Improving Haskell Theory Exploration

Chris Warburton

University of Dundee,
http://tocai.computing.dundee.ac.uk

July 25, 2016

Abstract

Theory Exploration is a promising approach to improving the quality and understanding of software. It extends previously existing methods available through testing, in languages which are amenable to formal analysis such as those based on pure functional programming. Current theory exploration techniques are limited by their use of exponential time algorithms, which although thorough are ultimately limited to finding simple properties of small systems. We propose a more powerful approach, which uses machine learning algorithms to intelligently choose which parts of a system to explore based on their similarity, hence focusing its efforts on areas which are most likely to lead to discoveries.

1 Introduction

As computers and software become more sophisticated, and as our reliance on them increases, the importance of *understanding*, *predicting* and *verifying* these systems grows; which is undermined by their ever-increasing complexity. The *functional programming* paradigm has been proposed for addressing these issues [32], by constructing programs which are more amenable to mathematical analysis. For example, in pure functional programming all values are *immutable*: defined once and never changed. Hence there is no way for a value to be altered between the point it is introduced and the point it is used, unlike in many *imperative* languages where we may have to search the whole program to ensure the value is not altered by any intermediate code. Similarly, the results of pure functions cannot depend on any state other than their arguments, and hence will always produce repeatable results. By making state implicit in this way, powerful type systems can be used to constrain the behaviour of programs, and to give a rich, composable structure to data.

Whilst use of pure functional programming languages, like Haskell and Idris, is relatively rare, their features are well suited to common software engineering practices like *unit testing*; where tasks are broken down into small, easily-specified "units", and tested in isolation for a variety of use-cases. Functional

ideas are thus spreading to mainstream software engineering in a more dilute form; seen, for example, in the recent inclusion of first-class functions in Java [34] and C++ [81].

Functional programming is also well suited to less-widespread practices, such as *property checking* (as popularised by QuickCheck) and *theorem proving*, which are promising methods for increasing confidence in software, yet can be prohibitively expensive. Here we investigate how the recent *theory exploration* approach can lower the effort required to pursue these goals, and in particular how machine learning techniques can mitigate the costs of the combinatorial algorithms involved.

Our contributions are:

1. The application of machine learning algorithms to theory exploration, for intelligently discovering interesting sub-sets of Haskell libraries, which are more tractable to explore.

2. A novel feature extraction method for transforming Haskell expressions into a form amenable to off-the-shelf learning algorithms.

3. An implementation of these feature extraction and theory exploration approaches.

4. A comparison of our methods with existing approaches, both for theory exploration in Haskell, and for machine learning in other languages.

We begin in §2 by providing a formal context for analysing Haskell expressions (§2.1) and describe the QuickSpec theory exploration system (§2.3). We give a brief overview of testing approaches and how they relate to Haskell (§2.2), as well as the machine learning approaches we are building on (§2.4.1). We discuss our contributions in more depth in §3, and provide implementation details in §4. A variety of related work is surveyed in §5, we briefly evaluate our implementations in §6 and give several potential directions for future research in §7.

2 Background

2.1 Haskell

We decided to focus on theory exploration in the Haskell programming language as it has mature, state-of-the-art implementations (QuickSpec [15] and HipSpec [14]). This is evident from the fact that the state-of-the-art equivalent for Isabelle/HOL, the Hipster [36] system, is actually implemented by translating to Haskell and invoking HipSpec.

Haskell is well-suited to programming language research; indeed, this was a goal of the language's creators [54]. Like most *functional* programming languages, Haskell builds upon λ-calculus, with extra features such as a strong type system and "syntactic sugar" to improve readability. For simplicity, we will focus

$$
\begin{aligned}
\mathit{expr} &\rightarrow \texttt{Var}\ \mathit{id} \\
&\mid \texttt{Lit}\ \mathit{literal} \\
&\mid \texttt{App}\ \mathit{expr}\ \mathit{expr} \\
&\mid \texttt{Lam}\ \mathcal{L}\ \mathit{expr} \\
&\mid \texttt{Let}\ \mathit{bind}\ \mathit{expr} \\
&\mid \texttt{Case}\ \mathit{expr}\ \mathcal{L}\ [\mathit{alt}] \\
&\mid \texttt{Type} \\
\mathit{id} &\rightarrow \texttt{Local}\ \mathcal{L} \\
&\mid \texttt{Global}\ \mathcal{G} \\
&\mid \texttt{Constructor}\ \mathcal{D} \\
\mathit{literal} &\rightarrow \texttt{LitNum}\ \mathcal{N} \\
&\mid \texttt{LitStr}\ \mathcal{S} \\
\mathit{alt} &\rightarrow \texttt{Alt}\ \mathit{altcon}\ \mathit{expr}\ [\mathcal{L}] \\
\mathit{altcon} &\rightarrow \texttt{DataAlt}\ \mathcal{D} \\
&\mid \texttt{LitAlt}\ \mathit{literal} \\
&\mid \texttt{Default} \\
\mathit{bind} &\rightarrow \texttt{NonRec}\ \mathit{binder} \\
&\mid \texttt{Rec}\ [\mathit{binder}] \\
\mathit{binder} &\rightarrow \texttt{Bind}\ \mathcal{L}\ \mathit{expr}
\end{aligned}
$$

Where: $\mathcal{S}$ = string literals
$\mathcal{N}$ = numeric literals
$\mathcal{L}$ = local identifiers
$\mathcal{G}$ = global identifiers
$\mathcal{D}$ = constructor identifiers

Figure 1: Simplified syntax of GHC Core in BNF style. [] and (,) denote repetition and grouping, respectively.

```
data Nat = Z
         | S Nat

plus :: Nat -> Nat -> Nat
plus    Z  y = y
plus (S x) y = S (plus x y)

mult :: Nat -> Nat -> Nat
mult    Z  y = Z
mult (S x) y = plus y (mult x y)

odd :: Nat -> Bool
odd    Z  = False
odd (S n) = even n

even :: Nat -> Bool
even    Z  = True
even (S n) = odd n
```

Figure 2: A Haskell datatype for Peano numerals with some simple arithmetic functions, including mutually-recursive definitions for `odd` and `even`. `Bool` is Haskell's built in boolean type, which can be regarded as `data Bool = True | False`.

on an intermediate representation of the GHC compiler, known as *GHC Core*, rather than the relatively large and complex syntax of Haskell proper. Core is based on System F_C, but for our machine learning purposes we are mostly interested in its syntax; for a more thorough treatment of System F_C and its use in GHC, see [77, Appendix C].

The sub-set of Core we consider is shown in Figure 1; compared to the full language [1] our major restriction is to ignore type hints (such as explicit casts, and differences between types/kinds/coercions). For brevity, we also omit several other forms of literal (machine words of various sizes, individual characters, etc.), as their treatment is similar to those of strings and numerals. We will use quoted strings to denote names and literals, e.g. `Local "foo"`, `Global "bar"`, `Constructor "Baz"`, `LitStr "quux"` and `LitNum "42"`, and require only that they can be compared for equality.

Figure 2 shows some simple Haskell function definitions, along with a custom datatype for Peano numerals. The translation to our Core syntax is routine, and shown in Figure 3. Although the Core is more verbose, we can see that similar structure in the Haskell definitions gives rise to similar structure in the Core; for example, the definitions of `odd` and `even` are identical in both languages, except for the particular identifiers used. It is this close correspondence which

[1] As of GHC version 7.10.2, the latest at the time of writing.

plus

```
Lam "a" (Lam "y" (Case (Var (Local "a"))
                       "b"
                       (Alt (DataAlt "Z") (Var (Local "y")))
                       (Alt (DataAlt "S") (App (Var (Constructor "S"))
                                               (App (App (Var (Global "plus"))
                                                         (Var (Local  "x")))
                                                    (Var (Local "y"))))
                                          "x")))
```

mult

```
Lam "a" (Lam "y" (Case (Var (Local "a"))
                       "b"
                       (Alt (DataAlt "Z") (Var (Constructor "Z")))
                       (Alt (DataAlt "S") (App (App (Var (Global "plus"))
                                                    (Var (Local  "y")))
                                               (App (App (Var (Global "mult"))
                                                         (Var (Local  "x")))
                                                    (Var (Local  "y"))))
                                          "x")))
```

odd

```
Lam "a" (Case (Var (Local "a"))
              "b"
              (Alt (DataAlt "Z") (Var (Constructor "False")))
              (Alt (DataAlt "S") (App (Var (Global "even"))
                                      (Var (Local  "n")))
                                 "n"))
```

even

```
Lam "a" (Case (Var (Local "a"))
              "b"
              (Alt (DataAlt "Z") (Var (Constructor "True")))
              (Alt (DataAlt "S") (App (Var (Global "odd"))
                                      (Var (Local  "n")))
                                 "n"))
```

Figure 3: Translations of functions in Figure 2 into the Core syntax of Figure 1. Notice the introduction of explicit λ abstractions (`Lam`) and the use of `Case` to represent piecewise definitions. Fresh variables are chosen arbitrarily as `"a"`, `"b"`, etc.

allows us to analyse Core expressions in place of their more complicated Haskell source.

Note that we exclude representations for type-level entities, including datatype definitions like that of `Nat`. GHC can represent these, but in this work we only consider reducible expressions (i.e. value-level bindings of the form `f a b ... = ...`).

2.2 QuickCheck

Although unit testing is the de facto industry standard for quality assurance in non-critical systems, the level of confidence it provides is rather low, and totally inadequate for many (e.g. life-) critical systems. To see why, consider the following Haskell function, along with some unit tests:

```
factorial 0 = 1
factorial n = n * factorial (n-1)

fact_base      = factorial 0 == factorial 1
fact_increases = factorial 3 <= factorial 4
fact_div       = factorial 4 == factorial 5 `div` 5
```

The intent of the function is to map an input n to an output $n!$. The tests check a few properties of the implementation, including the base case, that the function is monotonically increasing, and a relationship between adjacent outputs. However, these tests will *not* expose a serious problem with the implementation: it diverges on half of its possible inputs!

All of Haskell's built-in numeric types allow negative numbers, which this implementation doesn't take into account. Whilst this is a rather trivial example, it highlights a common problem: unit tests are insufficient to expose incorrect assumptions. In this case, our assumption that numbers are positive has caused a bug in the implementation *and* limited the tests we've written.

If we do manage to spot this error, we might capture it in a *regression test* and update the definition of `factorial` to handle negative numbers, e.g. by taking their absolute value:

```
factorial 0 = 1
factorial n = let nPos = abs n
              in nPos * factorial (nPos - 1)

fact_neg = factorial 1 == factorial (-1)
```

However, this is *still* not enough, since this function will also accept fractional values[2], which will also cause it to diverge. Clearly, by choosing what to test we are biasing the test suite towards those cases we've already taken into account, whilst neglecting the problems we did not expect.

[2]Since we only use generic numeric operations, the function will be polymorphic with a type of the form `forall t. Num t => t -> t`, where `Num t` constrains the type variable `t` to be numeric. In fact, Haskell will infer extra constraints such as `Eq t` since we have used `==` in the unit tests.

Haskell offers a partial solution to this problem in the form of *property checking*. Tools such as QuickCheck separate tests into three components: a *property* to check, which unlike a unit test may contain *free variables*; a source of values to instantiate these free variables; and a stopping criterion. Here is how we might restate our unit tests as properties:

```
fact_base        = factorial 0 == factorial 1
fact_increases n = factorial n <= factorial (n+1)
fact_div       n = factorial n == factorial (n+1) `div` (n+1)
fact_neg       n = factorial n == factorial (-n)
```

The free variables (all called `n` in this case) are abstracted as function parameters; these parameters are implicitly *universally quantified*, i.e. we've gone from a unit test asserting $factorial(3) \leq factorial(4)$ to a property asserting $\forall n, factorial(n) \leq factorial(n+1)$. Notice that unit tests like `fact_base` are valid properties; they just assert rather weak statements.

To check these properties, QuickCheck treats closed terms (like `fact_base`) just like unit tests: pass if they evaluate to `True`, fail otherwise. For open terms, a random selection of values are generated and passed in via the function parameter; the results are then treated in the same way as closed terms. The default stopping criterion for QuickCheck (for each test) is when a single generated test fails, or when 100 generated tests pass.

The ability to state *universal* properties in this way avoids some of the bias we encountered with unit tests. In the `factorial` example, this manifests in two ways:

- QuickCheck cannot test polymorphic functions; they must be *monomorphised* first (instantiated to a particular concrete type). This is a technical limitation, since QuickCheck must know which type of values to generate, but in our example it would bring the issue with fractional values to our attention.
- The generators used by QuickCheck depend only on the *type* of value they are generating: since `Int` includes positive and negative values, the `Int` generator will output both. This will expose the problem with negative numbers, which we weren't expecting.

Property checking is certainly an improvement over unit testing, but the problem of tests being biased towards expected cases remains, since we are manually specifying the properties to be checked.

We can reduce this bias further through the use of *theory exploration* tools, such as QuickSpec and HipSpec. These programs *discover* properties of a "theory" (e.g. a library), through a combination of brute-force enumeration, random testing and (in the case of HipSpec) automated theorem proving.

2.3 Theory Exploration

In this work we consider the problem of *(automated) theory exploration*, which includes the ability to *generate* conjectures about code, to *prove* those conjec-

tures, and hence output *novel* theorems without guidance from the user. The method of conjecture generation is a key characteristic of any theory exploration system, although all existing implementations rely on brute force enumeration to some degree.

We focus on QuickSpec [15], which conjectures equations about Haskell code (these may be fed into another tool, such as HipSpec, for proving). These conjectures are arrived at through the following stages:

1. Given a typed signature Σ and set of variables V, QuickSpec generates a list *terms* containing the constants (including functions) from Σ, the variables from V and type-correct function applications $f(x)$, where f and x are elements of *terms* . To ensure the list is finite, function applications are only nested up to a specified depth (by default, 3).

2. The elements of *terms* are grouped into equivalence classes, based on their type.

3. Each variable is instantiated to a particular value, generated randomly by QuickCheck.

4. For each class, the members are compared (using a pre-specified function, such as equality ==) to see if these instantiations have caused an observable difference between members. If so, the class is split up to separate such distinguishable members.

5. The previous steps of variable instantiation and comparison are repeated until the classes stabilise (i.e. no differences have been observed for some specified number of repetitions).

6. A set of equations are then conjectured, relating each class's members.

Such *conjectures* can be used in several ways: they can be simplified for direct presentation to the user (by removing any equation which can be derived from the others by rewriting), sent to a more rigorous system like HipSpec or Hipster for proving, or even serve as a background theory for an automated theorem prover [14].

As an example, we can consider a simple signature containing the expressions from Figure 2:

$$\Sigma_{\texttt{Nat}} = \{\texttt{Z}, \texttt{S}, \texttt{plus}, \texttt{mult}, \texttt{odd}, \texttt{even}\}$$

Together with a set of variables, say $V_{\texttt{Nat}} = \{a, b, c\}$, QuickSpec's enumeration will resemble the following:

$$\begin{aligned} \mathit{terms}_{\texttt{Nat}} = [&\texttt{Z},\ \texttt{S},\ \texttt{plus},\ \texttt{mult},\ \texttt{odd},\ \texttt{even},\ a,\ b,\ c,\ \texttt{S Z},\ \texttt{S}\ a,\ \texttt{S}\ b,\\ &\texttt{S}\ c,\ \texttt{plus Z},\ \texttt{plus}\ a,\ \ldots] \end{aligned}$$

```
          plus a b = plus b a
          plus a Z = a
plus a (plus b c) = plus b (plus a c)
          mult a b = mult b a
          mult a Z = Z
mult a (mult b c) = mult b (mult a c)
     plus a (S b) = S (plus a b)
     mult a (S b) = plus a (mult a b)
mult a (plus b b) = mult b (plus a a)
         odd (S a) = even a
   odd (plus a a) = odd Z
  odd (times a a) = odd a
       even (S a) = odd a
  even (plus a a) = even Z
 even (times a a) = even a
plus (mult a b) (mult a c) = mult a (plus b c)
```

Figure 4: Equations conjectured by QuickSpec for the functions in Figure 2; after simplification.

Notice that functions such as `plus` and `mult` are valid terms, despite not being applied to any arguments. In addition, Haskell curries functions, so these binary functions will be treated as unary functions which return unary functions. This is required as the construction of *terms* applies functions to one argument at a time.

These terms will be grouped into five classes, one each for `Nat`, `Nat -> Nat`, `Nat -> Nat -> Nat`, `Nat -> Bool` and `Bool`. As the variables a, b and c are instantiated to various randomly-generated numbers, these equivalence classes will be divided, until eventually the equations in Figure 4 are conjectured.

Although complete, this enumeration approach is wasteful: many terms are unlikely to appear in theorems, which requires careful choice by the user of what to include in the signature. Here we know that addition and multiplication are closely related, and hence obey many algebraic laws. Our machine learning technique aims to predict these kinds of relations between functions, so we can create small signatures which nevertheless have the potential to give rise to many equations.

QuickSpec (and HipSpec) is also compatible with Haskell's existing testing infrastructure, such that an invocation of `cabal test` can run these tools alongside more traditional QA tools like QuickCheck, HUnit and Criterion.

In fact, there are similarities between the way a TE system like QuickSpec can generalise from checking *particular* properties to *inventing* new ones, and the way counterexample finders like QuickCheck can generalise from testing *particular* expressions to *inventing* expressions to test. One of our aims is to understand the implications of this generalisation, the lessons that each can

learn from the other's approach to term generation, and the consequences for testing and QA in general.

2.4 Clustering

Our approach to scaling up QuickSpec takes inspiration from two sources. The first is relevance filtering, which makes expensive algorithms used in theorem proving more practical by limiting the size of their inputs. We describe this approach in more details in §5.3. Relevance filtering is a practical tool which has existing applications in software, such as the *Sledgehammer* component of the Isabelle/HOL theorem prover.

Despite the idea's promise, we cannot simply invoke existing relevance filter algorithms in our theory exploration setting. The reason is that relevance filtering is a supervised learning method, i.e. it would require a distinguished expression to compare everything against. Theory exploration does not have such a distinguished expression; instead, we are interested in relationships between *any* terms generated from a signature, and hence we must consider the relevance of *all terms* to *all other terms*.

A natural fit for this task is *clustering*, which attempts to group similar inputs together in an unsupervised way. Based on their success in discovering relationships and patterns between expressions in Coq and ACL2 (in the ML4PG and ACL2(ml) tools respectively), we hypothesise that clustering methods can fulfil the role of relevance filters for theory exploration: intelligently breaking up large signatures into smaller ones more amenable to brute force enumeration, such that related expressions are explored together.

2.4.1 Feature Extraction

Before describing clustering in detail, we must introduce the idea of *feature extraction*. This is the conversion of "raw" input data, such as audio, images or (in our case) Core expressions, into a form more suited to machine learning algorithms. By pre-processing our data in this way, we can re-use the same "off-the-shelf" machine learning algorithms in a variety of domains.

We use a standard representation of features as a *feature vector* of numbers $\mathbf{x} = (x_1, \ldots, x_d)$ where $x_i \in \mathbb{R}$ (we use **bold face** to represent vectors, including feature vectors). [3] For learning purposes this has some important advantages over raw expressions:

- All of our feature vectors will be the same size, i.e. they will all have length (or *dimension*) d. Many machine learning algorithms only work with inputs of a uniform size; feature extraction allows us to use these algorithms in domains where the size of each input is not known, may vary or may even be unbounded. For example, element-wise comparison of feature vectors is trivial (compare the ith elements for $1 \leq i \leq d$); for

[3] In fact, practical implementations will use an approximate format such as IEEE 754 floating point numbers.

expressions this is not so straightforward, as their nesting may give rise to very different shapes.

- Unlike our expressions, which are discrete, we can continuously transform one feature vector into another. This enables many powerful machine learning algorithms to be used, such as those based on *gradient descent* or, in our case, arithmetic means.
- Feature vectors can be chosen to represent the relevant information in a more compressed form than the raw data; for example, we might replace verbose, descriptive identifiers with sequential numbers. This reduces the input size of the machine learning problem, improving efficiency.

2.4.2 K-Means

Clustering is an unsupervised machine learning task for grouping n data points using a similarity metric. There are many variations on this theme, but in our case we make the following choices:

- For simplicity, we use the "rule of thumb" given in [53, pp. 365] to fix the number of clusters at $k = \lceil \sqrt{\frac{n}{2}} \rceil$.
- Data points will be d-dimensional feature vectors, as defined above.
- We will use euclidean distance (denoted e) as our similarity metric.
- We will use *k-means* clustering, implemented by Lloyd's algorithm [49].

This is a standard setup, supported by off-the-shelf tools; in particular we use the implementation provided by Weka [31], due to its use by ML4PG, which makes our results more easily comparable.

Since k-means is iterative, we will use function notation to denote time steps, so $x(t)$ denotes the value of x at time t. We denote the clusters as C^1 to C^k. As the name suggests, k-means uses the mean value of each cluster, which we denote as $\mathbf{m}^1$ to $\mathbf{m}^k$, hence:

$$m^i_j(t) = \overline{C^i_j}(t) = \frac{\sum_{\mathbf{x} \in C^i(t)} x_j}{|C^i(t)|} \quad \text{for } t > 0$$

Before k-means starts, we must choose *seed* values for $\mathbf{m}^i(0)$. Many methods have been proposed for choosing these values [2]. For simplicity, we will choose values randomly from our data set S; this is known as the Forgy method.

The elements of each cluster $C^i(t)$ are those data points closest to the mean value at the previous time step:

$$C^i(t) = \{\mathbf{x} \in S \mid i = \operatorname*{argmin}_j e(\mathbf{x}, \mathbf{m}^j(t-1))\} \quad \text{for } t > 0$$

As t increases, the clusters C^i move from their initial location around the "seeds", to converge on a local minimum of the "within-cluster sum of squared error" objective:

$$\underset{C}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in C^i} e(\mathbf{x}, \mathbf{m}^i)^2 \tag{1}$$

3 Contributions

3.1 Recurrent Clustering

We adapt the methodology of *recurrent clustering* proposed in [27, 28], and suggest a new recurrent clustering and feature extraction algorithm for Haskell, which we then evaluate as a relevance filter technique for theory exploration. As a clustering algorithm, the aim of recurrent clustering is to identify similarities in a set of data points (in our case, Core expressions). Its distinguishing characteristic is to *combine* feature extraction and clustering into a single recursive algorithm (shown as Algorithm 1), which goes beyond a simple syntactic characterisation, to allow the features of an expression to depend on those it references. Here we describe our approach to recurrent clustering and compare its similarity and differences to those of ML4PG and ACL2(ml).

We consider our algorithm in two stages: the first transforms the nested structure of expressions into a flat feature vector representation; the second converts the discrete symbols of Core syntax into features (real numbers), which we will denote as the function ϕ.

3.1.1 Expressions to Vectors

Our recurrent clustering algorithm makes use of the k-means algorithm described in §2.4.2, which considers the elements of a feature vector to be *orthogonal*. Hence we must ensure that similar expressions not only give rise to similar numerical values, but crucially that these values appear *at the same position* in the feature vectors. Since different patterns of nesting can alter the "shape" of expressions, simple traversals (breadth-first, depth-first, post-order, etc.) may cause features from equivalent sub-expressions to be mis-aligned. For example, consider the following expressions, which represent pattern-match clauses with different patterns but the same body (`Var (Local "y")`):

$$\begin{aligned} X = {} & \texttt{Alt}\ (\texttt{DataAlt "C"})\ (\texttt{Var}\ (\texttt{Local "y"})) \\ Y = {} & \texttt{Alt Default}\qquad\quad\ (\texttt{Var}\ (\texttt{Local "y"})) \end{aligned}$$

If we traverse these expressions in breadth-first order, converting each token to a feature using ϕ and padding to the same length with 0, we would get the following feature vectors:

$$\begin{aligned} breadthFirst(X) = &\ (\phi(\texttt{Alt}),\ \phi(\texttt{DataAlt}),\ \phi(\texttt{Var}),\ \phi(\texttt{"C"}),\quad \phi(\texttt{Local}),\ \phi(\texttt{"y"})) \\ breadthFirst(Y) = &\ (\phi(\texttt{Alt}),\ \phi(\texttt{Default}),\ \phi(\texttt{Var}),\ \phi(\texttt{Local}),\ \phi(\texttt{"y"}),\quad 0\quad\ ) \end{aligned}$$

Here the features corresponding to the common sub-expression `Local "y"` are misaligned, such that only $\frac{1}{3}$ of features are guaranteed to match (others may match by coincidence, depending on ϕ). These feature vectors might be deemed very dissimilar during clustering, despite the intuitive similarity of the expressions X and Y from which they derive.

If we were to align these features optimally, by padding the fourth column rather than the sixth, then $\frac{2}{3}$ of features would be guaranteed to match, making the similarity of the vectors more closely match our intuition and depend less on coincidence.

The method we use to "flatten" expressions, described below, is a variation of breadth-first traversal which pads each level of nesting to a fixed size c (for *columns*). This doesn't guarantee alignment, but it does prevent mis-alignment from accumulating across different levels of nesting. Our method would align these features into the following vectors, if $c = 2$: [4]

$$\begin{aligned} featureVec(X) = &\ (\phi(\texttt{Alt}),\ 0,\ \phi(\texttt{DataAlt}),\ \phi(\texttt{Var}),\ \phi(\texttt{"C"}),\quad \phi(\texttt{Local}),\ \phi(\texttt{"y"}),\ 0) \\ featureVec(Y) = &\ (\phi(\texttt{Alt}),\ 0,\ \phi(\texttt{Default}),\ \phi(\texttt{Var}),\ \phi(\texttt{Local}),\ 0,\quad\ \phi(\texttt{"y"}),\ 0) \end{aligned}$$

Here $\frac{1}{2}$ of the original 6 features align, which is more than $breadthFirst$ but not optimal. Both vectors have also been padded by an extra 2 zeros compared to $breadthFirst$; raising their alignment to $\frac{5}{8}$.

To perform this flattening we first transform the nested tokens of an expression into a *rose tree* of features, using the $toTree$ function shown in Figure 5. We follow the presentation in [11] and define rose trees recursively as follows: T is a rose tree if $T = (f, T_1, \ldots, T_{n_T})$, where $f \in \mathbb{R}$ and T_i are rose trees. T_i are the *sub-trees* of T and f is the *feature at* T. n_T may differ for each (sub-) tree; trees where $n_T = 0$ are *leaves*. The results are illustrated in Figure 6a.

These rose trees are then turned into matrices, as shown in Figure 6b, by gathering the features of adjacent (sub-) trees at each level of nesting, and concatenating them together (written as $+\!\!+$):

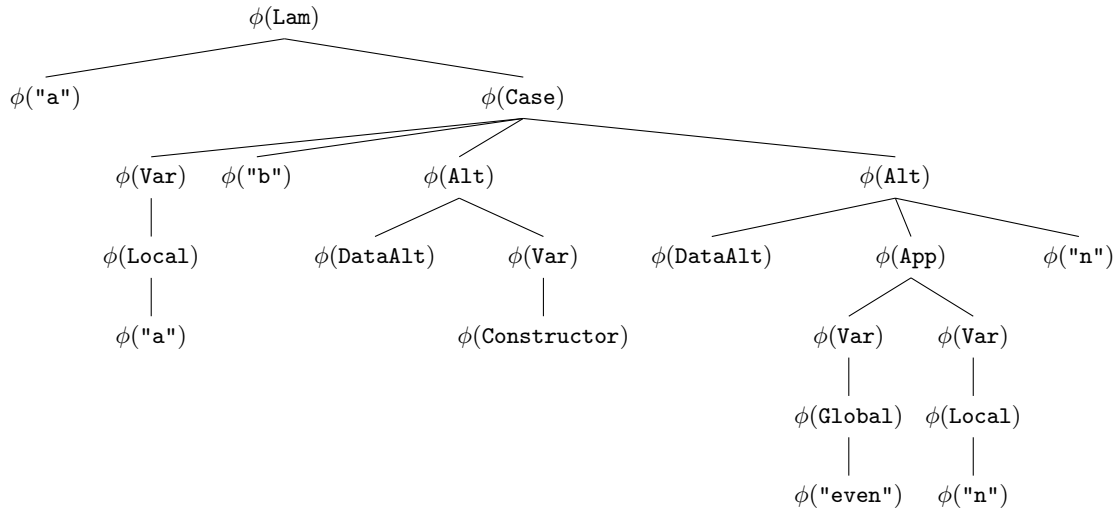
$$level(l, (f, T_1, \ldots, T_{n_T})) = \begin{cases} (f) & \text{if } l = 1 \\ level(l-1, T_1) +\!\!+ \ \ldots \ +\!\!+ \ level(l-1, T_{n_T}) & \text{if } l > 1 \end{cases} \tag{2}$$

Given a rose tree t we can define its matrix $\mathbf{M}$ by $\mathbf{M}_i = pad(level(i, t))$, where pad either truncates or appends zeros, until the row has a fixed length c. We also truncate/pad the number of rows to match a fixed number r. This

[4] In fact, the $toTree$ function would ignore the constructor identifier `"C"` and never produce the feature ϕ(`"C"`). However, this example is still accurate in terms of laying out the features as given.

$$
toTree(e) = \begin{cases}
(\phi(\texttt{Var}), toTree(e_1)) & \text{if } e = \texttt{Var } e_1 \\
(\phi(\texttt{Lit}), toTree(e_1)) & \text{if } e = \texttt{Lit } e_1 \\
(\phi(\texttt{App}), toTree(e_1), toTree(e_2)) & \text{if } e = \texttt{App } e_1\ e_2 \\
(\phi(\texttt{Lam}), toTree(e_1)) & \text{if } e = \texttt{Lam } l_1\ e_1 \\
(\phi(\texttt{Let}), toTree(e_1), toTree(e_2)) & \text{if } e = \texttt{Let } e_1\ e_2 \\
(\phi(\texttt{Case}), toTree(e_1), toTree(a_1), \dots) & \text{if } e = \texttt{Case } e_1\ l_1\ a_1\ \dots \\
(\phi(\texttt{Type})) & \text{if } e = \texttt{Type} \\
(\phi(\texttt{Local}), (\phi(l_1))) & \text{if } e = \texttt{Local } l_1 \\
(\phi(\texttt{Global}), (\phi(g_1))) & \text{if } e = \texttt{Global } g_1 \\
(\phi(\texttt{Constructor})) & \text{if } e = \texttt{Constructor } d_1 \\
(\phi(\texttt{LitNum})) & \text{if } e = \texttt{LitNum } n_1 \\
(\phi(\texttt{LitStr})) & \text{if } e = \texttt{LitStr } s_1 \\
(\phi(\texttt{Alt}), toTree(e_1), toTree(e_2)) & \text{if } e = \texttt{Alt } e_1\ e_2\ l_1\ \dots \\
(\phi(\texttt{DataAlt})) & \text{if } e = \texttt{DataAlt } g_1 \\
(\phi(\texttt{LitAlt}), toTree(e_1)) & \text{if } e = \texttt{LitAlt } e_1 \\
(\phi(\texttt{Default})) & \text{if } e = \texttt{Default} \\
(\phi(\texttt{NonRec}), toTree(e_1)) & \text{if } e = \texttt{NonRec } e_1 \\
(\phi(\texttt{Rec}), toTree(e_1), \dots) & \text{if } e = \texttt{Rec } e_1\ \dots \\
(\phi(\texttt{Bind}), toTree(e_1)) & \text{if } e = \texttt{Bind } l_1\ e_1
\end{cases}
$$

Figure 5: Transforming Core expressions of Figure 1 to rose trees. The recursive definition is mostly routine; each repeated element (shown as ...) has an example to indicate their handling, e.g. for `Rec` we apply $toTree$ to each e_i. We ignore values of $\mathcal{D}$, since constructors have no internal structure for us to compare; they can only be compared based on their types, which we do not currently support. We also ignore values from $\mathcal{S}$ and $\mathcal{N}$ as it simplifies our later definition of ϕ, and we conjecture that the effect on clustering real code is low.



(a) Rose tree for the expression `odd` from Figure 3. Each (sub-) rose tree is rendered with its feature at the node and sub-trees beneath.

$$\begin{bmatrix} \phi(\texttt{Lam}) & 0 & 0 & 0 & 0 & 0 \\ \phi(\texttt{"a"}) & \phi(\texttt{Case}) & 0 & 0 & 0 & 0 \\ \phi(\texttt{Var}) & \phi(\texttt{"b"}) & \phi(\texttt{Alt}) & \phi(\texttt{Alt}) & 0 & 0 \\ \phi(\texttt{Local}) & \phi(\texttt{DataAlt}) & \phi(\texttt{Var}) & \phi(\texttt{DataAlt}) & \phi(\texttt{App}) & \phi(\texttt{"n"}) \\ \phi(\texttt{"a"}) & \phi(\texttt{Constructor}) & \phi(\texttt{Var}) & \phi(\texttt{Var}) & 0 & 0 \\ \phi(\texttt{Global}) & \phi(\texttt{Local}) & 0 & 0 & 0 & 0 \\ \phi(\texttt{"even"}) & \phi(\texttt{"n"}) & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) Matrix generated from Figure 6a, padded to 6 columns. Each level of nesting in the tree corresponds to a row in the matrix.

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

(c) (Prefix of) the feature vector for `odd`, constructed by concatenating the rows of 6b. Ignoring padding, the features are in breadth-first order.

Figure 6: Feature extraction applied to the expression `odd` from Figure 3.

way, all expressions give rise to $r \times c$ matrices, where the left-most features at each each level are aligned.

Feature vectors are then simply the concatenation of matrix rows $\mathbf{M}_1 +\!\!+ \mathbf{M}_2 +\!\!+ \ldots +\!\!+ \mathbf{M}_r$, as shown in Figure 6c:

$$featureVec(e) = pad(level(1, toTree(e))) +\!\!+ \ldots +\!\!+ pad(level(r, toTree(e))) \tag{3}$$

3.1.2 Symbols to Features

We now define the function ϕ, which turns terminal symbols of Core syntax into features (real numbers). For known language features, such as $\phi(\texttt{Lam})$ and $\phi(\texttt{Case})$, we can enumerate the possibilities and assign a value to each, in a similar way to [27] in Coq. We use a constant α to separate these values from those of other tokens (e.g. identifiers), but the order is essentially arbitrary: [5]

$$\begin{array}{rrr}
\phi(\texttt{Alt}) = \alpha & \phi(\texttt{DataAlt}) = \alpha + 1 & \phi(\texttt{LitAlt}) = \alpha + 2 \\
\phi(\texttt{Default}) = \alpha + 3 & \phi(\texttt{NonRec}) = \alpha + 4 & \phi(\texttt{Rec}) = \alpha + 5 \\
\phi(\texttt{Bind}) = \alpha + 6 & \phi(\texttt{Let}) = \alpha + 7 & \phi(\texttt{Case}) = \alpha + 8 \\
\phi(\texttt{Local}) = \alpha + 9 & \phi(\texttt{Global}) = \alpha + 10 & \phi(\texttt{Constructor}) = \alpha + 11 \\
\phi(\texttt{Var}) = \alpha + 12 & \phi(\texttt{Lam}) = \alpha + 13 & \phi(\texttt{App}) = \alpha + 14 \\
\phi(\texttt{Type}) = \alpha + 15 & \phi(\texttt{Lit}) = \alpha + 16 & \phi(\texttt{LitNum}) = \alpha + 17 \\
\phi(\texttt{LitStr}) = \alpha + 18 &&
\end{array} \tag{4}$$

To encode *local* identifiers $\mathcal{L}$ we would like a quantity which gives equal values for α-equivalent expressions (i.e. renaming an identifier shouldn't affect the feature). To do this, the $toTree$ function maintains a *context* as it recurses through expressions (we elided this from Figure 5 for clarity). The context is a list of the local identifiers which are in scope, prepended as they are encountered. The context is initially empty, and only extended when $toTree$ calls itself recursively.

For example, when calculating $toTree(\texttt{Lam}\ i\ e)$ in context $\mathbf{x}$, we make the recursive call $toTree(e)$ in the context of $\mathbf{x}$ *prepended with* i (i.e. $(i) +\!\!+ \mathbf{x}$). Similar extensions of the context are performed in the cases of `Bind`, `Case`[6], `Alt` (which may introduce an arbitrary number of local identifiers) and `Let` (where identifiers are taken from occurrences of `Bind` in the first expression).

Well-formed Haskell declarations do not contain free variables (or, equivalently, free variables are encoded as global identifiers rather than local identifiers). Hence if we apply $toTree$ to (the Core expression corresponding to) such

[5] In [27], "similar" Gallina tokens like `fix` and `cofix` are grouped together to reduce redundancy; we do not group tokens, but we do put "similar" tokens close together, such as `Local` and `Global`.

[6] The $\mathcal{L}$ value in a `Case` expression is bound to the expression being matched against; a technical detail to preserve sharing.

declarations we are guaranteed that $l \in \mathcal{L}$ will appear in the context whenever we encounter $\phi(l)$. In which case we define:

$$\phi(l) = i + 2\alpha \quad \text{if } l \in \mathcal{L} \tag{5}$$

Where i is the index of the first occurrence of l in the context. This tells us how many binders a variable is nested inside, relative to the point it was introduced. This is invariant under renaming, as we wanted, and is known as the *de Bruijn index* of l. We again use α to separate these features from those of other constructs.

Since the $toTree$ function discards numerals, strings and constructor identifiers, the only remaining case is global identifiers $\mathcal{G}$. Since these are declared *outside* the body of an expression, we cannot perform the same indexing trick as we did for local identifiers. We also cannot directly encode the form of the identifiers, e.g. using a scheme like Gödel numbering, since this is essentially arbitrary and has no effect on their semantic meaning (referencing other expressions).

Instead, we use the approach taken in the latest versions of ML4PG and encode global identifiers *indirectly*, by looking up the expressions which they *reference*:

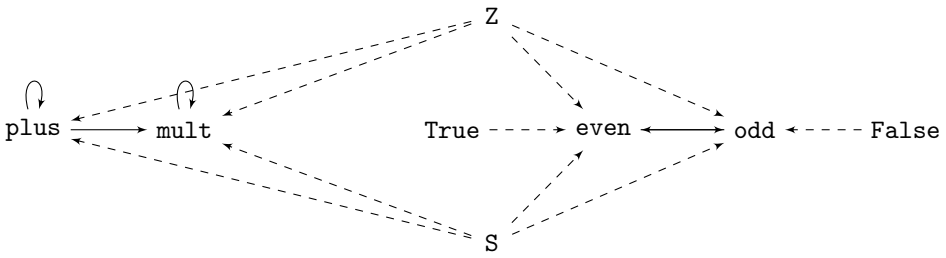
$$\phi(g \in \mathcal{G}) = \begin{cases} i + 3\alpha & \text{if } g \in C_i \\ f_{recursion} & \text{otherwise} \end{cases} \tag{6}$$

Where $\mathbf{C}$ are our clusters (in some arbitrary order). This is where the recurrent nature of the algorithm appears: to determine $\mathbf{C}$ we must perform k-means clustering *during* feature extraction; yet that clustering step, in turn, requires that we perform feature extraction.

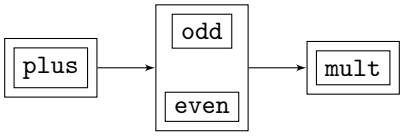
For this recursive process to be well-founded, we perform a topological sort on declarations based on their dependencies (the expressions they reference). In this way, we can avoid looking up expressions which haven't been clustered yet. To perform our sort, we construct a directed graph, where each declaration is a node and edges denote an "is referenced by" relation. An example dependency graph is given in Figure 7.

In a topological sort we say that $A < B$ for nodes A and B if we can reach B by following edges from A (if A and B cannot be reached from each other, their order is arbitrary). This is slightly complicated in Haskell (compared to Coq, for example), since general recursion is permitted and hence the dependency graph may contain cycles. To handle this, we use an algorithm such as Tarjan's [78] to produce a sorted list of *strongly connected components* (SCCs), where each SCC is a mutually-recursive sub-set of the declarations (as shown in Figure 7b). If an identifier cannot be found in any cluster, it must appear in the same SCC as the expression we are processing; hence we use the constant feature value $f_{recursion}$.

By working through the sorted list of SCCs, storing the features of each top-level expression as they are calculated, our algorithm can be computed *iteratively* rather than recursively, as shown in Algorithm 1.



(a) Dependency graph for Figure 3. Loops indicate recursive functions, double arrows indicate mutual recursion. Dashed lines show references to data constructors, which we do not consider in this work.



(b) One possible topological sorting of simply connected components for Figure 7a (ignoring constructors). `even` and `odd` are mutually recursive, neither can appear before the other, so they are grouped into one component and handled concurrently by Algorithm 1.

Figure 7: Sorting functions from Figure 3 into dependency order.

Algorithm 1 Recurrent clustering of Core expressions.

Require: List d contains SCCs of (identifier, expression) pairs, in dependency order.

```
1: procedure RecurrentCluster
2:     C ← []
3:     DB ← ∅
4:     for all scc in d do
5:         DB ← DB ∪ {(i, featureVec(e)) | (i, e) ∈ scc}
6:         C ← kMeans(DB)
       return C
```

As an example of this recurrent process, we can consider the Peano arithmetic functions from Figure 3. A valid topological ordering is given in Figure 7b, which can be our value for d (eliding Core expressions to save space):

$$d = [\{(\texttt{plus}, \dots)\}, \{(\texttt{odd}, \dots), (\texttt{even}, \dots)\}, \{(\texttt{mult}, \dots)\}]$$

We can then trace the execution of Algorithm 1 as follows:

- The first iteration through RecurrentCluster's loop will set $scc \leftarrow \{(\texttt{plus}, \dots)\}$.
- With $i = \texttt{plus}$ and e as its Core expression, calculating $featureVec(e)$ is straightforward; the recursive call $\phi(\texttt{plus})$ will become $f_{recursion}$ (since `plus` doesn't appear in $\mathbf{C}$).
- The call to $kMeans$ will produce $\mathbf{C} \leftarrow [\{\texttt{plus}\}]$, i.e. a single cluster containing `plus`.
- The next iteration will set $scc \leftarrow \{(\texttt{odd}, \dots), (\texttt{even}, \dots)\}$.
- With $i = \texttt{odd}$ and e as its Core expression, the call to `even` will result in $f_{recursion}$.
- Likewise for the call to `even` when $i = \texttt{odd}$.
- Since the feature vectors for `odd` and `even` will be identical, $kMeans$ will put them in the same cluster. To avoid the degenerate case of a single cluster, for this example we will assume that $k = 2$; in which case the other cluster must contain `plus`. Their order is arbitrary, so one possibility is $\mathbf{C} = [\{\texttt{odd}, \texttt{even}\}, \{\texttt{plus}\}]$.
- Finally `mult` will be clustered. The recursive call will become $f_{recursion}$ whilst the call to `plus` will become $2 + 3\alpha$, since $\texttt{plus} \in C_2$.
- Again assuming that $k = 2$, the resulting clusters will be $\mathbf{C} \leftarrow [\{\texttt{odd}, \texttt{even}\}, \{\texttt{plus}, \texttt{mult}\}]$.

Even in this very simple example we can see a few features of our algorithm emerge. For example, `odd` and `even` will always appear in the same cluster, since they only differ in their choice of constructor names (which are discarded by $toTree$) and recursive calls (which are replaced by $f_{recursion}$). A more extensive investigation of these features requires a concrete implementation, in particular to pin down values for the parameters such as r, c, $f_{recursion}$, α and so on.

3.1.3 Comparison

Our algorithm is most similar to that of ML4PG, as our transformation maps the elements of a syntax tree to distinct cells in a matrix. In contrast, the matrices produced by ACL2(ml) *summarise* the tree elements: providing, for

each level of the tree, the number of variables, nullary symbols, unary symbols, etc.

There are two major differences between our algorithm and that of ML4PG: mutual-recursion and types.

The special handling required for mutual recursion is discussed above (namely, topological sorting of expressions and the $f_{recursion}$ feature). Such handling is not present in ML4PG, since the Coq code it analyses must, by virtue of the language, be written in dependency order to begin with. Coq *does* have limited support for mutually-recursive functions, of the following form:

```
Fixpoint even n := match n with
                       | O   => true
                       | S m => odd m
                   end
    with odd  n := match n with
                       | O   => false
                       | S m => even m
                   end.
```

However, this is relatively uncommon and unsupported by ML4PG.

The more interesting differences come from our handling (or lack thereof) for types. Coq and ACL2 are at opposite ends of the typing spectrum, with the former treating types as first class entities of the language whilst the latter is untyped (or *unityped*). In both cases, we have a *single* language to analyse, by ML4PG and ACL2(ml) respectively.[7]

The situation is different for Haskell, where the type level is distinct from the value level, and there are strict rules for how they can influence each other. In particular, Haskell values can depend on types (via the type class mechanism) but types cannot depend on values.

In our initial approach, we restrict ourselves to the value level. This has several consequences:

- Although they are values, we cannot distinguish between data constructors, other than using exact equality. Hence they are discarded by $toTree$.
- Since Core uses a single `Lam` abstraction for both value- and type-level parameters, we cannot always distinguish between them. This can cause a function's Core arity to be greater than its Haskell arity.

There is certainly promise in including types in our analysis, by pairing every term with its type as in ML4PG. This will allow fine-grained distinction of expressions which are otherwise identical, especially data constructors. Integrating types into our algorithm, and extracting them from Core expressions, is hence left as future work.

[7] ML4PG can also analyse Coq's Ltac meta-language. Haskell has its own meta-language, Template Haskell, but here we only consider the regular Haskell which it generates.

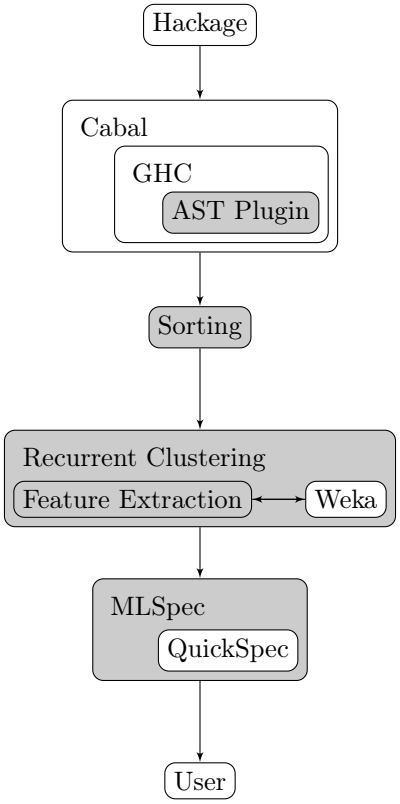


Figure 8: Components of the ML4HS theory exploration system. Custom components are shaded, arrows indicate data flow.

4 Implementation

We provide an implementation of our recurrent clustering algorithm in a tool called ML4HS, which consists of a loose collection of components shown in Figure 8. This arrangement makes it easy to swap out parts for experimentation. In the following, we describe the custom components in the order they appear in the diagram.

4.1 AST Plugin

The GHC compiler provides mechanisms for parsing Haskell source code and converting it to Core. It also includes a *renaming* transformation, which resolves global identifiers into a canonical form. This allows us to spot repeated use of a term, across multiple modules and packages, with a simple syntactic equality check.

Since we are interested in comparing definitions based on the terms they

reference, building our framework on top of GHC seems like a promising approach. Indeed, HipSpec already invokes GHC's API to obtain the definitions of Haskell functions, in order to transform them into a form suitable for ATP systems. However, our initial experiments showed that this technique is too fragile for use on many real Haskell projects.

This is due to many projects having a complex module structure, requiring particular GHC flags to be given, or using pre-processors such as `cpp` and Template Haskell to generate parts of their code. All of this complexity means that invoking GHC "manually" via its API is unlikely to obtain the definitions we require.

Thankfully there is one implementation detail which most Haskell packages agree on: the Cabal build system. All of the above complexities will be specified in a package's "Cabal file", such that the `cabal configure` and `cabal build` commands are very likely to work for most packages, without any extra effort. This shifted our focus to augmenting GHC and Cabal, such that definitions can be collected during the normal Haskell build process.

GHC provides a plugin mechanism for manipulating Core during a build, intended for optimisation passes, which we use to inspect definitions as they are being compiled. We provide a plugin called AstPlugin which emits a serialised version of each Core definition to the console (to satisfy the type system, it also implements a dummy "optimisation" which returns the Core unchanged).

Compared to Haskell, Core is a much simpler language and its representation is relatively stable compared to many existing representations of Haskell (which often change to support various language extensions). Three areas which make Core difficult to handle are:

Type variables: Parametric polymorphism (described in more detail in §5.1) can be thought of as values being parameterised by type-level objects. In System F, this is represented explicitly by a special abstraction form Λ, distinct from the λ used for values. Core only has one abstraction form, `Lam`, for both types and values. This alters function properties like arity.

Unified namespace: Haskell has distinct namespaces for values, types, data constructors, etc. Since Core does not make these distinctions, names may become ambiguous. For example, a type parameter `t` may be confused with a function argument `t`. To prevent this, overlapping namespaces are distinguished by prefices which are distinct from the available names; for example a type class constraint `Ord t` may give rise to a binder `Lam "$dOrd"` in Core, which is guaranteed not to conflict since this name would be invalid in Haskell. This causes difficulties when looking up names, as these prefixed forms do not easily map back to the Haskell source.

Violating encapsulation: Although Haskell allows names to be *private* to a module, when compiling Core we have full access to private definitions, as well as references to private names from within other definitions. Hence the definitions we receive from AstPlugin will include private values which we cannot import into a theory exploration tool.

In practice, we work around these issues with a post-processing stage: for each named definition appearing in the output of AstPlugin, we attempt to reference that name within the GHCi interpreter. Names with the above problems will cause an error, and are discarded.

The result of building a Haskell package with AstPlugin enabled is a database of Haskell definitions, similar in some respects to Hoogle [56]. Definitions are indexed by a combination of their package name, module name and binding name. The definitions themselves are s-expressions representing the Core AST, with non-local references replaced by a combination of package name, module name and binding name, which makes it trivial to look up references in the database. Each definition also has an associated arity and type, obtained during the post-processing step mentioned above.

4.2 Toplogical Sorting

As described in §3.1.2, we must topologically sort the output of AstPlugin in order for our recurrent clustering to be well-founded. Since our database keys (containing the package, module and binding names, as described above) match our representation of non-local references, it is simple to walk each syntax tree to obtain the set of references it makes. In addition, the resulting set of (identifier, list-of-referenced-identifiers) pairs exactly matches the (vertex, list-of-successor-vertices) format used to represent directed graphs by the popular `containers` library which ships with GHC. This provides an implementation of topological sort for strongly connected components, which we use as-is. A simple shell script loops through these SCCs, invoking the recurrent clustering component for each and appending the resulting features and clusters to the database.

4.3 Feature Extraction

The implementation of our feature extraction algorithm is a rather direct translation of the description given in §3 into Haskell. We parse the s-expressions generated by AstPlugin into algebraic data types which correspond directly to the definitions in Figure 1; this is routine, so we omit the details for brevity. Similarly, we can represent rose trees with a datatype corresponding to the definition given in §3.1.1:

```
data RoseTree = Node Feature [RoseTree]
```

For simplicity we use the representation `Feature = Int`, as we do not have fractional values. Since we represent the symbols *expr*, *id*, etc. from Figure 1 using different datatypes, we cannot write one big definition of $toTree$ or ϕ which works on all tokens. Each case shown in Figure 5 and equations 4, 5 and 6 appears in the implementation, although they are spread across several functions.

To support looking up local identifiers, our implementation of $toTree$ takes a context as argument, extending it as required. As an example of the complexity

this adds, here is the `Case` branch of $toTree$:

```
toTree :: Context -> Expr -> RoseTree
toTree ctx x = case x of
  ...
  Case e l as -> Node fCase (toTree ctx e : map (toTreeAlt (l:ctx)) as)
  ...
```

Breaking this down we can see `fCase` representing the value of ϕ(`Case`), and a list of sub-trees defined in parentheses. The first subtree is a straightforward recursive call in an unmodified context: `toTree ctx e`. The rest of the list is formed by applying the function `toTreeAlt (l:ctx)` to each element of the list `as` of `Alt` clauses.

The `toTreeAlt` function contains those cases of $toTree$ which handle symbols in alt. We prepend the identifier `l` to the context, to get the extended context `l:ctx`. This is because `l` will be bound the value of `e`, in order to avoid re-computing its value several times.

The other clauses are handled in a similar way. The trickiest is the `Let` clause, since the local identifiers aren't directly available; we must extract them from their `Rec`, `NonRec` and `Bind` wrappers first, which we do using helper functions.

As shown above, the values from equations 4 are encoded directly in $toTree$. For $phi(l \in \mathcal{L})$ we use standard Haskell functions to look up the required indices in the context:

```
phiL :: Context -> Local -> Feature
phiL ctx x = case elemIndex x ctx of
  Nothing -> error (concat ["Local '", show x,
                            "' not in context '",
                            show ctx, "'"])
  Just i  -> (2 * alpha) + i
```

As explained in §3.1.2, local identifiers should always exist in the context. If this precondition doesn't hold, we abort the program with an error rather than continuing.

Global identifiers are kept as-is until we have access to the clusters from the last iteration. This takes place outside Haskell, using the `jq` data processing tool.

Our implementation of $level$ exactly matches equation 2:

```
level :: Int -> RoseTree -> [[Feature]]
level 1 (Node f _)  = [f]
level n (Node _ ts) = concatMap (level (n-1)) ts
```

To produce feature vectors, we do not directly construct the matrix; instead we generate the rows and concatenate them together in one step, using the `concatMap` function:

```
featureVec :: Expr -> [Feature]
featureVec e = concatMap (\m -> pad (level m tree)) [1..r]
  where tree   = toTree [] e
```

```
pad xs = take c (xs ++ repeat 0)
```

By providing `featureVec` with the latest set of clusters, read from the AstPlugin database, we turn Core expressions into feature vectors, which are appended to the database.

We use the Weka system to perform our k-means clustering, as it is widely used, including by ML4PG. We select all feature vectors from our database, and write them in CSV format for Weka to process. The Weka CLI command is invoked, which appends a cluster number to each of these feature vectors; we read these off and append them to the database. As long as more SCCs remain unprocessed, we keep looping this process, using the database to communicate between the feature extraction and clustering phases.

4.4 MLSpec

We cannot supply these clusters as-is to QuickSpec, since it must be provided with a *signature*. These are constructed by our MLSpec tool, using information from the AstPlugin database. Tasks performed by MLSpec include:

- Monomorphising: Given values of polymorphic type, e.g. `safeHead :: forall t. [t] -> Maybe t` and `[] :: forall t. [t]`, a testing-based system like QuickSpec is unable to evaluate these expressions without instantiating the variable `t` to a specific type. Such an instantiation is called *monomorphising*, and in the case of MLSpec we build on previous work in QuickCheck by attempting to instantiate all type variables to `Integer`. We discard those cases where this is invalid, such as variable *type constructors* (e.g. `forall c. c Bool -> c Bool`) or incompatible class constraints (e.g. `forall t. IsString t => t`).

- Qualification: All names are *qualified* (prefixed by their module's name), to avoid most ambiguity. There is still the possibility that multiple packages will declare modules of the same name, although this is rare as it causes problems for any Haskell programmer trying to use those modules. In such cases the exploration process simply aborts.

- Variable definition: Once a QuickSpec theory has been defined containing all of the given terms, we inspect the types it references and append three variables for each to the theory (enough to discover laws such as associativity, but not too many to overflow the limit of QuickSpec's exhaustive search).

- Sandboxing: One difficulty with Haskell's packaging infrastructure is that all required packages and modules must be provided up-front, usually by specification in a Cabal file. Since MLSpec builds signatures *dynamically*, depending on the cluster information it is given, we do not know what packages it may need. To work around this problem, MLSpec invokes QuickSpec for each cluster using a library we have built called `nix-eval`. This provides an `eval` function, like those commonly found in

dynamic languages such as Python and Javascript, for evaluating Haskell expressions. The key feature of `nix-eval` is that these Haskell expressions may reference packages that are not installed on the system. When such expressions are evaluated, these packages will be automatically downloaded and installed into a sandbox using the Nix package manager, and GHC will be invoked in this sandbox to perform the evaluation.

5 Related Work

5.1 Haskell

Whilst §2.1 gave some brief background on Haskell, little explanation was given for why we chose this language rather than, for example Coq or ACL2 (for which recurrent clustering algorithms already exist), or a more widely used language like Java. Here we discuss the relevant language features from a high-level, which motivated our choice:

Functional: All control flow in Haskell is performed by function abstraction and application, which we can reason about using standard rules of inference such as *modus ponens*.

Pure: Execution of actions (e.g. reading files) is separate to evaluation of expressions; hence our reasoning can safely ignore complicated external and non-local interactions.

Statically Typed: Expression are constrained by *types*, which can be used to eliminate unwanted combinations of values, and hence reduce search spaces; *static* types can be deduced syntactically, without having to execute the code.

Non-strict: If an evaluation strategy exists for β-normalising an expression (i.e. performing function calls) without diverging, then a non-strict evaluation strategy will not diverge when evaluating that expression. This is rather technical, but in simple terms it allows us to reason effectively about a Turing-complete language, where evaluation may not terminate. For example, when reasoning about *pairs* of values `(x, y)` and projection functions `fst` and `snd`, we might want to use an "obvious" rule such as $\forall$`x y, x = fst (x, y)`. Haskell's non-strict semantics makes this equation valid; whilst it would *not* be valid in the strict setting common to most other languages, where the expression `fst (x, y)` will diverge if `y` diverges (and hence alter the semantics, if `x` doesn't diverge).

Algebraic Data Types: These provide a rich grammar for building up user-defined data representations, and an inverse mechanism to inspect these data by *pattern-matching*. For our purposes, the useful consequences of ADTs and pattern-matching include their amenability for inductive proofs and the fact they are *closed*; i.e. an ADT's declaration specifies all of the

normal forms for that type. This makes exhaustive case analysis trivial, which would be impossible for *open* types (for example, consider classes in an object oriented language, where new subclasses can be introduced at any time).

Parametricity: This allows Haskell *values* to be parameterised over *type-level* objects; provided those objects are never inspected. This has the *practical* benefit of enabling *polymorphism*: for example, we can write a polymorphic identity function `id :: forall t. t -> t`. [8] Conceptually, this function takes *two* parameters: a type `t` *and* a value of type `t`; yet only the latter is available in the function body, e.g. `id x = x`. This inability to inspect type-level arguments gives us the *theoretical* benefit of being able to characterise the behaviour of polymorphic functions from their type alone, a technique known as *theorems for free* [79].

Type classes: Along with their various extensions, type classes are interfaces which specify a set of operations over a type (or other type-level object, such as a *type constructor*). Many type classes also specify a set of *laws* which their operations should obey but, lacking a simple mechanism to enforce this, laws are usually considered as documentation. As a simple example, we can define a type class `Semigroup` with the following operation and associativity law:

```
op :: forall t. Semigroup t => t -> t -> t
```

$$\forall \texttt{x y z}, \texttt{op x (op y z)} = \texttt{op (op x y) z}$$

The notation `Semigroup t =>` is a *type class constraint*, which restricts the possible types `t` to only those which implement `Semigroup`. [9] There are many *instances* of `Semigroup` (types which may be substituted for `t`), e.g. `Integer` with `op` performing addition. Many more examples can be found in the *typeclassopedia* [83]. This ability to constrain types, and the existence of laws, helps us reason about code generically, rather than repeating the same arguments for each particular pair of `t` and `op`.

Equational: Haskell uses equations at the value level, for definitions; at the type level, for coercions; at the documentation level, for typeclass laws; and at the compiler level, for ad-hoc rewrite rules. This provides us with many *sources* of equations, as well as many possible *uses* for any equations we might discover. Along with their support in existing tools such as

[8] Read "`a :: b`" as "`a` has type `b`" and "`a -> b`" as "the type of functions from `a` to `b`".

[9] Alternatively, we can consider `Semigroup t` as the type of "implementations of `Semigroup` for `t`", in which case `=>` has a similar role to `->` and we can consider `op` to take *four* parameters: a type `t`, an implementation of `Semigroup t` and two values of type `t`. As with parameteric polymorphism, this extra `Semigroup t` parameter is not available at the value level. Even if it were, we could not alter our behaviour by inspecting it, since Haskell only allows types to implement each type class in at most one way, so there would be no information to branch on.

SMT solvers, this makes equational conjectures a natural target for theory exploration.

Modularity: Haskell has a module system, where each module may specify an *export list* containing the names which should be made available for other modules to import. When such a list is given, any expressions *not* on the list are considered *private* to that module, and are hence inaccessible from elsewhere. This mechanism allows modules to provide more guarantees than are available just in their types. For example, a module may represent email addresses in the following way:

```
module Email (Email(), at, render) where

data Email = E String String

render :: Email -> String
render (E u h) = u ++ "@" ++ h

at :: String -> String -> Maybe Email
at "" h  = Nothing
at u  "" = Nothing
at u  h  = Just (E u h)
```

The `Email` type guarantees that its elements have both a user part and a host part (modulo divergence), but it does not provide any guarantees about those parts. We also define the `at` function, a so-called "smart constructor", which has the additional guarantee that the `Email`s it returns contain non-empty `String`s. By ommitting the `E` constructor from the export list on the first line [10], the only way *other* modules can create an `Email` is by using `at`, which forces the non-empty guarantee to hold globally.

Together, these features make Haskell code highly structured, amenable to logical analysis and subject to many algebraic laws. However, as mentioned with regards to type classes, Haskell itself is incapable of expressing or enforcing these laws (at least, without difficulty [47]). This reduces the incentive to manually discover, state and prove theorems about Haskell code, e.g. in the style of interactive theorem proving, as these results may be invalidated by seemingly innocuous code changes. This puts Haskell in a rather special position with regards to the discovery of interesting theorems; namely that many discoveries may be available with very little work, simply because the code's authors are focused on *software* development rather than *proof* development. The same cannot be said, for example, of ITP systems; although our reasoning capabilities may be stronger in an ITP setting, much of the "low hanging fruit" will have already been found through the user's dedicated efforts, and hence theory exploration would be unlikely to discover unexpected properties.

[10] The syntax `Email()` means we're exporting the `Email` type, but not any of its constructors.

Other empirical advantages to studying Haskell, compared to other programming languages or theorem proving systems, include:

- The large amount of Haskell code which is freely available online, e.g. in repositories like Hackage, with which we can experiment.
- The existence of theory exploration systems such as HipSpec, and related tools which we may be able to re-use, including conjecture generators like QuickSpec; counterexample finders like QuickCheck, SmallCheck and SmartCheck; theorem provers like Hip [68]; and other related testing and term-generating systems like MuCheck [43], MagicHaskeller [38] and Djinn [3].
- The remarkable amount of infrastructure which exists for working with Haskell code, including package managers, compilers, interpreters, parsers, static analysers, etc.

5.2 Theory Exploration

We briefly described theory exploration in §2.3, as the task of discovering *new* theorems in a software or proof library, rather than proving/disproving user-provided statements. The idea was first introduced in the Theorema [13] system of Buchberger. This provided an interactive environment, similar to computer algebra systems and interactive theorem provers. In this setting, many of our concerns such as the generation of values and deciding which properties to explore are simply delegated to the user; the software would check for correctness, store results and perform searches; again, similar to interactive theorem provers.

Subsequent systems have investigated *automated* theory exploration, for tasks such as lemma discovery. By removing user interaction, these concerns about directing search must be solved by algorithms. As well as QuickSpec and HipSpec in Haskell, automated theory exploration has been applied to Isabelle [57, 35, 36].

We have focused our attention on QuickSpec, although it does not actually *prove* its results, and hence may not be considered a theory exploration system on its own. However, it does form a vital component of HipSpec, which uses off-the-shelf automated theorem provers (ATPs) to verify QuickSpec's conjectures, forming a complete theory exploration system as well as a capable inductive theorem prover (by exploiting theory exploration for lemma generation) [14]. Due to HipSpec's use in Hipster, improvements to QuickSpec also benefit work being pursued in Isabelle.

5.3 Relevance Filtering

The combinatorial nature of formal systems causes many proof search methods, such as resolution, to have exponential complexity [26]; hence even a modest size increase can turn a trivial problem into an intractable one. Finding efficient

alternatives for such algorithms, especially those which are NP-complete (e.g. determining satisfiability) or co-NP-complete (e.g. determining tautologies), seems unlikely, as it would imply progress on the famously intractable open problems of P = NP and NP = co-NP. On the other hand, we can turn this difficulty around: a modest *decrease* in size may turn an intractable problem into a solvable one. We can ensure that the solutions to these reduced problems coincide with the original if we only remove *redundant* information. This leads to the idea of *relevance filtering* (or, *premise selection*, when viewed as the *addition* of relevant information to an initially-empty problem). This is the core idea behind our restriction of theory exploration to intelligently-selected clusters of symbols, rather than whole libraries at a time.

Relevance filtering has mostly been used in automated proof search, where it simplifies problems by removing from consideration those clauses (axioms, definitions, lemmas, etc.) which are deemed *irrelevant*. The technique is used in Isabelle's Sledgehammer tool, during its translation of Isabelle/HOL theories to statements in first order logic: rather than translating the entire theory, only a sub-set of relevant clauses are included. This reduces the size of the problem and speeds up the proof search, but it creates the new problem of determining when a clause is relevant: how do we know what will be required, before we have the proof?

The initial approach taken by Sledgehammer, known as MePo (from *Meng-Paulson* [55]), gives each clause a score based on the proportion $\frac{m}{n}$ of its symbols which are "relevant" (where n is the number of symbols in the clause and m is the number which are relevant). Initially, the relevant symbols are those which occur in the goal to be proved, but whenever a clause is found which scores more than a particular threshold, all of its symbols are then also considered relevant. There are other heuristics applied too, such as increasing the score of user-provided facts (e.g. given by keywords like `using`), locally-scoped facts, first-order facts and rarely-occuring facts. To choose r relevant clauses for an ATP invocation, we simply order the clauses by decreasing score and take the first r of them.

Recently, a variety of alternative algorithms have also been investigated, for example the MaSH algorithm (Machine Learning for SledgeHammer) [40] uses the "visibility" of one theorem from another to determine the relevance of clauses. Visibility is essentially a dependency graph of which theorems were used in the proofs of which other theorems (although the theorems are actually represented as abstract sets of features). To select relevant clauses for a goal, the set of clauses which are visible from the goal's components is generated; this is further reduced by (an efficient approximation of) a naïve Bayes algorithm.

Another example is *multi-output ranking* (MOR), which uses a support vector machine (SVM) approach for selecting relevant axioms from the Mizar Mathematical Library for use by the Vampire ATP system [1]. Many more approaches are described and evaluated in [41], some of which may be directly applicable in the context of theory exploration.

5.4 Recurrent Clustering

Our recurrent clustering approach takes inspiration from the ML4PG [39] and ACL2(ml) [28] tools, used for analysing proofs in Coq and ACL2, respectively. Whilst both transform syntax trees into matrices, the algorithm of ML4PG most closely resembles ours as it assigns tokens directly to matrix elements. In contrast, the matrices produced by ACL2(ml) *summarise* information about the tree; for example, one column counts the number of variables appearing at each tree level, others count the number of function symbols which are nullary, unary, binary, etc. Whilst it may be interesting to contrast our current algorithm with an alternative based on that of ACL2(ml), it is unclear how such summaries could be extended to include types, which seems the next logical step for our approach. The ML4PG algorithm extends trivially, by using (term, type) pairs instead of just terms.

The way we *use* our clusters to inform theory exploration is actually more similar to that of ACL2(ml) than ML4PG. ML4PG can either present clusters to the user for inspection, or produce automata for recreating proofs. In ACL2(ml), the clusters are used to restrict the search space of a proof search, much like we restrict the scope of theory exploration.

ACL2(ml) reasons by analogy: finding theorem statements which are similar to the current goal, and attempting to prove the goal in a similar way. In particular, the lemmas used to prove a theorem are mutated by substituting symbols for those which appear in the same cluster. For example, if `plus` and `multiply` are clustered together, and we are trying to prove a goal involving `multiply`, then ACL2(ml) might consider an existing theorem involving `plus`. The lemmas used to prove that theorem will be mutated, for example replacing occurrences of `plus` with `mult`, in an attempt to prove the goal.

Whilst we do not currently reason by analogy, this is an interesting area for future work in theory exploration: given a set of theorems relating particular terms, we might form conjectures regarding similar terms found through clustering.

5.5 Feature Extraction

One major difficulty when applying statistical machine learning algorithms to *languages*, such as Haskell, is the appearance of recursive structures. This can lead to nested expressions of arbitrary depth, which are difficult to compare in numerical ways. One solution, as described in §2.4.1, is to use *feature extraction*; however, our method is not the only possible way to encode recursive structures as fixed-size features.

The simplest way to encode such inputs is to simply choose a desired size, then pad anything smaller and truncate anything larger. We use this to make our matrices a uniform size, borrowing the idea from ML4PG. Care must be taken to ensure that we are not discarding too much information, that we are not producing features with too many dimensions to be practical, and that there is a uniform "meaning" to each feature across different feature vectors. In our

case, we avoid many of these problems by transforming the recursive structure of expressions into matrices first; this gives each feature a stable meaning such as "the ith token from the left at the jth level of nesting".

Truncation works best when the input data is arranged with the most significant data first (in a sense, it is "big-endian"). This is the case for Haskell expressions, since the higher levels of the syntax tree are the most semantically significant; for example, the lower levels may never even be evaluated due to laziness. This allows us to truncate more aggressively than if the leaves were most significant.

By modelling our inputs as points in high-dimensional spaces, we can consider feature extraction as projection into a lower-dimensional space (known as *dimension reduction*). Truncation is a trivial dimension reduction technique; more sophisticated projection functions consider the *distribution* of the input points, and project with the hyperplane which preserves as much of the variance as possible (or, equivalently, reduces the *mutual information* between the points).

Techniques such as *principle component analysis* (PCA) can be used to find these hyperplanes, but unfortunately require their inputs to already have a fixed, integer number of dimensions. In the case of our recursive expressions (which we may consider to have *fractal* dimension), we would need another pre-processing stage to satisfy this requirement.

There are machine learning algorithms which can handle variable-size input, but these are often *supervised* algorithms which require an externally-provided error function to minimise. Error functions can be given for clustering, for example k-means implicitly minimises the function given in equation 1, but unsupervised algorithms may be preferred for efficiency as they are more direct.

One example of learning from variable-size input is to use *recurrent neural networks* (RNNs). These contain cyclic connections between neurons, unlike the traditional acyclic "feed-forward" NNs, allowing state to persist between observations. In this way, each data point can be divided into a sequence of arbitrary length, for example an s-expression, and fed into an RNN one token at a time for processing.

Unfortunately RNNs are difficult to train. The standard way to train NNs is the back-propagation algorithm; when this is extended to handle cycles we get the *backpropagation through time* algorithm [80]. However, this suffers a problem known as the *vanishing* (or *exploding*) *gradient*: error values change exponentially as they propagate back through the cycles, which prevents effective learning of correlations across a sequence, undermining the main advantage of RNNs. The vanishing gradient problem is the subject of current research, with countermeasures including *neuroevolution* (using evolutionary computation techniques rather than back-propagation) and *long short-term memory* (LSTM; introducing special nodes to "store" state, rather than having them loop around a cycle [30]).

Using sequences to represent recursive structures is also problematic: if we want our learning algorithm to exploit structure (such as the depth of a token), it will have to discover how to parse the sequences for itself, which seems

wasteful. The *back-propagation through structure* approach [25] is a more direct solution to this problem, using a feed-forward NN to learn recursive distributed representations [64] which correspond to the recursive structure of the inputs. Such distributed representations can also be used for sequences, which we can use to encode sub-trees when the branching factor of nodes is not uniform [42]. More recent work has investigated storing recursive structures inside LSTM cells [85].

A simpler alternative for generating recursive distributed representations is to use circular convolution [63]. Although promising results are shown for its use in *distributed tree kernels* [84], our preliminary experiments in applying circular convolution to functional programming expressions found most of the information to be lost in the process; presumably as the expressions are too small.

Kernel methods have also been applied to structured information, for example in [23] the input data (including sequences, trees and graphs) are represented using *generative models*, such as hidden Markov models, of a fixed size suitable for learning. Many more applications of kernel methods to structured domains are given in [6], which could be used to learn more subtle relations between expressions than recurrent clustering alone.

5.6 K-Means

We use the Weka tool to perform k-means clustering [31], since we are more concerned with the application of feature extraction to Haskell and its use in theory exploration, rather than precise tuning of learning algorithms. Since k-means is a standard method, there are many other implementations available. More interestingly, there are many other clustering algorithms we could use, such as *expectation maximisation* [11], but experiments with ML4PG have shown little difference in their results; in effect, the quality of our features is the bottleneck to learning, so there is no reason to avoid a fast algorithm like k-means.

In any case there are many conservative improvements to the standard k-means algorithm, which could be applied to our setup. For example, a more efficient approach like *yinyang k-means* [20] could make larger input sizes more practical to cluster, especially since recurrent clustering invokes k-means many times. The *k-means++* approach [2, 5] can be used to more carefully select the "seed" values for the first timestep, and the *x-means* algorithm [61] is able to estimate how many clusters to use (our *final* clusters should be tuned to maximise the performance of the subsequent theory exploration step, but x-means could still be useful in the recurrent clustering steps).

[11] In fact, k-means is very similar to expectation-maximisation, as it alternates between an *expectation step* (finding the mean value of each cluster) and a *maximisation step* (assigning points to the cluster they're most similar to; or alternatively, *minimising* the distance of each point to the centre of its cluster, as per equation 1).

6 Evaluation

We have applied our recurrent clustering algorithm to several scenarios, with mixed results. A major difficulty in evaluating these clusters is that we have no "ground truth", i.e. there is no objectively correct way to compare expressions. Instead, we provide a qualitative overview of the more interesting characteristics.

As a simple example, we clustered (Haskell equivalents of) the running examples used to present ACL2(ml) [28], shown in Figure 9. These include tail-recursive and non-tail-recursive implementations of several functions. We expect those with similar *structure* to be clustered together, rather than those which implement the same function. The results are shown in Figure 11, where we can see the "`Tail`" functions clearly distinguished, with little distinction between the tail recursive and naïve implementations.

Next we tested whether these same functions would be clustered together when mixed with seemingly-unrelated functions, in this case 207 functions from Haskell's `text` library. In fact, the `helperFib` and `fibTail` functions appeared together in a separate cluster from the rest. This was unexpected, with no obvious semantic connection between these two functions and the others in their cluster (although most are recursive, due to the nature of the `text` library).

We have also applied our recurrent clustering algorithm to a variety of the most-downloaded packages from HACKAGE (as of 2015-10-30), including `text` (as above), `pandoc`, `attoparsec`, `scientific`, `yesod-core` and `blaze-html`. Whilst we expected functions with a similar purpose to appear together, such as the various reader and writer functions of `pandoc`, there were always a few exceptions which became separate for reasons which are still unclear.

When clustering the `yesod` Web framework, the clustering did seem to match our intuitions, in particular since all 15 of Yesod's MIME type identifiers appeared in the same cluster.

Whilst recurrent clustering has produced results which merit further investigation, the application to theory exploration has yet to be tested empirically. This is due to QUICKSPEC's use of QUICKCHECK's `Arbitrary` type class to generate random values for instantiating variables. Whilst we can automatically define QUICKSPEC theories and invoke them with `nix-eval`, not all types have `Arbitrary` instances; those without cannot be given any variables in our signature, which severely limits the possible combinations which can be explored. In many cases, no variables can be included at all, leaving just equations involving constants. This has so far prevented us from measuring the direct impact on QUICKSPEC performance, either directly by exploring the sub-sets identified through recurrent clustering, or indirectly by comparing the equations generated by a full brute-force search to our recurrent clusters: those equations relating terms from different clusters would not be discovered by our method. It is this ratio of equations found through brute force to those found after narrowing-down by clusters which is one of our key objectives to maximise at this stage; until we begin to pursue the *interestingness* of the properties.

The following less-serious problems were also encountered while applying

```
(defun fact (n)
  (if (zp n) 1 (* n (fact (- n 1)))))

(defun helper-fact (n a)
  (if (zp n) a (helper-fact (- n 1) (* a n))))

(defun fact-tail (n)
  (helper-fact n 1))

(defun power (n)
  (if (zp n) 1 (* 2 (power (- n 1)))))

(defun helper-power (n a)
  (if (zp n) a (helper-power (- n 1) (+ a a))))

(defun power-tail (n)
  (helper-power n 1))

(defun fib (n)
  (if (zp n)
      0
      (if (equal n 1)
          1
          (+ (fib (- n 1)) (fib (- n 2))))))

(defun helper-fib (n j k)
  (if (zp n)
      j
      (if (equal n 1)
          k
          (helper-fib (- n 1) k (+ j k)))))

(defun fib-tail (n)
  (helper-fib n 0 1))
```

Figure 9: Common Lisp functions, both tail-recursive and non-tail-recursive.

```
fact n = if n == 0
           then 1
           else n * fact (n - 1)

helperFact n a = if n == 0
                   then a
                   else helperFact (n - 1) (a * n)

factTail n = helperFact n 1

power n = if n == 0
            then 1
            else 2 * power (n - 1)

helperPower n a = if n == 0
                    then a
                    else helperPower (n - 1) (a + a)

powerTail n = helperPower n 1

fib n = if n == 0
          then 0
          else if n == 1
                 then 1
                 else fib (n - 1) + fib (n - 2)

helperFib n j k = if n == 0
                    then j
                    else if n == 1
                         then k
                         else helperFib (n - 1) k (j + k)

fibTail n = helperFib n 0 1
```

Figure 10: Haskell equivalents of the Common Lisp functions in Figure 9.

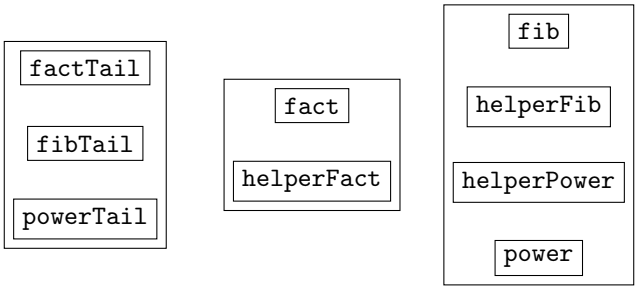


Figure 11: Typical clusters for the functions in Figure 10.

ML4HS to HACKAGE packages:

- Some packages, such as `warp` and `conduits`, get no declarations to cluster. This is because they make all of their declarations privately, e.g. in "internal" modules, then use separate modules to export the public declarations. GHC's renaming phase makes all references to such exports canonical, by pointing them to the private declarations. This forces us to ignore such declarations, as QUICKSPEC will not be able to access them.
- Since we do not support type-level entities, we ignore type classes. Unfortunately, this also means ignoring any value-level bindings (AKA "methods") which occur in a type class instance. Instead of being clustered, these result in references getting $f_{recursion}$ features. This is especially noticable in libraries like `scientific`, where only the functions for constructing and destructing numbers in scientific notation are clustered; all of the arithmetic is defined in type classes. One difficulty with supporting methods is that their namespace in Core is disjoint from that of regular Haskell identifiers: a transformation layer would be required, along with explicit type annotations to avoid ambiguity.

It seems like this recurrent clustering method has promise, although it will require a more thorough exploration of the parameters to obtain more intuitively reliable results. These clusters can then be used in several ways to perform theory exploration; the most naïve way being to explore each cluster as QUICKSPEC signature in its own right. ML4HS already provides this functionality, although the lack of test generators severely limits what can be discovered.

As alluded to previously, we also have the opportunity to reason by *analogy*. Similar to the work on ACL2(ml) [28], we could produce a general "scheme" from each equation we find (either through QUICKSPEC or by data mining test suites); like Isabelle approaches have shown [57], these schemes could then be instantiated to a variety of similar values, in an attempt to find new theorems which are analogues of existing results from a different context. "Mutating" existing theorem statements in such a way would also increase the chance of any result being considered interesting; since it's likely that the unmodified statement was deemed interesting, and the new result would not in general follow as a simple logical consequence.

7 Future Work

Our use of clustering to pre-process QUICKSPEC signatures has required many decisions and tradeoffs to be made. Hence our approach is just one possibility out of many alternatives which could be investigated to push this work further. In addition, there are other ways in which machine learning could aid theory exploration besides our relevance filter technique. The Gantt chart in Figure 12 shows how these relate to the short- and long-term direction being taken by this research. Below, we elaborate on the details, background and motivation for these choices.

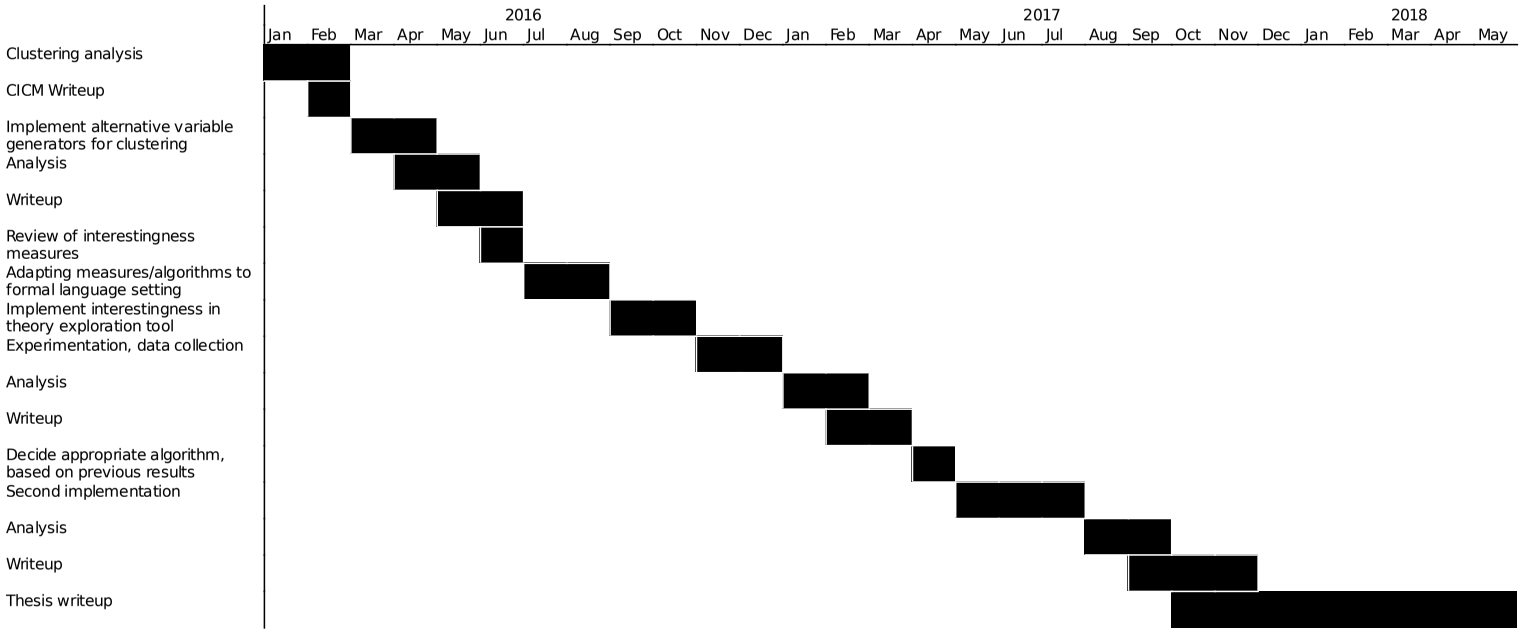


Figure 12: Gantt chart showing research direction, sub-tasks and approximate timeframes. CICM is the *Conference on Intelligent Computer Mathematics*.

7.1 Clustering Extensions

The most glaring omission in our algorithm is its disregard for types. By ignoring types, not only are we losing valuable information about expressions, but we also lose the ability to distinguish between constructors. This is because a constructor, like `True` or `Just`, has no internal structure; it is just a token. The distinguishing features of constructors are their types, which not only tell us which data type they construct, but also their arity, the types of their arguments, etc.

Our algorithm closely follows that of ML4PG, which *does* support types. This is handled by populating matrix cells with tokens *and* their types. Unfortunately this is more complicated in Haskell than it is in Coq, since types form a separate part of the language from terms, and we do not have an interactive Core environment to query for types (unlike ML4PG, which runs inside the Proof General environment).

One partial solution would be leave most Core expressions without types, but to include them for non-local identifiers (i.e. globals and constructors), which we can look up in a database. In fact, our ML4HS framework already includes such type information in its database, alongside the Core syntax trees. Integrating this information into our algorithm is the next logical step.

We can also compare the performance of our hand-selected features with *learned* representations, like those reviewed in [9]. This may provide an indication of how important it is to understand the language when identifying salient aspects of expressions, and how difficult various aspects of it might be to learn.

With more expressive features, it may also be useful to experiment with more powerful learning algorithms. An interesting possibility is to add a feedback loop between the theory exploration phase and the clustering phase, to more directly base the similarity of expressions on whether they (are predicted to) occur together in equations.

7.2 Theory Exploration Extensions

Our current approach is a rather conservative change to the existing theory exploration approaches, as it is essentially a wrapper around QuickSpec. There is potential for more radical changes to be made, which alter the search process itself.

7.2.1 Variable Instantiation

QuickCheck is certainly the most popular property checker for Haskell, which motivates its use in QuickSpec to instantiate variables to random values. However, this task of finding type inhabitants has also been solved in many other ways, which may be worth investigating in place of QuickCheck (or perhaps even as part of an ensemble).

The SmallCheck system [69] *enumerates* values rather than sampling them randomly. Whilst this does not make SmallCheck objectively "better" than

QuickCheck, one major advantage is that it may use much less memory, as the generated values are built up incrementally. In contrast, QuickCheck may generate very large values; in particular, generating tree structures naïvely can cause them to grow exponentially. For example, here is a potential generator for `RoseTree`s:

```
genRoseTree = do f        <- arbitrary
                 subtrees <- listOf genRoseTree
                 return (Node f subtrees)
```

The `listOf genRoseTree` call will return a list of arbitrary length, where each element is generated by `genRoseTree`. This allows an arbitrary number of recursive calls to be made for each invocation of `genRoseTree`, which will quickly exhaust the resources of any machine. Whilst such problems may be anticipated, or quickly spotted, in a property checking setting, this can be more difficult for our automated approach. For example, if a type does not have a generator available, we cannot use a library like `derive` to create one automatically, as it suffers from this naïvity problem.

A relative of SmallCheck is Lazy SmallCheck [66], which uses laziness to only produce parts of a datastructure as they are demanded. This may narrow down our search procedures greatly, especially when predicates are involved. QuickCheck allows predicates to restrict the values it tests with, and hence allows *conditional* equations to be discovered. However, its implementation uses a simple rejection sampling technique: values are generated just as if the predicate were not there, and afterwards are filtered to reject any which do not satisfy the predicate. This makes it difficult to use very specific predicates, as it is unlikely that many of our random samples will exactly match our criteria. On the other hand, Lazy SmallCheck will focus its search on exactly those parts of the datastructure which are checked by the predicate, as those are the parts being forced to evaluate. This makes it much more likely that we will find values which satisfy the predicate, allowing us to effectively explore more specific conditional properties.

Other approaches to generating inhabitants include Djinn [3], which uses a decision procedure for a sub-set of Haskell types which in particular can generate and apply functions (unlike the above tools, which generate values "bottom-up" from constructors, and only use functions when they have been explicitly written in a generator). MuCheck [43] is designed for *mutation testing*, and contains combinators for altering functions in common ways (e.g. changing the order of pattern match clauses); whilst not as exhaustive as the other approaches, mutating existing values in this way is claimed to yield values which correspond more closely to what a programmer might write. This is an interesting possibility for focusing theory exploration on to more "realistic" areas of the search space, and hence avoiding some of the more useless or bizarre expressions that random search and enumeration may produce.

In fact, the database generated by our AstPlugin may prove helpful in generating values, since its type information can be fed to a tool like Djinn to discover chains of function applications for building values, which would be

particularly useful in cases where constructors are private, like in our email example. This is similar to the HOOGLE tool, but also offers the ability to use dependency information to avoid potentially infinite recursion.

The Core syntax trees in our database could also be used to generate theories for automated theorem provers. HIPSPEC currently uses the GHC API to transform Core within its own process, however that approach suffers from the problems described in §4.1.

7.2.2 Interestingness

If we do succeed in producing a fast theory exploration system, which chooses productive combinations of terms and finds a large number of properties, we encounter the problem of managing the output: finding the needles we are interested in among the haystack of trivialities and coincidences.

This is governed by the "interestingness" criteria of the theory exploration system: what to keep and what to discard, and even what areas of the search space to prioritise. QUICKSPEC's approach, briefly mentioned in §2.3, is very simple: we discard equations which are direct consequences of others, and keep all the rest. Different, and more sophisticated notions of interestingness have been widely studied in other fields, which may be applied in the context of theory exploration.

7.2.2.1 Concept Formation

One directly applicable area to consider is *concept formation*, which considers the (automatic) generation of new definitions and axioms. Unlike theory exploration, such systems are not constrained by the requirement that their output be provable, and hence interestingness is an important way to judge the quality of the results.

Approaches vary, from those which are directly related to theory exploration (such as the scheme-instantiating approach of [57], which forms part of a theory exploration system), to others which are more closely related to theories of human learning and discovery [62, 58]. Those based on finding patterns in data, such as [82], may be useful in tandem with our expression database, and the results of value generators like QUICKCHECK.

Since tools like HIPSPEC already call out to third-party automated theorem provers, and indeed our own ML4HS system uses the external Weka program, there may also be merit in using external concept formation or conjecture generation tools (or reimplementations of their ideas), in order to build up more structure on top of that provided by the code we are exploring. For example, the approaches taken by AM [45, 46], Graffiti [19, 18] and HR [16, 17] could be used alongside those of QUICKSPEC.

7.2.2.2 Artificial Curiosity

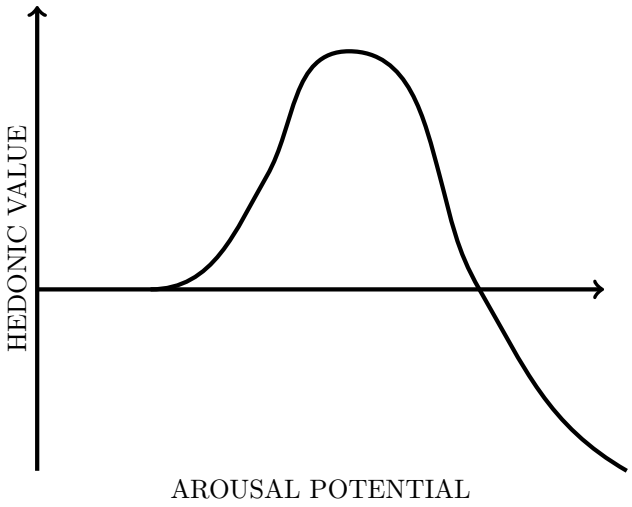


Figure 13: The Wundt curve, reproduced from [10]. The axes "hedonic value" and "arousal potential" are described as covering "reward value... preference or pleasure", and "all the stimulus properties that tend to raise arousal, including novelty and complexity", respectively.

Artificial curiosity (AC) describes active learning systems which are rewarded based on how interesting the input or data they discover is [74]. Although framed in the context of *reinforcement learning*, this reliance on interest is clearly relevant to our theory exploration setting.

As an unsupervised learning task, AC has no access to labels or meanings associated with its input; the only features it can learn are the structure and relationships inherent in the data, in a similar way to our recurrent clustering algorithm. The unifying principle of AC methods is to force systems away from inputs which are not amenable to learning; either because they are so familiar that there is nothing left to learn, or so unfamiliar that they are unintelligible. The resulting behaviour is characterised by the *Wundt curve* (shown in Figure 13) [12], which has been used in psychology to explain human aesthetics and preferences [10]. This same behaviour may be applicable to the theorems produced by a theory exploration system.

We can divide AC approaches into two groups: those which make *explicit* use of interestingness, learning from signals which follow a Wundt curve; whilst *implicit* approaches modify the *output* of their learning algorithm(s), to engineer the overall system behaviour to follow a Wundt curve as an emergent property.

A framework encompassing many examples of the explicit approach is given in [60] in the context of reinforcement learning; for comparison, many similar measures are surveyed in a data mining context in [24]. Many more reinforce-

[12] In practice, many measures avoid negative values for simplicity, in which cases we replace all negative points on the curve with zero.

ment learning examples can be found in [37, 48, 50, 51, 65, 67, 72, 59]; whilst more general descriptions are given in [71, 75, 52], which may be more amenable to our theory exploration setting.

Many of these reward signals are based on information theory, with a prominent example being *compression progress*: given a compressed representation of our previous observations, the "progress" is the space saved if we include the current observation. Observations which are incompressible or trivially compressible don't save any space, whilst observations which provide new insights into the structure of past experience can provide a space saving when compressed together. This seems particularly relevant for our problem of identifying interesting theorems: those new theorems ("observations") which shorten the proofs of previously discovered theorems may be more general, more powerful and therefore more *interesting* and hence worth keeping. In fact this is very similar to QuickSpec's interestingness criterion.

Another example of explicit artificial curiosity is given in [29], where world states which cause *disagreement* among a population of decision trees (a *random forest* [12]) are considered interesting. Since the models make stochastic predictions, the disagreement follows a Wundt curve as the complexity of state transitions increases: for parts of the state space which have been fully learned, the models will agree on accurate predictions; for parts which are unlearnable, the models cannot infer any structure, and will converge to reporting the same average value. Whilst the latter predictions may not be *accurate*, they will be *in agreement*, hence pushing down the interestingness of states which are too complex.

Many examples of the implicit case are based on *coevolution*: rewarding one part of the system for exploiting another part, and vice versa. In [73] a pair of learning algorithms place virtual "bets" on the outcome of actions, and the winner is rewarded at the expense of the loser. Due to the risk involved, each algorithm will only bet when it is confident in its prediction, and bets will only be actioned when each algorithm is confident in a *different* outcome. The overall behaviour of this system is therefore similar to the explicit measure of disagreement used in the random forest example. In terms of theory exploration, such a scheme could be used to find theorems which are *non-obvious*, and hence informative in some way to the user.

Another case is the "darwinian brain" of Fernando et al. [21, 22]. This coevolves a population of problem generators and problem solvers, rewarding the solvers based on their speed, and rewarding the generators based on the *variance* of the solvers' speed. This avoids trivial problems (which all solvers can quickly overcome) and complex problems (which no solver can manage), and focuses on those with the most possibility for learning. It is easy to imagine such a general architecture being populated by conjecture generators and theorem provers to form a theory exploration system.

7.2.2.3 Evolutionary Computation

Coevolution is a form of *evolutionary computation*; an umbrella term for heuristic search algorithms which mimic the process of evolution by natural selection among a population of candidate solutions [4]. Whilst *genetic algorithms* are perhaps the most well-known instance of evolutionary computation, their use of *strings* to represent solutions causes complications when comparing to a domain like theory exploration, where recursive structures of unbounded depth arise. Thankfully these problems are not insurmountable, for example *genetic programming* can operate on tree-structures natively [7], which makes evolutionary computation a useful source of ideas for reuse in our theory exploration setting (there are also precedents for using evolutionary computation in a theorem proving domain [76]).

Traditionally, evolutionary approaches assign solutions a *fitness* value, using a user-supplied *fitness function*. Fitness should correlate with how well a solution solves the user's problem; for example, the fitness of a solution to some engineering problem may depend on the estimated materials cost. If we frame the task of theory exploration in evolutionary computation terms, the fitness function would be our interestingness measure.

Pure exploration (i.e. for its own sake) has been studied in evolutionary computation for two main reasons: *artificial life* and *deceptive problems*. The former attempts to gain insight into the nature of life and biology through competition over limited resources. Whilst this may have utility in resource allocation, e.g. efficient scheduling of a portfolio of ATP programs, there is no direct connection to interestingness in theory exploration, so we will not consider it further (note that similar resource-usage ideas can also be found in the literature on *artificial economies*, e.g. [8]).

On the other hand, work on deceptive problems is highly relevant, as it has lead to studying various notions of intrinsic fitness, which are analogous to the interestingness measures we want. Deceptive problems are those where "pursuing the objective may prevent the objective from being reached" [44], which is caused by the fitness (objective) function having many local optima which are easy to find (e.g. by hill climbing), but few global optima which are hard to find. Many approaches try to avoid deception by augmenting the given fitness function to promote *diversity* and *novelty*, such as *niching methods* [70].

One example is *fitness sharing*, which divides up fitness values between identical or similar solutions. Say we have a user-provided fitness function f, and a population containing two identical solutions s_1 and s_2; hence $f(s_1) = f(s_2)$. In a fitness sharing scheme, we interpret fitness as a fixed resource, distributed according to f; when multiple individuals occupy the same point in the solution space, they must *share* the fitness available there. We can describe the fitness *allocated* to a solution by augmenting f, e.g. if we allocate fitness uniformly between identical solutions we get:

$$f'(x) = \frac{f(x)}{\sum_{i=1}^{n} \delta_{s_i x}}$$

Where n is the population size, s_i is the ith solution in the population and

δ is the Kronecker delta function. In the example above, assuming there are no other copies in the population, then $f'(s_1) = \frac{f(s_1)}{2} = \frac{f(s_2)}{2} = f'(s_2)$. By sharing in this way, the fitness of each solution is balanced against redundancy in the population: there may still be many copies of a solution, but only when the fitness is high enough to justify all of them.

There are many variations on this theme, such as sharing between "close" solutions rather than just identical ones and judging distance based on fitness (AKA phenotypically) rather than based on the location in solution space (AKA genetically). Yet the underlying principle is always the same: penalise duplication in order to promote diversity. This lesson can be carried over to our theory exploration context, where a theorem should be considered less interesting if it is "close" to others which have been found.

In a similar way, we can bias our search procedure, rather than our fitness function, towards diversity. The search procedure in population-based evolutionary algorithms consists of *selecting* one or more individuals from the population, e.g. via truncation (select the best n individuals, discarding the rest); then *transforming* the selected individuals, e.g. via mutation and crossover, to obtain new solutions.

Traditional selection methods are biased towards high fitness individuals (this is especially clear for truncation). Alternative schemes have been proposed which favour diversity *at the expense of* fitness. For example, the fitness uniform selection scheme (FUSS) [33] selects a target fitness f_t uniformly from the interval $[f_{min}, f_{max}]$ between the highest and lowest of the population. An individual s is then selected with fitness closest to f_t, i.e. $s = \underset{x}{\text{argmin}}|f(x) - f_t|$

In this way, the fitness function f is used to assign comparable quantities to solutions, but it is not treated as the objective; instead, the implicit objective is to maintain a diverse population, with individuals spread out uniformly in fitness space. This approach seems useful for informing our work in theory exploration, as it supports search criteria which *describe* solutions, but which we may not want to *optimise*. As a simple example, we might distinguish different forms of theorem by measuring how balanced their syntax trees are (-1 for left-leaning, +1 for right leaning, 0 for balanced); but it would be senseless to *maximise* how far they lean.

Once we begin this process of augmenting fitness functions, or abandoning their use as objectives, an obvious question arises: what happens if our new function contains nothing of the original? This kind of pure exploration scenario leads to a variety of ideas for *instrinsic* fitness, such as novelty [44], which can lead to learning useful "stepping stones" even in objective-driven domains. Such intrinsic notions of fitness are direct analogues of the interestingness measures we seek for theory exploration.

References

[1] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of automated reasoning*, 52(2):191–213, 2014.

[2] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[3] Lennart Augustsson. Djinn, a theorem prover in haskell, for haskell, 2005.

[4] Thomas Bäck, Ulrich Hammel, and Hans-Paul Schwefel. Evolutionary computation: Comments on the history and current state. *Evolutionary computation, IEEE Transactions on*, 1(1):3–17, 1997.

[5] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[6] Gökhan Bakir. *Predicting structured data*. MIT press, 2007.

[7] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

[8] Eric B Baum and Igor Durdanovic. Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 12(12):2743–2775, 2000.

[9] Yoshua Bengio, Aaron Courville, and Pierre Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, 2013.

[10] Daniel E Berlyne. Novelty, complexity, and hedonic value. *Perception & Psychophysics*, 8(5):279–286, 1970.

[11] Charles Blundell, Yee Whye Teh, and Katherine A Heller. Bayesian rose trees. *arXiv preprint arXiv:1203.3468*, 2012.

[12] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[13] Bruno Buchberger. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*, 38(2):9–32, 2000.

[14] 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.

[15] 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.

[16] Simon Colton, Alan Bundy, and Toby Walsh. Automatic concept formation in pure mathematics. 1999.

[17] Simon Colton, Alan Bundy, and Toby Walsh. Agent based cooperative theory formation in pure mathematics. In *Proceedings of AISB 2000 symposium on creative and cultural aspects and applications of AI and cognitive science*, pages 11–18, 2000.

[18] Ermelinda DeLaVina. Graffiti. pc: a variant of Graffiti. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 69:71, 2005.

[19] Ermelinda DeLaVina. Some history of the development of Graffiti. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 69:81, 2005.

[20] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup. In Francis R. Bach and David M. Blei, editors, *ICML*, volume 37 of *JMLR Proceedings*, pages 579–587. JMLR.org, 2015.

[21] Chrisantha Fernando. Design for a darwinian brain: part 1. philosophy and neuroscience. In *Biomimetic and Biohybrid Systems*, pages 71–82. Springer, 2013.

[22] Chrisantha Fernando, Vera Vasas, and Alexander W Churchill. Design for a darwinian brain: part 2. cognitive architecture. In *Biomimetic and Biohybrid Systems*, pages 83–95. Springer, 2013.

[23] Thomas Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.

[24] Liqiang Geng and Howard J Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys (CSUR)*, 38(3):9, 2006.

[25] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

[26] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[27] Jónathan Heras and Ekaterina Komendantskaya. Proof Pattern Search in Coq/SSReflect. *CoRR*, abs/1402.0081, 2014.

[28] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 389–406. Springer, 2013.

[29] Todd Hester and Peter Stone. Intrinsically motivated model learning for a developing curious agent. *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, November 2012.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[31] G. Holmes, A. Donkin, and I. H. Witten. WEKA: a machine learning workbench. *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference*, 1994.

[32] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[33] Marcus Hutter. Fitness uniform selection to preserve genetic diversity. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 1, pages 783–788. IEEE, 2002.

[34] Bill Joy Guy Steele Gilad Bracha Alex Buckley James Gosling. *The Java language specification*. Addison-Wesley Professional, 2015.

[35] Moa Johansson, Lucas Dixon, and Alan Bundy. Isacosy: Synthesis of inductive theorems. In *Workshop on Automated Mathematical Theory Exploration (Automatheo)*, 2009.

[36] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating Theory Exploration in a Proof Assistant. In StephenM. Watt, JamesH. Davenport, AlanP. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics*, volume 8543 of *Lecture Notes in Computer Science*, pages 108–122. Springer International Publishing, 2014.

[37] Frederic Kaplan and Pierre-Yves Oudeyer. Curiosity-driven development. In *Proceedings of International Workshop on Synergistic Intelligence Dynamics*, pages 1–8. Citeseer, 2006.

[38] Susumu Katayama. MagicHaskeller: System demonstration. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*, page 63. Citeseer, 2011.

[39] 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.

[40] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for sledgehammer. In *Interactive Theorem Proving*, pages 35–50. Springer, 2013.

[41] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In *Automated Reasoning*, pages 378–392. Springer, 2012.

[42] Stan C Kwasny and Barry L Kalman. Tail-recursive distributed representations and simple recurrent networks. *Connection Science*, 7(1):61–80, 1995.

[43] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mucheck: An extensible tool for mutation testing of haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 429–432. ACM, 2014.

[44] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[45] Douglas B Lenat. Automated Theory Formation in Mathematics. In *IJCAI*, volume 77, pages 833–842, 1977.

[46] Douglas B Lenat. On automated scientific theory formation: a case study using the AM program. *Machine intelligence*, 9:251–286, 1979.

[47] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. *ACM SIGPLAN Notices*, 48(12):81–92, 2014.

[48] Hod Lipson. *Curious and creative machines*. Springer, 2007.

[49] Stuart P Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[50] Matthew Luciw, Vincent Graziano, Mark Ring, and Jürgen Schmidhuber. Artificial curiosity with planning for autonomous perceptual and cognitive development. In *Development and Learning (ICDL), 2011 IEEE International Conference on*, volume 2, pages 1–8. IEEE, 2011.

[51] Luís Macedo and Amílcar Cardoso. Towards artificial forms of surprise and curiosity. In *Proceedings of the European Conference on Cognitive Science*, pages 139–144. Citeseer, 2000.

[52] Mary Lou Maher, Kathryn Elizabeth Merrick, and Rob Saunders. Achieving Creative Behavior Using Curious Learning Agents. In *AAAI Spring Symposium: Creative Intelligent Systems*, pages 40–46, 2008.

[53] Kantilal Varichand Mardia, John T Kent, and John M Bibby. *Multivariate analysis*. Academic press, 1979.

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

[55] Jia Meng and Lawrence C Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[56] Neil Mitchell. Hoogle overview. *The Monad. Reader*, 12:27–35, 2008.

[57] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39(2):1637–1646, February 2012.

[58] Dennis Müller and Michael Kohlhase. Understanding Mathematical Theory Formation via Theory Intersections in Mmt.

[59] Pierre-Yves Oudeyer. Intelligent adaptive curiosity: a source of self-development. 2004.

[60] Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, 1, 2007.

[61] Dan Pelleg, Andrew W Moore, et al. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML*, pages 727–734, 2000.

[62] Steven T. Piantadosi, Joshua B. Tenenbaum, and Noah D. Goodman. Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, 123(2):199–217, May 2012.

[63] Tony Plate. Holographic Reduced Representations: Convolution Algebra for Compositional Distributed Representations. In John Mylopoulos and Raymond Reiter, editors, *IJCAI*, pages 30–35. Morgan Kaufmann, 1991.

[64] Jordan B Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1):77–105, 1990.

[65] Dominik Maximilián Ramík, Christophe Sabourin, and Kurosh Madani. Autonomous knowledge acquisition based on artificial curiosity: Application to mobile robots in an indoor environment. *Robotics and Autonomous Systems*, 61(12):1680–1695, December 2013.

[66] Jason S Reich, Matthew Naylor, and Colin Runciman. Advances in lazy smallcheck. In *Implementation and Application of Functional Languages*, pages 53–70. Springer, 2013.

[67] S. Roa, G.-J. M. Kruijff, and H. Jacobsson. Curiosity-driven acquisition of sensorimotor concepts using memory-based active learning. *2008 IEEE International Conference on Robotics and Biomimetics*, February 2009.

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

[69] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, volume 44, pages 37–48. ACM, 2008.

[70] Bruno Sareni and Laurent Krähenbühl. Fitness sharing and niching methods revisited. *Evolutionary Computation, IEEE Transactions on*, 2(3):97–106, 1998.

[71] Tom Schaul, Yi Sun, Daan Wierstra, Fausino Gomez, and Jurgen Schmidhuber. Curiosity-driven optimization. *2011 IEEE Congress of Evolutionary Computation (CEC)*, June 2011.

[72] J. Schmidhuber. Curious model-building control systems. *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, 1991.

[73] Jürgen Schmidhuber. Artificial curiosity based on discovering novel algorithmic predictability through coevolution. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.

[74] Jürgen Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, 2006.

[75] Paul D Scott and Shaul Markovitch. Learning Novel Domains Through Curiosity and Conjecture. In *IJCAI*, pages 669–674. Citeseer, 1989.

[76] Lee Spector, David M Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298. ACM, 2008.

[77] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[78] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[79] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.

[80] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[81] Jeremiah Willcock, Jaakko Järvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine. Lambda expressions and closures for C++. 2006.

[82] Rudolf Wille. Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies. *Lecture Notes in Computer Science*, pages 1–33, 2005.

[83] Brent Yorgey. The typeclassopedia. *The Monad. Reader Issue 13*, page 17, 2009.

[84] Fabio Massimo Zanzotto and Lorenzo Dell'Arciprete. Distributed tree kernels. *arXiv preprint arXiv:1206.4607*, 2012.

[85] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. Long short-term memory over recursive structures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1604–1612, 2015.