$$
\begin{array}{lll}
\text{(Regexs) } r ::= & U & \textit{variable} \\
& \mid \text{s} \in \Sigma^* & \textit{base} \\
& \mid r* & \textit{star} \\
& \mid r_1 r_2 & \textit{concat} \\
& \mid r_1 | r_2 & \textit{or}
\end{array}
$$

Figure 1: Regex Syntax

let Δ be the set of user defined data types. let Σ^* be the set of words over the alphabet Σ

$$
\begin{array}{ll}
\text{(Lenses) } l ::= & const(s_1 \in \Sigma^*, s_2 \in \Sigma^*) \\
& \mid identity \\
& \mid iterate(l) \\
& \mid concat(l_1, l_2) \\
& \mid swap(l_1, l_2) \\
& \mid or(l_1, l_2) \\
& \mid l_1 \circ l_2
\end{array}
$$

Figure 2: Lens Syntax

Abstract

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Proof Theory; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis

General Terms Languages, Theory

Keywords Functional Programming, Proof Search, Program Synthesis, Type Theory

1. Introduction

We have regular expressions as normal regular expressions. We expand it with having user defined data types as well. We have lenses, defined in Figure 2, typed as in Figure 3. These lenses have underlying functions, given via Figure 4. We would like to be able to synthesize these lenses automatically, given a specification as two regular expressions, an a set of values that are mapped to each other.

2. Syntax Driven Lenses

Unfortunately, with retyping rule, it is a difficult task. There are an infinite number of possible retypings. They are still relevant, depending on what the examples are. Retyping is needed to express all lenses.

For example, consider $r_1 = a*$, $r_2 = b*$, with examples $a \leftrightarrow bb, aa \leftrightarrow b, aaa \leftrightarrow bbb$. Without retyping, the only possible lens is $iterate(const(a, b))$, which doesn't express it correctly. However, using retyping, we can first retype to $\epsilon + a + aa + aaaa*$ and $\epsilon + b + bb + bbbb*$. This is able to be solved.

Because of the difficulty for lenses including retyping, lets first only worry about syntactic lenses, lenses that don't include retyping.

For these syntactic lenses, one can prove that composition and identity can be applied only on Userdefs without losing expressibility.

We now have the means to enumerate all possible syntactic lenses between two regular expressions in a terminating way. We merely enumerate all the possible terms of a certain type, which there are a finite number of. Then we can run all the lenses on the inputs until we find a lens that satisfies the inputs.

However, this is inefficient, as the number of lenses are possibly exponential in the size of the regular expression. We can prune some

Constant Lens

$$\frac{s_1 \in \Sigma^* \qquad s_2 \in \Sigma^*}{const(s_1, s_2) : s_1 \Leftrightarrow s_2}$$

Identity Lens

$$\frac{}{identity : r \Leftrightarrow r}$$

Iterate Lens

$$\frac{l : r_1 \Leftrightarrow r_2 \qquad r_1^{!*} \qquad r_2^{!*}}{iterate(l) : r_1^* \Leftrightarrow r_2^*}$$

Concat Lens

$$\frac{l_1 : r_{1,1} \Leftrightarrow r_{1,2} \qquad l_2 : r_{2,1} \Leftrightarrow r_{2,2} \qquad r_{1,1}.!r_{2,1} \qquad r_{1,2}.!r_{2,2}}{concat(s_1, s_2) : r_{1,1}r_{2,1} \Leftrightarrow r_{1,2}r_{2,2}}$$

Swap Lens

$$\frac{l_1 : r_{1,1} \Leftrightarrow r_{1,2} \qquad l_2 : r_{2,1} \Leftrightarrow r_{2,2} \qquad r_{1,1}.!r_{2,1} \qquad r_{2,2}.!r_{1,2}}{concat(s_1, s_2) : r_{1,1}r_{2,1} \Leftrightarrow r_{2,2}r_{1,2}}$$

Or Lens

$$\frac{\begin{array}{c} l_1 : r_{1,1} \Leftrightarrow r_{1,2} \\ l_2 : r_{2,1} \Leftrightarrow r_{2,2} \qquad r_{1,1} \cap r_{2,1} = \emptyset \qquad r_{1,2} \cap r_{2,2} = \emptyset \end{array}}{or(l_1, l_2) : r_{1,1}|r_{2,1} \Leftrightarrow r_{1,2}|r_{2,2}}$$

Compose Lens

$$\frac{l_1 : r_1 \Leftrightarrow r_2 \qquad l_2 : r_2 \Leftrightarrow r_3}{l_2 \circ l_1 : r_1 \Leftrightarrow r_3}$$

Retype Lens

$$\frac{l : r_1 \Leftrightarrow r_2 \qquad \mathcal{L}(r_1) \equiv \mathcal{L}(r_1') \qquad \mathcal{L}(r_2) \equiv \mathcal{L}(r_2')}{l : r_1' \Leftrightarrow r_2'}$$

Figure 3: Lens Typing

of the possible lenses very quickly with a key insight, when we reduce the synthesis of the lens into subproblems, we can also make sub-problems for the input-output examples.

For example, consider the problem of synthesizing a lens l between a^*b^* and z^*y^* that satisfies the example $a \leftrightarrow y$. We can apply either a swap or a concat rule to this. If we apply a concat rule, we get the subproblems of synthesizing lenses between a^* and z^*, and synthesizing lenses betwen b^* and y^*. By the underlying function of concat, we know that for this concat lens to work on the provided example, we must have the lens between a^* and z^* send a to ϵ, and must have the lens between b^* and y^* send ϵ to y. However, when trying to apply an iterate rule for a^* and z^*, we immediately see there is no satisfying lens, as parsing a into a^*, we see a mathes the inner regular expression a once. However, on parsing ϵ into z^*, we see ϵ matches the inner regular expression z

- $const(s_1, s_2).putr(s_1) = s_2$
- $identity.putr(s) = s$
- $iterate(l).putr(\epsilon) = \epsilon$
- $iterate(l).putr(s.s') = (l.putr(s)).(iterate(l).putr(s'))$
- $concat(l_1, l_2).putr(s_1, s_2) = l_1(s_1).l_2(s_2)$
- $swap(l_1, l_2).putr(s_1, s_2) = l_2(s_2).l_1(s_1)$
- $or(l_1, l_2).putr(s) = l_1(s)$ if $s \in dom(l_1)$ or $l_2(s)$ if $s \in dom(l_2)$
- $(l_2 \circ l_1).putr(s) = l_2.putr(l_1.putr(s))$

Figure 4: Lens Semantics

CONSTANT LENS

$$\frac{s_1 \in \Sigma^* \qquad s_2 \in \Sigma^*}{\mathit{const}(s_1, s_2) : s_1 \Leftrightarrow s_2}$$

IDENTITY LENS

$$\frac{}{\mathit{identity} : r \Leftrightarrow r}$$

ITERATE LENS

$$\frac{l : r_1 \Leftrightarrow r_2 \qquad r_1^{!*} \qquad r_2^{!*}}{\mathit{iterate}(l) : r_1^* \Leftrightarrow r_2^*}$$

CONCAT LENS

$$\frac{\begin{array}{c} l_1 : r_{1,1} \Leftrightarrow r_{1,2} \\ l_2 : r_{2,1} \Leftrightarrow r_{2,2} \qquad r_{1,1}.!r_{2,1} \qquad r_{1,2}.!r_{2,2} \end{array}}{\mathit{concat}(s_1, s_2) : r_{1,1}r_{2,1} \Leftrightarrow r_{1,2}r_{2,2}}$$

SWAP LENS

$$\frac{\begin{array}{c} l_1 : r_{1,1} \Leftrightarrow r_{1,2} \\ l_2 : r_{2,1} \Leftrightarrow r_{2,2} \qquad r_{1,1}.!r_{2,1} \qquad r_{2,2}.!r_{1,2} \end{array}}{\mathit{concat}(s_1, s_2) : r_{1,1}r_{2,1} \Leftrightarrow r_{2,2}r_{1,2}}$$

OR LENS

$$\frac{\begin{array}{c} l_1 : r_{1,1} \Leftrightarrow r_{1,2} \\ l_2 : r_{2,1} \Leftrightarrow r_{2,2} \qquad r_{1,1} \cap r_{2,1} = \emptyset \qquad r_{1,2} \cap r_{2,2} = \emptyset \end{array}}{\mathit{or}(l_1, l_2) : r_{1,1}|r_{2,1} \Leftrightarrow r_{1,2}|r_{2,2}}$$

COMPOSE LENS

$$\frac{l_1 : r_1 \Leftrightarrow r_2 \qquad l_2 : r_2 \Leftrightarrow r_3}{l_2 \circ l_1 : r_1 \Leftrightarrow r_3}$$

Figure 5: Syntactic Lens Typing

$$\begin{array}{rll} \text{(Parse Tree) } p ::= & s \in \Sigma^* \\ & | \ u \in \Delta, s \in \Sigma^* \\ & | \ \mathit{starparse}([p_1; \ldots; p_n]) \\ & | \ p_1 p_2 \\ & | \ l.(p) \\ & | \ r.(p) \end{array}$$

Figure 6: Parse Tree Definition

zero times. Through the underlying function of iterate, we know this cannot be done, so there is no lens that does this. So we immediately know that there is no concat lens between a^*b^* and z^*y^* that sends a to y. We can then continue with trying the swap lens, which will eventually provide a lens.

We can formalize this first by definining a data structure for how a regular expression parses a string, as in Figure 6, and typing for that data structure as in Figure 7.

With the data structure for parsing, we can now define a typing for a lens which satisfies the parse tree of an example in Figure 8. Now, to generate a lens for an input output example, between two regular expressions, needs to generate the parse tree for the example with those expressions, and then find a lens between those two regular expressions and between the parse trees.

We say a lens $l : r \Leftrightarrow s$ satisfies a set of parse tree input output examples $p_1 \leftrightarrow q_1, \ldots, p_n \leftrightarrow q_n$ if we can show $l : r \Leftrightarrow s, p_i \leftrightarrow q_i$ for all i between 1 and n.

3. DNF Regular Expressions and Lenses

We can push the boundaries of what is expressible through syntactic lenses through finding a language, equivalent to regular expressions, with fewer equivalences.

CONSTANT PARSE TREE

$$\frac{s \in \Sigma^*}{\Delta \vdash s : s}$$

USERDEF PARSE TREE

$$\frac{\Delta' \vdash p : (r, s)}{\Delta' \cup \{(r, U)\} \vdash s : (U, s)}$$

CONCAT PARSE TREES

$$\frac{\Delta \vdash p_1 : (r_1, s_1) \qquad \Delta \vdash p_2 : (r_2, s_2)}{\Delta \vdash p_1 p_2 : (r_1 r_2, s_1 s_2)}$$

LEFT OR EXAMPLE

$$\frac{\Delta \vdash p : (r, s)}{\Delta \vdash l.(p) : (r|r', s)}$$

RIGHT OR EXAMPLE

$$\frac{\Delta \vdash p : (r, s)}{\Delta \vdash r.(p) : (r'|r, s)}$$

STAR EMPTY

$$\frac{}{\Delta \vdash \mathit{starparse}([]) : (r^*, \epsilon)}$$

STAR ADD PARSE

$$\frac{\Delta \vdash p : (r, s_1) \qquad \Delta \vdash \mathit{starparse}(ps) : (r^*, s_2)}{\Delta \vdash \mathit{starparse}(p :: ps) : (r^*, s_1 s_2)}$$

Figure 7: Parse Tree Typing

CONSTANT LENS

$$\frac{s_1 \in \Sigma^* \qquad s_2 \in \Sigma^*}{\mathit{const}(s_1, s_2) : s_1 \Leftrightarrow s_2, s_1 \leftrightarrow s_2}$$

IDENTITY LENS

$$\frac{\Delta' \vdash p : (r, s)}{\Delta' \cup \{(r, U)\} \vdash \mathit{identity} : U \Leftrightarrow U, s \leftrightarrow s}$$

CONCAT LENS

$$\frac{\begin{array}{c} \Delta \vdash l_1 : r_1 \Leftrightarrow s_1, p_1 \leftrightarrow q_1 \\ \Delta \vdash l_2 : r_2 \Leftrightarrow s_2, p_2 \leftrightarrow q_2 \qquad r_1.!r_2 \qquad s_1.!s_2 \end{array}}{\mathit{concat}(l_1, l_2) : r_1 r_2 \Leftrightarrow s_1 s_2, p_1 p_2 \leftrightarrow q_1 q_2}$$

SWAP LENS

$$\frac{\begin{array}{c} \Delta \vdash l_1 : r_1 \Leftrightarrow s_1, p_1 \leftrightarrow q_1 \\ \Delta \vdash l_2 : r_2 \Leftrightarrow s_2, p_2 \leftrightarrow q_2 \qquad r_1.!r_2 \qquad s_2.!s_1 \end{array}}{\mathit{swap}(l_1, l_2) : r_1 r_2 \Leftrightarrow s_2 s_1, p_1 p_2 \leftrightarrow q_2 q_1}$$

OR LENS LEFT

$$\frac{\begin{array}{c} \Delta \vdash l_1 : r_1 \Leftrightarrow s_1, p \leftrightarrow q \\ l_2 : r_2 \Leftrightarrow s_2 \qquad r_1 \cap s_1 = \emptyset \qquad r_2 \cap s_2 = \emptyset \end{array}}{\Delta \vdash \mathit{or}(l_1, l_2) : r_{1,1}|r_{2,1} \Leftrightarrow r_{1,2}|r_{2,2}, l.(p) \leftrightarrow l.(q)}$$

OR LENS RIGHT

$$\frac{\begin{array}{c} l_1 : r_1 \Leftrightarrow s_1 \\ \Delta \vdash l_2 : r_2 \Leftrightarrow s_2, p \leftrightarrow q \qquad r_1 \cap s_1 = \emptyset \qquad r_2 \cap s_2 = \emptyset \end{array}}{\Delta \vdash \mathit{or}(l_1, l_2) : r_{1,1}|r_{2,1} \Leftrightarrow r_{1,2}|r_{2,2}, r.(p) \leftrightarrow r.(q)}$$

ITERATE LENS EMPTY

$$\frac{l : r \Leftrightarrow s \qquad r^{!*} \qquad s^{!*}}{\Delta \vdash \mathit{iterate}(l) : r_1^* \Leftrightarrow r_2^*, \mathit{starparse}([]) \leftrightarrow \mathit{starparse}([])}$$

ITERATE LENS ADD EXAMPLE

$$\frac{\begin{array}{c} \Delta \vdash \mathit{iterate}(l) : r^* \Leftrightarrow s^*, \mathit{starparse}(ps) \leftrightarrow \mathit{starparse}(qs) \\ \Delta \vdash l : r \Leftrightarrow s, p \leftrightarrow q \end{array}}{\Delta \vdash \mathit{iterate}(l) : r^* \Leftrightarrow s^*, \mathit{starparse}(p :: ps) \leftrightarrow \mathit{starparse}(q :: qs)}$$

Figure 8: Syntactic Lens and Example Typing

$$\begin{array}{llll} \text{(Atoms)} & a & ::= & \mathrm{u} \in \Delta \mid dr* \\ \text{(Clauses)} & cl & ::= & [s_1; a_1; s_2; \ldots; s_n; a_n; s_{n+1}] \\ \text{(DNF Regex)} & dr & ::= & [cl_1 \ldots ; cl_n] \end{array}$$

Figure 9: DNF Regex Syntax

$$\begin{array}{llll} \text{(Atom Lenses)} & al & ::= & Iterate(dl) \mid Identity \\ \text{(Clause Lenses)} & cll & ::= & ([(s_{1,1}, s_{1,2}); al_1; (s_{2,1}, s_{2,2}); \\ & & & \ldots \\ & & & ; (s_{n,1}, s_{n,2}); al_n; (s_{n+1,1}, s_{n+1,2})], \\ & & & \sigma \in [0,n] \rightarrowtail [0,n]) \\ \text{(DNF Lenses)} & dl & ::= & ([cll_1; \ldots; cll_n], \sigma \in [0,n] \rightarrowtail [0,n]) \end{array}$$

Figure 10: DNF Regex Syntax

Semantics of DNF Lenses:

- $[cll_1; \ldots; cll_n] : [cl_1; \ldots; cl_n]$
  $[cll_1; \ldots; cll_n].putr(s) = \{cll_1.putr(s)$ if $s \in cl_1$, …, $cll_n.putr(s)$ if $s \in cll_n\}$

Semantics of Clause Lenses:

- $([(s_{1,1}, s_{1,2}); al_1; \ldots; al_n; (s_{n+1,1}, s_{n+1,2})], \sigma)$
  $.putr(s_{1,1}.s_1.\ldots.s_n.s_{1,n+1}) =$
  $s_{2,1}.(al_{\sigma(1)}.putr(s_{\sigma(1)})).\ldots.(al_{\sigma(n)}.putr(s_{\sigma(n)})).s_{2,n+1}$

Semantics of Atom Lenses:

- $identity.putr(s) = s$
- $iterate(l).putr(\epsilon) = \epsilon$
- $iterate(l).putr(s.s') = (l.putr(s)).(iterate(l).putr(s'))$

Figure 11: DNF Lens Semantics

A language which removes some of those equivalences is the language of regular expressions in disjunctive normal form. This language removes the equivalences corresponding to distributivity, associativity, and concatenation identity. Figure 9 defines this. The full DNF Regex is a list of Clauses. This corresponds to a chain of +s in the normal language of regular expressions. A Clause is a list of Atoms, with strings in between. This corresponds to a chain of composition. These strings correspond to regular expression base types, and their requirement to exist between every atom removes the ϵ equivalences. Atoms correspond to pieces that either cannot be broken up uniquely, or we feel should not be broken up. These are the star regular expressions, and the user defined regular expressions. The star regular expressions can be broken up in an infinite number of ways. We feel that a user grouping together aspects of their regular expression together signifies that it will likely stay together in the transformation.

As we have dnf regular expressions, it also makes sense to have dnf lenses, whose types are dnf regular expressions. We define dnf lenses in Figure 10. These dnf lenses have the comparable underlying functions to the underlying functions of normal lenses, and are given in Figure 11 TODO: make a nice diagram of how clause lenses work. They can be typed according to the rules given in Figure 12

One thing to note about these lenses is that they are slightly more expressive than the syntactic lenses given previously. Not all permutations are possible through only using swap. Furthermore, before we did not allow for permutations on union clauses, which previously were dealt with through the type change rule of pure lenses. Now, these permutations are possible. So, for example, there

IDENTITY ATOM LENS

$$\frac{}{identity : r \Leftrightarrow r}$$

ITERATE ATOM LENS

$$\frac{dl : dr_1 \Leftrightarrow dr_2 \qquad dr_1^{!*} \qquad dr_2^{!*}}{iterate(dl) : dr_1^* \Leftrightarrow dr_2^*}$$

CLAUSE LENS

$$\frac{\begin{array}{c} al_1 : a_{1,1} \Leftrightarrow a_{1,2} \\ \ldots \qquad al_n : a_{n,1} \Leftrightarrow a_{n,2} \qquad a_{i,j}.!a_{i+1,j} \end{array}}{\begin{array}{c} ([(s_{1,1}, s_{1,2}); al_1; \ldots; al_n(s_{n+1,1}, s_{n+1,2})], \sigma) : \\ ([a_{1,1}; \ldots; a_{n,1}], [s_{1,1}; \ldots; s_{n+1,1}]) \Leftrightarrow \\ ([a_{\sigma(1),2}; \ldots; a_{\sigma(n),2}], [s_{1,2}; \ldots; s_{n+1,2}]) \end{array}}$$

DNF REGEX LENS

$$\frac{\begin{array}{c} cll_1 : cl_{1,1} \Leftrightarrow cl_{1,2} \\ \ldots \qquad cll_n : cl_{n,1} \Leftrightarrow cl_{n,2} \qquad cl_i \cap cl_j = \emptyset \end{array}}{([cll_1; \ldots; cll_n], \sigma) : [cl_{1,1}; \ldots; cl_{n,1}] \Leftrightarrow [cl_{\sigma^{-1}(1),2}; \ldots; cl_{\sigma^{-1}(n),2}]}$$

Figure 12: DNF Lens Typing

are no syntactic lenses between $a + b*$ and $z * +y$. However, with allowing for permutations, the new language can synthesize these types of lenses.

Just as we used the parse trees of strings within a regular expression, to help guide the synthesis for syntactic lenses, we can do the same for dnf lenses.

4. Proofs

4.1 Soundness

We say that dnf lenses are *sound* if, there is a dnf lens between two dnf regular expressions, then between any two regular expressions equivalent to the two dnf regular expressions, there is a lens between those regular expressions such that the lens and dnf lens have the same semantics.

Definition 1 (repregex). *We define a representative regex for a dnf regex as follows:*

- $repregex([cl]) = repregex(cl)$
- $repregex([cl_1; \ldots; cl_n]) =$
  $repregex(cl_1)|repregex([cl_2; \ldots; cl_n]))$
- $regregex([s]) = s$
- $repregex([s_1; a_1; \ldots; a_n; s_{n+1}]) =$
  $s_1(repregex(a_1)repregex([s_2; a_2; \ldots; s_{n+1}]))$
- $repregex(U) = U$
- $repregex(dr^*) = repregex(dr)^*$

Lemma 1 (Equivalence of repregex). $\mathcal{L}(dr) = \mathcal{L}(repregex(dr))$, $\mathcal{L}(cl) = \mathcal{L}(repregex(cl))$*, and* $\mathcal{L}(a) = \mathcal{L}(repregex(a))$

Proof. By induction on the structure of dr, cl, and a □

Definition 2 (Adjacent Swapping Permutation). *Let* $\sigma_i : [0,n] \rightarrowtail [0,n]$ *be the permutation, where* $\sigma_i(i) = i + 1$, $\sigma_i(i+1) = i$, $\sigma_{i,j}(k \neq i, i+1) = k$

Lemma 2 (Expressibility of Adjacent Swapping Permutation). *Let σ_i be an adjacent element swapping permutation. The language of lenses can express* $([(s_1, s_1); identity; \ldots; identity; (s_n, s_n)], \sigma_i)$.

Proof. Consider the regular expressions $repregex([s_1; a_1; \ldots; s_i])$
$(repregex(a_i)(s_{i+1}repregex(a_{i+1})))$ $repregex([s_{i+2}; \ldots; a_n; s_{n+1}])$
and $repregex([s_{1,2}; a_{1,2}; \ldots; s_{i,1}])$ $((a_{i+1,1}s_{i+1,1})a_{i,1})$ $repregex([s_{i+2,1}; \ldots;$
Consider the lens between them
$concat(concat(identity, swap(identity, swap(identity, identity))), identity$

By inspection, this lens is equivalent to the adjacent swapping permutation. □

Lemma 3 (Expressibility of Permutation)**.** *The language of lenses can express* $([(s_1, s_1); identity; \ldots; identity; (s_n, s_n)], \sigma)$ *for any permutation* σ.

Proof. Let σ be a permutation. Consider the clause lens $([(s_1, s_1); identity; \ldots; identity; (s_n, s_n)], \sigma_i)$. From algebra, we know that the group of permutations is generated by all adjacent swaps $\sigma_i = (i, i + 1)$. So there exists an adjacency swap decomposition of $\sigma = \sigma_{i_1} \ldots \sigma_{i_m}$. Consider the dnf lens $([(s_1, s_1); identity; \ldots; identity; (s_n, s_n)], \sigma_{i_j})$ for each σ_{i_j}. By the above lemma, there exists a l_j for each of these adjacency swaps. Consider the lens $l = l_{i_1} \circ l_{i_2} \circ \ldots \circ l_{i_m}$ By the semantics, they are the same. □

Theorem 1 (Soundness)**.** *Let* r *and* s *be two regular expressions, and* dr *and* ds *be two dnf regular expressions. If* $\mathcal{L}(r) = \mathcal{L}(dr)$ *and* $\mathcal{L}(s) = \mathcal{L}(ds)$*, then if there exists a dnf lens* $dl : dr \Leftrightarrow ds$*, then there exists a lens* $l : r \Leftrightarrow s$ *such that* $dl.putr = l.putr$.

Proof. By induction on the typing dl.

DNF Lens Intro Let $\Delta \vdash ([cll_1; \ldots; cll_n], \sigma) : [cl_{1,1}, \ldots, cl_{n,1}] \Leftrightarrow [cl_{\sigma(1),2}, \ldots, cl_{\sigma(n),2}]$. This comes from the derivations that $\Delta \vdash cll_i : cl_{i,1} \Leftrightarrow cl_{i,2}$ for all i. Consider instead the lens $dl' = \Delta \vdash ([cll_1; \ldots; cll_n], \sigma_i d) : [cl_{1,1}, \ldots, cl_{n,1}] \Leftrightarrow [cl_{1,2}, \ldots, cl_{n,2}]$. By Lemma (TODO: this lemma), these two lenses are semantically equivalent. By Lemma (TODO: this lemma), $\Delta \vdash repregex(dl') : repregex(dr_1) \Leftrightarrow repregex(dr'_2)$, with $repregex(dl)$ semantically equivalent to dl. So $repregex(dl')$ is semantically equivalent to dl', and $DNFLens'$ is semantically equivalent dl, so $repregex(dl')$ is semantically equivalent to dl. Merely adding in a retyping rule at the end of $repregex(dl')$ (as they have the same type), completes this case.

Clause Lens Intro Let $\Delta \vdash ([(s_{1,1}, s_{1,2}); al_1; \ldots; al_n; (s_{n+1,1}, s_{n+1,2})], \sigma) : [s_{1,1}; a_{1,1}; \ldots; a_{1,n}; s_{1,n+1}] \Leftrightarrow [s_{1,2}; a_{\sigma(1),2}; \ldots; a_{\sigma(n),2}; s_{n+1,2}]$. Consider two lenses, $\Delta \vdash ([(s_{1,1}, s_{1,2}); al_1; \ldots; al_n; (s_{n+1,1}, s_{n+1,2})], \sigma_i d) : [s_{1,1}; a_{1,1}; \ldots; a_{1,n}; s_{1,n+1}] \Leftrightarrow [s_{1,2}; a_{1,2}; \ldots; a_{n,2}; s_{n+1,2}]$, and $\Delta \vdash ([(s_{1,2}, s_{1,2}); identitylens(a_{1,2}); \ldots; identitylens(a_{n,2}); (s_{n+1,2},$ $[s_{1,2}; a_{1,2}; \ldots; a_{n,2}; s_{n+1,2}] \Leftrightarrow [s_{1,2}; a_{\sigma(1),2}; \ldots; a_{\sigma(n),2}; s_{n+1,2}]$. By Lemma (TODO:), there exists a lens equivalent to the first one, call it $l_{transform}$. By Lemma (TODO:), there exists a lens equivalent to the second one, call it l_σ. Consider $l_\sigma \circ l_{transform}$. Go through semantics, oh look they are equivalent. □

5. Search Strategy

5.1 Equivalence Relation

5.2 Distance Metric

6. Implementation

6.1 Evaluation

7. Related Work

7.1 FlashFill

7.2 FlashExtract

7.3 SemFill

8. Future Work

8.1 Complex Data Structures

8.2 Richer Classes of Lenses

Acknowledgments

References