i.e. lazy) substitution when substituting under binders, storing a set of future substitutions and only executing when necessary. Our eager substitution mechanism is easier to define (it is difficult to store the deferred substitution set without violating Coq's strict positivity condition) and reason about, but it does not scale well to very large terms or tasks.
- Because of this, our α-equivalence implementation also differs from the significantly more complex Why3-O version, which must store lists of maps to reason about the deferred substitutions.

Note that only the first two changes are visible to external clients.

[1]Specifically, let-bindings are only generated if the variable matched on occurs most recently under a $\forall$ binder - this means that with a suitable generalization of the valuation, we can instantiate it with any value of the correct type to prove correctness.

7.2 Testing Foundational Why3

To demonstrate that Foundational Why3 is (mostly) compatible with Why3-O, practical to run, and encompasses a usable subset of Why3, we run two comprehensive test suites: the Why3 test suite and all EasyCrypt tests and examples. All tests were performed on a commercial laptop (2019 Thinkpad X1 Carbon with an 8-core Intel i7-8565U 1.80GHz CPU).

The Why3 test suite contains a wide variety of tests: for typechecking both well-typed and ill-typed objects, for checking valid and invalid goals, for replaying proofs of WhyML program verification, and for extracting code. We cannot quite run all the tests: we modify one standard library file (out of 45) as well as one typing example (out of 150) because they both use nested recursion (rose trees). We also omit five of the larger WhyML examples (out of 240), as they use features like nonuniform recursion, nested recursion, and more sophisticated termination. One of these examples is especially large and becomes prohibitively slow due to our eager substitution mechanism. Nevertheless, we can run nearly all the Why3 tests, with only a small slowdown. In total, it takes 8 mins 30 sec on average to run the slightly modified test suite against Foundational Why3 compared to an average of 5:56 when running against the Why3 master branch.[2] This slowdown is largely due to our less efficient data structures (especially for hash tables and hash-consing), eager substitution, and our use of big integers. We do not expect to equal the performance of hand-optimized OCaml, and this provides some initial evidence that our implementation is still quite practical.

While the Why3 test suite aims to capture a variety of corner cases (e.g. non-uniform recursion) and is useful to determine which features in Why3 we can and cannot handle, it is not necessarily realistic and representative of types of goals one might encounter in practice. Thus, we evaluate Foundational Why3 on the EasyCrypt suite of examples and tests. This is a particularly valuable set of tests for several reasons. First, EasyCrypt is built atop the Why3 logic API directly – this means that we are not limited to the types of verification conditions produced by WhyML. Second, it uses a very large part of the logic API, including

[2] As of February 21, 2025.

Test Suite	Tests	Avg Slowdown	25th %ile	75th %ile	Max
Unit Tests	23	1.6x	1.4x	1.8x	2x
Theories	120	1.8x	1.6x	2x	2.9x
Examples	40	1.7x	1.5x	1.8x	3.3x
Total	183	1.8x	1.6x	1.9x	3.3x

Figure 7.1: Result of running Foundational Why3 on EasyCrypt test suites

ADTs, pattern matching (including simultaneous matching), recursive functions, and more. Finally, EasyCrypt is a proof assistant for cryptography, implementing an ssreflect-like tactic language with an smt tactic that calls Why3. Thus, it provides strong evidence that Why3 (and by extension, Foundational Why3) is useful for higher-level functional-model reasoning, one of our goals.

We show the results of running Foundational Why3 on EasyCrypt's test suites in Figure 7.1. In total, we pass all 183 tests using a solver timeout of 30 seconds. This is highly encouraging and suggests that the subset of Why3 we have considered is large enough to handle the kinds of goals that arise in practice when proving properties about functional models.[3]

In total (running the tests in parallel on 8 cores), the test suite takes an average of 8:18 using the master branch and 14:55 using Foundational Why3. We show the breakdown of the tests by type, comprising simple unit tests, the EasyCrypt theories and standard library – which includes generic results about data structures, linear algebra, rings, polynomials, and more, as well as basic cryptographic and security primitives like MACs, pseudorandomness, public key encryption, and Diffie-Hellman key exchange – and more involved examples building on the basic primitives. In general, as the table shows, there is a 1.5-2x slowdown that is quite consistent between the tests. Note that the slowdown is larger than that of the Why3 test suite. We hypothesize that this results from EasyCrypt's many small calls to smt (and hence to Why3); thus the overhead of initializing Why3 and constructing and

[3]Note that EasyCrypt does use function types, which we do not (yet) have semantics for but Foundational Why3 does allow and includes the appropriate typechecks. However, EasyCrypt does not use any features prohibited by our type system, e.g. non-uniform types or fancier termination conditions.

transforming goals becomes more significant relative to the time spent by the solver. Indeed, there are 1226 calls to `smt` in the EasyCrypt theories and stdlib suite and another 684 calls in the examples. Nevertheless, despite the slowdown, Foundational Why3 is indeed practical on real-world examples.

Finally, we test to determine if our optimizations had a real impact on performance. While we have discussed the need for generating unique identifiers quickly, it is less clear that hash-consing and fast equality calculation is necessary. Therefore, we attempted to run the Why3 test suite without fast equality – i.e. simply using structural equality everywhere. Some tests timed out after about 1.5 hours; the "check valid goals" section of the benchmark suite – comprising only 15 tests – takes under 7 seconds with fast equality and over 18 minutes without (over 150x slowdown). We conclude that hash-consing and fast equality computations are absolutely critical for performance and are well worth the additional proof overhead.

7.3 Bugs and Issues Found in Why3

Throughout our work, we found several bugs in Why3. The first could lead the tool to crash due to a non-typechecking term:

- The pattern matching exhaustiveness robustness bug we discussed in §4.2.3 causes a transformation that should always succeed (rewriting when the term in question appears in the target) to crash with a fatal exception. Note that without the repeated (and otherwise unnecessary) typecheck, this would enable one to construct non-exhaustive pattern matches. We fixed this bug in our implementation by replacing the exhaustiveness check with a provably robust and stronger condition.

The second is a soundness bug but is likely not triggerable in practice:

- The mutable counter that assigns unique tags uses an OCaml `int`, which could overflow (though this is extremely unlikely[4]), violating the invariant that all tags are globally unique. While this would not be a soundness issue for hash-consed AST nodes (though it would be very confusing for the user and make the semantics unclear), it is a bigger problem for α-conversion, as substitution is not capture-avoiding in all cases. This can lead to unsoundness: $\forall x_1 \forall x_2, x_1 = x_2$ is false, but after α-conversion, if the counter wraps around, it can become $\forall x_1 \forall x_1, x_1 = x_1$, which is true. In Foundational Why3, we use a big integer to avoid this problem.

Finally, there are several cases where Why3's Coq back-end outputs ill-typed code. These violate the principle that after the appropriate transformations have been run, the output lies in the subset of Why3 supported by a particular solver.

- There are several such bugs related to termination. First, if one uses lexicographic termination and outputs to Coq, the resulting Coq Fixpoint fails the termination check.
- Next, Why3 allows termination on non-strict subterms (e.g. the size example in §2.2.2); this is similarly translated to a syntactically identical Fixpoint that fails to pass Coq's checker.
- Additionally, functions that rely on Why3's "lazy" termination analysis (e.g. a and b in §2.2.2) fail when given to Coq.
- Finally, Why3 allows non-uniform type parameters in mutually recursive types; once again, these are straightforwardly translated to the corresponding Inductive in Coq, which fails to typecheck, as Coq does not allow such non-uniformity.

These examples demonstrate that even if the IVL itself is sound, it is still very difficult to connect verification tools at different layers. Why3 is less powerful than Coq (with classical axioms), but this does not mean that it is a strict syntactic subset. Our work (and MetaCoq) could enable a translation to Coq that is provably sound and well-typed, but this relies on a formal semantics to avoid such edge cases and gaps. Though we did not examine the output

[4]On 64-bit machines, int is a 63-bit integer; if one generated 10 billion tags per second, it would take 30 years to overflow. However, int is only 31 bits on a 32-bit machine; this could easily overflow.

Component	LOC
Common (Sets, Maps, HLists, etc)	7K
Why3Sem	
Syntax (+Variables, Types, Contexts)	3K
Typing (+Typechecker)	6K
Basic Semantics (+Equivalence Lemma, Interpretations)	4K
ADT Encoding	2K
Recursive Function Encoding	6K
Inductive Predicate Encoding	2K
Substitution (Type+Term) and α-Equivalence	13K
Logic (+Tasks, Transformations, Full Interps)	2K
Total Why3Sem	38K
Proof System	8K
Compiler	
eliminate_definition	2.5K
compile and compile_match	11K
eliminate_algebraic	11K
eliminate_inductive	3K
Total Compiler	27.5K
Foundational Why3	
coq-util (Monads, State, etc)	1K
util (Set, Map, hash-consing, etc)	2K
core (Type, Term, Decl, Task, Trans + term_traverse)	6K
transform	1K
proofs (generic, substitution, eliminate_let)	11K
Total Foundational Why3	21K

Figure 7.2: Lines of Coq code for each component

from Why3 to other solvers, it is likely that similar subtle differences have crept in – formal proof is critical for ensuring compatibility.

7.4 Quantitative Metrics and TCB

Figure 7.2 gives the approximate number of lines of Coq code for each component in our development. We note that some of the values are a bit inflated; we generally did not optimize for size of proofs and definitions (for instance, we give a separate interface for std++ sets and maps rather than using them directly), our automation could be improved, and some proofs are repetitive (e.g. if the term and formula cases are similar). However, this gives a rough idea of the proof/formalization effort involved in each component.

While the Coq development is large, only small parts must be trusted. The main trusted component is our formalization of the P-FOLDR semantics: while we validate the semantics by showing that it is useful, the user must ultimately trust that it faithfully represents the meaning of the various components in the logic. To that end, we include the following in the TCB: the definitions of interpretations (pi_dom and pi_funpred), the definitions of valuations (val_typevar and val_vars), the basic semantics (get_arg_list, match_val_single, match_rep, term_rep, and formula_rep), the properties of ADTs and full interpretations, and the semantics of tasks and transformations (task_valid and valid_trans).[5] In total, these components comprise approximately 440 lines of Coq definitions, which are specifically designed to be intuitive and stay close to the pen-and-paper definitions. Note that these also rely on more basic data structures: lists, maps, and heterogeneous lists; each such definition is only a few lines and quite standard.

Foundational Why3 adds additional sources of trust (even after the soundness proofs are completed): the eval functions, the definition of the State Hoare Monad, our state invariant, the definition of task validity, and the definition of (stateful) transformation soundness: together these comprise about 500 lines of Coq definitions (about 150 of which are simple functions extracting the types, declarations, and identifiers from a task for the state invariant). Note that if one wanted to verify a single task's construction and compilation end-to-end, the state invariant would fall out of the TCB – one would prove that the API constructors establish the invariant and that the transformations preserve it. The OCaml implementation requires trusting the modified extraction directives (about 300 lines of code, of which 80 are integer constants) the Coq extraction pipeline, the dune postprocessing, and the OCaml compiler. Of course there are other sources of trust intrinsic to our approach: the Coq kernel, the consistency of CIC, the OCaml compiler used to compile Coq, the hardware, etc. Nevertheless, it is clear that the TCB is significantly smaller than that of existing auto-

[5]Note that we do *not* need to trust the encodings of ADTs, recursive functions, and inductive predicates – we prove that they satisfy the (trusted) specification.

mated tools. For instance, the specific files in Why3-O we implement in Foundational Why3 comprise about 7K LOC (which also rely on libraries, some of which we replaced with Coq versions); the broad parts of Why3 we are interested in (utilities, logic API, and transformations) comprise about 29K LOC. Moreover, the nature of this trust is different: Foundational Why3 requires one to trust semantic definitions designed to be interpretable and uncontroversial. Why3, as with any unverified tool, requires one to trust complex implementations without formal specifications.

We have proved soundness only for a core subset of Why3, and we identify 3 likely places for possible soundness gaps:

1. As we have seen, our termination checker differs significantly from Why3's. Termination checking is tricky to implement correctly; Coq for instance has found several critical bugs in the guard (termination) checker.[6] Why3's checker uses a lexicographic ordering on structural inclusion, which would not be too difficult to include in our semantics, and the "lazy" method that checks only the paths called on the control flow graph, which would be significantly more challenging to verify.
2. We do not model the connection between logical and WhyML code – one can define logic functions by writing pure WhyML functions with well-founded measures, a possible source of unsoundness for similar reasons as above. These termination conditions are checked at verification time; we would need to reason about the semantics and translation to SMT simultaneously.
3. A final source of unsoundness concerns the last step, where Why3 prints the transformed task into an SMT-LIB2 file. As we saw with the Coq back-end, it is easy to accidentally ignore subtle mismatches between the two layers, and verifying the final translation to SMT would be critical.

[6]`https://github.com/coq/coq/blob/master/dev/doc/critical-bugs.md`

7.5 Reusability and Axioms

Here, we discuss the potential utility of Why3Sem for modelling 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 (it depends on formula_rep, but this could be generalized); the proofs about the least predicate and inversion properties would hold. Our recursive function encoding, on the other hand, is unlikely to be as useful for other projects, as it is tightly wedded to details of pattern matching and Why3 typing rules (all of which are standard, but which force the tool in question to include a virtually identical set of features). We note that more broadly, our approach relies on the overall design of Why3, in particular in its distinction between typing and semantics. Dafny, for instance, combines the two: certain typing conditions (e.g. pattern matching exhaustiveness, recursive function termination) are checked at verification time. This increases flexibility (e.g. one can have a non-exhaustive match and prove that the excluded case is unreachable) but would require a somewhat different design. In particular, rather than prove that for well-typed proof tasks, transformations are sound, we would need to prove that the generated formulas imply *both* well-typing and semantic properties, a more monolithic approach.

Our compiler is more specialized to Why3, though as we mentioned, our inductive predicate encoding and axiomatization is a partial exception to this. We additionally showed in §5.5 that our approach to axiomatizing ADTs extends to other possible axiomatizations. Finally, the pattern matching compiler is quite general: our implementation already allows arbitrary term-like action types, the termination proofs would be useful for any matrix-decomposition-based method, and the proofs of soundness could be refactored to remove dependence on term_rep (though it is crucial that the semantics for pattern matching behaves like match_rep).

Separately, one could imagine performing a similar formalization for a stronger logic like the Calculus of Inductive Constructions. MetaCoq [102] (see §3.5) 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 the choice of axioms is telling: 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 are based primarily on the classical nature of Why3; we use classical logic (Law of the Excluded Middle + 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.
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 ϵ 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.

However, we do rely frequently on functional extensionality and heavily on the Uniqueness of Identity Proofs (UIP) axiom (implied by the law of the excluded middle) for dealing with dependent typecasts.

7.6 Future Work

Improving Foundational Why3's Guarantees Currently, there is a gap remaining between Foundational Why3 and Why3Sem, as the proofs of equivalence between the two layers are incomplete. We have shown how such proofs can be implemented, and there are no large obstacles to completing them.[7] Additionally, the theorem statements themselves can be strengthened – we could give a proper type system for Foundational Why3 beyond types_wf, very similar (though not identical) to that of P-FOLDR and prove (1) the semantics of well-typed Foundational Why3 terms and tasks can be given by well-typed core terms and tasks and (2) every Foundational Why3 function and transformation, given well-typed inputs, will succeed. Finally, we would show that the API constructors produce well-typed AST nodes. Composing these results would allow us to prove that constructing tasks using the API and then running transformations is sound, well-typed, and does not crash, with very few assumptions on the global state.

Generalizing Why3Sem As we discussed in §7.5, our formalization is particular to Why3, but many individual components are not – ADTs, inductive predicates, pattern matching, and recursive functions are all widely used in languages and verification tools. Our W-type-based ADT encoding in particular could be generalized, both to be completely independent of Why3 and to support a larger class of ADTs, e.g. those with function arguments, GADTs, and even possibly dependent types. Though many of these are not necessary for Why3, such an encoding could be helpful in more general contexts. It would also be useful to fully generalize the pattern matching compiler and its proofs, including to enable more heuristics about which rows/columns/constructors to expand and simplify. This would result in a generic framework for writing verified, efficient pattern matching compilers.

[7] Some transformations like eliminate_algebraic are implemented fairly differently – the Foundational Why3 implementation uses state to keep track of newly created symbol names, while the purely functional compiler version needs no such state. To prove equivalence, one could write an alternate core eliminate_algebraic and prove equivalence with the compiler's implementation, separately proving equivalence with the Foundational Why3 version as described in §6.3.4.

A Coq Back-end for Why3 Our proof system shows that one can soundly prove Why3 goals in Coq, but this system is impractical and slow. Instead, we could generalize our semantics for ADTs and full interpretations to allow users to provide candidate representations satisfying the needed properties. Then, we could use MetaCoq to prove that Coq types and functions have these properties, enabling one to reason about Why3 proof goals in idiomatic Coq rather than in the deep embedding. For example, one would use Coq `lists` rather than the W-type encoding, and induction would be done directly within Coq instead of as a tactic traversing the deeply embedded proof task AST. This would then serve as a verified alternative to the existing Why3 Coq back-end and would permit a seamless and proved-sound implementation of Why3's theory realization capabilities.

A Why3 Back-end for Coq Separately, we could use this verified Why3 implementation as a back-end to Coq itself to provide more automation. We would use MetaCoq to reify first-order Coq goals and convert them to our Why3 embedding, thereby increasing the automation achievable for many goals. However, it is unclear if this would give us much beyond Sniper, which is built on SMTCoq and uses a similar reification. It could enable use of solvers and features beyond those supported by SMTCoq, though without soundness proofs.

End-to-End Soundness There are still two pieces missing for this system to be a fully foundational IVL – first, while we have focused on the logically interesting and complex transformations for this thesis, there are additional transformations needed for practical use – for propositional simplification, eliminating constants, using SMT solver built-ins (e.g. ints), etc. Implementing and verifying these transformations should be significantly more straightforward than those we have already completed. Second, we must extend the soundness guarantees to the SMT solver output. Here the picture is a bit murky: while there is some work towards this goal (§1.2.1), it will likely require further advances in SMT solver proof certificates until it is practical to remove the solver itself from the TCB. We note, however, that soundness is not all-or-nothing; reducing the TCB to the single solver layer is a big improvement from additionally trusting another IVL layer and the boundary.

Connecting to Front-End Verifiers Lastly, we would like to connect this work with a front-end verifier, particularly VST. Foundational Why3 could serve as an alternate back-end for simpler C specifications whose proof does not require complex inductive reasoning. VST-A [114] could be particularly helpful in this respect, as it transforms the annotated program into a set of straight-line program fragments with easier proof goals. There are still open questions concerning the best way to integrate such a back-end with the existing VST methodology – since VST function specifications can involve arbitrary Coq propositions, we would need a principled and ergonomic way of combining first-order and higher-order specifications.

7.7 Conclusion

In this thesis, we presented a foundationally verified version of the Why3 intermediate verification language that includes the features critical for writing and reasoning about rich, functional specifications – polymorphism, algebraic data types, pattern matching, recursive functions and predicates, and inductive predicates. In Chapter 2, we described the P-FOLDR logic extending FOL with these features and formalized the syntax and type system of this logic. In Chapter 3, we gave a novel formal semantics for P-FOLDR in Coq and presented a sound-by-construction proof system to facilitate reasoning (including inductive reasoning) in this logic. In Chapter 4, we gave the first proved-correct general-purpose pattern matching compiler. In Chapter 5, we wrote and verified a compiler from P-FOLDR to polymorphic first-order logic, providing the first mechanized proof of a first-order axiomatization of ADTs. In Chapter 6, we described a Coq implementation of a subset of the Why3 OCaml API, using a lightweight design based on modified extraction to OCaml to bridge the gap between stateful and stateless code. Finally, in Chapter 7, we demonstrated that this implementation can be proved correct, is practical, and encompasses a large enough subset of Why3 to handle goals that arise in real-world examples. We believe that this work – the first real-world IVL with a machine-checked proof of correctness – fills a gap in the verification-tool landscape and that it can serve as a key building block to enable foundationally sound program verifiers that retain the automation, usability, and convenience of IVL- and SMT-based tools.

Bibliography

[1] AWS Encryption SDK for Dafny. Amazon Web Services, Feb. 2025.

[2] Abrahamsson, O., Ho, S., Kanabar, H., Kumar, R., Myreen, M. O., Norrish, M., and Tan, Y. K. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *Journal of Automated Reasoning 64*, 7 (Oct. 2020), 1287–1306.

[3] Abrahamsson, O., Myreen, M. O., Kumar, R., and Sewell, T. Candle: A Verified Implementation of HOL Light. In *13th International Conference on Interactive Theorem Proving (ITP 2022)* (Haifa, Israel, 2022), J. Andronick and L. de Moura, Eds., vol. 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 3:1–3:17.

[4] Adjedj, A., Lennon-Bertrand, M., Maillard, K., Pédrot, P.-M., and Pujet, L. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK, Jan. 2024), CPP 2024, Association for Computing Machinery, pp. 230–245.

[5] Anand, A., Appel, A. W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O. S., Sozeau, M., and Weaver, M. CertiCoq: A Verified Compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for Programming Languages* (Paris, France, Jan. 2017).

[6] Anand, A., and Rahli, V. Towards a Formally Verified Proof Assistant. In *Interactive Theorem Proving* (Vienna, Austria, 2014), G. Klein and R. Gamboa, Eds., Springer International Publishing, pp. 27–44.

[7] Appel, A. W. *Program Logics for Certified Compilers.* Cambridge University Press, Cambridge, 2014.

[8] Appel, A. W. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS) 37*, 2 (Apr. 2015), 7:1–7:31.

[9] Appel, A. W., and Leroy, X. Efficient Extensional Binary Tries. *Journal of Automated Reasoning 67*, 1 (Jan. 2023), 8.

[10] Arquint, L., Schwerhoff, M., Mehta, V., and Müller, P. A Generic Methodology for the Modular Verification of Security Protocol Implementations. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark, Nov. 2023), CCS '23, Association for Computing Machinery, pp. 1377–1391.

[11] Augustsson, L. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture* (Nancy, France, 1985), J.-P. Jouannaud, Ed., Springer, pp. 368–381.

[12] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. Cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Munich, Germany, 2022), D. Fisman and G. Rosu, Eds., Springer International Publishing, pp. 415–442.

[13] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal*

Methods for Components and Objects (Amsterdam, Netherlands, 2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds., Springer, pp. 364–387.

[14] BARRAS, B. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning 3*, 1 (Oct. 2010), 29–48.

[15] BARRETT, C., SHIKANIAN, I., AND TINELLI, C. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. *Electronic Notes in Theoretical Computer Science 174*, 8 (June 2007), 23–37.

[16] BARRETT, C., AND TINELLI, C. Satisfiability Modulo Theories. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer International Publishing, Cham, 2018, pp. 305–343.

[17] BARTHE, G., DUPRESSOIR, F., GRÉGOIRE, B., KUNZ, C., SCHMIDT, B., AND STRUB, P.-Y. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, A. Aldini, J. Lopez, and F. Martinelli, Eds. Springer International Publishing, Bertinoro, Italy, 2014, pp. 146–166.

[18] BESSON, F. Itauto: An Extensible Intuitionistic SAT Solver. In *12th International Conference on Interactive Theorem Proving (ITP 2021)* (2021), L. Cohen and C. Kaliszyk, Eds., vol. 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 9:1–9:18.

[19] BLANCHETTE, J. C., BÖHME, S., AND PAULSON, L. C. Extending Sledgehammer with SMT Solvers. In *Automated Deduction – CADE-23* (Wrocław, Poland, 2011), N. Bjørner and V. Sofronie-Stokkermans, Eds., Springer, pp. 116–130.

[20] BLANCHETTE, J. C., BÖHME, S., POPESCU, A., AND SMALLBONE, N. Encoding Monomorphic and Polymorphic Types. In *Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy, 2013), N. Piterman and S. A. Smolka, Eds., Springer, pp. 493–507.

[21] Blanchette, J. C., and Popescu, A. Mechanizing the Metatheory of Sledgehammer. In *Frontiers of Combining Systems* (Nancy, France, 2013), P. Fontaine, C. Ringeissen, and R. A. Schmidt, Eds., Springer, pp. 245–260.

[22] Blot, V., Cousineau, D., Crance, E., de Prisque, L. D., Keller, C., Mahboubi, A., and Vial, P. Compositional Pre-processing for Automated Reasoning in Dependent Type Theory. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, USA, Jan. 2023), CPP 2023, Association for Computing Machinery, pp. 63–77.

[23] Blot, V., Dubois de Prisque, L., Keller, C., and Vial, P. General Automation in Coq through Modular Transformations. In *Proceedings of the Seventh Workshop on Proof eXchange for Theorem Proving* (Pittsburgh, United States, July 2021).

[24] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages* (Wrocław, Poland, 2011), p. 53.

[25] Bobot, F., and Paskevich, A. Expressing Polymorphic Types in a Many-Sorted Language. In *Frontiers of Combining Systems* (Saarbrücken, Germany, 2011), C. Tinelli and V. Sofronie-Stokkermans, Eds., Springer, pp. 87–102.

[26] Böhm, C., and Berarducci, A. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science 39* (Jan. 1985), 135–154.

[27] Boldo, S., and Melquiond, G. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic* (Tuebingen, Germany, July 2011), pp. 243–252.

[28] Boulmé, S. *Formally Verified Defensive Programming (Efficient Coq-verified Computations from Untrusted ML Oracles).* PhD thesis, Université Grenoble-Alpes, Sept. 2021.

[29] Braibant, T., Jourdan, J.-H., and Monniaux, D. Implementing Hash-Consed Structures in Coq. In *Interactive Theorem Proving* (Rennes, France, 2013), S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., Springer, pp. 477–483.

[30] Cao, Q., Beringer, L., Gruetter, S., Dodds, J., and Appel, A. W. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning 61*, 1 (June 2018), 367–422.

[31] Chakarov, A., Geldenhuys, J., Heck, M., Hicks, M., Huang, S., Jaloyan, G. A., Joshi, A., Leino, R., Mayer, M., McLaughlin, S., Mritunjai, A., Claudel, C. P., Porncharoenwase, S., Rabe, F., Rapoport, M., Reger, G., Roux, C., Rungta, N., Salkeld, R., Schlaipfer, M., Schoepe, D., Schwartzentruber, J., Tasiran, S., Tomb, A., Torlak, E., Tristan, J., Wagner, L., Whalen, M., Willems, R., Xiang, J., Byun, T. J., Cohen, J., Wang, R., Jang, J., Rath, J., Syeda, H. T., Wagner, D., and Yuan, Y. Formally Verified Cloud-Scale Authorization. In *ICSE 2025: 47th International Conference on Software Engineering* (Ottawa, Canada, 2025).

[32] Chareton, C., Bardin, S., Bobot, F., Perrelle, V., and Valiron, B. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *Programming Languages and Systems* (2021), N. Yoshida, Ed., Springer International Publishing, pp. 148–177.

[33] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, Oct. 2015), SOSP '15, Association for Computing Machinery, pp. 18–37.

[34] Chlipala, A. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, Nov. 2013.

[35] Cohen, J. M. Implementing OCaml APIs in Coq. In *CoqPL'25 The Eleventh International Workshop on Coq for Programming Languages* (Denver, USA, Jan. 2025).

[36] Cohen, J. M., and Johnson-Freyd, P. A Formalization of Core Why3 in Coq. *Proceedings of the ACM on Programming Languages 8*, POPL (Jan. 2024), 60:1789–60:1818.

[37] Cohen, J. M., Wang, Q., and Appel, A. W. Verified Erasure Correction in Coq with MathComp and VST. In *Computer Aided Verification* (Haifa, Israel, 2022), S. Shoham and Y. Vizel, Eds., Springer International Publishing, pp. 272–292.

[38] Conchon, S., Coquereau, A., Iguernlala, M., and Mebsout, A. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories* (Oxford, UK, July 2018).

[39] Couchot, J.-F., and Lescuyer, S. Handling Polymorphism in Automated Deduction. In *Automated Deduction – CADE-21* (Bremen, Germany, 2007), F. Pfenning, Ed., Springer, pp. 263–278.

[40] Czajka, Ł., and Kaliszyk, C. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning 61*, 1 (June 2018), 423–453.

[41] Dardinier, T., Sammler, M., Parthasarathy, G., Summers, A. J., and Müller, P. Formal Foundations for Translational Separation Logic Verifiers. *Proceedings of the ACM on Programming Languages 9*, POPL (Jan. 2025), 20:569–20:599.

[42] Daumas, M., and Melquiond, G. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS) 37*, 1 (Jan. 2010), 2:1–2:20.

[43] de Moura, L., and Bjørner, N. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary, 2008), C. R. Ramakrishnan and J. Rehof, Eds., Springer, pp. 337–340.

[44] Denis, X., Jourdan, J.-H., and Marché, C. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering* (Berlin, Germany, 2022), A. Riesco and M. Zhang, Eds., Springer International Publishing, pp. 90–105.

[45] Eilers, M., and Müller, P. Nagini: A Static Verifier for Python. In *Computer Aided Verification* (Oxford, UK, 2018), H. Chockler and G. Weissenbacher, Eds., Springer International Publishing, pp. 596–603.

[46] Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., and Barrett, C. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification* (Heidelberg, Germany, 2017), R. Majumdar and V. Kunčak, Eds., Springer International Publishing, pp. 126–133.

[47] Erbsen, A., Gruetter, S., Choi, J., Wood, C., and Chlipala, A. 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* (June 2021), PLDI 2021, Association for Computing Machinery, pp. 604–619.

[48] Filliâtre, J.-C. One Logic to Use Them All. In *Automated Deduction – CADE-24* (Lake Placid, New York, 2013), M. P. Bonacina, Ed., Springer, pp. 1–20.

[49] Filliâtre, J.-C., and Conchon, S. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, Sept. 2006), ML '06, Association for Computing Machinery, pp. 12–19.

[50] Filliâtre, J.-C., and Paskevich, A. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems* (Rome, Italy, 2013), M. Felleisen and P. Gardner, Eds., Springer, pp. 125–128.

[51] Forster, Y., Kirst, D., and Wehr, D. Completeness theorems for first-order logic analysed in constructive type theory: Extended version. *Journal of Logic and Computation 31*, 1 (Jan. 2021), 112–151.

[52] Gäher, L., Sammler, M., Jung, R., Krebbers, R., and Dreyer, D. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proceedings of the ACM on Programming Languages 8*, PLDI (June 2024), 192:1115–192:1139.

[53] Garchery, Q. A Framework for Proof-carrying Logical Transformations. In *Proceedings of the Seventh Workshop on Proof eXchange for Theorem Proving* (Pittsburgh, United States, July 2021), vol. 336, pp. 5–23.

[54] Gonthier, G., and Mahboubi, A. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning 3*, 2 (2010), 95–152.

[55] Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the ACM (JACM) 40*, 1 (Jan. 1993), 143–184.

[56] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, USA, Oct. 2015), Association for Computing Machinery, pp. 1–17.

[57] Herms, P. *Certification of a Tool Chain for Deductive Program Verification.* PhD thesis, Université Paris Sud - Paris XI, Jan. 2013.

[58] Herms, P., Marché, C., and Monate, B. A Certified Multi-prover Verification Condition Generator. In *Verified Software: Theories, Tools, Experiments* (Philadelphia, PA, 2012), R. Joshi, P. Müller, and A. Podelski, Eds., Springer, pp. 2–17.

[59] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods* (Pasadena, California, USA, 2011), M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., Springer, pp. 41–55.

[60] Jacobs, B., Vogels, F., and Piessens, F. Featherweight VeriFast. *Logical Methods in Computer Science Volume 11, Issue 3* (Sept. 2015).

[61] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: A software analysis perspective. *Formal Aspects of Computing 27*, 3 (May 2015), 573–609.

[62] Koh, N., Li, Y., Li, Y., Xia, L.-y., Beringer, L., Honoré, W., Mansky, W., Pierce, B. C., and Zdancewic, S. From C to interaction trees: Specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal, Jan. 2019), Association for Computing Machinery, pp. 234–248.

[63] Korkut, J., Stark, K., and Appel, A. W. A Verified Foreign Function Interface between Coq and C. *Proceedings of the ACM on Programming Languages 9*, POPL (Jan. 2025), 24:687–24:717.

[64] Krauss, A. Partial Recursive Functions in Higher-Order Logic. In *Automated Reasoning* (Seattle, USA, 2006), U. Furbach and N. Shankar, Eds., Springer, pp. 589–603.

[65] Krebbers, R. Efficient, Extensional, and Generic Finite Maps in Coq-std++. In *Coq Workshop 2023* (Białystok, Poland, July 2023).

[66] Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. CakeML: A verified implementation of ML. *ACM SIGPLAN Notices 49*, 1 (Jan. 2014), 179–191.

[67] Lammich, P. Automatic Data Refinement. In *Interactive Theorem Proving* (Rennes, France, 2013), S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., Springer, pp. 84–99.

[68] Lammich, P. Refinement to Imperative/HOL. In *Interactive Theorem Proving* (Nanjing, China, 2015), C. Urban and X. Zhang, Eds., Springer International Publishing, pp. 253–269.

[69] Lammich, P. Refinement based verification of imperative data structures. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (Saint Petersburg, Florida, USA, Jan. 2016), CPP 2016, Association for Computing Machinery, pp. 27–36.

[70] Laville, A. Implementation of lazy pattern matching algorithms. In *ESOP '88* (Nancy, France, 1988), H. Ganzinger, Ed., Springer, pp. 298–316.

[71] Le Fessant, F., and Maranget, L. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy, Oct. 2001), Association for Computing Machinery, pp. 26–37.

[72] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal, 2010), E. M. Clarke and A. Voronkov, Eds., Springer, pp. 348–370.

[73] Leino, K. R. M., and Rümmer, P. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *Tools and Algorithms for the Construction and Analysis of Systems* (Paphos, Cyprus, 2010), J. Esparza and R. Majumdar, Eds., Springer, pp. 312–327.

[74] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[75] Leroy, X. Well-Founded Recursion Done Right. In *CoqPL'24 The Tenth International Workshop on Coq for Programming Languages* (London, UK, Jan. 2024).

[76] Lourenço, C. B., and Pinto, J. S. Why3-do: The Way of Harmonious Distributed System Proofs. In *Programming Languages and Systems* (Munich, Germany, 2022), I. Sergey, Ed., Springer International Publishing, pp. 114–142.

[77] Maissen, A. Adding Algebraic Data Types to a Verification Language. Tech. rep., ETH Zurich, Apr. 2022.

[78] Maranget, L. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, USA, Jan. 1992), LFP '92, Association for Computing Machinery, pp. 21–31.

[79] Maranget, L. Warnings for pattern matching. *Journal of Functional Programming 17*, 3 (May 2007), 387–421.

[80] Maranget, L. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* (Victoria, British Columbia, Canada, Sept. 2008), ML '08, Association for Computing Machinery, pp. 35–46.

[81] Martin-Löf, P. Constructive Mathematics and Computer Programming. In *Studies in Logic and the Foundations of Mathematics*, L. J. Cohen, J. Łoś, H. Pfeiffer, and K.-P. Podewski, Eds., vol. 104 of *Logic, Methodology and Philosophy of Science VI.* Elsevier, Jan. 1982, pp. 153–175.

[82] Monniaux, D., and Boulmé, S. The Trusted Computing Base of the CompCert Verified Compiler. In *Programming Languages and Systems* (Munich, Germany, 2022), I. Sergey, Ed., Springer International Publishing, pp. 204–233.

[83] Müller, P., Schwerhoff, M., and Summers, A. J. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation* (St Petersburg, Florida, USA, 2016), B. Jobstmann and K. R. M. Leino, Eds., Springer, pp. 41–62.

[84] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. Ynot: Dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, British Columbia, Canada, Sept. 2008), ICFP '08, Association for Computing Machinery, pp. 229–240.

[85] Nipkow, T., Wenzel, M., Paulson, L. C., Goos, G., Hartmanis, J., and Van Leeuwen, J., Eds. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2002.

[86] O'Connor, R. Essential Incompleteness of Arithmetic Verified by Coq. In *Theorem Proving in Higher Order Logics* (Oxford, UK, 2005), J. Hurd and T. Melham, Eds., Springer, pp. 245–260.

[87] O'Hearn, P. W. Resources, concurrency, and local reasoning. *Theoretical Computer Science 375*, 1 (May 2007), 271–307.

[88] Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, 1998.

[89] Owre, S., Rushby, J. M., and Shankar, N. PVS: A prototype verification system. In *Automated Deduction—CADE-11* (Saratoga Springs, New York, USA, 1992), D. Kapur, Ed., Springer, pp. 748–752.

[90] Parthasarathy, G., Dardinier, T., Bonneau, B., Müller, P., and Summers, A. J. Towards Trustworthy Automated Program Verifiers: Formally Validating

Translations into an Intermediate Verification Language. *Proceedings of the ACM on Programming Languages 8*, PLDI (June 2024), 208:1510–208:1534.

[91] Parthasarathy, G., Müller, P., and Summers, A. J. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification* (2021), A. Silva and K. R. M. Leino, Eds., Springer International Publishing, pp. 704–727.

[92] Pereira, J. C., Klenze, T., Giampietro, S., Limbeck, M., Spiliopoulos, D., Wolf, F. A., Eilers, M., Sprenger, C., Basin, D., Müller, P., and Perrig, A. Protocols to Code: Formal Verification of a Next-Generation Internet Router, May 2024.

[93] Pereira, M., and Ravara, A. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification* (2021), A. Silva and K. R. M. Leino, Eds., Springer International Publishing, pp. 677–689.

[94] Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Copenhagen, Denmark, July 2002), IEEE Computer Society, pp. 55–74.

[95] Riazanov, A., and Voronkov, A. The design and implementation of VAMPIRE. *AI Communications 15*, 2,3 (Aug. 2002), 91–110.

[96] Rosskopf, S., and Nipkow, T. A Formalization and Proof Checker for Isabelle's Metalogic. *Journal of Automated Reasoning 67*, 1 (Dec. 2022), 1.

[97] Sakaguchi, K. Program extraction for mutable arrays. *Science of Computer Programming 191* (June 2020), 102372.

[98] Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., and Garg, D. 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 (June 2021), Association for Computing Machinery, pp. 158–174.

[99] Schurr, H.-J., Fleury, M., and Desharnais, M. Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant. In *Automated Deduction – CADE 28* (2021), A. Platzer and G. Sutcliffe, Eds., Springer International Publishing, pp. 450–467.

[100] Shah, A., Mora, F., and Seshia, S. A. An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes. *Proceedings of the AAAI Conference on Artificial Intelligence 38*, 8 (Mar. 2024), 8099–8107.

[101] Smans, J., Jacobs, B., and Piessens, F. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems (TOPLAS) 34*, 1 (May 2012), 2:1–2:58.

[102] Sozeau, M., Forster, Y., Lennon-Bertrand, M., Nielsen, J., Tabareau, N., and Winterhalter, T. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *Journal of the ACM 72*, 1 (Jan. 2025), 8:1–8:74.

[103] Sozeau, M., and Mangin, C. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages 3*, ICFP (July 2019), 86:1–86:29.

[104] Swierstra, W. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics* (Munich, Germany, 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Springer, pp. 440–451.

[105] Tekriwal, M., Appel, A. W., Kellison, A. E., Bindel, D., and Jeannin, J.-B. Verified Correctness, Accuracy, and Convergence of a Stationary Iterative Linear Solver: Jacobi Method. In *Intelligent Computer Mathematics: 16th International Conference, CICM 2023* (Cambridge, UK, Sept. 2023), Springer-Verlag, pp. 206–221.

[106] The Coq Development Team. The Coq Proof Assistant. Zenodo, Sept. 2024.

[107] Tuerk, T., Myreen, M. O., and Kumar, R. Pattern Matches in HOL:. In *Interactive Theorem Proving* (Nanjing, China, 2015), C. Urban and X. Zhang, Eds., Springer International Publishing, pp. 453–468.

[108] Vogels, F., Jacobs, B., and Piessens, F. A Machine Checked Soundness Proof for an Intermediate Verification Language. In *SOFSEM 2009: Theory and Practice of Computer Science* (Špindleruv Mlýn, Czech Republic, 2009), M. Nielsen, A. Kučera, P. B. Miltersen, C. Palamidessi, P. Tůma, and F. Valencia, Eds., Springer, pp. 570–581.

[109] Vogels, F., Jacobs, B., and Piessens, F. A machine-checked soundness proof for an efficient verification condition generator. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Catania, Italy, Mar. 2010), SAC '10, Association for Computing Machinery, pp. 2517–2522.

[110] Wang, Q., Pan, M., Wang, S., Doenges, R., Beringer, L., and Appel, A. W. Foundational Verification of Stateful P4 Packet Processing. In *14th International Conference on Interactive Theorem Proving (ITP 2023)* (Białystok, Poland, 2023), Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[111] Wils, S., and Jacobs, B. Certifying C program correctness with respect to CompCert with VeriFast, Oct. 2021.

[112] Wolf, F. A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J. C., and Müller, P. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification* (2021), A. Silva and K. R. M. Leino, Eds., Springer International Publishing, pp. 367–379.

[113] Xia, L.-y., Zakowski, Y., He, P., Hur, C.-K., Malecha, G., Pierce, B. C., and Zdancewic, S. Interaction trees: Representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages 4*, POPL (Dec. 2019), 51:1–51:32.

[114] Zhou, L., Qin, J., Wang, Q., Appel, A. W., and Cao, Q. VST-A: A Foundationally Sound Annotation Verifier. *Proceedings of the ACM on Programming Languages 8*, POPL (Jan. 2024), 69:2069–69:2098.