Chapter 1

Similarity and Clustering

To improve the performance of theory exploration tools, we need to avoid brute-force search in favour of a more informed strategy. This, in turn, requires more information about the terms in the given signature. One generally available source of information is the syntax of a term's definition.

Unfortunately these definitions are over-complicated due to the relatively large syntax of Haskell, which includes many convenience features ("syntactic sugar") and alternative ways to express the same operation (e.g. binding using `let` or `where`, pattern matching with `if` or `case`, function abstraction via `f x = ...` or `f = \x -> ...`, etc.). This obscures the simplicity of the lambda calculus underlying the language, and more importantly it makes syntactic analysis less useful for our purposes by drawing distinctions between semantically identical terms. Since theory exploration is concerned only with semantics (e.g. telling us how code actually behaves), we perform our analyses on the syntax of the *GHC Core* language instead.

1.1 GHC Core

GHC Core is an intermediate language of the GHC compiler based on System F_C, and described in detail in [12, Appendix C]. We ignore type annotations and coercions, since these have no effect on the semantics (they have no "computational content"). The resulting sub-set of Core[1] is shown in Figure 1.1.

GHC's translation from Haskell to Core is routine: Figure 1.2 shows the Core representation of some simple functions. Although Core is more verbose, it has fewer redundant features than the full Haskell syntax: for example all pattern-matching is represented with `Case`, all function abstraction uses `Lam`(bda), and so on. We can see that similar structure in the Haskell definitions gives rise to similar structure in the Core, such as the definitions of `odd` and `even` being identical except for the identifiers. It is this close correspondence which allows us to analyse Core expressions in place of their more complicated Haskell source.

[1]As of GHC version 7.10.2.

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

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

Figure 1.1: Simplified syntax of GHC Core in BNF style. [] and (,) denote repetition and grouping, respectively. Other forms of literal, like machine words, are treated in a similar way and have been omitted for brevity.

Haskell definitions

```
plus     Z  y = y
plus (S x)  y = S (plus x y)

mult     Z  y = Z
mult (S x)  y = plus y (mult x y)

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

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

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 1.2: Example Haskell functions and their translations into the Core syntax of Figure 1.1. 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.

Note that in test data such as our Theory Exploration Benchmark, data constructors in definitions are replaced with normal functions, e.g. replacing occurrences of a constructor `S :: Nat -> Nat` with a function `s` defined as `s x = S x` (this is known as the η-expansion of `S`). This removes even more distinctions from the syntax, and also reduces cross-language differences (in particular, the Isabelle language used by the IsaScheme, IsaCoSy and Hipster theory exploration systems treats constructors in a different way to Haskell, so having constructors in our data would complicate cross-language comparisons).

1.2 Feature Extraction

Raw syntax trees, even in Core form, are difficult to analyse due to their recursive structure, use of discrete labels and heavy reliance on references. This makes them unsuitable for many machine learning algorithms, which instead require a fixed-size vector of continuous numbers known as a *feature vector*.

The process of turning raw data into feature vectors is called *feature extraction*, and we adapt the methodology of *recurrent clustering* proposed in [6, 7], since this handles recursive tree structures, resolves references and produces numeric values. We suggest a new recurrent clustering and feature extraction algorithm for Haskell (shown in Algorithm 1) and compare its similarity and differences to the existing approaches of ML4PG and ACL2(ml).

We describe our algorithm in two stages: the first transforms the nested structure of expressions into a flat vector representation; the second converts the discrete symbols of Core syntax into features (real numbers), which we will denote using the function ϕ. Before introducing our algorithm, we briefly describe *clustering*, and its use in *recurrent clustering*.

1.2.1 Clustering

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 [10, 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 [9].

We choose k-means because it is a standard approach and 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. There are many other clustering algorithms we could use, such as *expectation maximisation*, but experiments with ML4PG have shown little difference in their results; it seems that the quality of the features is the bottleneck to learning, so there is no reason to avoid a fast algorithm like k-means. We use a Haskell implementation provided by the package `http://hackage.haskell.org/k-means`.

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 [1]. 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:

$$\operatorname*{argmin}_C \sum_{i=1}^{k} \sum_{\mathbf{x} \in C^i} e(\mathbf{x}, \mathbf{m}^i)^2 \tag{1.1}$$

Many conservative improvements can be made to the standard k-means algorithm described above. For example, a more efficient approach like *yinyang k-means* [5] could make larger input sizes more practical to cluster; the *k-means++* approach [1, 2] can be used to more carefully select the "seed" values for the first timestep; and the *x-means* algorithm [11] is able to estimate how many clusters to use (instead of relying on our "rule of thumb"). However, like the choice of k-means versus other clustering algorithms, these algorithmic improvements would need higher quality data than we are capable of producing in these early experiments.

Recurrent Clustering

Recurrent clustering has been implemented in the ML4PG tool for analysing Coq proofs [8] and ACL2(ML) for ACL2 [7]. Both approaches transform syntax trees into fixed-size matrices, then concatenate the rows of these matrices to produce feature vectors. ACL2(ML) produces matrices by *summarising* 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. In contrast, the algorithm of ML4PG converts tokens of the syntax tree directly into numbers, arranging them in the matrix based on their position in the syntax tree. It is this latter algorithm that we adapt for analysing GHC Core expressions.

The key to recurrent clustering is its treatment of identifiers. From a purely symbolic perspective, they are atomic: two identifiers can either be identical or distinct. Consider the `odd` and `even` functions in Figure 1.2, which differ only in their choice of identifiers. These functions are clearly different (opposite, in fact), so their identifiers should play a role in determining their similarity. Yet these distinctions, e.g. between `True` and `False`, seem intuitively small relative to the distinction between, for example, `isEmpty` and `isValidEmailAddress`.

Recurrent clustering solves this dichotomy by introducing a finer granularity to the treatment of identifiers, by first clustering the definitions that they refer to. If two definitions appear in the same cluster, their identifiers will be denoted by similar numbers during subsequent feature extraction; if they appear in different clusters, their numbers will differ more during feature extraction. It is this use of clustering to determine features that makes the approach *recurrent*.

1.2.2 Expressions to Vectors

The first step of our feature extraction algorithm is to convert the tree structure of Core expressions to a flat vector. Since k-means clustering considers the elements of a feature vector to be *orthogonal*, 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. Different patterns of nesting can alter the "shape" of expressions, so 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 Core expressions, which represent pattern-match clauses with different patterns but the same body (`Var (Local "y")`):

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

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

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

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

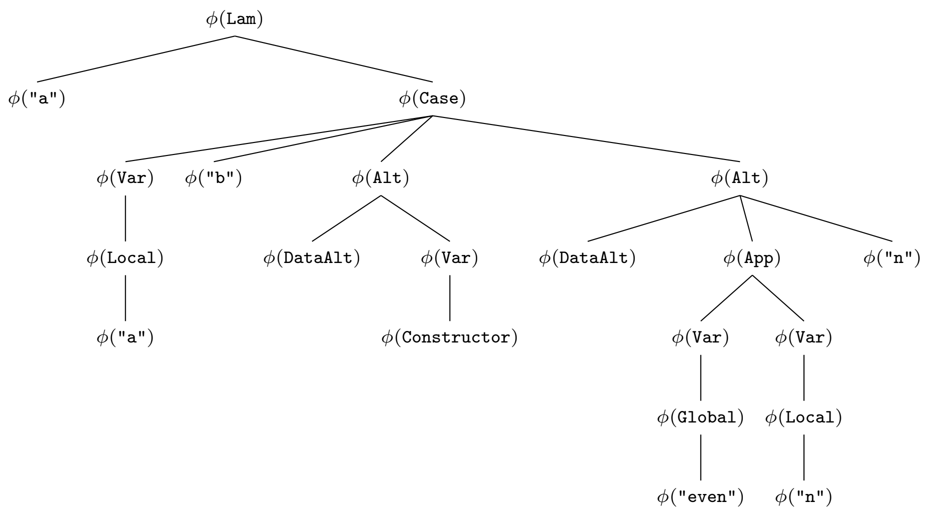


Figure 1.3: Rose tree for the expression `odd` from Figure 1.2. Each (sub-) rose tree is rendered with its feature at the node and sub-trees beneath.

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 $c = 2$:

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

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, via the function $toTree$ defined below. Intuitively, rose trees are simply an s-expression representation of the Core syntax, with the feature extraction function ϕ mapped over the labels: Figure 1.3 shows the result of transforming the `odd` function from Figure 1.2.

We follow the presentation in [3] and define rose trees recursively as follows: T is a rose tree if $T = (f, T_1, \dots, T_{n_T})$, 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 recursive definition of $toTree$ is mostly routine; each repeated element (shown as $\dots$) has an example to indicate their handling, e.g. for `Rec` we apply

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

Figure 1.4: Matrix generated from Figure 1.3, padded to 6 columns. Each level of nesting in the tree corresponds to a row in the matrix.

toTree to each e_i. We ignore values of $\mathcal{D}$, since constructors have no internal structure for us to compare. We also ignore values from $\mathcal{S}$ and $\mathcal{N}$ as it simplifies our later definition of ϕ, and we conjecture that the effect of such "magic values" on clustering real code is low.

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

These rose trees are then turned into matrices, as shown in Figure 1.4. Each row i of the matrix contains the features at depth i in the rose tree, read left-to-right, followed by any required padding. These matrices are then either truncated, or padded with rows (on the bottom) or columns (on the right) of zeros, to fit a fixed size $r \times c$.

Finally, matrices are turned into vectors by concatenating the rows from top to bottom, hence Figure 1.4 will produce a vector beginning $(\phi(\texttt{Lam}), 0, 0, 0, 0, 0, \phi(\texttt{"a"}), \phi(\texttt{Case}), 0, 0, 0, 0, \phi(\texttt{Var}), \phi(\texttt{"b"}), \phi(\texttt{Alt}), \phi(\texttt{Alt}), \dots$

1.2.3 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 [6] 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 [2]:

$$\begin{array}{rrr}
\phi(\texttt{Alt}) = \alpha & \phi(\texttt{DataAlt}) = \alpha + 1 & \phi(\texttt{LitAlt}) = \alpha + 2 \\
\phi(\texttt{Default}) = \alpha + 3 & \phi(\texttt{NonRec}) = \alpha + 4 & \phi(\texttt{Rec}) = \alpha + 5 \\
\phi(\texttt{Bind}) = \alpha + 6 & \phi(\texttt{Let}) = \alpha + 7 & \phi(\texttt{Case}) = \alpha + 8 \\
\phi(\texttt{Local}) = \alpha + 9 & \phi(\texttt{Global}) = \alpha + 10 & \phi(\texttt{Constructor}) = \alpha + 11 \\
\phi(\texttt{Var}) = \alpha + 12 & \phi(\texttt{Lam}) = \alpha + 13 & \phi(\texttt{App}) = \alpha + 14 \\
\phi(\texttt{Type}) = \alpha + 15 & \phi(\texttt{Lit}) = \alpha + 16 & \phi(\texttt{LitNum}) = \alpha + 17 \\
\phi(\texttt{LitStr}) = \alpha + 18 & &
\end{array} \tag{1.2}$$

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, we use the *de Bruijn index* of the identifier [4], denoted i_l:

$$\phi(l) = i_l + 2\alpha \quad \text{if } l \in \mathcal{L} \tag{1.3}$$

We again use α to separate these features from those of other constructs.

We discard the contents of literals and constructor identifiers when converting to rose trees, so 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 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 by ML4PG and encode global identifiers *indirectly*, by looking up the expressions which they *reference*:

$$\phi(g \in \mathcal{G}) = \begin{cases} i + 3\alpha & \text{if } g \in C_i \\ f_{recursion} & \text{otherwise} \end{cases} \tag{1.4}$$

[2] In [6], "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`.

Where C_i are our clusters (in some arbitrary order). This is where the recurrent nature of the algorithm appears: to determine the contents of C_i, 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 handle mutual recursion we use Tarjan's algorithm [13] to produce a sorted list of *strongly connected components* (SCCs), where each SCC is a mutually-recursive sub-set of the declarations. If an identifier cannot be found amongst those clustered so far, it must appear in the same SCC as the expression we are processing; hence we give that identifier the constant feature value $f_{recursion}$.

Algorithm 1 Recurrent clustering of Core expressions.

```
Require: List $d$ contains SCCs of (identifier, expression) pairs, in dependency order.
1: procedure RecurrentCluster
2:     $\mathbf{C} \leftarrow []$
3:     $DB \leftarrow \varnothing$
4:     for all $scc$ in $d$ do
5:         $DB \leftarrow DB \cup \{(i, featureVec(e)) \mid (i,e) \in scc\}$
6:         $\mathbf{C} \leftarrow kMeans(DB)$
       return $\mathbf{C}$
```

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.

As an example of this recurrent process, we can consider the Peano arithmetic functions from Figure 1.2. A valid topological ordering is given in Figure 1.5b, which can be our value for d (eliding Core expressions to save space):

$$d = [\{(\texttt{plus}, \dots)\}, \{(\texttt{odd}, \dots), (\texttt{even}, \dots)\}, \{(\texttt{mult}, \dots)\}] \qquad (1.5)$$

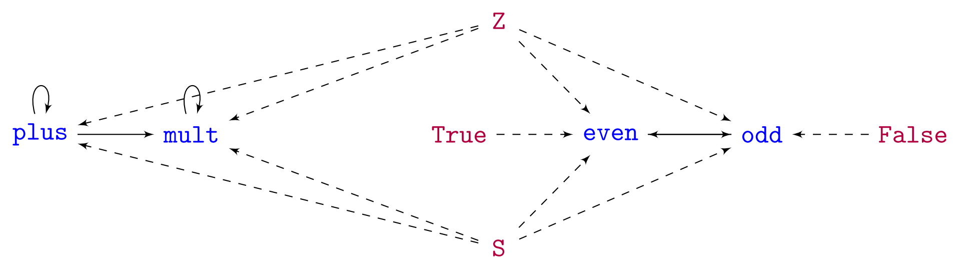
We can then trace the execution of Algorithm 1 as follows:

- The first iteration through RecurrentCluster's loop will set $scc \leftarrow \{(\texttt{plus}, \dots)\}$.

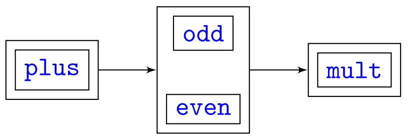
- With $i = \texttt{plus}$ and e as its Core expression, calculating $featureVec(e)$ is straightforward; the recursive call $\phi(\texttt{plus})$ will become $f_{recursion}$ (since `plus` doesn't appear in $\mathbf{C}$).

- The call to $kMeans$ will produce $\mathbf{C} \leftarrow [\{\texttt{plus}\}]$, i.e. a single cluster containing `plus`.

- The next iteration will set $scc \leftarrow \{(\texttt{odd}, \dots), (\texttt{even}, \dots)\}$.



(a) Dependency graph for Figure 1.2. 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 1.5a (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 1.5: Sorting functions from Figure 1.2 into dependency order.

- With $i =$ odd and e as its Core expression, the call to even will result in $f_{recursion}$.

- Likewise for the call to even when $i =$ 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 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 $f_{recursion}$).

1.3 Implementation

We provide an implementation of our recurrent clustering algorithm in a tool called ML4HS [3]. We obtain Core ASTs from Haskell packages using a plugin for the GHC compiler [4], which emits a serialised version of each Core definition as it is being compiled. This approach is more robust than, for example, parsing source files, since it avoids the complications of preprocessors and language extensions altering the syntax.

A post-processing stage determines which Core definitions are suitable for exploration, based on their visibility (whether they are encapsulated inside their module or visible to importers), whether a data generator is available, etc. This information, along with type signatures, arity, etc. are stored alongside the Core definitions in JSON format.

The definitions are then sorted topologically, based on the non-local identifiers appearing in their ASTs, and feature vectors are constructed using a Haskell implementation of the approach described in §1.2. Since the features associated with each identifier may vary between iterations (as the clusters change), we leave the raw identifiers in the vector so their features can be extracted in the correct context.

We implement Algorithm 1 as a Haskell executable, which performs the clustering and associates each Core definition with a cluster number. These numbers are used to finish the deferred feature extraction of identifiers, the resulting feature vectors are clustered, and the process is repeated until all SCCs have been processed. Once the recurrent clustering is complete, we output a JSON array of the clusters (serialised in an arbitrary order).

1.4 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) [7], shown in Figure 1.6. 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 1.8, 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

[3] Available at `https://github.com/warbo/ml4hsfe`

[4] Available at `https://github.com/warbo/ast-plugin`

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.

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 righ[illegible]hich we investigate as a potential solution to the Signature Selection probl[illegible]

```
(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 1.6: Common Lisp functions used with ACL2(ML), 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 1.7: Haskell equivalents of the Common Lisp functions in Figure 1.6.

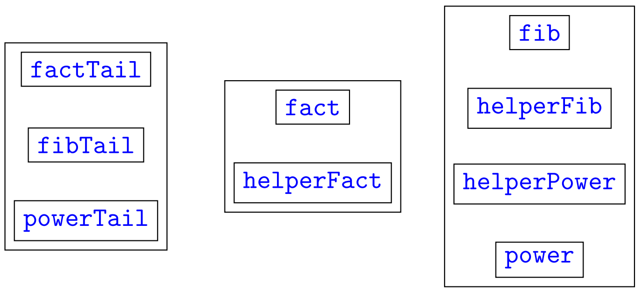


Figure 1.8: Typical clusters for the functions in Figure 1.7.

Bibliography

[1] 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.

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

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

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

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