Set Constraints, Pattern Match Analysis, and SMT

Joseph Eremondi*

University of British Columbia, Vancouver, Canada
jeremond@cs.ubc.ca

Abstract. Set constraints provide a highly general way to formulate program analyses. However, solving arbitrary boolean combinations of set constraints is \textsc{NEXPTIME}-hard. Moreover, while theoretical algorithms to solve arbitrary set constraints exist, they are either too complex to implement or too slow to ever run.

We present a translation that converts a set constraint formula into an SMT problem. Our technique allows for arbitrary boolean combinations of set constraints, and leverages the performance of modern SMT solvers. To show the usefulness of unrestricted set constraints, we use them to devise a pattern match analysis for functional languages, which ensures that missing cases of pattern matches are always unreachable. We implement our analysis in the Elm compiler and show that the our translation is fast enough to be used in practical verification.

Keywords: Program analysis · SMT · pattern-matching · set constraints

1 Introduction

Set constraints are a powerful tool for expressing a large number of program analyses in a generic way. Featuring recursive equations and inequations over variables denoting sets of values, set constraints allow us to model the sets of values an expression could possibly take. While they were an active area of research in decades prior, they have not seen widespread adoption. In their most general form, finding solutions for a conjunction of set constraints is \textsc{NEXPTIME}-complete. While efficient solvers have been developed for restricted versions of the set constraint problem [4, 24], solvers for unrestricted set constraints are not used in practice.

However, since the development of set constraints, there have been significant advances in solvers for \textsc{SAT}modulo theories (SMT). Although SMT requires exponential time in theory, solvers such as Z3 [28] and CVC4 [8] are able to solve a wide range of satisfiability problems in practice. Given the success of SMT-solvers in skirting the theoretical intractability of \textsc{SAT}, one wonders, can these solvers be used to solve set constraints? We show that this is possible with reasonable performance. Our full contributions are as follows:

* This material is based on work supported by the NSERC CGS-D scholarship.
• We devise a pattern match analysis for functional programming languages, expressed in terms of unrestricted set constraints (Sec. 2).

• We provide a method for translating unrestricted set constraint problems into SAT modulo UF, a logical theory with booleans, uninterpreted functions, and first order quantification (Sec. 3). Additionally, we show that projections, a construct traditionally difficult to formulate with set constraints, are easily formulated using disjunctions in SMT (Sec. 3.1).

• We implement the above translation and analysis, showing that they are usable for verification despite the theoretical NEXPTIME-completeness (Sec. 4).

Motivation: Pattern Match Analysis

Our primary interest in set constraints is using them to devise a function pattern match analysis. Many functional programming languages feature algebraic datatypes, where values of a datatype $D$ are formed by applying a constructor function to some arguments. Values of an algebraic type can be decomposed using pattern matching, where the programmer specifies a number of branches with free variables, and the program takes the first branch that matches the given value, binding the corresponding values to the free variables. If none of the patterns match the value, a runtime error is raised.

Many modern languages, such as Elm [12] and Rust [23] require that pattern matches be exhaustive, so that each pattern match has a branch for every possible value of the given type. This ensures that runtime errors are never raised due to unmatched patterns, and avoids the null-pointer exceptions that plague many procedural languages. However, the type systems of these languages cannot express all invariants. Consider the following pseudo-Haskell, with an algebraic type for shapes, and a function that calculates their area.

```haskell
data Shape =  
    Square Double  
  |  Circle Double  
  |  NGon [Double]  

area :: Shape -> Double  
area shape = case shape of  
  NGon sides -> ...  
  Square len -> len * len  
  Circle r -> pi * r * r  
  _ -> error "This cannot happen"
```

The above code is perfectly safe, since `simpleArea` can only be called from `area`, and will never be given an `NGon`. However, it is not robust to changes. If we add the constructor `Triangle Double Double Double` to our `Shape` definition, then both matches are still exhaustive, since the `_` pattern covers every possible case. However, we now may face a runtime error if `area` is given a `Triangle`. In general, requiring exhaustiveness forces the programmer to either manually raise an error or return a dummy value in an unreachable branch.

We propose an alternate approach: remove the catch-all case of `simpleArea`, and use a static analysis to determine that only values matching `Circle` or `Square`
will be passed in. Such analysis would mark the above code safe, but would signal unsafety if Triangle were added to the definition of Shape.

The analysis for this particular case is intuitive, but can be complex in general:

- Because functions may be recursive, we need to be able to handle recursive equations (or inequations) of possible pattern sets. For example, a program dealing with lists may generate a constraint of the form $X \subseteq \text{Nil} \cup \text{Cons}(\top, \text{Cons}(\top, X))$.

- We wish to encode first-match semantics: if a program takes a certain branch in the pattern match, then the matched value cannot possibly match any of the previous cases.

- We want our analysis to be as precise as possible, to raise as few false-negative as is possible. One way this to achieve this is to track what conditions must be true for a branch to be taken, and to only enforce constraints from that branch when it is reachable. If we use logical implication, we can express constraints of the form “if $x$ matches pattern $P_1$, then $y$ must match pattern $P_2$.”

Sec. 2 gives such an analysis, while Sec. 3 describe solving these constraints.

## 2 A Set Constraint based Pattern Match Analysis

Here, we describe an annotated type system type for pattern match analysis. It tracks the possible values that expressions may take. Instead of requiring that each match be exhaustive, we restrict functions to reject inputs that may not be covered by a pattern match in the function’s body. Types are refined by constraints, which are solved using an external solver (Sec. 3).

### 2.1 \(\lambda\text{Match} \): Syntax

We present \(\lambda\text{Match}\), a small, typed functional language, whose syntax we give in Fig. 1. In addition to functions and applications, we have a form \(K_D(\overline{t})\) which applies the data constructor \(K\) to the argument sequence \(\overline{t}\) to make a term of
\[
\begin{align*}
\Gamma \vdash \Delta \vdash t : \tau & \quad \text{(Expression typing)} \\
\text{VAR} & : \quad \Gamma | \Delta \vdash x : \forall X. \tau \\
\Gamma | \Delta \vdash x : [\tau / X] \tau & \\
\text{LAM} & : \quad \Gamma | \Delta \vdash x : \tau_1 \rightarrow \tau_2 \\
\Gamma | \Delta \vdash \lambda x. t : \tau_1 \rightarrow \tau_2 & \\
\text{APP} & : \quad \Gamma | \Delta \vdash t_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma | \Delta \vdash t_2 : \tau_1 & \\
\Gamma | \Delta \vdash \lambda x. t : \tau_1 \rightarrow \tau_2 & \\
\text{Ctor} & : \quad \Gamma | \Delta \vdash K(\tau) \in \Delta(D) \\
K(\tau) & \in \Delta(D) \\
\Gamma | \Delta \vdash K_D(t) : D & \\
\text{MAT} & : \quad \Gamma | \Delta \vdash P : \tau|\tau' \\
\Gamma | \Delta \vdash t : \tau & \\
\Gamma | \Delta \vdash P : \tau|\tau' & \\
\Gamma | \Delta \vdash t : \tau' & \\
\text{LET} & : \quad \Gamma | \Delta \vdash \text{let } x = t_1 \text{ in } t_2 : \tau_2 \\
x : \tau_1, X, \Gamma | \Delta \vdash x : \forall X. \tau_1 & \\
\Gamma | \Delta \vdash t_2 : \tau_2 & \\
\Gamma | \Delta \vdash \text{let } x = t_1 \text{ in } t_2 : \tau_2 & \\
\text{Ctor} & : \quad \Gamma | \Delta \vdash K(\tau) \in \Delta(D) \\
K(\tau) & \in \Delta(D) \\
\Gamma | \Delta \vdash K_D(\overline{P}) : D \cup \overline{\tau'} & \\
\end{align*}
\]

**Fig. 2.** Underlying Typing for Expressions and Patterns

Type \( D \). Conversely, the form \( \text{match } t' \text{ with } \{ P_i \Rightarrow t_i \} \) chooses the first branch \( P_i = t_i \) for which \( t' \) matches pattern \( P_i \), and then evaluates \( t_i \) after binding the matching parts of \( t' \) to the variables of \( P_i \). (Here, the notation \( \overline{O} \) denotes a sequence of objects matching metavariable \( O \)).

During typechecking, we will use two sorts of environments. Type environments \( \Gamma \) store free type variables, and map program variables to their types. Datatype environments \( \Delta \) store the names of each datatype \( D \), along with the name and argument-types of each constructor of \( D \).

### 2.2 The Underlying Type System

The language’s underlying type system is in the style of Damas and Milner [13], where monomorphic types are separated from polymorphic type schemes. We present declarative typing rules for the underlying system in Fig. 2, noting that they are standard. The analysis we present below assumes that inference has been performed ahead of time and that all expressions have a known underlying type. Alternately, one can imagine that the analysis operates on the underlying typing derivations, rather than plain terms.

### 2.3 Annotated Types

For our analysis, we annotate types with set expressions (Fig. 3), representing the possible shapes that the evaluation of an expression might have in some context.
Set constraints, pattern match analysis, and SMT

\[ V \in \text{SetVariable} \]

**Set expressions**

\[ E ::= V \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E \mid K_D(E) \mid K_D^{-1}(E) \mid \top \mid \perp \]

**Set constraints**

\[ C ::= E_1 \subseteq E_2 \mid C_1 \land C_2 \mid C_1 \lor C_2 \mid \neg C \mid T \mid \bot \]

**Fig. 3.** \( \lambda_{\text{MATCH}} \): annotations

Since each syntactic variant has a top-level annotation \( E \), we use \( T^E \) to denote an annotated type \( T \) along with its top-level annotation \( E \). A annotated types \( T^E \) replace underlying types \( \tau \) in our rules, and our analysis emits constraints on \( E \) that dictate its value. Constructors are interpreted element-wise when applied to set expressions. The interpretation of set expressions is intuitive, but we give them a formal semantics in Sec. 3.

2.4 The Analysis

We present our pattern match analysis is Fig. 4. The analysis is phrased as an annotated type system in the style of Nielson and Nielson [29]. The judgment \( \Gamma \mid \Delta \vdash C_p : T^E \& C \) says that, under context \( \Gamma \) with datatypes from \( \Delta \), if \( C_p \) holds, then \( t \) has type \( T^E \) and can take only forms from \( E \), where the constraint \( C \) hold. \( C_p \) is an input to the judgment called the path constraint, which must hold for this part of the program to have been reached. The set expression \( E \) and constraint \( C \) are outputs of the judgment, synthesized by traversing the expression. We need an external solver for set constraints to find a value for each variable \( V \) that satisfies \( C \). This is precisely what we define in Sec. 3.

The analysis supports higher-order functions, and it is polyvariant: refined types use polymorphism, so that precise analysis can be performed at each instantiation site. A variant of Damas-Milner style inference with let-generalization is used to generate these refined types. Moreover, the analysis is push-button: no additional input need be provided by the programmer. It is a sound but conservative analysis: it accounts for all possible values an expression may take, but declare some matches unsafe when they will not actually crash. The lack of polymorphic recursion is a source of imprecision, but a necessary one for preserving termination without requiring annotations from the programmer.

We generate two sorts of constraints. First, we constrain what values expressions could possibly take. For example, if we apply a constructor \( K_D(T) \), and we know the possible forms \( T \) for \( T \), then in any context, this expressions can only ever evaluate to values in the set \( K(T) \). Second, we generate safety constraints, which must hold to ensure that the program encounters no runtime errors. Specifically, we generate a constraint that when we match on a term \( t \), all of its possible values are covered by the left-hand side of one of the branches.
Variables  Our analysis rule \textsc{AVAR} for variables looks up a scheme from \( \Gamma \). However, typing schemes now quantify over type and set variables, and carry a constraint along with the type. We the take instantiation of type variables as given, since we know the underlying type of each expression. Each set variable is instantiated with a fresh variable. We then give \( x \) the type from the scheme, with the constraint that the instantiated version of the scheme’s constraint must hold if this piece of code is reachable (i.e. if the path condition is satisfiable).

Functions and Applications  The analysis rule \textsc{ALAM} for functions is straightforward. We generate a fresh set variable with which to annotate the argument type in the environment, and check the body in this extended environment. Since functions are not algebraic datatypes and cannot be matched upon, we emit \( \top \) as a trivial set of possible forms for the function itself.

We know nothing about the forms that the parameter-type annotation \( V \) may take, since it depends entirely on what concrete argument is given when the function is applied. However, when checking the body, we may encounter a pattern match that constrains what values \( V \) may take without risking runtime failure. So our analysis may emit safety constraints involving \( V \), but it will not constrain it otherwise. Generally, \( (T_1^{E_1} \rightarrow T_2^{E_2}) \) means that the function can safely accept any expression matching \( E_1 \), and may return values matching \( E_2 \).

Applications are analyzed using \textsc{AAPP}. Annotations and constraints for the function and argument are both generated, and we emit a constraint equating the argument’s annotated type with its codomain, under the assumption that the path condition holds and this function call is actually reachable. The meta-function \texttt{equate} traverses the structure of the argument and function codomain type, constraining that parallel annotations are equal. This traversal is possible because the underlying type system guarantees that the function codomain and argument have identical underlying types.

Constructor  As we mentioned above, applying a constructor to arguments can only produce a value that is that constructor wrapped around its argument’s values. The rule \textsc{ACTOR} for a constructor \( K \) infers annotations and constraints for each argument, then emits those constraints and applies \( K \) to those annotations.

Pattern matching  It is not surprising that in a pattern match analysis, the interesting details are found in the case for pattern match expressions. The rule \textsc{AMAT} begins by inferring the constraint and annotation for the discriminate \( t \).

We then iterate through each branch, performing two tasks. First, for each pattern \( P_i \) that is the left-hand side of a case, we generate the environment which binds the pattern variables to the correct types and annotations. As we show in Fig.~4, this is done for constructors by taking the \( i \)th projection of the constructor’s annotation. For \( P_1 \), the type of the whole pattern is \( E \), the annotation for \( t \). However, because of the first-match semantics of pattern matching, we know that if an expression matches \( P_1 \), then it does not match any of \( P_1 \ldots P_{i-1} \). So for
\[
\Gamma \vdash \Delta \vdash t : T_E \quad \text{(Pattern Match Analysis)}
\]

\[
\text{AVar} \quad \Gamma(x) = \forall X, \Gamma \vdash T_E \quad \forall \Gamma^V \text{ fresh}
\]

\[
\Gamma \vdash \Delta \vdash C_p \vdash x : [V^V/X]\left[[P^V/X]\Gamma^V\right] (P^V) \land (C_p \implies [V^V/X])
\]

\[
\text{ALAM} \quad V_i \text{ fresh} \quad x : T_i^V, \Gamma \vdash \Delta \vdash t : T_2^E \land C
\]

\[
\Gamma \vdash \Delta \vdash C_p \vdash \lambda x.t : (T_1^V \rightarrow T_2^E)^x \land C
\]

\[
\text{AAP} \quad \Gamma \vdash \Delta \vdash C_p \vdash t_1 : (T_1^E_1 \rightarrow T_2^E_2) \land C_1 \quad \Gamma \vdash \Delta \vdash C_p \vdash t_2 : T_2^E_1 \land C_2
\]

\[
\Gamma \vdash \Delta \vdash C_p \vdash t_1 \land t_2 : T_2^E_2 \land C_1 \land C_2 \land (C_p \implies \text{equate}(T_1^E_1, T_1^E_2))
\]

\[
\text{ACTOR} \quad K(T) \in \Delta(D) \quad \Gamma \vdash \Delta \vdash C_p \vdash t : T_E
\]

\[
\text{AMAT} \quad V \text{ fresh} \quad \Gamma \vdash \Delta \vdash C_p \vdash t : T_E \land C_{\text{discr}}
\]

\[
\Gamma \vdash \Delta \vdash C_p \vdash t_1 : T_E \mid \Gamma_1 \quad \ldots \quad \Gamma \vdash \Delta \vdash C_p \vdash t_n : T_E \mid \Gamma_n
\]

\[
C_p := (E \subseteq \mathcal{P}[P_1]) \ldots C_p := (E \subseteq \neg \mathcal{P}[P_1] \land \neg \mathcal{P}[P_{n-1}] \land \mathcal{P}[P_1])
\]

\[
\Gamma_1 \vdash \Delta \vdash C_p \land t_1 : T_1^E \quad \ldots \quad \Gamma_n \vdash \Delta \vdash C_p \land t_1 : T_n^E
\]

\[
C_1 := (E \cap \mathcal{P}[P_1] \neq \emptyset) \ldots C_n := (E \cap \neg \mathcal{P}[P_1] \land \neg \mathcal{P}[P_{n-1}] \land \mathcal{P}[P_1] \neq \emptyset)
\]

\[
C_{\text{result}} = (C_1 \implies E_1' \subseteq V) \land \ldots \land (C_n \implies E_n' \subseteq V)
\]

\[
C_{\text{may be}} := (C_p \implies (E \subseteq \mathcal{P}[P_1] \land \ldots \land \mathcal{P}[P_n]))
\]

\[
T_1^V := \text{freshen}(T_1) \quad x : T_1^V, \Gamma \vdash \Delta \vdash C_p \vdash t_1 : T_1^E \land C_1
\]

\[
C_e := \text{equate}(T_1^V, T_1^E)
\]

\[
\mathcal{V} = (\mathcal{FV}(E) \cup \mathcal{FV}(C_1)) \setminus (\mathcal{FV}(\Gamma) \cup \mathcal{FV}(C_p)) \quad C_e \land C_1 \text{ satisfiable}
\]

\[
\text{ALET} \quad x : \forall X \vdash \Gamma, C_e \land C_1 \implies T_1^E, \Gamma \vdash \Delta \vdash C_p \vdash t_2 : T_3 \land C_2
\]

\[
\Gamma \vdash \Delta \vdash t \text{ with } \{P \Rightarrow t_1'; \ldots; P_n \Rightarrow t_n'; \} : T_V
\]

\[
\text{ALET} \quad \Gamma \vdash \Delta \vdash C_p \vdash \begin{array}{l}
\text{let } x = t_1 \text{ in } t_2 : T_2 \land C_2
\end{array}
\]

\[
\Gamma \vdash \Delta \vdash P : T_E \mid \Gamma' \quad \text{(Analysis pattern environments. } P, T_E \text{ are input, } \Gamma' \text{ is output)}
\]

\[
\text{VAR} \quad \Gamma \vdash \Delta \vdash x : T_E((x : T_E), \Gamma)
\]

\[
K(T_1, \ldots, T_n) \in \Delta(D)
\]

\[
\text{Ctor} \quad \Gamma \vdash \Delta \vdash P : T_E^{-1}(E) ; \Gamma_1 \ldots \Gamma_n \vdash \Gamma \vdash \Delta \vdash P : T_E^{-n}(E) ; \Gamma_n'
\]

\[
\mathcal{P}[P] = E \quad \text{(Set expressions for patterns)}
\]

\[
\mathcal{P}[x] = T \quad \mathcal{P}[K(T)] = K(\mathcal{P}[T])
\]

Fig. 4. Pattern Match Analysis
each $P_i$, we extend the environment with annotations obtained by intersecting $E$ with the negation of all previous patterns. The translation $P[P]$ gives the set expression containing exactly the terms that match $P$.

Having obtained the extended environment for each branch, we perform our second task: we check each right-hand-side in the new environment, obtaining an annotation. Additionally, when checking the right-hand sides of branches, we add a conjunct to our path constraint which is the condition that must hold for this branch to have been taken. This ensures that safety constraints for the branch are only enforced when the branch can actually be taken.

To determine the annotation for the entire expression, we could naively take the union of the annotations for each branch. However, we can be more precise than this. Since we have an annotation for the input value, we can use that annotation to determine which branches could possibly be taken. We generate a fresh variable for the return annotation, and constrain that it contains the result of each branch, provided that it is possible we actually took that branch. This uses both negative set constraints and implication, justifying the need for a solver that supports these features.

Finally, we emit a safety constraint, saying that if it is possible to reach this part of the program (that is, if $C_p$ holds) then the inputs to the match must be contained within the values actually matched.

**Let expressions** Our last rule deals with the generalization of types into type schemes. This rule essentially performs Damas-Milner style inference, but for the annotations, rather than the types. In the rule $\text{ALET}$, when defining $x = t_1$, we check $t_1$ in a context extended with its type variables, and a monomorphic version of its own type. The metafunction $\text{freshen}$ takes the underlying type for $t_1$ and adds fresh annotation variables across the entire type. This allows for monomorphic recursion. The metafunction $\text{equate}$ generates the constraint that equates the freshly generated variables on $T'$ to the corresponding annotations on $T_1$ obtained when checking $t_1$. Again this traversal is possible because the underlying types must be identical. Once we have a constraint for the definition, we check that its constraint is in fact satisfiable, ensuring that none of the safety constraints are violated. In our implementation, this is where the call to the external solver is made.

To generate a type scheme for our definition, we generalize over all free variables in the inferred annotation and constraint, except those that are free in $\Gamma$ or $C_p$. Finally, we check the body of the let-expression in a context extended with the new variable and type scheme. Because let expressions are where constraints are actually checked, we assume that all top-level definitions of a program are wrapped in let -declarations. A program is well well formed if it can be typed with initial environment $\cdot$ and path constraint $T$.

**Example: Safety Constraints** To illustrate our analysis, we return to the $\text{Ngon}$ code from Sec. 1. We assume that all $\text{Double}$ terms are given annotation $\top$. Then, the $\text{simpleArea}$ function would be given the following annotated type
scheme:
\[
\forall V_1, V_2 . \quad V_1 \subseteq (\text{Square}(\top) \cup \text{Circle}(\top)) \\
\wedge (V_1 \cap \text{Square}(\top) \neq \bot \implies \top \subseteq V_2) \\
\wedge (V_1 \cap \text{Circle}(\top) \cap \neg \text{Square}(\top) \neq \bot \implies \top \subseteq V_2) \\
\Rightarrow \text{Ngon}^{V_1} \rightarrow \text{Double}^{V_2}
\]

This is satisfiable whenever \( V_1 \subseteq (\text{Square}(\top) \cup \text{Circle}(\top)) \), since we can set \( V_2 \) to \( \top \). When we call \text{simpleArea} from \text{area}, we are in the branch after the \text{Ngon} case has been checked. The scheme for \text{simpleArea} is instantiated with the path constraint \( V_4 \subseteq \top \cap \neg (\text{Ngon}(\top)) \), where \( V_4 \) is the annotation for \text{shape}, because it is called after we have a failed match with \text{Ngon sides}.

Suppose we instantiate \( V_1, V_2 \) with fresh \( V'_1, V'_2 \). The call to \text{simpleArea} creates a constraint that \( V'_4 = V'_2 \). Taking this equality into account, the safety constraint is instantiated to \( V'_4 \subseteq \top \cap \neg (\text{Ngon}(\top)) \implies V'_4 \subseteq (\text{Square}(\top) \cup \text{Circle}(\top)) \). This is satisfiable for any value of \text{shape}, so at every call to \text{area} the analysis sees that the safety constraint is satisfied.

**Example: Precision on results of matching** To illustrate the precision of our analysis for the results of pattern matching, we turn to a specialized version of the classic \text{map} function:

\[
\text{intMap} : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{List} \text{Int} \rightarrow \text{List} \text{Int} \\
\text{intMap} \ f \ \text{l} = \text{case l of} \\
\quad \text{Nil} \rightarrow \text{Nil} \\
\quad \text{Cons} \ h \ t \rightarrow \text{Cons} \ (f \ h) \ (\text{intMap} \ f \ t)
\]

Suppose we have concrete arguments \( f : (\text{Int}^{V_{11}} \rightarrow \text{Int}^{V_{12}})^{V_1} \) and \( l : (\text{List} \text{Int})^{V_2} \). The safety constraint for the match is that \( V_2 \subseteq \text{Nil} \cup \text{Cons}(\top, \top) \), which is always satisfiable since the match is exhaustive. The result of the case expression is given a fresh variable annotation \( V_3 \). From the first branch, we have the constraint that \( V_2 \cap \text{Nil} \neq \bot \implies \text{Nil} \subseteq V_3 \).

The analysis is more interesting for the second branch. The bound pattern variables \( h \) and \( t \) are given annotations \( \text{Cons}^{-1}(V_2) \) and \( \text{Cons}^{-2}(V_2) \) respectively, since they are the first and second arguments to \text{Cons}. Because our recursion is monomorphic, the recursive call \text{intMap} \ f \ t \ generates the trivial constraint \( (V_2 \cap \neg \text{Nil} \cap \text{Cons}(\top, \top)) \neq \bot \implies V_1 \subseteq V_1 \), and the more interesting constraint \( (V_2 \cap \neg \text{Nil} \cap \text{Cons}(\top, \top)) \neq \bot \implies \text{Cons}^{-2}(V_2) \subseteq V_2 \). This second constraint may seem odd, but it essentially means that without polymorphic recursion, our program’s pattern matches must account for any length of list. This is where having set constraints is extremely useful: if we were to use some sort of symbolic execution to try to determine a single logical value that \( l \) could take, then treating the recursive call monomorphically would create an impossible equation. But the set \( \text{Cons}(a, \text{Nil}), \text{Cons}(a, \text{Cons}(b, \text{Nil})), \ldots \) satisfies our set constraints, albeit in an imprecise way.

When checking the body, suppose that \( V_5 \) is the fresh variable ascribed to the return type of \text{intMap}. For the result of the second branch, we have the constraints
V ∈ V (Set variables)
E ∈ E (Set expressions)

\[
E ::= \bot \mid \top \mid \neg E_1 \mid f^a(E_1) \mid V
\]

\(V_2 \cap \neg \text{Nil} \cap \text{Cons} (\top, \top) \neq \bot \implies \text{Cons}(V_1, V_5) \subseteq V_3\). This essentially says that if the input to the function can be \text{Cons}, then so can the output, but if the input is always \text{Nil}, then this branch contributes nothing to the overall result. Finally, we have a constraint \(V_5 = V_3\), generated by the metafunction \text{equate}.

All together, we our result annotation \(V_3\) is constrained by:

\[
\begin{align*}
V_2 \cap \neg \text{Nil} \neq \bot & \implies \text{Nil} \subseteq V_3 \\
\land \quad V_2 \cap \neg \text{Nil} \cap \text{Cons} (\top, \top) \neq \bot & \implies \text{Cons}(V_1, V_3) \subseteq V_3
\end{align*}
\]

This captures the fact that \text{intMap} will return an empty result for empty input, and non-empty results for non-empty input.

### 3 Translating Set Constraints to SMT

While the above analysis provides a fine-grained way to determine which pattern matches are or are not safe, it depends entirely on the existence of an external solver to check the satisfiability of the resulting set constraints. We show that such a solver can be implemented by translating set constraints into an SMT formula, allowing for a simple, performant implementation.

#### 3.1 A Primer in Set Constraints

We begin by making precise the definition of the set constraint problem. Consider a set of (possibly 0-ary) functions \(\mathcal{F} = \{f_1^a, \ldots, f_n^a\}\), where each \(a \geq 0\) is the arity of the function \(f_i^a\). The \textit{Herbrand Universe} \(\mathcal{H}_\mathcal{F}\) is defined inductively: each \(f_0^0 \in \mathcal{F}\) is in \(\mathcal{H}_\mathcal{F}\), and if \(a > 0\) and \(h_1, \ldots, h_a\) are in \(\mathcal{H}_\mathcal{F}\), then \(f_i^a(h_1, \ldots, h_a)\) is in \(\mathcal{H}_\mathcal{F}\). (We write \(\mathcal{H}_\mathcal{F}\) as \(\mathcal{H}\) when the set \(\mathcal{F}\) is clear.) Each \(f_i^a\) is injective, but is otherwise uninterpreted, behaving as a constructor in a functional language.

Suppose we have some infinite set \(\mathcal{V}\) of variables. A \textit{set expression} is an expression generated by the grammar listed in Fig. 5. In addition to variables,
we can form set expressions using union, intersection, complement, and elementwise function application. We can use a Herbrand universe to give semantics to our expressions: given a substitution $\sigma : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{H})$, we can assign a meaning $\mathcal{H}[E]_\sigma \subseteq \mathcal{H}$ for an expression $E$ by mapping variables to their substitutions, and applying the corresponding set operations. The full semantics are given in Fig. 5.

A set constraint atom is a constraint of the form $E_1 \subseteq E_2$. These are also referred to as positive set constraints in previous work. A set constraint literal is either an atom or its negation $\neg(E_1 \subseteq E_2)$, which we write as $E_1 \nsubseteq E_2$. Constraints which contain negative literals are called negative set constraints.

An unrestricted set constraint is simply a boolean combination (i.e. using $\land$, $\lor$, and $\neg$) of set constraint atoms. For example, $(X \subseteq Y \implies Y \subseteq X) \land (Y \nsubseteq Z)$ is an unrestricted set constraint. In our examples, we will use other boolean operations such as $\implies$ and $\iff$, noting that they can be decomposed using $\land$, $\lor$, and $\neg$. Similarly, we use $E_1 = E_2$ as a shorthand for $E_1 \subseteq E_2 \land E_2 \subseteq E_1$.

Given an set constraint $C$, the satisfiability problem is to determine whether there exists a substitution $\sigma : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{H})$ such that, if each atom $E_1 \subseteq E_2$ in $C$ is replaced by the truth-value of $\mathcal{H}[E_1]_\sigma \subseteq \mathcal{H}[E_2]_\sigma$, then the resulting boolean expression is true. Since solving for arbitrary boolean combinations of set constraints is difficult, we focus on a more restricted version of the problem. The conjunctive set constraint problem for a sequence of literals $\overline{L}$ is to find a variable assignment that causes $\bigwedge \overline{L}$ to be true. We explain how to extend our approach to arbitrary boolean combinations in Sec. 3.7.

One can intuitively see that the Herbrand universe $\mathcal{H}$ closely matches the set of terms that can be formed from a collection of algebraic datatypes, and that allowing negative constraints and arbitrary boolean expressions satisfies the desiderata for our pattern match analysis.

### 3.2 Projection

Many analysis problems rely on a notion of projection. For a set expression $E$, we denote the $j$th projection of $E$ for function $f_i^a$ by $f_i^{-1}(E)$. For a substitution $\sigma$, we have $\mathcal{H}[f_i^{-1}(E)]_\sigma = \{ h_j \mid f_i^a(h_1, \ldots, h_j, \ldots h_n) \in E \}$.

While we don’t explicitly include projections in our grammar for set expressions, we can easily express them using boolean formulae. Given some constraint $C[f_i^{-1}(E)]$, we can replace this with:

$$
C[X_j] \land (E \cap f_i^a(\top, \ldots, \top)) = f_i^a(X_1, \ldots, X_j, \ldots X_n) \land (E = \bot \iff X_j = \bot)
$$

where each $X_k$ is a fresh variable. Intuitively, the first condition specifies that our variable holds the $j$th component of every $f(h)$ in $E$. The second condition is necessary because $f_i^a(X_1, \ldots, X_j, \ldots X_n) = \bot$ if any $X_k$ is empty, so any value of $X_j$ vacuously satisfies $E' = f_i^a(X_1, \ldots, X_j, \ldots X_n)$ if $E'$ and some $X_i$ are empty.

We now describe how to solve set constraints using SMT. The key is that any conjunction of set constraint literals can be translated into a formula in first-order monadic logic, for which satisfiability is decidable. We translate the search
for a satisfying assignment into a search for a solution to an SMT problem over UF, the theory of booleans, uninterpreted functions and first-order quantification. We gradually build up our translation, first translating set constraints into monadic logic, then translating monadic logic into SMT, then adding optimizations for efficiency. The complete translation is given in Sec. 3.6

3.3 Set Constraints and Monadic Logic

Monadic first order logic, sometimes referred to as the monadic class, consists of formulae containing only unary predicates, boolean connectives, and constants. Bachmair et al. [7] show that any conjunction $\bigwedge \mathcal{T}$ of positive set constraint atoms can be converted into an equisatisfiable monadic formula, and and Charatonik and Pacholski [11] extend this to negative constraints with equality. We summarize their procedure here, with a full definition in Fig. 6. For each sub-expression $E$ of $\bigwedge \mathcal{T}$, we create a predicate $P_E(x)$, which denotes whether an element $x$ is contained in $E$. Along with this, the formula $\mathcal{E}[E]$ gives the statement that must hold for $P_E$ to respect the semantics of set expressions. This is similar to the Tsieten transformations used to efficiently convert arbitrary formulae to a normal form [35]. Given $P_E(x)$ for each $E$, we can represent the constraint $E_1 \subseteq E_2$ as $\forall x. (P_{E_1}(x) \implies P_{E_2}(x))$. Similarly, $E_1 \not\subseteq E_2$ corresponds to $\exists x. (P_{E_1}(x) \land \neg P_{E_2}(x))$.  

The key utility of having a monadic formula is the finite model property [1, 27]:

**Theorem 1.** Let $\mathcal{T}$ be a theory in monadic first-order logic with $N$ predicates. Then, for any sentence $S$ in $\mathcal{T}$, there exists a model satisfying $S$ if and only if there exists a model satisfying $S$ with a finite domain of size at most $2^N$.

The intuition behind this is that if there exists a model $\mathcal{D}$ satisfying $S$, then then $S$ we can combine all elements in $\mathcal{D}$ that have identical truth-values for each predicate. This is enough to naively solve set constraints: we convert them into formulae monadic logic, then search the space of all models of size up to $2^N$ for one that satisfies the monadic formulae. However, this is terribly inefficient, and disregards much of the information we have from the set constraints.

3.4 Monadic Logic in SMT

To understand how to translate monadic logic into SMT, we first look at what exactly a model for a monadic theory is. Suppose $\mathbb{E} = \{\mathbf{T}, \mathbf{F}\}$ is the set of booleans. For brevity, we refer to elements of this set as *bits*, and say a bit is set if it is $\mathbf{T}$. For our purposes, a model consists of a set $D$, called the *domain*, along with *interpretations* $I_P : D \to \mathbb{E}$ for each predicate $P$ and $f^a_i : D \to D$

---

1 The original monadic translation used “reverse-skolemization” to transform constants and functions into existentially-bound variables. However, SMT can easily solve problems with uninterpreted functions and constants, so we do not use this step.
Set Constraints, Pattern Match Analysis, and SMT

\[ \mathcal{M} \in \mathcal{M} \text{ (Monadic formulae)} \]

\[ \mathcal{E}[E] = M \text{ (Predicates for set expressions)} \]

\[
\begin{align*}
\mathcal{E}[-] &= \forall x. \neg P_a(x) \\
\mathcal{E}[\top] &= \forall x. P_\top(x) \\
\mathcal{E}[X] &= T \\
\mathcal{E}[E_1 \land E_2] &= \forall x. P_{E_1 \land E_2}(x) \iff (P_{E_1}(x) \land P_{E_2}(x)) \\
\mathcal{E}[E_1 \lor E_2] &= \forall x. P_{E_1 \lor E_2}(x) \iff (P_{E_1}(x) \lor P_{E_2}(x)) \\
\mathcal{E}[\neg E] &= \forall x. P_{\neg E}(x) \iff \neg P_E(x) \\
\mathcal{E}[f^n(E_1, \ldots, E_n)] &= (\forall x_1 \ldots x_n. P_{f^n(E_1, \ldots, E_n)}(f^n_1(x_1, \ldots, x_n)) \iff P_{E_1}(x_1) \land \cdots \land P_{E_n}(x_n)) \\
&\quad \quad (\bigwedge_{g^n_i \neq f^n_j} \forall x_1 \ldots x_n. P_{f^n(E_1, \ldots, E_n)}(g^n_i(x_1, \ldots, x_n)) \iff \bot)
\end{align*}
\]

\[
\begin{align*}
\mathcal{L}[L] &= M \text{ (Predicates for literals)} \\
\mathcal{L}[\bigwedge \bar{T}] &= M \text{ (Predicates for conjunction)} \\
\mathcal{L}[E_1 \subseteq E_2] &= \forall x. P_{E_1}(x) \implies P_{E_2}(x) \\
\mathcal{L}[E_1 \nsubseteq E_2] &= \exists y. P_{E_1}(y) \land \neg P_{E_2}(y)
\end{align*}
\]

\[ \text{Fig. 6. Translating Set Constraints to Monadic Logic} \]

for each function, which define the value of \( P(x) \) and \( f(x_1, \ldots, x_n) \) for each \( x, x_1, \ldots, x_n \in D \). A naive search for a satisfying model could guess \( M \leq 2^N \), set \( D = \{1 \ldots M\} \), and iterate through all possible truth assignments for each \( I_P \), and all possible mappings for each \( f^n \), searching for one that satisfies the formulae in the theory.

However, we can greatly speed up this search if we instead impose structure on \( D \). Specifically, if we have predicates \( P_1 \ldots P_N \), we take \( D \subseteq \mathbb{B}^N \): each element of our domain is a boolean sequence with a bit for each sub-expression \( E \). The idea is that each element of \( \mathbb{B}^N \) models a possible equivalence class of predicate truth-values. For \( b \in D \), we want \( b_i \) to be \( T \) when \( P_{E_i}(b) \) holds. This means that our maps \( I_P \) are already fixed: \( I_{P_{E_i}}(b) = b_i \) i.e. the \( i \)th bit of sequence \( b \).

However, with this interpretation, \( \mathbb{B}^N \) is too large to be our domain. Suppose we have formulae \( E_1 \) and \( E_j \) where \( E_j = \neg E_i \). Then there are sequences in \( \mathbb{B}^n \) with both bits \( i \) and \( j \) set to \( T \). To respect the consistency of our logic, we need \( D \) to be a subset of \( \mathbb{B}^N \) that eliminates such inconsistent elements.

Suppose that we have a function \( \text{inDomain} : \mathbb{B}^N \to \mathbb{B} \), which determines whether a bit-sequence is in the domain of a potential model. If \( \mathcal{L}[[\bigwedge \bar{T}]] \) contains the formula \( \forall x_1 \in D \ldots \forall x_n \in D. \Phi[x_1 \ldots x_n] \), for some \( \Phi \), we can instead write:

\[ \forall b_1 \in \mathbb{B}^N \ldots \forall b_n \in \mathbb{B}^N. \text{inDomain}(b_1) \land \cdots \land \text{inDomain}(b_n) \implies \Phi[b_1 \ldots b_n] \]

That is, our domain can only contain values that respect the semantics of set expressions. Similarly, if \( \mathcal{L}[[\bigwedge \bar{T}]] \) contains \( \exists x. \Phi[x] \), we can write \( \exists b \in \mathbb{B}^N. \text{inDomain}(b) \land \Phi[b] \). Since all functions in a model are implicitly closed over the domain, we also
specify that \( \forall \vec{b} \in (\mathbb{B}^n)^a, \text{inDomain}(\vec{b}) \Rightarrow \text{inDomain}(f^a_i(\vec{b})) \). This ensures that our formulae over boolean sequences are equivalent to the original formulae.

This is enough to express \( \mathcal{L}[\bigwedge \overline{\mathcal{L}}] \) as an SMT problem. We assert that a function \( \text{inDomain} : \mathbb{B}^N \to \mathbb{B} \) exists, and that there is a function \( f^a_i : (\mathbb{B}^N)^a \to \mathbb{B}^N \) for each function in our Herbrand universe. We modify each formula in \( \mathcal{L}[\bigwedge \overline{\mathcal{L}}] \) to constrain a boolean sequences variable \( b_i \in \mathbb{B}^n \) in place of each variable \( x_i \in D \) as described above. We add \( \text{inDomain} \) qualifiers to existentially and universally quantified formulae, and replace each \( P_{E_i}(x_j) \) with the \( i \)th bit of \( b_j \). We add a constraint asserting that each \( f^a_i \) is closed over the values satisfying \( \text{inDomain} \). The SMT solver searches for values for all existential variables, functions, and \( \text{inDomain} \) that satisfy this formula.

### 3.5 Reducing the Search Space

While this translation corresponds nicely to the monadic translation, it has more unknowns than are needed. Specifically, \( \text{inDomain} \) will always reject boolean sequences that violate the constraints of each \( E_i[] \). For example, the bit for \( P_{E_1 \cap E_2}(b) \) in \( b \) must always be exactly \( P_{E_1}(b) \land P_{E_2}(b) \). In fact, for each form except function applications and set variables, the value of a bit for an expression can be recursively determined by values of bits for its immediate subexpressions (Fig. 7). This means that our boolean sequences need only contain slots for expressions of the form \( X \) or \( f^a_i(E_1, \ldots, E_a) \), shrinking the problem’s search space.

What’s more, we now only need to include the constraints from \( \mathcal{E}[] \) for expressions of the form \( X \) or \( f^a_i(E_1, \ldots, E_a) \), since the other constraints hold by definition given our definitions of each \( P_E \).

Similarly, our constraints restrict the freedom we have in choosing \( f^a_i \). Specifically, we know that \( P_{f^a_i(E_1, \ldots, E_a)}(f^a_i(b_1, \ldots, b_a)) \) should hold if and only if \( P_{E_i}(b_i) \) holds for each \( i \leq i \leq a \). Similarly, we know that \( P_{f^a_i(E_1, \ldots, E_a)}(g^a_i(b_1, \ldots, b_a)) \) should always be \( \mathbf{F} \) when \( f \neq g \). So for each \( f^a_i \), it suffices to find a mapping from inputs \( b_1, \ldots, b_a \) to the value of \( P_X(f^a_i(b_1, \ldots, b_a)) \) for each variable \( X \). This again reduces the number of unknowns in the SMT problem.
3.6 The Complete Translation

We summarize our complete translation. Given a conjunction of literals $\bigwedge L$, let $X_1, \ldots, X_k, E_{k+1}, \ldots, E_N$ be the sequence of variable and function-application sub-expressions of $L$. We define $P_E(b)$ for each sub-expression $E$ of $L$ as in Fig. 7.

As unknowns, we have:

- a function inDomain : $\mathbb{B}^N \rightarrow \mathbb{B}$;
- for each negative literal $E_i \not\subseteq E'_i$, an existential variable $y_i \in \mathbb{B}^N$;
- for each function $f^a_i$ and each variable $X \in L$, a function $f^a_{iX} : (\mathbb{B}^N)^a \rightarrow \mathbb{B}$, which takes a sequence of $N$ bits and computes the value of the bit for $P_X$ in the result.

We define the following known functions:

- For each $f^a_i$ and each sub-expression of the form $f^a_i(E_1, \ldots, E_n)$, a function $f^a_{i\text{I}SMT}(E_1, \ldots, E_n) : (\mathbb{B}^N)^a \rightarrow \mathbb{B}$ where $f^a_{i\text{I}SMT}(E_1, \ldots, E_n)(b_1, \ldots, b_a) = T$ if and only if $P_{E_1}(b_1) \land \ldots \land P_{E_n}(b_a)$;
- For each $f^a_i$ and each sub-expression of the form $g^a_j(E_1, \ldots, E'_d)$, where $f \neq g$, a function $f^a_ig^a_{jSMT}(E_1, \ldots, E'_d)$ : $(\mathbb{B}^N)^a \rightarrow \mathbb{B}$ that always returns $F$;
- For each $f^a_i$, a function $f^a_{i\text{I}SMT} : (\mathbb{B}^N)^a \rightarrow \mathbb{B}^N$, where $f^a_{i\text{I}SMT}(b_1, \ldots, b_a)$ is the sequence:
  $f^a_{iX_1}(b_1, \ldots, b_a) \ldots f^a_{iX_k}(b_1, \ldots, b_a)f^a_{iE_{k+1}}(b_1, \ldots, b_a) \ldots f^a_{iE_N}(b_1, \ldots, b_a)$

We assert that the following hold:

- for each negative constraint $E_i \not\subseteq E'_i$ with corresponding existential variable $y_i$, that inDomain$(y_i) \land P_{E_i}(y_i) \land \neg P_{E'_i}(y_i)$ holds;
- for each positive constraint $E_i \subseteq E'_i$, that
  $\forall x \in \mathbb{B}^N$. (inDomain$(x) \land P_{E_i}(x)) \implies P_{E'_i}(x)$ holds;
- For each function $f^a_i$, that:
  $\forall x_1 \ldots x_a$. (\bigwedge_{j=1}^a \text{inDomain}(x_j)) \implies \text{inDomain}(f^a_{i\text{I}SMT}(x_1, \ldots, x_a))$ holds.

A solution to these assertions exists iff the initial set constraint is satisfiable.

3.7 Arbitrary Boolean Combinations

As we described earlier, we would like to allow arbitrary boolean combinations of set constraints, both to enrich our pattern match analysis and to allow us to use projections. Doing so is quite simple: for each atom $E_i \subseteq E'_i$ in a constraint $C$, we introduce a boolean variable $\ell_i$, which the SMT solver must guess. We modify our translation so that $L[E_i \subseteq E'_i] = \forall x. \ell_i \implies (P_{E_i}(x) \implies P_{E'_i}(x))$ and $L[E_i \not\subseteq E'_i] = \ell_i \implies (\exists y. P_{E_i}(y) \land \neg P_{E'_i}(y))$. That is, $\ell_i \iff E_i \subseteq E'_i$.

Finally, we assert the formula that is $C$ where each occurrence of $E_i \subseteq E'_i$ is replaced by $\ell_i$ and $E_i \not\subseteq E'_i$ is replaced by $\neg \ell_i$. Thus, we force our SMT solver to guess a literal assignment for each atomic set constraint, and then determine if it can solve the conjunction of those literals. When $\ell_i$ is false, then $L[E_i \subseteq E'_i]$ will be vacuously true, with the opposite holding for negative constraints.
Table 1. Compilation Time (ms) of Exhaustiveness versus Pattern Match Analysis

| Library      | EX-TN | PMA-TN | EX-FP | PM-FP | EX-TP | PM-TP |
|--------------|-------|--------|-------|-------|-------|-------|
| elm-graph    | 50    | 168    | 45    | 178   | 44    | 173   |
| elm-intdict  | 42    | 115    | 38    | 5121* | 35    | 113   |
| elm-interval | 40    | 69     | 39    | 1217* | 36    | 1261  |

4 Implementation and Evaluation

To evaluate our translation, we implemented the analysis from Sec. 2 in a modified version of the Elm language compiler [12]. We compared the running time of the Elm compiler’s exhaustiveness checking to our pattern match analysis on a machine with 32GB of memory and an i7-3770 processor. For SMT we used Z3 4.8.5 [28] with mbqi and UFBV.

In order to make the analysis practical, we implemented several optimizations on top of our pattern match analysis. Trivially satisfiable constraints were removed, and set expressions were simplified where possible. When a match was exhaustive, its safety constraint was omitted, and since the non-safety constraints should always be satisfiable, calls to Z3 were only made when the list of safety constraints was non-empty. A union-find algorithm was used to identify combine variables constrained to be equal, and intermediate variables were merged. Since the constraint of an annotated scheme is copied at each instantiation, these ensured that the size of type annotations did not explode. For simplicity, annotated types were not carried across module boundaries; imported functions were assumed to accept any input and always have return annotation $\top$. Similarly, a conservative approximation was used in place of the full projections when determining pattern variables’ annotations.

We ran our tests on the Elm graph[19], intdict[20], and interval[9] libraries. Each of these initially contained safe partial matches, but were modified to return dummy values in unreachable code when Elm 0.19 was released.

The results of the evaluation are given in Table 1. Runs with the prefix EX used the exhaustiveness check of the original Elm compiler, while those marked PMA used our pattern-match analysis. We tested the compilers on three variants of each library, a true-negative (-TN) version in which all matches were exhaustive, a false-positive (-FP) version in which a match was non-exhaustive but safe, and a true-positive (-TP) version in which a required branch was missing and running the program would result in an error. Cases marked with an asterisk (*) are those which were rejected by the Elm compiler, but which our analysis marked as safe. Notably, the elm-graph library relied on an invariant too complex for our analysis to capture.

Our analysis is slower than exhaustiveness checking in each case. However, the pattern match analysis requires less than one second in the majority of cases, and in the worst case requires only six seconds. The slowdown was most prominent in the false-positive cases that our analysis marks as safe, where Z3 was not able to quickly disprove the satisfiability of the satisfiability of the
set constraints. Conversely, in the -TN cases where Z3 was not called, our analysis cause very little slowdown. Given the infrequency of partial matches, we feel this is acceptable performance for a tool integrated into a compiler.

5 Discussion

Future Work While our translation of set constraints to SMT attempts to minimize the search space, we have not investigated further optimizations of the SMT problem. The SMT was given relatively small problems. Few programs contain hundreds of constructors or pattern match cases. Nevertheless, more can be done to reduce the time spent in the SMT solver for larger problems. Solvers like CVC4 [8] are highly configurable with regards to their strategies for solving quantification. Fine tuning the configuration could decrease the times required to solve our problems without requiring a custom solver. Conversely, a solver specialized to quantified boolean arithmetic could yield faster results.

Likewise, type information could be used to speed up analysis. While we have modeled patterns using the entire Herbrand space, values of different data types reside in disjoint universes. Accounting for this could help partition one problem with many variables into several problems with few variables.

Related Work: Set Constraints The modern formulation of set constraints was established by Heintze and Jaffar [21]. Several independent proofs of decidability for systems with negative constraints were given, using a number-theoretic reduction [2, 34], tree automata [18], and monadic logic [11]. Charatonik and Podelski established the decidability of positive and negative constraints with projection [31]. The first tool aimed at a general, practical solver for set constraints was BANE [4], which used a system of rewrite rules to solve a restricted form of set constraints [5]. Banshee was a rewrite of BANE, improving performance with code generation and incremental analysis [24]. Neither of these implementations allow for negative constraints or unrestricted projections. Several survey papers give a more in-depth overview of set constraint history and research [3, 22, 30].

Related Work: Pattern Match Analysis Several pattern match analyses have been presented in previous work. Koot [25] presents a higher-order pattern match analysis as a type-and-effect system, using a presentation similar to ours. This work was extended by Koot and Hage [26], who present an analysis based on higher-order polymorphism. This improves the precision of the analysis, but suffers from the same problems as our regarding polymorphic recursion. All of these efforts use restricted versions of set constraints, and do not allow for unrestricted projection, negation, and boolean combinations of constraints.

Previous versions of type inference for pattern matching have utilized conditional constraints [6, 32, 33]. Castagna et al. [10] describe a similar system, albeit more focused on type-case than pattern matching. These systems are similar in
expressive power to the constraints that we used in our final implementation, but our underlying constraint logic is more powerful. There are restrictions on the where unions and intersections can appear in conditional constraints [6], and there is not full support for projections or negative constraints. In particular, negative constraints allow for analyses to specify that a function’s input set must not be empty, so that the type error can point to the function definition rather than the call-site, avoiding the lazy inference described by Pottier [33]. While these have not been integrated into our implementation, the flexibility of our constraint logic makes it easy to incorporate these and other future improvements.

Another related line of work is datasort refinements. [14–17]. As with our work, the goal of datasort refinements is to allow partial pattern matches while eliminating runtime failures. This is achieved by introducing refinements of each algebraic data type corresponding to its constructors, possibly with unions or intersections. Datasort refinements are presented as a type system, not as a standalone analysis, so their handling of polymorphism and recursive types is more precise than ours. However, checking programs with refined types requires at least some annotation from the programmer, where our analysis can check programs without requiring additional programmer input.

5.1 Conclusion

Prior to our work, unrestricted set constraints had been analyzed primarily in theory. With our translation, they can be solved in practice. SMT solvers are a key tool in modern verification, and they can now be used to solve set constraints. Moreover, our work shows how reasoning with unrestricted boolean combinations of constraints can make analysis more precise. We have shown that even \textsc{NEXPTIME}-completeness is not a complete barrier to the use of set constraints in practical verification.

Bibliography

1. Ackermann, W.: Solvable cases of the decision problem. Studies in logic and the foundations of mathematics, North-Holland Pub. Co. (1954)
2. Aiken, A., Kozen, D., Wimmers, E.: Decidability of systems of set constraints with negative constraints. Information and Computation 122(1), 30 – 44 (1995), \url{http://www.sciencedirect.com/science/article/pii/S089054018571139X}
3. Aiken, A.: Introduction to set constraint-based program analysis. Science of Computer Programming 35(2), 79 – 111 (1999), \url{http://www.sciencedirect.com/science/article/pii/S0167642399000076}
4. Aiken, A., Fähndrich, M., Foster, J.S., Su, Z.: A toolkit for constructing type-and constraint-based program analyses. In: Leroy, X., Ohori, A. (eds.) Types in Compilation. pp. 78–96. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
5. Aiken, A., Wimmers, E.L.: Type inclusion constraints and type inference. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. pp. 31–41. FPCA ’93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/165180.165188

6. Aiken, A., Wimmers, E.L., Lakshman, T.K.: Soft typing with conditional types. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 163–173. POPL ’94, ACM, New York, NY, USA (1994), http://doi.acm.org/10.1145/174675.177847

7. Bachmair, L., Ganzinger, H., Waldmann, U.: Set constraints are the monadic class. In: [1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science. pp. 75–83 (June 1993)

8. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi’c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV ’11). Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (Jul 2011), http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf, snowbird, Utah

9. Bell, R.K.: elm-interval. https://github.com/r-k-b/elm-interval/ (2019)

10. Castagna, G., Nguyen, K., Xu, Z., Abate, P.: Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 289–302. POPL ’15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2676726.2676991

11. Charatonik, W., Pacholski, L.: Negative set constraints with equality. In: Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science. pp. 128–136 (July 1994)

12. Czaplicki, E.: An introduction to elm (2019), https://guide.elm-lang.org/

13. Dumas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 207–212. POPL ’82, ACM, New York, NY, USA (1982), http://doi.acm.org/10.1145/582153.582176

14. Dunfield, J.: Refined typechecking with stardust. In: Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification. pp. 21–32. PLPV ’07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1292597.1292602

15. Dunfield, J., Pfenning, F.: Type assignment for intersections and unions in call-by-value languages. In: Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures and Joint European Conference on Theory and Practice of Software, pp. 250–266. FOSSACS’03/ETAPS’03, Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1754809.1754827

16. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming
17. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. pp. 268–277. PLDI ’91, ACM, New York, NY, USA (1991), http://doi.acm.org/10.1145/113445.113468
18. Gilleron, R., Tison, S., Tommasi, M.: Solving systems of set constraints with negated subset relationships. In: Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science. pp. 372–380 (Nov 1993)
19. Graf, S.: elm-graph. https://github.com/elm-community/graph/ (2019)
20. Graf, S.: elm-intdict. https://github.com/elm-community/intdict (2019)
21. Heintze, N., Jaffar, J.: A decision procedure for a class of set constraints. In: [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science. pp. 42–51 (June 1990)
22. Heintze, N., Jaffar, J.: Set constraints and set-based analysis. In: Borning, A. (ed.) Principles and Practice of Constraint Programming. pp. 281–298. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
23. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press (2018)
24. Kodumal, J., Aiken, A.: Banshee: A scalable constraint-based analysis tool kit. In: Hankin, C., Siveroni, I. (eds.) Static Analysis. pp. 218–234. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
25. Koot, R.: Higher-Order Pattern Match Analysis. Master’s thesis, Universiteit Utrecht, the Netherlands (2012)
26. Koot, R., Hage, J.: Type-based exception analysis for non-strict higher- order functional languages with imprecise exception semantics. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 127–138. PEPM ’15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2678015.2682542
27. Löwenheim, L.: Über möglichkeiten im relativkalkül. Mathematische Annalen 76(4), 447–470 (Dec 1915), https://doi.org/10.1007/BF01458217
28. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
29. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design. Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel). pp. 114–136. Springer-Verlag, Berlin, Heidelberg (1999), http://dl.acm.org/citation.cfm?id=646005.673740
30. Pacholski, L., Podelski, A.: Set constraints: A pearl in research on constraints, pp. 549–561. Springer Berlin Heidelberg, Berlin, Heidelberg (1997), https://doi.org/10.1007/3-540-61419-3_42
31. Pacholski, W.C.L.: Set constraints with projections. J. ACM 57(4), 23:1–23:37 (May 2010), http://doi.acm.org/10.1145/1734213.1734217
32. Palmer, Z., Menon, P.H., Rozenshtein, A., Smith, S.: Types for flexible objects. In: Garrigue, J. (ed.) Programming Languages and Systems. pp. 99–119. Springer International Publishing, Cham (2014)

33. Pottier, F.: A versatile constraint-based type inference system. Nordic J. of Computing 7(4), 312–347 (Dec 2000), http://dl.acm.org/citation.cfm?id=763845.763849

34. Stefansson, K.: Systems of set constraints with negative constraints are nextrime-complete. In: Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science. pp. 137–141 (Jul 1994)

35. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983), https://doi.org/10.1007/978-3-642-81955-1_28