The HR3 System for Automated Code Generation in Creative Settings

Simon Colton,[1,2] **Alison Pease,**[3] **Michael Cook**[1] **and Chunyang Chen**[1]
[1] SensiLab, Faculty of Information Technology, Monash University, Australia
[2] Game AI Group, School of Electronic Engineering and Computer Science, Queen Mary University of London, UK
[3] Department of Computing, University of Dundee, UK
s.colton@qmul.ac.uk a.pease@dundee.ac.uk mike@gamesbyangelina.org chunyang.chen@monash.edu

Abstract

We describe the HR3 system for automated code generation, and its use in creative tasks. We outline the motivations and overall ideology behind its construction, most notably by identifying some distinctions in AI methodology which can be ignored when AI tasks are viewed as code generation problems to be solved. We further describe the nature of the approach in terms of: a programmatic interface to a Java API; production rule-based batch processing of data; on-demand code generation and inspection, and the usage of randomised and meta-level codebases. To support the claim that the approach is general purpose, we describe five applications in three areas normally covered by separate Computational Creativity systems, namely mathematical discovery, datamining and generative art. We end by discussing future directions for the HR3 system and how this project might address some higher-level issues in Computational Creativity.

Introduction

In (Colton, Powley, and Cook 2018), we proposed investigating and automating the creative act of software engineering as a major driving force for Computational Creativity research. We further suggested two main principles for undertaking projects where automated code generation was a central technique, namely: (i) that this could/should be done to *problematise the world*, i.e., used to introduce new problems/hypotheses/conjectures and creative affordances instead of/in addition to merely solving given problems, and (ii) generated programs should be celebrated as creations in their own right, not just as a means to an end. In this way, automatic code generation could provide a suitable testing ground for cutting edge Computational Creativity techniques such as framing (Charnley, Pease, and Colton 2012) and dialogue generation to convince users of the value of the generated code artefacts, and address difficult philosophical issues in the field, such as (a lack of) autonomy and intentionality in hand-programmed creative systems.

Existing techniques for code generation include automated program synthesis (Gulwani, Polozov, and Singh 2017), genetic programming (Krawiec 2016) and machine learning techniques such as inductive logic programming (Muggleton 1991). These are surveyed in (Colton, Powley, and Cook 2018), and a tentative position is presented that their limitations preclude them becoming the basis for a general-purpose automated code generation approach. Another contribution to automated code generation is the HR series of theory formation systems. HR1 (Colton 2002) was a mathematical concept formation program employed in discovery tasks such as the invention of integer sequences (Colton, Bundy, and Walsh 2000). HR2 (Colton and Muggleton 2006) was a more general-purpose datamining system, but has mainly been applied to mathematical discovery tasks, often in conjunction with other reasoning systems, for example to classify finite algebras (Sorge et al. 2008).

These systems can be seen as automated code generators, as HR1 and HR2 could produce Prolog code to represent the concepts it invented, and HR2 could also take code as input to generate data instead of reading it from a file. The code could be written in Prolog or Java and could wrap around code from systems like the Maple computer algebra system (Redfern 1999), so HR2 was essentially making discoveries about Maple code (Colton 2004). The HR3 system has been developed from scratch over the last five years to fully embrace automated code generation based on the central approach of the earlier systems. An early report on the design decisions was given in (Colton, Ramezani, and Llano 2014), along with a comparison of HR3 with HR2 and details of two applications. We describe the latest version of the system here, with five new applications, most notably with HR3 being used for the first time in generative art.

Ideology and Motivation

We are developing HR3 as a *general-purpose code generation* intelligent system for tasks requiring creativity. Our long-term aim is for it to undertake any coding task that a human programmer could complete, and coding tasks that people wouldn't be expected to hand-code, e.g., for generating images of imaginary faces. To achieve this, we view as many traditional AI tasks as possible as automated code generation problems. Our ideology also includes attempting to remove the following distinctions, which seem somewhat artificial in a context of automated code generation:

• *Task Distinctions.* Broadly speaking, **generative** AI methods produce valuable artefacts such as paintings, videogames, poems, musical composition, mathematical concepts, etc., which become an object of interaction/consumption/study. In contrast, **analytic** methods produce more information about given artefacts, and **support**

methods provide additional code which makes the whole project work. In a generative art project, for example, we might require software which (a) produces images (generative), (b) provides information about the textures in the images (analytic), and (c) presents certain images based on texture (support). A human programmer could write code for all these tasks, hence we aim for HR3 to do similarly.

• *Domain Distinctions.* Certain AI approaches, like datamining, are domain independent, but the same is not always true of the practical implementations of these approaches. This is particularly the case for the kinds of generative systems seen in Computational Creativity research, e.g., it would be unusual currently to see a music-generating system write a poem or paint a picture. Often systems are further task-localised, e.g., to generate harmonisations of given melodies rather than producing new ones. We aim for HR3 to be domain independent by enabling it to work with data of any type in a domain-independent way.

• *Clarity Distinctions.* There is often a distinction made between 'black box' techniques, the results/processing of which are difficult to understand, and more comprehensible techniques. We aim for HR3 to be able to generate both simple programs if necessary for people to understand them (for example, in datamining), and complex programs where some other criteria such as correctness, speed, variety or beauty in the processing and/or output is more important than clarity, e.g., in generative art.

There are other distinctions in AI methodology that we aim to blur, such as the difference between training and testing stages in machine learning applications, which approaches such as online-learning (Fiat and Woeginger 1998) and one-shot learning already address. Ultimately, we aim for another major distinction to be removed, which is that between a program and its programmer. That is, as HR3 can output code, we ultimately aim for it to re-write, augment and enhance parts of its own program, and we briefly discuss potential benefits of this in the conclusions section.

We gain motivation from the meteoric success of deep learning (DL) approaches in AI. These approaches are applied to both analytic and generative tasks and are domain independent. Moreover, as argued in (Colton, Powley, and Cook 2018), DL clearly performs automatic programming, although the output is not code. Deep learning is so powerful because it produces very large (but not overfitting) programs represented as artificial neural networks (ANNs). This comes at the cost of comprehensibility, as it is usually difficult to understand how an ANN performs a prediction, or generates an artefact. Methods to understand ANNs come from visualisation (which led to the huge growth in generative uses of DL), as well as methods akin to human neuroimaging (seeing which parts of an ANN fire for given inputs) and psychology (asking how an ANN views a series of inputs), but these are rarely as specific or comprehensible as those produced by symbolic AI approaches.

Charnley, Pease and Colton (2012) argue that software explaining how and why it made something is important in accepting the software as creative, and Colton, Pease and Saunders (2018) add that communicating authenticity will likely be required for acceptance of output as valid, in certain areas like poetry. Hence, we believe that more explainable AI system are preferred in creative settings over black box approaches. As described below, HR3's operation is not constrained by a rigid representation scheme, and the Java output it produces can, in principle, manipulate data in any way. General code is more flexible than ANNs, hence HR3 could be more task independent than DL, and code is easier to understand than ANN processing, hence HR3 could be as powerful, yet more comprehensible, than DL.

In the next section, we describe how HR3 operates as a Java API in data-centric creative projects. We then illustrate how HR3 operates, by presenting five new applications in three distinct areas, namely mathematical discovery, datamining and generative art. To conclude, we return to the ideology above to see where HR3 adheres, and where improvements are needed. We end by discussing how this project addresses some Computational Creativity issues, and by describing some directions for future work.

Automated Code Generation

HR3 is a Java Application Programming Interface (API) which can be called upon in various ways for creative projects, to automatically build and employ a **codebase** comprising a set of **methods**. Projects with HR3 are data-centric, with each method comprising **procedures** that manipulate a **database** which is either read from a file or generated initially by user-supplied **background methods**. The simplest way to employ HR3 is to write a single Java file adhering to a few constraints. Normally, this file grows and is constantly tweaked during the project, so we think of it as a **sketchpad**. Sketchpads employ the HR3 API in a codebase **generation phase**, followed by a codebase **interrogation phase**. A GUI is available as an Integrated Development Environment (IDE). While sketchpads can be developed in other IDEs like Eclipse, the HR3 IDE allows the staggering of the generation and interrogation phases, so codebases stay in memory while the user alters and executes the interrogation code repeatedly. As codebase generation can be slow, while interrogation isn't usually, this saves time.

Importantly (and somewhat ironically for a code generation system), no compilable code is generated until requested by the user, which is usually during the interrogation phase. To explain this, we note that in the worst-case scenario, automated code generation – for instance via a genetic programming (GP) approach – must: (i) generate a representation for a new program, e.g., by crossover and mutation of programs represented as trees (ii) translate the program into compilable code (iii) write this to a file (iv) compile the file into a program (v) run the program (vi) collate the output and (vii) analyse the output, e.g., to estimate fitness. The generation of millions of programs in this way can be slow, which is why in GP, there are many optimisations available. With HR3, for efficiency, we avoid stages (i) to (iv). That is, data is manipulated internally in such a way that it contains the output from methods that HR3 has invented. Standalone Java programs are not created, compiled and executed during codebase generation, but are rather produced on-demand during the interrogation phase later.

Production Rule Batch Applications

HR3 starts by executing the user-given background methods expressed as Java code in a sketchpad. The first background method always produces a set of string constants that act as labels for what we call data **records**. The other background methods flesh out the records by each producing an ordered list of tuples (one list per record) by generating data programmatically and/or reading it from a file. For example, in the datamining applications below, the background methods read data from a CSV file. The first background method generates record IDs using the line numbers in the file and the other methods each extract one value per line, producing singleton tuples in the ordered lists. In contrast, in the numerical discovery application below, the first background method generates a set of integers, and the others calculate the divisors and digits of each integer, producing different length tuples for different integers. The two generative art applications below similarly start with an empty database and background methods which generate appropriate data.

The background methods are taken as the **seed** codebase that HR3 will construct all future methods from. The user directs the usage of **production rules** (PRs, described below) which manipulate information about existing methods into that for a set of newly invented procedures. The application of a PR to an existing method generates new output as an ordered list of tuples for each record, and also a new procedure, represented as a tree capturing the series of PR steps used to construct it – see figure 1 for an example procedure. Because quite different procedures can produce the same output, we define a **method** as a pair which contains (i) the data output by the manipulations of the PRs for the method on the database, and (ii) a set of different procedures which produced that output.

Starting with the seed codebase, the repeated application of PRs to existing methods builds up the codebase. **Unary** PRs are applied to one existing method, and **binary** PRs are applied to two. Under normal operation, each production rule is applied to the **batch** of methods (unary) or pairs of methods (binary) in the codebase that the PR hasn't previously been applied to. For a background or generated method, m, and an ordered list of records R, we write $m(r)$ for the output of m when applied to a $r \in R$ and we write $m(R)$ for the ordered list of outputs of m when applied to every $r \in R$, in order. With this notation, the core components of a codebase containing H methods are:

- A set $R = \{r_1, \ldots, r_n\}$ of record IDs
- Sets C_i, C_f, C_s of integer, float and string constants resp.
- A set of methods $M = \{m_1, \ldots, m_H\}$, where $\forall\, i$:

$$m_i(R) = \langle \{m_i(r_1), \ldots, m_i(r_n)\}, \{proc_1, \ldots, proc_k\} \rangle$$

where $m_i(R)$ is an ordered list ranging over $r \in R$, such that $m_i(r)$ is a set of typed tuples of the same length, i.e., $m_i(r) = \{t_0, \ldots, t_n\}$ and $\forall\, t \in m_i(r), \exists\, l$ s.t. $t_i = \langle c_1, \ldots, c_l \rangle$, with each c_i being a member of C_i, C_f or C_s

- A set, E, of procedures for methods, m, where $\forall\, r \in R, m(r) = \{\}$

Informally, for every method HR3 invents, it records (a) the output of that method when applied to the database, as an ordered list of sets of tuples, one set per record, and (b) the set of procedures that generate methods producing exactly this output. HR3 also separately keeps a set of procedures that led to empty outputs for every record in the database.

The main innovation in the HR projects has been the use of production rules which generate data as positive examples of mathematical concepts (in HR1 and HR2) and output by procedures (in HR3). The data generation processes are cumulative, as they manipulate output from existing procedures into the output for multiple new ones. This is more efficient than starting from scratch each time with the data generated from the background methods, and is analogous to how GP approaches cache sub-tree outputs (Keijzer 2004). There are currently 27 PRs, split into six categories, given in the following table with an indication of whether they are unary (1), binary (2) or neither (0).

Logical:	conjunction(2), disjunction(2), existential(1), instantiation(1), inversion(1), negation(2), overlap(2), unifyVariables(1)
Mathematical:	exponential(1), trigonometry(1)
Meta:	cull(0), tag(0)
Numerical:	banding(1), bounds(1), count(1), interArithmetic(2), interNumCompare(2), intraArithmetic(1), intraNumCompare(1), makeFractional(1), round(1)
Programmatic:	interBitwise(2), randomChooser(1)
Statistical:	normalise(1), numSummary(2), rank(1) sampling(1)

A sequence of PR applications specified by the user in the sketchpad are carried out by HR3 during the codebase construction phase. Many PRs have a **parameterisation** specifying different ways they can be applied, e.g., the *numSummary* production rule calculates: min, max, mean, summation, standard deviation and range values. These can be all applied, or a subset specified with a parameterisation. With all possible parameterisations, the PRs produce 49 different types of Java statement. However, this belies a larger number, as there is a different application of the *instantiation* rule per constant in C_i, C_f and C_s, and multiple applications of *unifyVariables*, *overlap* and *existential*, depending on the types in the tuples output by the methods. The PRs in the *Meta* category don't produce new procedures, but rather *cull* selectively reduces the codebase for memory/time-intensive projects, and *tag* labels certain methods, so future PR steps can be applied only to batches of methods tagged appropriately. We include as much processing as possible like this at production rule level, to increase homogenuity.

The application of a unary PR to an existing method m involves first manipulating $m(r)$ for every $r \in R$ to produce a new set of tuples $m'(r)$, and then manipulating the procedure for m into a new one for m'. For each existing method in a batch application of a PR, HR3 cycles through all possible parameterisations of the PR, curtailed by the user if required. As an example, the *existential* production rule is parameterised by an integer which represents a position, p, in the tuples of $m(r)$. For a given record r, it operates by first removing the entry at position p from each tuple

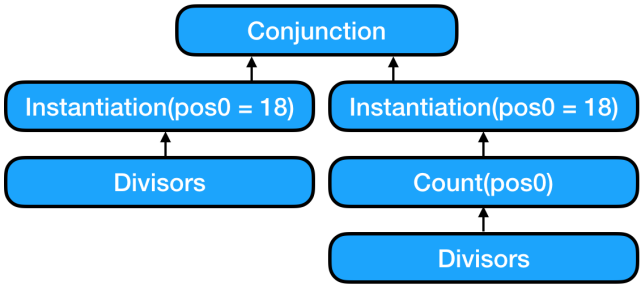

Figure 1: Example procedure for a Boolean method over integers, with PR name and parameterisations in brackets.

in $m(r)$, and then removing any repetitions in the resulting set, recording the remaining altered tuples in $m'(r)$. As another example, the *trigonometry* PR is also parameterised by a position p and cycles through all possibilities for this. It replaces the value x in each tuple at p with $t(x)$ cycling through $t \in \{sin, cos, tan\}$.

Each binary PR manipulates the output of two existing methods, e.g., given methods m_1 and m_2 and record r, the conjunction/disjunction/negation PRs produce tuples thus:

$$\begin{aligned} \textit{Conjunction:} \quad & m'(r) = \{t : t \in m_1(r) \text{ and } t \in m_2(r)\} \\ \textit{Disjunction:} \quad & m'(r) = \{t : t \in m_1(r) \text{ or } t \in m_2(r)\} \\ \textit{Negation:} \quad & m'(r) = \{t : t \in m_1(r) \text{ and } t \notin m_2(r)\} \end{aligned}$$

The storage, retrieval and manipulation of tuples in HR3 has been optimised to make the application of such binary production rules as efficient as possible, as there can be millions of applications of binary rules, with far fewer for unary PRs. With the *negation* PR, HR3 applies it to both pairs (m_1, m_2), and (m_2, m_1), and for all three of these binary PRs, HR3 checks that $m_1 \neq m_2$. As another example, the *interArithmetic* PR calculates new tuples by adding/subtracting/dividing/multiplying values in the same position in each pair of tuples of m_1 and m_2.

As mentioned above, different sequences of PR manipulations of the database can lead to different procedures that produce exactly the same output. It would be redundant to store this output repeatedly, so instead HR3 adds details of any procedure with exactly the same output as that in an existing method m, to the set of procedures that form part of m. It only adds a new method to the codebase if the output, $m(R)$, is distinct from all the other methods currently in the codebase. We have undertaken much experimentation with representation and hashing schemes, as the retrieval of a match to a new output is one of the slowest parts of HR3's process. Whenever HR3 invents a method m for which $\forall r \in R, m(r) = \{\}$, it does not add this method to the codebase, but stores the procedure for it in a set, E, along with others that also produced empty outputs.

On-Demand Code Generation and Inspection

After the codebase construction, HR3 moves onto the user's code in the sketchpad which details how to interrogate the codebase. HR3 can be instructed to turn a particular procedure from a method into executable Java code via the *Spoon API* for Java code generation (Pawlak et al. 2015). In the next section, we show how HR3 forms a codebase of methods which apply to integers, and from which the user extracts some coincidences. In one run of the sketchpad, HR3 invented a Boolean procedure which checks whether an integer is a multiple of 18 and has 18 divisors, as portrayed in figure 1. The code generated by HR3 for this procedure is given in figure 2. We see that the comments to the method include a definition in a mathematical form which refers to the code for calculating divisors, as given by the user. The comments also give a flattened version of the tree for the procedure and the set of examples (output) that HR3 calculated for it during the codebase construction.

The code in figure 2 is in its most compressed form, and refers directly to the PRs HR3 employed in constructing method number 7255. This presentation is for people who understand HR3's processing. However, the user can specify that HR3 replaces the calls to methods such as `count` and `instantiation` by code which manipulates data directly. If this is not specified, then the methods referred to in the body of `method7255` are included in the Java file, in addition to the background code, which is extracted from the sketchpad, to make the file stand-alone. To make the file executable, a suitable `main` method, which calls `method7255` with appropriate inputs, is also added.

The stand-alone code for a method provides an accurate representation of how it manipulates data, but also allows users to run the code on a larger set of records, e.g., the user could apply `method7255` to the integers between one and a million, even though the codebase was constructed using a much smaller set. The HR3 API enables users to search for methods, e.g., all Boolean methods which output true for a particular record, or all methods with more than 17 positive examples, etc. Users can also employ the API to output code for **conjectures** involving methods of interest, m. For instance, they can ask for **equivalence** conjectures, i.e., all the other methods m' which output the same or similar (based on average per-record set equality) tuples as m over the database, up to a user-given minimum **correctness level**, such as 90%. The Java for equivalences contains the code for particular procedures of m and m', as well as a `main` method to check equivalence of their outputs, enabling the user to check the conjecture over a larger set of inputs.

The user can also use HR3 to produce **implication** conjectures, where the output from method m' is a subset or superset of that of m, and **mutual-exclusion** conjectures, where methods m' are found which share little or no output with m, again with a minimum correctness level. The set E of empty procedures can also be the source of **non-existence** conjectures, and the user can ask for Java code to check these over a larger set of inputs. The HR3 API includes techniques for presenting, sorting and filtering conjectures, and for checking them against random data, as described below.

Random and Meta-Codebases

HR3 works directly with Java code, rather than a formalism like first order logic, so can in principle produce algorithms for any task, given the correct application of the right production rules. Such an expressive approach means that HR3 generates multiple methods which look different but produce the same output. In some cases, these highlight a

```
// Def: [a] : (18.0=|x1:(divisors(a,x1))|) & (divisors(a,18.0))
// Proc: [Conjunction,[Instantiation,[Count,[divisors,0],0],0,0,18],[Instantiation,[divisors,0],0,0,18]]
// Ex: 180, 252, 288, 396, 450, 468, 612, 684, 828, 882, 972, 1044, 1116, 1332, 1476, 1548, 1692, 1908
public ArrayList<Object[]> method7255_0() {
    ArrayList<Object[]> divisorsAsTuples = divisorsAsTuples(stringData);
    ArrayList<Object[]> instantiation_1 = instantiation(divisorsAsTuples, 0, 0, 18.0);
    ArrayList<Object[]> count_1 = count(divisorsAsTuples, 0);
    ArrayList<Object[]> instantiation_2 = instantiation(count_1, 0, 0, 18.0);
    return conjunction(instantiation_2, instantiation_1);
 }
```

Figure 2: Code fragment generated for the procedure given in figure 1.

discovery about the data, but in others the equivalence is due to the nature of the procedures alone. In the latter case, conjectures identifying such patterns are rarely interesting, and it is frustrating to check the conjecture only to find that it expresses an algorithmic tautology. The lack of formalism largely rules out a deductive approach to showing equivalence in these case. Instead, we implemented techniques to generate a **random codebase** by shuffling the data generated by the original background methods.

When the procedures involved in a conjecture are applied to the random data, if it is still true empirically, it can be fairly safely ignored, as the lack of semantics in the shuffled data indicates a very low probability that the nature of the data is responsible for the pattern expressed by the conjecture. Hence the only alternative is that the procedures themselves force any data into the pattern, and so the conjecture is not interesting. During codebase interrogation, the user can instruct HR3 to employ this method to filter out such uninteresting conjectures. The usage of random codebases is covered in more detail in (Colton, Ramezani, and Llano 2014), and it suffices here to note that the approach can be used on any conjecture type. Moreover, HR3 employs random codebases, along with random sequences of PR applications, to check that the on-demand code it produces for method m outputs the same as the $m(R)$ calculated during the codebase construction phase. Used during the development of new PRs, this has occasionally highlighted mismatches which have been corrected.

Given a **ground codebase**, G, HR3 constructs a **meta-codebase**, G^M, by first taking all the methods of G as the records in G^M, and then giving each distinct tuple of constants in $m(R)$ (for any method m in G) a unique label. It then automatically constructs and adds a single background method, b, to G^M. The output $b(r)$ for a record r in G^M is a set of singleton tuples, with each of these meta-tuples containing a label corresponding to the ground-tuple in G that the ground-method in G (corresponding to r in G^M), outputs. The user can also specify that HR3 adds some additional background methods to G^M that (i) calculate values based on what the ground methods output (ii) express conjectures about the ground methods, and (iii) capture how the methods were constructed, using the procedures in G. For instance, HR3 can add a background method to the meta-codebase which outputs the list of production rule names that went into constructing the ground methods. Once constructed in this fashion, G^M is ready for the codebase construction stage, and HR3 can construct meta-methods which highlight discoveries about the ground codebase.

Example Applications

We aim here to show robustness to different tasks/domains, rather than how successful HR3 is for a particular application or providing operational statistics, etc. We show our ideology of expressing different AI tasks as code generation problems in action, and highlight the practical usage of HR3 sketchpads to achieve goals in creative projects.

Mathematical Discovery

HR1 and HR2 were both effective in number theory, generating new integer sequences and making conjectures (Colton, Bundy, and Walsh 2000). Applying HR3 to such tasks, we employ the following simple sketchpad file:

```
package projects.maths_discovery.integer_sequences;
import java.util.ArrayList;
import ide.Sketchpad;
import production_rules.PR;

public class IntegerSequences extends HR3Sketchpad {

 public ArrayList<String> makeIntegers(int l, int u) {
   // Code generating integers between l and u
 }

 public ArrayList<Integer> divisors(String n) {
   // Code calculating divisors of n
 }

 public ArrayList<Integer> digits(String n) {
   // Code calculating digits of n
 }

 public void generateCodeBase() {
  generateRecords(IntegerSequences.class,
                  "makeIntegers", 1, 1000);
  addBackgroundMethods(IntegerSequences.class,
                       "divisors", "digits");
  applyPR(PR.Count);
  applyPR(PR.Conjunction);
  applyPR(PR.Existential);
  applyPR(PR.Instantiation, "useInteger:1,2,3");
  applyPR(PR.Count);
  applyPR(PR.Inversion, "useArity:1");
  applyPR(PR.Conjunction, "repeats:2");
 }

 public void interrogateCodeBase() {
  ArrayList<Integer> methodNums =
     getBooleanMethodsTrueFor("23", "53", "73", "113");
  printSeparator();
  println(methodNums.size() + " methods");
  for (Integer mNum : methodNums) {
    println(mNum + ". " + getDefinition(mNum, 0) + "["
            + getOutput(mNum) + "]");
  }
 }

}
```

This contains three background methods (bodies omitted for brevity), plus a method for generating the codebase and one for interrogating it. The code for generating integers, and calculating the divisors and digits of an integer take the most natural `Integer` input and output `ArrayLists` of `Integers`. The `IntegerSequence` class uses API methods inherited from `HRSketchpad`, including `generateRecords`, `addBackgroundMethods`,

```
Prod Rule     Progress Procedures Different Boolean  Out-Repeats Out-Empties  Tuples  Memory(Gb)  Time(ms)  Methods/s
-------------------------------------------------------------------------------------------------------------------
Background    ........      2          2          0          0           0        8771     0.00415       72         28
Count         ........      4          4          0          0           0        9866     0.00441       89         45
Conjunction   ........     10         10          0          0           0        9866     0.00562       95        105
Existential   ........     20         16          6          0           0       10866     0.00876      108        185
Instantiation ........     50         40         30          5           0       10866     0.00651      131        382
Count         ........     58         47         30          6           0       10903     0.01073      137        423
Inversion     ........     88         77         60          6           0       10903     0.01802      150        587
Conjunction   ........   1994        792        743        646         551       10903     0.08788      324       6154
Conjunction   ........ 276917      16897      16817     198056       61959       10903     1.54067    32914       8413
-------------------------------------------------------------------------------------------------------------------
1314 methods
15. exists x1 ((x1=|x2:(digits(a,x2))|) & (x1=|x3:(divisors(a,x3))|))[1, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, ...]
20. digits(a,3.0)[3, 13, 23, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 43, 53, 63, 73, 83, 93, 103, 113, 123, 130, ...]
22. 2.0=|x1:(divisors(a,x1))|[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, ...]
25. 2.0=|x1:(digits(a,x1))|[10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, ...]
...
53. -(divisors(a,2.0))[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, ...]
54. -(divisors(a,3.0))[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28, 29, 31, 32, 34, 35, ...]
...
```

Figure 3: Example HR3 output from the `IntegerSequences.java` sketchpad given above.

`applyPR` and `getBooleanMethodsTrueFor`, and there are many more available. In the sketchpad, the user specifies generating integers 1 to 1,000 for R, and provides some batch parameters to focus the application of the PRs, i.e., specifying that *instantiation* should only use integers 1, 2 and 3; *inversion* (which finds the complement of tuple sets) should only be applied to methods of arity 1; and the final *conjunction* batch step should be repeated twice. The interrogation of the codeBase in the sketchpad is a puzzle: what do the numbers 23, 53, 73 and 113 have in common?

The output from running this sketchpad is in figure 3. We see that – using a single thread on a 2.6Ghz MacBook Pro laptop – the generation phase took around 33 seconds, and HR3 generated and tested 276,917 procedures (at 8,413 per second), producing 16,897 different methods (in terms of procedure outputs), of which 16,817 were Boolean. 198,056 PR steps generated procedures with the same output as one in a method already in the codebase, and 61,959 steps produced an empty procedure. HR3 answers the puzzle with 1,314 procedures, with six given in figure 3: the integers 23, 53, 73, 113 have the same number of (distinct) digits as divisors, namely 2, making them prime numbers, they all have the digit 3 in them, but are not divisible by 2 or 3.

To take the application further than previously with HR2, we dropped the constraints on the *instantiation* PR, ran the codebase generation again, and altered the sketchpad `interrogateCodeBase` instructions. In particular, we added code to produce a meta-codebase, G^M, from the ground one G. Recalling that each record in G^M represents a method in G, the background methods in G^M were specified to be both the tuple labels (see above) and the production rule steps for each procedure of m in G. Using the `applyPR` API call, we applied the *count* and *banding* PRs to build a meta-codebase. Using API calls interrogating G^M and cross-referencing the methods in G, with around 10 lines of bespoke code in the sketchpad, we extracted all methods of G which employed the *instantiation* PR grounding variables to a particular number n which was also equal to the number of tuples output by G. We then interpreted these methods and their output as numerical coincidences.

The output was slim, and we were able to cherrypick and tweet some of the more interesting coincidences, like: "Did you know that between 1 and 1,000, there are 17 multiples of 17 with the digit 7 in them, and 18 multiples of 18 with an 8 in them?" and: "Between 1 and 1,000, there are 19 primes with a 1 and a 9 in them, and 54 numbers with a 5 and a 4 in them". Running the sketchpad repeatedly with different integer ranges, we produced more results, such as: "Between 1 and 10,000, there are 36 multiples of 36 with a 3 and a 6 in them, and 45 multiples of 45 with a 4 and a 5 in them" and: "Between 1 and 2018, there are 18 multiples of 18 with exactly 18 divisors". It is beyond the scope of this paper (where we are assessing the generality, rather than the power, of the approach) to evaluate this application thoroughly. However, we are currently studying the value (if any) of coincidences for everyday creativity, and aim to use HR3 to find coincidences in text and other media.

Datamining

We re-frame datamining as the automatic generation of triples of algorithms which are related via given data. Standard association rule (AR) mining extracts relationships of the form $a = v_1 \wedge b = v_2 \wedge c = v_3 \rightarrow d = v_4 \wedge e = v_5$. While this representation is useful for understanding discoveries about the data, for it to be used operationally (e.g., to see if it holds for a different dataset), the AR will need to be expressed or interpreted as code. In this context, we see that the left hand side (LHS) of the AR is captured by an algorithm extracting all data records from a database for which the value in column a is v_1, in column b is v_2 and column c is v_3, with the algorithm for the RHS similar. The implication of the AR is a third algorithm which relates the LHS and RHS algorithms, in this case checking whether the output of the LHS is a subset of the output for the RHS algorithm.

For various (good) reasons of efficiency, correctness checking and avoiding redundancy, standard datamining is usually limited to finding ARs of the above form, possibly with negation added. HR3 can perform datamining in this standard way by constructing a codebase with only the *instantiation* and *conjunction* PRs on data supplied in a CSV file, then using implication conjecture making at the interrogation stage. However, HR3 is able to construct and investigate a much richer variety of algorithms for the LHS and RHS of ARs, but currently the relating algorithm is fixed to the implication (subset), equivalence (equality) and mutual-exclusivity conjectures mentioned above. For instance, by employing the *negate* PR, HR3 can mine ARs with LHS and RHS such as: $a \neq v_1 \wedge b = v_2 \wedge c \neq v_3$. By also employing

exists, this extends to: $a = b \wedge c = v_3$ and the arithmetic PRs enable constructions such as : $a + b = v_1 \wedge c \neq d$, etc.

Re-framed thus, we applied code generation to datamining two substantial datasets. The first contains the position, size and colour of around 1.9m GUI elements from roughly 10,000 Android app screens. We used HR3 to find ARs linking GUI elements to the score on the Android store. The second dataset contains traces from simple arcade-style videogames, compiled to construct a forward model for the game (Dockhorn and Appledoorn 2018). We also used HR3 to mine ARs that pay into a forward model for each game. Both applications were successful, and HR3 was able to generate thousands of useful conjectures containing statements of various types as above, interpreted as association rules. The API enabled us to sample the data for efficiency, then run generated Java code over the entire dataset to check the validity of interesting ARs. We used the API to calculate support, confidence and Z-values for each AR, with the latter being very useful in sorting results, as high Z-values often indicate surprising, yet well-supported results. The correctness minimum limit was also very useful for experimentation, and the GUI staggering of codebase generation and interrogation saved much time. In both cases, we used random codebases to discard dull conjectures, which worked very well, removing large numbers of conjectures, without (on inspection) discarding any interesting ones.

Generative Art

To expand the tasks HR3 can be applied to, we looked at pixel-based art of the kind generated in (Sims 1991). Background methods in the sketchpad produce record IDs as (x, y) pixels in ranges specified by the user (normally 250×250), and extract the x and the y coordinates. The code generation phase involves initially applying the *trigonometry*, *exponential*, *interArithmetic* and *conjunction* PRs in batches. This produces thousands of large, incomprehensible, functions (with 50+ nodes in the procedure tree), which calculate an output for each pixel based on its coordinates, using trigonomety, surds, exponentials and arithmetic. Codebase construction ends with (a) the *makeFractional* PR removing integer parts of outputs, (b) *normalise* mapping outputs to integers in the range 0 to 255, and (c) *overlap* constructing methods which output triples of these integers. Tagging is employed so only methods output by the previous PR step are employed in the next one, which accelerates a search for more complex procedures, required here.

Around 20 lines of code were added to the sketchpad to take each method outputting triples, and interpret the output as (r, g, b) values in a `BufferedImage` object. For any images of interest, the user generated Java code for the corresponding method, which produced larger (4000×4000 pixel) images for high-res printing and screen display (see figure 4(a)). The intended artwork for this project (entitled *Style Please* as a pun on the phrase 'Style Police') was a montage of a face which changes styles over time. Meta-level codebase generation was employed to collate sets of images in a particular style. In particular, the outputs from the ground methods were processed using the *banding* PR at the meta-level, followed by the *count* PR and another

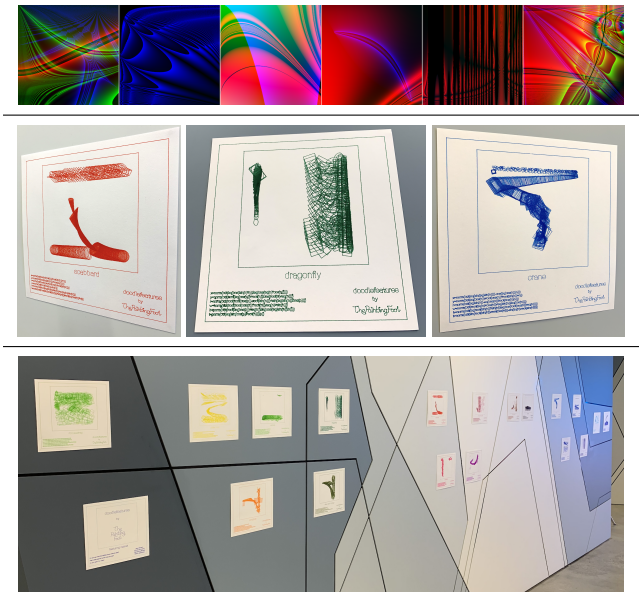

Figure 4: (a) pixel-based generated images (b) the 'scabbard', 'dragonfly' and 'crane' plotted artworks from HR3 generated images (c) pop-up exhibition 'DoodleFeatures'.

banding step. This identified images (i.e., sets of (r,g,b) triples in the method outputs) where, for example, the number of green(ish) pixels was higher than that of red ones, and many other visually obvious styles, such as greyscale, rough/smooth textured, monotone, etc. 100,000 images of (100×100) pixels were produced to provide material for 50 different styles of montage, and the artwork was cycled through them on a 3m by 2m screen for a day.

In another generative art project, we used an AxiDraw plotter to physically produce abstract art pieces, employing HR3 within The Painting Fool project (Colton 2011). Similarly to the pixel-based art, the project sketchpad directed HR3 to produce methods which output a sextuplet for inputs ranging over the integers 1 to n (for a changeable n), rather than coordinates. It used the same codebase generation phase as previously, but with additional **overlap** steps at the end. The sextuplet for an input x were interpreted as the (i) x coordinate (ii) y coordinate (iii) rotation (iv) width (v) height, and (vi) shape type [circle, square, triangle] for a geometric shape that the plotter is able to draw.

Code was added to the sketchpad to (a) render the sequences of n shapes onto a `BufferedImage` to save, and (b) write out the quadruples for each sequence into a Javascript file to be read by the AxiDraw plotter. 100,000 images were passed through a pre-trained ResNet neural model (Krizhevsky, Sutskever, and Hinton 2012) which categorises images into one of around 1,000 classes, corresponding to real-world objects like 'umbrella' and environments such as 'seashore'. All the roughly 100 images which scored 0.8 or above (indicating that ResNet was certain that the images looked like exemplars of the category) were inspected and 18 chosen for a pop-up exhibition called 'DoodleFeatures', as portrayed in figure 4. For each, the ResNet category and a representation of the rendering method was added to the Javascript before this directed an AxiDraw plot.

Conclusions and Future Work

In addition to the applications above, HR3 has solved the Countdown Numbers game (Colton 2014), performed invariant discovery in formal methods and addressed dynamic investigation problems (Colton, Ramezani, and Llano 2014). We have concentrated here on breadth of applications, but we plan more in-depth evaluation of HR3's strength for particular applications, and its ability to empower people in a co-creative setting. We believe it significant that code generation has been applied to quite different tasks across application domains. The flexibility of the HR3 approach comes via: casting disparate AI tasks as automated programming; the production rule approach, gaining efficiency by separating code generation from output generation; using meta-level codebases for support tasks, and random codebases to help find the most interesting methods produced.

In the mathematical discovery and generative art applications, additional sketchpad code was needed from the user to complete the project, which indicates room for improvement, as HR3 should be able to generate support code. That said, the meta-codebase generation did help with aspects of the support code, and we have only just begun to explore the affordances of meta codebases. We plan to expand the domains and tasks to which HR3 can be applied, including producing glue code (Liu, Bastani, and Yen 2006); data compression; image filtering; and program synthesis tasks (Gulwani, Polozov, and Singh 2017). This latter application will likely require more goal-based search than is currently implemented in HR3. We also plan to add more automation to the approach, which currently relies too much on the user correctly organising production rule steps in the sketchpad. We aim for a (different) meta-level approach, where HR3 can write its own sketchpads to control code generation, so the user can supply just some background code and/or data.

We also aim for HR3 to be more intelligent in the application of the production rules, for instance, with better abilities to work with functions producing unique outputs, i.e., avoiding PR applications which will certainly lead to empty methods. We also plan for it to output code in different programming languages to Java and for it to improve as a programmer, with (a) more production rules increasing its expressivity, especially with programmatic constructs such as loops and conditionals (b) more sophisticated code styles employing techniques like inlining and variable naming, and (c) more access to relevant data types such as images.

Returning to our ideology, we see that HR3's generated code has been applied across generative, analytic and support tasks, across domains, and (in the generative art example) the image generation code is too large to comprehend, which contrasts with the more comprehensible output in the datamining and mathematical discovery applications. Hence the HR3 implementation blurs the distinctions given above into a continuum. It has problematised the world and introduced artistic affordances by generating stand-alone code inspected in its own right. We plan to add framing abilities so that HR3 can explain and motivate the problems it introduces, and suggest ways to capitalise on new affordances.

Software systems written to be taken seriously as creative in their own right, often suffer criticism that the human programmer is the creative one, with the software a productivity, or at best, inspiration tool. We plan for future versions of HR3 to alter their own code in an attempt to improve its abilities, and contribute code to other creative AI projects. In this way, we hope to argue that the software is fully independent and hence worthy of being talked about as creative.

Acknowledgements

We wish to thank the anonymous reviewers for their very helpful input. The third author was supported by a Research Fellowship from the Royal Academy of Engineering.

References

Charnley, J.; Pease, A.; and Colton, S. 2012. On the notion of framing in computational creativity. In *Proc ICCC*.

Colton, S., and Muggleton, S. 2006. Mathematical applications of ILP. *Machine Learning* 64.

Colton, S.; Bundy, A.; and Walsh, T. 2000. Automatic invention of integer sequences. In *Proc. AAAI*.

Colton, S.; Pease, A.; and Saunders, R. 2018. Issues of authenticity in autonomously creative systems. In *Proc. ICCC*.

Colton, S.; Powley, E.; and Cook, M. 2018. Investigating and automating the creative act of software engineering. In *Proc ICCC*.

Colton, S.; Ramezani, R.; and Llano, T. 2014. The HR3 discovery system: Design decisions and implementation details. In *Proc. AISB Symposium on Scientific Discovery*.

Colton, S. 2002. *Automated Theory Formation in Pure Mathematics*. Springer.

Colton, S. 2004. Automated conjecture making in number theory using HR, Otter and Maple. *J. Symbolic Computation* 39(5).

Colton, S. 2011. The Painting Fool: Stories from building an automated painter. In McCormack, J., and d'Inverno, M., eds., *Computers and Creativity*. Springer.

Colton, S. 2014. Countdown numbers game: Solved, analysed, extended. In *Proc. AISB Symposium on AI and Games*.

Dockhorn, A., and Appledoorn, D. 2018. Forward model approximation for general video game learning. In *Proc. IEEE Conf. on Computational Intelligence and Games*.

Fiat, A., and Woeginger, G., eds. 1998. *Online Algorithms: The State of the Art*. Springer.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program synthesis. *Foundations & Trends in Prog. Languages* 4(1).

Keijzer, M. 2004. Alternatives in subtree caching for genetic programming. In *Proc. of the EuroGP conference*.

Krawiec, K. 2016. *Behavioral Program Synthesis with Genetic Programming*. Springer.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

Liu, J.; Bastani, F.; and Yen, I. 2006. Glue code synthesis for distributed software programming. In *Advances in Systems, Computing Sciences and Software Engineering*. Springer.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing* 8(4).

Pawlak, R.; Monperrus, M.; Petitprez, N.; Noguera, C.; Seinturier, L. 2015. Spoon: A library for implementing analyses & transformations of Java code. *Software: Practice & Experience* 46.

Redfern, D. 1999. *The Maple Handbook*. Springer.

Sims, K. 1991. Artificial evolution for computer graphics. *Computer Graphics* 25(4):319–328.

Sorge, V.; Meier, A.; McCasland, R.; and Colton, S. 2008. Automatic construction and verification of isotopy invariants. *Journal of Automated Reasoning* 40(2-3).