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 NEXPTIME-hard. Moreover, while theoretical algorithms to solve arbitrary set constraints exist, they are either too complex to realistically 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 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 NEXPTIME-complete. While efficient solvers have been developed for restricted versions of the set constraint problem [4, 26], 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 SAT modulo theories (SMT). Although SMT requires exponential time in theory, solvers such as Z3 [31] 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 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:

- We devise a pattern match analysis for a strict functional language, 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 our translation and analysis, showing that they are usable for verification despite NEXPTIME-completeness (Sec. 4).

Motivation: Pattern Match Analysis

Our primary interest in set constraints is using them to devise a functional *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 [25] 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.

```
                        area :: Shape -> Double
                        area shape = case shape of
        data Shape =      NGon sides -> ...
          Square Double   _ -> simpleArea shape
          | Circle Double   where simpleArea sshape = case sshape of
          | NGon [Double]       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`

$x \in \text{ProgVariabe}, X \in \text{TypeVariable}, D \in \text{DataType}, K \in \text{DataConstructor}$

Terms

$$t ::= x \mid \lambda x.\, t \mid \texttt{match } t \texttt{ with } \{\overrightarrow{P \Rightarrow t;}\}$$

$$\mid t_1\ t_2 \mid K_D(\overrightarrow{t}) \mid \texttt{let } x = t_1 \texttt{ in } t_2$$

Patterns

$$P ::= x \mid K_D(\overrightarrow{P})$$

Underlying Types

$$\tau ::= X \mid D \mid \tau_1 \to \tau_2$$

Datatype environments

$$\Delta ::= \cdot \mid D = \overrightarrow{K(\overrightarrow{T})}, \Delta$$

Underlying Type Schemes

$$\sigma ::= \forall \overrightarrow{X}.\, \tau$$

Type Environments

$$\Gamma ::= \cdot \mid X, \Gamma \mid x : T, \Gamma$$

Fig. 1. λ_{Match}: syntax

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 $X \subseteq Nil \cup Cons(\top, Cons(\top, X))$.

- 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 wish to avoid false negatives by tracking 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 describes solving these constraints. Both are implemented and evaluated in Sec. 4.

2 A Set Constraint-based Pattern Match Analysis

Here, we describe an annotated type system 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 $\lambda_{\textbf{Match}}$ Syntax

We present λ_{Match}, a small, typed functional language, whose syntax we give in Fig. 1. Throughout, for a given metavariable $\mathcal{M}$ we write $\overrightarrow{\mathcal{M}}^i$ for a sequence of objects matching $\mathcal{M}$. We omit the positional index i when it is unneeded.

$\boxed{\Gamma \vdash t : \tau}$ (Expression typing)

$$\text{CTOR}\dfrac{\begin{array}{c}K(\overrightarrow{\tau}) \in \Delta(D)\\ \overline{\Gamma \vdash t : \tau}\end{array}}{\Gamma \vdash K_D(\overrightarrow{t}) : D} \qquad \text{LAM}\dfrac{x : \tau_1, \Gamma \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \text{APP}\dfrac{\begin{array}{c}\Gamma \vdash t_1 : \tau_1 \to \tau_2\\ \Gamma \vdash t_2 : \tau_1\end{array}}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\text{VAR}\dfrac{\Gamma(x) = \forall \overrightarrow{X}.\tau}{\Gamma \vdash x : \overrightarrow{[\tau'/X]}\tau} \qquad \text{MAT}\dfrac{\Gamma \vdash t : \tau \qquad \overrightarrow{\Gamma \vdash P : \tau | \Gamma'} \qquad \overrightarrow{\Gamma' \vdash t' : \tau'}}{\Gamma \vdash \texttt{match } t \texttt{ with } \{\overrightarrow{P \Rightarrow t';}\} : \tau'}$$

$$\text{LET}\dfrac{x : \tau_1, \overrightarrow{X}, \Gamma \vdash t_1 : \tau_1 \qquad x : \forall \overrightarrow{X}.\tau_1, \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : \tau_2}$$

$\boxed{\Gamma \vdash P : \tau | \Gamma'}$ (Pattern typing and binding generation)

$$\text{VAR}\dfrac{}{\Gamma \vdash x : \tau | (x : \tau), \Gamma} \qquad \text{CTOR}\dfrac{K(\overrightarrow{\tau}) \in \Delta(D) \qquad \overrightarrow{\Gamma \vdash P : \tau | \Gamma'}}{\Gamma \vdash K_D(\overrightarrow{P}) : D | \bigcup \overrightarrow{\Gamma'}}$$

Fig. 2. Underlying Typing for Expressions and Patterns

In addition to functions and applications, we have a form $K_D(\overrightarrow{t})$ which applies the data constructor K to the argument sequence $\overrightarrow{t}$ to make a term of type D. Conversely, the form $\texttt{match } t' \texttt{ with } \{\overrightarrow{P \Rightarrow t;}\}$ chooses the first branch $P_i \Rightarrow t_i$; 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. We use Haskell-style shadowing for non-linear patterns: e.g. (x, x) matches any pair, and binds the second element to x. We omit advanced matching features, such as guarded matches, since these can be desugared into nested simple matches. We use type environments Γ store free type variables and types for program variables. We assume a fixed datatype environment Δ that stores 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 underlying type system is in the style of Damas and Milner [13], where monomorphic types are separated from polymorphic type schemes. The declarative typing rules for the underlying system are standard (Fig. 2). We do not check the exhaustiveness of matches, as this overly-conservative check is precisely what we aim to replace. The analysis we present below operates on these underlying typing derivations, so each expression has a known underlying type.

$V \quad \in \quad$ SETVARIABLE

Set constraints

$C \quad ::= \quad E_1 \subseteq E_2 \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C$

Annotated Types

$T \quad ::= \quad X^E \mid D^E \mid (T_1 \to T_2)^E$

Set expressions

$E \quad ::= \quad V \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E$

$\qquad \mid K_D(\overrightarrow{E}) \mid K_D^{-i}(E) \mid \top \mid \bot$

Annotated Schemes

$S \quad ::= \quad \forall \overrightarrow{X}, \overrightarrow{V}. \, C \Rightarrow T$

Fig. 3. λ_{MATCH}: annotations

2.3   Annotated Types

For our analysis, we annotate types with *set expressions* (Fig. 3). We define
their semantics formally in Sec. 3, but intuitively, they represent possible shapes
that the value of an expression might have in some context. We have variables,
along with intersection, union and negation, and $\top$ and $\bot$ representing the sets
of all and no values respectively. The form $K_D(E_1 \ldots E_a)$ denotes applying the
arity-a constructor K of datatype D to each combination of values from the sets
denoted by $\overrightarrow{E}$. Conversely, $K_D^{-i}(E)$ denotes the *i*th *projection* of K: it takes the
*i*th argument of each value constructed using K from the set denoted by E.

   Set constraints then specify the inclusion relationships between those sets.
These are boolean combinations of atomic constraints of the form $E_1 \subseteq E_2$. Our
analysis uses these in *annotated type schemes*, to constrain which annotations a
polymorphic type accepts as instantiations. The idea is similar to Haskell's type-
class constraints, and we adopt a similar notation. 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. Annotated types T^E replace underlying types τ
in our rules, and our analysis emits constraints on E that dictate its value. We
note that boolean operations such as $\implies$ and $\iff$, can be decomposed into
$\wedge$, $\vee$, and $\neg$. Similarly, we use $E_1 = E_2$ as a shorthand for $E_1 \subseteq E_2 \wedge E_2 \subseteq E_1$,
and **T** and **F** as shorthands for $\bot \subseteq \top$ and $\top \subseteq \bot$ respectively.

2.4   The Analysis

We present our pattern match analysis in Fig. 4. The analysis is phrased as an
annotated type system in the style of Nielson and Nielson [32]. The judgment
$\Gamma | C_p \vdash t : T^E \mid C$ says that, under context Γ, if C_p holds, then t has the
underlying type of T and can take only forms from E, where the constraint C
holds. 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. We
write the conversion between patterns and set-expressions as $\mathcal{P}[\![P]\!]$.

   The analysis supports higher-order functions, and it is *polyvariant*: refined
types use polymorphism, so that precise analysis can be performed at each in-
stantiation site. A variant of Damas-Milner style inference with let-generalization

$\boxed{\Gamma|C_p \vdash t : T_E \mid C}$ (Pattern Match Analysis)

$$\text{AVar}\frac{\Gamma(x) = \forall \overrightarrow{X}, \overrightarrow{V}. C \Rightarrow T^E \qquad \overrightarrow{V'} \text{ fresh}}{\Gamma|C_p \vdash x : [\overrightarrow{V'/V}][\overrightarrow{\tau'/X}](T^E) \mid (C_p \implies [\overrightarrow{V'/V}]C)}$$

$$\text{AApp}\frac{\Gamma|C_p \vdash t_1 : (T_1^{E_1} \to T_2^{E_2})^{E_3} \mid C_1 \qquad \Gamma|C_p \vdash t_2 : T_1^{E_1'} \mid C_2}{\Gamma|C_p \vdash t_1\ t_2 : T_2^{E_2} \mid C_1 \wedge C_2 \wedge (C_p \implies T_1^{E_1} \equiv T_1^{E_1'})}$$

$$\text{ACtor}\frac{K(\overrightarrow{T}) \in \Delta(D) \qquad \overrightarrow{\Gamma|C_p \vdash t : T^E \mid C}}{\Gamma|C_p \vdash K_D(\overrightarrow{t}) : D^{K(\overrightarrow{E})} \mid \bigwedge \overrightarrow{C}}$$

$$\text{ALam}\frac{V \text{ fresh} \qquad x : T_1^V, \Gamma|C_p \vdash t : T_2^E \mid C}{\Gamma|C_p \vdash \lambda x.t : (T_1^V \to T_2^E)^\top \mid C}$$

$$\text{AMat}\frac{\begin{array}{c} V \text{ fresh} \qquad \Gamma|C_p \vdash t : T^E \mid C_{dsc} \qquad \overrightarrow{\Gamma|C_p \vdash P_i : T^{E \cap \overline{\mathcal{P}}_i(\overrightarrow{P})}|\Gamma_i}^i \\ \overrightarrow{C_i := (E \cap \mathcal{P}[\![P_i]\!] \cap \overline{\mathcal{P}}_i(\overrightarrow{P}) \not\subseteq \bot)}^i \qquad \overrightarrow{\Gamma_i|C_i \wedge C_p \vdash t_i' : T'^{E_i'}}^i \\ C_{res} := \bigwedge \overrightarrow{C_i \implies E_i' \subseteq V}^i \qquad C_{saf} := (C_p \implies (E \subseteq \bigcup \overrightarrow{\mathcal{P}[\![P_i]\!]}^i)) \end{array}}{\Gamma|C_p \vdash \texttt{match } t \texttt{ with } \{\overrightarrow{P_i \Rightarrow t_i'}\} : T'^V \mid C_{dsc} \wedge C_{res} \wedge C_{saf}}$$

$$\text{ALet}\frac{\begin{array}{c} T_1'^{V'} := \mathbf{freshen}(T_1) \qquad x : T_1', \overrightarrow{X}, \Gamma|C_p \vdash t_1 : T_1^E \mid C_1 \\ \overrightarrow{V} = (\mathsf{FV}(E) \cup \mathsf{FV}(C_1)) \setminus (\mathsf{FV}(\Gamma) \cup \mathsf{FV}(C_p)) \qquad T_1'^{V'} \equiv T_1^E \wedge C_1 \text{ satisfiable} \\ x : (\forall \overrightarrow{X}, \overrightarrow{V}. (T_1'^{V'} \equiv T_1^E \wedge C_1) \Rightarrow T_1^E), \Gamma|C_p \vdash t_2 : T_2^{E_2} \mid C_2 \end{array}}{\Gamma|C_p \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : T_2^{E_2} \mid C_2}$$

$\boxed{\Gamma \vdash P : T^E | \Gamma'}$ (Analysis pattern environments. P, T^E are input, Γ' is output)

$$\text{Var}\frac{}{\Gamma \vdash x : T^E | (x : T^E), \Gamma} \qquad \text{Ctor}\frac{K(T_1, \ldots, T_n) \in \Delta(D) \qquad \overrightarrow{\Gamma \vdash P : T^{K^{-i}(E)} : \Gamma_i'}^i}{\Gamma \vdash K_D(\overrightarrow{P}) : D^E | \bigcup \overrightarrow{\Gamma_i'}^i}$$

Fig. 4. Pattern Match Analysis

is used to generate these refined types. Moreover, the analysis is push-button: no additional input need be provided by the programmer. It is sound but conservative: it accounts for all possible values an expression may take, but may 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 $K_D(\overrightarrow{t})$, and we know the possible forms $\overrightarrow{E}$ for $\overrightarrow{t}$, then in any context, this expressions

$\boxed{T_1 \equiv T_2 := C}$ (Type equating)

$(T_1 \to T_1')^{E_1} \equiv (T_2 \to T_2')^{E_2} := (T_1 \equiv T_2) \wedge (T_1' \equiv T_2') \wedge E_1 = E_2$

$X^{E_1} \equiv X^{E_2} := E_1 = E_2 \qquad D^{E_1} \equiv D^{E_2} := E_1 = E_2 \qquad T_1 \equiv T_2 := \mathbf{F}\ otherwise$

$\boxed{\mathbf{freshen}(T) := T}$ (Annotation freshening where V fresh)

$\mathbf{freshen}(X^E) := X^V \qquad \mathbf{freshen}(D^E) := D^V$

$\mathbf{freshen}((T_1 \to T_2)^E) := (\mathbf{freshen}(T_1) \to \mathbf{freshen}(T_2))^V$

$\boxed{\mathcal{P}[\![x]\!] := E}$ (Set expression matched by pattern)

$\mathcal{P}[\![x]\!] := \top \qquad \mathcal{P}[\![K(\overrightarrow{P})]\!] := K(\overrightarrow{\mathcal{P}[\![P]\!]})$

$\boxed{\overline{\mathcal{P}}_i(\overrightarrow{P}) := C}$ (Not-yet covered pattern at branch i)

$\overline{\mathcal{P}}_0(P_0 \ldots P_n) = \top \qquad \overline{\mathcal{P}}_i(P_0 \ldots P_n) = \neg\mathcal{P}[\![P_0]\!] \cap \ldots \neg\mathcal{P}[\![P_{i-1}]\!]$ when $0 < i \leq n$

Fig. 5. Auxiliary Metafunctions

can only ever evaluate to values in the set $K(\overrightarrow{E})$. 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 AVAR for variables looks up a scheme from Γ. However, typing schemes now quantify over type and set variables, and carry a constraint along with the type. We then 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 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 constraints 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, $(T_1^{E_1} \to T_2^{E_2})$ means that the function can safely accept any expression matching E_1, and may return values matching E_2.

Applications are analyzed using 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 domain, under the assumption that the path

condition holds and this function call is actually reachable. The metafunction $T_1^{E_1} \equiv T_1^{E_1'}$ (defined in Fig. 5) traverses the structure of the argument and function domain type, constraining that parallel annotations are equal. This traversal is possible because the underlying type system guarantees that the function domain and argument have identical underlying types.

Constructors: 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 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 matching. The rule AMAT begins by inferring the constraint C_{dsc} and annotation E for the discriminee t.

For each branch, we perform two tasks. First, for each branch's pattern P_i, we use an auxiliary judgment to generate the environment Γ_i binding the pattern variables to the correct types and annotations, using projection to access the relevant parts. For P_1, the annotation of the whole pattern is E i.e. the annotation for t. However, the first-match semantics mean that if we reach P_i, then the discriminee does not match any of $P_1 \ldots P_{i-1}$. So for each P_i, we extend the environment with annotations obtained by intersecting E with the negation of all previous patterns, denoted $\overline{\mathcal{P}}_i(\overrightarrow{P})$ (Fig. 5).

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 E_i'. When checking the results, we augment the path constraint with C_i, asserting that some possible input matches this branch's pattern, obtained via $\mathcal{P}[\![]\!]$ (Fig. 5), but none of the previous. 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. We generate a fresh variable V for the return annotation, and constrain that it contains the result E_i' of each branch, provided that it C_i holds, and it is possible we actually took that branch. This uses implication, justifying the need for a solver that supports negation and disjunction.

Finally, we emit a safety constraint C_{saf}, 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 ALET 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. 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 **freshen** takes the underlying type for t_1 and adds fresh annotation variables across the entire type. This allows for monomorphic recursion. The metafunction $\equiv$ constrains the freshly generated variables on T' to be equal 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 variables free in the inferred annotation or constraint but not free in Γ 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, and are typed with environment $\cdot$ and path constraint $\mathbf{T}$.

Example - Safety Constraints: To illustrate our analysis, we return to the Ngon code from Sec. 1. We assume that all Double terms are given annotation $\top$. Then, the simpleArea function would be given the annotated type scheme $\forall V_1, V_2.\, C_1 \wedge C_2 \wedge C_3 \Rightarrow \mathtt{Ngon}^{V_1} \rightarrow \mathtt{Double}^{V_2}$, where

$$C_1 := V_1 \subseteq \mathtt{Square}(\top) \cup \mathtt{Circle}(\top) \quad C_2 := (V_1 \cap \mathtt{Square}(\top) \not\subseteq \bot) \implies \top \subseteq V_2$$
$$C_3 := ((V_1 \cap \mathtt{Circle}(\top) \cap \neg\mathtt{Square}(\top)) \not\subseteq \bot) \implies \top \subseteq V_2$$

C_1 is the C_{saf} generated by the AMAT rule, saying that the function can safely accept input from $\mathtt{Square}(\top) \cup \mathtt{Circle}(\top)$. C_2 and C_3 are conjuncts of C_{res}, describing how, if the input overlaps with Square then the output can be anything, and that if the input overlaps with Circle but not Square, then the output can be anything. C_2 and C_3 are trivially satisfiable: $\mathtt{Circle}(\top) \cap \neg\mathtt{Square}(\top)$ is $\mathtt{Circle}(\top)$, so they are essentially saying that V_2 must be $\top$.

When we call simpleArea from area, we are in the branch after the Ngon case has been checked. The scheme for simpleArea is instantiated with the path constraint $V_4 \subseteq \top \cap \neg(\mathtt{Ngon}(\top))$, where V_4 is the annotation for shape, because it is called after we have a failed match with Ngon sides.

Suppose we instantiate V_1, V_2 with fresh V_1', V_2'. The call to simpleArea creates a constraint that $V_4 = V_1'$. Taking this equality into account, the safety constraint is instantiated to $V_4 \subseteq \top \cap \neg(\mathtt{Ngon}(\top)) \implies V_4 \subseteq (\mathtt{Square}(\top) \cup \mathtt{Circle}(\top))$. This is satisfiable for any value of shape, so at every call to area the analysis sees that the safety constraint is satisfied. If we add a Triangle constructor, then the constraint is unsatisfiable any time V_4 is instantiated to a set with Triangle.

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 map function:

```
intMap : (Int -> Int) -> List Int -> List Int ->
intMap f l = case l of
  Nil -> Nil
  Cons h t -> Cons (f h) (intMap f t)
```

Suppose we have concrete arguments $\mathtt{f} : (\mathtt{Int}^{V_{11}} \rightarrow \mathtt{Int}^{V_{12}})^{V_1}$ and $\mathtt{l} : (\mathtt{List\,Int})^{V_2}$. The safety constraint for the match is that $V_2 \subseteq \mathtt{Nil} \cup \mathtt{Cons}(\top, \top)$, 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 $V_2 \cap \mathtt{Nil} \not\subseteq \bot \implies \mathtt{Nil} \subseteq V_3$.

The analysis is more interesting for the second branch. The bound pattern variables h and t are given annotations $\texttt{Cons}^{-1}(V_2)$ and $\texttt{Cons}^{-2}(V_2)$ respectively, since they are the first and second arguments to $\texttt{Cons}$. Because our recursion is monomorphic, the recursive call $\texttt{intMap f t}$ generates the trivial constraint $(V_2 \cap \neg \texttt{Nil} \cap \texttt{Cons}(\top, \top)) \not\subseteq \bot \implies V_1 \subseteq V_1$, and the more interesting constraint $(V_2 \cap \neg \texttt{Nil} \cap \texttt{Cons}(\top, \top)) \not\subseteq \bot \implies \texttt{Cons}^{-2}(V_2) \subseteq V_2$. 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 $\texttt{Cons}(\texttt{a}, \texttt{Nil}), \texttt{Cons}(\texttt{a}, \texttt{Cons}(\texttt{b}, \texttt{Nil})), \ldots$ 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 $\texttt{intMap}$. For the result of the second branch, we have the constraints $V_2 \cap \neg \texttt{Nil} \cap \texttt{Cons}(\top, \top) \not\subseteq \bot \implies \texttt{Cons}(V_{12}, V_5) \subseteq V_3$. This essentially says that if the input to the function can be $\texttt{Cons}$, then so can the output, but if the input is always $\texttt{Nil}$, then this branch contributes nothing to the overall result. Finally, we have a constraint $V_5 = V_3$, generated by the metafunction $\equiv$.

Our result annotation V_3 is constrained by $(V_2 \cap \texttt{Nil} \not\subseteq \bot \implies \texttt{Nil} \subseteq V_3)$ $\wedge\ (V_2 \cap \neg \texttt{Nil} \cap \texttt{Cons}(\top, \top) \not\subseteq \bot \implies \texttt{Cons}(V_{12}, V_3) \subseteq V_3)$, capturing how $\texttt{intMap}$ returns nil empty result for nil input, and non-nil results for non-nil input.

3 Translating Set Constraints to SMT

While the above analysis provides a fine-grained way to determine which pattern matches may not be safe, it depends on the existence of an external solver to check the satisfiability of the resulting set constraints. We provide a simple, performant solver by translating set constraints into an SMT formula.

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 $\mathcal{F} = \{f_1^{a_1}, \ldots, f_n^{a_n}\}$, where each $a \geq 0$ is the arity of the function f_i^a. The *Herbrand Universe* $\mathcal{H}_{\mathcal{F}}$ is defined inductively: each $f_i^0 \in \mathcal{F}$ 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 like a constructor in a strict functional language. We assume all terms are finite, although similar analyses can account for laziness and infinite data [28].

This allows us to formalize the semantics of set expressions. The syntax is the same as inFig. 3, although we use the notation $f_i^a(\overrightarrow{E})$ instead of $K_D(\overrightarrow{E})$ to denote that we are using arbitrary function symbols from some Herbrand universe $\mathcal{H}$, instead of specific constructors for a datatype. Given a substitution $\sigma : \mathcal{V} \to \mathcal{P}(\mathcal{H})$, we can assign a meaning $\mathcal{H}[\![E]\!]_\sigma \subseteq \mathcal{H}$ for an expression E by

$$\mathcal{H}[\![\bot]\!]_\sigma = \emptyset$$
$$\mathcal{H}[\![\top]\!]_\sigma = \mathcal{H}$$
$$\mathcal{H}[\![V]\!]_\sigma = \sigma(V)$$
$$\mathcal{H}[\![\neg E_1]\!]_\sigma = \mathcal{H} \setminus \mathcal{H}[\![E_1]\!]_\sigma$$

$$\mathcal{H}[\![E_1 \cap E_2]\!]_\sigma = \mathcal{H}[\![E_1]\!]_\sigma \cap \mathcal{H}[\![E_2]\!]_\sigma$$
$$\mathcal{H}[\![E_1 \cup E_2]\!]_\sigma = \mathcal{H}[\![E_1]\!]_\sigma \cup \mathcal{H}[\![E_2]\!]_\sigma$$
$$\mathcal{H}[\![f_i^a(E_1,\ldots,E_a)]\!]_\sigma = \{f_i^a(h_1,\ldots,h_a)$$
$$\mid h_1 \in \mathcal{H}[\![E_1]\!]_\sigma,\ldots,h_a \in \mathcal{H}[\![E_a]\!]_\sigma\}$$

Fig. 6. Semantics of Set Expressions

mapping variables to their substitutions, and applying the corresponding set operations. The full semantics are given in Fig. 6. Note that the expressions on the left are to be interpreted as *syntax*, whereas those on the right are mathematical sets.

A *set constraint atom* $\mathcal{A}$ 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* $\mathcal{L}$ is either an atom or its negation $\neg(E_1 \subseteq E_2)$, which we write as $E_1 \not\subseteq E_2$.

Constraints which contain negative literals are called *negative set constraints*. An unrestricted set constraint, denoted by metavariable $\mathcal{C}$, is a boolean combination (i.e. using $\wedge$, $\vee$ and $\neg$) of set constraint atoms, as we defined in Fig. 3. For example, $(X \subseteq Y \implies Y \subseteq X) \wedge (Y \not\subseteq Z)$ is an unrestricted set constraint.

Given a set constraint C, the satisfiability problem is to determine whether there exists a substitution $\sigma : \mathcal{V} \to \mathcal{P}(\mathcal{H})$ 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 $\overrightarrow{L}$ is to find a variable assignment that causes $\bigwedge \overrightarrow{L}$ to be true. We explain how to extend our approach to arbitrary boolean combinations in Sec. 3.7.

One can 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 analyses (including ours) on a notion of *projection*. For a set expression E, we denote the jth projection of E for function f_i^a by $f_i^{-j}(E)$. For a substitution σ, we have $\mathcal{H}[\![f_i^{-j}(E)]\!]_\sigma = \{h_j \mid f_i^a(h_1,\ldots,h_j,\ldots h_a) \in E\}$.

While we don't explicitly include projections in our grammar for set expressions, we can easily express them using boolean formulae. Given some constraint $C[f_i^{-j}(E)]$, we can replace this with:
$C[X_j] \wedge (E \cap f_i^a(\top,\ldots,\top)) = f_i^a(X_1,\ldots,X_j,\ldots X_a) \wedge (E = \bot \iff X_j = \bot)$
where each X_k is a fresh variable. The first condition specifies that our variable holds the jth component of every $f(\overrightarrow{h})$ in E. The second condition is neces-

sary because $f_i^a(X_1, \ldots, X_j, \ldots X_a) = \bot$ if *any* X_k is empty, so any value of X_j vacuously satisfies $E' = f_i^a(X_1, \ldots, X_j, \ldots X_a)$ if E' and some X_i are empty.

3.3   Set Constraints and Monadic Logic

The first step in our translation is converting a conjunction of set constraint literals into a formula in first-order monadic logic, for which satisfiability is decidable. We then translate this 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.

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] found a translation from a conjunction $\bigwedge \overrightarrow{L}$ of positive set constraint atoms to an equisatisfiable monadic formula, which was later extended to negative set constraints with equality [11]. We summarize their procedure here, with a full definition in Fig. 7. For each sub-expression E of $\bigwedge \overrightarrow{L}$, we create a predicate $P_E(x)$, denoting 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 [38]. 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 $\exists x. (P_{E_1}(x) \wedge \neg P_{E_2}(x))$. [1]

The key utility of having a monadic formula is the *finite model property* [1, 29]:

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

The intuition behind this is that if there exists a model satisfying $\mathcal{S}$, then then we can combine objects 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.

Example - Translation: Consider C_3 from the safety constraint example in Sec. 2. We see that $\mathcal{L}[\![((V_1 \cap \mathtt{Circle}(\top) \cap \neg\mathtt{Square}(\top)) \not\subseteq \bot) \implies \top \subseteq V_2]\!]$ is
$(\exists y. P_{V_1 \cap \mathtt{Circle}(\top) \cap \neg\mathtt{Square}(\top)}(y) \wedge \neg\mathbf{F}) \implies (\forall x. \mathbf{T} \implies P_{V_2}(x))$. Applying the $\mathcal{E}[\![\,]\!]$ equivalences for $\cap, \cup$ and $\neg$ with basic laws of predicate logic gives us:

---

[1]   The original translation transformed constants and functions into existential variables. We skip this, since SMT supports uninterpreted functions and constants.

$M \in \mathcal{M}$ (Monadic formulae)

$\boxed{\mathcal{E}[\![E]\!] = M}$ (Predicates for set expressions)

$$\mathcal{E}[\![\top]\!] = \forall x.\, P_\top(x) \quad | \quad \mathcal{E}[\![\bot]\!] = \forall x.\, \neg P_\bot(x) \quad | \quad \mathcal{E}[\![X]\!] = \mathbf{T}$$

$$\mathcal{E}[\![E_1 \cap E_2]\!] \quad = \quad \forall x.\, P_{E_1 \cap E_2}(x) \iff (P_{E_1}(x) \wedge P_{E_2}(x))$$

$$\mathcal{E}[\![E_1 \cup E_2]\!] \quad = \quad \forall x.\, P_{E_1 \cup E_2}(x) \iff (P_{E_1}(x) \vee P_{E_2}(x))$$

$$\mathcal{E}[\![\neg E_1]\!] \quad = \quad \forall x.\, P_{\neg E_1}(x) \iff \neg P_{E_1}(x)$$

$$\mathcal{E}[\![f_i^a(E_1, \ldots, E_a)]\!] \quad = \quad (\forall x_1 \ldots x_a.\, P_{f_i^a(E_1,\ldots,E_a)}(f_i^a(x_1,\ldots,x_a)) \iff P_{E_1}(x_1) \wedge \ldots P_{E_a}(x_a))$$

$$(\bigwedge_{g_j^{a'} \neq f_i^a} \forall x_1 \ldots x_{a'} P_{f_i^a(E_1,\ldots,E_a)}(g_j^{a'}(x_1,\ldots,x_a')) \iff \mathbf{F})$$

$\boxed{\mathcal{L}[\![L]\!] = M}$ (Literal predicates) $\boxed{\mathcal{L}[\![\bigwedge \overrightarrow{L}]\!] = M}$ (Conjunction)

$$\mathcal{L}[\![E_1 \subseteq E_2]\!] = \forall x.\, P_{E_1}(x) \implies P_{E_2}(x) \qquad \mathcal{L}[\![\bigwedge \overrightarrow{L}]\!] = \mathcal{E}[\![E_1]\!] \wedge \ldots \wedge \mathcal{E}[\![E_n]\!] \wedge \bigwedge \overrightarrow{\mathcal{L}[\![L]\!]}$$

$$\mathcal{L}[\![E_1 \not\subseteq E_2]\!] = \exists y.\, P_{E_1}(y) \wedge \neg P_{E_2}(y) \qquad \text{where } E_1 \ldots E_n \text{ all subexpressions of } \overrightarrow{L}$$

Fig. 7. Translating Set Constraints to Monadic Logic

$(\exists y.\, P_{V_1}(y) \wedge P_{\mathtt{Circle}(\top)}(y) \wedge \neg P_{\mathtt{Square}(\top)}(y)) \implies \forall x.\, P_{V_2}(x)$.
Finally, adding the $\mathcal{E}[\![\,]\!]$ conditions for functions gives us:
$(\forall x.\, P_{\mathtt{Circle}(\top)}(f_{Circle}(x))) \quad \wedge \quad (\forall x.\, \neg P_{\mathtt{Circle}(\top)}(f_{Square}(x)))$
$\wedge \quad (\forall x.\, P_{\mathtt{Square}(\top)}(f_{Square}(x))) \quad \wedge \quad (\forall x.\, \neg P_{\mathtt{Square}(\top)}(f_{Circle}(x)))$
$\wedge \quad ((\exists y.\, P_{V_1}(y) \wedge P_{\mathtt{Circle}(\top)}(y) \wedge \neg P_{\mathtt{Square}(\top)}(y)) \quad \implies \quad \forall x.\, P_{V_2}(x))$.
C_3 is satisfiable iff there is a model defining predicates $P_{V_1}, P_{V_2}, P_{\mathtt{Circle}(\top)}$ and
$P_{\mathtt{Square}(\top)}$, and functions f_{Circle}, f_{Square} in which the above formula is true.

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 $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ is the set of
booleans, which we call *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 :
$D \to \mathbb{B}$ for each predicate P and $f_i^a : D^a \to D$ for each function, which define
the value of $P(x)$ and $f(x_1, \ldots, x_a)$ for each $x, x_1, \ldots, x_a \in D$. 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_i^a, 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 $\mathbf{T}$ 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 ith bit of sequence b.

$$
\begin{aligned}
P_\top(b) &:= \mathbf{T} \\
P_X(b) &:= \text{bit for } X \text{ in } b \\
P_{f_i^a(E_1,\ldots E_a)}(b) &:= \text{bit for } f_i^a(E_1,\ldots E_a) \text{ in } b \\
P_{\neg E_1}(b) &:= \neg P_{E_1}(b)
\end{aligned}
$$

$$
\begin{aligned}
P_\bot(b) &:= \mathbf{F} \\
P_{E_1 \cap E_2}(b) &:= P_{E_1}(b) \wedge P_{E_2}(b) \\
P_{E_1 \cup E_2}(b) &:= P_{E_1}(b) \vee P_{E_2}(b)
\end{aligned}
$$

Fig. 8. Recursive Definition of Predicates for the SMT Translation

However, with this interpretation, $\mathbb{B}^N$ is too large to be our domain. Suppose we have formulae E_i 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 $\mathbf{T}$. 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 $\mathcal{D} : \mathbb{B}^N \to \mathbb{B}$, which determines whether a bit-sequence is in the domain of a potential model. If $\mathcal{L}[\![\bigwedge \overrightarrow{L}]\!]$ contains the formula $\forall x_1 \in D \ldots \forall x_n \in D.\, \Phi[x_1 \ldots x_n]$, for some Φ, we can instead write:
$\forall b_1 \in \mathbb{B}^N \ldots \forall b_n \in \mathbb{B}^N.\, \mathcal{D}(b_1) \wedge \ldots \wedge \mathcal{D}(b_n) \implies \Phi[b_1 \ldots b_n]$.
That is, our domain can only contain values that respect the semantics of set expressions. Similarly, if $\mathcal{L}[\![\bigwedge \overrightarrow{L}]\!]$ contains $\exists x.\, \Phi[x]$, we can write $\exists b \in \mathbb{B}^N.\, \mathcal{D}(b) \wedge \Phi[b]$. Since all functions in a model are implicitly closed over the domain, we also specify that $\forall \overrightarrow{b} \in (\mathbb{B}^n)^a.\, \overrightarrow{\mathcal{D}(b)} \implies \mathcal{D}(f_i^a(\overrightarrow{b}))$. This ensures that our formulae over boolean sequences are equivalent to the original formulae.

This is enough to express $\mathcal{L}[\![\bigwedge \overrightarrow{L}]\!]$ as an SMT problem. We assert the existence of $\mathcal{D} : \mathbb{B}^N \to \mathbb{B}$ along with $f_i^a : (\mathbb{B}^N)^a \to \mathbb{B}^N$ for each function in our Herbrand universe. We modify each formula in $\mathcal{L}[\![\bigwedge \overrightarrow{L}]\!]$ 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 $\mathcal{D}$ qualifiers to existentially and universally quantified formulae, and replace each $P_{E_i}(x_j)$ with the ith bit of b_j. We add a constraint asserting that each f_i^a is closed over the values satisfying $\mathcal{D}$. The SMT solver searches for values for all existential variables, functions, and $\mathcal{D}$ 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, $\mathcal{D}$ will always reject boolean sequences that violate the constraints of each $\mathcal{E}[\![E_i]\!]$. For example, the bit for $P_{E_1 \cap E_2}$ in b must always be exactly $P_{E_1}(b) \wedge P_{E_2}(b)$. 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. 8). This means that our boolean sequences need only contain slots for expressions of the form X or $f_i^a(E_1,\ldots E_a)$, 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 $f_i^a(E_1,\ldots E_a)$, since the other constraints hold *by definition* given our definitions of each P_E. Similarly, our constraints restrict the free-

dom we have in choosing f_i^a. Specifically, we know that $P_{f_i^a(E_1,\ldots,E_a)}(f_i^a(b_1,\ldots,b_a))$ should hold if and only if $P_{E_i}(b_i)$ holds for each $i \leq a$. Similarly, we know that $P_{f_i^a(E_1,\ldots,E_a)}(g_j^{a'}(b_1,\ldots,b_{a'}))$ should always be $\mathbf{F}$ when $f \neq g$. So for each f_i^a, it suffices to find a mapping from inputs $b_1,\ldots,b_a$ to the value of $P_X(f_i^a(b_1,\ldots,b_a))$ for each variable X. This reduces the number of unknowns in the SMT problem.

3.6 The Complete Translation

Given a conjunction of literals $\bigwedge \overrightarrow{L}$, let $X_1,\ldots X_k, E_{k+1},\ldots E_N$ be the sequence of variable and function-application sub-expressions of $\overrightarrow{L}$. We define $P_E(b)$ for each sub-expression E of $\overrightarrow{L}$ as in Fig. 8.

As unknowns, we have:

- a function $\mathcal{D} : \mathbb{B}^N \to \mathbb{B}$;

- for each negative literal $E_i \not\subseteq E_i'$, an existential variable $y_i \in \mathbb{B}^N$;

- for each function f_i^a and each variable $X \in \overrightarrow{L}$, a function $f_{iX}^a : (\mathbb{B}^N)^a \to \mathbb{B}$, which takes a sequences of N bits, and computes the value of the bit for P_X in the result.

We define the following known functions:

- $f_{if_i^a(E_1,\ldots,E_a)}^a : (\mathbb{B}^N)^a \to \mathbb{B}$ for each f_i^a and each sub-expression of the form $f_i^a(E_1,\ldots,E_a)$, where $f_{if_i^a(E_1,\ldots,E_a)}^a(b_1,\ldots,b_a) = P_{E_1}(b_1) \wedge \ldots \wedge P_{E_a}(b_a)$;

- $f_{ig_j^{a'}(E_1,\ldots,E_{a'})}^a : (\mathbb{B}^N)^a \to \mathbb{B}$ returning $\mathbf{F}$, for each f_i^a and each sub-expression of the form $g_j^{a'}(E_1,\ldots,E_{a'})$ where $f \neq g$;

- $f_{iSMT}^a : (\mathbb{B}^N)^a \to \mathbb{B}^N$ for each f_i^a, where $f_{iSMT}^a(b_1,\ldots,b_a)$ is the sequence:
  $f_{iX_1}^a(b_1,\ldots,b_a)\ldots f_{iX_k}^a(b_1,\ldots,b_a)f_{iE_{k+1}}^a(b_1,\ldots,b_a)\ldots f_{iE_N}^a(b_1,\ldots,b_a)$

We assert that the following hold:

- for each negative constraint $E_i \not\subseteq E_i'$ with corresponding existential variable y_i, that $\mathcal{D}(y_i) \wedge P_{E_i}(y_i) \wedge \neg P_{E_I'}(y_i)$ holds;

- $\forall x \in \mathbb{B}^N . (\mathcal{D}(x) \wedge P_{E_i}(x)) \implies P_{E_i'}(x)$ for each positive $E_i \subseteq E_i'$;

- $\forall x_1 \ldots x_a . (\bigwedge_{j=1\ldots a} \mathcal{D}(x_j)) \implies \mathcal{D}(f_{iSMT}^a(x_1,\ldots,x_a))$ for each function f_i^a

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

3.7 Arbitrary Boolean Combinations

Allowing arbitrary boolean combinations of set constraints enriches our pattern match analysis and to allow us to use projections. To do this, for each atom $E_i \subseteq E_i'$ in a constraint C, we introduce a boolean ℓ_i, which the SMT solver guesses. We modify our translation so that $\mathcal{L}[\![E_i \subseteq E_i']\!] = \ell_i \implies \forall x . (P_{E_i}(x) \implies P_{E_i'}(x))$ and $\mathcal{L}[\![E_i \not\subseteq E_i']\!] = \neg\ell_i \implies (\exists y . P_{E_i}(y) \wedge \neg P_{E_i'}(y))$. So l_i is true 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 ℓ_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 ℓ_i is false,

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

then $\mathcal{L}[\![E_i \subseteq E_i']\!]$ will be vacuously true, with the opposite holding for negative constraints.

4   Evaluation and Discussion

We implemented our translation [18] atop Z3 4.8.5 with `mbqi` and `UFBV`. On an i7-3770 CPU 32GB RAM machine, we compared the running time of Elm's exhaustiveness check with an implementation of our analysis [17].

In order to make the analysis practical, we implemented several optimizations on top of our analysis. Trivially satisfiable constraints were removed, and obvious simplifications were applied to set expressions. When a match was exhaustive, its safety constraint was omitted, and since non-safety constraints should be satisfiable, calls to Z3 were only made for non-empty safety constraint lists. A union-find algorithm was used to 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 ⊤. 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[21], intdict[22], 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 the invariant that a connected component's depth-first search forest has exactly one element, which was 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 constraints. Conversely, in the **-TN** cases where Z3 was not called, our analysis cause very little slowdown. Partial matches tend to occur rarely in code, so we feel this is acceptable performance for a tool integrated into a compiler.

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 solver 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 [23]. Several independent proofs of decidability for systems with negative constraints were given, using a number-theoretic reduction [2, 37], tree automata [20], and monadic logic [11]. Charatonik and Podelski established the decidability of positive and negative constraints with projection [34]. 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 improved BANE's performance with code generation and incremental analysis [26]. 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, 24, 33].

Related Work - Pattern Match Analysis: Several pattern match analyses have been presented in previous work. Koot [27] 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 [28], 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, 35, 36], similar to our path constraints. Castagna et al. [10] describe a similar system, albeit more focused on type-case than pattern matching. Catch [30] uses a similar system of *entailment*, with a restricted constraint language to ensure finiteness. 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 where unions and intersections can appear in conditional constraints [6], and there is not full sup-

port 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 [36]. While these have not been integrated into our implementation, our constraint logic makes it easy to incorporate these and other future improvements.

Another related line of work is *datasort refinements*. [14–16, 19]. 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.

Conclusion: Unrestricted set constraints previously were used only 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. We have shown that even 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), 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), 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`
9. Bell, R.K.: elm-interval. `https://github.com/r-k-b/elm-interval/` (2019), commit a7f5f8a
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.: Introduction to Elm (2019), `http://guide.elm-lang.org/`
13. Damas, 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 Languages. pp. 281–292. POPL ’04, ACM, New York, NY, USA (2004), `http://doi.acm.org/10.1145/964001.964025`
17. Eremondi, J.: Forked elm-compiler. `https://github.com/JoeyEremondi/elm-compiler-patmatch-smt` (2019), commit 9581aaf
18. Eremondi, J.: Setconstraintssmt. `https://github.com/JoeyEremondi/SetConstraintsSMT` (2019), commit 03bb754
19. 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`

20. 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)
21. Graf, S.: elm-graph. `https://github.com/elm-community/graph/` (2019), commit 3672c75
22. Graf, S.: elm-intdict. `https://github.com/elm-community/intdict` (2019), commit bf2105d
23. 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)
24. 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)
25. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press (2018)
26. Kodumal, J., Aiken, A.: Banshee: A scalable constraint-based analysis toolkit. In: Hankin, C., Siveroni, I. (eds.) Static Analysis. pp. 218–234. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
27. Koot, R.: Higher-Order Pattern Match Analysis. Master's thesis, Universiteit Utrecht, the Netherlands (2012)
28. 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`
29. Löwenheim, L.: Über möglichkeiten im relativkalkül. Mathematische Annalen 76(4), 447–470 (Dec 1915), `https://doi.org/10.1007/BF01458217`
30. Mitchell, N., Runciman, C.: Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. SIGPLAN Not. 44(2), 4960 (Sep 2008), `https://doi.org/10.1145/1543134.1411293`
31. 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)
32. 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`
33. 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/BFb0017466`
34. 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`
35. Palmer, Z., Menon, P.H., Rozenshteyn, A., Smith, S.: Types for flexible objects. In: Garrigue, J. (ed.) Programming Languages and Systems. pp. 99–119. Springer International Publishing, Cham (2014)

36. 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`
37. Stefansson, K.: Systems of set constraints with negative constraints are NEXPTIME-complete. In: Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science. pp. 137–141 (Jul 1994)
38. 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`