$$
\begin{array}{lrll}
\text{(String)} & s,t & ::= & s \in \Sigma^* & \\
\text{(Regexs)} & r,q & ::= & \mathrm{s} & \text{Base} \\
& & | & r^* & \text{Star} \\
& & | & r_1 \cdot r_2 & \text{Concat} \\
& & | & r_1 | r_2 & \text{Or}
\end{array}
$$

Figure 1: Regex Syntax [B: Recommend using capital letters (R, S, ...) for regular expressions. And left-justifying the RH column and putting it in italic (with more space separating it from the rest) so that it's clear that it's commentary.]

- $\mathcal{L}(s) = \{s\}$
- $\mathcal{L}(r_1 \cdot r_2) = \{s_1 \cdot s_2 | s_1 \in \mathcal{L}(r_1) \wedge s_2 \in \mathcal{L}(r_2)\}$
- $\mathcal{L}(r_1 | r_2) = \{s | s \in \mathcal{L}(r_1) \vee s \in \mathcal{L}(r_2)\}$
- $\mathcal{L}(r^*) = \{s_1 \cdot \ldots \cdot s_n | n \in \mathbb{N} \wedge s_i \in \mathcal{L}(r)\}$

Figure 2: Regex Semantics [B: The vertical bars in set-comprehension notation need more space around them.]

$$
\begin{array}{lll}
\text{(Lenses) } l ::= & const(s_1 \in \Sigma^*, s_2 \in \Sigma^*) & \text{Const} \\
| & identity & \text{Identity} \\
| & iterate(l) & \text{Iterate} \\
| & concat(l_1, l_2) & \text{Concat} \\
| & swap(l_1, l_2) & \text{Swap} \\
| & or(l_1, l_2) & \text{Or} \\
| & l_1 \circ l_2 & \text{Compose}
\end{array}
$$

Figure 3: Lens Syntax

[B: Why doesn't the title appear?]

Abstract

[B: Recommend inlining everything into one file so that it's easier to find things...]

The syntax for nonempty[B: why not empty ones too??] regular expressions is given in Figure 1. These regular expressions have an underlying semantics, which expresses languages, formalized in Figure 2.

We focus on the language of bijective lenses, where the functions between views are bijections, and inverses to each other. We define the syntax for these bijective lenses in Figure 3. The semantics and typing for these lenses is defined in Figure 4. The semantics is only defined alongside the typing, as the functions are only well defined when the term is well typed.

A close inspection of the laws behind the lenses shows that the typing of the lenses corresponds tightly with the actual lenses. For example, consider the concatenation rule:

$$
\frac{\begin{array}{cc} l_1 : r_1 \Leftrightarrow q_1 & l_2 : r_2 \Leftrightarrow q_2 \\ \mathcal{L}(r_1).!\mathcal{L}(r_2) & \mathcal{L}(r_2).!\mathcal{L}(q_2) \end{array}}{concat(l_1, l_2) : r_1 \cdot r_2 \Leftrightarrow q_1 \cdot q_2}
$$

Concatenation lenses come alongside a regular expression concatenation. Perhaps, instead of writing out lenses explicitly, we can merely write its type, and the lenses can be inferred from the types. Utilizing types for synthesis is not a new idea, and has been explored in systems like Myth [1]. However, we have found that type directed synthesis is well suited for domain specific languages like the language of lenses. The constraints of the type systems of these languages make the search space tighter, and allow us to synthesize very complicated programs very quickly.

Our approach to synthesis is one of enumeration. We would like to enumerate the programs that satisfy the desired typing until we find one that satisfies the examples. We enumerate these programs by creating subproblems based on typing rules could potentially create the desired type.

For example, given the type $r_1 \cdot r_2 \Leftrightarrow q_1 \cdot q_2$. We know that it can be created from a Concat Lens rule, which would create the subproblems of finding a lens of type $r_1 \Leftrightarrow q_1$ and a lens of type $r_2 \Leftrightarrow q_2$. However, this is made more difficult due to the Compose Lens rule, and the Retype Lens rule. Both of these rules can be applied to any type, and the subproblems are not immediately apparent. For example, if the lens of type $r_1 \cdot r_2 \Leftrightarrow q_1 \cdot q_2$ came from a Compose Lens rule, then for some regular expression r, we would have the subproblems of finding a lens with type $r_1 \cdot r_2 \Leftrightarrow r$, and finding a lens with type $r \Leftrightarrow q_1 \cdot q_2$. However, merely from

CONSTANT LENS

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

IDENTITY LENS

$$
\frac{}{identity : r \Leftrightarrow r \triangleright \lambda s.s, \lambda s.s}
$$

ITERATE LENS

$$
\frac{l : r \Leftrightarrow q \triangleright putr, putl \qquad \mathcal{L}(r)^{!*} \qquad \mathcal{L}(q)^{!*}}{\begin{array}{c} iterate(l) : r^* \Leftrightarrow q^* \triangleright \\ \lambda s.\text{let } s_1 \cdot \ldots \cdot s_n = s \text{ where } s_i \in \mathcal{L}(r) \text{ in } (putr\ s_1) \cdot \ldots \cdot (putr\ s_n), \\ \lambda s.\text{let } s_1 \cdot \ldots \cdot s_n = s \text{ where } s_i \in \mathcal{L}(q) \text{ in } (putl\ s_1) \cdot \ldots \cdot (putl\ s_n) \end{array}}
$$

CONCAT LENS

$$
\frac{\begin{array}{cc} l_1 : r_1 \Leftrightarrow q_1 \triangleright putr_1, putl_1 & l_2 : r_2 \Leftrightarrow q_2 \triangleright putr_2, putl_2 \\ \mathcal{L}(r_1).!\mathcal{L}(r_2) & \mathcal{L}(q_1).!\mathcal{L}(q_2) \end{array}}{\begin{array}{c} concat(l_1, l_2) : r_1 r_2 \Leftrightarrow q_1 q_2 \triangleright \\ \lambda s.\text{let } s_1 \cdot s_2 = s \text{ where } s_i \in \mathcal{L}(r_i) \text{ in } (putr_1\ s_1) \cdot (putr_2\ s_2), \\ \lambda s.\text{let } s_1 \cdot s_2 = s \text{ where } s_i \in \mathcal{L}(q_i) \text{ in } (putl_1\ s_1) \cdot (putl_2\ s_2) \end{array}}
$$

SWAP LENS

$$
\frac{\begin{array}{cc} l_1 : r_1 \Leftrightarrow q_1 \triangleright putr_1, putl_2 & l_2 : r_2 \Leftrightarrow q_2 \triangleright putr_1, putl_2 \\ \mathcal{L}(r_1).!\mathcal{L}(r_2) & \mathcal{L}(q_2).!\mathcal{L}(q_1) \end{array}}{\begin{array}{c} swap(l_1, l_2) : r_1 r_2 \Leftrightarrow q_2 q_1 \triangleright \\ \lambda s.\text{let } s_1 \cdot s_2 = s \text{ where } s_i \in \mathcal{L}(r_i) \text{ in } (putr_2\ s_2) \cdot (putr_1\ s_1), \\ \lambda s.\text{let } s_1 \cdot s_2 = s \text{ where } s_i \in \mathcal{L}(q_i) \text{ in } (putl_2\ s_2) \cdot (putl_1\ s_1) \end{array}}
$$

OR LENS

$$
\frac{\begin{array}{cc} l_1 : r_1 \Leftrightarrow q_1 \triangleright putr_1, putl_1 & l_2 : r_2 \Leftrightarrow q_2 \triangleright putr_2, putl_2 \\ \mathcal{L}(r_1) \cap \mathcal{L}(r_2) = \emptyset & \mathcal{L}(q_1) \cap \mathcal{L}(q_2) = \emptyset \end{array}}{\begin{array}{c} or(l_1, l_2) : r_1 | q_1 \Leftrightarrow r_2 | q_2 \triangleright \\ \lambda s.\{putr_1(s) \text{ if } s \in \mathcal{L}(r_1)\ , putr_2(s) \text{ if } s \in \mathcal{L}(r_2)\ \}, \\ \lambda s.\{putl_1(s) \text{ if } s \in \mathcal{L}(q_1)\ , putl_2(s) \text{ if } s \in \mathcal{L}(q_2)\ \} \end{array}}
$$

COMPOSE LENS

$$
\frac{l_1 : r_1 \Leftrightarrow r_2 \triangleright putr_1, putl_1 \qquad l_2 : r_2 \Leftrightarrow r_3 \triangleright putr_2, putl_2}{l_2 \circ l_1 : r_1 \Leftrightarrow r_3 \triangleright putr_2 \circ putr_1, putl_2 \circ putl_1}
$$

RETYPE LENS

$$
\frac{l : r_1 \Leftrightarrow r_2 \triangleright putr, putl \qquad \mathcal{L}(r_1) = \mathcal{L}(r_1') \qquad \mathcal{L}(r_2) = \mathcal{L}(r_2')}{l : r_1' \Leftrightarrow r_2' \triangleright putr, putl}
$$

Figure 4: Lens Semantics and Typing

1.	r	$\equiv$	r
2.	$(r \cdot r') \cdot r''$	$\equiv$	$r \cdot (r' \cdot r'')$
3.	$(r \mid r') \mid r''$	$\equiv$	$r \mid (r' \mid r'')$
4.	$r \mid q$	$\equiv$	$q \mid r$
5.	$r \cdot (r' \mid r'')$	$\equiv$	$(r \cdot r') \mid (r \cdot r'')$
6.	$(r' \mid r'') \cdot r$	$\equiv$	$(r' \cdot r) \mid (r'' \cdot r)$
7.	$r \cdot \epsilon$	$\equiv$	r
8.	$(r \mid q)^*$	$\equiv$	$(r^* \cdot q)^* \cdot r^*$
9.	$r \cdot q^*$	$\equiv$	$\epsilon \mid (r \cdot (q \cdot r)^* \cdot q)$
10.	$(r^*)^*$	$\equiv$	r^*
11.	r^*	$\equiv$	$(r^n)^* \cdot r^{<n}$

Figure 5: Regular Expression Equivalences

(Atoms)	a,b	::=	$dr*$ Iterate DNF
(Conjunct)	co,bo	::=	$[s_0; a_1; \ldots; a_n; s_n]$ Conjoin
(DNF Regex)	dr,dq	::=	$[co_1; \ldots; co_n]$ DNF Or

Figure 6: DNF Regex Syntax

the resulting type, $r_1 \cdot r_2 \Leftrightarrow q_1 \cdot q_2$, we do not know which regular expression r to use.

Similarly, if the lens of type $r_1 \cdot r_2 \Leftrightarrow q_1 \cdot q_2$ came from a Retype Lens rule, then for some regular expressions r, and q, where $\mathcal{L}(r) = \mathcal{L}(r_1 \cdot r_2)$ and $\mathcal{L}(q) = \mathcal{L}(q_1 \cdot q_2)$, we have the subproblem of finding a lens with type $r \Leftrightarrow q$. However, merely from the resulting type, $r_1 \cdot r_2 \Leftrightarrow q_1 \cdot q_2$, we do not know which regular expressions r and q to use, we only know that they have to meet some semantic restrictions.

However, neither of these rules are admissible and both are important. Without the Retype Lens rule, it would not be possible to have a lens of the type $a \cdot (b \cdot (c \cdot d)) \Leftrightarrow (a \cdot b) \cdot (c \cdot d)$, despite the fact that we expect this lens to be inhabited by the lens $identity$. Without the Compose Lens rule, it would not be possible to have a lens l of the type $a^* \cdot b^* \cdot c^* \cdot d^* \Leftrightarrow c^* \cdot a^* \cdot d^* \cdot b^*$, which has $l.putr(a^h \cdot b^i \cdot c^j \cdot d^k) = c^j \cdot a^h \cdot d^k \cdot b^i$, a simple permutation of the letters.

These rules are important, but on each application of the rules, there are an infinite number of potential subproblems. Making the composition admissible is not the most complicated problem, as it's lack of admissibility comes a bit from the rigid structure of the rules, and only arises in an edge case. However, retyping is not so easily made admissible. For the handling of the Retype Lens rule, we can create a search space for these potential subproblems by searching through the lenses of equivalent regular expressions. For this, we use the equational theory presented by Conway [**?**], and proven complete by Krob [**?**], shown in Figure 5.

However, enumerating through all possible regular expressions for lenses is inefficient. Beyond that, we are not merely searching for the applications of these rules to a single regular expression, but instead the applications of these to two regular expressions. This makes this search space incredibly broad. Instead of searching through this search space, we would instead like to find a normal form for regular expressions, and find a way to create lenses on regular expressions of that form.

To this end, we have created DNF regular expressions. Intuitively, a DNF regular expression is a regular expression with the distributivity laws (6 and 7) always applied, with no associativity between operations of the same type, and with base regular expressions at fixed locations. We formalize the syntax of this language in Figure 6.

The outermost layer is a list of conjuncts. This layer intuitively corresponds to the choices involved in regular expression matching, and so represents where the Ors of normal regular expressions exist.

- $\mathcal{L}(dr^*) = \{s_1 \cdot \ldots \cdot s_n | n \in \mathbb{N} \wedge s_i \in \mathcal{L}(dr)\}$
- $\mathcal{L}([s_0; a_1; \ldots; a_n; s_n]) = \{s_0 \cdot s_1' \cdot \ldots \cdot s_n' \cdot s_n | s_i' \in \mathcal{L}(a_i)\}$
- $\mathcal{L}([co_1; \ldots; co_n]) = \{s | s \in \mathcal{L}(co_i) \text{ for some } i \in [1, n]\}$

Figure 7: DNF Regex Semantics [B: I think I know what the middle rule must mean, but I can't understand what's written very well.]

ConcatConjunct:
$ConcatConjunct : Conjunct \to Conjunct$
$ConcatConjunct([s_0; a_1; \ldots; a_n; s_n], [t_0; b_1; \ldots; b_m; t_m]) =$
$\quad ConcatConjunct([s_0; a_1; \ldots; a_n; s_n \cdot t_0; b_1; \ldots; b_m; t_m])$

ConcatDNF:
$ConcatDNF : DNF \to DNF$
$ConcatDNF([co_0; \ldots; co_n], [bo_0; \ldots; bo_m]) =$

$$[\ ConcatConjunct(co_0, bo_0);\ \ldots\ ConcatConjunct(co_0, bo_m);$$
$$\ldots$$
$$ConcatConjunct(co_n, bo_0);\ \ldots\ ConcatConjunct(co_n, bo_m)\ ]$$

OrDNF:
$OrDNF : DNF \to DNF$
$OrDNF([co_0; \ldots; co_n], [bo_0; \ldots; bo_m])$**=**
$\quad [co_0; \ldots; co_n; bo_0; \ldots; bo_m]$

RepeatDNF:
$RepeatDNF : \mathbb{N} \to DNF \to DNF$
$RepeatDNF(0, dr) = [[\epsilon]]$
$RepeatDNF(n, dr) =$
$\quad ConcatDNF(dr, RepeatDNF(n-1, dr))$

StarModNDNF:
$StarModNDNF : \mathbb{N}_{\geq 1} \to DNF \to DNF$
$StarModNDNF(1, dr) = [[\epsilon]]$
$StarModNDNF(n, dr) =$
$\quad OrDNF(StarModNDNF(n-1, dr), RepeatDNF(n-1, dr))$

Figure 8: DNF Regex Functions [B: Instead of these heavy alphabetic names, could we just lift the usual symbolic notations for all the operations to the various forms of DNF regexps?]

The second layer is a list of alternating strings and stars. After the choice has been made about what will be expressed, the base strings and iterated portions remains to be expressed. We keep it in a normal form, by requiring a (possibly empty) string between each clause. The clause corresponds to the concatenated data, and is a concatenation of the fixed data of base strings, and the iterated data of stars. Finally is the atom, which is a star. This corresponds to the iteration that takes place in normal stars. There is no fixed way to break the choices of this into the clause, as there is an arbitrarily large number of choices made in the stars. This intuition is formalized by the semantics, given in Figure 7.

We can create a dnf regular expression from a regular expression through repeated application of the distributivity rule, and through removal of association information. To do this, we require some functions defined on DNF regular expressions, which we define in Figure 8. From this we can define a function which converts a regular expression into an equivalent DNF regular expression, $ToDNFRegex$.

Definition 0.1.

- $ToDNFRegex(s) = [[s]]$
- $ToDNFRegex(r^*) = [[ToDNFRegex(r)^*]]$

- $ToDNFRegex(r_1 \cdot r_2) =$
  $ConcatDNF(ToDNFRegex(r_1), ToDNFRegex(r_2))$
- $ToDNFRegex(r_1|r_2) =$
  $OrDNF(ToDNFRegex(r_1), ToDNFRegex(r_2))$

We show that this transformation is a valid transformation.

Theorem 0.2 (Completeness of DNF Regexs)**.** *For all regular expressions r, $\mathcal{L}(r) = \mathcal{L}(ToDNFRegex(r))$.*

Furthermore, we can use this to prove that the language of DNF regular expressions is sound.

Theorem 0.3 (Soundness of DNF Regexs)**.** *The function $ToDNFRegex$ has a right inverse $FromDNFRegex$.*

We can further normalize the DNF regular expressions by providing an order on conjuncts, and ordering the conjuncts in the DNF regular expression according to that order. With this, we have normalized the regular expression equivalences 1-7. In other words, if two regular expressions, r_1 and r_2, are equivalent up to the transformations of regular expressions in equivalences 1-7, then $ToDNFRegex(r_1) = ToDNFRegex(r_2)$. A reasonable approach to further normalizing this is to attempt to minimize the regular expression as much as possible, using the star laws. Unfortunately, while this approach works well for normalizing, it doesn't work well for finding lenses. For example, there is clearly a lens of type $\epsilon|(a \cdot a^*) \Leftrightarrow b|(c \cdot d^*)$, namely $or(const(\epsilon, a), concat(const(a, b), iterate(const(a, d))))$. However, if one simplifies as much as possible, then they will have to find a lens of type $a^* \Leftrightarrow b|(c \cdot d^*)$. Unlike the type before being minified, where there was a type directed way to find this lens, there is no longer a type directed way: should the lens be an iterate lens or an or lens. We interpret this as the need to gain semantic information through the expansion of the regular expressions. A regular expression merely of the form a^* only cares about how the iterated case is handled, where a regular expression of the form $\epsilon|(a \cdot a^*)$ potentially acts differently on the empty string case than on the nonempty case.

Because of this, instead of writing equivalences for DNF regular expressions like exist for normal regular expressions, instead we write rewrite rules as shown in Figure 9.[B: Don't understand what's going on here.] Intuitively, these rewrite rules correspond to breaking a regular expression into a regular expression that encodes additional semantics. We define these rules as rewrites that turn atoms into DNF Regular expressions, and a rule that expresses how these rewrites can become rewrites on the DNF regular expressions themselves.

Roughly, the Atom Sumstar rewrite corresponds to Regular Expression equivalence 8, the Atom Unrollstar rewrite corresponds to Regular Expression equivalence 9, and the Atom Powerstar rewrite corresponds to Regular Expression equivalence 11. There is no rewrite corresponding to Regular Expression equivalence 10, as that equivalence is an ambiguity introducing equivalence. An application of Atom Sumstar[B: Wrong font] corresponds to doing different things after the last time a certain event has occured, and before the last time that event occurs. An application of Atom Unrollstar corresponds to making a different action for the empty case of an iteration, the first case of an iteration, and all further cases of the iteration. An application of Atom Powerstar corresponds to doing different things if the iteration is occuring a different number of times, modulo n, for some fixed n.

Armed with these rewrites, we have the capabilities to define a sufficiently strong language for lenses on these regular expressions. The syntax is defined in Figure 10. Similarly to the language of lenses, we aim to have the typing of the lenses correspond closely to the syntax for the lenses themselves. The typing and semantics of these dnf lenses are defined in Figure 11. Intuitively, a DNF Regex Lens corresponds roughly to an n-ary version of

ATOM SUMSTAR

$$\frac{A \subsetneq [1, n]}{\begin{array}{c}([co_1; \ldots; co_n])^* \Downarrow_a \\ ConcatDNF((ConcatDNF([[\epsilon; dr_A^*; \epsilon]], dr_{\bar{A}}))^*, dr_A^*) \\ \text{where } dr_S = [co_{S_1}; \ldots; co_{S_{|S|}}]\end{array}}$$

ATOM UNROLLSTAR

$$\frac{}{dr^* \Downarrow_a OrDNF([[\epsilon]], ConcatDNF(dr, [[\epsilon; dr^*; \epsilon]]))}$$

ATOM POWERSTAR

$$\frac{n \in \mathbb{N}_{\geq 1}}{\begin{array}{c}dr^* \Downarrow_a ConcatDNF([[\epsilon; [[RepeatDNF(n, dr)]]^*; \epsilon]], \\ StarModNDNF(n, dr))\end{array}}$$

DNF REWRITE STAR

$$\frac{dr \Downarrow_{dr} dr'}{dr^* \Downarrow_a [[\epsilon; dr'^*; \epsilon]]}$$

ATOM DNF REWRITE

$$\frac{a_j \Downarrow_a dr}{\begin{array}{c}[cl_1; \ldots; cl_{i-1}; \\ [s_0; a_1; \ldots; s_{j-1}; a_j; s_j; \ldots; a_m; s_m] \\ ; cl_{i+1}; \ldots; cl_n] \Downarrow_{dr} \\ OrDNF(OrDNF([cl_1; \ldots; cl_{i-1}], ConcatDNF(ConcatDNF(\\ [[s_0; a_1; \ldots; s_{j-1}]], dr), [s_j; \ldots; a_m; s_m])), [cl_{i+1}; \ldots; cl_n])\end{array}}$$

DNF REWRITE COMPOSITION

$$\frac{dr \Downarrow_{dr} dr' \qquad dr' \Downarrow_{dr} dr''}{dr \Downarrow_{dr} dr''}$$

IDENTITY DNF REWRITE

$$\frac{}{dr \Downarrow_{dr} dr}$$

Figure 9: DNF Regex Rewrite Rules

(Atom Lenses)	al	$::= Iterate(dl)$	Iterate
		$\mid identity$	Identity
(Conjunct Lenses)	col	$::= ([(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)],$ $\sigma \in S_n)$	Clause_n
(DNF Lenses)	dl	$::= ([col_1; \ldots; col_n], \sigma \in S_n)$	DNF_n

Figure 10: DNF Lens Syntax

an or lens, a Conjunct Lens corresponds to const lenses for the strings, and a combination of Concat and Swap lenses for the Atoms. Furthermore, these lenses are sufficiently strong that Compose Lenses are not necessary. Indeed, these lenses are strong enough to express everything expressible in the language of lenses.

Theorem 0.4 (Completeness of DNF Lenses)**.** *If there exists a derivation of $l : r \Leftrightarrow q \triangleright putr, putl$, then there exists a derivation of $dl : dr \Leftrightarrow dq \triangleright putr, putl$ such that $\mathcal{L}(dr) = \mathcal{L}(r)$, and $\mathcal{L}(dq) = \mathcal{L}(q)$.*

Furthermore, they only express things expressible in the language of lenses.

Theorem 0.5 (Soundness of DNF Lenses)**.** *If there exists a derivation of $dl : dr \Leftrightarrow dq \triangleright putr, putl$, then there exists a derivation of $l : r \Leftrightarrow q \triangleright putr, putl$ such that $\mathcal{L}(r) = \mathcal{L}(dr)$, and $\mathcal{L}(q) = \mathcal{L}(dq)$.*

However, these lenses are much more suited to synthesis. The vast majority of the rules have a syntax directed synthesis algorithm. Furthermore, even the rewrite rules that don't have an immediate

ITERATE ATOM LENS

$$\frac{dl : dr \Leftrightarrow dq \triangleright putr, putl \qquad \mathcal{L}(dr)^{!*} \qquad \mathcal{L}(dq)^{!*}}{\begin{array}{c} iterate(dl) : dr^* \Leftrightarrow dq^* \triangleright \\ \lambda s.\text{let } s_1 \cdot \ldots \cdot s_n = s \text{ where } s_i \in \mathcal{L}(dr) \text{ in} \\ (putr\ s_1) \cdot \ldots \cdot (putr\ s_n), \\ \lambda s.\text{let } s_1 \cdot \ldots \cdot s_n = s \text{ where } s_i \in \mathcal{L}(dq) \text{ in} \\ (putl\ s_1) \cdot \ldots \cdot (putl\ s_n) \end{array}}$$

CONJUNCT LENS

$$\frac{\begin{array}{c} al_1 : a_1 \Leftrightarrow b_1 \triangleright putr_1, putl_1 \\ \ldots \\ al_n : a_n \Leftrightarrow b_n \triangleright putr_n, putl_n \\ \sigma \in S_n \qquad \forall i \in [1, n-1] \mathcal{L}(s_{i-1}\, a_i \epsilon).!\mathcal{L}(s_i\, a_{i+1} \epsilon) \\ \forall i \in [1, n-1] \mathcal{L}(t_{i-1}\, b_{\sigma(i)} \epsilon).!\mathcal{L}(t_i\, b_{\sigma(i+1)} \epsilon) \end{array}}{\begin{array}{c} ([(s_0, t_0); a_1; \ldots; a_n; (s_n, t_n)], \lambda i : [1, n].a_i, \sigma) : \\ [s_0; a_1; \ldots; a_n; s_n] \Leftrightarrow [t_0; b_1; \ldots; b_n; t_n] \triangleright \\ \lambda s.\text{let } s_0 \cdot s_1' \cdot \ldots \cdot s_n' \cdot s_n = s \text{ where } s_i' \in \mathcal{L}(a_i) \text{ in} \\ s_0 \cdot (putr_{\sigma(1)}\ s'_{\sigma(1)}) \cdot \ldots \cdot (putr_n\ s_n') \cdot s_n, \\ \lambda s.\text{let } s_0 \cdot s_1' \cdot \ldots \cdot s_n' \cdot s_n = s \text{ where } s_i' \in \mathcal{L}(b_i) \text{ in} \\ s_0 \cdot (putl_{\sigma^{-1}(1)}\ s'_{\sigma^{-1}(1)}) \cdot \ldots \cdot (putl_{\sigma^{-1}(n)}\ s'_{\sigma^{-1}(n)}) \cdot s_n \end{array}}$$

DNF REGEX LENS

$$\frac{\begin{array}{c} col_1 : co_1 \Leftrightarrow bo_1 \triangleright putr_1, putl_1 \\ \ldots \\ col_n : co_n \Leftrightarrow bo_n \triangleright putr_n, putl_n \\ \sigma \in S_n \\ i \neq j \Rightarrow co_i \cap co_j = \emptyset \qquad i \neq j \Rightarrow bo_i \cap bo_j = \emptyset \end{array}}{\begin{array}{c} ([col_1; \ldots; col_n], \sigma) : [co_1; \ldots; co_n] \Leftrightarrow [bo_{\sigma(1)}; \ldots; bo_{\sigma(n)}] \triangleright \\ \lambda s.\{putr_1(s) \text{ if } s \in \mathcal{L}(co_1), \ldots, putr_n(s) \text{ if } s \in \mathcal{L}(co_n)\}, \\ \lambda s.\{putl_1(s) \text{ if } s \in \mathcal{L}(bo_1), \ldots, putl_n(s) \text{ if } s \in \mathcal{L}(bo_n)\} \end{array}}$$

DNF REWRITE LENS

$$\frac{dr \Downarrow_{dr} dr' \qquad dq \Downarrow_{dr} dq' \qquad dl : dr \Leftrightarrow dq \triangleright putr, putl}{dl : dr' \Leftrightarrow dq'}$$

Figure 11: DNF Lens Typing and Semantics

syntax directed synthesis algorithm have an underlying semantic meaning which can be used to direct the solution.

Acknowledgments

References

[1] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

A. Proofs

Lemma A.1. $\mathcal{L}(\Delta)\,ConcatDNF(dr, dq) = \{s \cdot t | s \in \mathcal{L}(\Delta)\,dr \wedge \mathcal{L}(\Delta)\,dq\}$

Proof. □

Theorem 0.2 (Completeness of DNF Regexs)**.** *For all regular expressions r, $\mathcal{L}(r) = \mathcal{L}(\,ToDNFRegex(r))$.*

Proof.

Definition A.2.

- $ToDNFRegex(s) = [[s]]$
- $ToDNFRegex(U) = [[\epsilon; U; \epsilon]]$
- $ToDNFRegex(r \cdot q) = ConcatDNF(\,ToDNFRegex(r),\, ToDNFRegex(q))$
- $ToDNFRegex(r|q) = OrDNF(\,ToDNFRegex(r),\, ToDNFRegex(q))$
- $ToDNFRegex(r^*) = [[\epsilon;\, ToDNFRegex(r)^*; \epsilon]]$

We do be structural induction. □

Theorem 0.3 (Soundness of DNF Regexs)**.** *The function $ToDNFRegex$ has a right inverse $FromDNFRegex$.*

Proof. pf of soundness of dnf regexs □

A.1 DNF Lens Completeness

Theorem 0.4 (Completeness of DNF Lenses)**.** *If there exists a derivation of $l : r \Leftrightarrow q \triangleright putr, putl$, then there exists a derivation of $dl : dr \Leftrightarrow dq \triangleright putr, putl$ such that $\mathcal{L}(dr) = \mathcal{L}(r)$, and $\mathcal{L}(dq) = \mathcal{L}(q)$.*

A.2 DNF Lens Soundness

Lemma A.3 (Creation of Lens from Identity Perm Clause Lens)**.** *If $\Delta \vdash ([(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)], id) : [s_0; a_1; \ldots; a_n; s_n] \Leftrightarrow [t_0; b_1; \ldots; b_n; t_n]$, and for each $al_i : a_i \Leftrightarrow b_i$, there exists a $\Delta \vdash l_i : r_i \Leftrightarrow q_i$ such that $\mathcal{L}(\Delta)a_i = \mathcal{L}(\Delta)r_i$, and $l_i.putr = cll_i.putr$, then there exists a $l : r \Leftrightarrow q$ such that $l.putr = ([al_1; \ldots; al_n], id).putr$, $\mathcal{L}(\Delta)[s_0; a_1; \ldots; cl_n; s_n]$ = $\mathcal{L}(\Delta)r$, and $\mathcal{L}(\Delta)[t_0; b_1; \ldots; b_n; t_n]$ = $\mathcal{L}(\Delta)q$.*

Proof. Base case: $\Delta, \Gamma \vdash [(s_0, t_0)] : [s_0] \Leftrightarrow [t_0]$. Then consider l = $const(s_0, t_0)$, $r = s$, $q = t$. $\mathcal{L}(\Delta)[s_0] = s_0 = \mathcal{L}(\Delta)s_0$ by definition.
$\mathcal{L}(\Delta)[t_0] = t_0 = \mathcal{L}(\Delta)s_0$ by definition.
$[(s_0, t_0)].putr(s_0) = t_0 = const(s, t).putr(s_0)$.

By induction assumption, I know there exists a $l : r \Leftrightarrow q$ such that $l.putr = ([(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)], id).putr$, $\mathcal{L}(\Delta)[s_0; a_1; \ldots; a_n; s_n]$ = $\mathcal{L}(\Delta)r$, and $\mathcal{L}(\Delta)[t_0; b_1; \ldots; a_n; t_n]$ = $\mathcal{L}(\Delta)q$.
By assumption, I know that there exists a $\Delta \vdash l_{n+1} : r_{n+1} \Leftrightarrow q_{n+1}$ such that $\mathcal{L}(\Delta)r_{n+1} = \mathcal{L}(\Delta)a_{n+1}$ and $\mathcal{L}(\Delta)q_{n+1} = \mathcal{L}(\Delta)b_{n+1}$, where $l_{n+1}.putr = al_{n+1}$.
Consider the lens $\Delta, \Gamma \vdash concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))) : r \cdot r_{n+1} \cdot s_{n+1} \Leftrightarrow q \cdot q_{n+1} \cdot t_{n+1}$.
$\mathcal{L}(\Delta)r \cdot r_{n+1} \cdot s_{n+1} = \mathcal{L}(\Delta)r \cdot \mathcal{L}(\Delta)r_{n+1} \cdot \{s_{n+1}\}$ by definition
$\mathcal{L}(\Delta)r \cdot r_{n+1} \cdot s_{n+1} = \mathcal{L}(\Delta)[s_0; a_1; \ldots; a_n; s_n] \cdot \mathcal{L}(\Delta)r_{n+1} \cdot \{s_{n+1}\}$ by induction assumption.
$\mathcal{L}(\Delta)r \cdot r_{n+1} \cdot s_{n+1} = \{s_0\} \cdot \mathcal{L}(\Delta)a_1 \cdot \ldots \cdot \mathcal{L}(\Delta)a_n \cdot \{s_n\} \cdot \mathcal{L}(\Delta)r_{n+1} \cdot \{s_{n+1}\}$ by definition.
$\mathcal{L}(\Delta)r \cdot r_{n+1} \cdot s_{n+1} = \{s_0\} \cdot \mathcal{L}(\Delta)a_1 \cdot \ldots \cdot \mathcal{L}(\Delta)a_n \cdot \{s_n\} \cdot \mathcal{L}(\Delta)a_{n+1} \cdot \{s_{n+1}\}$ by problem assumption.
$\mathcal{L}(\Delta)r \cdot r_{n+1} \cdot s_{n+1} = \mathcal{L}(\Delta)[s_0; a_0; \ldots; a_{n+1}; s_{n+1}]$ by definition.

Similarly for $\mathcal{L}(\Delta)q \cdot q_{n+1} \cdot t_{n+1} = \mathcal{L}(\Delta)[t_0; b_0; \ldots; b_{n+1}; t_{n+1}]$ TODO, split into lemma?.

$concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))).putr(s_0 s'_1 \ldots s'_{n+1} s_{n+1}) =$
$l.putr(s_0 s'_1 \ldots s'_n s_n) concat(l_{n+1}, const(s_{n+1}, t_{n+1})).putr(s'_{n+1} s_{n+1})$ by definition

$concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))).putr(s_0 s'_1 \ldots s'_{n+1} s_{n+1}) =$
$l.putr(s_0 s'_1 \ldots s'_n s_n) l_{n+1}.putr(s'_{n+1}) const(s_{n+1}, t_{n+1}).putr(s_{n+1})$ by definition

$concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))).putr(s_0 s'_1 \ldots s'_{n+1} s_{n+1}) =$
$[(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)].putr(s_0 s'_1 \ldots s'_n s_n) l_{n+1}.putr(s'_{n+1}) const(s_{n+1}, t_{n+1}).putr(s_{n+1})$ by induction assumption

$concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))).putr(s_0 s'_1 \ldots s'_{n+1} s_{n+1}) =$
$[(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)].putr(s_0 s'_1 \ldots s'_n s_n) al_{n+1}.putr(s'_{n+1}) const(s_{n+1}, t_{n+1}).putr(s_{n+1})$ by problem assumption

$concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))).putr(s_0 s'_1 \ldots s'_{n+1} s_{n+1}) =$
$t_0 \cdot al_1.putr(s'_1) \cdot \ldots \cdot al_n.putr(s'_n) \cdot t_n \cdot al_{n+1}.putr(s'_{n+1}) \cdot t_{n+1}$ by definition

$concat(l, concat(l_{n+1}, const(s_{n+1}, t_{n+1}))).putr(s_0 s'_1 \ldots s'_{n+1} s_{n+1}) =$
$[(s_0, t_0); al_0; \ldots; al_{n+1}; (s_{n+1}, t_{n+1})].putr(s_0 s'_0 \ldots s'_{n+1} s_{n+1})$ by definition □

Definition A.4 (Adjacent Swapping Permutation). *Let $\sigma_i \in S_n$ be the permutation where $\sigma_i(i) = i+1$, $\sigma_i(i+1) = i$, $\sigma_{i,j}(k \neq i, i+1) = k$*

Definition A.5 (Clause Permuting Bijection). *Let $[s_0; a_1; \ldots; a_n; s_n]$ be a clause, let σ be a permutation, and let Δ be a regular expression context. Define $f_{\sigma,\Delta,[s_0;a_1;\ldots;a_n;s_n]}$ as: $f_{\sigma,\Delta,[s_0;a_1;\ldots;a_n;s_n]} : \mathcal{L}(\Delta)[s_0; a_1; \ldots; a_n; s_n] \to \mathcal{L}(\Delta)[s_0; a_{\sigma(1)}; \ldots; a_{\sigma(n)}; s_n]$, where $s_0 s'_1 \ldots s'_n s_n \mapsto s_0 s'_{\sigma(1)} \ldots s'_{\sigma(n)} s_n$.*

Lemma A.6 (Expressibility of Adjacent Swapping Permutation Lens). *Let σ_i be an adjacent element swapping permutation, $[s_0; a_1; \ldots; a_n; s_n]$ be a clause, Δ be a regular expression context, and Γ be a lens context. There exists regular expressions r and q, and a lens l, such that $\mathcal{L}(\Delta)r = \mathcal{L}(\Delta)[s_0; a_1; \ldots; a_n; s_n]$, $\mathcal{L}(\Delta)q = \mathcal{L}(\Delta)[s_0; a_{\sigma(1)}; \ldots; a_{\sigma(n)}; s_n]$, and $\Delta, \Gamma \vdash l : r \Leftrightarrow q \triangleright f_{\sigma,\Delta,[s_0;a_1;\ldots;a_n;s_n]}, f^{-1}_{\sigma,\Delta,[s_0;a_1;\ldots;a_n;s_n]}$*

Proof. By the soundness of regular expressions, there exists a regular expressions r_1, r_2, r_3, r_4 such that $\mathcal{L}(\Delta)r_1 = \mathcal{L}(\Delta)[s_1; a_1; \ldots; a_{i-1}; s_{i-1}]$, $\mathcal{L}(\Delta)r_2 = \mathcal{L}(\Delta)a_i$, $\mathcal{L}(\Delta)r_3 = \mathcal{L}(\Delta)a_{i+1}$, and $\mathcal{L}(\Delta)r_4 = \mathcal{L}(\Delta)[s_i; a_{i+1}; \ldots; a_n; s_n]$. Consider the following deduction

$$\dfrac{\dfrac{\Delta, \Gamma \vdash identity : r_1 \Leftrightarrow r_1 \triangleright id, id \quad \dfrac{\dfrac{}{\Delta, \Gamma \vdash l : r_2 \Leftrightarrow r_2 \triangleright id, id} \quad \dfrac{\dfrac{}{\Delta, \Gamma \vdash l : s_i \Leftrightarrow s_i \triangleright id, id} \quad \dfrac{}{\Delta, \Gamma \vdash l : r_3 \Leftrightarrow r_3 \triangleright id, id}}{\Delta, \Gamma \vdash l : s_i \cdot r_3 \Leftrightarrow r_3 \cdot s_i \triangleright swap(id, id), swap(id, id)}}{\Delta, \Gamma \vdash l : r_2 \cdot s_i \cdot r_3 \Leftrightarrow r_3 \cdot s_i \cdot r_2 \triangleright id, id^{-1}}}{\Delta, \Gamma \vdash l : r_1 \cdot r_2 \cdot s_i \cdot r_3 \Leftrightarrow r_1 \cdot r_3 \cdot s_i \cdot r_2 \triangleright id, id^{-1}} \quad \dfrac{}{\Delta, \Gamma \vdash l : r_4 \Leftrightarrow r_4 \triangleright id, id}}{\Delta, \Gamma \vdash l : r_1 \cdot r_2 \cdot s_i \cdot r_3 \cdot r_4 \Leftrightarrow r_1 \cdot r_3 \cdot s_i \cdot r_2 \cdot r_4 \triangleright f_{\sigma,\Delta,[s_0;a_1;\ldots;a_n;s_n]}, f^{-1}_{\sigma,\Delta,[s_0;a_1;\ldots;a_n;s_n]}}$$

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 $repregex([s_{1,2}; a_{1,2}; \ldots; s_{i,1}])$ $((a_{i+1,1} s_{i+1,1}) a_{i,1})$ $repregex([s_{i+2,1}; \ldots; a_n; s_{n+1}])$ 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 A.7 (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.

□

Lemma A.8 (Creation of Lens from Identity Perm DNF Lens). *If $\Delta \vdash ([cll_1; \ldots; cll_n], id) : [cl_1; \ldots; cl_n] \Leftrightarrow [bl_1; \ldots; bl_n]$, and for each $cll_i : cl_i \Leftrightarrow bl_i$, there exists a $\Delta \vdash l_i : r_i \Leftrightarrow q_i$ such that $\mathcal{L}(\Delta)cl_i = \mathcal{L}(\Delta)r_i$, and $l_i.putr = cll_i.putr$, then there exists a $l : r \Leftrightarrow q$ such that $l.putr = ([cll_1; \ldots; cll_n], id).putr$, $\mathcal{L}(\Delta)[cl_1; \ldots; cl_n] = \mathcal{L}(\Delta)r$, and $\mathcal{L}(\Delta)[bl_1; \ldots; bl_n] = \mathcal{L}(\Delta)q$.*

Proof. Base case: $\Delta, \Gamma \vdash [cll_1] : [cl_1] \Leftrightarrow [bl_1]$. Then consider $l = l_1$, $r = r_1$, $q = q_1$. $\mathcal{L}(\Delta)[cl_1] = \mathcal{L}(\Delta)cl_1$ by definition.
$\mathcal{L}(\Delta)[cl_1] = \mathcal{L}(\Delta)r_1$ by problem assumption.
Symmetrically for $\mathcal{L}(\Delta)[bl_1]$.
$[cll].putr = cll.putr$ by definition
$[cll].putr = l_1.putr$ by problem assumption.

By induction assumption, I know there exists a $l : r \Leftrightarrow q$ such that $l.putr = ([cll_1; \ldots; cll_n], id).putr$, $\mathcal{L}(\Delta)[cl_1; \ldots; cl_n] = \mathcal{L}(\Delta)r$, and $\mathcal{L}(\Delta)[bl_1; \ldots; bl_n = \mathcal{L}(\Delta)q$.
By assumption, I know that there exists a $\Delta \vdash l_{n+1} : r_{n+1} \Leftrightarrow q_{n+1}$ such that $\mathcal{L}(\Delta)r_{n+1} = \mathcal{L}(\Delta)cl_{n+1}$ and $\mathcal{L}(\Delta)q_{n+1} = \mathcal{L}(\Delta)bl_{n+1}$, where $l_{n+1}.putr = cll_{n+1}$.
Consider the lens $\Delta, \Gamma \vdash or(l, l_{n+1}) : r|r_{n+1} \Leftrightarrow q|q_{n+1}$.
$\mathcal{L}(\Delta)r|r_{n+1} = \mathcal{L}(\Delta)r \cup \mathcal{L}(\Delta)r_{n+1}$ by definition
$\mathcal{L}(\Delta)r|r_{n+1} = \mathcal{L}(\Delta)[cl_1; \ldots; cl_n] \cup \mathcal{L}(\Delta)r_{n+1}$ by induction assumption.
$\mathcal{L}(\Delta)r|r_{n+1} = \bigcup_{i=1\ldots n} \mathcal{L}(\Delta)cl_i \cup \mathcal{L}(\Delta)r_{n+1}$ by definition.
$\mathcal{L}(\Delta)r|r_{n+1} = \bigcup_{i=1\ldots n} \mathcal{L}(\Delta)cl_i \cup \mathcal{L}(\Delta)cl_{n+1}$ by problem assumption.
$\mathcal{L}(\Delta)r|r_{n+1} = \bigcup_{i=1\ldots n+1} \mathcal{L}(\Delta)cl_i$ by distributivity of $\cup$.
$\mathcal{L}(\Delta)r|r_{n+1} = \mathcal{L}(\Delta)[cl_1; \ldots; cl_n]$ by definition.
Similarly for $\mathcal{L}(\Delta)q|q_{n+1} = \mathcal{L}(\Delta)[bl_1; \ldots; bl_n]$ TODO, split into lemma?.

$$or(l, l_{n+1}).putr(s) = \begin{cases} l.putr(s) & \text{if } s \in \mathcal{L}(\Delta)r \\ l_{n+1}.putr(s) & \text{if } s \in \mathcal{L}(\Delta)r_{n+1} \end{cases}$$

by definition.

$$or(l, l_{n+1}).putr(s) = \begin{cases} [cll_1; \ldots; cll_n.putr(s) & \text{if } s \in \mathcal{L}([cl_1; \ldots; cl_n])r \\ l_{n+1}.putr(s) & \text{if } s \in \mathcal{L}(\Delta)r_{n+1} \end{cases}$$

by induction assumption.

$$or(l, l_{n+1}).putr(s) = \begin{cases} cll_1.putr(s) & \text{if } s \in \mathcal{L}(\Delta)cl_1 \\ \ldots & \\ cll_n(s).putr & \text{if } s \in \mathcal{L}(\Delta)cl_n \\ l_{n+1}.putr(s) & \text{if } s \in \mathcal{L}(\Delta)r_{n+1} \end{cases}$$

by definition of putr.

$$or(l, l_{n+1}).putr(s) = \begin{cases} cll_1.putr(s) & \text{if } s \in \mathcal{L}(\Delta)cl_1 \\ \ldots & \\ cll_n(s).putr & \text{if } s \in \mathcal{L}(\Delta)cl_n \\ cl_{n+1}.putr(s) & \text{if } s \in \mathcal{L}(\Delta)cl_{n+1} \end{cases}$$

by problem assumption.
$or(l, l_{n+1}).putr(s) = [cll_1; \ldots; cll_{n+1}].putr(s)$ by definition of putr.

□

Lemma A.9 (Soundness of DNF Lenses, Clauses, and Atoms)**.**

1. *Let dr and dq be two dnf regular expressions, and $\Delta, \Gamma \vdash dl : dr \Leftrightarrow dq$. Then there exists a l, r, and q such that $\Delta, \Gamma \vdash l : r \Leftrightarrow q$, $l.putr$=$dl.putr$, $\mathcal{L}(\Delta)r$=$\mathcal{L}(\Delta)dr$, and $\mathcal{L}(\Delta)q$=$\mathcal{L}(\Delta)dq$. $\mathcal{L}(\Delta)r$ = $\mathcal{L}(\Delta)dr$ and $\mathcal{L}(\Delta)q$ = $\mathcal{L}(\Delta)dq$*
2. *Let cl and bl be two clauses, and $\Delta, \Gamma \vdash cll : cl \Leftrightarrow bl$. Then there exists a l, r, and q such that $\Delta, \Gamma \vdash l : r \Leftrightarrow q$, $l.putr$=$cll.putr$, $\mathcal{L}(\Delta)r$=$\mathcal{L}(\Delta)cl$, and $\mathcal{L}(\Delta)q$=$\mathcal{L}(\Delta)bl$. $\mathcal{L}(\Delta)r$ = $\mathcal{L}(\Delta)cl$ and $\mathcal{L}(\Delta)q$ = $\mathcal{L}(\Delta)bl$*
3. *Let a and b be two atoms, and $\Delta, \Gamma \vdash al : a \Leftrightarrow b$. Then there exists a l, r, and q such that $\Delta, \Gamma \vdash l : r \Leftrightarrow q$, $l.putr$=$al.putr$, $\mathcal{L}(\Delta)r$=$\mathcal{L}(\Delta)a$, and $\mathcal{L}(\Delta)q$=$\mathcal{L}(\Delta)b$. $\mathcal{L}(\Delta)r$ = $\mathcal{L}(\Delta)a$ and $\mathcal{L}(\Delta)q$ = $\mathcal{L}(\Delta)b$*

   Proof.

   $\Delta, \Gamma \vdash ([cll_1; \ldots; cll_n], \sigma) : [cl_1; \ldots; cl_n] \Leftrightarrow [bl_{\sigma(1)}; \ldots; bl_{\sigma(n)}]$
   By Induction assumption, for each $\Delta, \Gamma \vdash cll_i : cl_i \Leftrightarrow bl_i$ there exists a $\Delta, \Gamma \vdash l_i : r_i \Leftrightarrow q_i$.
   By Lemma A.8 there exists a $\Delta, \Gamma \vdash l : r \Leftrightarrow q$ such that $l.putr = ([cll_1; \ldots cll_n], id).putr$, $\mathcal{L}(\Delta)r = \mathcal{L}(\Delta)[cl_1; \ldots; cl_n]$, and $\mathcal{L}(\Delta)q = \mathcal{L}(\Delta)[bl_1; \ldots; bl_n]$.
   By Lemma TODO, $([cll_1; \ldots cll_n], id).putr = ([cll_1; \ldots cll_n], \sigma).putr$.
   By Lemma TODO, $\mathcal{L}(\Delta)[bl_1; \ldots; bl_n] = \mathcal{L}(\Delta)[bl_{\sigma(1)}; \ldots; bl_{\sigma(n)}]$.
   As such, $l.putr = ([cll_1; \ldots cll_n], \sigma).putr$, $\mathcal{L}(\Delta)r = \mathcal{L}(\Delta)[cl_1; \ldots; cl_n]$, and $\mathcal{L}(\Delta)q = \mathcal{L}(\Delta)[bl_{\sigma(1)}; \ldots; bl_{\sigma(n)}]$.

- $\Delta, \Gamma \vdash ([(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)], \sigma \in S_n) : [s_0; a_1; \ldots; a_n; s_n] \Leftrightarrow [t_0; b_{\sigma(1)}; \ldots; b_{\sigma(n)}; t_n]$
  From induction assumption, I know for each $\Delta, \Gamma \vdash al_i : a_i \Leftrightarrow b_i$, there exists a $\Delta, \Gamma \vdash l_i : r_i \Leftrightarrow q_i$ such that $al_i.putr = l_i.putr$, and $\mathcal{L}(\Delta)r_i = \mathcal{L}(\Delta)a_i$ and $\mathcal{L}(\Delta)q_i = \mathcal{L}(\Delta)b_i$.
  By Lemma A.3, there exists a $\Delta, \Gamma \vdash l : r \Leftrightarrow q$ such that $l.putr = ([(s_0, t_0); al_1; \ldots; al_n; (s_n, t_n)], id).putr$, $\mathcal{L}(\Delta)r = \mathcal{L}(\Delta)[s_0; cl_1; \ldots; cl_n; s_{n+1}]$, $\mathcal{L}(\Delta)q = \mathcal{L}(\Delta)[t_0; bl_1; \ldots; bl_n; t_{n+1}]$.
  By Lemma A.7 there exists a $\Delta, \Gamma \vdash l_\sigma : r' \Leftrightarrow q'$, such that $l_\sigma.putr(t_0; t'_1; t_1; \ldots; t'_n t_n) = t_0; t'_{\sigma(1)}; t_1; \ldots; t'_{\sigma(n)} t_n$, $\mathcal{L}(\Delta)r' = \mathcal{L}(\Delta)[s_0; cl_1; \ldots; cl_n; s_{n+1}]$, and $\mathcal{L}(\Delta)q' = \mathcal{L}(\Delta)[s_0; cl_{\sigma(1)}; \ldots; cl_{\sigma(n)} s_{n+1}]$.
  Consider the lens $\Delta \vdash l_\sigma \circ l : r \Leftrightarrow q'$.
  $l_\sigma \circ l.putr(s_0 s'_1 \ldots s'_n s_n) = l_\sigma.putr(l.putr(s_0 s'_1 \ldots s'_n s_n))$ by definition of compose
  $l_\sigma \circ l.putr(s_0 s'_1 \ldots s'_n s_n) = l_\sigma.putr(t_1 al_1.putr(s'_1) \ldots al_n.putr(s'_n) t_n)$ by definition of l
  $l_\sigma \circ l.putr(s_0 s'_1 \ldots s'_n s_n) = t_1 al_{\sigma(1)}.putr(s'_{\sigma(1)}) \ldots al_{\sigma(n)}.putr(s'_{\sigma(n)}) t_n$ by definition of l_σ
  $l_\sigma \circ l.putr(s_0 s'_1 \ldots s'_n s_n) = ([(s_0, t_0); al_0; \ldots; al_n; (s_n, t_n)], \sigma)(s_0 s'_1 \ldots s'_n s_n)$ by definition of clause lens

- $\Delta, \Gamma \vdash iterate(dl) : dr^* \Leftrightarrow dq^*$
  From induction assumption, I know that there exists $\Delta, \Gamma \vdash l : r \Leftrightarrow q$, such that $dl.putr = l.putr$, $\mathcal{L}(\Delta)r$=$\mathcal{L}(\Delta)dr$, and $\mathcal{L}(\Delta)q = \mathcal{L}(\Delta)dq$.
  Consider $\Delta, \Gamma \vdash iterate(l) : r^* \Leftrightarrow q^*$.
  $\mathcal{L}(\Delta)r^* = s_0 \ldots s_n$ such that $s_i \in \mathcal{L}(\Delta)r$ by definition of $\mathcal{L}(\Delta)r^*$.
  $\mathcal{L}(\Delta)r^* = s_0 \ldots s_n$ such that $s_i \in \mathcal{L}(\Delta)dr$ as $\mathcal{L}(\Delta)dr = \mathcal{L}(\Delta)r$.
  $\mathcal{L}(\Delta)r^* = \mathcal{L}(\Delta)dr^*$ by definition of $\mathcal{L}(\Delta)dr^*$.
  Similarly for $\mathcal{L}(\Delta)q^*$

$iterate(l).putr(s_0 \dots s_n) = l.putr(s_0) \dots l.putr(s_n)$ by definition of $iterate(l).putr$.
$iterate(l).putr(s_0 \dots s_n) = dl.putr(s_0) \dots dl.putr(s_n)$ as $dl.putr$=$l.putr$.
$iterate(l).putr(s_0 \dots s_n) = iterate(dl).putr(s_0 \dots s_n)$ by definition of $iterate(dl).putr$.

- $\Delta, \Gamma \vdash identity : U \Leftrightarrow U$
  Consider $\Delta, \Gamma \vdash identity : U \Leftrightarrow U$
  $identity.putr(s) = s$ by definition of dnf $identity.putr$.
  $identity.putr(s) = identity.putr(s)$ by definition of normal $identity.putr$.

□

Theorem 0.5 (Soundness of DNF Lenses)**.** *If there exists a derivation of* $dl : dr \Leftrightarrow dq \triangleright putr, putl$*, then there exists a derivation of* $l : r \Leftrightarrow q \triangleright putr, putl$ *such that* $\mathcal{L}(r) = \mathcal{L}(dr)$*, and* $\mathcal{L}(q) = \mathcal{L}(dq)$*.*

Proof. By mutual induction on the typing dl, the typing of cll, and the typing of al.

DNF Lens Intro Let $\Delta \vdash ([cll_1; \dots; cll_n], \sigma) : [cl_{1,1}, \dots, cl_{n,1}] \Leftrightarrow [cl_{\sigma(1),2}, \dots, cl_{\sigma(n),2}]$. This comes from the derivations that $\Delta \vdash cll_i : cl_{i,1} \Leftrightarrow cl_{i,2}$ for all i. Consider instead the lens $dl' = \Delta \vdash ([cll_1; \dots; cll_n], \sigma_i d) : [cl_{1,1}, \dots, cl_{n,1}] \Leftrightarrow [cl_{1,2}, \dots, cl_{n,2}]$. 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 $\Delta \vdash ([(s_{1,1}, s_{1,2}); al_1; \dots; al_n; (s_{n+1,1}, s_{n+1,2})], \sigma) : [s_{1,1}; a_{1,1}; \dots; a_{1,n}; s_{1,n+1}] \Leftrightarrow [s_{1,2}; a_{\sigma(1),2}; \dots; a_{\sigma(n),2}; s_{n+1,2}]$. Consider two lenses, $\Delta \vdash ([(s_{1,1}, s_{1,2}); al_1; \dots; al_n; (s_{n+1,1}, s_{n+1,2})], \sigma_i d) : [s_{1,1}; a_{1,1}; \dots; a_{1,n}; s_{1,n+1}] \Leftrightarrow [s_{1,2}; a_{1,2}; \dots; a_{n,2}; s_{n+1,2}]$, and $\Delta \vdash ([(s_{1,2}, s_{1,2}); identitylens(a_{1,2}); \dots; identitylens(a_{n,2}); (s_{n+1,2}, s_{n+1,2})], \sigma) : [s_{1,2}; a_{1,2}; \dots; a_{n,2}; s_{n+1,2}] \Leftrightarrow [s_{1,2}; a_{\sigma(1),2}; \dots; a_{\sigma(n),2}; s_{n+1}$
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.

□