A Mechanized First-Order Theory of Algebraic Data Types with Pattern Matching

Joshua M. Cohen
Princeton University, USA

Abstract

Algebraic data types (ADTs) and pattern matching are widely used to write elegant functional programs and to specify program behavior. These constructs are critical to most general-purpose interactive theorem provers (e.g. Lean, Rocq/Coq), first-order SMT-based deductive verifiers (e.g. Dafny, VeriFast), and intermediate verification languages (e.g. Why3). Such features require layers of compilation - in Rocq, pattern matches are compiled to remove nesting, while SMT-based tools further axiomatize ADTs with a first-order specification. However, these critical steps have been omitted from prior formalizations of such toolchains (e.g. MetaRocq). We give the first proved-sound sophisticated pattern matching compiler (based on Maranget's compilation to decision trees) and first-order axiomatization of ADTs, both based on Why3 implementations. We prove the soundness of exhaustiveness checking, extending pen-and-paper proofs from the literature, and formulate a robustness property with which we find an exhaustiveness-related bug in Why3. We show that many of our proofs could be useful for reasoning about any first-order program verifier supporting ADTs.

2012 ACM Subject Classification Software and its engineering → Semantics; Theory of computation → Logic and verification

Keywords and phrases Pattern Matching Compilation, Algebraic Data Types, First-Order Logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.5

Acknowledgements I would like to thank Philip Johnson-Freyd, Andrew W. Appel, and the anonymous reviewers for their suggestions and feedback. This research was supported in part by National Science Foundation grant CCF-2219757.

1 Introduction

Algebraic data types form the backbone of functional programming languages; with recursion and exhaustive pattern matching, they enable elegant, concise, and type-safe functions over all kinds of list- and tree-like data. Reasoning about such functions requires only simple induction, and many proof assistants (e.g. Rocq, Lean, Isabelle) make heavy use of functional programming and induction. For instance, Rocq and Lean are implementations of the Calculus of Inductive Constructions (CIC), which extends the Calculus of Constructions with inductive types. But even SMT-based tools which target simpler first-order logic without induction still want to allow reasoning about ADTs, pattern matching, and/or recursive functions at the source level (see §2) – e.g. Dafny [27], VeriFast [21] and Frama-C [23] for C, Gobra [42] for Go, and Creusot [16] for Rust. In fact, these facilities are so useful that many *intermediate verification languages* upon which these deductive verifiers are built also include such features – both Why3 [10] (the back-end of Frama-C and Cruseot) and Viper [33] (the back-end of Gobra) have support for ADTs; Why3 additionally includes pattern matching and recursion. These SMT-based tools must *axiomatize* recursive types into first-order formulas. The exact set of axioms may vary; they typically include assertions of injectivity, disjointness, inversion, and non-circularity.

Similarly, while complex, nested, and simultaneous pattern matches are very useful at the source level, they must be compiled to simpler patterns in the underlying logic. In Rocq, this compilation is performed by the front-end, so the user sees the compiled version when

printing a term. In SMT-based tools, this compilation occurs when ADTs are axiomatized; the resulting compiled matches can be replaced with let-bindings and case analysis. ML and Haskell compilers must also eliminate pattern matching; many schemes have been studied and implemented [31, 26, 25]. The underlying algorithms are broadly similar in all these cases, though the correctness proofs depend on the setting – eager, lazy, or purely logical.

It is vital that these compilation steps are sound or else a verifier could "prove" something false. This is especially important because ADTs and their semantics are central to many verifiers, enabling many useful constructs (pattern matching, recursive functions, inductive predicates) while relying on subtle checks to provide guarantees of well-foundedness (e.g. strict positivity, termination checking, pattern matching exhaustiveness). Despite the centrality of these features, existing verifier formalizations often omit them. Neither Featherweight VeriFast [22], a formalization of a core subset of VeriFast, nor a proof-producing version of Viper [15] include ADTs, though the underlying tools do. Meanwhile, the MetaRocq formalization of CIC within Rocq [38] includes only simple patterns, assuming that Rocq's front-end has already done the pattern matching compilation.

A key difficulty is in specifying the semantics against which to prove such compilation steps correct. Cohen and Johnson-Freyd [13] addressed this by giving a Rocq formalization of the semantics of the Why3 logic (we will refer to this semantics as Why3Sem), which includes (mutually recursive) ADTs and pattern matching. Using Why3Sem, we take a key step towards filling this formalization gap by giving a mechanized proof of the Why3 pattern matching compiler and ADT axiomatization. Many of our results and methods are not specific to Why3 and would extend to other similar systems provided they had an appropriate formal semantics. In particular, we make the following contributions:

1. We give the first verified implementation of a sophisticated, general-purpose pattern matching compiler (based on the technique of Maranget [31]).
2. We use this compiler to implement exhaustiveness checking for Why3Sem, prove several results about the correctness and robustness of such matching (which extend proofs in the literature), and discover an exhaustiveness-related bug in Why3.
3. We use this to give the first mechanized proved-sound first-order axiomatization of ADTs.

Our proofs can be found at `https://github.com/joscoh/why3-semantics/tree/itp25`.

2 Background

Verification of Imperative Programs

Reasoning about functional programs is critical even in imperative settings. To manage the complexity of functional-correctness proofs about imperative programs, one frequently separates the proof 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 mathematical reasoning about the domain of interest. This approach has been successful at scale in a variety of domains. For instance, the VST [2] program logic for C in Rocq has been used to verify programs in cryptography [3], error-correcting codes [14], and numerical methods [40]; the functional correctness proofs use pure Rocq reasoning while VST handles the C-specific obligations. Dafny, a first-order, SMT-based verification-aware programming language, supports reasoning about both functional and imperative programs, and a similar layered approach enabled verification of distributed systems in Ironfleet [19] and a large-scale authorization engine used in production at AWS [11]. Other efforts involve a combination of tools; in the VerifiedSCION

project [36], the Gobra verifier is used for the imperative proofs, while the functional model proofs were completed in Isabelle. It is therefore critical that even imperative verifiers enable users to write and reason about functional programs; hence, support for features like ADTs and pattern matching is widespread among verifiers for both functional and imperative languages.

Why3 and its Semantics

Why3 is a verification framework serving as a target for many front-ends (e.g. C, Rust) and supporting multiple back-end solvers, including SMT solvers (e.g. Z3, CVC5, Alt-Ergo) and proof assistants (e.g. Rocq, Isabelle). Its logic [17] extends first-order logic with polymorphism, ADTs, pattern matching, recursive functions, and inductive predicates. This logic is broadly similar to those implemented by other SMT-based verifiers including Dafny and VeriFast and is compiled to logics for back-end solvers via *transformations*. In this paper, we are primarily concerned with two such transformations: `compile_match`, which compiles pattern matching to simple patterns, and `eliminate_algebraic`, which axiomatizes ADTs.

Why3Sem[13, 12] is a formal semantics for this Why3 logic in Rocq. It includes a deep embedding of Why3 types, patterns, terms, formulas, and definitions, as well as a formalized type system and a Tarski-style (denotational) semantics, in which Why3 terms and formulas are interpreted as objects in Rocq's logic. Why3Sem explicitly encodes recursive definitions as well-founded Rocq objects (e.g. ADTs as W-types [32]) but is designed so that users only need a set of higher-level properties about such recursive definitions, not the complex encodings.

Terms and formulas are given semantics under a particular *interpretation* of types ($[\![\cdot]\!]^\tau$) and function/predicate symbols ($[\![\cdot]\!]^\lambda$) and under a particular *valuation* (v) of type and free term variables (unlike [13], we disambiguate interpretations via superscripts). These interpretations must be consistent with the declared recursive definitions; for instance, for each ADT a (ignoring polymorphism for simplicity), the following must hold:

1. Injectivity: for constructor c, if $[\![c]\!]^\lambda(\boldsymbol{t_1}) = [\![c]\!]^\lambda(\boldsymbol{t_2})$, then $\boldsymbol{t_1} = \boldsymbol{t_2}$.
2. Disjointness: for constructors c_1 and c_2, if $[\![c_1]\!]^\lambda(\boldsymbol{t_1}) = [\![c_2]\!]^\lambda(\boldsymbol{t_2})$, then $c_1 = c_2$.
3. Given x of type $[\![a]\!]^\tau$, there is a function $\mathbf{find}(x)$ that gives constructor c and arguments $\boldsymbol{t_1}$ such that $x = [\![c]\!]^\lambda(\boldsymbol{t_1})$ (called `find_constr_rep` in Rocq).
4. A generalized induction principle over any mutual ADT m (not needed for this work).

Terms ($[\![\cdot]\!]^t_v$) and formulas ($[\![\cdot]\!]^f_v$) are interpreted in the natural way, with most Why3 logical constructs mapping to their Rocq counterparts (e.g. $[\![f_1 \wedge f_2]\!]^f_v = [\![f_1]\!]^f_v \wedge [\![f_2]\!]^f_v$). The most interesting and relevant case is for pattern matching (Figure 1), which proceeds in several steps. First, `match_val_single` or $[\![p, \tau, d]\!]^p$ (we will generally omit τ) matches a pattern p of type τ against d, returning an optional map of newly bound variables and their valuations (we use a monadic notation for the as-pattern case – ";" is a bind operator that propagates `None`). The constructor case is the most complex: if the type is an ADT, it uses **find** to determine if the constructor matches and recursively checks the arguments using the row matching $[\![ps, tys, ds]\!]^R$. A pattern match is evaluated (`match_rep` or $[\![t, ps]\!]^{ps}_v$) by iterating through the pattern-term list ps until a match is found, interpreting the corresponding term under v extended with the new bindings. If no such pattern is found, a default value is returned (all Why3 types are inhabited). Our work extends Why3Sem with an exhaustiveness check, making the default case unreachable. Finally, a pattern match term has the semantics (the formula case is similar):

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

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

$$
\begin{array}{rcl}
\mathsf{orep}\ v\ a\ o_1\ o_2 & = & \textbf{match}\ o_1\ \textbf{with} \mid \texttt{Some}\ m \to\ \texttt{Some}\ [\![a]\!]^t_{m \cup v} \mid \texttt{None}\ \to o_2\ \textbf{end} \\
[\![t, []]\!]^{ps}_v & = & \text{default} \\
[\![t, (p, t_1) :: ps]\!]^{ps}_v & = & \mathsf{orep}\ v\ t_1\ [\![p, [\![t]\!]^t_v]\!]^p\ [\![t, ps]\!]^{ps}_v
\end{array}
$$

Figure 1 Semantics for matching patterns ($[\![\cdot]\!]^p$), rows ($[\![\cdot]\!]^R$), and pattern-lists ($[\![\cdot]\!]^{ps}_v$)

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

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

Figure 2 A simultaneous pattern match and the corresponding pattern matrix

3 An Algorithm for Compiling Pattern Matches

The problem of compiling pattern matches to simpler constructs like decision trees is well-studied (§10), particularly for ML and Haskell compilers. Why3's pattern matching compiler is based on pattern matrix decomposition and is very similar to the techniques of Le Fessant and Maranget [26] and Maranget [31], which form the basis of the algorithm used in the OCaml compiler. We describe Why3's algorithm in detail, discuss its termination (§4), prove its soundness (§5), detail how it is used to implement exhaustiveness checking (§6), and describe an exhaustiveness bug we discovered in Why3 (§7). 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.

As is standard [26, 31], we view the (simultaneous) pattern match as a matrix P (Figure 2 shows an example); we will refer to the last column in the matrix as "actions" (in our case, terms or formulas). Patterns consist of variables, wildcards, constructors, disjunctions, and as-patterns; we will show later that we can assume the first column contains only wildcards and constructors. We then define two decompositions (Figure 3): specialization for constructor c ($S(c, P)$) and the default matrix ($D(P)$). Specialization gives the remaining matrix if the term in the first column is an instance of constructor c; it removes all non-c constructors from the matrix, replaces c with its k arguments, and replaces a wildcard with k wildcards. The default matrix gives the result assuming that t does not match *any* of the constructors appearing in the first column.

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
$\_$	$\begin{pmatrix} \_ & \overset{k}{\ldots} & \_ & p_{j,2} & \ldots & p_{j,n} & a_j \end{pmatrix}$

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

Figure 3 Definition of matrix decompositions $S(c, P)$ and $D(P)$ on row $\begin{pmatrix} p_{j,1} & \cdots & p_{j,n} & a_j \end{pmatrix}$

`compile`(ts, P)

1. If P is empty (no rows), return "Non-exhaustive".
2. Otherwise, if ts is empty, then return the first action in P.
3. Otherwise, let $ts = t :: tl$. 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 $t = c(al)$ for constructor c, if c is in the first column, return **`compile`**$(al +\!\!+ tl, S(c, P))$,[1] else return **`compile`**$(tl, D(P))$.
   c. Otherwise, let *base* be [ ] if all constructors for the ADT occur in the first column and [_→ **`compile`**$(tl, D(P))$] otherwise. Then construct list ps as follows: for each constructor c in the first column, add $c(vs) \to$**`compile`**$(vs +\!\!+ tl, S(c, P))$, where vs are fresh variables. Return **match** t **with** $ps +\!\!+ base$ **end**.

Figure 4 Compiling pattern matrix P and term list ts ($+\!\!+$ is concatenation)

With these decompositions, we can define the **`compile`** algorithm (Figure 4). We demonstrate **`compile`** on the example of Figure 2, which includes both nested and simultaneous matching. The first column contains both constructors; the specialized matrices are:

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

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 $P_4 := S(\mathsf{cons}, P_1) = \begin{pmatrix} \_ & \_ & x_4 \end{pmatrix}$. Expanding gives:

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

Each case can be simplified easily. The first **`compile`** evaluates to x_1 by Step 2, while the second evaluates to x_4 by repeated applications of Step 3a. The P_2 case is broadly similar; we omit the details but show the full compiled pattern match:

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

[1] The Why3 implementation (and our Rocq one) reverses al (and likewise vs in Step 3c and the $q_1 \ldots q_k$ in the constructor case for $S(c, P)$). We show the non-reversed version for simplicity.

This example also demonstrates how the algorithm checks exhaustiveness. Suppose we had not included the last case in the original pattern match (x_4). 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 $\texttt{compile}(l_2, P_1)$. Here we are again in Step 3c, but we find that $S(\mathsf{cons}, P_1)$ is empty, triggering Step 1 and causing the compilation to fail.

Now we fill in implementation details missing from Figure 4. The `simplify` transformation removes all non-constructor, non-wildcard patterns from the first column. It expands disjunctions and replaces variables at the outer level with let-bindings in the action column (it does not expand within constructors). Note that $\overline{p}$ represents the rest of the columns in the matrix, t_1 is the first matched term, and $+\!\!+$ again represents concatenation (i.e. gluing matrices together):

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

Our implementation, consistent with Why3, computes `simplify`, D, and all S matrices in a single pass, but we prove equivalence with a version that allows us to reason separately and assume the matrix is simplified when considering S and D.

Why3 implements the check in Step 3c that all ADT constructors appear in the first column in 2 ways: either the caller provides a function to retrieve the list of constructors for an ADT or `compile` relies on metadata from the constructor function symbols. The former requires more information but is useful for reporting unmatched cases (which we do not consider). We prove the two equivalent in any well-typed context.

4 Termination of `compile`

The first difficulty with formalizing `compile` is proving termination – the function recurses on non-structurally-smaller matrices $S(c, P)$ and $D(P)$. A termination argument is very tricky, as the matrix both expands (in `simplify` and in the wildcard case for $S(c, P)$) and contracts (in $D(P)$ and the constructor case of $S(c, P)$). Thus, several natural candidates for decreasing measures (e.g. the total size of the patterns in the matrix) fail. To build intuition, we first consider the case when there are no disjunctions. Here, we can almost use the total matrix size as a measure. The only complication is that the $S(c, P)$ case would not decrease unless the constructor size decreases by enough to offset the additional k wildcards. However, this wildcard increase can be statically bounded, so we could set the constructor size appropriately. Therefore, a version of `compile` that first expanded all disjunctions terminates. We lift this to the general case by considering the total size of the fully expanded matrix (which is never actually computed), parameterized by the constructor increase.

More formally, we give the definition of the full expansion of a pattern matrix ($E^M(P)$) in Figure 5. The difficult case is for constructor patterns: we must expand the entire row of arguments, which involves concatenating all the resulting expansions of each pattern in the row. 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: $|c(ps)|_n = n + \sum_{p \in ps} |p|_n$. To give a suitable upper bound, we let m be the largest number of arguments that *any* constructor in the matrix P takes. In the worst case, every row in the matrix but one starts with a wildcard, causing m new wildcards to appear per row. Thus, the total amount of measure added is bounded by $\text{len}(P') * m$, where $\text{len}(P')$ is the number

$$
\begin{aligned}
E^p(\_) &= [\_] \\
E^p(x) &= [\_] \\
E^p(p|q) &= E^p(p) +\!\!+ E^p(q) \\
E^p(p\ as\ x) &= E^p(p) \\
E^p(c(ps)) &= \Cup\!\!\!\Cup_{ps' \in E^R(ps)} [c(ps')]
\end{aligned}
\qquad
\begin{aligned}
E^R([]) &= [[]] \\
E^R(p :: ps) &= \Cup\!\!\!\Cup_{p' \in E^p(p), ps' \in E^R(ps)} [p' :: ps'] \\
E^M(P) &= \Cup\!\!\!\Cup_{r \in P} [E^R(r)]
\end{aligned}
$$

Figure 5 Expansion of patterns (E^p), rows (E^R), and pattern matrices (E^M)

$$
\begin{aligned}
[\![al, []]\!]^M_v &= \texttt{None} \\
[\![al, (r,t) :: rs]\!]^M_v &= \mathsf{orep}\ v\ t\ [\![al, r]\!]^R\ [\![al, rs]\!]^M_v
\end{aligned}
$$

Figure 6 Semantics of pattern-matrix matching (`matches_matrix`)

of rows of the current matrix P'. Since P' is created via calls to `simplify`, we can upper bound $\mathrm{len}(P')$ by $\mathrm{len}(E^M(P))$. We let $b(P) = \mathrm{len}(E^M(P)) * m$. The full termination metric is then $|E^M(P)|_{b(P)+1}$. Proving that `compile` terminates according to this measure involves 4 key steps:

1. We show that simplification does not change full expansion: $E^M(\texttt{simplify}(P)) = E^M(P)$.
2. We show the default case decreases: $\forall n, |E^M(D(P))|_n < |E^M(P)|_n$. This is not too hard to show, as either constructors or wildcards are removed when constructing D.
3. For the specialization case, we show that if constructor c appears in the first column of P and takes k arguments, $|E^M(S(c,P))|_n + n \leq |E^M(P)|_n + \mathrm{len}(E^M(P)) * k$. This proof is complex, as we must reason about the nested expansions for constructors.
4. Finally, since $(b(P)+1)$ depends on P, we show monotonicity: 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 [31, 26] follow the same basic structure but with additional optimizations (e.g, not always examining the first column, swapping columns, data structure sharing, etc); this termination metric should suffice in these settings with almost identical proof.

We implement `compile` in Rocq using Equations [39] and this size metric. Our function is parametric in the action type, constructor-finding method, and a flag governing Step 3b (see §7), returning an option that is `None` if the match is non-exhaustive.

5 Soundness of `compile`

We want to prove the semantic correctness of `compile`. To state this theorem, we first define the semantics of matching against a pattern matrix ($[\![al, P]\!]^M_v$), which builds on pattern- and row-matching (Figure 1). Namely, $[\![al, P]\!]^M_v$ (Figure 6) iterates through each row until it finds a match (with $[\![\cdot]\!]^R$), then evaluates the matched action under the valuation v extended with the new bindings (this reduces to `match_rep` for a single-column matrix). We define $[\![ts]\!]^T_v$ to be the (heterogeneous) list generated by $[\![t]\!]^t_v$ for each t in ts (recall that $[\![t]\!]^t_v$ is the interpretation for terms under valuation v). Then the correctness theorem is:

▶ **Theorem 1.** *Let tms be a term-list, let P be a pattern matrix, and let v be a valuation. Suppose that tms and P are well-typed.*[2] *Also suppose that there are no names in common between the free variables of tms and the pattern variables of P. Then, if* `compile` *tms* $P =$ `Some` *t, then* $[\![[\![tms]\!]_v^T, P]\!]_v^M =$ `Some` $[\![t]\!]_v^t$.

In other words, if `compile` succeeds and produces term t, then the pattern match succeeds and results in a term semantically equivalent to t. The proof involves many pieces.

First, we prove that `simplify` preserves the semantics. Note that a row beginning with a disjunction becomes multiple rows in the simplified 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. For example, given `match y, z with | x, y → f x y end`, the result should be equivalent to `f y z`. However, `simplify` gives `let y = z in (let x = y in f y x)`, equivalent to `f z z`. To avoid this, we require the condition on variable names in Theorem 1. Then we prove:

▶ **Lemma 2.** *Assume that $t :: ts$ and P have no variable names in common. Then* $[\![[\![t :: ts]\!]_v^T, \mathit{simplify}\ t\ P]\!]_v^M = [\![[\![t :: ts]\!]_v^T, P]\!]_v^M$

Next, we reason about the matrices $D(P)$ and $S(c, P)$, assuming that the input matrix is simplified and well-typed. First, we define the notion "term t is semantically equivalent to $c(al)$" (i.e. $[\![t]\!]_v^t = [\![c]\!]^\lambda(al)$) for constructor c in ADT a; we call this predicate `tm_semantic_constr`. Then we prove that, given any term t of ADT type a, we can find the constructor c and arguments al such that `tm_semantic_constr` t c al; this is a direct consequence of `find_constr_rep` (§2). We prove 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 3.** *Suppose* `tm_semantic_constr` *t c al. Then* $[\![c(ps), [\![t]\!]_v^t]\!]^p = [\![al, ps]\!]^R$.

The proof is straightforward, relying on the definition of $[\![\cdot]\!]^p$ and the injectivity and disjointness of constructor interpretations (§2). Similarly, we then prove that if `tm_semantic_constr` holds of a different constructor, then the term does not match:

▶ **Lemma 4.** *Suppose $c \neq c'$ and* `tm_semantic_constr` *t c al. Then* $[\![c'(ps), [\![t]\!]_v^t]\!]^p =$ `None`.

Along with some additional lemmas about concatenation of pattern-rows, we now have all the pieces to prove the D and S cases. Recall that $D(P)$ is intended to represent the remaining pattern match if the term t does not match any constructor 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:

▶ **Lemma 5.** *Suppose* `tm_semantic_constr` *t c al but c does not appear in the first column of pattern matrix P. Then* $[\![[\![t :: ts]\!]_v^T, P]\!]_v^M = [\![[\![ts]\!]_v^T, D(P)]\!]_v^M$.

▶ **Lemma 6.** *Suppose t does not have ADT type. Then* $[\![[\![t :: ts]\!]_v^T, P]\!]_v^M = [\![[\![ts]\!]_v^T, D(P)]\!]_v^M$.

The proofs are simple; the first follows by Lemma 4, while the second uses the fact that there can be no constructors in the first column by typing. Next, 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:

[2] We omit typing details for brevity. For the purposes of this paper, well-typing ensures that all sizes are consistent and that constructors in patterns have the correct ADT type. For typing details, see [12].

▶ **Lemma 7.** *Suppose* `tm_semantic_constr` t c al. *Then* $[\![[\![t :: ts]\!]_v^T, P]\!]_v^M = [\![al +\!\!+ [\![ts]\!]_v^T, S(c, P)]\!]_v^M$.

We reason by induction on P. In the case where the first row starts with $c(ps)$, we split the pattern-row concatenation, using Lemma 3 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. Lastly, in the wildcard case, we prove that matching a row against all wildcards gives `Some` $\emptyset$. With this, we have all the pieces we need for the proof of Theorem 1. The full proof is quite complex; we give a sketch of an interesting case.

Proof. We want to prove that, for any valuation v, $[\![[\![t :: ts]\!]_v^T, P]\!]_v^M =$ `Some` $[\![tm]\!]_v^t$. Step 3a uses Lemmas 5 and 6 depending on t's type; we focus on Step 3c. t must have ADT type, so 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 $[\![tm]\!]_v^t =$ `match_rep` t $(ps +\!\!+ base)$, where ps and $base$ are defined as in Figure 4. We consider 2 possible cases:

1. Assume c appears in the first column of P. Then, we can split ps into $ps_1 +\!\!+ (c(vs) \to$ `compile`$(vs +\!\!+ ts)) +\!\!+ ps_2$ such that c does not appear in the patterns in ps_1 or ps_2 (we ignore the `option` return type for simplicity, as we assume all recursive calls succeed). Recall that `match_rep` works by iterating over the pattern list until a match succeeds. By Lemma 4, no pattern in ps_1 matches t. The first successful match is therefore against the $c(vs)$ term: we simplify first by invoking Lemma 3 and then by noting that since vs are variables, each inner match succeeds. The resulting valuation m maps $vs \to al$. Thus, letting tm_1 be such that `compile`$(vs +\!\!+ ts) =$ `Some` tm_1, we have that $[\![tm]\!]_v^t = [\![tm_1]\!]_{m \cup v}^t$. By the induction hypothesis, we have that $[\![[\![vs +\!\!+ ts]\!]_{m\cup v}^T, S(c, P)]\!]_{m\cup v}^M =$ `Some`$[\![tm_1]\!]_{m\cup v}^t$. Finally, we note (after splitting the concatenation in $[\![\cdot]\!]^T$) that $[\![vs]\!]_{m\cup v}^T = al$, since m maps $vs \to al$. The vs variables are fresh (and hence do not appear in ts or $S(c, P)$); therefore $[\![ts]\!]_{m\cup v}^T = [\![ts]\!]_v^T$ and so the desired equality holds by Lemma 7.
2. If c does not appear in the first column of P, nothing in ps matches by Lemma 4; therefore $[\![tm]\!]_v =$ `match_rep` t $base$. Crucially, $base$ cannot be empty since there is at least one constructor not present in the first column of P. Thus, there must be a wildcard to match – the complete result follows from Lemma 5 and the induction hypothesis.

◀

Our work is the first machine-checked proof of such a compilation scheme based on pattern matrix decomposition. Maranget [31] gives a brief pen-and-paper correctness argument for similar scheme against a more restrictive syntactic definition of matching, discussed below.

6 Pattern Matching Exhaustiveness

An exhaustiveness check follows as an immediate corollary of Theorem 1.

▶ **Corollary 8.** *Under the assumptions of Theorem 1, if* $[\![[\![tms]\!]_v^T, P]\!]_v^M =$ `None`*, then* `compile` *tms* $P =$ `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 1 (and hence Corollary 8).

It is worthwhile to compare and contrast this with existing pen-and-paper proofs of a virtually identical scheme to prove non-exhaustiveness by Maranget [30]. Maranget discusses two versions of exhaustiveness checking: for ML-like call-by-value languages and for Haskell-like lazy languages. The theorems 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*.

By contrast, our setting is purely logical, not evaluation-based. Why3's logic does not have any notion of values, and terms can include free variables or uninterpreted symbols. This means that unlike the syntactic match relation in Maranget's theorems (defined over values), we can only reason about matching semantics under a particular interpretation and valuation. However, once an interpretation/valuation are fixed (and we reason semantically rather than syntactically), the setting becomes similar to the call-by-value one. But there is still a crucial difference: uninterpreted symbols can be interpreted as *any* constructor; such ambiguity is not present for values.

Similarly, lazy matching is distinct from our setting. 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. Once again, our setting has no such ordering, partial evaluation, or monotonicity, but 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. $v \leq w \iff \exists I, [\![v]\!]^t_I = [\![w]\!]^t_I$). This satisfies similar properties – 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 I and term list ts that does not match ($[\![[\![ts]\!]^T_I, P]\!]^M_I = \texttt{None}$). Equivalently, if matching succeeds *for all interpretations and for all terms of the correct type*, then `compile` returns `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).

7 Robustness of `compile`

While Why3 includes exhaustiveness checking, Why3Sem previously omitted it. We fix this by adding a check to the Why3Sem type system, requiring that for any term or formula **match** t **with** ps **end**, `compile` $[t]$ ps is `Some` (where ps becomes a single-column matrix). Since `compile` produces a pattern match in Step 3c, we must prove that this match itself passes the exhaustiveness checker. We prove this by showing that `compile` produces simple, obviously exhaustive patterns.

The larger difficulty is to show that the existing functions over Why3 terms that preserve typing also preserve the exhaustiveness check. Writing such a theorem is surprisingly difficult; such functions modify the term list (e.g. for substitution and rewriting), the term and pattern types (for type substitution), the action type (when axiomatizing recursive functions), and the variables in the pattern matrix (for α-conversion). In the end, we need a generic *robustness* theorem to show that exhaustiveness checking still succeeds under 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 latter turns out to be the trickier condition to formalize, as we need to ensure that an ADT type cannot be transformed into a non-ADT type. We encode this as an asymmetric relation ty_rel. We finally require that the two pattern matrices have the same "shape" – they are equal up to changing variable names.

We will state such a robustness theorem shortly, but unfortunately it does not hold of Why3's `compile`. If we allow the term list to change arbitrarily, a check that succeeds due to Step 3b may then fail. For example, given `match [1] with | [x] → x end` and the equality `H: [1] = y`, if one rewrites with `H`, the resulting term fails the exhaustiveness checker and Why3 crashes with a fatal error. While rewriting in such a "backwards" manner is unlikely in practice, this non-robustness also rules out potential optimizations like constant subexpression elimination. This is a bug in Why3 which we reported.[3]

One possible fix is to remove Step 3b of `compile`. This is the common approach to exhaustiveness checking (e.g. in Rocq and OCaml). But Why3 uses this step to compile simultaneous matching, implemented via tuples. Instead, we parameterize `compile` by a flag `simpl_constr` indicating whether Step 3b is used. In our Rocq 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 9) holds; for compilation, `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 §8.4. Here, we give the full robustness theorem.

▶ **Theorem 9.** *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 "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,[4] 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 will succeed on P_2, tms_2, and tys_2.

The proof is similar to Theorem 1; we prove the cases for `simplify`, S, D, and `compile`.

8 A Sound First-Order Axiomatization of ADTs

8.1 Axiomatizing ADTs

Figure 7 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. To compile this to first-order logic for SMT solvers, the recursive types, recursive functions, and pattern matching must be eliminated. We do not focus on the elimination of recursive functions here (though we do prove such a step sound); this transformation replaces `length` with its unfolding axiom.

[3] `https://gitlab.inria.fr/why3/why3/-/issues/903`
[4] `t_fun_equiv` is a relation that rules out the backwards rewriting case; we omit the precise definition.

```
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 7 ADT example in Why3

The elimination of ADTs (`eliminate_algebraic`) can be viewed as a 3-step process: first, the `compile_match` transformation uses `compile` to reduce everything to simple patterns. Then, the ADT types and constructors are replaced by abstract symbols as new abstract function symbols and axioms are introduced. Lastly, all pattern matches are eliminated using the newly introduced function symbols. Figure 8 shows 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; it returns the corresponding argument for the constructor matched. The indexer function describes which constructor an ADT instance belongs to. The disjointness axioms assert 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 some are only needed in certain situations (e.g. the indexer axioms imply disjointness). Nevertheless, we prove all the generated axioms sound to cover all possible cases.

The elimination of pattern matches (`rewriteT`/`rewriteF` in Why3) is less standard. 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 arguments given by the projections. If the pattern match occurs in a formula (as with `length`, which becomes `length_def`), `match t with | c(a) → f ...` becomes either `((∀ a. t = c(a) → f) ∧ ...)` or `((∃ a, t = c(a) ∧ f) ∨ ... )` depending on the polarity of the formula in which the match appears (we prove the two equivalent).

Of course, this is not the only possible ADT axiomatization. Dafny uses a broadly similar encoding [27], with projections and indexers, as well as axioms defining an order on ADTs (used as a guard on recursive calls, rather than the static checks in Why3). Sniper [9, 8], a tool to transform Rocq goals into first-order SMT ones (see §10), generates injectivity, disjointness, and inversion axioms. Other first-order axiomatizations of ADTs for SMT solvers [6, 37] provide `isf` functions indicating which constructor the element belongs to (rather than the indexers of Why3) 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. Sniper relies on Rocq's built-in simplification to automatically simplify the pattern match per constructor.

8.2 Proving Soundness

Soundness is a statement about transformations (steps in the Why3 compilation pipeline) over *proof tasks*, consisting of a context Γ, a set of assumptions Δ, and a goal g. A transformation may produce many proof tasks; soundness states that if all the resulting tasks are valid, so

```
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

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 8 Result of axiomatizing ADTs and eliminating pattern matching

was the original one. Validity ($\Gamma, \Delta \vDash g$) means that the assumptions Δ imply the goal g under all *full* interpretations – those consistent with defined types, functions, and predicates. To prove a transformation sound, we must prove that if $\Gamma', \Delta' \vDash g'$, then $\Gamma, \Delta \vDash g$, where primes represent the transformation result. In particular, we must show that for every full interpretation I over Γ, if $I \vDash \Delta$, then $I \vDash g$. We do this by constructing a translated interpretation I' over Γ' and showing:

▶ **Property 1.** *I' is a full interpretation for Γ'.*

▶ **Property 2.** *If $I \vDash \Delta$, then $I' \vDash \Delta'$.*

▶ **Property 3.** *If $I' \vDash g'$, then $I \vDash g$.*

For a rewriting transformation (`compile_match` and `rewriteF`) we must show that the rewrite preserves the semantics – we need both directions of the implication for Properties 2 and 3. Meanwhile, for the full `eliminate_algebraic`, we must show how to construct the interpretation I' and then prove that this satisfies the added axioms. Proving the result well-typed is also tricky – the context, the declared symbols, and the defined ADTs change substantially – but we do not give the details. In the following, we briefly describe the various parts of the proof: (1) defining the new interpretation I' and proving the added axioms satisfied, (2) compiling and eliminating pattern matches, and (3) putting it all together to prove soundness.

```
Definition indexer_interp {m a} (al: arg_list ...):=
(*Cast head of al to ADT type*)
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 9 Interpretation of indexer symbols

8.3 Defining the Interpretation

We are given an interpretation I and ADT a. Since I is full, it correctly interprets a so the properties of §2 hold. We describe how to interpret each newly added symbol in turn. First, an indexer symbol i is interpreted as follows: given semantic arguments al (a heterogeneous list of elements of the interpretations of the argument types of i), the first type argument of i is a, so the first (only) element of al has type $[\![a]\!]^\tau$ (here we ignore polymorphism for simplicity) – call this element x. `find_constr_rep` gives c_1 and al_1 such that $x = [\![c_1]\!]^\lambda(al_1)$. The interpretation simply returns the index of c_1 in the list of constructors of a. We show a simplified Rocq definition (without dependent type obligations) in Figure 9.

Projection symbols are similar. This time, given the ith projection symbol p of constructor c and semantic arguments al, we again know that the only element of al is an ADT type and again use `find_constr_rep` to get 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 selector is a bit more complicated. 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 polymorphism which we again omit for simplicity). Given selector semantic arguments al, we again know 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 $(i+1)^{\text{st}}$ argument of al – this encodes the idea that the selector should return its $(i+1)^{\text{st}}$ argument when called on the ith constructor.

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 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`, we would use the well-founded relation `adt_smaller` defined in Why3Sem that denotes structural inclusion and is used to prove the termination of Why3's recursive functions.

We construct I' by using the appropriate definitions for all of the new function symbols, ensuring that the newly added symbols do not overlap. Proving that I' satisfies the added axioms is quite straightforward (though dependent types add some complications) and only relies on properties of constructor interpretations. For example, to prove the inversion axiom, we use `find_constr_rep` to identify the input argument's constructor, then prove the clause in the disjunction corresponding to this constructor by unfolding the definitions of the projection interpretations and relying on injectivity and disjointness of constructors.

8.4 Compiling and Eliminating Pattern Matches

The transformation that eliminates complex and nested patterns, `compile_match`, walks over the given term or formula and calls `compile` (§3) on each pattern match. The main property

we need is semantic equivalence, which follows from Theorem 1. Recall that this theorem states 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 (§7), 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, proving that our new exhaustiveness check is more restrictive than the old one:

▶ **Theorem 10.** *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), while the other is in Step 3c. Here, we use the stronger exhaustiveness check to know that every constructor's compilation (using $S(c, P)$) succeeds; therefore, the particular one needed also succeeds. This is not obvious, as the two checks operate over different term lists – fresh variables and the constructor application arguments (hence we generalize the theorem to allow different term lists). The full semantic equivalence result for `compile_match` requires an additional α-conversion to satisfy the unique-name hypothesis of Theorem 1.

In fact, we need more information about `compile_match` for the pattern matching elimination functions `rewriteT/F`: the resulting pattern matches are simple, obviously exhaustive, and organized such that each pattern match consists of a nonempty list of unique constructors applied to variables, optionally followed by a wildcard. As an implicit corollary, this proves that every Why3 term, no matter how sophisticated the pattern match, can be reduced to an equivalent term consisting only of these very restricted matches.

Proving the correctness of the pattern matching elimination (`rewriteT/rewriteF`) is quite tricky, 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 omit the full proof, but we give a brief sketch of the main ideas in the term pattern matching case. Here, `match t with | ... c(vs)` $\rightarrow$ `e ... | _` $\rightarrow$ `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: the uniqueness of constructors allows us to reason about pattern matching largely syntactically rather than by unfolding the (recursive) definition of $[\![p, d]\!]^p$. In particular, on matched term t, if `tm_semantic_constr` t c al holds, we prove that if $c(vs) \rightarrow e$ appears in the pattern match, then `match_rep` evaluates to $[\![e]\!]^t_{vs \rightarrow al}$ (and otherwise `match_rep` evaluates to $[\![d]\!]^t$). We then reason by cases: either the matched term's constructor appears in the pattern match or not. In both cases, we simplify with the corresponding lemma and prove the equivalence.

Finally, we complete the proof of soundness as described in §8.2. Proving Properties 1 and 3 is straightforward using the semantic equivalence of `rewriteF`. We almost proved Property 2; however, `rewriteF` is also run on the newly added axioms. We show that this rewrite has no semantic effect on formulas without pattern matches or constructor applications.

9 Discussion

We have endeavored to keep our proved-correct implementations of `compile_match` and `eliminate_algebraic` as close as possible to their Why3 counterparts. These have proven useful in practice, both in Why3 itself and in tools like EasyCrypt [7] which supports ADTs, pattern matching, and recursive functions by compiling to Why3. While `compile` and `compile_match` are virtually identical to the Why3 implementation, our version of `eliminate_algebraic` differs from the Why3 version slightly (it is stateless and requires all types in a mutual block to be axiomatized or kept) but produces similar output.

Furthermore, we note that our proofs could extend to other solvers and settings. As we discussed in §8, we could follow a nearly identical approach for other ADT axiomatizations. Our pattern matching compiler is quite general: it 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 Why3Sem (though it is crucial that the semantics for pattern matching behaves like `match_rep`). We note also that our proofs do not rely on axioms beyond those already present in Why3Sem (classical logic, functional extensionality, and indefinite description); other than UIP, these axioms are only needed to define Why3Sem and thus our proofs would suffice in intutitionistic settings.

10 Related Work

Pattern Matching Compilation

Pattern matching compilation is a well-studied problem. Augustsson [4] presents a simple compilation scheme, introducing (implicitly) some of the ideas of the matrix-decomposition approach. Laville [25] and Maranget [29] 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 [26] and Maranget [31] to develop further heuristics and optimizations for efficient matching. Maranget [30] 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 mechanizing the proofs of such compilation schemes. Tuerk et al. [41] implement a pattern matching compiler for the HOL proof assistant using a non-decision-tree approach that aims to produce patterns closer to handwritten versions. The authors do not prove the compilation correct but extend CakeML's [24] proof-producing code generator to show that the compiled match in CakeML preserves the semantics of the compiled HOL match. CakeML itself provides a simple verified pattern matching compiler and restrictive exhaustiveness checker. It is quite limited; CakeML cannot prove the following match exhaustive:

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

The CertiCoq [1] verified compiler from Rocq to C does not prove anything about non-simple patterns, relying on Rocq's front-end to compile patterns before reification. It is based on the MetaRocq [38] formalization of Rocq, which makes the same assumption.

ADT Axiomatization

First-order axiomatizations of ADTs are common to support ADT theories within SMT solvers [6, 37] as well as in SMT-based verification tools which include higher-level reasoning

support (e.g. Dafny [27]). Other tools (e.g. Viper[28] and Gobra [42]) include ADTs as syntactic sugar for the derived axioms and do not include any complex pattern matching.

There is also little prior work proving the soundness of such axiomatizations. Sniper [9, 8] is a tool that turns certain Rocq goals into first-order formulas to enable use of SMT solvers; it axiomatizes ADTs and eliminates pattern matches (which have already been compiled by Rocq's front-end). This approach is *certifying*: it provides tactics to generate Rocq definitions and assertions (by reifying to MetaRocq) as well as proof scripts to prove that the assertions hold. Such proofs are generally quite simple, since one can use Rocq's built-in mechanisms for simplifying pattern matches and performing case analysis on inductive types. This approach is better suited for Rocq but would not extend to SMT-based systems like Why3 or Dafny, which cannot reason about ADTs except via axiomatization and translation to SMT. Furthermore, we give a *certified*, rather than certifying, implementation, proving soundness once and for all.

Soundness of Intermediate Verification Languages

There are several recent efforts to prove the soundness of IVLs. Parthasarathy et. al. [35] gives a formal semantics and certifying procedure (in Isabelle) for the Boogie [5] IVL, a simple imperative language with assertions in first-order logic. Parthasarathy et. al. [34] extend this with a certifying translation from the Viper IVL to Boogie. This translation is concerned with separation logic rather than with a functional assertion language. Darndinier et. al. [15] extend this further by connecting to a verified frontend language. Featherweight VeriFast [22] gives semantics and a certifying implementation for a subset of the VeriFast separation logic verifier; though VeriFast includes ADTs and recursive functions, Featherweight VeriFast does not. Herms [20] proves sound an early version of Why3 that did not include recursive types or functions. Garchery [18] validates some logical Why3 transformations; focusing primarily on propositional transformations and an induction principle over integers. Cohen and Johnson-Freyd [13] prove the soundness of Why3's axiomatization of inductive predicates and elimination of let-bindings with Why3Sem.

11 Conclusion

We have presented a sophisticated, real-world pattern matching compiler and exhaustiveness checker along with a first-order axiomatization of ADTs, the first such implementations mechanized and proved sound in a proof assistant. These features and compilation steps are critical in enabling powerful assertion languages that allow users to express elegant, functional specifications amenable to logical reasoning. Proved-correct implementations make it possible to retain these advantages while ensuring that the verifier is foundationally sound. Additionally, since our transformations are computable functions within a proof assistant, they could be directly connected to higher level language semantics (e.g. CompCert) or verifiers (e.g. VST) to improve automation without compromising soundness guarantees.

References

1 Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A Verified Compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for Programming Languages*, Paris, France, January 2017.

2 Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, Cambridge, 2014. doi:10.1017/CBO9781107256552.

3 Andrew W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7:1–7:31, April 2015. doi:10.1145/2701415.

4 Lennart Augustsson. Compiling Pattern Matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Nancy, France, 1985. Springer. doi:10.1007/3-540-15975-4_48.

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

6 Clark Barrett, Igor Shikanian, and Cesare Tinelli. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, June 2007. doi:10.1016/j.entcs.2006.11.037.

7 Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A Tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 146–166. Springer International Publishing, Bertinoro, Italy, 2014. doi:10.1007/978-3-319-10082-1_6.

8 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. Compositional Pre-processing for Automated Reasoning in Dependent Type Theory. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pages 63–77, Boston, USA, January 2023. Association for Computing Machinery. doi:10.1145/3573105.3575676.

9 Valentin Blot, Louise Dubois de Prisque, Chantal Keller, and Pierre Vial. General Automation in Coq through Modular Transformations. In *Proceedings of the Seventh Workshop on Proof eXchange for Theorem Proving*, Pittsburgh, United States, July 2021. doi:10.4204/EPTCS.336.3.

10 François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, page 53, Wrocław, Poland, 2011.

11 Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clement Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. Formally Verified Cloud-Scale Authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2508–2521, Ottawa, Canada, April 2025. IEEE Computer Society. doi:10.1109/ICSE55347.2025.00166.

12 Joshua M. Cohen. *A Foundationally Verified Intermediate Verification Language*. PhD thesis, Princeton University, United States – New Jersey, 2025.

13 Joshua M. Cohen and Philip Johnson-Freyd. A Formalization of Core Why3 in Coq. *Proceedings of the ACM on Programming Languages*, 8(POPL):60:1789–60:1818, January 2024. doi:10.1145/3632902.

14 Joshua M. Cohen, Qinshi Wang, and Andrew W. Appel. Verified Erasure Correction in Coq with MathComp and VST. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 272–292, Haifa, Israel, 2022. Springer International Publishing. doi:10.1007/978-3-031-13188-2_14.

15 Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. Formal Foundations for Translational Separation Logic Verifiers. *Proceedings of*

the ACM on Programming Languages, 9(POPL):20:569–20:599, January 2025. doi:10.1145/3704856.

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

17 Jean-Christophe Filliâtre. One Logic to Use Them All. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 1–20, Lake Placid, New York, 2013. Springer. doi:10.1007/978-3-642-38574-2_1.

18 Quentin Garchery. A Framework for Proof-carrying Logical Transformations. In *Proceedings of the Seventh Workshop on Proof eXchange for Theorem Proving*, volume 336, pages 5–23, Pittsburgh, United States, July 2021. doi:10.4204/EPTCS.336.2.

19 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, Monterey, California, USA, October 2015. Association for Computing Machinery. doi:10.1145/2815400.2815428.

20 Paolo Herms. *Certification of a Tool Chain for Deductive Program Verification*. PhD thesis, Université Paris Sud - Paris XI, January 2013.

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

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

23 Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. doi:10.1007/s00165-014-0326-7.

24 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, January 2014. doi:10.1145/2578855.2535841.

25 Alain Laville. Implementation of lazy pattern matching algorithms. In H. Ganzinger, editor, *ESOP '88*, pages 298–316, Nancy, France, 1988. Springer. doi:10.1007/3-540-19027-9_20.

26 Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 26–37, Florence, Italy, October 2001. Association for Computing Machinery. doi:10.1145/507635.507641.

27 K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Dakar, Senegal, 2010. Springer. doi:10.1007/978-3-642-17511-4_20.

28 Alessandro Maissen. Adding Algebraic Data Types to a Verification Language. Technical report, ETH Zurich, April 2022.

29 Luc Maranget. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 21–31, San Francisco, USA, January 1992. Association for Computing Machinery. doi:10.1145/141471.141499.

30 Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, May 2007. doi:10.1017/S0956796807006223.

31 Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, Victoria, British Columbia, Canada, September 2008. Association for Computing Machinery. doi:10.1145/1411304.1411311.

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

33 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, St Petersburg, Florida, USA, 2016. Springer. doi:10.1007/978-3-662-49122-5_2.

34 Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language. *Proceedings of the ACM on Programming Languages*, 8(PLDI):208:1510–208:1534, June 2024. doi:10.1145/3656438.

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

36 João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix A. Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. Protocols to Code: Formal Verification of a Next-Generation Internet Router, May 2024. doi:10.48550/arXiv.2405.06074.

37 Amar Shah, Federico Mora, and Sanjit A. Seshia. An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8099–8107, March 2024. doi:10.1609/aaai.v38i8.28649.

38 Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *Journal of the ACM*, 72(1):8:1–8:74, January 2025. doi:10.1145/3706056.

39 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):86:1–86:29, July 2019. doi:10.1145/3341690.

40 Mohit Tekriwal, Andrew W. Appel, Ariel E. Kellison, David Bindel, and Jean-Baptiste Jeannin. Verified Correctness, Accuracy, and Convergence of a Stationary Iterative Linear Solver: Jacobi Method. In *Intelligent Computer Mathematics: 16th International Conference, CICM 2023*, pages 206–221, Cambridge, UK, September 2023. Springer-Verlag. doi:10.1007/978-3-031-42753-4_14.

41 Thomas Tuerk, Magnus O. Myreen, and Ramana Kumar. Pattern Matches in HOL:. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 453–468, Nanjing, China, 2015. Springer International Publishing. doi:10.1007/978-3-319-22102-1_30.

42 Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular Specification and Verification of Go Programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 367–379. Springer International Publishing, 2021. doi:10.1007/978-3-030-81685-8_17.