Exploring Software with Symbolic and Statistical Algorithms

Chris Warburton
University of Dundee
cmwarburton@dundee.ac.uk

ABSTRACT

We aim to aid programmers in understanding and maintaining software by increasing the capability of *theory exploration* systems for discovering novel, unexpected properties of programs and libraries. Existing theory exploration systems find equivalent program fragments via brute-force search, which doesn't scale well to realistic codebases. We narrow down the scope of each exploration run using statistics derived from the semantic similarity of the source code being explored, using a process called *recurrent clustering*.

1. INTRODUCTION

As computers become ever more important in research, industry and society, so does the ability to program them effectively, economically and *correctly*. Whilst tools exist to help programmers understand and manipulate software, ranging from linters [8] through to formal verification systems [1], these are often limited in scope (e.g. to syntax rather than semantics) or require precise guidance from the user. One reason is that automating the search for patches [6], refactorings, optimisations [11], proofs [12], and other routine programming tasks is prohibitively expensive due to the size of the search space.

We investigate the use of machine learning techniques to restrict such search spaces more intelligently than uninformed local search, specifically the use of *clustering* to aid *theory exploration* (TE) in discovering program equivalences.

2. BACKGROUND

2.1 Theory Exploration

Our work builds on the existing TE system QuickSpec[4], which finds equivalent expressions in the Haskell programming language [10], built from a user-provided *signature* of constants and variables. For example, given a signature of the list-manipulating functions `reverse`, `length` and `append`, plus two list variables `x` and `y`, QuickSpec will discover that `reverse (reverse x)` is equivalent to `x`; that `length (reverse x)` is equivalent to `length x`; that `length (append x y)` is equivalent to `length (append y x)`; and so on. These are discovered by enumerating all well-typed combinations of the signature's constants and variables, then randomly instantiating the variables [2] and comparing the resulting closed terms. Any expressions which remain indistiguishable after hundreds of instantiations are conjectured to be equivalent and, after removing redundancies, are produced as output. These conjectures can be sent through a theorem prover [3] and used to inform tests, optimisations, refactoring, verification, or simply to help the programmer learn more about the code.

Due to its exponential complexity, this enumerating procedure is limited to producing small expressions from signatures with only a few elements. Such small signatures give little chance for serendipitous discoveries, which undermines the algorithm's potential to present programmers with new, unexpected information.

2.2 Recurrent Clustering

To reduce the user's need to cherry-pick signatures, our approach accepts a large signature (e.g. a complete Haskell package) and uses statistical machine learning methods to select smaller signatures automatically, using a *recurrent clustering* method inspired by those of ML4PG [9] and ACL2(ml) [7].

Recurrent clustering attempts to cluster expressions based on their Abstract Syntax Trees (ASTs), an example of which is shown in Figure 1. First the recursively-structured ASTs are transformed into vectors of fixed length, by truncating the tree structure and listing the node labels in breadth-first order.

These labels are then turned into *features* (real numbers), using the function ϕ which replaces keywords with fixed values and local variables with context-dependent values (known as *de Bruijn indices* [5]). To replace global variables, ϕ first performs another round of clustering (hence the name "recurrent"), without including the current expression; each global variable is replaced by its cluster index found by this "inner" round of clustering.

These recursive calls to the clustering algorithm (we use a standard *k-means* implementation) allow expressions to be compared in a principled way: the similarity of expressions depends on the similarity of the expressions they reference, and so on recursively. In practice, this recursive algorithm can be implemented in an iterative way, by accumulating the set of expressions to cluster in topological order of their dependency graph.

Figure 1 shows the AST for the following Haskell function `odd`, which determines if a Peano numeral is odd:

```
odd  Z     = False
odd  (S n) = even n

even Z     = True
even (S n) = odd n
```

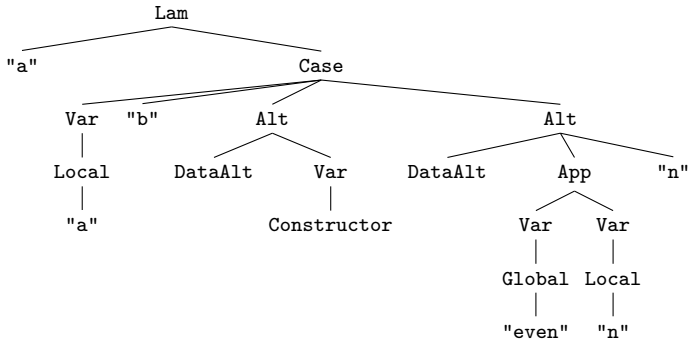

Figure 1: AST for the `odd` function. The variable names `a`, `b`, etc. are chosen arbitrarily.

The `odd` function references four global variables: `Z`, `S`, `False` and `even`. Our algorithm doesn't yet distinguish between data constructors, so the only global reference we replace is `even`. Likewise, the definition of `even` references `Z`, `S`, `True` and `odd`. For mutual recursion like this, there is no valid topological order, so we use a sentinel value as the feature. These feature vectors are sent through a k-means clustering algorithm, using the WEKA machine learning library. For example, the feature vector for `odd` (padded to 6 labels for each level of the tree) will begin:

$$(\phi(\texttt{Lam}), 0, 0, 0, 0, 0, \phi(\texttt{"a"}), \phi(\texttt{Case}), 0, 0, 0, 0, \phi(\texttt{Var}), \phi(\texttt{"b"}), \ldots$$

3. IMPLEMENTATION

We obtain ASTs from Haskell projects using a bespoke plugin for the GHC compiler. From these ASTs, we extract type information which is used to append variables to the signature; and dependency information, which is used to topologically sort the clustering rounds. We then invoke our iterative algorithm, which alternates between feature extraction and clustering until all elements of the signature have been clustered. Each cluster is converted into a QUICKSPEC signature, by extracting those functions which a) have an argument type which we can randomly generate and b) have an output type which we can compare. QUICKSPEC is invoked on each signature, and the resulting sets of conjectures are combined.

4. RESULTS

We have developed a machine learning approach for analysing Haskell code, including a bespoke feature extraction method and a full implementation pipeline for turning Haskell packages into sets of equations.

5. DIFFICULTIES AND FUTURE WORK

The most difficult aspect of performing this exploration was obtaining real code in a usable format, due to the myriad edge-cases encountered. We make extensive use of Haskell's existing infrastructure, including the GHC compiler, the CABAL build system, the HACKAGE code repository and the NIX package manager. Unfortunately, some of these systems are rather monolithic, which makes it difficult to invoke particular algorithms, such as GHC's type class resolution, on their own. Whilst we have worked around these issues, e.g. using meta-programming, this introduces unnecessary latency, fragility and complexity.

Our investigation of recurrent clustering and theory exploration only scratches the surface of combining symbolic and statistical AI algorithms. As well as benchmarking our approach against other techniques, there are many similar opportunities to be explored, where the reasoning power of symbolic algorithms can be guided and supervised by statistical, data-driven processes.

6. REFERENCES

[1] Robert S Boyer and J Strother Moore. Proof-Checking, Theorem Proving, and Program Verification. Technical report, DTIC Document, 1983.

[2] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[3] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Automated Deduction–CADE-24*, pages 392–406. Springer, 2013.

[4] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In Gordon Fraser and Angelo Gargantini, editors, *Tests and Proofs*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer Berlin Heidelberg, 2010.

[5] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[6] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, 2009.

[7] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-Pattern Recognition and Lemma Discovery in ACL2. *Lecture Notes in Computer Science*, pages 389–406, 2013.

[8] S. C. Johnson. Lint, a C Program Checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[9] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine Learning in Proof General: Interfacing Interfaces. In Cezary Kaliszyk and Christoph Lüth, editors, *UITP*, volume 118 of *EPTCS*, pages 15–41, 2013.

[10] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[11] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310. ACM, 2016.

[12] Dan Rosén. Proving equational Haskell properties using automated theorem provers. Master's thesis, University of Gothenburg, Sweden, 2012.