The H-Machine

L. Stabile

May 12, 2017

Abstract

The H-Machine is a hypergraph-based language and interpreter, based on rule matching by hypergraph isomorphism. Rules are part of the hypergraph being constructed, and can be matched and modified as can other data, thus forming a complete meta-system. The H-Machine language is extremely simple, yet can express a wide range of data structures and computations, with inherent parallelism. The matching system allows a full range of expression of recursive relations, implicit iteration, and, via meta-rules, a modular way to describe computation at a high level.

This paper describes the H-Machine – the language and an implementation – and illustrates its capabilities using several examples: Building a bottom-up FFT butterfly dataflow network organically, starting from the top-down recursive equations; using meta-rules to generate rules for compactness of expression; using meta-rules to modify, copy, and propagate other rules for optimization; and modifying behavior using simple modular expressions which specify the interaction among data structures, in this case interactions between the FFT structures, iterations of the Rule 30 cellular automaton, and a color circle.

Data and rules have a simple graphical interpretation which offers good visibility into the resulting structures. The examples illustrated show top-down tree structures, derived data flow graphs, cellular automaton matrices, ancillary relations, and rules – predicate subgraph, edges to be added, edges to be deleted -- within a single graphical model, using the Graphviz toolkit for rendering. Implementation is briefly described, with links to code and a gallery of generated H-Machine runs.

Introduction

Chaos and Complexity [18], sketched a graph automaton system to ground the philosophical ideas which were the main topics of that paper. The H-Machine (H for "hypergraph") was defined abstractly therein, with the following basic structure.

A hypergraph G is established which grows according to rules. Rules are themselves subgraphs of G, and have a predicate part and a consequent part. The predicate is matched by isomorphism against other subgraphs of G, and the mappings of the pattern to the data nodes and edges are used to instantiate new edges defined in the consequent.

Hypergraphs are useful due to their generality. The abstract H-Machine puts no constraints on the size of edges. A distinct advantage of this graph representation is the ability to mix freely data and metadata, where the term "metadata" applies both to application domains and meta-domains, as well as to internal structures, specifically that rules may freely match against, create, and modify other rules.

Starting with an initial graph G_0, the graph evolves as follows:

$$G_n = G_{n-1} \cup \bigcup_{i,j} r\left(subgraphs(G_{n-1})_i, (subgraphs(G_{n-1})_j\right)$$

where

$r(h_r, h_d) = \phi$ if h_r is not a rule or is not isomorphic to h_d; else a graph, the new edges.

$subgraphs(h) \equiv$ collection of all subgraphs of h

A straightforward evaluation of the above implies finding all subgraphs of G_{n-1} and matching each rule subgraph against each other subgraph (rule and non-rule). [18] expands on this formulation and discusses its overall computational and philosophical implications.

Facing Reality

To build a practical version of the H-Machine, the above abstract formulation needs considerable modification. Relevant goals, decisions, and conclusions may be summarized as follows:

1. While the primary goal is something that runs with considerable parallelism, a reasonably practical sequential system is necessary at least for development purposes.
2. Unordered sets, while mathematically pure, are very inefficient. After much experimentation, we settled on a hypergraph model whose edges are ordered multi-sets, i.e., n-tuples. Edge size remains unconstrained.
3. Pure isomorphism is a good model for matching, and the H-Machine description in [18] explicitly eschews variables as part of the syntax and semantics of matching. In the implementation, variables are introduced; however they are only syntactic markers for optimization. Variables may be handled as data as may any other node, and we'll see that this is a very useful property in particular for rules that manipulate rules.
4. The notion of locality in the graph, and developing a spreading-activation model, is crucial to its efficiency. Obviously testing each subgraph is not a practical direction.
5. The non-deterministic aspects of the H-Machine, i.e., that all possible subgraph matches are executed, has been retained. This is a very powerful device for expressing computation across sets without introducing explicit iteration. It's also hard to control, but therein lies the challenge.

This results in a basic architecture as follows:

1. The primary view of the graph for evolution purposes is node-centric. This means that we look for rules and matches in the neighborhood of a node, and when new edges are produced, we search neighborhoods of the nodes in that neighborhood, including any new nodes created. Thus we have a spreading-activation model.
2. One looks for rules to match against in two primary ways built into the kernel: (a) a global rule, i.e., one attached to a distinguished global-rule-pool node; and (b) a local rule, one attached to the node under consideration with a *rule* attribute.
3. Rules are parts of the graph (using a representation to be described), and are subject to the matching process as is any other subgraph.
4. Distinguished attribute and node names are defined for creating new nodes. In [18] the pure model was to start with some known countably infinite mapping, such as a chain representing the natural numbers, as "addresses", and the system simply manipulates edges. This could result in some nice formal properties, and avoid special constructs, but adds a layer of processing that's just too inefficient.
5. While [18] discussed the idea of a purely monotonic system, and defined time-like relations as a way to map to a human-perceivable world, we have not tackled that representation here yet. Deletion of edges is also introduced as a practical matter, although its use has been kept reasonably low.
6. A goal is to keep the knowledge contained in the primitive kernel as small as possible.

The H Language

In this document we'll typically refer to "the H language" to mean the rule language as described by the following BNF, and "the H-Machine" as the system upon which these rules run.

Since H is implemented within Common Lisp it naturally has fundamentally a Lisp format.

Most of the diagrams in this paper are directly generated from H-Machine runs, by dumping the edge set to GraphViz [9]. I am indebted to the GraphViz developers for providing this tool.

References in this paper to code refer to the files found at the link noted in reference [11].

Nodes, Edges, Rules, and The Graph, G

Simply, a node is a primitive object and an edge is an ordered multiset of nodes. In the implementation, a node can be any symbol, number, or string. Two strings are considered the same node when they are string-equal. Similarly, two numbers are considered to be the same node when they are =. So

a b “xyz” 1.1 11e-1 10 20

are all nodes (the first two numbers are the same node)

An edge is simply a list of nodes, and we use Lisp syntax to denote edges.

(a b c) (x next y) (n235 value 5)(n356 red)

are all edges.

When generating new nodes, we use the form n<number>, where <number> is an internally assigned sequence number.

We’ll refer informally to “the graph G” or just “G” to denote the currently evolving graph in the system. G is always connected, and there are no “duplicate” nodes or edges in G.

The syntax of a rule is simple and the BNF describes it in its entirety. A rule consists of two main parts, the *predicate (pred)*, and the *addition (add)*. The pred denotes pattern edges which must match a subgraph of G, and the add denotes new edges to be added, using the bindings of variables found by the preds.

For example, to define the transitivity of <, we write:

```
(rule
 (name trans-less-than)
 (pred
  (?x < ?y)
  (?y < ?z))
 (add
  (?x < ?z)))
```

When this rule is run it will match the subgraph

```
(3 < 4)(4 < 5)
```

and produce

```
(3 < 5)
```

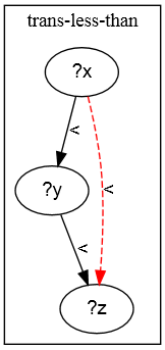
Rules can also have a *delete (del)* clause, in which case the edges resolved by the pred bindings are removed from G.

There is also a *not* clause, which specifies edges for which, if an isomorphism is not found, the overall predicate matches, otherwise it doesn’t. However we require that the pred part of the rule matches, to avoid runaway not processing. This works by resolving the edges in the not, and only creating the new add edges implied by the pred if the not edges do not exist. Not is currently used only in the print-gc rules (see h.lisp).

Rules have other adornment, as explained below.

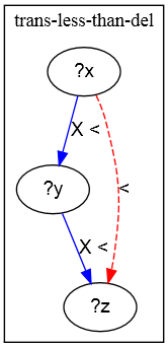
Rules as they lexically appear in H also contain two significant pieces of syntactic sugar, dealing with rules that generate or refer to other rules. These constructs, nested rules and rule-component lists, are abbreviations for more complex forms involving handling rule edges directly. These forms are sufficiently complex that the sugar is called for. This is explained in later sections.

Rules have a simple graphical interpretation which we’ll rely on throughout the paper. This is most easily explained by example, trans-less-than, from above:



The dark, solid nodes and edges represent a graph which must match a subgraph in G. Upon such a match, mapped nodes in G are bound to the indicated variable nodes. The red dotted edges are then produced, based on those bindings. This example shows the convention that 3-node edges are interpreted as object-attribute-value pairs. Other forms will become evident as we proceed.

The figure below specifies deletion. The blue-X marked edges are edges to be matched and deleted; the dotted red edge is to be added as before.



Syntax

```
<node>               ::= <var> | <const>
<var>                ::= <var-symbol>
<const>              ::= <const-symbol> | <number> | <string>
<var-symbol>         ::= ?<const-symbol>
<const-symbol>       ::= <symbol-not-beginning-with-?>
<edge>               ::= (<node> ...)
<rule>               ::= (rule
                          [(name <name>)]
                          [(root-var <var>)]
                          [(attach-to <node>)]
                          [(local)]
                          (pred
                           <pred-edge> ...)
                          (not
                           <not-edge> ...)
                          (add
                           <add-edge> ...)
                          (del
                           <del-edge> ...))
<pred-edge>          ::= (<node> ...) | (<var> new-node <new-node-indicator>)
<not-edge>           ::= (<node> ...)
<add-edge>           ::= (<node> ...) |
                         (print <node> ...) |
                         (<node> rule <rule>) |
                         (<node>|<edge> ...)
<del-edge>           ::= (<node> ...)
<new-node-indicator> ::= sn<number>
```

Note liberties are taken with the BNF above: Each disjunction does not necessarily represent a disjoint partitioning -- read it as more-general-to-more-specific, left-to-right. So e.g. an <add-edge> is generally a list of nodes but has a special "print" form and "rule" form.

```
[(name <name>)]
```

Optional name assigned to a rule. Generally essential when matching against rules for propagation purposes. Although in many cases a rule can be identified by matching on some of its components, it's normally better to give it a name so it's easily specified and found.

```
[(local)]
```

If this is present, then the rule is local, and is thus not in the global rule pool, and will not be matched in the course of executing a node. If absent, then, the rule is global.

See below for info on node execution and rule resolution.

```
[(attach-to <node>)]
```

If present, the rule is automatically made local and attached to the given node via the *rule* attribute. Typically this is used with the global-node for data inits and so forth.

```
[(root-var <node>)]
```

<node> here is a node within the rule. It's typically a var, but may be a const as well. Normally, without this present, the system will match a node against a rule by trying all orientations of the rule, i.e., matching each node in the rule in turn against the given object node. This method is general and helps assure a match even when you're not sure you're attaching the rule to the "right" node. Good performance depends on being able to detect the non-matches quickly, and having rules that don't over-match. While this generally works, if the root var is specified, then it is the only orientation matched against, which can help performance.

Rule representation and semantics

Rules are represented (“encoded”) within a graph G by expanding the edge lists in pred, add, etc., into nodes that explicitly define the edge content as relations in G. For example

```
(r pred
    (?x next ?y))
 =>
(r pred p1)
(p1 elem0 ?x)
(p1 elem1 next)
(p1 elem2 ?y)
```

Note that the elem<n> symbols become special indicators for rule processing. One can therefore match directly against rule content, and rules are parts of the graph. However, as with the rest of the H language, there are no reserved symbols and these may be used in rules and data as any other symbol can.

Generating rules

Rules can be generated by other rules using the "elem<n>" edge representation above directly. However this is rather tedious and hence some syntactic sugar is called for. The nested rule construct shown in the BNF above defines a new rule, and expands it into elem<n> edges. It does so when its rule matches; hence any bound variables will be substituted into the generated rule. This is a very useful mechanism, and it's possible for one rule to match data such that a whole family of rules is generated (see the section, *Rule Generation*).

An add clause may also contain an edge, i.e., a list. This is specifically for rule handling. This construct expands the given edge into an encoded graph fragment using *elem<n>* properties. This forms an easy way to specify new components for rules. See for example fft-delta.lisp.

Execution

A program in H is a sequence of rule definitions. Rules are used both to define literal data, and derive new data from old.

In [18] the fundamental computational mechanism is described as an exhaustive search through all subgraphs of a hypergraph, comparing each subgraph against another, and deriving new edges when a match occurs. For a finite graph this is at least computable, but hardly efficient.

To make this practical, some reasonable means of control must be introduced. This is accomplished as follows.

A node is *executed* by *running* all rules accessible by the node. By running a rule we mean matching the rule against a set of subgraphs found in a neighborhood around the node, and deriving any new edges from any set of matches. Note that more than one matching subgraph may be found in a neighborhood; each of them can produce new edges via the add clause of the rule.

Local and Global Rules

Rules are accessible by a given node iff the following conditions are met:

1. They are attached to the node using the *rule* relation, i.e., (<node> rule <rule-node>). A given node may have an arbitrary number of rule relations; all of them are run.

2. They are part of the global rule pool, and the node contains a link to the pool, using the relation *global-rule-pool*.

A basic principle of the H-Machine is that actions only occur due to connections found in the evolving graph. No rule is found, for example, in some hidden kernel "oracle" which magically knows of their existence. Every object node, then, has certain system-conventional links, in particular to sets of rules.

Global rules are attached to an object via a pool. The pool is global-rule-pool-node, and rules are found under the grp-rule attribute. So the global rule pool looks like this:

(global-rule-pool-node grp-rule rule1) (global-rule-pool-node grp-rule rule2) ...

Each object then has a link to the pool:

(<obj-node> global-rule-pool global-rule-pool-node)

When a rule is defined, it is placed in the global rule pool (attached to the global-rule-pool-node), unless the (local) flag is present in the rule definition.

Local rules are attached to an object using the "rule" attribute:

(<obj-node> rule rule1) (<obj-node> rule rule2) ...

Nodes may be populated with local rules by propagating them from one object to another. This leaves the question of their initialization. For this we define a local rule pool. It is distinguished from the global rule pool in that no default link is created from nodes to the local rule pool, and there is no execution semantics associated with the local rule pool. Instead, the local rule pool exists purely as a way to match against the set of rules, typically by name. We might see, for example,

```
(rule
 (name next-rule)
 (local)
 (pred
  (?x next ?y)
  (?x local-rule-pool ?p)
  (?p lrp-rule ?next-rule)
  (?next-rule name next-rule))
 (add
  ...other edges for ?x and ?y...
  (?y rule ?next-rule)))
```

In this case the rule propagates itself down a chain of *next* relations.

This technique of propagation is essential to controlling rules, both from the parallel and sequential viewpoints. While it may seem clumsy to add rule propagator matching to all rules of interest, rule mutation allows us to modularize this: A new rule is defined, which matches on the original rule, and adds pred/add edges to the original rule as illustrated above. So the original rule remains syntactically clean. We'll see in following sections application of this technique.

To summarize, then:

- Each rule is added to the global-rule-pool-node unless it has the (local) flag present.
- Rules are always added to the local-rule-pool-node and are available then for matching and propagation.
- One can say attach-to a node in a rule, in which case the rule is local, in the local rule pool, and attached as a rule to the given node. Any number of attach-to's may be supplied.
- The global-node is often used as an attachment point, since it is specially treated when using the primary execution function in the H-Machine interpreter. Specifically, the global-node is executed first; this allows rules which are for initialization, i.e., rules which create data or rules, to run and get things started.

Execution Loop

When a rule is run, and matches are found, new nodes and edges are produced. All nodes affected by the created edges (whether old or new) are placed on a queue for execution. In this way we propagate execution in a local neighborhood.[1]

Ideally, the queue should account for all execution needs. However, especially when deletion is used, this cannot always be guaranteed. The basic loop normally used, then, is one which tries to compensate for shortcomings in the methods as currently developed.

The loop is as follows:

[1] Since we're using hypergraphs, this is not as "local" as it might seem. For example, by default all "attribute" nodes are expanded, so some explosion is to be expected.

```
execute-global-all-objs-loop:
        exec-until-no-new-edges:
                exec-until-no-new-edges:
                        execute global-node
                execute queue
                execute all-objs
```

This executes from the queue, global node, or across all nodes, until no new edges are produced.

Rule matching and edge production

Rules have one purpose, and that is to produce edges. The predicate pred specifies the subgraph which needs to match. Variables are denoted with a leading question mark. Note that, to the extent possible, we have tried to stick to the rule that variables are only an optimization. That is, that main driving algorithm of the H-Machine is subgraph isomorphism, and although we have made "the usual" modifications regarding ordering and duplication, variables are not inherently necessary, except to make matching tractable. A consequence of this is that variables can be used in rules and meta-rules with well-defined meaning.

Rule semantics then are as follows:

- The pred subgraph of a rule r is matched against a set of edges E, a subgraph of G, normally supplied as some neighborhood found by expanding the surrounding edges of the executed node.
- All possible matches of r to a subset of E are found. The effect of this is to bind variables found in the pred to some nodes in E. Each such binding is then used, in turn, to add (or delete) edges as determined by the add and del clauses.
- Variables in the add clause are substituted with the bindings found by the pred match. These new edges are then added to G. Similarly, the del clause substitutions result in edges which are deleted from G.

An example:

(rule (pred (?x owns ?y)) (add (?x paid-for ?y)))

applied to

(john owns ford) (john owns stove)

produces

(john paid-for ford) (john paid-for stove)

Some handy built-in variables are bound during the execution of the add and del clauses:

?this-obj

Bound to the node on which the rule was executed.

?this-rule ?this-rule-name

Bound to the rule and rule name which matched and generated the edges.

?root-var

Bound to the root var found in the rule.

These are handy for diag printing, or for passing/deleting rules without a lot of extra fuss. For example one idiom used reasonably often is

(del (?this-obj rule ?this-rule))

which deletes the rule that just ran from the object -- when you know it's done, it can be disposed of and thus not run redundantly.

As mentioned above, rules have one purpose, and that is to produce edges. Edges produced can include new nodes. In some idealized models (as mentioned in [18]) one might require that new nodes are never produced, and that one simply uses a countable pool of pre-existing nodes. It seems possible to formalize such a model, but it does not seem practical. So in the H machine we adopt a notation to create nodes.

However in the spirit of this Platonic view that nodes all really exist already, we denote new nodes in the *predicate* (pred) part of the rule. The relation is *new-node*, and is specially detected by the kernel.

A quick example will set the stage:

```
(rule
 (name add-link)
 (pred
  (?x next ?y)
  (?n new-node sn1))
 (add
  (?y next ?n)))
```

In this case we're saying in essence that if you match against this new-node edge, you get a new node, bound to ?n. Plato springs to life! The sn<number> syntax is special, and causes (eventually) the creation of an actual new node (n<number>), which can then be used in the add clause.[2]

A rule can specify any number of new-node relations, the sn<number> designations are considered scoped within their defined rule [3]. The variable to which the new node is bound is arbitrary (?n in the above example).

Printing

The print edge has built-in semantics. It is useful for tracing and debugging. Simply, any edge of the form:

(print <node> ...)

will be printed when added.

The *print* edge is also the site of another experiment, the garbage collection of edges. Print edges don't need to stay in the system, and can be deleted as soon as the printing occurs. So we use a rule which picks this up. Since print edges are immediately printed, by the kernel, they are obsolete as soon as a rule can detect them. We generate a series of rules which detects (print …) and deletes that edge. However a problem is encountered here. The print node will have a *rule* relation, or a link to a rule pool, so that rules can be found for it. Recall that a principle of the H-Machine is that G is always connected. However a print-gc rule will find this and delete it, thus cutting off print from further gc.

I thus reluctantly developed not. Thus we have, for 3-node print edges,

…(pred (print ?x1 ?x2) (not (print rule ?r))…

Print-gc is the only place I have used not so far.

Diagrams

Graphs have a natural graphical interpretation; hypergraphs are a bit harder to display. For the H-machine we have a set of drawing interpretations and heuristics. These are imperfect but illustrate the system. Note most of the diagrams in this document were generated by Graphviz [9].

Simple heuristics are used for edges of 2, 3, and 4 nodes, coupled with meta-information from G:

- Three-node edges are by default drawn as node-attribute-value, where the attribute is a labeled arc.
- Two-node edges are drawn using the first node as an object, and the second as a property name, which is a borderless text box attached to the object by a single unlabeled edge.
- 4-node edges are treated specially if the third node has the *two-input-op* property, in which case such an edge is drawn as a two-input dataflow node, in the form (input1 input2 op output).
- Edges of 4 nodes or greater with no special markers are simply listed as a chain of nodes connected by unlabeled directed edges.

---

[2] It's eventually because in fact, a node of the form nn<number> is created by the rule definer, and it in turn causes a new node to be created at run time.

[3] The naming convention derives from "scoped".

In rules, edges in the pred clause are solid black, and edges in the add clause are dotted red. Edges in rules which are in the pred and in the del clause are marked in blue, with an "X".

Color is used primarily to make the diagrams nicer, but also to help illustrate the information-passing properties of the propagation rules.

A fixed set of colors is added to the system via the color-circle-data rule. This is literal data and forms a color circle (like a "color wheel", although it does not follow the traditional color order). Each node is named for a color and is related to the next color by the *next-color* relation.

The names of the color nodes are chosen to be the same as the color names supported by Graphviz (gv) (see [9]). Therefore, in a gv dump, using the node name gets the correct color with no extra effort.

By default, a given node will be colored according to its color relation. So due to

(x color red)

the node x will appear in the gv image colored red.

As an example of the flexibility as well as cuteness of the H language, to get each node of the color circle itself to be displayed with the correct color, the node needed a *color* attribute to a node with its color name, i.e., itself. So this rule does the trick:

```
(rule
 (name color-color)
 (attach-to global-node)
 (root-var global-node)
 (pred
   (global-node next-color)
   (?x next-color ?y))
 (add
   (?x color ?x))
 (del
   (?this-obj rule ?this-rule)))
```

This says, first, that it's a global-node rule, which means it will be executed up-front. Also, when done, we know we will have filled in all the color nodes, so it deletes itself from the global node. For the rest, it knows ?x is a color node because it has a *next-color* relative. It then simply adds the *color* link to itself.

Note that by simply stating the *next-color* relation, we take care of all the color-circle nodes. There is no need for an explicit "iteration" indicator. This aspect of the H-machine is very powerful, but also delicate, since combinatorial explosion must be contained.

Extended Example: The FFT Butterfly

In the mid-eighties, while working on Common Lisp compilers and related tools [1], I developed a simple test program based on the Fast Fourier Transform. As we were developing the language system, this test covered a fair bit of ground, including testing numeric processing, declarations, arrays, recursion handling, and other fundamentals. The program directly defined the recursive FFT, essentially as follows:

$$F_N(x) = G\,(F_{N/2}\,(\text{odd}(x)),\ F_{N/2}\,(\text{even}(x))), \qquad (1)$$

where x is an array of length N, and G is a linear combination based on the complex exponential W_N [15]. When provided with a simple square pulse input, the program printed an ascii-art version of the real part of the result, a signature sin(x)/x shape, which showed to some level of confidence that the program was working as desired. See Figure 1 below. The program itself may be found in fft.lisp. [11]

The algorithm of course takes a classic divide-and-conquer approach to a problem that is “naturally” quadratic and converts it to n*log*n or closely similar complexity. Writing it using direct top-down recursion retains the n*log*n behavior, but adds a considerable extra factor of overhead, so FFTs in the real world are

normally coded bottom–up as tight loops, and will employ such tools as parallel threads, hardware dataflow, and a host of other techniques.[4]

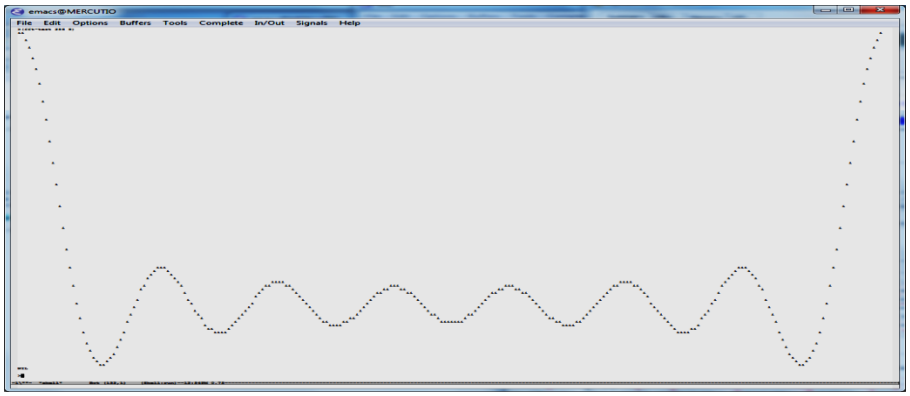


Figure 1 – Output of FFT test program, the real part of a 128-point DFT of a unity square pulse of width 8 at the origin.

The H-Machine FFT

The H-Machine version of the FFT connects the top-down and bottom-up worlds, effectively by deriving the bottom-up butterfly model directly from the recursive equations. Note that in this experiment only the butterfly data flow topology is formed; actual numeric calculation is not made. In the description to follow, please see the rules in h.lisp [11].

We assume that G is initially populated with a set of rules, and with a *sigma* chain, representing the natural numbers, out to some reasonable but small number. We start by defining a node to be the top of a tree of a certain number *N* of levels. Rules match this and add two nodes downward for each existing node, stopping when the level reaches zero. Other rules produce the *tree-next* relation, which connects nodes across the tree. The result of this propagation is that all the bottom-level nodes will be connected together by *next*. The tree is asymmetrical, in that the initial node defines that the two next-to-top nodes will have the *zero* and *max* property (the choice of which node has which property is arbitrary). Rules propagate these properties down the tree, and when the bottom is reached, a loop rule detects the *zero* and the *max* at the bottom level, and then connects them together, forming the *next* loop, with a distinguished *zero* node. The *next* chain is of length 2^N. Forming the chain into a cycle is needed to emulate the *mod* operation required to construct the butterflies.

With an array thus defined, we need to take the odd and even parts. First, rules running on each array mark nodes as even (*ev*) or odd (*od*), with the basis that zero is even. Then other rules construct a new array of only the points marked odd or even; they are then endowed with loops and a distinguished zero node, as above.

Rules for fft processing detect these arrays, and connect them together via the combining function *fft-comb*. The set of these now forms the basic bottom-up flow graph. More rules detect these combining functions, and form the butterflies with the array elements, as a set of half-butterflies (*hb*).

The preceding is defined at the top level by the fft-rule, which detects the presence of edges of the form (*x* fft *y*)(*x* level *l*), where $l = log_2(N)$, *N* the number of points in the fft. Fft-rule is shown as Figure 2 below. It is evident that the structure of fft-rule follows closely that of the recursive equation (1).

[4] See [2] and [4] for surveys of current FFT architectures, and [12] for a list of papers in this area.

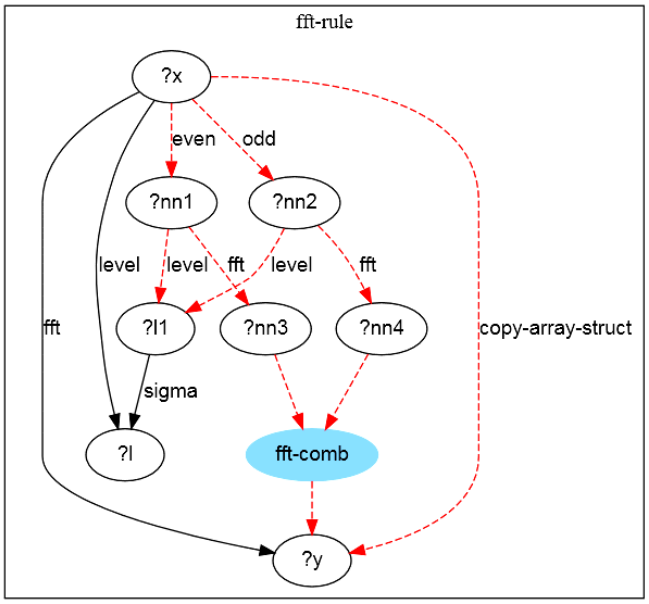
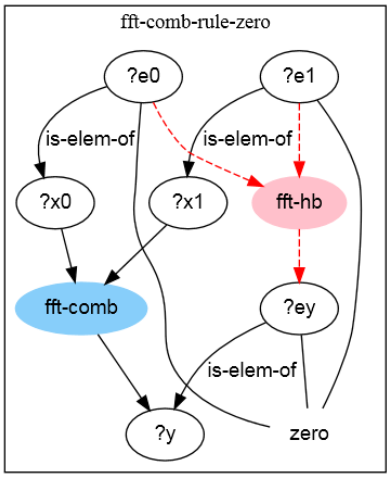


Figure 2 -- FFT Rule

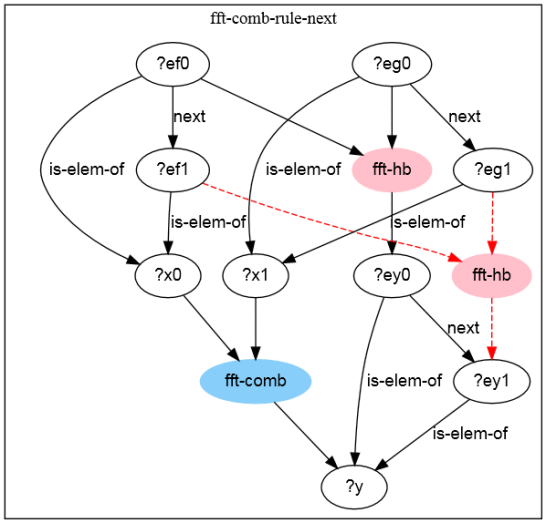
In Figure 2, nodes whose names are prefixed by "?" are variables, and those variables of the form ?nn<n> represent new nodes. The relation of these to the *new-node* relation is not shown, for clarity. The black solid edges are predicate edges. These must match a subgraph in G, in which case the variables are bound to the associated nodes (new nodes are included in the predicate binding). Using these bindings, the red dotted edges are then instantiated upon match.

As shown, the fft relation between two nodes (the edge (?x fft ?y) in this case) spawns more edges, some of which have fft edges. This then causes the rule to apply again, and thus the recursion is produced. Each fft is connected to a node with an odd or even relation. As described above, those relations are also detected, and spawn more rules to produce the odd and even arrays. The crucial edge copy-array-struct is added, which causes the array structure of the input node to be copied to the output node. This is needed to form the full set of nodes required for the butterflies. Finally, the ffts of the reduced odd-even relations are combined using the *fft-comb* node. Note in the H-Machine this is simply a four-node edge of the form (input1 input2 fft-comb output). Fft-comb is tagged with the property *two-input-op* and this is recognized by the graphics dumper and turned into a standard dataflow function graphic.

Many other rules must run to produce the full fft butterly graph. Key is the fft-comb-rule set of rules, which detect the *fft-comb* and produce the butterflies from the array elements it references. The basis rule for this is fft-comb-rule-zero (below), which detects the *zero* elements connected by an *fft-comb* and connects those elements via a half-butterfly (*hb*).



Fft-comb-rule-next, below, then detects that a half butterfly has been contstructed and builds one attached to the next element, and so forth down the line. These rules then fill in all the butterflies at all levels.



The results of this are shown in the following series of figures, for an 8-point fft between the nodes x and xfft. Figure 3 shows the recursive structure of the top-down build and bottom-up construction of the fft-comb operation. Next, Figure 4 shows the recursive top-down/bottom-up structure using relations that display the symmetry and flow well. The relation *d* was defined solely for the display as a convenient "connection" between the top-down and bottom-up parts of the graph growth process. There is no actual "data flow" to be perceived across the *d* relation. Figure 5 shows the construction of the butterflies. Finally, Figure 6 shows the previous two components connected, via the relations *d* and *d-casz*, added for display purposes.

The rules used to generate the fft structures are shown in Figure 7. See also the rule text, and the gallery for higher-resolution forms and the complete set of rules. [11]

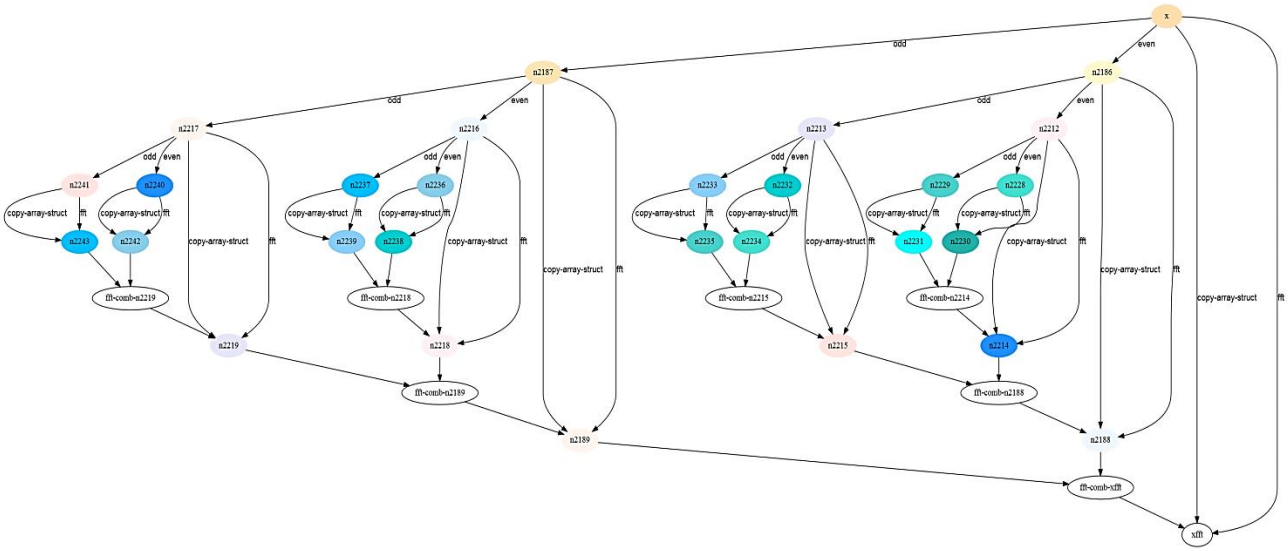


Figure 3 – FFT, recursive structure

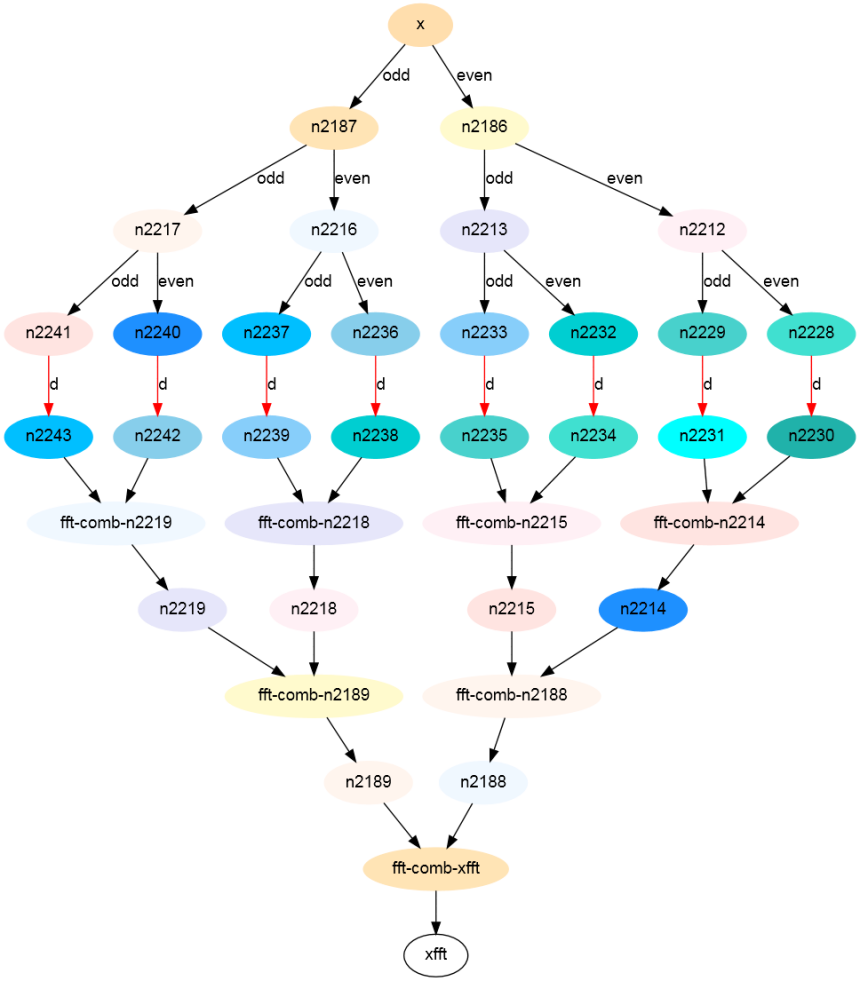


Figure 4 – FFT, showing symmetry of top-down and bottom-up components

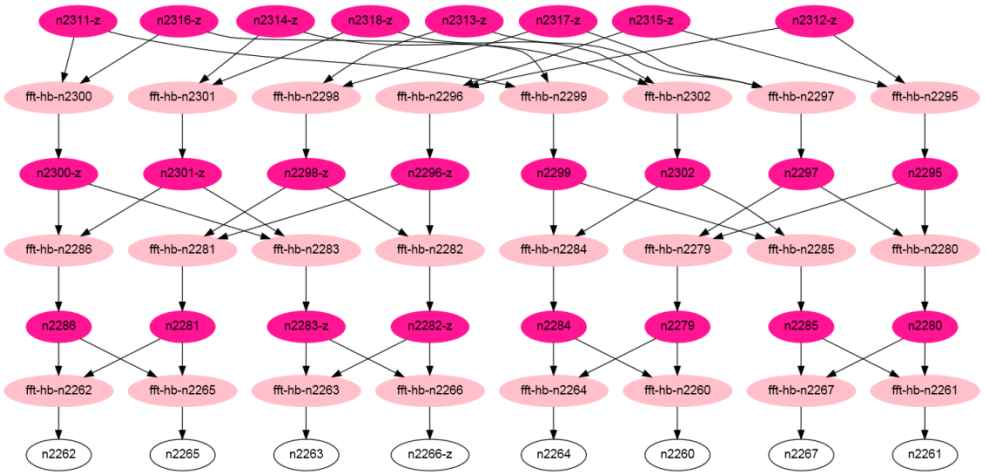


Figure 5 -- FFT, the generated butterflies

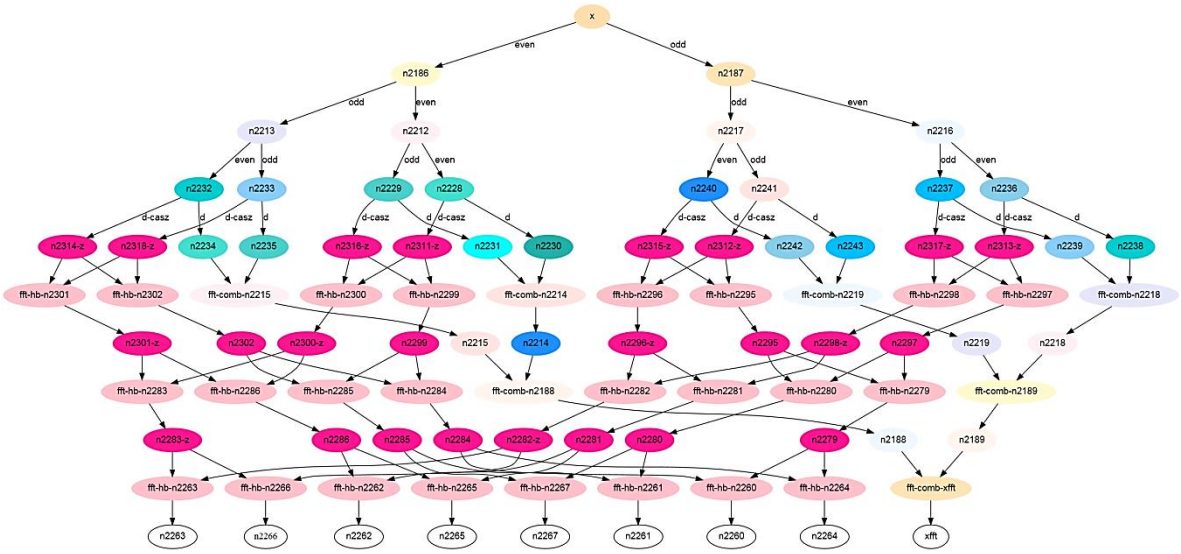


Figure 6 -- FFT, 8 points, complete top-down/bottom-up construction

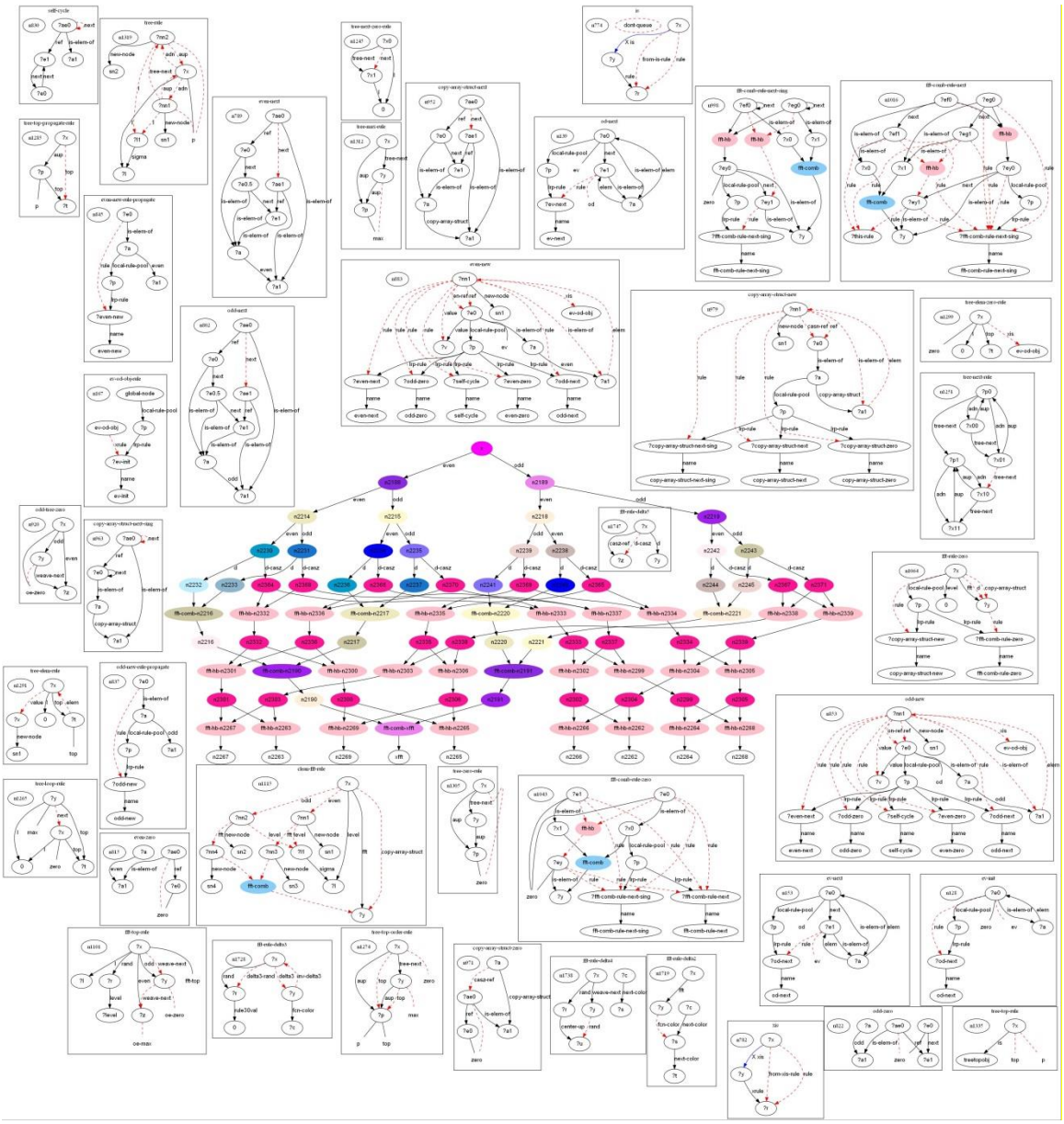


Figure 7 -- 8-point FFT with the essential rules which generated it. In [11], see gallery/fft-8-just-fft-rules.svg for a higher-resolution version.

Graph Mutation

We will continue to use the FFT example as part of describing mutation rules and structures. *Mutation* is defined as changes made to some graph structure originating from rules which are outside the rules used to create the structure. For example, generating a tree using the tree rules, we might define a rule which operates on a node so many levels down from the top and so many across, and adding onto or modifying the surrounding structure in some way. However instead of such literal-address types of rules, we seek more general ways to specify how a graph should mutate as it grows.

One way to do this, described in this section, is to develop multiple kinds of structures, and let them interact in some way. We'll use three of them: one of them is a tree, as described above.

The second is based on the Rule-30 one-dimensional cellular automaton from [21]. This is built as a set of unfolding layers, each of which adds a new row of nodes in accordance with the Rule-30 rules.

The primary reason for choosing Rule-30 is that it provides a pseudo-randomness capability, i.e., down the center cell. [5] Secondarily, it provides good exposition for rule-generating rules. This is covered in a later section, *Rule Generation.*

To build a rule-30 graph, a set of layers is built based on an initial natural number parameter. A rule-30 node has relations *up*, *next*, and *rule-30-val*, the latter of which is 1 or 0. Note these values show in the graphics as nodes colored blue and pink, respectively. *Up* refers to the node at the previous layer, and *next* refers to the adjacent node in the layer. A rule for each pattern of interest checks the *up* and *next* nodes for the matching pattern of *rule-30-vals*, and creates new nodes with appropriate values for the next layer. Meanwhile another rule tracks the *up* relation along the center node, and propagates a *center-up* relation down the center. When the bottom is reached, a loop rule joins the bottom and top via *center-up*, and this is used subsequently as a circular pseudo-random-value generator.

Figure 9 shows a rule-30 run carried out to 60 levels.[6] Note the distinctive triangular pattern seen in the figure, as in [21]. For comparison, Figure 9 shows rule 110 from [21].

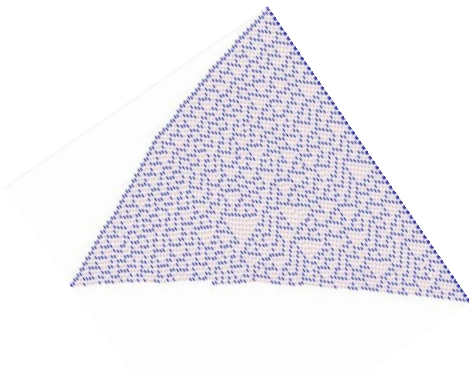


Figure 9 -- Rule-30 run to 60 levels

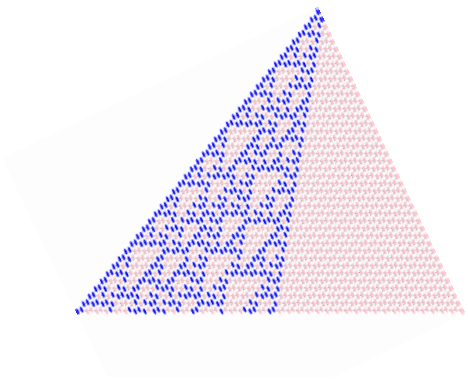


Figure 9 – A run of rule-110

The third interacting structure is the *color circle*, a ring of colors related by the *next-color* relation. This is built from literal data and is shown in Figure 10. The names of the colors are selected from those supported by Graphviz [9].

An additional useful structure to build onto a tree is a *linear weave*, a chain through all the nodes in the tree. This is added to the odd-even tree via the weave-next and related rules. This supplies a linear ordering to the tree nodes. The rules for this can be found in h.lisp. Figure 10 shows the *weave-next* relation, highlighted in red.

[5] See [21] and [8] for analyses of the randomness properties of Rule-30.

[6] Note the last row is incomplete.

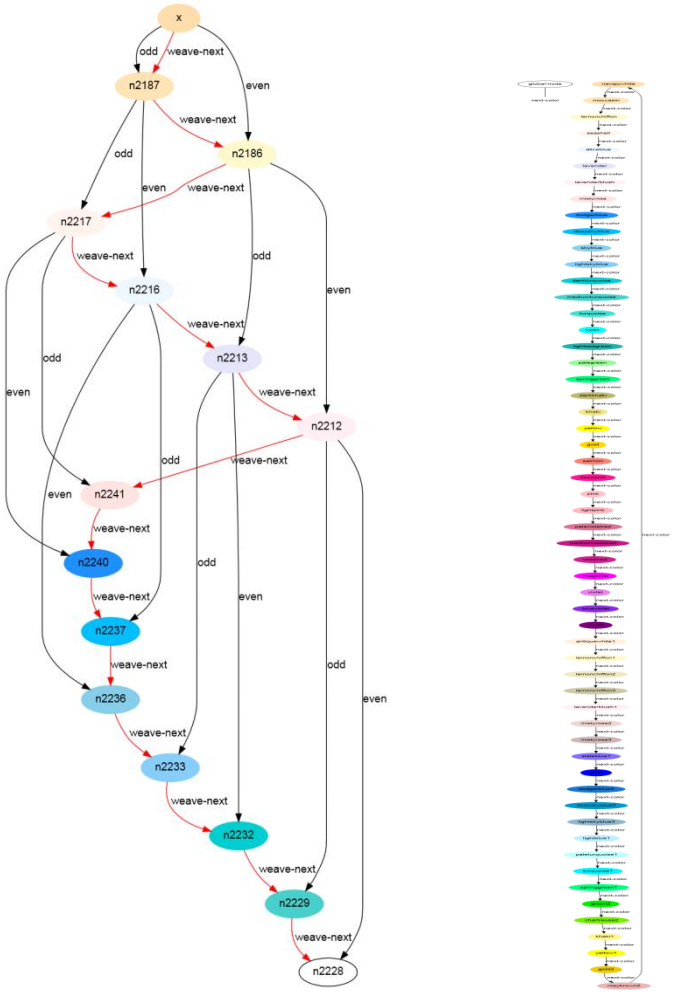
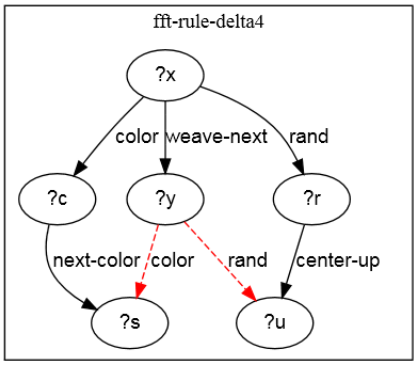


Figure 10 – The *weave-next* relation and the color circle

The rule fft-rule-delta4, shown below, propagates *rand* and *color* along the weave chain, looping back for color or rule-30 node should it reach the end of its initial chain (the *next-color* relation for colors and the *center-up* relation for rule-30).



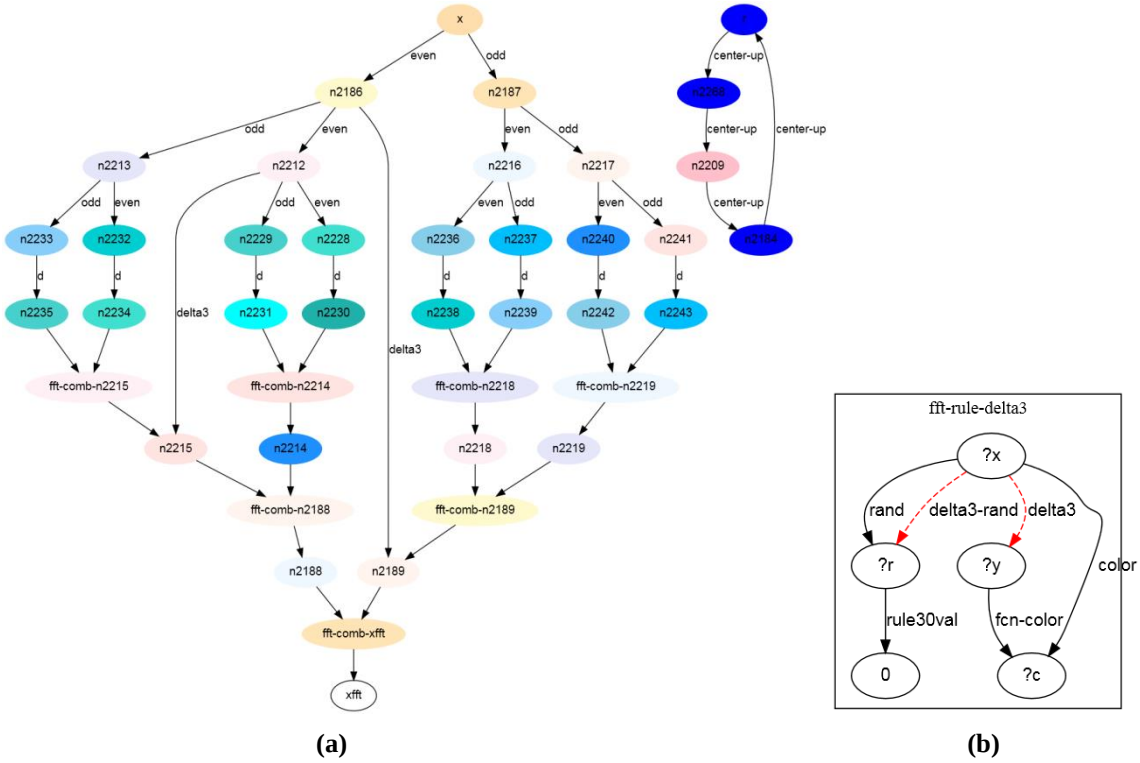
Thus, fft-rule-delta4 assigns a color to each node, which produces the colorization noted in the figures. Fft-rule-delta4 also supplies each node with a pseudo-random bit, given by the relation chain *rand.rule30val*.

Given then the assignment of random bits and colors to the nodes, the rule fft-rule-delta3 (Figure 11(b)). says that if for two nodes *x* and *y*, x.*color* = *y.fcn-color*, and *x.rand.rule30val = 0*, then we'll add a new

edge, *delta3*, between *x* and *y*. The relation *fcn-color* is attached to an fft-comb output node and indicates the color of the associated *two-input-op* which impinges on *y*.

We are thus able to combine conjunctively the selection properties of these structures, rule-30 and color-circle, using a simple compact rule.

The result of this rule is shown Figure 11(a). This is an 8-point fft, but the butterflies are not shown since they are not affected by the mutation. Two edges are chosen for addition based on the criteria. Note that the color of node n2186 is the same as that of fft-comb-n2189, and that of n2212 is the same as fft-comb-n2215. The upper-right part of the figure shows the *center-u*p random sequence, in this case for a rule-30 iteration size of only four.



(a) **(b)**

Figure 11 – Delta3 graph mutations (a), and corresponding rule fft-rule-delta3 (b).

Note that fft-rule-delta3 also installs an edge, *delta3-rand*, from the source node to the rule-30 node which matched. This provides a view into the random part of the mapping which assigned the delta3 edges. Displaying this mapping gives us an interesting looking view, shown in Figure 12, of rule 30 "eating" the fft.

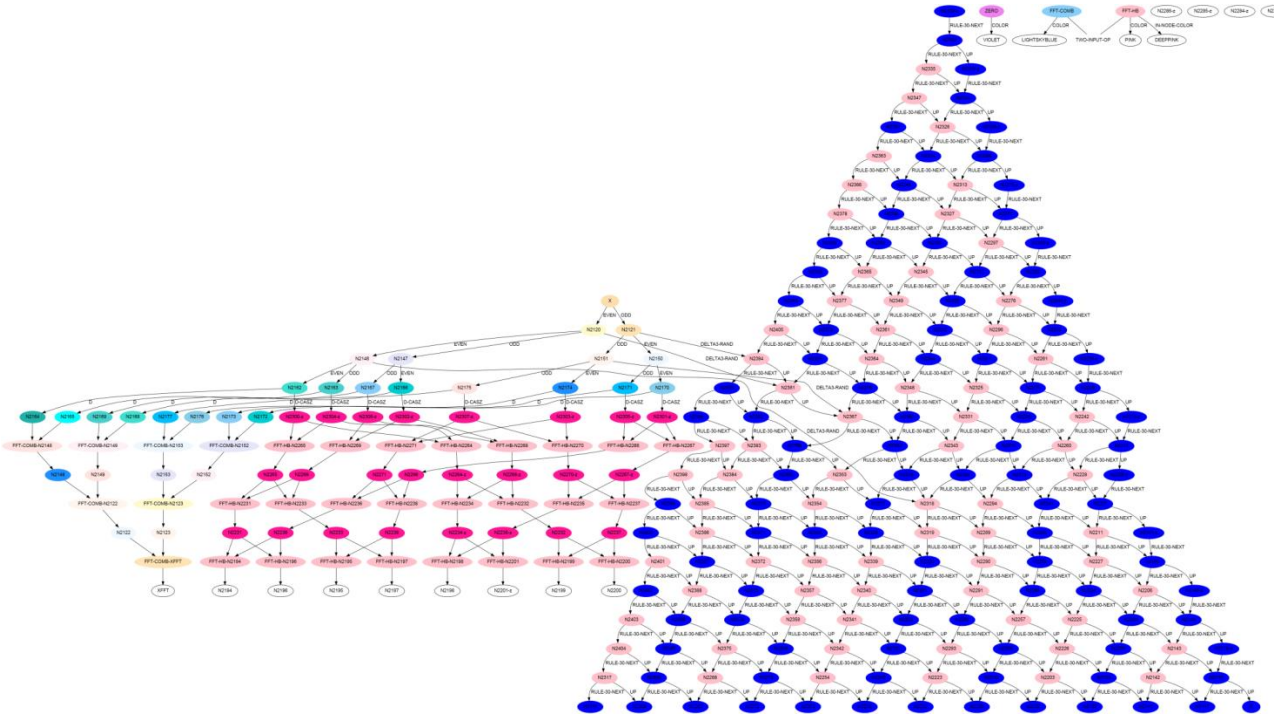


Figure 12 -- Rule 30 eats the Fourier Transform

Comments on Modularity

Using the H-machine for a while as a programming language, there are some interesting notions of modularity which have emerged.

First is the observation that the use of isomorphism/matching allows computation in what are effectively scoped contexts, without explicitly defining a scoping model. For instance I can use the attribute names *elem* or *next* in many objects and they do not conflict, as long as I narrow down the use as appropriate, i.e., by using other attributes to isolate the unique set of subgraphs intended to match. Sometimes one can assign a unique "type name" or similar explicit tag to refine these attributes, i.e., to create the explicit notion of an object that exists in some sense independent of its properties, but it's not required (although good "engineering practice" may call for a universal set of such tags in a given project).

A side benefit is that one can match on these attributes as well, since they are nodes in the hypergraph G, and find if desired, for example, all edges which contain a given attribute. However, one area where this has become an issue is in the graph display dump, which is partly driven by attribute name. One may expose a *next*, for instance, and get more nodes and edges dumped than desired. The solution I have so far for this is simply to add more edges which denote attributes for display purposes only.

The second notion of modularity that has emerged is similar to that found in Aspect Oriented Programming [13]. In the H-machine, rules are distinct entities and do not need to be included in some lexical scope.[7] In addition, rules can match on and modify other rules. This supplies the capability to alter system behavior across a wide range using rules and meta-rules expressed outside the modules affected, in the same way that expressing an Aspect can affect code semantics over a wide set of objects and/or methods.

[7] Nested rules offer lexical scoping; however this is really just syntactic sugar for more complex manipulation of edges which one must do instead.

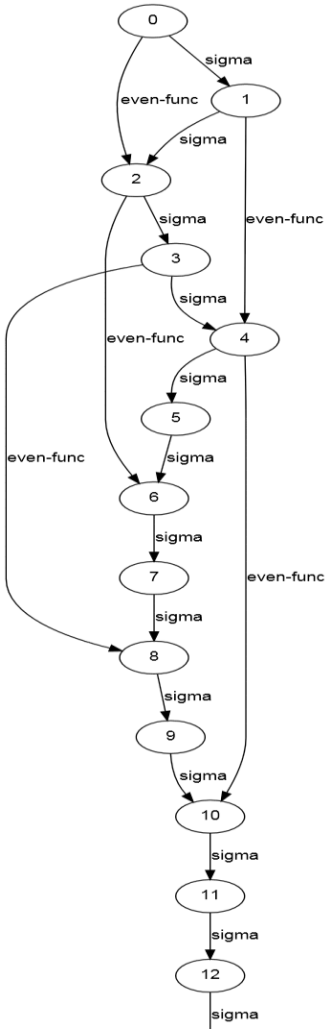


Figure 13

A good example of this is in the rule file fft-delta.lisp. Here rules match on the fact that an fft relation exists, and/or other relations defined external to the rules in the file. From there colors are defined; random values are assigned by matching against the rule-30 nodes; based on the randomness mutations to the data flow graph are produced; and a few edges added solely for display purposes. And another rule, fft-rule-opt, matches on the fft rule itself, changes it from global to local, and adds rule-propagation clauses. This rule thus optimizes the fft-rule, which is written in a "clean", but global and inefficient, style.

These rules in fft-delta.lisp act on the fft-rule itself and its results. The rules producing those results know nothing of the attributes, such as *color* and *rand*, which the external fft-delta rules will add.

Meta-Rules

Rule Propagation

Execution of rules in an efficient manner is a major challenge in a system like the H-Machine. While clearly one of the primary goals is to develop a parallel execution model, reasonable sequential performance is important, at least for development purposes.

The most basic way to manipulate rules is to pass around references to them. Execution of nodes is most efficient when only the necessary and sufficient rules are directly attached to a node. The only issue is that you need to know that a given rule is required on a given node, and you need to get it there.

Global rules of course solve this problem by always being accessible. Generally, things are easier to get working with global rules since they "wash" over all nodes looking for a match. Of course they are very inefficient, since they get tested only to fail many times over.[8]

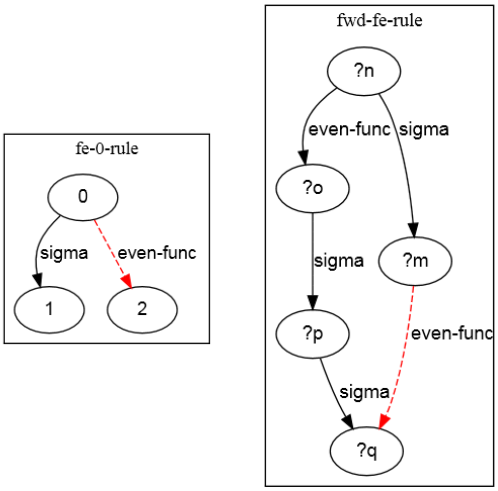
By defining local rules we allow a number of potential models where rules can be passed, grouped, copied, and so forth, as befits the need of the underlying execution environment, be it sequential or parallel.

An example upon which we can build is the "even function" (the *even-func* relation), which is a map from the natural numbers to the even natural numbers. In this case it's 2n+2, or $0 \rightarrow 2$, $1 \rightarrow 4$, $2 \rightarrow 6$, … [9] The desired graph is shown in Figure 13, through domain element 4. *Sigma* represents the successor function, and thus we have a small ordered subset of the natural numbers as given.

Conferring the *even-func* relation on the sigma chain is straightforward given two rules, the fe-0-rule and fwd-fe-rule:

[8] Some basic heuristics are employed in the kernel to keep this from completely exploding; nevertheless the penalty can be high.

[9] At the time, I didn't want a node to map to itself, to test some internal aspects of the match. Hence the choice of 2n+2 rather than 2n.



Once the fe-0-rule runs and maps 0, if fwd-fe-rule is global, then *even-func* will be added to all nodes until the natural number limit of G is reached.

Should it be desired only to produce part of the function, say from some n down to zero,[10] a different style needs to be imposed, i.e., reductive recursion. The fundamental model in this case is to start from n, go down to zero leaving rules behind, and then come back up, running the rules, in particular fwd-fe-rule, until there are no further such rules (n is reached).

First, we define fe-rule-gen, which starts the process by supplying the node n (to which it's attached) with fe-0-rule, fwd-fe-rule, and back-fe-rule, under the relation *rule*, and looked up via the local rule pool. Thereafter, back-fe-rule walks back along *sigma* from n, adding itself, fe-0-rule, and fwd-fe-rule to each successive node. When fe-0-rule fires the remaining nodes up to n will have fwd-fe-rule defined, and will fire and install *even-func*. Figure 14 below shows the complete set of rules for this case.[11] A flow diagram is shown in Figure 15.

[10] Or that we might just want to evaluate 2n+2.

[11] Note in fwd-fe-rule an extra *sigma* and a *queue* property is defined. This is a kernel command to queue the node for execution and assure the rule is tested. The tradeoff in this exposition is that fe-rule-gen and back-fe-rule bear all responsibility for rule transport, while fwd-fe-rule and fe-0-rule are left clean but for that one kernel mote.

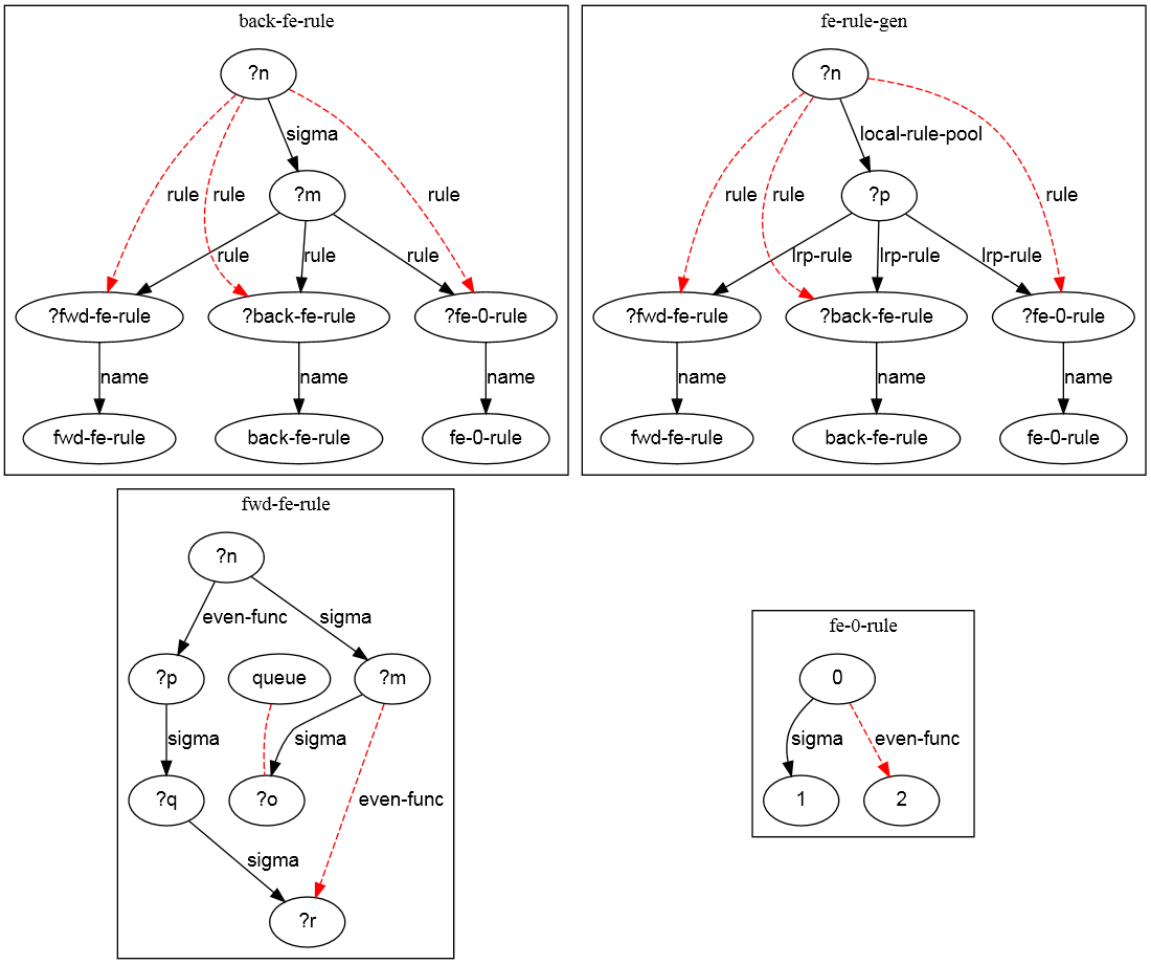


Figure 14 -- Rule-propagating rules for the bounded even function.

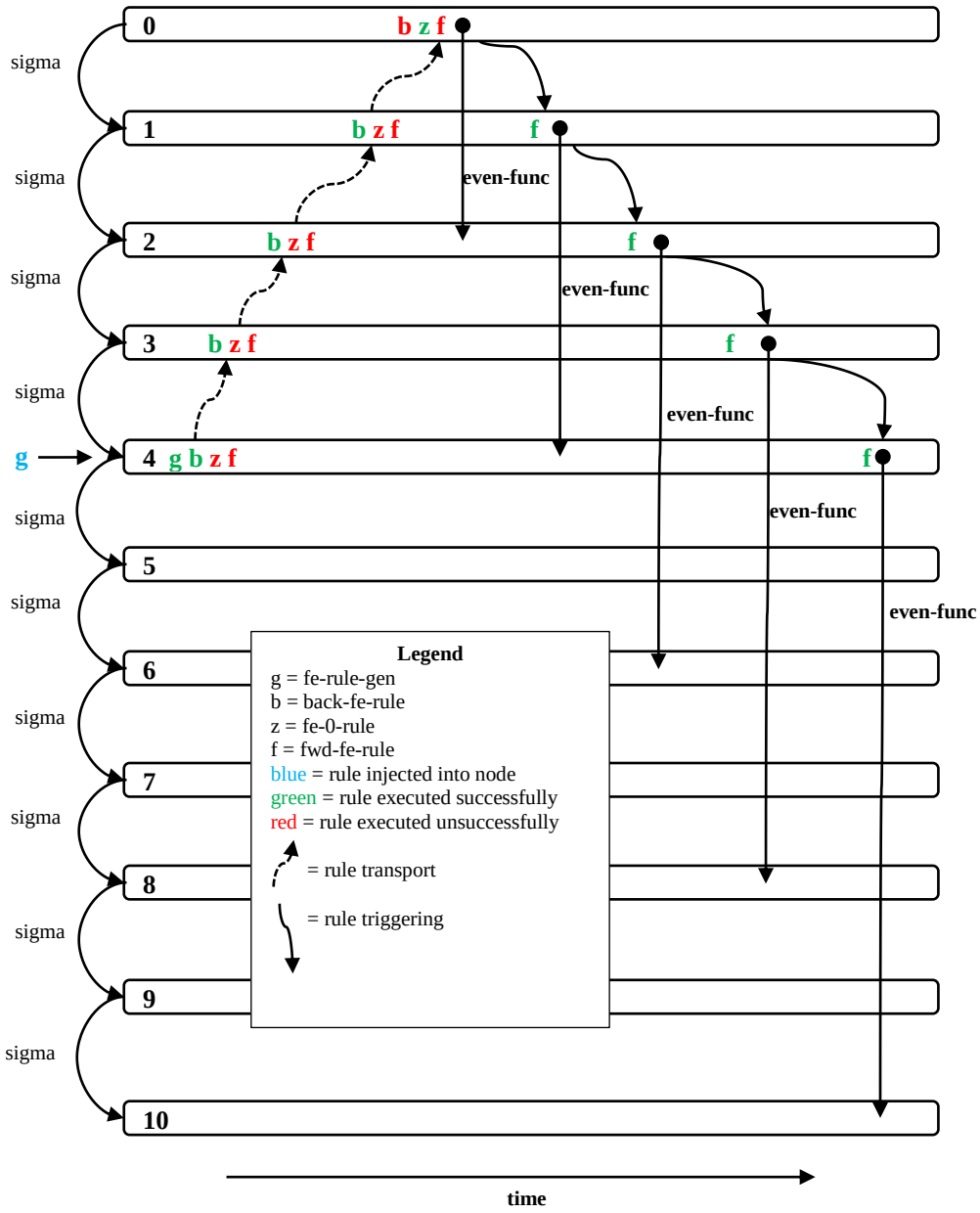


Figure 15 -- Execution of the even-func rules

Many embellishments are possible on this sort of model. For instance, one can delete rules as they are used and it's certain they are not needed;[12] for the *even-func* given above it's possible to arrange deletions so that we end up with net no rules on the nodes involved, just the desired *even-func* relation requested. Another embellishment is to store the needed rules in another object, and refer to them using an "is" rule of some kind. A very basic "is" rule is to state that if some node x *is* another node y, then x inherits y's rules. This simplistic rule is useful for factoring into groups sets of rules which need to be passed around. Examples of this may be found in the code.

[12] See the fft rules in the code.

Another embellishment is rule copying, the topic of the next section.

Rule Copying

Thus far we have described manipulating only references to rules. This means that a single rule node is used and passed around. This is fine in a purely sequential system with a large, constant-access-time memory. In a parallel system, we expect the edges to model spatial distance as well. In this case, passing pointers from node to node will generally imply an increased "distance" the rule must travel to interact with nodes that will produce new edges.

Rule copying is a way to address this problem. Below is the text of the rule-copying rule (note this does not use any of the rule-handling syntactic sugar):

```
(rule
 (name copy-rule-rule)
 (pred
  (?r copy-rule ?y)
  (?r name ?name)
  (?r root-var ?x-root-var)
  (?nn1 new-node sn1))
 (add
  (?nn1 root-var ?x-root-var)
  (?nn1 name ?name)
  (?nn1 type rule)
  (?r rule (rule
    (name copy-rule-rule-pred)
    (pred
     (?r pred ?p)
     (?nn2 new-node sn2))
    (add
     (?r rule (rule
       (name copy-rule-rule-pred-2)
       (pred
        (?r name ?name)
        (?r pred ?p)
        (?p elem0 ?pe0)
        (?p elem1 ?pe1)
        (?p elem2 ?pe2))
       (add
        (?y rule ?nn1)
        (?nn1 pred ?nn2)
        (?nn2 elem0 ?pe0)
        (?nn2 elem1 ?pe1)
        (?nn2 elem2 ?pe2)))))))
  (?r rule (rule
    (name copy-rule-rule-add)
    (pred
     (?r add ?a)
     (?nn3 new-node sn3))
    (add
     (?r rule (rule
       (name copy-rule-rule-add-2)
       (pred
        (?r name ?name)
        (?r add ?a)
        (?a elem0 ?ae0)
        (?a elem1 ?ae1)
        (?a elem2 ?ae2))
       (add
        (?y rule ?nn1)
        (?nn1 add ?nn3)
        (?nn3 elem0 ?ae0)
        (?nn3 elem1 ?ae1)
        (?nn3 elem2 ?ae2)))))))))
```

Copy-rule-rule first requires that the relation *copy-rule* exist between the rule to be copied, and an object to which the rule will be copied under a *rule* attribute. Its operation is straightforward, detecting the encoded patterns of the edges which make up the *pred* and *add* components.[13]

[13] Note that del and not are not supported, and only rule edge sizes of three or less. This limitation is ok given the examples in which it is used.

The rule copier was tested on the *even-func* example, propagating copies of fwd-fe-rule down the *sigma* chain of nodes. See fe.lisp.

Copy-rule-rule can also copy itself, although it's very inefficient in the current implementation. We might use this to build a graph automaton which starts minimally and grows, completely by local matching and edge production, carrying its rules with it. One would always need to propagate a copy of copy-rule-rule, but then other rules could piggy-back on that and themselves propagate.

Rule Generation

Generating rules is a useful abstraction technique. We'll use as the example for this section the rules which generate the rules for the Rule-30 cellular automaton. Note that while written for Rule 30, the generator can capture any of the one-dimensional CAs described in [21].

Rule30.lisp contains the rules for this. Of interest here are rule-30-rule-gen, rule-30-zero-rule-gen, rule-30-max-rule-gen, and rule-30-data.

The essence of a rule-30 rule is to observe three adjacent *next*-related cells to determine the value of a new cell, installed below the center cell and related to the original by the *up* relation:

```
(rule
 (name rule-30-rule)
 (pred
  (?y level ?l)
  (?l1 sigma ?l)
  (?x next ?y)
  (?y next ?z)
  (?x rule30val ?xval)
  (?y rule30val ?yval)
  (?z rule30val ?zval)
  (?nn1 new-node sn1))
 (add
  (?nn1 up ?y)
  (?nn1 level ?l1)
  (?nn1 rule30val ?nval)))
```

The variables ?xval, ?yval, ?zval, and ?nval represent the left, center, and right cells of a triple, with the new value, under the center, denoted by ?nval. These variables need to match against some values that denote the desired cell pattern. This is done by the rule generator:

```
(rule
 (name rule-30-rule-gen)
 (attach-to global-node)
 (root-var global-node)
 (pred
  (global-node rule-30-data)
  (rule-30-data ?xval ?yval ?zval ?nval))
 (add
  (global-rule-pool-node grp-rule
    (rule
     (root-var ?y)
     (pred
      (?y level ?l)
      (?l1 sigma ?l)
      (?x next ?y)
      (?y next ?z)
      (?y interior)
      (?x rule30val ?xval)
      (?y rule30val ?yval)
      (?z rule30val ?zval)
      (?nn1 new-node sn1))
     (add
      (?nn1 up ?y)
      (?nn1 interior)
      (?nn1 level ?l1)
      (?nn1 rule30val ?nval)))))
 (del
  (global-node rule ?this-rule)))
```

Note the lexical scoping of the rules and the rule-30-data. This is simply a shorthand for generating the edges for the nested rule required directly. In execution, any edge matching the *rule-30-data* pred will

generate a new rule. Thus the scoping looks lexical for written convenience, but the mechanism is driven at run time. There are also generators for the boundary-case rules.

The last piece of this is the data:

```
(rule
 (name rule-30-data)
 (attach-to global-node)
 (root-var global-node)
 (pred
  (global-node rule ?rule-30-data)
  (?rule-30-data name rule-30-data))
 (add
  (print rule-30-data)
  (global-node rule-30-data)
  (rule-30-data 0 0 0 0)
  (rule-30-data 0 0 1 1)
  (rule-30-data 0 1 0 1)
  (rule-30-data 0 1 1 1)
  (rule-30-data 1 0 0 1)
  (rule-30-data 1 0 1 0)
  (rule-30-data 1 1 0 0)
  (rule-30-data 1 1 1 0))
 (del
  (global-node rule ?this-rule)))
```

Each entry in the binary-matrix pattern above matches a cell pattern of rule 30. One simply changes that to get another CA rule. The rule runs on the global node and deletes itself once run.

Note that this method very compactly expresses the CA generation rules and how to carry them out. In particular, there is no explicit expression of iteration.

Conclusion

Graph automata and rule-based systems have been well-studied, from various perspectives.[14] The primary contribution of this paper is to demonstrate the use of meta-rules in a hypergraph context, using practical examples as beginning guides. A secondary contribution rests in the simplicity of the rule system, and the versatility obtained by thinking in hypergraph terms while retaining that simplicity. The graphics are also a relatively straightforward result of this simplicity, and provide an interesting pedagogical and developmental tool.

This paper has not emphasized performance, and the topic deserves separate treatment. The H-Machine code right now is very compact; however, it is not very efficient. Improving the efficiency can take several forms, e.g., subgraph search heuristics, indexing, rule compilation, bootstrapping, parallel processing, and a host of other possible techniques.

Subgraph matching is largely about finding the subgraph, i.e., extracting the smallest subgraph with the highest probability that some subgraph of that will match the query graph. See for example recent work in [17] and [19]. Controlling possible explosion has led to a body of techniques involving indexing, rule query pre-processing, and so forth. Deeper study and incorporation of these techniques is called for in expanding the scale of the H-Machine.

Using the meta-rule system to modify and optimize other rules is another promising direction. The current H-Machine has just scratched the surface here, and automation of these techniques is part of future efforts.

[14] The literature on these topics is vast and has a long history. Graph transformation is covered comprehensively by Ehrig [7]. Davis [5], and Davis and Lenat [6] did early work in meta-rules. Hewitt *et. al.*'s Omega [10] inspired the pursuit of self-contained meta-systems. Wolfram's NKS [21] provides a comprehensive view of simple CAs, and extends the ideas to graphs as well, with regard to very simple physical models and models of universal computation. Miller and Fredkin [14] describe a 3D reversible CA, also with conjectured universality. Wu and Rosenfeld [22] describe early forms of graph automata and their mathematical properties. Also similar to the work herein is Radul and Sussman's propagator model [16]. These are a small sample of the work in these fields.

The ability to build bottom-up dataflow graphs directly from high-level recursive equations inspires ideas of compilation to various representations, including those of sequential and parallel machines. This appears to be another path worth pursuing.

The H language itself is very simple, and at first I considered it to be like an assembly language for rules – i.e., low-level, and into which one translated a more "powerful" language. But H has shown that it conveys computational intent very compactly, allows an easy coupling of different kinds of data, and has attractive modularity properties. So while for some purpose a set of low-level rules might be developed into which one translates another language, I believe H stands on its own for many high-level expressions of computation.

References

1. Donald C. Allen, Seth A. Steinberg, Lawrence A. Stabile. Recent Developments in Butterfly Lisp, Proceedings of the Sixth National conference on Artificial intelligence - Volume 1, 1987.
2. T.Angeline and D.Narain Ponraj. A Survey on FFT Processors, International Journal of Scientific & Engineering Research Volume 4, Issue 3, March 2013. http://www.ijser.org/researchpaper/A-Survey-on-FFT-Processors.pdf.
3. Claude Berge. Hypergraphs, North-Holland, 1989.
4. Anwar Bhasha Pattan, Dr. M. Madhavi Latha. Fast Fourier Transform Architectures: A Survey and State of the Art, International Journal of Electronics & Communication Technology, 2014. http://www.iject.org/vol5.4/1/25-Anwar-Bhasha-Pattan.pdf.
5. Randall Davis. Meta-Rules: Reasoning about Control, MIT AI Lab Memo 576, March 1980.
6. Randall Davis and Douglas B. Lenat. Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill, 1982.
7. Hartmut Ehrig *et. al.* Fundamentals of Algebraic Graph Transformation, Springer, 2006.
8. Dustin Gage, et. al. Cellular Automata: Is Rule 30 Random?, http://www.cs.indiana.edu/~dgerman/2005midwestNKSconference/dgelbm.pdf
9. Graphviz, http://www.graphviz.org.
10. Carl Hewitt, Giuseppe Attardi, Maria Simi. Knowledge Embedding in the Description System Omega, Proceedings of the First AAAI Conference on Artificial Intelligence, 1980.
11. H-Machine code and pictures, https://app.sugarsync.com/wf/D744968_07071716_9998333. Most easily accessed by downloading the zip file (about 26 MB). Note that most of the graph dumps contain the rules as well.
12. Eric Jackowski. 734 Proposal FFT Survey, ca. 2006, http://homepages.cae.wisc.edu/~ece734/project/s06/jackowski_pro.doc.
13. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J. M.; Irwin, J. Aspect-oriented programming, Proceedings of the 11th European Conference on Object-Oriented Programming, 1997, http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf
14. Daniel B. Miller and Edward Fredkin. Two-state, Reversible, Universal Cellular Automata In Three Dimensions, 2005, http://www.digitalphilosophy.org.
15. Alan Oppenheim and Ronald Schafer. Digital Signal Processing, Prentice-Hall, 1975, page 290.
16. Alexey Radul and Gerald Jay Sussman. The Art of the Propagator, MIT-CSAIL-TR-2009-002, 2009.
17. Xuguang Ren, Junhu Wang. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs, Proceedings of the VLDB Endowment, Vol. 8, No. 5, 2015
18. L. Stabile. Chaos and Complexity, 2013.

19. Zhao Sun, *et. al*. Efficient Subgraph Matching on Billion Node Graphs, Proceedings of the VLDB Endowment, Vol. 5, No. 9, 2012.
20. Wikipedia, Hypergraph, https://en.wikipedia.org/wiki/Hypergraph. A good introduction, and a comprehensive bibliography.
21. Stephen Wolfram, A New Kind of Science, Wolfram Media, 2002. http://mathworld.wolfram.com/Rule30.html, http://mathworld.wolfram.com/Rule110.html.
22. Angela Wu and Azriel Rosenfeld, Cellular Graph Automata. I. Basic Concepts, Graph Property Measurement, Closure Properties, Information and Control 42, 305-329, 1979.