Improving Haskell Theory Exploration

Chris Warburton

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

November 17, 2015

Abstract

Theory Exploration is a promising approach to improving the quality and understanding of software, above that available through testing, in languages which are amenable to formal analysis such as those based on pure functional programming. Current techniques are limited by their computational cost, which we mitigate through a divide and conquer approach: reducing the problem size whilst using a machine learning approach to preserve relevant details.

1 Introduction

As computers and software become more capable, 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 [27], in part by constructing programs which are more amenable to mathematical analysis. For example, by separating *computation* from *effects*, we can analyse each in isolation; in particular, pure computations can be evaluated without fear of damaging side-effects, and results are repeatable (since there can be no dependence on external state).

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 [30] and C++ [79].

Functional programming is also well suited to more radical 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:

- A framework for applying theory exploration tools such as QuickSpec to Haskell's existing package system.
- A feature extraction method for transforming Haskell expressions into a form amenable to off-the-shelf machine learning algorithms.
- 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 the results of theory exploration systems like QuickSpec (§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). We discuss our contributions in more depth in §3, and provide implementation details §4. A variety of related work is surveyed in §5, and we give several potential directions for future research in §6 before concluding in §7.

2 Background

2.1 Haskell

We decided to focus our theory exploration research on Haskell as it has mature, state-of-the-art implementations (QuickSpec [14] and HipSpec [13]). This is evident from the fact that the state-of-the-art equivalent for Isabelle/HOL, the Hipster [32] system, is actually implemented by translating to Haskell and invoking HipSpec.

Haskell is a programming language which is well-suited to research; indeed, this was a goal of the language's creators [45]. Like most members of the *functional programming* paradigm, Haskell is essentially a variant of λ-calculus; in fact an intermediate representation of the GHC compiler, known as *GHC Core*, is based on a variant of the polymorphic λ-calculus known as System F_C. For simplicity, we will focus on this Core language rather than the relatively large and complex syntax of Haskell proper. For a full treatment of System F_C and its use in GHC, see [74, Appendix C].

The sub-set of Core we consider is shown in figure 1; compared to the full language [1] our major changes are to erase types and use a custom representation of names. There are also several other forms of literal (machine words of various sizes, individual characters, etc.) which we omit for brevity, as their treatment is similar to those of strings and numerals.

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

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

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

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

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 (see 2.1 for more details on Haskell):

```
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.

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
```

[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.

```
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.[3]

2.3 Theory Exploration

In this work we focus on the problem of *(automated) theory exploration*, which includes the ability to *generate* conjectures about code, to *prove* those conjectures, and hence output *novel* theorems without guidance from the user. In [77] we identify the method of conjecture generation as a key characteristic of any theory exploration system, and in particular lament the existing reliance on brute force methods.

[3] See section 5.5 for more information on automated theorem proving.

$$
\begin{aligned}
names &= \{N_1, \ldots, N_n\} \\
types &= \{T_1, \ldots, T_m\} \cup \{a \to b \mid a \in types \wedge b \in types\} \\
exprs &= names \cup \{f\ x \mid f \in exprs \wedge x \in exprs \wedge f : a \to b \wedge x : a\} \\
classes &= \{c \mid \forall xy \in c, x \in exprs \wedge y \in exprs \wedge x =_{QC} y\} \\
conjectures &= \bigcup_{c \in classes} \{\ulcorner a = b \urcorner \mid a \in c \wedge b \in c\}
\end{aligned}
$$

Figure 2: General model of QuickSpec conjecture generation, given a signature Σ containing names N_1 to N_n and types T_1 to T_m. We use $a =_{QC} b$ to denote that a and b are indistinguishable by QuickCheck, and use $\ulcorner \ldots \urcorner$ for quasiquotation.

We focus on QuickSpec [14], which discovers equations about Haskell code. These are found through the process modelled in figure 2: given a typed signature Σ, QuickSpec enumerates all type-correct combinations of terms from Σ up to some depth; it then groups them into equivalence classes using the QuickCheck counterexample finder, then conjectures equations relating the members of these classes.

The resulting set of *conjectures* can be used in several ways: it can be simplified for direct presentation to the user, sent to a more rigorous system like HipSpec and Hipster for proving, or even serve as a background theory for an automated theorem prover [13].

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.

We attempt to mitigate the cost of running QuickSpec, by providing a machine learning layer analogous to that of premise selection in theorem proving. Of particular concern is the set of expressions *exprs*, which grows exponentially as the size of the expressions increases, and hence slows down the rest of the algorithm. Although complete, the current enumeration approach is wasteful: many combinations are unlikely to appear in theorems, which requires careful choice by the user of what to include in the signature. The key idea of premise selection is to make such choices automatically, by only including those expressions which are considered *relevant* to the problem.

2.4 Clustering and Feature Extraction

Our approach to scaling up these Haskell theory exploration tools takes inspiration from two sources. The first is premise selection, 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 §5.3.1. Premise selection is a practical tool, used in production systems like the *Sledgehammer* component of the Isabelle/HOL theorem prover.

Despite the idea's promise, we cannot simply invoke existing premise selection algorithms in our theory exploration setting. The reason is that theory exploration has no distinct *goal* to compare expressions against. Instead, we are interested in relationships between *any* components of a theory, and hence we must consider the relevance of *everything* to *everything else*. A natural fit for such problems is *clustering*, which attempts to group similar inputs together in an unsupervised way. See §5.3.2 for more information.

Our hypothesis is that clustering methods, as found in systems like ML4PG and ACL2(ml), can be used as relevance filters to break up large signatures into smaller sets more amenable to brute force enumeration.

Both of these approaches, premise selection and clustering, use *machine learning* (ML) algorithms to analyse (logical or software) expressions, and hence rely on *feature extraction* to transform the data into a suitable representation and remove irrelevant details. This has two benefits:

- *Feature vectors* (ordered sets of features) are chosen to be more compact than the data they're extracted from: feature extraction is *lossy compression*. This reduces the size of the machine learning problem, improving efficiency (e.g. running time).
- We avoid learning irrelevant details, such as the text encoding system used, improving *data* efficiency (the number of samples required to spot a pattern).

Another benefit of feature extraction is to *normalise* the input data to a fixed-size representation. Many ML 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.

As an example, say we want to learn relationships between the following program fragments:

```
data Maybe a = Nothing | Just a

data Either a b = Left a | Right b
```

We might hope our algorithm discovers relationships like:

- Both are valid Haskell code.
- Both describe datatypes.

- Both datatypes have two constructors.
- `Either` is a generalisation of `Maybe` (we can define `Maybe a = Either () a` and `Nothing = Left ()`).
- There is a symmetry in `Either`: `Either a b` is equivalent to `Either b a` if we swap occurences of `Left` and `Right`.
- It is trivial to satisfy `Maybe` (using `Nothing`).
- It is not trivial to satisfy `Either`; we require an `a` or a `b`.

However, this is too optimistic. Without our domain-knowledge of Haskell, an ML algorithm cannot impose any structure on these fragments, and will treat them as strings of bits. Our high-level hopes are obscured by low-level details: the desirable patterns of Haskell types are mixed with undesirable patterns of ASCII bytes, of letter frequency in English words, and so on.

In theory we could throw more computing resources and data at a problem, but available hardware and corpora are always limited. Instead, feature extraction lets us narrow the ML problem to what we, with our domain knowlege, consider important.

There is no *fundamental* difference between raw representations and features: the identity function is a valid feature extractor. Likewise, there is no crisp distinction between feature extraction and machine learning: a sufficiently-powerful learner doesn't require feature extraction, and a sufficiently-powerful feature extractor doesn't require any learning! [4]

Rather, the terms are distinguished for purely *practical* reasons: by separating feature extraction from learning, we can distinguish straightforward, fast data transformation (feature extraction) from complex, slow statistical analysis (learning). This allows for modularity, separation of concerns, and in particular allows "off-the-shelf" ML to be re-used across a variety of different domains.

In our case, we will use a novel feature extraction algorithm, described in §3.1, to transform expressions in Haskell Core into a fixed size representation, suitable for clustering via the standard *k-means* algorithm.

3 Contributions

3.1 Recurrent Clustering

We take the *recurrent clustering* approach found in ML4PG and ACL2(ml), and implement a variant in the context of Haskell theory exploration. Here we give a brief description of recurrent clustering, describe our algorithm, and compare its similarity and differences to ML4PG and ACL2(ml).

[4] Consider a classification problem, to assign a label $l \in L$ to each input. If we only extract a single feature $f \in L$, we have solved the classification problem without using a separate learning step.

3.1.1 Overview

The purpose of recurrent clustering, as with other clustering algorithms, is to identify similarities between a set of inputs. In our case, we want compare the abstract syntax trees of Core expressions, which presents a difficulty for clustering: many leaves are *references*, containing identifiers for other expressions. For example, consider the following expressions which are identical except for an identifier:[5]

$$X = \texttt{App (Var (Local "f")) (Var (Global "base:Prelude.(\&\&)"))}$$
$$Y = \texttt{App (Var (Local "f")) (Var (Global "base:Prelude.(++)"))}$$
$$Z = \texttt{App (Var (Local "f")) (Var (Global "base:Prelude.map"))}$$

Two simple methods for comparison are to ignore identifiers completely, or to compare them for equality. Neither is completely satisfactory, as the former will *over-estimate* similarity (considering X, Y and Z to be the same), whilst the latter will *under-estimate* similarity (considering X, Y and Z to be equidistant).

Ideally we would like something in between these two extremes, which requires a more sophisticated notion of similarity for identifiers. In the above example, we may intuitively consider `(&&)` and `(++)` to be closer to each other than to `map`; both in terms of *definition* (e.g. only the former can be specified by pattern-matching on their first argument), and by their *properties* (e.g. only the former can form monoids).

Recurrent clustering tackles this problem in a straightforward way: we look up the expressions *referenced by* each identifier, and use *their* similarity in place of the identifiers'. This gives rise to a recursive process, where feature extraction is interleaved with multiple rounds of clustering. For this recursive process to be well-founded, we impose a topological ordering on expressions based on their dependencies (the expressions they reference). This is slightly complicated in Haskell (compared to Coq, for example), since general recursion is permitted and several mutually-recursive expressions may appear at the same level.

3.1.2 Algorithm

Our feature extraction algorithm is given in figure 3. Here we highlight some key features:

- Since they are comparatively rare, we ignore the particular value of each literal.
- Since we are currently focused on expressions, we ignore particular data constructors and types.
- The `lookupL` and `lookupG` functions return the index containing their argument, if found. In the case of `lookupL`, this acts as a de Bruijn index

[5] The `(&&)` function is boolean AND, `(++)` appends lists and `map` applies a function to the elements of a list.

```
data RoseTree a = Node a [RoseTree a]

eRT :: [Local] -> [[Global]] -> Expr -> RoseTree Feature
eRT env db e = case e of
  Var (Global i) -> Node (lookupG db  i) []
  Var (Local  i) -> Node (lookupL env i) []
  Lit (LitNum _) -> Node sLITNUM []
  Lit (LitStr _) -> Node sLITSTR []
  Lam  i  e      -> Node sLAM [eRT (i:env) db e]
  App  e1 e2     -> Node sAPP [eRT    env  db e1,
                               eRT    env  db e2]
  Let  bs e      -> Node sLET (map (bRT (ids bs:env) db) bs ++
                                   eRT (ids bs:env) db  e
  Case e i alts  -> Node sCASE (eRT    env  db  e :
                            map (aRT (i:env) db) alts)

aRT :: [Local] -> [[Global]] -> Alt -> RoseTree Feature
aRT env db alt = case alt of
  (DataAlt _, vs, e) -> eRT (vs ++ env) db e
  (LitAlt  _, _,  e) -> eRT env db e
  (Default,   _,  e) -> eRT env db e

bRT :: [Local] -> [[Global]] -> Bind -> RoseTree Feature
bRT env db b = case b of
  NonRec i e -> eRT env db e
  Rec es     -> Node sREC (map (eRT env db . snd) es)

level :: Int -> RoseTree Feature -> [Feature]
level 0 (Node x xs) = [x]
level n (Node x xs) = concatMap (level (n-1)) xs

rt :: [[Global]] -> Expr -> [Feature]
rt db e = concat (pad cols (map (`level` eRT [] db e) [0..rows]))

recurrentCluster :: [[(Global, Expr)]] -> [[Global]]
recurrentCluster = go ([],[])
  where go (fs, db) []       = db
        go (fs, db) (es:ess) = let fs' = fs ++ map (extract db) es
                                   db' = kMeans fs'
                               in go (fs', db') ess
        extract db (i, e) = (i, rt db e)
```

Figure 3: Feature extraction for Haskell Core. We send named expressions to `recurrentCluster`, topologically sorted: if an element contains multiple `(Global, Expr)` pairs, they are mutually-recursive. Syntax trees are first converted into `RoseTree`s of `Feature`s, these trees are converted to matrices, then flattened into feature vectors, then `kMeans` clusters the feature vectors. The result is a list of clusters, naming their elements. More details are given in §3.1.2.

to give alpha-equivalent terms equal feature vectors. For `lookupG`, this is the ID of the cluster it appears in; this ensures that references to similar expressions result in similar features.

- `lookupL` and `lookupG` return a sentinel value `sUNKNOWN` if the given ID is not found. In practice, this only occurs for global, mutually-recursive references, since the toplogical sort will populate the `db` with all other global identifiers, and local identifiers which don't occur in the environment would be rejected as invalid Haskell before the translation to Core.
- The `Feature` type is abstract, but in practice it will be `Float`.
- Our algorithm contains several parameters, including `rows` and `cols` which determine how to truncate the matrices (defaults are 30). The `kMeans` function also contains a parameter for the cluster number; we set this as $\sqrt{n}$ where n is the number of feature vectors being clustered.

3.1.3 Comparison

Our algorithm is most similar to that of ML4PG, as our transformation maps each element in a tree to a distinct cell in its 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 `sUNKNOWN` sentinel). 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.[6]

[6] 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.

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.
- Since Core uses a single `Lam` abstraction for both value- and type-level parameters, we cannot completely erase type-level parameters unless they are fully applied. 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.

4 Implementation

Most of the work carried out so far has been either preliminary investigations, to determine if a particular approach is worth pursuing; or at an infrastructure level to support experimentation.

4.1 Circular Convolution

We have investigated the reduction to a fixed-size of tree structures of the following form (where `FV` denotes a feature vector):

```
data Tree = Leaf FV
          | Branch FV Tree Tree
```

Just like in the non-recursive case, the simplest way to reduce our dimensionality is to truncate. To reduce an arbitrary `t :: Tree` to a maximum depth `d > 0`, we can use `reduceD d t` where:

```
reduceD 1 (Branch fv l r) = Leaf   fv
reduceD n (Leaf   fv)     = Leaf   fv
reduceD n (Branch fv l r) = Branch fv (reduceD (n-1) l)
                                      (reduceD (n-1) r)
```

This is simple, but suffers two problems:

- We must choose a conservatively large `d`, since we're throwing away information
- The number of feature vectors grows exponentially as `d` increases

The first problem can't be avoided when truncating, but the second can be mitigated by truncating the *width* of the tree as well, to some `w > 0`. One way to truncate the width is tabulating the tree's feature vectors, then truncating the table to `w` × `d`, as ML4PG does.

Rather than truncating our trees, we can *combine* the feature vectors of leaves into their parents, and hence *fold* the tree up to any finite depth. Following [82] we have investigated the use of *circular convolution* (`cc`) as our combining function. Hence, to reduce `t :: Tree` to a single feature vector, we can use `reduceC t` where:

```
reduceC (Leaf fv)       = fv
reduceC (Branch fv l r) = cc fv (cc (reduceC l) (reduceC r))
```

Circular convolution has the following desirable properties:

Non-commutative: `cc a b` $\not\approx$ `cc b a`, hence we can distinguish between `Branch fv a b` and `Branch fv b a`

Non-associative: `cc a (cc b c)` $\not\approx$ `cc (cc a b) c`, hence we can distinguish between `Branch v1 a (Branch v2 b c)` and `Branch v1 (Branch v2 a b) c)`.

Here we use $\not\approx$ to represent that these quantities differ *with high probability*, based on empirical testing. This is the best we can do, since the type of combining functions `(FV, FV) -> FV` has a co-domain strictly smaller than its domain, and hence it must discard some information. In the case of circular convolution, the remaining information is spread out among the bits in the resulting feature, which is known as a *distributed representation* [59].

Distributed representations mix information from all parts of a structure together into a fixed number of bits, storing an *approximation* whose accuracy depends on the size of the value and the amount of storage used.

4.1.1 Extracting Features from XML

This method of folding with circular convolution has been implemented in the Haskell program Tree Features [7]. The feature is then:

$$feature(n) = 2^{\text{MD5}(s) \pmod L} \tag{1}$$

Where MD5 is the MD5 hash algorithm and L is the length of the desired vector, given as a commandline argument. The result of $feature(s)$ is a bit vector of length L, containing only a single 1. Due to the use of a hashing function, the position of that 1 can be deterministically calculated from the input s, yet the *a priori* probability of each position is effectively uniform.

In other words, the hash introduces unwanted patterns which are *theoretically* learnable, but in practice a strong hash can be treated as irreversible and hence unlearnable.

[7] Available online at http://chriswarbo.net/tree-features. The application parses arbitrary trees of XML and uses binary features, ie. feature vectors are bit vectors.

To calculate the initial feature of an XML element, before any folding occurs, we concatenate its name and attributes to produce a string

4.1.2 Application to Coq

Prior to 2014-09-08, the Coq proof assistant provided a rudimentary XML import/export feature, which we can use to access tree structures for learning. We achieve this by using the `external` primitive of the Ltac tactic language: `external "treefeatures" "32" t1 t2 t3.` will generate an XML tree representing the Coq terms `t1`, `t2` and `t3`, and send it to the standard input of a `treefeatures` invocation, using a vector length argument of 32.

Coq will also interpret the standard output of the command, to generate terms and tactics. This functionality isn't yet used by Tree Features, but it is clear that a feedback loop can be constructed, allowing the construction of powerful Ltac tactics which invoke external machine learning systems.

4.2 `nix-eval`

To facilitate theorem proving and machine learning, our theory exploration experiments require access to the *definitions* of the terms under investigation. This is difficult, as many terms are functions, which Haskell does not let us inspect. This limitation can be worked around by programming at a meta-level: manipulating *representations* of Haskell expressions, rather than the expressions themselves. To retain our ability to *evaluate* expressions, we then need a mechanism to transform these representations into the expressions they represent; this usually takes the form of an `eval` function.

`eval` is a common feature in languages such as Python and Javascript, where extra source files can be imported dynamically. In that case `eval` can be implemented easily by representing expressions as strings, then treating those strings as if they were the content of a file being imported.

On the other hand, Haskell does not natively support dynamic importing of extra source files, making implementation of `eval` trickier. Comprehensive implementations do exist, such as `hint`, which are effectively wrappers around GHC, but all are limited to using the Haskell modules which are already installed on the system. Since we want to explore *arbitrary* Haskell code, this is not enough for our purposes.

We have implemented a library called `nix-eval`, which provides an `eval` function for Haskell expressions which may reference packages that are not installed on the system. These packages will be automatically downloaded and installed into a sandbox when a call to `eval` is forced.

`nix-eval` is build on top of the Nix package manager and GHC. Nix provides a language for defining packages, a `cabal2nix` tool for generating Nix package definitions from those of Haskell's Cabal tool, and a `nix-shell` tool for evaluating arbitrary shell commands in a sandbox containing arbitrary packages.

The internal representation of `nix-eval` includes a list of package names, a list of module names, a list of GHC options and a string containing Haskell code. When evaluated, `nix-shell` is invoked to create a sandbox containing an instance of GHC and the Haskell packages listed in the expression. GHC is invoked in this sandbox using the `runhaskell` command, along with any options

given in the expression. The code to be evaluated is prepended with `import` statements for each required module, and wrapped in a simple `main` function for printing the result of its evaluation to standard output. The expression, therefore, must have type `String`; although this is not enforced statically.

The type of the `eval` function is `Expr -> IO (Maybe String)`, where `Expr` is the type of expressions described above. The `IO` wrapper comes from our invocation of external tools, and `Maybe String` allows us to represent failure in a clean way. The restriction of expressions to `String`s is not onerous; some concrete type is required, to communicate results from the GHC invocation, but we consider the particular choice of encoding to be out of scope for `nix-eval`. It would be a straightforward exercise to wrap our `eval` function in a generic encoding mechanism, e.g. using JSON.

4.3 ASTPlugin

The Glasgow Haskell Compiler provides mechanisms for parsing Haskell code into tree structures, and a *renaming* transformation which makes all names unique; either by prefixing them with the name of the module which defines them, or by suffixing the name with a number. 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 can 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 its Core representation, intended for optimisation passes, which we use to inspect definitions as they are being compiled. We provide a plugin called AstPlugin which implements an optimisation pass which returns its argument unchanged, but which also emits a serialised version of the definition to the console.

There are several complications in the implementation of AstPlugin. We are working with Core, which as a variant of System F_C [74] has slightly different syntax and semantics to Haskell. This is mostly a benefit, since Core is

a much simpler language than Haskell 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 more difficult to handle are:

Type variables: In §2.1 we saw that parametric polymorphism can be thought of as values being parameterised by type-level objects. In System F, this is represented explicitly by a special binding form Λ, distinct from the λ binding form for value parameters. Core also has explicit bindings for type-level objects, although for simplicity it uses the same λ representation for both types and values. This difference from Haskell, which only has explicit binding of values, 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 λ `$dOrd` in Core. This causes difficulties when looking up names, as these prefixed forms do not easily map back to the Haskell source.

Violating encapsulation: As discussed in §2.1, 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 type-check a Haskell expression which imports that name and passes it to QuickSpec. Those which fail, due to the reasons above and others, are discarded.

The result of building a Haskell package with AstPlugin is a database of Haskell definitions, similar in some respects to Hoogle. 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 (AKA database keys). Each definition also has an associated arity and type, obtained during the post-processing step mentioned above.

Such data is useful for multiple purposes:

- Dependency tracking: We have a tool which annotates each definition with its dependencies, i.e. those (package, module, name) combinations which it references. Definitions are then topologically sorted into dependency order, as this is required for our recurrent clustering technique.

- Machine learning: The Core s-expressions are used as input to our machine learning approaches, for guiding theory exploration using statistical properties of the theory being explored. As mentioned in §2.4, it is inefficient to apply intensive machine learning algorithms directly to high-dimensional data representations, so we use various feature extraction algorithms to reduce their dimensionality.

- Theory exploration: The MLSpec tool, described below, uses the output of AstPlugin to construct theories suitable for QuickSpec to explore.

- Reflection: As mentioned above, HipSpec uses the GHC API to obtain definitions of Haskell terms, which are then converted for use by theorem provers. Although we have not yet investigated this use, it seems likely that the output of AstPlugin would provide a more comprehensive and robust alternative to the GHC API approach.

- Counterexample Generation: As mentioned in §**??**, there are many Haskell property checkers, which differ mostly in the way they generate values for testing. Since our theory exploration tools are built on these property checkers, we rely on the existence of value generators for at least some of the types in our theory; yet existing approaches to deriving such generators (e.g. as found in the `derive` package) rely on choosing an arbitrary constructor then recursing. As demonstrated in §2.1, a type's constructors might not be exported, in which case these methods fail. Also, the exponential complexity of such naive recursion makes these generators unusable in many cases. It seems that a database of inspectable Haskell functions may be useful for solving this problem in a more practical way, although we have not yet performed experiments in this area.

4.4 MLSpec

When investigating heuristic methods like those of machine learning, it is important to use as much realistic data as possible to predict the system's performance on real tasks. One bottleneck for theory exploration is the lack of theories which are available to explore: we must manually select a set of terms to explore, and sometimes specify variables too.

Our MLSpec tool can automatically build theories suitable for exploration by QuickSpec, based on the databases constructed by AstPlugin described above. Features of 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 expressions such as `safeHead [] :: forall t. Maybe t` 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, in which case the exploration process will abort.
- 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: MLSpec is built on top of `nix-eval`, which allows the theory being explored to reference packages which aren't yet installed on the system.

4.5 ML4HS

ML4HS is our top-level theory exploration framework, combining AstPlugin, TreeFeatures, the Weka machine learning library and MLSpec.

Inspired by the use of premise selection (see §5.3.1) to reduce the search space in ATP, we select sub-sets of the given theory to explore, chosen to try and keep together those expressions which combine in interesting ways, and to separate those which combine in uninteresting ways.

We hypothesise that similarity-based clustering of expressions, inspired by those of ML4PG and related work in ACL2(ml) (see §5.3.2), is an effective method for performing this separation.

5 Related Work

5.1 Haskell

Haskell is a convenient choice for our purposes for several reasons. Here we discuss the relevant language features from a high-level:

Functional: All control flow 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* [75].

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
```

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

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

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* [81]. 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 SMT solvers, this makes equational conjectures a natural target for our investigation.

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

[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.

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 [40]). This reduces the incentive to manually discover, state and prove theorems about Haskell code, e.g. in the style of interactive theorem proving (see §5.4), as these results may be invalidated by seemingly innocuous code changes. We will revisit the second-class status of theorem proving in Haskell in our discussion of theory exploration (§2.3), but we will remark that Haskell is thus in a unique 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 TE would be more likely to fit into an automation role for the user, rather than providing truly unexpected new discoveries.

Other empirical advantages to studying Haskell, compared to either 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 conjecture generation and theory exploration systems for Haskell, such as QuickSpec and HipSpec, and counterexample finders like QuickCheck, SmallCheck and SmartCheck.
- The remarkable amount of infrastructure which exists for working with Haskell code, including package managers, compilers, interpreters, parsers, static analysers, etc.

5.2 Exploration in Artificial Intelligence

Theorem proving is not the only area of AI to have automation extended from problem *solving* to problem *invention*. Here we consider similar approaches in a variety of domains, and identify techniques which could be re-purposed in the context of theory exploration.

5.2.1 Evolutionary Computation

Evolutionary computation is an umbrella term for heuristic search algorithms which mimic the process of evolution by natural selection among a population

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

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 [6], which makes evolutionary computation a useful source of ideas for reuse in our theory exploration setting.

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" [38], 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* [65].

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) [28] 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 [38], 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.

5.2.2 Artificial Curiosity

Artificial curiosity (AC) describes active learning systems which are rewarded based on how interesting the input or data they discover is [69]. Although framed in the context of *reinforcement learning*, this 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

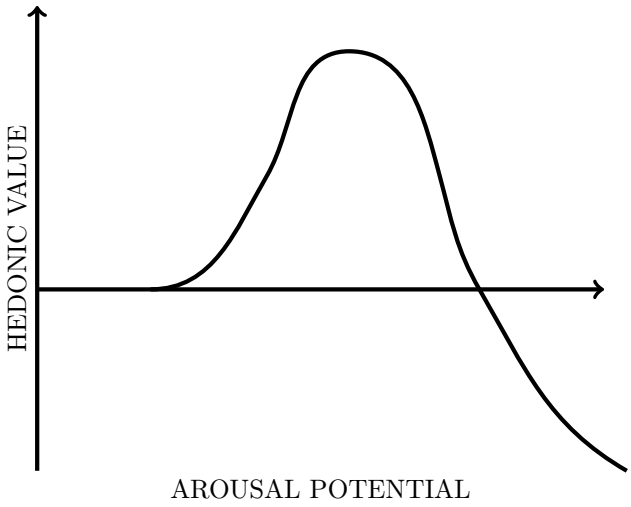

Figure 4: The Wundt curve, reproduced from [9]. The axes "hedonic value" and "arousal potential" are described as covering "reward value... preference or pleasure", and "all the stimulus properties that tend to raise arousal, including novelty and complexity", respectively.

relationships inherent in the data, which is very much what we would like a theory exploration system to do. 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 4) [11], which has been used in psychology to explain human aesthetics and preferences [9].

We can divide AC approaches into two groups: the first, which we call *explicit*, send inputs which follow a Wundt curve to their learning algorithm; the second, the *implicit* approaches, instead modify the *output* of their learning algorithm(s), such that the overall system follows a Wundt curve as an emergent property.

In the explicit case, the *implicit reward* signals being learned are analogous to our notion of interestingness. A framework encompassing many examples is given in [55] in the context of reinforcement learning.

One particularly general measure is *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 information relevant to past experience can provide a saving. This can be translated to a theorem proving context very naturally: our observations

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

are theorems and their proofs, whilst new theorems which generalise known results will allow us to compress their proofs.

[68]

Two sources of intrinsic reward are proposed in [25] for *random forests*. A random forest is a population of decision trees, where each tree is trained on a sub-set of the available examples, each decision is made using a sub-set of the available features, and the predictions of every tree are averaged to obtain that of the forest [11]. The first intrinsic reward is the *disagreement* between predictions; for a forest with m models (trees), predicting features $x_1 \dots x_n$ of the state resulting from taking action a in state s, we simply sum the Kullback-Leibler divergence D_{KL} of each prediction $P_1 \dots P_m$ from every other prediction:

$$D(s,a) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{m} D_{KL}(P_j(x_i|s,a)||P_k(x_i|s,a)) \tag{2}$$

$D(s,a)$ is an explicit AC reward, as it follows a Wundt curve as the complexity of 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 average of past observations; these predictions may not be accurate, but they will be in agreement. Hence it is the states which are amenable to learning which produce the largest disagreement.

The second intrinsic reward is simply a measure of distance from previous observations, which pushes the system towards unseen states regardless of how learnable they are (similar to the R_{max} technique). This is too simple to meet our definition of AC, but it does force the models to generalise their predictions to unexplored states, acting to increase disagreement in the forest.

A key advantage of random forests is that their models are *inspectable*: they not only give predictions, but also *reasons* for those predictions (i.e. we can see which paths are taken through each decision tree).

[33] [41] [42] [43] [61] [63] [66] [68] [67] [71] [73] [44] [48] [54] [56] [69]

Whilst clearly of relevance to theory exploration, artificial curiosity is usually framed in the context of a *reinforcement learning* and *intrinsic reward*, especially in the field of developmental robotics. This requires non-trivial choices to be made in deciding which of its concepts are of relevance to our domain, and how they may be translated across. For example, much of developmental robotics studies continuous, real-valued sensorimotor signals which may not have any direct analogue in the manipulation of logical formulae. However, if we take a higher-level view, the study of such signals may provide insight for predicting and tuning the behaviour of off-the-shelf ATP algorithms.

The most obvious contrast between developmental robotics and theory exploration is that the latter is not physically embodied (e.g. in a robot). Embodiment has been proposed as a necessary property of intelligent systems, as it provides *grounding* [2]. Embodiment emerged as a response to the symbolic techniques of GOFAI, and in this sense the fields of theory exploration and developmental robotics seem incompatible. Nevertheless, TE can be seen to avoid

the problems of GOFAI in two ways:

- Firstly, the abstract, mathematical domain being explored is not a *model* of some external, physical environment; the domain *is* our environment; hence there is no issue of grounding terms with some external meaning.
- Secondly, there is a physical aspect of TE in that *resource usage* is a critical factor. If it weren't, then brute force enumeration of proofs would be a viable solution. In this sense, we can provide physical inputs to our algorithms, such as measures of time and space used.

5.3 Statistics of Formal Systems

The core problem of assigning "interestingness" to logical formulae is the application of statistical reasoning to the discrete, semantically-rich domain of formal systems. This problem has been tackled from various directions for a variety of reasons; here we summarise those contributions which seem of particular importance for theory exploration.

5.3.1 Relevance Filtering

[36]

The combinatorial nature of formal systems causes many proof search methods, such as resolution, to have exponential complexity [21]; 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*.

Relevance filtering simplifies a proof search problem by removing from consideration those clauses (axioms, definitions, lemmas, etc.) which are deemed *irrelevant*. The technique is used in Sledgehammer 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, known as MePO (from *Meng-Paulson* [46]), gives each clause a score based on the proportion 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, 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, including:

MaSH: Machine Learning for SledgeHammer [35]. The distinguishing feature of MaSH is its use of "visibility", which is essentially a dependency graph of which theorems were used in the proofs of which other theorems; although theorems are 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 naive Bayes algorithm.

MOR: *Multi-output ranking* uses a support vector machine (SVM) approach for selecting relevant axioms from the Mizar Mathematical Library for use by the Vampire ATP system [1]. It compares favourably to SNoW and SInE.

SInE

BliStr

HOLyHammer

MoMM

SNoW

MPTP 0.2

MaLARea

MaLARea SG1

5.3.2 Clustering

[34] [24]

5.3.3 Probability of Sentences

The most important property of a logical formula is its truth value. Although we may be able to determine some truth values exactly, e.g. using decision or semi-decision procedures, it may be more efficient to *approximate* truth values. One straightforward extension of truth values is *probabilities*, where we can assign probability 1 to formulae which are known to be true, 0 to formulae known to be false, and intermediate values to those which we do not yet know.

[29]

5.3.4 Interestingness in Concept Formation

[49] [58] [80] [16] [17] [15] [39] [50] [12] [31] [72] [15] [19]

5.3.5 Learning From Structured Data

One major difficulty with formal mathematics as a domain in which to apply statistical machine learning is the use of *structure* to encode information in objects. In particular, *trees* appear in many places: from inductive datatypes, to recursive function definitions; from theorem statements, to proof objects. Such nested structures may extend to arbitrary depth, which makes them difficult to represent with a fixed number of features, as is expected by most machine learning algorithms. Here we review a selection of solutions to this problem, and compare their distinguishing properties.

5.3.5.1 Truncation and Padding The simplest way to limit the size of our inputs is to truncate anything larger than a particular size (and pad anything smaller). This is the approach taken by ML4PG [23], which limits itself to trees with at most 10 levels and 10 elements per level; each tree is converted to a 30×10 matrix (3 values per tree node) and learning takes place on these normalised representations.

Truncation is unsatisfactory in the way it balances *data* efficiency with *time* efficiency. Specifically, truncation works best when the input data contains no redundancy and is arranged with the most significant data first (in a sense, it is "big-endian"). The less these assumptions hold, the less we can truncate. Since many ML algorithms scale poorly with input size, we would prefer to eliminate the redundancy using a more aggressive algorithm, to keep the resulting feature size as low as possible.

5.3.5.2 Dimension Reduction A more sophisticated approach to the problem of reducing input size is to view it as a *dimension reduction* technique: our inputs can be modelled as points in high-dimensional spaces, which we want to project into a lower-dimensional space ($\{0,1\}^N$ in the case of N-bit vectors).

Truncation is a trivial dimension reduction technique: take the first N coordinates (bits). More sophisticated projection functions consider the *distribution* of the points, and project with the hyperplane which preserves as much of the variance as possible (or, equivalently, reduces the *mutual information* between the points).

There are many techniques to find these hyperplanes, such as *principle component analysis* (PCA) and *autoencoding*; however, since these techniques are effectively ML algorithms in their own right, they suffer some of the same constraints we're trying to avoid:

- They operate *offline*, requiring all input points up-front
- All input points must have the same dimensionality

In particular, the second constraint is precisely what we're trying to avoid. Sophisticated dimension reduction is still useful for *compressing* large, redundant features into smaller, information-dense representations, and as such provides a good complement to truncation.

The requirement for offline "batch" processing is more difficult to overcome, since any learning we perform for feature extraction will interfere with the core learning algorithm that's consuming these features (this is why deep learning is often done greedily).

5.3.5.3 Sequencing The task of dimension reduction changes when we consider *structured* data. Recursive structures, like trees and lists, have *fractal* dimension: adding layers to a recursive structure gives us more *fine-grained* features, rather than *orthogonal* features. For data mining context-free languages (e.g. those of programming and theorem-proving systems), we will mainly be concerned with tree structures of variable size.

Any investigation of variable-size input would be incomplete without mentioning *sequencing*. This is a lossless approach, which splits the input into fixed-size *chunks*, which are fed into an appropriate ML algorithm one at a time. The sequence is terminated by a sentinel; an "end-of-sequence" marker which, by construction, is distinguishable from the data chunks. This technique allows us to trade *space* (the size of our input) for *time* (the number of chunks in a sequence).

Not all ML algorithms can be adapted to accept sequences. One notable approach is to use *recurrent ANNs* (RANNs), which allow arbitrary connections between nodes, including cycles. Compared to *feed-forward* ANNs (FFANNs), which are acyclic, the *future output* of a RANN may depend arbitrarily on its *past inputs* (in fact, RANNs are universal computers).

The main problem with RANNs, compared to the more widely-used FFANNs, is the difficulty of training them. If we extend the standard backpropagation algorithm to handle cycles, we get the *backpropagation through time* algorithm [78]. However, this suffers a problem known as the *vanishing gradient*: error values decay exponentially as they propagate back through the cycles, which prevents effective learning of delayed dependencies, undermining the main advantage of RANNs. The vanishing gradient problem is the subject of current research, with countermeasures including *neuroevolution* (using evolutionary computation techniques to train an ANN) and *long short-term memory* (LSTM; introducing a few special, untrainable nodes to persist values for long time periods [26]).

The application of *kernel methods* to structured information is discussed in [18], where the input data (including sequences, trees and graphs) are represented using *generative models*, such as hidden Markov models, of a fixed size.

[18] [57] [5] [59] [20] [37] [60] [82]

5.4 Interactive Theorem Proving

ITP is based around a *proof checker* C, which is a decision procedure for determining if a given value P (known as a *proof object* or *witness*) consititutes a proof of a given statement S:

$$C\colon (P \times S) \to Boolean$$

The process of ITP can hence be understood as the search for an appropriate P for our *goal* S:

$$ITP\colon (S \times C) \to \{P \mid C(P, S) = True\}$$

It just so happens that a useful way to implement such a system is via pure functional programming, with the result that many ITP systems (AKA *proof assistants*) such as Coq and Agda appear very similar to languages like Haskell. In particular, we can represent statements S as types and proofs P as values, in which case the proof checker C is simply a type-checker. There are some obvious quirks, such as the need to be *total* (not Turing-complete) in order to ensure soundness, but overall many of the features we introduced for Haskell, such as parametricity, type classes and ADTs are directly translatable to the ITP setting.

This coincidence is due to the *Curry-Howard correspondence* [76], which identifies programming languages with systems of logic. Functional programming languages correspond to intuitionistic logics, and are hence a natural fit for reasoning on computers. With additional axioms, we can extend these to more familiar classical logics, although these can make it harder to compute.

Most differences between functional programming and ITP stem from different *expectations* of the user. In the case of Haskell, the desire for expressivity outweighs the desire for soundness, and hence its designers opted to make it Turing-complete. For an ITP language like Agda, soundness far outweighs the inconvenience of having to prove termination, so the opposite tradeoff is made. Similarly, since many ITP "programs" (proofs) will never be executed, the language can avoid optimisation in favour of simplicity (and hopefully correctness). This approach is summed up by the *de Bruijn criterion*, which requires that the system "generates 'proof-objects' (of some form) that can be checked by an 'easy' algorithm" [7, § 2]. One nice consequence of this approach, which is followed for example by Isabelle [53] and Coq [10], is that the proof assistants themselves can become arbitrarily complex and even buggy, yet as long as the proof checker remains simple we can maintain a high degree of confidence in the results.

Their emphasis on *interactivity*, mostly caused by operating in undecidable domains, means ITP can require an enormous effort for non-trivial proof or verification tasks (e.g. see [22]). One common way to mitigate this problem is by implementing powerful *tactics*: meta-programs which automate as much of a proof as possible. The most striking example of such meta-programming is the *Sledgehammer* component of Isabelle/HOL [47], which invokes a multitude

of ATP systems on (a translated form of) the current goal to see if any can provide a proof which Isabelle's core checker will accept.

5.5 Automated Theorem Proving

ATP systems are similar in principle to proof assistants, but are based around a *proof search* algorithm rather than a proof checker. By using a fixed algorithm for proving, such as *resolution* [64, § 9.6] or *superposition* [3], ATP programs are limited to particular decidable or semi-decidable fragments of logic. In particular, most ATP systems (such as E [70] and Vampire [62]) operate in classical first-order logic. This is the most striking difference from ITP systems (e.g. Coq, Agda and Isabelle) which operate in higher-order logics.

In addition to its interest to logicians, ATP has been actively researched in the field of artificial intelligence, dating back to the founding of the field at the 1956 Dartmouth conference. Even by that time Newell and Simon had developed their Logic Theory Machine [52], which was subsequently able to prove theorems like those in Principia Mathematica [51]. The approaches now known as *good old-fashioned AI* (GOFAI) were due in part to the success of automated theorem proving, as attempts were made to formulate many problems in a way amenable to these powerful first-order reasoners.

Recently there has been a trend away from this direction, towards statistical formulations amenable to machine learning, which we will review in §5.

6 Future Work

7 Conclusion

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] Michael L Anderson. Embodied cognition: A field guide. *Artificial intelligence*, 149(1):91–130, 2003.

[3] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

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

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

[7] Henk Barendregt and Herman Geuvers. Proof-Assistants Using Dependent Type Systems. *Handbook of automated reasoning*, 2:1149–1238, 2001.

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

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

[10] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

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

[12] Alan Bundy, Flaminia Cavallo, Lucas Dixon, Moa Johansson, and Roy McCasland. The Theory behind Theory Mine. *IEEE Intelligent Systems*, 30(4):64–69, July 2015.

[13] 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.

[14] 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.

[15] Simon Colton. *Automated theory formation in pure mathematics*. Springer Science & Business Media, 2012.

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

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

[20] 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.

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

[22] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, et al. A formal proof of the Kepler conjecture. *arXiv preprint arXiv:1501.02155*, 2015.

[23] Jónathan Heras and Ekaterina Komendantskaya. ML4PG: proof-mining in Coq. *CoRR*, abs/1302.6421, 2013.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] Marcus Hutter, John W. Lloyd, Kee Siong Ng, and William T. B. Uther. Probabilities on Sentences in an Expressive Logic. *Journal of Applied Logic*, 11(4):386–420, December 2013.

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

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

[32] 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.

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

[34] 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.

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

[36] 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.

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

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

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

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

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

[42] 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.

[43] 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.

[44] 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.

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

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

[47] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.

[48] J Meyer and S Wilson. A possibility for implementing curiosity and boredom in model-building neural controllers. 1991.

[49] 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.

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

[51] Allen Newell, John Calman Shaw, and Herbert A Simon. Elements of a theory of human problem solving. *Psychological review*, 65(3):151, 1958.

[52] Allen Newell, Herbert Simon, et al. The logic theory machine–A complex information processing system. *Information Theory, IRE Transactions on*, 2(3):61–79, 1956.

[53] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

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

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

[56] Pierre-Yves Oudeyer and L Smith. How evolution may work through curiosity-driven developmental process. *Topics Cogn. Sci*, 2014.

[57] F. Oveisi, S. Oveisi, A. Erfanian, and I. Patras. Tree-Structured Feature Extraction Using Mutual Information. *IEEE Transactions on Neural Networks and Learning Systems*, 23(1):127–137, January 2012.

[58] 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.

[59] 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.

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

[61] 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.

[62] Alexandre Riazanov. *Implementing an efficient theorem prover*. PhD thesis, University oF Manchester, 2003.

[63] 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.

[64] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

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

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

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

[68] 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.

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

[70] Stephan Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 735–743. Springer, 2013.

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

[72] 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.

[73] Bas R. Steunebrink, Jan Koutník, Kristinn R. Thórisson, Eric Nivel, and Jürgen Schmidhuber. Resource-Bounded Machines are Motivated to be Effective, Efficient, and Curious. *Lecture Notes in Computer Science*, pages 119–129, 2013.

[74] 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.

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

[76] Philip Wadler. Propositions as types. *Communications of the ACM*, 2015.

[77] Chris Warburton and Ekaterina Komendantskaya. Scaling Automated Theory Exploration. *Proceedings of AI4FM 2015*, 2015.

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

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

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

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

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