6.945 Final Project Report

Yuri D. Lensky, Nathan Villagaray-Carski, Kelly Casteel

May 20, 2013

1 Domain

This project is intended to be a starting point for building platforms that automate and aid the study of model-able systems. Such a description, although accurate, is close to useless, so an attempt will be made here to describe the domain by way of the development process, an approach possibly justifiable by invoking a loose version of a mantra espoused by Professor Gerald Sussman: that the best way to teach something is to describe it in the precise language of computer code. Here is described the process of that translation from idea to code.

The initial inspiration for this work was the observation that several meaningful and powerful results in physics can be produced by relatively simple and apparently mechanical symbolic manipulations. Unfortunately, spontaneously generating such results on a computer, even ones as basic as the conservation of energy, proved exceedingly difficult; several approaches were tried, but any interesting results only came after significant massaging of the input and algorithms.

From here, on the advice of Professor Sussman, the Automated Mathematician (AM) software by Doug Lenat was studied for its applications in this domain. Although it appeared very relevant, the algorithms and concepts were very specific to mathematics — so much so, that they were not usable for systems that assumed and built on a large amount of mathematical concepts and incorporated external, "realistic" assumptions. Note that this was also recognized by Doug Lenat himself, who, frustrated that his system was not extensible enough, produced two new pieces of software: Eurisko and Cyc[1].

Having been repeatedly beaten back by the difficulty of automating symbolic analysis, the program took another decisive turn in light of the philosophies of Professors Sussman and Adams;

[1] See Doug Lenat's paper at `http://eksl.isi.edu/files/library/Lenat_Brown-1984-why-AM-and-EURISKO-work.pdf` for his own criticisms of AM

the former strongly believes in and based a class around computational classical mechanics, where actual examples of motion are numerically calculated and plotted, and the latter likes to explain away the weirdness of quantum mechanics by relying a view of physics not as something to be necessarily understood by people, but used by them to explain the world. This suggests that perhaps in exploring physics, it is most logical and efficient to make use of a computer's strongest facilities and advantages with respect to humans — raw calculation power. Thus the computer is used to generate examples of interesting (determined by the user) quantities that satisfy initial conditions and the laws of the system being modeled. These are then numerically studied by the program to find functions that, given these "interesting" quantities, produce equally "interesting" outputs.

Such an approach has potential applications in the study of all the sciences, physical or social, engineering, optimization problems, and essentially any complex system where large amounts of data can be obtained either by generation in the computer or experimentally.

2 Method description

The system is split into two large parts — one, fairly robust and completely prototyped, focuses on defining laws and generating meaningful examples from those laws in a very extensible framework. The second, at a heavily experimental and dynamic stage, is the system of generating and exploring relationships between interesting values associated with examples of laws. Each piece may be used independently, and so their description will take this into account.

2.1 The law datatype and related operations

The code described in this section is in the file "laws.scm".

A law has the following properties:

name Aesthetic symbol used to visually identify the law from others to the user.

example sets A list of example sets that contain tables with sample value that satisfy the law.

input specifications A list of symbolic expressions that specify both names for and functions that generate various values that need to be related in some way to satisfy a law. See Section 2.1.2 for more information.

output specification A list of symbolic expressions that specify both names for and functions that generate from variables in the input specifications various fundamental values that may be of interest in future analysis. See Section 2.1.3.

2.1.1 Example sets

Example sets consist of a symbolic (aesthetic) name, initial conditions which specify the values in the example, and a results table of simultaneously law- and initial condition- satisfying output values.

2.1.2 Input specifications

Input specifications are lists whose first element is a symbol that names the input, and whose remainder consists of an arbitrary number (0 to memory limits) of generation specifications that are used to generate values of the input from initial conditions and other inputs given that the particular input is not passed as an initial condition itself.

Generation specifications are in turn lists of two elements. The first is a list of variables that must have values in order for the generator to run — effectively, the dependencies of the generating function. The second is a quoted lisp expression that can be evaluated in the REPL environment and returns a type appropriate to the relevant variable.

When choosing generation functions, the first one that has all required variables defined will be chosen. Note this is different from choosing the most specific one, for various definitions of specific.

2.1.3 Output specification

Output specifications are again two-element lists. The first element is a symbol to name the variable, and the second is a quoted lisp expression that is to be run in the REPL environment with additional bindings for each of the input variables per their names in the input specifications.

2.1.4 Creating and working with laws

Laws are created with the function `make-law` which takes arguments in the order outlined in Section 2.1. An example law for Newton's one dimensional force law is show in Listing 1.

Listing 1: Law definition for $-\frac{\partial V}{\partial x} = \dot{p_x}$.

```
(define newton-F
  (make-law 'newton-F '()
            '((V)
              (P ((m X) (X->constant-m-P m X)))
              (X)
              (t (() (random-float))))
            '((Vx (V (X t)))
              (Pt (P t))
              (Xt (X t))
              (T t))))
```

Initial conditions are quoted lists in the format of the first argument to a **`let*`** statement. Any number of variables can be defined, and re-used after they are defined, but at least enough initial conditions to fill each input of a law is required. An example of valid initial conditions is given in Listing 2.

Listing 2: Sample initial conditions for the law in Listing 1.

```
(define harmonic-initial-conditions
  '((k 3)
    (m 3)
    (V (harmonic-V k))
    (X (harmonic-x k m))))
```

From these initial conditions and the law, an example set can be generated: the function `generate-example-set` takes a name, a law, initial conditions, and a number of examples to be generated. See Listing 3.

Listing 3: Example set creation.

```
(define harmonic-example-set (generate-example-set 'harmonic newton-F
   harmonic-initial-conditions 30))
```

This works by calling a function `make-example-generator`, which produces a function that, when called, generates new examples that satisfy both the initial conditions and the law itself.

2.2 Analyzing example data

All the code described in Section 2.1 is robust, fast, and easily extensible. The remainder of the code described here is very experimental.

The code described here is in the file “oam.scm”.

Although many approaches had been tried in order to effectively generate functions that would combine the columns of the example tables in interesting ways, the current implementation is heavily stream-based. The primary function of interest is the `make-term-generator-stream`, which takes a number of arguments its generated functions should be able to handle, and returns a stream of streams of functions, with the top-level stream being a finite stream indexed over possible combinations of arguments used (i.e. in a list of 4 arguments, functions that combine arguments 3 and 4 would be in a separate stream from functions that just use 3 or 4, or any combination of arguments other than 3 and 4). Although this stream can be collapsed with the provided `stream-collapse` method, it is separated this way intentionally.

2.2.1 Integration with laws

Although not currently integrated with the laws code, this function generation framework is intended to be, and built so that it is relatively simple.

3 Problems and future work

There is no shortage of future work to do on this system. First, basic deficiencies in the system need to be repaired before it is use-able — these will be considered “bugs” and are discussed in Section 3.1 . Second, improvements to the algorithms and approaches used in the system can be augmented greatly, as this is a novel approach to science and as such has many unknown and non-obvious properties — these will be considered “improvements” and are discussed in Section 3.2.

3.1 Bugs

All the glaring problems with this code lie in the analysis system described in Section 2.2.

The way the stream approach to function generation currently works is as follows. Call N the number of arguments passed to any function you want to generate. Each generated function can actually use anywhere between 1 and N arguments of those arguments. Streams are generated of functions that take a certain combination, and these streams are in turn part of a stream that contains all possible such streams. This works well, and the reason for the division was to allow for a heuristic that says, for example, "explore each combination of used arguments up to n times before moving to the next one".

The most severe bug enters in the expression that makes a function that takes n arguments, `make-n-function-stream`. Functions of one argument are easy (things like exponentials, trigonometric functions, etc). But properly generating functions of $n > 1$ arguments is very poorly implemented. The author has yet to find a good way of generating and intermingling functions such as (assuming a_n is argument n) $e^(a_1 + a_2 + a_3)$ and $e^(a_1 + a_2 a_3)$. So although the functions generated were sufficient to arrive at a simple linear relation like $V + Ap^2$ where A is some constant, more complicated and interesting relations are beyond reach. This is a fundamental limitation and needs serious consideration.

The other obvious bug is that there are no built-in functions for exploring examples or laws or collections of laws. This is a result of being unable to settle on a consistent interface, not a limitation of current systems.

3.2 Improvements

After settling on an interface to properly generate term-generating functions, the first task would be to create functions to explore laws and examples. These would work by generating linear combinations of terms, and then, to check if they are interesting, for N terms $N - 1$ differences would be taken between terms across rows, and the (existent or non-existent) solution of the resulting set of linear equations would give coefficients that make the combination of terms return a consistent value across rows. So, in the simple case of two terms, i.e. potential energy V and momentum squared p^2, with row values V_n and p_n, if we could find some constant A that ensured $V + Ap^2$ was constant across rows, it is true that $(V_i - V_j) + A(p_i^2 - p_j^2) = 0$, and $A = (V_j - V_i)/(p_i^2 - p_j^2)$. Another improvement would be to then re-express A in terms of constants of the problem, which would eventually allow us to derive, for example, conservation of energy.

The other large undertaking would be to figure out a good way to tie heuristics into the function generation more closely. In the file "stream-utils" there are some fairly generic stream-combination utilities that could easily be extended to support random or functional selection of elements from different streams, and a stream interface that supports saving and restoring streams, but neither these functions nor the stream interface is particularly easy to use. This is all complicated by the fact that there is no standard for discerning good heuristics from bad ones.

There is thus a great deal of work to be done, and the author hopes to get a chance to do so in the near future.

4 Alternative Approach

We also tried a completely different approach to deriving useful results using symbolic manipulations on a starting set of equations and rules. At a high level, this approach models the equation space as a graph and uses a conventional search algorithm to search through the graph for a meaningful result. This attempt was ultimately unsuccessful, but is described here for reference.

4.1 Best-first Search (search.scm)

This search algorithm explores the expression space in an efficient manner. It is highly general and can efficiently explore any space that can be modeled as a graph. In general, the algorithm explores nodes in the graph, exploring the “best” unexplored node first. The algorithm runs until it finds a node that is “good enough” or it reaches a state where it has not found an improved node in a “some number” of iterations. “Best,” “good enough,” and “some number” are all general notions defined by the user of the algorithm.

When a node is expanded, it generates a “summary” of information about the node which will be used to compare and determine which nodes are “better” and more promising to expand later. The summary can contain information about the node itself, the path to the node, and the summaries of any other node. A user-supplied function describes how to compare two different summaries.

The search function can be called as follows:

```
(search node generate-summary summary-lt? good-enough
   max-nonimproved-expansions get-neighbors traverse-edge)
```

Arguments:

- node - A node in the graph from which to start exploration.
- generate-summary - A function that computes a summary for a given node in the graph. It takes the following arguments:
  - node - the previous node in the path
  - new-node - the node to summarize

  - prev - a hash table mapping $nodeA \rightarrow nodeB$, where $nodeB$ is the previous node in the best path to $nodeA$
  - summary - a hash table mapping nodes to their corresponding summaries

- summary-lt? - A function that returns whether the first summary is “less than” the second summary (i.e. whether the second summary is better than the first summary)
- good-enough - A function that determines whether the given node (and best path to it) is good enough to halt the search and present to the user
- max-nonimproved-expansions - An integer indicating the maximum number of expansions without finding a better node before stopping. If the algorithm expands <max-nonimproved-expansions> nodes without improving the best node, it will halt and present the best node found to the user.
- get-neighbors - A function that returns the edges leaving the given node
- traverse-edge - A function that returns the node that is arrived at by traversing the given edge from the given node

4.1.1 Algorithm Specifics

The algorithm itself is a fairly standard best-first-search (`https://en.wikipedia.org/wiki/Best-first_search`) implementation.

One interesting aspect while coding this was that Scheme does not provide a standard heap implementation, so a simple heap library was created (heap.scm). It uses a weight-balanced tree as the main backing and a hash table for value->key lookup (necessary for the heap-decrease-key! function used in the search algorithm). The keys for the wt-tree are the summaries, and the values are the nodes. Since weight-balanced trees require unique keys (and my search algorithm does not require unique node summaries), a small hack to use the summary and the node as the key (and wrap the user-supplied comparator function appropriately) was implemented.

4.2 Application to the Problem

In order for this search algorithm to be useful to our problem, the expression space is modeled as a graph. Nodes correspond to a single expression, and an edge from expression A to expression

B corresponds to a derivation, where A and B are different representations of equivalent expressions. The user specifies the summary function, a function for comparing the summaries, and a halting condition.

4.2.1 Results

Unfortunately, we were unable to get this approach working, as we spent much of the time developing the generalized Best-first-search architecture, hoping to incorporate modeling the equation space as a graph at a later point. This proved more difficult than expected - we tried using scmutils to back the derivations, but were not able to get this working. In particular, we had trouble writing appropriate functions to use for the get-applicable-edges (which would return a list of rules that could be applied to a given equation) and apply-edge (which would apply a given rule to a given equation) functions. In hindsight, we perhaps would have had more success manually creating equations and rules using the generic operator framework introduced in the psets, instead of attempting to use scmutils for this purpose.