Submission to POPL 2016 Student Research Competition, Undergraduate Category

Polyvariant Pattern Match Analysis with Implication Constraints

Joey Eremondi
Utrecht University
j.s.eremondi@students.uu.nl

Abstract

The use of static types in functional languages eliminates a large class of runtime errors. However, incomplete pattern matches are a common source of errors in functional programs. This research provides an approximate analysis, based on a polyvariant type-and-effect system with subeffecting, which traces the flow of data through programs in order to identify which cases actually need to be matched in order to prevent a runtime crash. A modified constraint-solving algorithm is presented, which increases knowledge about the result of a case split, and about variables within case branches, based on knowledge of the expression split upon.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Program analysis; D.3.3 [*Programming Languages*]: Language Constructs and Features - Patterns; D.2.4 [*Programming Languages*]: Software/Program Verification

General Terms Languages, Verification, Theory

Keywords Type and effect systems, pattern matching, polyvariant, subeffects, functional programming, Elm

1. Problem and Motivation

While Hindley-Milner (HM) inference prevents a large class of runtime errors, whenever Abstract Data Types are present, incomplete pattern matches are a common source of crashes.

A potential solution to this is to require all pattern matches to be total, as was recently adopted by the Elm language. However, this is inflexible: when a function is not total on its input type, but the programmer believes unmatched patterns are never passed to the function, code snippets like the following frequently arise:

```
foo l = case l of
  (h :: t) -> doSomething h t
  [] -> Debug.crash "Impossible"
```

As an alternative to requiring exhaustive matches, we use static analysis, based on type-and-effect systems, to track the flow of data through a program and remove warnings from pattern matches which are incomplete but guaranteed to never cause an error.

[Copyright notice will appear here once 'preprint' option is removed.]

2. Background and Related Work

Type and effect systems, described in detail in [6], perform program analysis through a process very similar to a type-inference algorithm. The exhaustiveness problem is described in [4]. In [5], the problem of finding safe, non-exhaustive matches is addressed for first-order functions in Haskell.

Alternate approaches, such as higher-ranked analysis [3], can improve on the annotations assigned by type and effect systems, at the cost of a more complex analysis.

This work is heavily based on [2], which presents pattern match analysis on lists and booleans in Haskell, and briefly discusses implication constraints. We expand upon this with the use of arbitrary user-defined datatypes, and elaborate on the ways implication constraints improve precision.

3. Approach and Uniqueness

Elm is an eager language with HM types. Annotations for each type, with a subeffecting lattice, are defined recursively:
$A := \top \mid A_1 \to A_2 \mid \{\} \mid \{P_i, \dots, P_k\} \mid \alpha$ (variable)
$P := C(A_1, \dots, A_m)$
$\forall S.\, S \sqsubseteq \top$
$S \to R \sqsubseteq S' \to R' \iff R \sqsubseteq R' \wedge S' \sqsubseteq S$
S, S' finite $\implies (S \sqsubseteq T \iff \forall s \in S.\, \exists t \in T.\, s \sqsubseteq' t)$
$C(a_1, \dots a_m) \sqsubseteq' C(b_1, \dots b_m) \iff \forall 1 \leq i \leq m.\, a_i \sqsubseteq b_i$

Here C is the name of some constructor. We can see that this subeffecting relation corresponds to pattern matching: if S is a set of pattern forms a value can take on, and R is a set of patterns matched in a particular case expression, we can see that $S \sqsubseteq R$ means that there will never be an unmatched pattern failure when attempting to give a value annotated with S to a case-expression whose case patterns together are annotated with R. To accommodate polyvariance, we define annotation-schemes of the form $\sigma = \forall(\alpha_1 \cdots \alpha_n).\, A$.

3.1 Outline of Inference Rules

Inference-rules are omitted for brevity, but are briefly described. Our inference is based on the generation of constraints, which are later solved. While type inference and pattern analysis could be performed simultaneously, we treat constraint generation and solving as a separate process which occurs after type-checking.

Variable instantiation works as in HM: a new annotation is created, with fresh type variables replacing quantified variables, including variables in stored constraints. Likewise, function annotations are inferred as in HM. An n-argument constructor C is placed in the initial environment with annotation scheme $\forall(\alpha_1 \dots \alpha_n).\, \alpha_1 \to \dots \to \alpha_n \to \{C(\alpha_1, \dots, \alpha_n)\}$. Literals are treated as 0-argument constructors.

Let-expressions quantify over all variables not free in their environment. In order to allow recursion without causing infi-

nite annotations, defined variables are monomorphically inserted into the environment when evaluating their value, with a special $returnsTop$ annotation. This value unifies to $\top$ with a set, but to $\alpha \to returnTop$ with function types. Thus, no assumptions are made about values returned by recursive calls, but we still warn when a recursive call is given a value that it cannot match. We also store constraints on variables defined in let-expressions, so that we can instantiate and solve those constraints later.

3.2 Case Constraints with Implications

The most interesting inference rules are for case-expressions. If-expressions can be analysed similarly. Suppose the expression we match on has annotation E, and that the case-expression has result annotation R, which we will constrain.

An annotation M is formed by joining annotations M_i of each matched pattern, and we enforce the constraint that $E \sqsubseteq M$, ensuring that all possible patterns are matched. If the patterns of M are determined to be exhaustive, M is replaced by $\top$.

For each branch matching M_i, we constrain the result expression to annotation R_i. We then have the following implication constraint: $matches(M_i, E) \implies R_i \sqsubseteq R$, where $matches(P, S) \iff S = \top \vee \exists s \in S.\, s \sqsubseteq' P$. That is, we only let the result of this branch contribute to the overall result of the case-expression if it is possible for the case-expression's argument to match that branch's pattern. Note that the M_i values are always literal patterns we can extract directly from the source code of the pattern match.

To annotate all variables defined in a pattern P, we make an annotation A_P for the entire pattern, and generate a constraint $\forall s \in E.\, (matches(P, s) \implies \{s\} \sqsubseteq A_P)$. Since annotations are either $\top$ or finite, the $\forall$ quantifier does not affect decidability. Each variable x_i in the pattern is then assigned s fresh variable α_i, which is added to the environment when evaluating the result of the case-branch. We constrain α_i by its position in the pattern P.

For example, consider the following code:

```
case l of ((x :: [] ) :: []) -> foo
```

We have the constraint $Cons(1, Cons(1, P)) \sqsubseteq A_x$. This says that A_x contains every anotation stored in the first argument of the $Cons$ that is the first argument of the $Cons$ that is the annotation of P. For any constructor C and $i \in \mathbb{N}$, we define $C(i, \top) = \top$.

These two rules together allow us to make our analysis more precise. The first one allows us to use the annotation of a value to refine which branches of a case statement contribute to the result of a split on that value. The second allows us to use the annotation on a value to refine what values variables from pattern matches have when evaluating a particular branch of a case split.

3.3 Constraint Solving

Inference follows a two-phase approach: constraint generation from inference rules, and constraint solving. In addition to the HM-style constraints of unification, generalization, instantiation, and conjunction, we have subeffecting and implication-subeffecting constraints. A standard inference algorithm solves HM constraints. Subeffecting constraints encountered or instantiated are converted into constraints on variables and literal annotations, then emitted.

We use a worklist algorithm to solve subeffecting constraints, which are only solved for instantiated variables. We begin with all emitted constraints on our worklist, and with every variable having a lower-bound (LB) and upper-bound (UB) bound annotations. Here, the LB set represents patterns the variable may embody while being fully matched in every case-expression it is passed to, and is initialized to $\top$. The upper-bound set contains all patterns the variable could embody, and is initialized to $\{\}$.

For each constraint $a \sqsubseteq b$ in our worklist, we set $UB(b) := UB(b) \sqcup UB(a)$ and $LB(a) := LB(b) \sqcap LB(a)$. If $UB(b)$ is increased, constraints matching $b \sqsubseteq c$ are added to our worklist for updating. We do the same for $LB(A)$ and $c \sqsubseteq a$.

For a constraint $matches(Lit, a) \implies c \sqsubseteq d$, we check if the current upper-bound values of a can match Lit, and if so, solve for $c \sqsubseteq d$, doing nothing otherwise. This constraint is re-added to our worklist whenever a is modified. Similarly, for a constraint $\forall s \in a.\, matches(Lit, s) \implies \{s\} \sqsubseteq b$, if $a = \top$ we set $UB(b) := \top$, and otherwise, we join with $UB(b)$ all sub-patterns s of $UB(a)$ which match Lit. Constrains of the form $C(i, a) \sqsubseteq b$ are solved by looking up the value of the ith argument of each element of a with C as its constructor (recursing deeper if necessary), joining them, then joining that value with $UB(b)$.

When the worklist is empty, we iterate through all variables and see if there are any patterns in the UB set which are not matched by the LB set. If so, a pattern-match warning is emitted.

4. Results and Contributions

Our system can analyse the full feature-set of an eager functional language, accounting for user-provided datatypes. Because it is based heavily on Algorithm W and the worklist algorithm, it is relatively easy to implement. Tools based on this analysis would allow for rapid development, letting the programmer write the cases they know how to handle and add cases as they progress, rather than forcing them to add wildcard matches with calls to `crash` which may be forgotten later.

As with most program analyses, ours is a conservative approximation. Due to the difficulties of recursion, there are some places where a warning is emitted when no error can occur.

An analysis such as this provides a strong foundation for future analyses. For example, because our analysis tracks the flow of literals through a program, it could be used to implement constant propagation. Similarly, adding special constructors for exceptions like `crash` could identify parts of code which are guaranteed to be completely runtime-error free.

Our analysis is more precise than in [2]. Consider:

```
let
  map = \f -> \l -> case l of
    [] -> []
    (a :: b) -> (f a) :: (map f b)
  in case map (\x->x) [1,2] of
    (h :: t) -> (h,t)
```

In [2] this code (falsely) raises a pattern-match warning.

In our analysis, map is assigned the annotation
$\forall ABCDE.\, (E \to A) \to B \to C$, with the constraints
$matches(\{Nil()\}, B) \implies Nil \sqsubseteq C$,
$matches(\{Cons(\top, \top)\}, B) \implies Cons(A, \top) \sqsubseteq C$,
$s \in B \wedge matches(\{Cons(\top, \top)\}, s) \implies s \sqsubseteq D$,
and $Cons(1, D) \sqsubseteq E$.

When we instantiate `map` in the body of the let-expression, and solve the constraints, the the constraint on the list causes the condition of the Nil implication to be false, allowing the analysis to see that the result applying `map` is non-empty and that the match is safe.

Similarly, because the argument we provide has one element matching $Cons$, and our instantiation unifies A and D, we can annotate `h` with $\{1()\}$, the 1st argument to the $Cons$ of the annotation on `[1,2]`.

A modified version of the Elm compiler featuring this analysis is in progress, with the source code viewable at [1].

Acknowledgments

This work was based on a project completed for the *Automatic Program Analysis* class, led by Juriaan Hage at Utrecht University, and was funded by the Utrecht Excellence Scholarship.

References

[1] J. Eremondi. Github repository: elm-pattern-effects. `github.com/JoeyEremondi/elm-pattern-effects`, 2015.

[2] R. Koot. Higher-order pattern match analysis. Master's thesis, Universiteit Utrecht, 2012. http://dspace.library.uu.nl/handle/1874/256268.

[3] R. Koot and J. Hage. 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*, PEPM '15, pages 127–138, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3297-2. . URL `http://doi.acm.org/10.1145/2678015.2682542`.

[4] L. Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17:387–421, 2007. ISSN 1468-7653. . URL `http://journals.cambridge.org/article`$_S$0956796807006223.

[5] N. Mitchell and C. Runciman. A static checker for safe pattern matching in haskell. In M. C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6, chapter 2. Intellect, 2005.

[6] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.