ρ_s-calculus
a language for stateful graph rewriting

Otto Jung <rose1@vau.place>

Abstract

We present a new programming language based on rewriting semantics. The language is homoiconic and stateful, and its rewrite rules are expressive enough to interpret the language itself. First we formally specify the semantics of ρ_s, and then we study some of its properties. By adopting a well-known notion of expressiveness, we show that most local features are expressible in the language.

Acknowledgements

Thank you to my supervisor for your invaluable insights, guidance and humor. We have spent countless hours in discussions around this thesis, and I have learned a lot from you in the process.

I am extremely lucky to have been a student of my teachers. The knowledge and skills that I acquired from them were essential in writing this work.

And I also want to thank my sister that always supports me, and my father, for the aspiration that he gave me for studying.

Contents

Motivation **4**

Context **4**

Term rewriting . . . 4
Term graph rewriting . . . 5
Programming by rewriting . . . 5

Preliminaries **8**

Notation and definitions . . . 8
Syntax for graphs . . . 10

ρ_s **calculus** **13**

Static language . . . 13
Dynamic language . . . 22
Properties . . . 30

Graph calculi **33**

Modularity . . . 35
Equality . . . 36
Expressiveness . . . 37

Expressiveness of ρ_s **38**

Library features . . . 38
Computable library features . . . 41

Conclusions and future work **55**

Appendix **55**

Motivation

The primary goal of this work is an attempt to utilize graph rewriting for general programming. To this end, we have designed a programming language based on dynamic graph rewriting rules, supporting direct manipulation of local state. These features are a natural extension of existing work, and they allow to conveniently express many programming concepts, without big losses to evaluation efficiency. Furthermore, thanks to these features, the semantics of the language can remain simple (relative to its goals).

The typical applications that we have in mind for this language are program transformation software and different types of calculators.

Context

Basing a language on rewriting semantics is a very old idea, and the theory of rewriting is very rich and reasonably well understood. In the following subsections we give a brief overview of both theory and practice, and show that it is not our goal that is original, but our approach.

Term rewriting

Our starting point is term rewriting systems [1]. These systems operate on "directed equations" called "rewrite rules", that are used to replace equivalent, in some sense, expressions, but only in the specified direction. By taking several rewrite rules and repeatedly applying the replacements, any computational process can be modeled.

As an example, take this addition algorithm expressed in terms of rewrite rules:

$$\begin{aligned} \texttt{add}(x, 0) &\to x \\ \texttt{add}(x, \texttt{s}(y)) &\to \texttt{s}(\texttt{add}(x, y)) \end{aligned}$$

Indeed, if natural numbers are encoded as $\{0, \texttt{s}(0), \texttt{s}(\texttt{s}(0)), ...\}$, then any expression of the form $\texttt{add}(x, y)$ eventually evaluates to the sum of x and y.

Central themes in term rewriting are:

- Termination – conditions under which the process of replacements terminates.
- Confluence – a phenomenon of different sequences of replacements resulting in the same expression.
- Completion – a method of constructing confluent systems.
- Conditional rewriting – an extension to rewrite process that allows a restriction on application of rules based on arbitrary predicates accompanying them.

- Theorem proving – a study in application of term rewriting to verification of logical formulas.

A broad overview of existing research is presented in [2]. For a more in-depth understanding, there is the classic book “Rewriting and all that” [3].

Term graph rewriting

The expressions, being the object of term rewriting, are actually trees[1]. It did not take long to generalize rewriting to graphs[1], which resulted in term graph rewriting [5].

The first use of graph rewriting was due to Wadsworth [6], with a goal to propose a *more efficient* implementation for languages based on lambda calculus. It was later proven that term graph systems are a superset of “regular” term rewriting systems, not of general term rewriting systems [7]. However, the inequality result relies on a rather technical difference between the two systems, and conceptually one can easily consider them a true generalization, especially in the context of programming. Furthermore, it is a standard technique to use graph rewriting for implementation of term rewriting languages [8], because of the efficiency that comes from the fact that identical subterms can be shared.

As mentioned, the main difference between term rewriting and term graph rewriting is the type of data structure that is being rewritten. Thus, many research topics overlap, but some developments are unique to graphs. Interaction nets [9] are one such example.

For a survey on term graph rewriting, see [10].

Programming by rewriting

Some of the earliest studies in term rewriting [11, 12, 13, 14] develop a method for automatic transformation of sets of equations into efficient programs that implement their logical meaning, and describe the usefulness of programming with such equations. We already saw how rules of addition can be translated into rewrite rules, now let us see an example in list processing, where we may have the following axioms:

$$\begin{aligned} \mathtt{car}(\mathtt{cons}(x, y)) &= x \\ \mathtt{cdr}(\mathtt{cons}(x, y)) &= y \end{aligned}$$

These directly translate to the appropriate rewrite rules by simply replacing the equal sign, so that the bigger expression reduces to the smaller one.

Likewise, here is a definition of “append”:

$$\begin{aligned} \mathtt{append}(\mathtt{nil}, z) &= z \\ \mathtt{append}(\mathtt{cons}(x, y), z) &= \mathtt{cons}(x, \mathtt{append}(y, z)) \end{aligned}$$

To concatenate a list x with a list y, we rewrite expression $\mathtt{append}(x, y)$ until it reaches its **normal form**, i.e., one which cannot be rewritten further.

[1]That are rooted, directed, labelled and ordered [4].

It is now known that any equational program (like those in logic programming) can similarly be expressed as a rewriting system [11].

Following the initial success, there were many further attempts to connect programming concepts with term rewriting. For example, a connection between Aspect Oriented Programming [15] and term rewriting is explored in [16], and the work [17] attempts to bring an exception mechanism [18] into a term rewriting language.

On the practical side, one of the first languages based on term rewriting was "Macsyma" [19], which made term rewriting the standard for implementing computer algebra systems [20], and gave rise to the well known "Maxima" [21], "Mathematica" [22], and "Maple" [23] systems that are developed to this day. Solving polynomial equations is a task perfectly suited to be programmed in terms of rewriting, and a typical use case for computer algebra systems.

Other application domains include automated theorem proving, universal algebra, parallel processing, transition systems, expert systems, and functional programming. More specifically, rewriting is used in re-engeneering of Cobol programs [24] and program transformations in general [25], as well as in semantics, where it gives meaning to programming languages [26], in constraint solving, where it underlies the solvers [27], and in everyday tasks, to normalize email addresses of `/etc/sendmail.cf` into canonical forms.

Concurrent Clean [28] deserves a special mention here. Even though the rules of Clean are limited to those expressible by function definitions from functional languages, its semantics is implemented in terms of term graph rewriting. Clean allows users to control its parallel order of evaluation, making efficient evaluation possible.

On the implementation side, we can divide existing work into two groups of languages, depending on their handling of the problem of confluence. The problem is that in the paradigm of functional programming, the only acceptable evaluation process is a confluent one – every expression must have at most one "value". In practice, when dealing with rewrite rules, there are two main ways to achieve this behaviour.

The first is to limit the expressive power of available rule choices, often by permitting only some specified types of patterns. For example, it is known that orthogonal [29] term rewrite systems are confluent. This notion of orthogonality may then be translated to the rewrite system of interest, and applied to the same effect.

The second way is to present an ability to specify a reduction strategy [30], that may, or may not, limit the resulting values, depending on whether every strategy can be encoded. Some term rewriting languages that focus on reduction strategies are "Stratego" [31], "PORGY" [32, 33], and "ELAN" [34]. But many others employ reduction strategies, among which are "Maude" [35], and "Tom" [36], "AGG" [37], "PROGRES" [38], "Fujaba" [39], "GrGen" [40], and "GP" [41].

To illustrate how strategies work, we briefly describe them in case of Stratego. Assume we have regular term rewrite rules $s1_1$ and s_2. Possibly, we want to apply them in sequence – first s_1, then s_2. For this, there is a built-in operator of sequential composition. After the two rules have been combined, they remain being a rule of the Stratego system, so further combinations are possible.

By default, rules in Stratego only apply to the root of the expression. If we would want to apply a rule to every subexpression at once, there is a builtin operator for that too. Finally, we can control how many times a rule is applied – maybe just one time, or however many times until it fails – by applying another builtin operator.

We chose the second way for our language, but instead of providing builtin evaluation strategies and combinators for them, we gave the rewrite rules enough power to express arbitrary evaluation strategy.

Dynamic rewriting, also called "higher order rewriting"[2], is when rules can manipulate rules – create them during rewrite process, or in some way modify the existing ones. In other words, dynamic rewriting is when rewrite rules are first-class objects [43] in the language.

Languages such as "ρ-calculus" [44], "ρ_g-calculus" [45], and the newer version of "Stratego" [25] are in the group focused on dynamic rewriting.

Take ρ-calculus as an example. A rewrite rule in this language is similar to abstraction in lambda calculus [46]. Given the term $T = 2 + 0$, and the rewrite rule $R = x + 0 \to x$, application of the rule to the term can be explicitly represented as $A = [x + 0 \to x](2 + 0)$. The result of the application is $U = \{2\}$. All of the objects mentioned here (T, R, A and U), are first-class in ρ-calculus. In particular, the rule constructor operator "$\to$" is a regular symbol, like "0" or "+". In general, terms are generated by the following grammar [47]:

$$t ::= x \mid f(t, ..., t) \mid \{t, ..., t\} \mid [t](t) \mid t \to t$$

We decided to also support dynamic rewriting in our language, similarly to how ρ-calculus does it.

Finally, let us discuss the "conditional" part of conditional rewriting. Languages such as "Pure" [48], "Concurrent Clean" [28], and "Stratego" [31] allow conditions to be based on equality of normal forms. In other words, in these languages one can restrict a rewrite rule to be applicable only when two chosen terms evaluate to the same term.

Another approach is taken by "PTTR" [49] and "PρLog" [50], among other relational languages. These languages allow first-order logical formulas to be associated with a rewrite rule, serving as a condition for applicability of the rule. In particular, this ensures that every rewrite rule is decidable, which is not the case with the first approach.

We adopt an entirely different way of conditional rewriting, which is related to how we handle state. It is however easy to show that conditions based on

[2]But this term is overloaded, ex. in [42] it stands for a different thing.

formulas (as in the second approach above) can be expressed using our method, but only for those formulas that yield finite relations.

We do not know whether there exist languages that support direct handling of state, similar to how we do it.

Preliminaries

Our definitions are fairly standard, but the notation is not. It is also not required to be familiar with the notation to understand the remaining parts.

Notation and definitions

Our main data structure is the ordered graph defined as follows:

Graph

Given an infinite set $\mathcal{U}$,

- a vertex is an element of $\mathcal{U}$,
- a graph is a partial function of type $\mathcal{U} \rightharpoonup \mathcal{U}^*$.

In the remaining parts we assume that the set $\mathcal{U}$ is fixed. Even though we will usually consider finite graphs, the set of vertices $\mathcal{U}$ is required to be infinite in order to support creation of fresh nodes during evaluation, since rewriting can generate those.

For the ease of readability, most of our variables have an associated type:

- n, i, k are numbers
- f, ψ are functions
- R, P, Q are relations
- g, h are graphs
- a, b, c, d, e are vertices

A graph is a function that associates vertices to *lists of* vertices. A list, or n-tuple, is written as $\langle x_1, ..., x_n \rangle$. We adopt some set notation for lists:

- If x is a list then $y \in x$ is true iff $x = \langle z_1, ..., y, ..., z_n \rangle$
- If x and y are lists, then notation $x \cup y$ stands for list concatenation, i.e. $\langle x_1, x_2, ... \rangle \cup \langle y_1, y_2, ... \rangle = \langle x_1, x_2, ..., y_1, y_2, ... \rangle$

Basic graph terminology follows. Vertex a is said to be a **child of** b in g when $a \in g(b)$. If either $a \in g(b)$ or $b \in g(a)$, then a and b are **adjacent**. A sequence of vertices $a_1, a_2, ..., a_n$ is called a **path** from a_1 to a_n if for each $i \in \{1, 2, ..., n-1\}$ it is true that $a_{i+1} \in g(a_i)$. If there is a path from a_1 to a_n, then a_n is said to be **reachable from** a_1, written as $\mathbf{E}^*_g(a_1, a_n)$.

We say that g is an **open graph** if there are vertices a, b such that $b \in g(a)$ and $b \notin \texttt{dom}(g)$. If graph is not open, then it is **closed**. Unless stated otherwise, all our graphs are closed by default.

The set of all graphs is $\mathbb{G}$ and $\emptyset$ is the empty graph (which is also the empty set).

We write $a \in g$ to mean that there exist edges to or from a in the graph g, i.e.

$$a \in g \iff \exists_{b}\ a \in g(b) \vee b \in g(a)$$

Also, if g is a graph, then $a(b_1, ..., b_n) \in g$ is the same as $g(a) = \langle b_1, ..., b_n \rangle$.

Vertices, edges, and roots of a graph are its basic parts, for access to which we use these functions:

- $\mathbf{V}(g)$ is the set of all vertices of g, i.e.
  $\mathbf{V}(g) = \{a : a \in g\}$
- $\mathbf{E}(g)$ is the set of all edges of g, i.e.
  $\mathbf{E}(g) = \{\langle a, b \rangle : b \in g(a)\}$
- $\mathbf{R}(g)$ is the set of roots of g, i.e.
  $\mathbf{R}(g) = \{a : a \in g \wedge \nexists_{b}\ a \in g(b)\}$
- $\mathbf{R}(g)_i$ is $g(a)_i$ if $\{a\} = \mathbf{R}(g)$ and $\perp$ otherwise
- $\mathbf{R}(g)_{1...}$ is $g(a)$ if $\{a\} = \mathbf{R}(g)$ and $\perp$ otherwise

Graph h is a **subgraph** of graph g, written $h \leq g$, if h has some of the vertices of g, and some of the edges connecting those vertices in g. Written formally:

$$\begin{aligned} h \leq g \iff \quad & \mathbf{V}(h) \subseteq \mathbf{V}(g) \\ & \wedge \forall_{a \in \mathbf{V}(h)}\ h(a) = g(a) \vee h(a) = \langle\rangle \end{aligned}$$

One way to get a subgraph of g is to pick a vertex and all of the vertices and edges that are reachable from it. We say that the resulting graph is **a closure** of g, and use notation $g(a)^*$ for it, with precise meaning being:

$$g(a)^* = h \iff \quad h \subseteq g \wedge \forall_{b}\ (h(b) = g(b) \iff \mathbf{E}^*_g(a, b))$$

A **cycle** is a path of positive length from any vertex to itself. If a graph contains cycle as a subgraph, then it is a **cyclic graph**, otherwise it is an **acyclic graph**.

Finally, to talk about modification of graphs we introduce a notation for that too.

Given two vertices $a \in g, b \notin g$, to replace a by b in g we write $g[a := b]$. Formally:

$$\frac{h = g[a := b]}{\begin{array}{rcl} g(a) = \langle c_1, ..., c_n \rangle & \iff & h(b) = \langle c_1, ..., c_n \rangle \\ g(d) = \langle c_1, ..., a, ..., c_n \rangle & \iff & h(d) = \langle c_1, ..., b, ..., c_n \rangle \end{array}}$$

If a is a vertex in g, then to replace its list of children by x we write $g[a := x]$. Formally, this expands to $(g \setminus (\{a\} \times \mathcal{U})) \cup \{\langle a, x \rangle\}$.

To insert a graph h into a graph g such that the root b of h is at vertex $a \in g$, we write $g[a := h]$. This is equivalent to adding h to g, and then changing a to point to the children of b:

$$g[a := h] = (g \cup h)[a := h(b)]$$
$$\text{where } \{b\} = \mathbf{R}(h)$$

Syntax for graphs

When writing down examples of graphs, a concise notation is convenient. In this section we propose one that is actually used in the implementation of our graph-based language. Intuition for this notation is that it is the standard notation for trees, except that some subgraphs can be named, and whenever the same name is used in different places, the relevant subgraph is shared in those places. Let us illustrate it by examples:

Example 1.

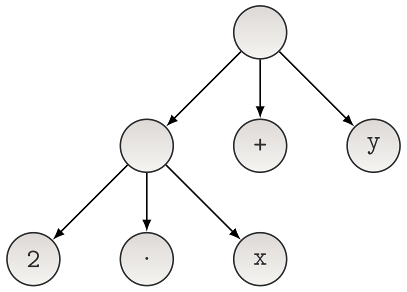

Figure 1: Expression `((2 · x) + y)`.

Figure 1 shows a tree-like term translated into a graph. Nodes on the graph that are not named are the same nodes that do not have a name in the original expression (names are purely for the reader, there are no names and no labels on the actual graph, just elements of $\mathcal{U}$ that we decided to give names to).

If $\mathcal{U} = \mathbb{N}$, then both the figure and the textual expression represent graph g (and every graph isomorphic to it) given by the equations:

$$
\begin{aligned}
g(0) &= \langle 1, 2, 3 \rangle \\
g(1) &= \langle 4, 5, 6 \rangle \\
g(2) &= \langle \rangle \\
g(3) &= \langle \rangle \\
g(4) &= \langle \rangle \\
g(5) &= \langle \rangle \\
g(6) &= \langle \rangle
\end{aligned}
$$

where 4 is named "2", 5 is named "$\cdot$", 6 is named "x", 2 is named "$+$", and 3 is named "y".

Example 2.

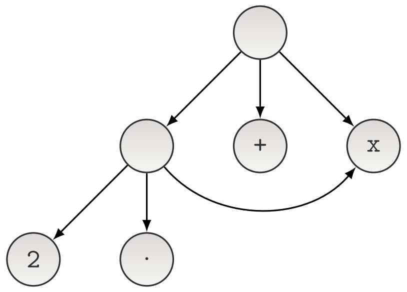

Figure 2: Expression `((2 · x) + x)`.

On Figure 2, the name `x` is now used in two places, so the node that it names must be shared.

Example 3.

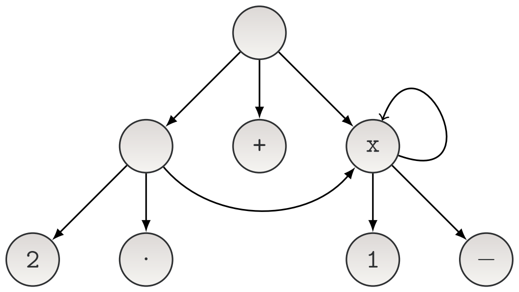

Figure 3: Expression `(`**`let`**` ((x (1 — x))) ((2 · x) + x))`.

On Figure 3 we see the same graph as before, except that `x` is now a subgraph with a cycle. This example illustrates that **let** is a special form that gives names to subgraphs.

Example 4.

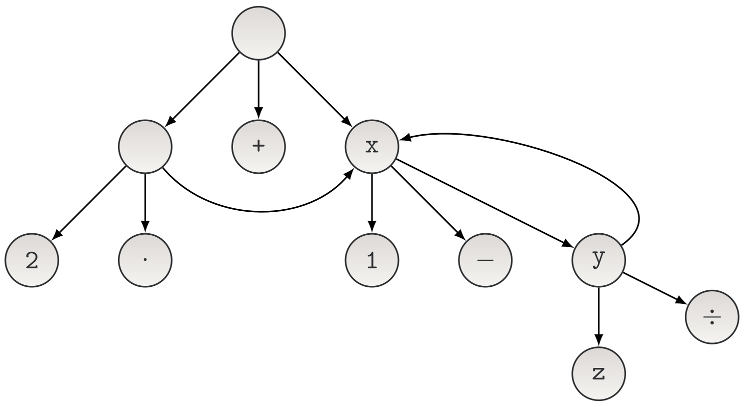

Figure 4: Expression `(let ((x (1 − y)) (y (z ÷ x))) ((2 · x) + x))`.

The example on Figure 4 demostratets how names can be cross-referenced within **let** – the subgraph of `x` refers to `y`, and the subgraph of `y` refers to `x`.

For the formal definition of parsing of such expressions, we have the grammar (in BNF [47] notation)

$$\begin{array}{rlll} \textbf{Tree} & t & ::= & v \mid (t_1 \ldots t_n) \mid l \\ \textbf{Let} & l & ::= & (\textbf{let}\ (b_1 \ldots b_n)\ t_1 \ldots t_k) \\ \textbf{Binding} & b & ::= & (v\ t) \\ \textbf{Variable} & v & & \end{array}$$

and the function that does the parsing

$$
\begin{aligned}
&\texttt{parse} : \textbf{Tree} \to \mathbb{G}\\
&\texttt{parse}(v) = \{\langle v, \langle\rangle\rangle\}\\
&\texttt{parse}((t_1 \ldots t_n)) = \begin{cases} g, & \text{if } |\mathbf{R}(g)| = 1\\ g \cup \{r\}, & \text{otherwise}\end{cases}\\
&\quad \text{where}\\
&\qquad g = \texttt{parse}(t_1) \cup \ldots \cup \texttt{parse}(t_n)\\
&\qquad r = \langle x, \langle r_1, \ldots, r_n\rangle\rangle\\
&\qquad \{r_i\} = \mathbf{R}(\texttt{parse}(t_i))\\
&\qquad x \notin g\\
&\texttt{parse}((\textbf{let}\ (b_1 \ldots b_n)\ t_1 \ldots t_k)) = g\\
&\quad \text{where}\\
&\qquad g = (g_1 \cup \ldots \cup g_n)[v_1 := u_1][v_1 := r_1]\ldots[v_n := u_n][v_n := r_n]\\
&\qquad g_i = \texttt{parse}(t_i)\\
&\qquad \{r_i\} = \mathbf{R}(u_i)\\
&\qquad u_i = \texttt{parse}(m_i)\\
&\qquad b_i = (v_i\ m_i)
\end{aligned}
$$

The function is non-deterministic in the sense that depending on the choices of x the result is a different graph, but all of them represent the same expression. Furthermore, every graph isomorphic to the resulting graph is also represented by the parsed expression.

ρ_s calculus

In this chapter we give a formal semantics for our language. It is dynamic, meaning that rewrite rules can appear, and dissapear, during evaluation. And it is non-local, meaning that in a single step of evaluation, we can perform rewrites in multiple places.

Static language

To begin with, we introduce a static version of the language, static-ρ_s. A program in static-ρ_s is a ruleset plus an input graph, and its evaluation is the repeated application of rewrite rules from the ruleset to the input graph.

Program structure

A single rewrite rule in static-ρ_s consists of a series of "blocks". A block is like a traditional rule from term-rewriting, but with a dedicated "input argument". Blocks are connected with a sort of an "and", which means that in order to match the rule, all match patterns must be matched, and in order to rewrite according to the rule, rewrites need to happen in order, block-by-block, left-to-right.

The role of the input argument will be apparent when we get to the semantics, but for now, let us define the structure first:

Rewrite block

A rewrite block $\mathscr{b}$ is a triple $\langle \texttt{input}, \texttt{mpattern}, \texttt{rpattern} \rangle$ where

- $\texttt{input}(\mathscr{b}) : \mathcal{U}$ is the argument that binds to the root of the graph to be matched and replaced;
- $\texttt{mpattern}(\mathscr{b}) : \mathbb{G}$ is the match pattern;
- $\texttt{rpattern}(\mathscr{b}) : \mathbb{G}$ is the replace pattern.

The set of all rewrite blocks is $\mathcal{B}$.

Example 5.
For an example block we can take an arbitrary input argument, and any two expressions:

$$\mathscr{b} = \langle \texttt{v}, \texttt{(c · (x + y))}, \texttt{((c · x) + (c · y))} \rangle$$

The implied meaning of this rewrite block is that it simplifies arithmetic expressions according to the distributive law, but that is only if we assume that `+` and `·` are constants and that other blocks in the program also treat them as arithmetic operators.

A rewrite rule also has an input argument:

Rewrite rule

A rewrite rule r is a pair $\langle \texttt{input}, \texttt{blocks} \rangle$ where

- $\texttt{input}(r) : \mathcal{U}$ is the argument that binds to the root of the subgraph that the rule currently applies to;
- $\texttt{blocks}(r) : \mathcal{B}^*$ is the list of blocks.

The set of all rewrite rules is $\mathcal{R}$.

Again, the role of input argument will be clear later.

Static program

A static program p is a triple $\langle \texttt{input}, \texttt{constant}, \texttt{ruleset} \rangle$ where

- $\texttt{input}(p) : \mathbb{G}$ is the input term;
- $\texttt{constant}(p) : 2^{\mathcal{U}}$ is the set of constant nodes;
- $\texttt{ruleset}(p) : 2^{\mathcal{R}}$ is the set of rewrite rules.

The set of constants is global per program, and thus blocks and rules have precise meaning only in the context of a program.

Program evaluation

Consider a single rewrite block. By analogy to the standard term-rewriting, a successful match means that there is a way to assign some nodes of the input graph to the nodes of the match pattern graph, preserving the graph structure and ensuring that constant nodes map to themselves. In other words, there must be a homomorphism from the pattern graph to the input graph, restricted by constants. We generalize this notion of homomorphism to multi-ary homomorphism, which maps a single vertex to a list of vertices. The rationalization for it is that we want to be able to have match patterns that are generic in the arity of the input term.

Multi-ary homomorphism

A function $\phi : \mathcal{U} \rightharpoonup \mathcal{U}^*$ is a multi-ary homomorphism from h to g iff

$$\bigforall_{a} \left(h(a) = \langle a_1, a_2, ..., a_n \rangle \iff \bigforall_{b \in \phi(a)} g(b) = \phi(a_1) \cup \phi(a_2) \cup ... \cup \phi(a_n)\right)$$

We write $\phi[\![h]\!] = g$ for short.

It is not required for ϕ to be the smallest such function.

Matching. With that, we are ready to define what constitutes a match. The notation $g \models^{\phi} h$ denotes a successful match, in which the graph g matches graph h with the multi-ary homomorphism function ϕ.

We can look at matching as a model checking problem, where the formula is the match pattern, which is a formula with no constants, with all variables being free, and where the model is the input graph. Then the input graph g is a model of match pattern h if the image of h under ϕ is a subgraph of g, with the root of g corresponding to the root of h. The following rule captures this intuition:

$$\frac{\phi[\![h]\!] \leq g \quad \phi(d) = \langle e \rangle \quad \mathbf{R}(h) = \{d\} \quad \mathbf{R}(g) = \{e\}}{g \models^{\phi} h}$$

Recall that the homomorphism function returns not a single node, but a list of nodes, possibly empty, which means that every pattern node can match zero or more input nodes. This in turn suggests that matching is non-deterministic, and it in fact is.

Example 6.
The graph `(x y)` matches the graph `(1 2)` with these multi-ary homomorphisms:[3]

$$\begin{aligned} \phi_1(r) &= \langle r \rangle \\ \phi_1(\mathtt{x}) &= \langle \mathtt{1} \rangle \\ \phi_1(\mathtt{y}) &= \langle \mathtt{2} \rangle \end{aligned}$$

$$\begin{aligned} \phi_2(r) &= \langle r \rangle \\ \phi_2(\mathtt{x}) &= \langle \rangle \\ \phi_2(\mathtt{y}) &= \langle \mathtt{1}, \mathtt{2} \rangle \end{aligned}$$

$$\begin{aligned} \phi_3(r) &= \langle r \rangle \\ \phi_3(\mathtt{x}) &= \langle \mathtt{1}, \mathtt{2} \rangle \\ \phi_3(\mathtt{y}) &= \langle \rangle \end{aligned}$$

$$\begin{aligned} \phi_4(r) &= \langle r \rangle \\ \phi_4(\mathtt{x}) &= \langle \rangle \\ \phi_4(\mathtt{y}) &= \langle \rangle \end{aligned}$$

where r is the root of both the pattern and the input graph. Note that the homomorphism function can give the value $\langle \rangle$ to its every node, except for the root, and that will result in a valid match.

Example 7.
The graph `(x y)` matches the graph `(1 2 3)` with ϕ defined as

$$\begin{aligned} \phi(r) &= \langle r \rangle \\ \phi(\mathtt{x}) &= \langle \mathtt{1} \rangle \\ \phi(\mathtt{y}) &= \langle \mathtt{2}, \mathtt{3} \rangle \end{aligned}$$

being one of possible homomorphisms.

[3]Technically, listed homomorphisms are only those smallest homomorphisms that match, but there are infinitely more – those that have redundant mappings.

Example 8.

The graph `(x (y z))` matches the graph `(1 (2 3) (2 4))` with ϕ defined as

$$\begin{aligned}\phi(r) &= \langle r\rangle\\ \phi(\mathtt{x}) &= \langle \mathtt{1}, z_1, z_2\rangle\\ \phi(m) &= \langle\rangle\\ \phi(\mathtt{y}) &= \langle\rangle\\ \phi(\mathtt{z}) &= \langle\rangle\end{aligned}$$

being one of possible homomorphisms, where z_1 and z_2 are the roots of subgraphs `(2 3)` and `(2 4)`, and m is the root of the subgraph `(y z)`.

Replacement. Replacement may result in creation of fresh nodes that were not present in the original graph. To express such behaviour we introduce new notation. Let X be a set of nodes, then $\phi \mid X$ is an extension of ϕ:

$$\frac{\phi(a) = \bot}{(\phi \mid X)(a) = \langle X_a\rangle} \qquad \frac{\phi(a) \neq \bot}{(\phi \mid X)(a) = \phi(a)}$$

Note that in the first consequence, X is used an an *indexed* set. This indexation is arbitrary because all fresh nodes are equivalent. But its construction can be done deterministically – sort the universe $\mathcal{U}$, then sort X, and then map the sorted elements according to their positions.

After a successful match there is the replace step which is handled by the function `replace`.

While in term-rewriting the rules are applied to subterms of the input term, our rules are non-local, so they can potentially modify multiple places in the graph. Yet, applications in our language also happen "at a point", and so the replace also happens at a chosen vertex.

Thus, the function `replace` receives a replace pattern h, an input graph g together with a vertex a, a multi-ary homomorphism function ϕ, and a set of fresh nodes X. Then it constructs a new graph g_s by applying the homomorphism function to the replace pattern. And finally, it inserts the new graph back, such that its root is at a in the original graph.

$$\begin{aligned}\mathtt{replace}(h, g, a, \phi, X) &= g[a := g_{\mathrm{s}}]\\ \text{where } g_{\mathrm{s}} &= \phi'[\![h]\!]\\ \text{and } \phi' &= \phi \mid X\end{aligned}$$

Note that only the children of the input subgraph are being replaced. The intuition for this is that the children are the value of the input subgraph, and replacement changes just the value, not the reference itself.

Example 9.

Given a replace pattern $h =$ `(x y z)`, with root at a, an input graph $g =$ `()` with root also at a, a good enough set of fresh nodes X, and a homomorphism ϕ defined as

$$\begin{aligned}\phi(a) &= \langle a \rangle \\ \phi(\mathtt{x}) &= \langle \mathtt{1} \rangle \\ \phi(\mathtt{y}) &= \langle \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5} \rangle \\ \phi(\mathtt{z}) &= \langle \mathtt{6} \rangle\end{aligned}$$

then $\mathtt{replace}(h, g, a, \phi, X)$ returns the graph `(1 2 3 4 5 6)`.

Rule closure. So far, we have discussed semantics related to a single match/replace pattern pair, but as we have defined it, rewrite rules in static-ρ_s consist of an arbitrary number of such pairs, embedded in rewrite blocks. Rule semantics is similar to that of blocks in that there are two steps – first the match, and then the replace. Match step is the match of all of the rewrite blocks, and the rewrite step is a sequence of replacements according to replace patterns of the rewrite blocks. Crucially, the homomorphism is shared between blocks.

Before we proceed with the semantics for rules, let us introduce another extension of the homomorphism function:

$$\frac{\phi(a) = \bot}{\phi(a \mid b) = \langle b \rangle} \qquad \frac{\phi(a) \neq \bot}{\phi(a \mid b) = \phi(a)} \; (\phi \text{ with default value})$$

We write $r \models^{\phi}_{X} g \odot a$ to say that the rule r matches g at a, where X is the set of constants, and ϕ is the relevant homomorphism.

Computationally, homomorphism is the result of a succesful match – it is the only unknown in the formula. Even though our rewrite rules contain multiple match patterns, the statement is true for them as well. By introducing a composition operator "$\circ$", we can define match results inductively.

$$\frac{\begin{gathered} a = \mathtt{input}(\mathcal{b}) \\ \langle b \rangle = \phi(a \mid a) \\ g' = g(b)^{*} \\ h = \mathtt{mpattern}(\mathcal{b}) \\ \phi' \in \min_{|\phi_i|} \{\phi_i : h \models^{\phi_i} g' \wedge \phi \subseteq \phi_i\} \end{gathered}}{\phi \circ \mathcal{b} = \phi' \pmod{g}}$$

The composition simply appends the minimal required mappings to the existing homomorphism. The subgraph to which the match pattern applies is a subgraph of the input term, so we need to have it in the context, which is written as $(\mathrm{mod}\, g)$. This subgraph also depends on the `input` of $\mathcal{b}$, so the order of blocks matters here (unless there are no cycles between them). When `input` is bound by ϕ to be the actual input node to be matched, it is also checked that its value is a singleton list. When `input` is not bound, then it is independent of the currently matched subgraph, and so it should be understood as a "memory reference", as it can be used to model state.

Now it is a matter of picking the initial ϕ and composing it with all the blocks that the rule is made of. That initial ϕ is one that maps constant nodes to themselves, and with the root of the currently chosen subgraph mapped to the `input` of the rule. This way, if the block `input` is the same as the rule `input`, then the block is applied to the root of the currently chosen subgraph.

$$\frac{\begin{array}{c} \texttt{input}(r) = c \\ \phi_0 = (id \cap X^2) \cup \{\langle c, a \rangle\} \\ \texttt{blocks}(r) = \langle \mathscr{b}_1, \mathscr{b}_2, ..., \mathscr{b}_n \rangle \\ \phi = \phi_0 \circ \mathscr{b}_1 \circ \mathscr{b}_2 \circ ... \circ \mathscr{b}_n \pmod{g} \end{array}}{r \models^{\phi}_{X} g \odot a}$$

Here, id is the identity function.

Example 10.

Given a rule r with $\texttt{input}(r) = c$ and $\texttt{blocks}(r) = \langle \mathscr{b}_1, \mathscr{b}_2, \mathscr{b}_3 \rangle$ with

$$\begin{array}{l} \texttt{input}(\mathscr{b}_1) = c \\ \texttt{mpattern}(\mathscr{b}_1) = \texttt{(x y z)} \\ \texttt{input}(\mathscr{b}_2) = \texttt{x} \\ \texttt{mpattern}(\mathscr{b}_2) = \texttt{x} \\ \texttt{input}(\mathscr{b}_3) = \texttt{z} \\ \texttt{mpattern}(\mathscr{b}_3) = \texttt{z} \end{array}$$

and an input graph $g = \texttt{(1 2 3 4 5 6)}$ with root in a, then $r \models^{\phi}_{\emptyset} g \odot a$ returns the homomorphism ϕ defined as:

$$\begin{array}{l} \phi(a) = \langle a \rangle \\ \phi(\texttt{x}) = \langle 1 \rangle \\ \phi(\texttt{y}) = \langle 2, 3, 4, 5 \rangle \\ \phi(\texttt{z}) = \langle 6 \rangle \end{array}$$

and it is the only homomorphism that matches.

The reason is that we have variables `x` and `y` as input arguments, which forces them to match only singleton lists of variables.

In principle, by requiring all nodes to be input arguments in this way, we can express regular, non multi-ary homomorphism.

Example 11.

With multiple blocks at our disposal, we can finally show an example match that fails:

Given a rule r with $\texttt{input}(r) = c$ and $\texttt{blocks}(r) = \langle \mathscr{b}_1, \mathscr{b}_2, \mathscr{b}_3 \rangle$ with

$$\begin{aligned}
&\texttt{input}(\mathscr{b}_1) = c\\
&\texttt{mpattern}(\mathscr{b}_1) = \texttt{(x (y z))}\\
&\texttt{input}(\mathscr{b}_2) = \texttt{x}\\
&\texttt{mpattern}(\mathscr{b}_2) = \texttt{x}\\
&\texttt{input}(\mathscr{b}_3) = \texttt{y}\\
&\texttt{mpattern}(\mathscr{b}_3) = \texttt{y}
\end{aligned}$$

and an input graph $g = \texttt{(1 (2 3) (2 4))}$ with root at a, then there is no homomorphism that makes $r \models^{\phi}_{\emptyset} g \odot a$ true.

The reason is that the vertices $\texttt{y}, \texttt{z}$ need to be mapped to both $\langle \texttt{2}, \texttt{3} \rangle$ and $\langle \texttt{2}, \texttt{4} \rangle$, which is not possible by our definition of multi-ary homomorphism.

The replacement step is now also defined inductively, but the result of replacement is a graph, not a homomorphism.

$$\frac{\begin{array}{c} a = \texttt{input}(\mathscr{b}) \\ \langle b \rangle = \phi(a \mid a) \\ h = \texttt{rpattern}(\mathscr{b}) \\ X = \texttt{regular} \setminus \mathbf{V}(g) \\ g' = \texttt{replace}(h, g, b, \phi, X) \end{array}}{g \circ \mathscr{b} = g' \pmod{\phi}}$$

The context is now the homomorphism ϕ that is the result of the previously performed match. The set of fresh nodes X is the set of "regular" nodes that are not present in the original graph. For static-ρ_s every node is regular, but for ρ_s there are two non-regular nodes (named **eval** and **root**). Basically, a node is regular if it can be swaped with any other regular node, not affecting the final structure. A formal definition will be given in later chapters.

Likewise, we define the base step of the induction – it is the input subgraph in its original form – and get to the (unique) resulting value by applying the composition operator:

$$\begin{array}{c} \texttt{replace*}(r, g, \phi) = g \circ \mathscr{b}_1 \circ \mathscr{b}_2 \circ \ldots \circ \mathscr{b}_n \pmod{\phi} \\ \text{where } \langle \mathscr{b}_1 \circ \mathscr{b}_2 \circ \ldots \circ \mathscr{b}_n \rangle = \texttt{blocks}(r) \end{array}$$

Putting the two steps together, we get the match-rewrite step semantics:

$$\frac{\begin{array}{c} r \models^{\phi}_{X} g \odot a \\ g' = \texttt{replace*}(r, g, \phi) \end{array}}{g \odot a \rhd g' \pmod{r, X}}$$

The context is a rewrite rule r and a set of constant nodes X.

A step in the evaluation of a static-ρ_s program is a match-rewrite step with any of the available rewrite rules applied to any of the input graph nodes:

$$\frac{\begin{array}{c} r \in \texttt{ruleset}(p) \\ X = \texttt{constant}(p) \\ g \odot a \triangleright g' \pmod{r, X} \end{array}}{g \odot a \triangleright g' \pmod{p}} \qquad \frac{g \odot a \triangleright g' \pmod{p}}{g \triangleright g' \pmod{p}}$$

Example 12.
The following example demostrates a non-confluent system.
Given a program p defined by:

$$\begin{array}{l}
\texttt{input}(p) = \texttt{(0)} \\
\texttt{constant}(p) = \{\texttt{1}, \texttt{2}\} \\
\texttt{ruleset}(p) = \{r_1, r_2\} \\
\texttt{input}(r_1) = c \\
\texttt{blocks}(r_1) = \langle b_1 \rangle \\
\texttt{input}(b_1) = c \\
\texttt{mpattern}(b_1) = \texttt{(x)} \\
\texttt{rpattern}(b_1) = \texttt{(1)} \\
\texttt{input}(r_2) = c \\
\texttt{blocks}(r_2) = \langle b_2 \rangle \\
\texttt{input}(b_2) = c \\
\texttt{mpattern}(b_2) = \texttt{(x)} \\
\texttt{rpattern}(b_2) = \texttt{(2)}
\end{array}$$

after a single match-rewrite step, we get two values – `(1)` and `(2)`, depending on which of the two rules is chosen.

Finally, the semantics of static ρ_s is just the transitive reflexive closure of the relation $g \triangleright g' \pmod{p}$, without the intermediate edges:

$$\frac{\begin{array}{c} g = \texttt{input}(p) \\ g \triangleright^* g' \pmod{p} \\ \nexists_{g''}\ g' \triangleright g'' \pmod{p} \end{array}}{\langle p, g' \rangle \in \text{static-}\rho_s}$$

So, the values of a static program p are: $\{g : \langle p, g \rangle \in \text{static-}\rho_s\}$.

Example 13.

Given a program p defined by:

$$\begin{aligned}
&\texttt{input}(p) = \texttt{(1 + 2 + 3 + 4 + 5)}\\
&\texttt{constant}(p) = \{+, -\}\\
&\texttt{ruleset}(p) = \{r\}\\
&\texttt{input}(r) = c\\
&\texttt{blocks}(r) = \langle \mathscr{b} \rangle\\
&\texttt{input}(\mathscr{b}) = c\\
&\texttt{mpattern}(\mathscr{b}) = \texttt{(xs} + \texttt{ys)}\\
&\texttt{rpattern}(\mathscr{b}) = \texttt{(xs} - \texttt{ys)}
\end{aligned}$$

The final value of the program is `(1 − 2 − 3 − 4 − 5)`, but the intermediate ones are with all the combinations of $+$ and $-$ in them.

Dynamic language

The language that we are proposing is a dynamic version of static-ρ_s, called ρ_s. The main difference is that in ρ_s programs are just graphs. But, conceptually, the transition from static to dynamic semantics is very simple:

1. define redex,
2. specify evaluation strategy,
3. specify encoding of static-ρ_s programs on the input graph.

A redex is a subgraph that encodes a static-ρ_s program. More specifically:

Redex

A redex in g is a closure $g(c)^*$ for some c such that $g(c) = \langle \mathbf{eval}, a, b \rangle$.

The meaning of $\mathbf{eval}, a$ and b is that **eval** is a node specifically chosen to denote the root of a redex (all redexes share it), a represents an encoded static-ρ_s program, and b limits the scope of application of the program[4].

The evaluation strategy is "parallel inner-most" which means that bottom-most redexes are reduced first, except that when there are many bottom-most redexes, then only one of them is reduced first.

The last step is the most arbitrary, so we will deal with it later.

The particular details are a bit more complicated, so in the following we define the precise semantics of ρ_s.

There are two notions that we will use in definitions – those of environment and focus. An environment is just a subgraph of the original graph that encodes

[4]In our definition, a redex does not have to be immediately reducible, it is enough to look like a reducible expression.

the “currently applicable” rewrite rule. It always comes with an accompanying subgraph to which it applies, called the “body”. A focus is a vertex that identifies the subgraph to which evaluation is restricted. It is common to define term-rewriting semantics by “subterm recursion”. What we do is “subgraph recursion”, where the subgraph is uniquely determined by the focused vertex. We use the notation $g \odot a$ to say that g is currently focused on a.

To provide a high-level description of the semantics, we imagine an evaluator that implements it. The evaluator runs one step at a time, with each step associated with a particular environment. Given an input graph, the evaluator first searches for a valid bottom-most environment. When found, the evaluator decodes its rewrite rule, and tries to apply it to the accompanying body. Application is successful if there is any vertex in the body that matches the rule. Each step ends with a succesful application that changes the input graph, and the whole process is just a series of steps applied to the input graph.

Every ambiguity, be it with choosing the appropriate environment (since there can be many “bottom-most”), or with choosing the one vertex among many body vertices that match the rule, is resolved by running all possibilities in parallel. Formally this means that every choice is non-deterministic.

Reduction

We use small-step operational semantics for our calculus, which means that a relation is defined that evaluates a single step of the evaluation process, so that the transitive closure of it is the target language semantics. We first define the relation $g \odot a \underset{h}{\rightarrow} g'$. The interpretation of it is that the graph g focused on a reduces to g' in the environment h. We then fill in the defaults, and then close it to get the ρ_s calculus. The following rules operate on the whole graph, with a “focus” serving as a pointer to the currently evaluating subgraph. This is because, unlike with trees, the “substructure” that we recurse into is a general subgraph, and it is not easy to perform operations on a subgraph and then somehow propagate the results up.

$$\frac{\nexists_{g'}\ g \odot a \underset{\emptyset}{\rightarrow} g'}{g \odot a \Downarrow}\ (\texttt{R-NORM})$$

$$\frac{g \odot a \Downarrow \quad d \in g(a)^* \quad g \odot d \triangleright g' \pmod{h}}{g \odot a \underset{h}{\rightarrow} g'}\ (\texttt{R-REWR})$$

$$\frac{d(\mathbf{eval}, e, c) \in g(a)^* \quad d \notin g(c)^* \quad g \odot c \xrightarrow[g(e)^*]{} g'}{g \odot a \underset{h}{\rightarrow} g'}\ (\texttt{R-EVAL})$$

Rule `R-NORM` is just a normal form predicate. It refers to the main relation, but with an empty environment, so it does not ever recurse to itself because the next rule is the only user of `R-NORM`, and it rejects empty environments. That next one is R-REWR, it checks if the focused subgraph is already normalized, then picks a node, and performs a match-rewrite step on it by decoding the static program from environment h and then performing one step in the evaluation of the program. The last one, R-EVAL, searches for the special node, named **eval**, and if found, it updates the environment, and recurses down. If g is finite, then there is always a finite amout of recursion steps to be done, because the predicate does not work unless there are no cycles in the subgraph (the second clause in the premise checks that).

Note that there is a great lot of non-determinism that comes from three factors:

1. the rule R-REWR can choose any of the subgraph nodes to rewrite,
2. the match-rewrite step is also not deterministic, as there could be many ways to match a pattern,
3. the rule R-EVAL can recurse on any of the evals below it.

Before we define the values of a ρ_s program, we must turn our attention to the problem of garbage collection. After a replace step is performed, the graph is often split in such a way that there is the "new" part that we consider the true result, and there is the part that is no longer needed and cannot be accessed from the new part. For example, this happens if we replace the children of a node a with the empty list of children when a was the only parent for those children. To deal with it, we decide that every graph has a root, and it is only the vertices reachable from the root that are interesting to us. With another special vertex $\mathbf{root} \in \mathcal{U}$ we are ready to define the full semantics, similarly to how we did it in the case of the static language:

$$\frac{g \odot a \underset{\emptyset}{\rightarrow} g'}{g \rightarrow g'} \qquad \frac{\begin{array}{c} g \xrightarrow{*} g' \\ \nexists_{g''}\ g' \rightarrow g'' \\ a = \mathbf{root} \end{array}}{\langle g, g'(a)^{*} \rangle \in \rho_s}$$

There are at least two technical observations that require attention. First, the **root** is not actually required to be one of the roots of the input graph or the resulting graph. But in all our examples, it is. Second, it is important to note that if g is in normal form with respect to $g \rightarrow g'$, then its closure starting from **root** also is (see Lemma 3).

Parsing

At this point we are left to describe how the application of $g \odot d \triangleright g' \pmod{h}$ works, that is, how parsing of the encoded program h is done.

Let us assume that a static program p is encoded. First, we require $\mathbf{R}(h)_{1\ldots} = \langle a, b, c \rangle$, and then for each e_i in $\langle e_1, e_1, \ldots e_n \rangle = h(c)$, it must be that $h(e_i) = \langle a_n, b_i, c_i \rangle$.

The input term is the whole ρ_s program, so $\texttt{input}(p) = g$. There is always only a single rewrite rule in the encoded programs, so $\texttt{ruleset}(p) = \{r\}$. Then $\texttt{input}(r) = a$, and $\texttt{constant}(p) = \mathbf{R}(b)_{1...}$. The subgraph $h(c)^*$ encodes $\texttt{blocks}(r)$ such that each e_i points to an encoded block.

Given that $\texttt{blocks}(r) = \langle \mathscr{b}_1, \mathscr{b}_2, ..., \mathscr{b}_n \rangle$, we have

- $\texttt{input}(\mathscr{b}_i) = a_i$,
- $\texttt{mpattern}(\mathscr{b}_i) = b_i$,
- and $\texttt{rpattern}(\mathscr{b}_i) = c_i$.

If any of the assignments fail, then the encoding is invalid, and the result of the match/rewrite step is as if the patterns did not match the input subgraph.

Formalizing the above description, we get the decoding function:

$$
\begin{array}{l}
\texttt{decode}(g, h) = p \\
\quad \text{where} \\
\qquad \mathbf{R}(h)_{1...} = \langle a, b, c \rangle \\
\qquad h(c) = \langle e_1, e_1, ...e_n \rangle \\
\qquad h(e_i) = \langle a_i, b_n, c_i \rangle \\
\qquad \texttt{input}(p) = g \\
\qquad \texttt{ruleset}(p) = \{r\} \\
\qquad \texttt{constant}(p) = \mathbf{R}(b)_{1...} \\
\qquad \texttt{input}(r) = a \\
\qquad \texttt{blocks}(r) = \langle \mathscr{b}_1, \mathscr{b}_2, ..., \mathscr{b}_n \rangle \\
\qquad \texttt{input}(\mathscr{b}_i) = a_i \\
\qquad \texttt{mpattern}(\mathscr{b}_i) = b_i \\
\qquad \texttt{rpattern}(\mathscr{b}_i) = c_i
\end{array}
$$

and the semantics for match/rewrite step:

$$
\frac{\begin{array}{c} p = \texttt{decode}(g, h) \\ g \odot d \triangleright g' \pmod{p} \end{array}}{g \odot d \triangleright g' \pmod{h}}
$$

Example 14.

Given input graph $g =$ `(1 + 2 + 3 + 4 + 5)` and environment h:

```
(g (+ -)
   ((g (xs + ys) (xs - ys))))
```

we get `decode`$(g, h) = p$, where p is defined by:

$$\begin{aligned}
&\texttt{input}(p) = \texttt{(1 + 2 + 3 + 4 + 5)}\\
&\texttt{constant}(p) = \{\texttt{+}, \texttt{-}\}\\
&\texttt{ruleset}(p) = \{r\}\\
&\texttt{input}(r) = c\\
&\texttt{blocks}(r) = \langle b\rangle\\
&\texttt{input}(b) = c\\
&\texttt{mpattern}(b) = \texttt{(xs + ys)}\\
&\texttt{rpattern}(b) = \texttt{(xs - ys)}
\end{aligned}$$

This is the same program as in the Example 13. So, given a ρ_s program:

```
(eval (g (+ -)
         ((g (xs + ys) (xs - ys))))
  (1 + 2 + 3 + 4 + 5))
```

its final value is:

```
(eval (g (+ -)
         ((g (xs + ys) (xs - ys))))
  (1 - 2 - 3 - 4 - 5))
```

Example 15.
Given a ρ_s program:

```
(eval (g ()
         ((g (x y z) (y))
          (x x x)
          (z z z)))
      (1 2 3 4 5 6))
```

its evaluation is fully deterministic, with intermediate values being (in order):

```
(eval (g ()
         ((g (x y z) (y))
          (x x x)
          (z z z)))
      (2 3 4 5))
```

```
(eval (g ()
         ((g (x y z) (y))
          (x x x)
          (z z z)))
      (3 4))
```

```
(eval (g ()
         ((g (x y z) (y))
          (x x x)
          (z z z)))
      ())
```

Example 16.

In this example we demonstrate horizontal composition of rewrite rules. Recall that every redex in ρ_s stands for a single rewrite rule. However, by composing several redexes we can achieve the same result as if there was a single redex encoding multiple rewrite rules.

The following ρ_s program:

```
(let ((body (x))
      (const (x 0 1)))
  (eval (g const
           ((g (x) (0))))
        body)
  (eval (g const
           ((g (x) (1))))
        body))
```

emulates that of the Example 12. Its values are:

```
(let ((body (0))
      (const (x 0 1)))
  (eval (g const
           ((g (x) (0))))
        body)
  (eval (g const
           ((g (x) (1))))
        body))
```

```
(let ((body (1))
      (const (x 0 1)))
  (eval (g const
           ((g (x) (0))))
        body)
  (eval (g const
           ((g (x) (1))))
        body))
```

Example 17.

Vertical composition is another type of placement for redexes where one is on top of the other. Our evaluation order dictates that in such situation, the bottom rule should be tried first, and only if it fails, then the top one should be tried.

The following ρ_s program:

```
(let ((const (0 1 / undefined)))
  (eval (g const
           ((g (x / x) (1))))
        (eval (g const
                 ((g (x / 0) (undefined))))
              (3 + (5 / 5)))))
```

evaluates to:

```
(let ((const (0 1 / undefined)))
  (eval (g const
           ((g (x / x) (1))))
        (eval (g const
                 ((g (x / 0) (undefined))))
              (3 + (1)))))
```

Example 18.

Let us demonstrate how non-local rewriting works. In the following example, `switch` will be a subgraph that is rewritten from `(off)` to `(on)`. What is important, is that `switch` is not present in the body of the eval block. In fact, the body is ignored completely, and only the switch is modified.

Given a ρ_s program:

```
(let ((switch (off)))
  (eval (g (off on)
           ((switch (off) (on))))
        body))
```

its final value is:

```
(let ((switch (on)))
  (eval (g (off on)
           ((switch (off) (on))))
        body))
```

Example 19.

In the Context section we have stated that conditionals based on formulas can be expressed. The following example demonstrates what we meant.

In this example we encode the relation of addition of numbers up to 2 and use it as a conditional expression.

The following ρ_s program

```
(let ((body (1 + 1 = 2))
      (const (0 1 2 3 4 + = true false))
      (R-addition
       ((0 0 0)
        (0 1 1)
        (0 2 2)
        (1 0 1)
        (1 1 2)
        (1 2 3)
        (2 0 2)
        (2 1 3)
        (2 2 4))))
  (eval (g const
           ((g (x + y = z) (false))
            (x x x) (y y y) (z z z)))
        (eval (let ((tuple (x y z)))
                (g const
                   ((g (x + y = z) (true))
                    (x x x) (y y y) (z z z)
                    (R-addition (xs tuple ys)
                       R-addition)
                    (tuple tuple tuple))))
              body)))
```

reduces `body` to `(true)`. If `body` would initially be `(1 + 1 = 3)`, then it would be reduced to `(false)`.

Since rules in ρ_s can rewrite other rules, relations like `R-addition` can be constructed during evaluation. Furthermore, we can treat some variables as free. For example, we can make `z` to be treated as a free variable by changing the rule

```
(let ((tuple (x y z)))
  (g const
     ((g (x + y = z) (true))
      (x x x) (y y y) (z z z)
      (R-addition (xs tuple ys) R-addition)
      (tuple tuple tuple))))
```

into

```
(let ((tuple (x y r)))
  (g const
     ((g (x + y = z) (x + y = r))
      (x x x) (y y y) (z z z)
      (R-addition (xs tuple ys) R-addition)
      (tuple tuple tuple))))
```

which rewrites `body` = `(1 + 1 = ?)` to `(1 + 1 = 2)`.

Many more examples can be found (and run) in the implementation repository[51] under `example/` directory. For instance:

- `example/loop-pattern.scm` is a program that uses a cyclic match pattern.
- `example/addition-fork.scm` is a program that adds natural numbers.
- `example/lambda.scm` is a program that interprets lambda calculus expressions.

Properties

One of the most obvious things with ρ_s is that it is non-confluent. We show that even a one-rule, one-step static-ρ_s program can be non-confluent.

Proof. Example of a one-rule non-confluent static-ρ_s program is one that consists of the block $\langle a,$ `(x + y)` , `(x)` $\rangle$ where `+` is a constant. This program rewrites `(1 + 2 + 3)` into `(1)` and `(1 + 2)`. $\square$

In the following lemma we prove that unreachable nodes remain unreachable forever. First we define connected components of a graph and prove that it cannot be that after a step of evaluation different components merge.

Connected component

A graph g is connected if for all its vertices $a, b \in \mathbf{V}(g)$, either a is reachable from b, or b is reachable from a.
A connected component of a graph h is a subset of h that is a connected graph and is not a subset of any other connected component.

Lemma 1. Let g be a graph and $h_1 \neq h_2$ be its connected components. Then for every g', a such that $g \odot a \underset{\emptyset}{\rightarrow} g'$ there do not exist connected components of g' that share vertices with both h_1 and h_2.

Proof. Assume that the statement is false. This means that there must be vertices $b_1 \in h_1, b_2 \in h_2$ that are adjacent in g' but not in g since h_1 and h_2 do not share any vertices (given $h_1 \neq h_2$). By examining definition of reduction relation we see that new edges are only created in `replace`. Specifically, these edges are:

- (CASE A) edges connecting the focused root to the children of the root of graph g_s[5],
- (CASE B) edges connecting vertices of graph g_s.

[5]We refer to the graph used in definition of `replace`, page 17.

One of those edges must be between b_1 and b_2.

Let us examine where vertices of g_s come from. The graph is created by applying homomorphism function to the replace pattern. Thus, its vertices are either from the relative blocks (specifically, from their match patterns, or from their input arguments), or they are fresh. All vertices of the encoded blocks come either from h_1 or from h_2 or from an another connected component, call it h_3. This is because every environment is a connected graph (see Parsing, page 24). If they come from h_3, then:

- (CASE A) Note that the focused root is also in h_3 because it is an element of the body of the current redex (see rule `R-REWR`, page 23), and the body is connected with the environment (see Parsing, page 24). This way, both b_1 and b_1 are from h_3.
- (CASE B) In this case, new edges connect vertices of h_3 and fresh nodes. Since no fresh node comes from h_1 or h_2(see creation of fresh nodes, page 20), this case also fails.

In cases where vertices of the encoded blocks come from h_1 or from h_2, the choice is symmetric. Assume that they come from h_1, then:

- (CASE A) the root is also in h_1, so both b_1 and b_1 are from h_1.
- (CASE B) new edges connect vertices of h_1 and fresh nodes. Since no fresh node comes from h_2(see creation of fresh nodes, page 20), this case also fails.

We have tried every case and each time arrived at a contradiction, thus the assumption must be false.

$\square$

The lemma allows us to focus on a single connected component, that of **root**, and ignore those subgraphs that disconnect during evaluation, because once they have disconnected, they will never connect to **root** again. This implies that implementations of ρ_s can collect garbage during evaluation, without the need to wait until the whole process finishes, as it would seem to be required by the definition of the final semantics of ρ_s.

In the following lemma we prove that, in particular, redexes only modify themselves.

Lemma 2. Let g, g' be graph and a be a vertex such that $g \odot a \underset{\emptyset}{\to} g'$. If $g(b) \neq g'(b)$ then b and all $c \in g'(b)$ are either not in g, or reachable from a child of a redex reachable from a in g.

Proof. Definition of relation $g \odot a \underset{\emptyset}{\to} g'$ has two cases – `R-EVAL` and `R-REWR`. The case of `R-REWR` does not apply to empty environments. In case of `R-EVAL`, the definition recurses on a redex g_r (reachable from a) with a new environment

h. Apply definition of $g \odot a \underset{h}{\rightarrow} g'$. In case of `R-EVAL`, we now just recurse on a smaller subgraph, so inductive hypothesis can be used. In case of `R-REWR`, we have the parsing and then running of a static-ρ_s program. Parsing phase assures that our environment graph is connected and reachable from a child of g_{r} (FACT A). By examining semantics of evaluation of static programs, we comfirm again that modifications to the original graph all happen in function `replace` (see definition of `replace`, page 17). Furthermore, what is rewritten is either a block input argument, or a focused vertex, or a fresh node, previously not present in g. By FACT A we know that input arguments are reachable from children of g_{r}. By definition of `R-REWR`, focused vertex is reachable from children of g_{r}. By definition of fresh nodes, they are not present in g (see creation of fresh nodes, page 20). □

Therefore, not only the unreachable subgraphs cannot be rewritten, but roots as well.

Corollary 1. If g, g' are such graphs that $g \to g'$, then $\mathbf{R}(g) \subseteq \mathbf{R}(g')$.

Proof. Assume that statement is true. Then there must be a root vertex a in g that is not a root of g'. From the definition of a root, there must be c such that a is its child, i.e. $a \in g'(c)$. But by Lemma 2 this means that a is reachable from children of a redex of g, contradiction. □

When can a subgraph be substituted with its value? One generic case is when it does not share its body with anyone. To state this formally, we introduce a concept of a context.

Graceful context

Fix a graph h. A graceful context is a function $f_h : \mathbb{G} \rightharpoonup \mathbb{G}$ that for every g returns $g \cup h$ if $\mathbf{V}(g) \cap \mathbf{V}(h) = \{b\} = \mathbf{R}(g)$. Graph h is referred to as "context constant" and graphs g are referred to as "context arguments".

A graceful context does not access anything but the root of its argument.

Theorem 1. If g is a graph, then for every k, if there is a unique g' in $g \xrightarrow{(k)} g'$, then for every graceful context f_h defined on g, g' and for every graph g'' we have

$$\langle f_h(g), g'' \rangle \in \rho_s \iff \langle f_h(g'), g'' \rangle \in \rho_s$$

Proof. Consider three cases.

1. If $h = \emptyset$, then $f_h(g) = g$ and $f_h(g') = g'$, thus, by definition of ρ_s, these graphs evaluate to the same thing.

2. If $\mathbf{V}(h) \cap \mathbf{V}(g) = \emptyset$, then this implies $\mathbf{V}(h) \cap \mathbf{V}(g') = \emptyset$ because no program can rewrite its own root (by Corollary 1). Then by Lemma 2, h has no impact on g, g', and at least one of the graphs will dissapear at the end

(because it is not reachable from **root**). If h dissapears, then go to the first case. If g, g' dissapear, then the result is h.

3. If $\mathbf{V}(h) \cap \mathbf{V}(g) = \{b\}$,

   then there are two groups of redexes – those from which there is a path to the context argument (GROUP A), and those from which there is not (GROUP B). Focusing on the latter, note that reductions in this group change only the context constant, and independently of the context argument. In other words, for every possible reduction of these redexes taking $f_h(g)$ to $f_{h'}(g)$ there is a mirroring reduction taking $f_h(g')$ to $f_{h'}(g')$. By repeating this logical step, there are two possible outcomes – either reductions in this group will never terminate, or they will stop at at particular point. In the first case, this satisfies our definitions, since both sides would not terminate. In the second case, we can apply inductive hypothesis, since only the context has changed and it has simplified. Thus, assume that context constants do not have redexes of GROUP B that are able to reduce further. Then, observe that between g and g' there are no intermediate normal forms. This implies that GROUP B of context redexes cannot reduce until g reduces to g'. The reason is that our evaluation strategy ensures that the bottom-most redex is reduced first. This concludes the proof.

□

This property is particularly useful for implementations of ρ_s. If interpreters can detect if a redex body is shared, they can compile the redex and apply it many times at once.

For more general contexts, the issue becomes that their redexes can start modifying the shared body with unpredictable consequences. In that case, the only thing that we can hope to assert is that the value that we would like to substitute will be in the set of the final results, wraped in its context, of course.

Lemma 3. If graph g is in normal form, then its every closure h also is.

Proof. Note that every redex of g is a closure of g. Also, a closure starting from a of a closure of g is either $g(a)^*$ or $\emptyset$, which follows from its definition. This implies that the set of redexes of h is a subset of redexes of g – those closures equal to $\emptyset$ stop being redexes. But this means that none of the redexes of h are active, since none of them were active in g. Thus, h is in normal form, having no active redexes. □

Graph calculi

In this section we introduce a general model for graph-based languages, of which ρ_s-calculus is a special case. We then adopt a definition of expressiveness of tree-based languages to graph-based languages, with the primary goal of using it later for ρ_s.

In programming terms, calculus is just the semantics. It is to be interpreted as such a relation that joins input terms with fully evaluated terms; it is the "evaluation function" of a graph-based language.

Graph calculus

Relation $\mathcal{C} : \mathbb{G} \times \mathbb{G}$ is a graph calculus if $\underset{g \in \mathtt{im}(\mathcal{C})}{\forall} \{g\} = \{h : \mathcal{C}(g, h)\}$.

The set of all graph calculi is $\mathbb{C}$.

Since every evaluated graph can only be further evaluated to itself, the relation restriction is placed. But in order to support non-deterministic languages, we do not really insist on the "evaluation function" to be a function. Our definition does not rely on a separate value domain because our values are graphs, just like the inputs.

Example 20.

Of course, ρ_s is a graph calculus. The only condition of the definition is satisfied by the fact that ρ_s is defined by a transitive *reflexive* closure, so normal forms evaluate to normal forms.

Example 21.

To illustrate a very basic example of a graph calculus, we define a language of arithmetic expressions, named "`gexp`". It can only parse zeroes and ones as numbers. The semantics can be ilustrated by a single example: for the graph $((\mathbf{1} + \mathbf{1}) \times (\mathbf{1} + (\mathbf{1} + ((\mathbf{0} - \mathbf{1}) + \mathbf{1}))))$, `gexp` returns the graph $\mathbf{4}$.

To describe it formally, we need to define two functions: one for the calculations, and one for the encoding of natural numbers. Calculations can be defined by the function `calc`:

$$\begin{aligned}
&\mathtt{calc} : \mathcal{U}^* \rightharpoonup \mathbb{G} \rightharpoonup \mathbb{Q} \\
&\mathtt{calc}[\![\langle a, +, b\rangle]\!](g) = \mathtt{calc}[\![\langle a\rangle]\!] + \mathtt{calc}[\![\langle b\rangle]\!] \\
&\mathtt{calc}[\![\langle a, -, b\rangle]\!](g) = \mathtt{calc}[\![\langle a\rangle]\!] - \mathtt{calc}[\![\langle b\rangle]\!] \\
&\mathtt{calc}[\![\langle a, \times, b\rangle]\!](g) = \mathtt{calc}[\![\langle a\rangle]\!] \times \mathtt{calc}[\![\langle b\rangle]\!] \\
&\mathtt{calc}[\![\langle a, \div, b\rangle]\!](g) = \mathtt{calc}[\![\langle a\rangle]\!] \div \mathtt{calc}[\![\langle b\rangle]\!] \\
&\mathtt{calc}[\![\langle \mathbf{0}\rangle]\!](g) = 0 \\
&\mathtt{calc}[\![\langle \mathbf{1}\rangle]\!](g) = 1 \\
&\mathtt{calc}[\![\langle a\rangle]\!](g) = \mathtt{calc}[\![g(a)]\!]
\end{aligned}$$

where nodes $\boldsymbol{+}, \boldsymbol{-}, \boldsymbol{\times}, \boldsymbol{\div}, \mathbf{0}, \mathbf{1}$ are some fixed nodes, exact value of which depends on $\mathcal{U}$.

The function that encodes numbers to graphs is:

$$\mathtt{num}(n) = \{\langle f(n), \langle\rangle\rangle\}$$

where f is an arbitrary injective function that maps natural numbers to vertices. This function is also fixed for $\mathcal{U}$, and we call it the encoder of `gexp`.

Then using these functions, the actual semantics is:

$$\begin{aligned}\langle g, g' \rangle \in \texttt{gexp} \iff & \quad g' = \texttt{num}(\texttt{calc}[\![a]\!](g)) \land \{a\} = \mathbf{R}(g) \\ \lor & \quad g' = \texttt{num}(\texttt{num}^{-1}(g))\end{aligned}$$

Modularity

Languages are often understood in terms of their "features". But what exactly those "features" are, is hard to pin down. In our case, we can be most abstract and say that a feature is something that has a corresponding set of languages that implement it. Those that do are the `good` languages (with respect to that feature).

Language feature

Given a set $\mathcal{F}$, if α is a member of $\mathcal{F}$, then α is a feature and $\texttt{good}(\alpha)$ is a set of calculi.

It is true that there are infinitely many ways to define a calculus by the set of its features, and with one set no different than the other, it makes decomposition impossible. To combat this issue, we will narrow our scope to "named" features.

First, we need to define isomorphisms.

Ordered graph isomorphism

Given two graphs g and h, the isomorphism between them is a bijection $\psi : \mathcal{U} \rightharpoonup \mathcal{U}$ mapping vertices of g to h:

$$\bigforall_{x} g(x) = \langle y_1, y_2, ..., y_n \rangle \iff h(\psi(x)) = \langle \psi(y_1), \psi(y_2), ..., \psi(y_n) \rangle$$

We write $\psi[\![g]\!] = h$ for short. Graphs g and h are isomorphic, which is written as $g \cong h$.

Now we want to express the intuition of constants, or "special symbols". Then a feature will be identifiable by a fixed set of constants. Special nodes are best understood by their complement – regular nodes. A regular node is a node that can be replaced by any other node, on every occaision, except by the special nodes.

In the following definition, the notation $[x]_R$ denotes the equivalence class of x in R, i.e. $[x]_R = \{y : R(x, y)\}$.

Special node

Every calculus $\mathcal{C}$ has a set of equivalent nodes:

$$\frac{\displaystyle\forall_{g,h}\ \mathcal{C}(g,h) \iff \mathcal{C}(\psi[\![g]\!],\psi[\![h]\!]) \qquad \psi(a)=b}{a \cong_{\mathcal{C}} b}$$

With that, consider the following system of equations:

$$\begin{aligned}\texttt{regular}_{\mathcal{C}} &= \{a : [a]_{\cong_{\mathcal{C}}} = \mathcal{U} \setminus \texttt{special}_{\mathcal{C}}\}\\ \texttt{special}_{\mathcal{C}} &= \texttt{dom}(\cong_{\mathcal{C}}) \setminus \texttt{regular}_{\mathcal{C}}\end{aligned}$$

The set of regular nodes of $\mathcal{C}$ is the biggest set $\texttt{regular}_{\mathcal{C}}$ for which the equations hold, or the empty set if there is no biggest one. Then the set of special nodes, $\texttt{special}_{\mathcal{C}}$, is just the complement of $\texttt{regular}_{\mathcal{C}}$. The index "$\mathcal{C}$" in $\texttt{special}_{\mathcal{C}}$ can be omitted if it is unambiguous.

Note that if $\texttt{dom}(\cong_{\mathcal{C}}) = \mathcal{U}$, then the set of regular nodes of $\mathcal{C}$ is just the biggest equivalence class of $\cong_{\mathcal{C}}$, and for $\texttt{dom}(\cong_{\mathcal{C}}) = \mathcal{U}$ smaller than $\mathcal{U}$, the set of regular nodes is empty.

Example 22.
For ρ_s, the set of special nodes is just two nodes: $\texttt{special}_{\rho_s} = \{\textbf{eval}, \textbf{root}\}$.

Example 23.
For the language `gexp` with encoder f, the set of special nodes is infinite: $\texttt{special}_{\texttt{gexp}} = \{+, -, \times, \div, \mathbf{0}, \mathbf{1}\} \cup \texttt{im}(f)$.

Equality

If we know which nodes are special, then we can say which graphs will return the same values, regardless of the language semantics.

Graph equivalence

Two graphs are equivalent, in a given calculus, if there is an isomorphism between them that maps special nodes to themselves. Notation $g \cong_{\mathcal{C}} h$ stands for the equivalency between g and h in the context of $\mathcal{C} \in \mathbb{C}$.

It is easy to see that graph equivalence is an equivalence relation, and that equivalent graphs reduce to the "same value".

Expressiveness

We now say that a feature is redundant if, for every program that uses it, we can "disable" the feature and add a graph that implements it instead. To make sure that implementation is separate from the original program, we add not exactly the implementing graph, but a graph equivalent to it. However, the implementation can (and often must) share special nodes with the original program.

Redundancy

A feature α is redundant for calculus $\mathcal{C}$ if there exists a calculus $\mathcal{C}' \in$ `good`(α) and graph h for which we have

$$\forall_{g,g',h'} \; h' \cong_{\mathcal{C}'} h \wedge \mathbf{V}(g) \cap \mathbf{V}(h') \subseteq \texttt{special}_{\mathcal{C}'} \\ \implies \mathcal{C}'(g,g') \iff \mathcal{C}(g \cup h, g')$$

Graph h is said to be the implementation of α (for $\mathcal{C}$). Graphs g are referred to as "the original programs".

Example 24.
Let us introduce a feature for `gexp` and then show that it is redundant in that calculus.

For the feature definition it is enough to show a single example of a calculus that has the semantics that we are intereseted in. This example calculus we call `gexp`$_{+5}$ and define it exactly the same way as `gexp`, except that the function `calc` of `gexp`$_{+5}$ is extended by a single case:

$$\texttt{calc}[\![\langle\mathbf{5}\rangle]\!](g) = 5 \text{ if } g(\mathbf{5}) = \bot$$

So we are expressing the constant 5.

Note that for the original calculus the value of $(\mathbf{5} + \mathbf{5})$ was not defined (because it could only parse ones and zeroes), but for the new calculus it is defined and equal to $\mathbf{10}$.

Our target feature α is defined by `good`$(\alpha) = \{$`gexp`$_{+5}\}$.

We pick the graph $(\mathbf{1} + (\mathbf{1} + (\mathbf{1} + (\mathbf{1} + \mathbf{1}))))$, with its root equal to the special node $\mathbf{5}$, to be the implementation of α. This way, every time the root is mentioned in a `gexp` program, its children become the children of the implementation graph, and the original semantics simply ignores the root and recurses on $(\mathbf{1} + (\mathbf{1} + (\mathbf{1} + (\mathbf{1} + \mathbf{1}))))$.

The conditional "if $g(\mathbf{5}) = \bot$" is needed for the reason that when adding the implementation graph to the original program, only the implementation graph must define the children of $\mathbf{5}$, otherwise the program graph will not be a proper function.

Redundancy is a stronger notion[6] of expressiveness than the the usual notion adopted for tree-based languages [52]. This is because, by definition, the usual notion allows us to add a graph (that implements the feature) before applying the evaluation function, but also allows to add more than just constant graphs.

At the same time, our notion is, in a sense, more general. That is because languages that support macros can turn constant graphs into what the usual notion permits, and we can always consider languages that have macro expansion as a first phase, and then proceed normally.

Expressiveness of ρ_s

In this section we study the space of features that are redundant for ρ_s. For that reason, we group them into conceptual categories, such as "library features". Categories are defined as templates parameterized by functions that express the conceptual semantics. When such template is instantiated, it becomes a rule that defines step transition in the semantics of a new language. This rule then, together with `R-NORM`, `R-REWR` and `R-EVAL`, defines a new version of relation $g \odot a \underset{h}{\rightarrow} g'$, which in turn is used to define a new language, in exactly the same way as ρ_s is defined by `R-NORM`, `R-REWR` and `R-EVAL`. The newly created language is then said to have the feature that we intended to express.

Library features

It turnes out that in order to express most practically useful features, one cannot simply follow the pattern "feature = subtree" when specifying features (the way it was possible in case of `gexp`) because it will not work. First of all, ρ_s only evaluates closed graphs. Second, even if we could evaluate open graphs, an implementation that is placed under a special node would be subject to rewriting by the rules that lie above it. Thus, one should instead put the implementation "on top of" the special node that defines the feature. To prevent the original program the from modifying the implementation code, we can chose implementations that only share a single additional[7] special node, children of which are not defined in the implementation.

We will also assume that there only is a single "input argument" in the feature application. In the following rule, a is the special node that names the library feature, and b is the input argument:

$$\frac{g(a) = \langle b \rangle}{\texttt{apply}\ a\ \texttt{to}\ b\ \texttt{in}\ g}$$

Based on that, let us define a template:

[6]In the sense that there are fewer redundant features than there are expressible.

[7]The nodes **eval**, **root** are the existing special nodes and the node that "names" the feature is the additional one.

$$\frac{g \odot a \Downarrow \quad g' \in f(g(b)^*) \quad \texttt{apply}\ a\ \texttt{to}\ b\ \texttt{in}\ g}{g \odot a \xrightarrow[h]{} g[b := g']}\ (\texttt{R-ALIB}_{(a,f)})$$

This template defines rule `R-ALIB` that is parameterized by the special node $a : \mathcal{U}$ and by $f : \mathbb{G} \to 2^{\mathbb{G}}$. Let us now formally define the family of features generated by the template:

Atomic library feature

Fix a vertex a and a function f, and let the calculus $\mathcal{C}$ be defined by the rules `R-NORM`, `R-REWR`, `R-EVAL` and $\texttt{R-ALIB}_{(a,f)}$. Then every feature α with $\mathcal{C} \in \texttt{good}(\alpha)$ is an atomic library feature defined by f and a.

Note that the function f is stateless and works on a local (focused) part of the graph, which is in normal form. While the normal form requirement is obvious – redexes do no activate unless their body is in normal form – the other restrictions do limit the space of features expressible in ρ_s. The reason is that redexes in ρ_s can modify not only the currently focused subgraph (i.e. it is non-local), and that each application of a feature implemented with **eval** can result in a different subgraph (i.e. **eval** is stateful). On the other hand, we can see that `R-ALIB` defines features that work in a single step of evaluation. This is a very strict requirement, which means that, conceptually, every such feature must be evaluated atomically. Some features surely can be expressed atomically, for example those that require only a single rewrite rule application to implement, because single rewrites indeed happen atomically.

One particularly useful family of atomic features that are expressible in ρ_s are "atomic compare-and-swap" features. The intuition here is that such a feature compares the input graph to the fixed graph g, and if they are equal, rewrites the input to the fixed output graph h, and all of this happens in one step of the evaluation:

Atomic compare-and-swap feature

Fix graphs $g \neq \emptyset$ and h that are in normal form. An atomic library feature defined by function $f(x) = \begin{cases} \{h\}, & \text{if } x = g \\ \emptyset, & \text{otherwise} \end{cases}$, is an atomic compare-and-swap feature defined by g, h and a.

Lemma 4. Every atomic compare-and-swap feature is redundant for ρ_s.

Proof. Let g, h and a define the target compare-and-swap feature.

The approach that we take is to build a rewrite rule that matches exactly and only the graph g, and rewrites the vertices in the input graph to point to newly created lists of vertices.

First, assume that all vertices of g and h are constants (in the sense that rules treat them as such). Note that these vertices are special in the target language, so we can use them in our rewrite rules without the fear of them getting replaced (only regular nodes get replaced).

Then, inductively construct a rewrite rule according to the following scheme: for each vertex e such that $g(e) = \langle b_1, \ldots, b_n \rangle$, add this rewrite block:

$$\langle e, \quad \texttt{(}b_1 \ldots b_n\texttt{)}\ , \quad e\ \rangle$$

The added blocks perform the comparison, without rewriting anything yet. If $n > 0$ then each block checks that its argument e has exactly n children, which are b_i.

Then, to rewrite the graph, for each vertex c such that $h(c) = \langle d_1, \ldots, d_n \rangle$, add this rewrite block:

$$\langle c, \quad c\ , \quad \texttt{(}d_1 \ldots d_n\texttt{)}\ \rangle$$

Let r be one of the roots of g, then add these rewrite blocks at the end of the rule:

$$\begin{array}{lll} \langle a, & \texttt{(b)}\ , & \texttt{()}\ \rangle \\ \langle b, & \texttt{r}\ , & \texttt{b}\ \rangle \end{array}$$

Their purpose is to check that the root of the input graph `b` is one of the roots of g, and then to disconnect the input argument from the implementation once the graph has been rewritten.

If we assume that g does not have any leaves (that is nodes a for which $g(a) = \langle\rangle$), then this single rule we just constructed is enough. Leaves, however, are not checked by the constructed rule because patterns with no children accept every input graph. In order to check the leaves we must use vertical composition.

For each leaf $e \in g$, construct an additional rewrite rule with just four rewrite blocks:

$$\begin{array}{lll} \langle e, & \texttt{(x xs)}\ , & e\ \rangle \\ \langle \texttt{x}, & \texttt{x}\ , & \texttt{x}\ \rangle \\ \langle a, & \texttt{(b)}\ , & \texttt{()}\ \rangle \\ \langle \texttt{b}, & \texttt{b}\ , & \texttt{b}\ \rangle \end{array}$$

Such groups of blocks matche only if e is not a leaf. Thus, if we put these rules such that they are in the body of the first rule, we prevent activation of the first rule in case any of the nodes supposed to be leaves are not leaves.

Every rule created by these blocks only matches if the input argument has been given to the implementation, and if they succeed in the match, they disconnect the argument as well.

□

Not all features need to be atomic. We add another assumption to our template in order to capture the regular, nonatomic features:

$$\frac{g \odot a \Downarrow \quad g' \in f(g(b)^*) \quad \texttt{apply } a \texttt{ to } b \texttt{ in } g \quad R(g_-, g, g_+)}{g_- \odot a \xrightarrow[h]{} g_+[b := g']} \; (\texttt{R-LIB}_{(a,f,R)})$$

The rule R-LIB is now parameterized by $a : \mathcal{U}$, by $f : \mathbb{G} \to 2^{\mathbb{G}}$, and by $R : \mathbb{G} \times \mathbb{G} \times \mathbb{G}$.

Additionally, R must be such that:

$$R(g_-, g, g_+) \implies g_- \xrightarrow{*} g \xrightarrow{*} g_+$$

Basically, by adding R we allow the input and the output graphs to be separated by an arbitrary number of steps.

Now we can define the regular library feature.

Library feature

Fix a vertex a and a function f, and for each relation R let calculi $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \ldots$ be defined by the rules R-NORM, R-REWR, R-EVAL and $\texttt{R-LIB}_{(a,f,R)}$. Every feature α with $\underset{i}{\forall}\ \mathcal{C}_i \in \texttt{good}(\alpha)$ is a library feature defined by f and a.

Computable library features

So far we have not mentioned computability of parameterized functions, but it matters. If we limit ourselves to finite graphs, then by Church-Turing thesis [53], it is not possible to express functions in ρ_s that are not computable. But library functions act on subgraphs, not on numbers or strings, so the usual notion of computability does not apply to them directly. So as it is usually done in practice, we need to first encode graphs as strings, and then judge whether the function on econded graphs is computable:

Computable graph function

A function $f : \mathbb{G} \to \mathbb{G}$ is computable when there exists a computable function $\psi : \Sigma^* \to \Sigma^*$ (for some fixed alphabet Σ), and a bijection $\phi : \mathbb{G} \to \Sigma^*$, such that $\underset{g,h}{\forall}\ f(g) = h \iff \psi(\phi(g)) = \phi(h)$.

Computable library feature

A library feature defined by a computable function is a computable library feature.

Theorem 2. Every computable library feature is redundant for ρ_s.

Proof. The outline of the proof is as follows:

1. design a low-level language capable of arbitrary graph manipulations,
2. embed this language into ρ_s,
3. design a higher-level language that is more convenient for large programs,
4. write graph encoding and decoding algorithms in this language,
5. translate this language to the first one in order to show that encoding is possible in ρ_s,
6. implement a Turing Machine emulator in ρ_s that works on encoded graphs,
7. ensure that atomicity is preserved.

Step 1. We will design a very simple stack-based language, "`gstack`", with the notable properties that there is no addition/subtraction/zero test, but instead cons/car/cdr and null test builtins (since we are dealing with graphs), and no builtin looping construct (since the source code but can contain many natural loops through itself).

The syntax of `gstack` is:

$$\begin{array}{llcl}
\textbf{Command} & c & ::= & \texttt{null} \mid \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{set} \\
 & & \mid & (\texttt{push}\ v) \mid (\texttt{pop}\ v) \\
 & & \mid & (\texttt{if null?}\ c\ c) \mid (\texttt{if eq?}\ c\ c) \mid (\texttt{and}\ c\ c) \\
\textbf{Variable} & v & ::= & v_1 \mid v_2 \mid v_3 \mid \ldots \underset{v_i \in \mathcal{U}}{\forall}
\end{array}$$

A program in `gstack` is just a Command. As for the semantics, it is easier to define it in ρ_s directly. One variable among all the vs is chosen to be "an input", and one other to be "an output".

Step 2. We are going to write a ρ_s program that has a `gstack` program encoded in the variable `start`. Here is the first part:

```
(let ((const (null cons car cdr set
               push pop
               if null? eq? and
               do stack))
      (stack ())
      (do (start)))
```

It defines constant nodes, which mostly coincide with the constants in `gstack` syntax definitions, but also contain `do` and `stack`. The first one, `do`, is a node that serves as a pointer to the currently running instruction. At the begining it points to `start`. The second one, `stack`, is a node whose children are the stack content. Initialy the stack is empty, so `stack` is child-free.

What follows after these definitions are the rules for interpreting every instruction (we will pause after the first 5):

```
  (eval (g const
           ((do (command) (rest))
            (command (and null rest) command)
            (stack (ss) (() ss))))
        body)

  (eval (g const
           ((do (command) (rest))
            (command (and cons rest) command)
            (stack (s1 s2 ss) ((s1 ys) ss))
            (s1 s1 s1)
            (s2 (ys) s2)))
        body)

  (eval (g const
           ((do (command) (rest))
            (command (and car rest) command)
            (stack (s1 ss) (x ss))
            (s1 (x xs) s1)
            (x x x)))
        body)

  (eval (g const
           ((do (command) (rest))
            (command (and cdr rest) command)
            (stack (s1 ss) ((xs) ss))
            (s1 (x xs) s1)
            (x x x)))
        body)

  (eval (g const
           ((do (command) (rest))
            (command (and set rest) command)
            (stack (s1 s2 ss) (s1 ss))
            (s1 (xs) (ys))
            (s2 (ys) s2)))
        body)
```

The above rules manipulate the stack, and then move the code pointer `do` to the next instruction. More precisely on each instruction:

- `null` pushes a fresh node with no children to the stack.
- `cons` pops two items from the stack, then makes a new "pair" node with children equal to the first node and the second node's children.
- `car` pops a node from the stack and pushes its first child back on the stack.

- `cdr` pops a node from the stack and pushes a new node with children equal to that of the popped node, except for the first one, back on the stack.

- `set` pops two nodes from the stack, sets the first one's children as the second one's, and then pushes the fist node back on the stack.

Every instruction so far, except for the `set`, is similar to those of LISP[54], but they are not excatly the same. For instance, `car` does return the first element of `cons`, but `cdr` does not return the second one – it returns a new node that has all of the children of the second node. This matters if we have "strong equality" (the `eq?`) in the language.

The `set` instruction has no equivalent in many LISP variants – it is neither `setcar`, nor `set-cdr!`, because it modifies the `pair` as if it was a vector.

On the implementation side, there are already two notable properties. First, every rule ignores the main input node that they all bind to `g`. This is because we only need to rewrite the pointer position and the stack which are constant nodes. Second, there are many blocks of the type `(`x x x`)`. The only purpose of those is to assert that x matches a single node, not zero or more.

Continuing with the rules:

```
(eval (g const
        ((do (command) (rest))
         (command (and push-expr rest) command)
         (push-expr (push v) push-expr)
         (v (x) v)
         (stack (ss) (x ss))
         (x x x)))
      body)

(eval (g const
        ((do (command) (rest))
         (command (and pop-expr rest) command)
         (pop-expr (pop v) pop-expr)
         (v (cs) (s1))
         (stack (s1 ss) (ss))
         (s1 s1 s1)))
      body)
```

These are rules for `push` and `pop` instructions. The `push` works as it classically does – simply pushes a constant to the stack. But the `pop` specifically receives an output variable, children of which it sets to the children of the node on the top of the stack.

Together, these instructions can express all of the instructions that a typical stack machine needs:

- `drop` is (pop a),

- `dup` is (and (pop a (and (push a) (push a)))),

- `swap` is `(and (pop `a` (and (pop `b`) (and (push `a`) (push `b`)))))`,

and so on... Assuming that a and b above are fresh nodes not used for anything else.

Finally, we have conditional execution rules:

```
(eval (g const
         ((do (command) (then))
          (command (if null? then else) command)
          (stack (s1 ss) (ss))
          (then then then)
          (else else else)
          (s1 s1 s1)))
      (eval (g const
               ((do (command) (else))
                (command (if null? then else) command)
                (stack (s1 ss) (ss))
                (s1 (x xs) s1)
                (then then then)
                (else else else)
                (x x x)))
            body))

(eval (g const
         ((do (command) (else))
          (command (if eq? then else) command)
          (stack (s1 s2 ss) (ss))
          (then then then)
          (else else else)
          (s1 s1 s1)
          (s2 s2 s2)))
      (eval (g const
               ((do (command) (then))
                (command (if eq? then else) command)
                (stack (s1 s1 ss) (ss))
                (s1 s1 s1)
                (then then then)
                (else else else)))
            body))
```

On the high level, the first two rules interpret the `if null?` instruction by poping the top of the stack, then checking if it is a node with no children, and then moving the code pointer to the appropriate branch. Similarly, `if eq?` is interpreted by popping two items from the stack, and checking if they are the same node.

Since there is no direct negation in ρ_s, in each pair of rules, one is placed "on top of" the other to prevent the top one from running unless the bottom one fails, emulating the proper negation.

There is no restriction on what the branches of `if` are, as long as they point to `gstack` code. So, in particular, we can emulate loops (and more generally GOTOs[55]) by pointing branches to previous instructions.

Note that evaluation is deterministic, even though ρ_s is not. This is because at each step there is at most one rewrite that the interpreter can perform.

Example 25.

Let us write a `gstack` program that reverses a list. For this we choose the input node to be `x`, the output node to be `r`, and the program under the `start` variable to be:

```
(let ((loop
       (progn
        (pop int)
        (push int)
        (if null?
            (return r)
          (progn
           (push r)
           (push int)
           car
           cons
           (pop r)
           (push int)
           cdr
           loop)))))
  (progn
   null
   (pop r)
   (push x)
   loop))
```

This program uses `progn` as syntactic sugar for `and`. Basically, expression of the form

$$\texttt{(progn } x_1\ x_2\ \texttt{...}\ \ x_{n-1}\ x_n\texttt{)}$$

translates to

$$\texttt{(and } x_1\ \texttt{(and } x_2\ \texttt{(and ...}\ \ \texttt{(and } x_{n-1}\ x_n\texttt{) ...)))}.$$

This program is located in `example/gstack.scm` in the implementation repository and can be run from there by the ρ_s interpreter. If the children of the node `x` are $\langle x_1, x_2, ..., x_n\rangle$, then, when the program finishes, the children of `r` become $\langle x_n, ..., x_2, x_1\rangle$.

Step 3. In this step we design a LISP-like language for implementation of graph encoding and decoding algorithms. We call this language `glisp`. The syntax of `glisp` is:

$$\begin{array}{llcl}
\textbf{Expression} & e & ::= & v \\
 & & | & (\texttt{null}) \mid (\texttt{cons}\ e\ e) \mid (\texttt{car}\ e) \mid (\texttt{cdr}\ e) \mid (\texttt{set}\ v\ e) \\
 & & | & (\texttt{if}\ (\texttt{null?}\ e)\ e\ e) \mid (\texttt{if}\ (\texttt{eq?}\ e\ e)\ e\ e) \mid (\texttt{progn}\ e\ \ldots\ e) \\
 & & | & (\texttt{define}\ v\ e) \\
 & & | & (\texttt{lambda}\ (v\ \ldots\ v)\ e\ \ldots\ e) \\
\textbf{Variable} & v & ::= & v_1 \mid v_2 \mid v_3 \mid \ldots \ \underset{v_i \in \mathcal{U}}{\forall}
\end{array}$$

Essentially, `glisp` extends `gstack` by `lambda` and `define` in the most straightforward way.

The precise semantics of `define` and `lambda` are exactly the same as in Scheme [56]. In fact, the language itself can be expressed in terms of Scheme macros, as we have done in our implementation repository in `test/glisp/builtins.scm`.

What is important is that there are only graphs in `glisp`. All the builtins that agree in name with `gstack` (such as `null`, `cons`, `cdr`, ...), have identical semantics to that of `gstack`, except that they do not operate on the stack, but on input arguments and return values (the way functions created by `lambda` form do).

The details are not important – `glisp` is, basically, a simplified Scheme.

Step 4. At this point we ought to describe the algorithms for encoding and decoding of graphs. First thing to note is that we are going to encode graphs as graphs. But the resulting graphs are going to be inputs to Turing Machines, so their structure is very restricted – they represent strings over a finite (size 3) alphabet. The resulting graphs are always of the following form:

$$(x_1\ x_2\ \ldots\ x_{n-1}\ x_n)$$

where $x_1, ..., x_n$ are elements of $\{\mathbf{o}, +, /\} \subset \mathcal{U}$.

The idea of the encoding algorithm is to:

1. number vertices,
2. construct an adjacency-list representation of the graph,
3. convert adjacency-list to be number-based.

Each vertex is going to get an assigned "number" first. These numbers are, of course, also graphs:

- $(\mathbf{o})$ is 0,
- $(+\ \mathbf{o})$ is 1,
- $(+\ +\ \mathbf{o})$ is 2,

- `(+ + + o)` is 3,
  ...

An adjacency-list is then constructed. Here are few examples:

- graph `(x y z)` has adjacency-list representation:

```
(let ((r (x y z)))
  ((r x y z)
   (x)
   (y)
   (z)))
```

- graph `((2 · x) + x)` has adjacency-list representation:

```
(let ((z (2 · x))
      (m (z + x)))
  ((m z + x)
   (z 2 · x)
   (2)
   (·)
   (x)
   (+)))
```

Finally, encoded graphs have indexes instead of the actual subgraphs in them, and have individual lists separated by /. For the graph `(x y z)`, the final number-based adjacency-list is the following graph:

```
(+ + o + + + o + + + + o + + + + + o /
 + + + o /
 + + + + o /
 + + + + + o /)
```

Encoding. To get indexes of vertices, we order them. The following function takes a graph and outputs all of its vertices in order:

```
(define order-nodes
  (lambda (graph)
    (define visited-list (null))

    (define add-to-visited
      (lambda (node)
        (set visited-list (cons node visited-list))))

    (define loop
      (lambda (g)
        (define consed (cons g g))
        (if (null? (in-children? visited-list g))
            (null)
            (if (eq? eval-node g)
                (progn
                 (foreach-child loop g))
                (progn
                 (add-to-visited g)
                 (foreach-child loop g))))))

    (loop graph)

    (cons
     eval-node
     (reverse-children visited-list))))
```

It always puts **eval** first, no matter if it was present in the graph. This way, it will always be encoded as 1 (indexes are 1-based), retaining its speciality compared to other nodes that get indexes related to their position in the graph.

We use helper functions, like `in-children?` and `foreach-child` in the implementation. They are trivial, so we move their definitions to Appendix A.

Having this function, we create an ordered list of vertices, and then once we need to get an index of a vertex, we look for its position on the list. This finishes the first step of the encoding.

Next, the creation of adjacency list:

```
(define graph->adjlist
  (lambda (g)
    (define return (null))
    (define visited-list (null))

    (define add-to-return
      (lambda (node)
        (set return (cons node return))))
    (define add-to-visited
      (lambda (node)
        (set visited-list (cons node visited-list))))

    (define loop
      (lambda (g)
        (define consed (cons g g))
        (if (null? (in-children? visited-list g))
            (null)
            (progn
             (add-to-return consed)
             (add-to-visited g)
             (foreach-child loop g)))))

    (loop g)

    (reverse-children return)))
```

The resulting adjacency-list is basically the same as in the previous examples. We then need to separate the lists by the specially chosen vertex, `separator` = /, and then flatten the list so that it is closer to our target string. For that, we have the functions `intersperse` and `flatten`.

In the last step, we replace vertices of adjacency-list by their indexes:

```
(define flat-adjlist->tape
  (lambda (ordered-nodes x)
    (if (null? x) x
        (progn
         (define first (car x))
         (define encoded-first
           (if (eq? first separator)
               (make-singleton separator)
               (index-of ordered-nodes first)))
         (concat
          encoded-first
          (flat-adjlist->tape ordered-nodes (cdr x))))
             )))
```

By putting it all together we get the encoding function:

```
(define encode-graph
  (lambda (g)
    (define ordered-nodes (order-nodes g))
    (define adjlist (graph->adjlist g))
    (define separated
      (intersperse (make-singleton separator) adjlist)
         )
    (define flat (flatten separated))
    (flat-adjlist->tape ordered-nodes flat)))
```

Decoding.

Large part of the decoding process is parsing:

```
(define read-number
  (lambda (tape)
    (define head (car tape))
    (set tape (cdr tape))

    (if (null? (n-zero? (make-singleton head)))
        (n-zero)
        (n-successor
         (read-number tape)))))

```

```
(define separator-next?
  (lambda (tape)
    (define head (car tape))
    (if (eq? head separator)
        true-node
        false-node)))

(define skip-separator
  (lambda (tape)
    (set tape (cdr tape))))
```

Apart from the input tape, the decoder function may receive an ordered list of vertices. This list would be the translation of numbers from the tape, to the actual vertices on the list. Recall that the orignal transformation passes encoded tape to a Turing Machine, then receives the result and decodes it to a graph. If on the resulting tape we see an index that was previously used for a vertex a, then that index must still represent a. But there may be more different vertices in the output than there were in the input. For that reason we may need to extend our ordered list with new, fresh nodes:

```
(define make-additional
  (lambda (ordered-nodes g)
    (define old-node-count (children-count
       ordered-nodes))
    (define new-node-count (count-separators g))
    (define fresh-count
      (n-successor (monus new-node-count
         old-node-count)))

    (make-n-fresh-nodes fresh-count)))

(define count-separators
  (lambda (g)
    (if (null? g) (n-zero)
        (if (eq? (car g) separator)
            (n-successor (count-separators (cdr g)))
            (count-separators (cdr g))))))
```

The decoding process itself comes down to recursively collecting lists of children and modifying existing ones:

```
(define decode-graph
  (lambda (ordered-nodes g)
    (define all-nodes
      (concat ordered-nodes
              (make-additional ordered-nodes g)))

    (define read-node
      (lambda ()
        (define n (read-number g))
        (define v (child-ref all-nodes n))
        v))

    (define read-list
      (lambda ()
        (loop (read-node) (null))))

    (define loop
      (lambda (current current-children)
        (if (null? (separator-next? g))
            (progn
             (skip-separator g)
             (set current (reverse-children current-children))
             (if (null? g) g (read-list)))
            (progn
             (define v (read-node))
             (loop current (cons v current-children))))
        current))

    (read-list)))
```

Step 5. On one hand we have a stack machine that works on graphs, and on the other, a LISP-like functional language that also works on graphs. In practice, a translation would be a trivial matter. However, it would be neither succinct, nor easily provable correct. For this reason, we do not attempt to give the full description of such a translation, and instead suggest to look into existing work on this topic.

For example, the work on a portable LISP compiler [57] describes in detail compilation of LISP-like languages into a generic register machine. Veriables of `gstack` can be used as register names, and the command `(and null (pop` v`))` can be seen as an allocation of fresh memory space into variable v, thus providing functionalities that register machines require.

A more generic approach is adopted in [58]. Here the focus is placed on the problem of compilation of `lambda` into primitive machine-level instructions.

Note that, in any case, the translation does not have to be implmenented in ρ_s. The `glisp` language is introduced purely for conveinence, while the original transformation needs to be implemented in `gstack` because that is the language that we interpret in ρ_s.

Step 6. Implementing a Turing Machine in terms of a rewrite system is the easiest step. We start by defining the list of constants that contain, among other things, the alphabet and the states of the turing machine, by defining a variable `state` that keeps the current state, another variable `action` that controls the execution, and variables `tape-left` and `head` that represent parts of the tape.

```
(let ((const (>> << HALT NEWSTATE MOVE _ 0 1 2 3 4 5 6
    7 8 9 q0 q1 q2 q3 q4 q5 q6 q7 q8 q9))
      (state (q0))
      (action (MOVE >>))
      (tape-left  (_ _ _ _))
      (head (_)))
```

There are three possible `actions`:

1. move the head to the right, encoded as `(MOVE >>)`,
2. move the head to the left, encoded as `(MOVE <<)`,
3. calculate a new state, encoded as `(NEWSTATE)`.

With that, there are only two rules needed to run the turing machine:

```
  (eval (g const
          ((action (MOVE >>) (NEWSTATE))
           (tape-left  (xs x) (xs x h))
           (head (h) (y))
           (tape-right (y ys) (ys _))
           (x x x)
           (y y y)))
        body)

  (eval (g const
          ((action (MOVE <<) (NEWSTATE))
           (tape-left  (xs x) (_ xs))
           (head (h) (x))
           (tape-right (y ys) (h y ys))
           (x x x)
           (y y y)))
        body)
```

Each rule deals with the shift to the right or to the left. The tape is represented by `tape-left`, `head` and `tape-right`, such that if children of these three variables are concatenated, then they represent the actual tape. The node `_` represents a blank symbol on the tape. Initially, `tape-left` and `head` contain only a single blank symbol. In the process of running the machine, new blank symbols are added, emulating the infinite aspect of the tape. On the other hand, `tape-right` is initialized with the input string.

The presented part is generic to every Turing Machine, but the transition function must be implemented separately. Here is an example of a transition relation that negates its input, i.e. transforms `(0 0 0 1 1 1 1)` into `(1 1 1 0 0 0 0)`:

```
  (eval (g const
          ((action (NEWSTATE) (MOVE >>))
           (head (0) (1))
           (state (q0) (q0))))
        body)

  (eval (g const
          ((action (NEWSTATE) (MOVE >>))
           (head (1) (0))
           (state (q0) (q1))))
        body)

  (eval (g const
          ((action (NEWSTATE) (MOVE >>))
           (head (1) (0))
           (state (q1) (q1))))
        body)

  (eval (g const
          ((action (NEWSTATE) (HALT))
           (head (_) (_))
           (state (q1) (HALT))))
        body))
```

This example is placed in `example/turing-machine.scm` and can be run from there by the ρ_s interpreter.

What is important is that because ρ_s is nondeterministic, the transition function can be a relation, in which case the program would emulate a nondeterministic Turing Machine.

Step 7. So far, our construction works under the assumption that no **eval** in the original program has access to our input argument. This is because our decoding algorithm constructs the graph step-by-step, thus a parallel **eval** could in that time observe graphs that are neither the original input, nor the final result, but something inbetween.

The solution to this issue is to adopt our algorithm from Lemma 4 that allows to compare and rewrite whole graphs in one step. Given a slight modification of that algorithm, we can get a builtin function `construct-cmp-and-swap` which returns a function that performs the actual compare-and-swap atomically, and singnals the success status. Then, once an input argument has been given, we make a copy that is isomorphic to it and record the isomorphism. We then encode that copy, transform it on the Turing Machine, and decode back to get the result that is isomorphic to the intended result. Finally, we pass the input argument (that could be modified by now), the result of the transformation, and the recorded isomorphism to `construct-cmp-and-swap` function, which in turn dynamically constructs the neccessary rules without the resulting graph (to which the input is swaped), but using the isomorphism that we have given to it. If compare operation detects changes in the input graph, we start from the begining – by making the copy of the new input, and continuing from there.

We know that such a `construct-cmp-and-swap` function that constructs the necessary rules based on isomorphism can exist in `glisp` because, as we have already seen, any computable function is constructible in `glisp`.

□

Conclusions and future work

Given that any computable library feature is expressible, we can safely implement any local features in the host interpreter of ρ_s, achieving much better performance. The space of these features allows for implementation of arbitrary evaluation orders, including nondeterministic ones, and for many practical programming features, such as macros.

It is, however, not clear what happens when we go beyond library features. One area is atomicity, where we have shown that some atomic library features are expressible. It is our conjecture that every atomic library feature is a regular library feature. But even a positive answer would not imply that ρ_s has no use for modeling concurrency, since nondeterministic communication between parallel redexes is still possible.

Going even further, stateful features must be studied. One easy way to bound the expressiveness of ρ_s is by a stateful function that, apart from the focused subgraph, modifies its own local state. We expect there to be a gradation between library features and fully stateful features.

Finally, our definition of expressiveness allows one to study composition of features, and based on that, to develop a bigger theory of features themselves.

On the implementation side, given the very dynamic nature of ρ_s, ideas for efficient, just-in-time, compilation of redexes is worth studying, for it may be transferrable to other languages.

Appendix

Appendix A Definitions of helper procedures for `glisp` code:

```
;; The values of unbound variables are themselves,
;; so the below lines is just aliasing:
(define eval-node eval)
(define separator /)
(define bit0 o)
(define bit1 +)

(define true-node (null))
(define false-node (cons true-node true-node))

```

```
(define make-singleton
  (lambda (x)
    (cons x (null))))

(define concat
  (lambda (left right)
    (if (null? left) right
        (cons (car left)
                (concat (cdr left) right)))))

(define copy-children
  (lambda (x)
    (cons (car x) (cdr x))))

(define flatten
  (lambda (x)
    (if (null? x) x
        (concat
         (car x)
         (flatten (cdr x))))))

(define intersperse
  (lambda (separator list)
    (if (null? list) list
        (cons
         (car list)
         (cons
          separator
          (intersperse
           separator (cdr list)))))))

(define foreach-child
  (lambda (func list)
    (if (null? list) list
        (progn
         (func (car list))
         (foreach-child func (cdr list))))))

;; Shallow comparison, only 1 level deep.
(define equal-children?
  (lambda (a b)
    (if (null? a)
        (if (null? b)
            true-node
            false-node)
        (if (null? b)
            false-node
            (if (eq? (car a) (car b))
                (equal-children?
                 (cdr a) (cdr b))
                false-node)))))

(define n-zero
  (lambda ()
    (make-singleton bit0)))

(define n-successor
  (lambda (n)
    (cons bit1 n)))

(define n-pred
  (lambda (n)
    (cdr n)))

(define n-zero?
  (lambda (n)
    (equal-children? n (n-zero))))

(define n-one
  (lambda ()
    (n-successor (n-zero))))

(define n-one?
  (lambda (n)
    (equal-children? n (n-one))))

;; Subtraction bounded by zero.
;; So that monus(5, 3) is 2, but monus(5, 10) is 0.
(define monus
  (lambda (a b)
    (if (null? (n-zero? b)) a
        (if (null? (n-zero? a))
            (n-zero)
            (monus (n-pred a) (n-pred b))))))

;; Starts from 1.
;; If element is not found, returns the length of the
   list + 1.
(define index-of
  (lambda (list element)
    (if (null? list)
        (n-one)
        (if (eq? element (car list))
            (n-one)
            (n-successor
             (index-of (cdr list)
                       element))))))

(define child-ref
  (lambda (list index)
    (if (null? list) list
        (if (null? (n-one? index))
            (car list)
            (child-ref (cdr list)
                       (n-pred index))))))

(define make-n-fresh-nodes
  (lambda (n)
    (if (null? (n-zero? n))
        (null)
        (cons (null)
                (make-n-fresh-nodes
                 (n-pred n))))))

(define in-children?
  (lambda (list item-node)
    (if (null? list)
        false-node
        (progn
         (define first (car list))
         (if (eq? first item-node)
             true-node
             (in-children?
              (cdr list) item-node))))))

(define reverse-children
  (lambda (x)
    (define loop
      (lambda (x buf)
        (if (null? x) buf
            (loop (cdr x)
                  (cons (car x) buf)))))
    (loop x (null))))

(define children-count
  (lambda (x)
    (if (null? x) (n-zero)
        (n-successor (children-count (cdr x))))))
```

Appendix B My public PGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGKsRb8BEADUcpaqwbYO6MdTBkSPl46LiO+KVp3a75YEKBOAXX28MIdzZdLy
J1iVlgfMRkdsmG1etDYc++UaiyN27PyMCHe07JSx2hjhEmXkQLkNxXDyAtppMbhq
uV/PiocSa8Op+eTOHv5D46fJLacZQ2PhIuMf0kfO0LSIwJC2gWFiJMxP7IXxhvmp
c7lCF5azKEwTocbAtEetXmumY5oeORfqP+jzS7UQfyEmzey9ks/8wCd8xxFAKF8c
0gDkSSmoNJS95q+7bCK4xJZyV82qlKC0rQ7btK3BIClzp9QZpifxCZXK1Xt8Xw9b
F6KPYo/tgyD/2N9sVaRzsJi3FN2sssp/X7s/gQWeiL+KfboQfHR2IruZl4hHmgUC
oqfJgVheCQurABTBRA0ggB7zs9mafwpVgHU3pRBNbBNrhFLDAanSvJ2jdVErhcvz
nCih5rpRDqZqGysVZOgJfhYJUxxh5/WGKAHmkmWB3pZQ968C1CAFggQr6nXlilhe
uU1IIH3KUGHl0IYhbpH5OmQ1CCWL3j0AYUHswgUq8dfLuVUq4Ti82uaC3X6kNSb1
bjW0bkiHngDoZTvMV4stONPGwgo7oKZ6La5N/Ao97MEdW9OZk7BCSLBiZKJ11RVd
1qpL3GjTM+qndZfcZFWNa7Xc3T9cIhXCaUfor4h75+w5PuKyPjqKvZUK2QARAQAB
tB5PdHRvIEp1bmcgPGhlbGxvQHZhdXBsYWNlLmNvbT6JAk4EEwEIADgWIQQfaVxp
nGiubO7SfRalxg0X0snEdgUCYqxFvwIbAwULCQgHAgYVCgkICwIEFgIDAQIeAQIX
gAAKCRClxg0X0snEdk0eEACQutyI9ft0pHD6Gp8o61qKNGC2xV2GcKtG2i0re3WH
KdVoBMuKN096JrU4TQA672Azupb2VrCj7W5wmUnfoTiqqHTlEKTxQbO9dvAnBb5L
yGs3zwarm8ZdCyS2RRHpxFcY0JQabVsxhHi/ZLvdiL4o5GlTU/jfapp6z4PKVefn
NNCx9LCs6VGKjUYpP2aJawrJfcZhwoD7Oj4zmNmviQww2rgDVyBkfxYklhUjKME0
EnlD14lSUZbA4YhjGbpEgbutTYjs+VVmn9t8OgemT0ufi3rCmxFUpSXKpMLSTNdx
UPLhnp7xL4IsoQWOwxj9rnNCtdaBkiumw6HBGH2i9EKgTa2ltaNGuJYVrLiHCigK
15TKIdwTLvmvKBjQOv4UuvJVGlhtxG+CPxrol8hmhnBNSfDPcKcOGfxyi3OZoN2g
RplJJ28I7Ua1V3zaC//Tdwi80bXRP6CjSlXicg2uPxv8u3cPdzJIP72kp+B3jFkG
weZ+74Z+Sk/6QpjWfT1PTJGLcKiupBLSdFmyDh/Y5easPKy41ZaX9JsKBRiAIRnG
6szU72sVTvm2/kCEKqwSkk1Tls1cwnjDJHbArrDiK7I2ugfvJr9ZTkF0EGD0cwgn
CoKQNpfwVCZiQT0S+krWwlxabP/hcaH6n4BHGvFPM9dxwQJDUoPz93vGN3HSwXoQ
H7kCDQRirEW/ARAAu42hCjzUgjCClZpJy/3JUxgKJZl7+cb4nKLzvrGQg/tRln1/
HRPif+Sic8vBHwmpPm+X2Lk5kp+sYh6U2QBhQPajgs8DttHeg0t+ZqF7Q9dL05xs
QmEzNuaxmuQIYHq1EmVtKkrWjRtETRQv9EDFIAldce4FLnngc+GcW9nZYGZfMZnE
KsAZbkrjdfki0N4lhEh7yNcsM24OAYYZkP/a+vreZRUsN+Z0mZgdctW0uVemkCwn
1IjcjgdrirpT9PXcpeRurDxvALTIlXDMZqdr4yZ+SQJrYkAm1s/o1FOFVp5j0V4L
2gv88RXYU7FWHMhQMdlHvUmc7rCB7xHAkUx5H/tG9nVWwLF+YyrNTfMgkzxX265L
FKFyVTsWn2x6sLA22yCQNb3+pf7oNnhft4EzMzS9pRq3gxABpx1YYLX0FnXXrlvm
lDJECTf47UVpyVSmtP9PRNGIRPZACd1kvj+TkRlJtNvPU0RSkrtskd0eZQ5BCl4W
+QmJo3Q+ndhKojWnKe6SaFQN/eVqrRIj5hwWikxXPhB4f7KpbkRSE42Ye1Mb6mi4
9nzJl3xEFn4FtvtEbl/3pNn37uuKCc9Mpm1ueLE3Yat6bmsRSdAiPZcCdYPlLKC9
QuJ/a7s6SiNSb7UL8WzYMkYpU9lECDePoT7N2SvZubaNt9xe0u/cidwaN8UAEQEA
AYkCNgQYAQgAIBYhBB9pXGmcaK5s7tJ9FqXGDRfSycR2BQJirEW/AhsMAAoJEKXG
DRfSycR2ozwQAJZ8CY/hFjIGKNzYMiXyIN7v/5GxviFeGsq0TjIgVxpNaG4qr4dJ
ohxN/3k3K8DyIL2Q0Oqnx2M+lwzDhtY8UfgX27/1tJEthrbpD0ciA12Nfl2GxLIQ
Rrj0PICHdvhlcxHf/jea3zSNdtffA7dkD/KcKUm1to29xHrfB98mKprXn1dHg6Aa
bTe8J2tZwtaNl+sUrJN6eHj07t2Rq+Lms0a76DAzF5ZjyWkJryas0PJBurc6pE8M
X5vL0QIkMqPGt1PIWP5+++qBKgd2y3MHwRrHOFpu+5pjD9LJNDRVdk/WwLAzJa7a
zampSkW+nwXlpM9h1q1XrE9GY9l1YGR7F2TsnVZUGKQnwG/3KrjVT8sL5NfN+hLT
eGqY/LKC9B9KHQvLcTezlO1Dt+kka4Du/Aka/9CWUbRv0mIivePQHYUUDquvH15V
idEVNOV/Dwo7meHKSu6OsrL6SSop5QjiZAHrW2QGDp6jTY/vmXKeIJLlvvaFylKW
bbIzMZkOXqg9rIf/l8XRPQuHahhGkE3o0YdR4j2anIgm3QHqUnkBlkKGnBFpVmjs
gF3voRFf/47NyxIZeqw5VjREhoDxbvuKVE2UX0lufaFRD8KcNRU+R+h4POGsSej4
Oh5kF8QX9W0EZ2cfs/htfYfoRtJFnpxhlZrwEh6lDXVxYNMmSP5myleT
=mLdw
-----END PGP PUBLIC KEY BLOCK-----
```

References

[1] Saul Gorn. *Handling the Growth by Definition of Mechanical Languages*. Proceedings of the April 18-20 Spring Joint Computer Conference. ACM, 1967, pp. 213–224. DOI: 10.1145/1465482.1465513.

[2] Nachum Dershowitz. “A Taste of Rewrite Systems”. In: *Handbook of Theoretical Computer Science*. Elsevier, 1993, pp. 243–320.

[3] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.001.

[4] Maarten Mol and Arend Rensink. “On A Graph Formalism for Ordered Edges”. In: *ECEASST* 29 (Jan. 2010). DOI: 10.14279/tuj.eceasst.29.417.

[5] H. Ehrig et al. *Term graph rewriting*. Vol. 2. World Scientific, 1999. Chap. 1. DOI: 10.1142/9789812815149_0001.

[6] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford, 1971. URL: https://books.google.pl/books?id=kl1QIQAACAAJ.

[7] Hartmut Ehrig. “Introduction to the algebraic theory of graph grammars (a survey)”. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 1–69. ISBN: 978-3-540-35091-0.

[8] D. A. Turner. “A new implementation technique for applicative languages”. In: *Software: Practice and Experience* 9.1 (1979), pp. 31–49. DOI: 10.1002/spe.4380090105. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090105.

[9] Yves Lafont. “Interaction Nets”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 95–108. ISBN: 0897913434. DOI: 10.1145/96709.96718.

[10] M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen. *Term Graph Rewriting: Theory and Practice*. Wiley, 1993. ISBN: 9780471935674. URL: https://books.google.pl/books?id=A9hQAAAAMAAJ.

[11] Christoph Hoffmann and Michael O’Donnell. “Programming with Equations”. In: *ACM Trans. Program. Lang. Syst.* 4 (Jan. 1982), pp. 83–112. DOI: 10.1145/357153.357158.

[12] O’Donnell and Michael J. *Computing in systems described by equations*. Springer, 1977.

[13] Nachum Dershowitz. “Computing with rewrite systems”. In: *Information and Control* 65.2-3 (1985), pp. 122–157.

[14] Michael J O’Donnell. “Equational logic as a programming language”. In: *Workshop on Logic of Programs*. Springer. 1985, pp. 255–255.

[15] Gregor Kiczales et al. “Aspect-oriented programming”. In: *European conference on object-oriented programming*. Springer. 1997, pp. 220–242.

[16] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “Term Rewriting Meets Aspect-Oriented Programming”. In: *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*. Ed. by Aart Middeldorp et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 88–105. ISBN: 978-3-540-32425-6. DOI: `10.1007/11601548_8`.

[17] Germain Faure and Claude Kirchner. “Exceptions in the rewriting calculus”. In: *International Conference on Rewriting Techniques and Applications*. Springer. 2002, pp. 66–82.

[18] Flaviu Cristian. “Exception handling and software fault tolerance”. In: *IEEE Transactions on Computers* 31.06 (1982), pp. 531–540.

[19] William A Martin and Richard J Fateman. “The MACSYMA system”. In: *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*. 1971, pp. 59–75.

[20] *Wikipedia: Computer algebra system*. URL: `https://en.wikipedia.org/wiki/Computer_algebra_system`.

[21] *Wikipedia: Maxima*. 1982. URL: `https://en.wikipedia.org/wiki/Maxima_(software)`.

[22] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.

[23] André Heck and Wolfram Koepf. *Introduction to MAPLE*. Vol. 1993. Springer, 1993.

[24] Mark van den Brand et al. “Industrial applications of ASF+ SDF”. In: *International Conference on Algebraic Methodology and Software Technology*. Springer. 1996, pp. 9–18.

[25] Martin Bravenboer et al. “Program transformation with scoped dynamic rewrite rules”. In: *Fundamenta Informaticae* 69.1-2 (2006), pp. 123–178.

[26] Gilles Kahn. “Natural semantics”. In: *Annual symposium on theoretical aspects of computer science*. Springer. 1987, pp. 22–39.

[27] J Jouannaud. “Solving equations in abstract algebras: A rule-based survey of unification”. In: *Computational Logic*. MIT-Press. 1991, pp. 257–321.

[28] EGJMH Nöcker et al. “Concurrent clean”. In: *International Conference on Parallel Architectures and Languages Europe*. Springer. 1991, pp. 202–219.

[29] *Wikipedia: Orthogonality*. URL: `https://en.wikipedia.org/wiki/Orthogonality_(term_rewriting)`.

[30] *Wikipedia: Reduction strategy*. URL: `https://en.wikipedia.org/wiki/Reduction_strategy`.

[31] Eelco Visser et al. “A core language for rewriting”. In: *Electronic Notes in Theoretical Computer Science* 15 (1998), pp. 422–441.

[32] Maribel Fernández, Hélène Kirchner, and Olivier Namet. “A Strategy Language for Graph Rewriting”. In: *Logic-Based Program Synthesis and Transformation.* Ed. by Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 173–188. ISBN: 978-3-642-32211-2.

[33] Maribel Fernández and Olivier Namet. “Strategic programming on graph rewriting systems”. In: *arXiv preprint arXiv:1012.5560* (2010).

[34] Peter Borovansk et al. “An overview of ELAN”. In: *Electronic Notes in Theoretical Computer Science* 15 (1998), pp. 55–70.

[35] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. “Towards a strategy language for Maude”. In: *Electronic Notes in Theoretical Computer Science* 117 (2005), pp. 417–441.

[36] Emilie Balland et al. “Tom: Piggybacking rewriting on java”. In: *International Conference on Rewriting Techniques and Applications.* Springer. 2007, pp. 36–47.

[37] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. “The AGG approach: Language and environment”. In: *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools.* World Scientific, 1999, pp. 551–603.

[38] Andy Schürr, Andreas J Winter, and Albert Zündorf. “The PROGRES approach: Language and environment”. In: *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools.* World Scientific, 1999, pp. 487–550.

[39] Ulrich Nickel, Jörg Niere, and Albert Zündorf. “The FUJABA environment”. In: *Proceedings of the 22nd international conference on Software engineering.* 2000, pp. 742–745.

[40] Rubino GeiSS et al. “GrGen: A fast SPO-based graph rewriting tool”. In: *International Conference on Graph Transformation.* Springer. 2006, pp. 383–397.

[41] Detlef Plump. “The graph programming language GP”. In: *International Conference on Algebraic Informatics.* Springer. 2009, pp. 99–122.

[42] Tobias Nipkow and Christian Prehofer. “Higher-order rewriting and equational reasoning”. In: *Automated deductiona basis for applications* 1 (1998), pp. 399–430.

[43] *Wikipedia: First-class citizen.* URL: `https://en.wikipedia.org/wiki/First-class_citizen`.

[44] Horatiu Cirstea and K Kirchner. “The rewriting calculus-Part II”. In: *Logic Journal of the IGPL* 9.3 (2001), pp. 377–410. DOI: `10.1093/jigpal/9.3.339`.

[45] Clara Bertolissi. “The Graph Rewriting Calculus: Confluence and Expressiveness”. In: *Theoretical Computer Science*. Ed. by Mario Coppo, Elena Lodi, and G. Michele Pinna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 113–127. ISBN: 978-3-540-32024-1.

[46] Paul Bernays. “Alonzo Church. An unsolvable problem of elementary number theory. American journal of mathematics, vol. 58 (1936), pp. 345–363.” In: *The Journal of Symbolic Logic* 1.2 (1936), pp. 73–74. DOI: `10.2307/2268571`.

[47] *Wikipedia: BackusNaur form.* URL: `https://en.wikipedia.org/wiki/Backus%5C%E2%5C%80%5C%93Naur_form`.

[48] Albert Gräf. *The Pure Programming Language.* 2008. URL: `https://agraef.github.io/pure-docs/pure.html#recursive-macros`.

[49] D Stott Parker, Mantis HM Cheng, and MH van Emdem. “A Prolog Technology Term Rewriter”. In: *Research paper.(Internet)* (1994).

[50] Besik Dundua, Temur Kutsia, and Klaus Reisenberger-Hagmayer. “An Overview of PρLog”. In: *International Symposium on Practical Aspects of Declarative Languages.* Springer. 2017, pp. 34–49.

[51] *Implementation Repository.* URL: `https://git.vau.place/schrec.git/about/`.

[52] Matthias Felleisen. “On the expressive power of programming languages”. In: *Science of computer programming* 17.1-3 (1991), pp. 35–75.

[53] *Wikipedia: ChurchTuring thesis.* URL: `https://en.wikipedia.org/wiki/Church%5C%E2%5C%80%5C%93Turing_thesis`.

[54] John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine.* MIT Press, 1960. DOI: `10.1145/367177.367199`. URL: `http://hdl.handle.net/1721.1/6096`.

[55] *Wikipedia: Goto.* URL: `https://en.wikipedia.org/wiki/Goto`.

[56] Gerald Jay Sussman and Guy L Steele. “Scheme: A interpreter for extended lambda calculus”. In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 405–439.

[57] Martin L Griss and Anthony C Hearn. “A portable LISP compiler”. In: *Software: Practice and Experience* 11.6 (1981), pp. 541–605.

[58] P. Fradet and D. Le Métayer. “Compilation of lambda-calculus into functional machine code”. In: *TAPSOFT ’89.* Ed. by J. Díaz and F. Orejas. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 155–166. ISBN: 978-3-540-46118-0. DOI: `10.1007/3-540-50940-2_34`.