An Isomorphism Between Substructural Proofs and Deterministic Finite Automata*

Henry DeYoung and Frank Pfenning

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA USA
{hdeyoung,fp}@cs.cmu.edu

Abstract

Proof theory is a powerful tool for understanding computational phenomena, as most famously exemplified by the Curry–Howard isomorphism between intuitionistic logic and the simply-typed λ-calculus. In this paper, we identify a fragment of intuitionistic linear logic with least fixed points and establish a Curry–Howard isomorphism between a class of proofs in this fragment and deterministic finite automata. Proof-theoretically, closure of regular languages under complementation, union, and intersection can then be understood in terms of cut elimination. We also establish an isomorphism between a different class of proofs and subsequential string transducers. Because prior work has shown that linear proofs can be seen as session-typed processes, a concurrent semantics of transducer composition is obtained for free.

1998 ACM Subject Classification F.4.1 Mathematical Logic; F.1.1 Models of Computation

Keywords and phrases linear logic, deterministic finite automata, concurrent processes, session types, subsequential transducers

Digital Object Identifier 10.4230/LIPIcs.CSL.2016.0

1 Introduction

Proof theory is a powerful tool for understanding computational phenomena, as most famously exemplified by the Curry–Howard isomorphism between intuitionistic logic and the simply-typed λ-calculus [12]. Under this isomorphism, propositions are types, proofs are well-typed programs, and proof normalization is computation. Another example is the recently discovered isomorphism [3, 4] between intuitionistic linear logic [9, 5] and the session-typed π-calculus [10]. Here, linear propositions are session types, sequent calculus proofs are well-typed concurrent processes, and proof reductions during cut elimination effect interprocess communication.

Stepping outside programming calculi, deterministic finite automata (DFAs) [16, 11] are a mathematical model of very basic computation. When a DFA is fed a finite string of symbols as input, it changes state as it reads each symbol and, once it has finished reading, returns a Boolean answer — either *accept* or *reject*. Given the remarkable successes in using proof theory to understand computational phenomena, it's quite natural to wonder if a similar story can be told for DFAs.

More specifically, because the intuitive description of a DFA's computational behavior sounds a bit like a process that realizes a Boolean-valued function on finite strings, is there a Curry–Howard isomorphism for DFAs that is somehow related to the aforementioned isomorphism between intuitionistic linear logic and the session-typed π-calculus?

Yes, indeed there is, as this paper demonstrates. The contributions are threefold:

* This work was partially supported by someone.

Proofs as DFAs. In the fragment of intuitionistic linear logic with least fixed points described in Section 2, we can define propositions `String` and `Ans` that characterize finite strings and Boolean answers, respectively. Under the isomorphism with the session-typed π-calculus, these propositions become session types of processes. Section 4.2 gives an encoding of DFAs as processes of type $\texttt{String} \vdash \texttt{Ans}$ in which DFA transitions are matched by process reductions. By proving in Section 4.3 the converse — that every process of type $\texttt{String} \vdash \texttt{Ans}$ corresponds to a DFA — we establish an isomorphism between DFAs and processes (which are proofs) of type $\texttt{String} \vdash \texttt{Ans}$.

Closure properties by cut elimination. As one would hope, this isomorphism allows logical ideas to be leveraged into insight about DFAs. Section 5 demonstrates that the closure of regular languages under complementation, intersection, and union can be understood proof-theoretically in terms of cut elimination. In fact, cut elimination can be seen as *constructing* the very same DFAs traditionally used in proving those closure properties.

Proofs as string transducers. String transducers, specifically subsequential string transducers [15, 13], generalize DFAs by producing a finite string, rather than Boolean answer, as output. Section 6 establishes an isomorphism between subsequential transducers and processes of type $\texttt{String} \vdash \texttt{String}$. With this isomorphism, a concurrent semantics of transducer composition comes for free (Section 7).

2 Linear logic proofs as session-typed processes

Recently, a Curry–Howard isomorphism between intuitionistic linear logic [9, 5] and the session-typed π-calculus [10] has been discovered [3, 4]: Propositions are session types, proofs are well-typed processes, and cut reductions effect interprocess communication. In this section, we present a fragment of intuitionistic linear logic and give its interpretation as a session-typing discipline for concurrent processes. Our treatment of corecursive process definitions will somewhat differ from previous work, but the underlying ideas are the same.

In session-based concurrency, a process provides a service along a designated channel, perhaps also using services provided by other processes. Therefore, the basic session-typing judgment is the linear hypothetical judgment

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash P :: x{:}A\,,$$

meaning "Using services A_i that are assumed to be offered along channels x_i, process P provides service A along channel x." (The channels x_i and x must all be distinct and are binding occurrences with scope over the process P.) This basic session-typing judgment can be seen as an annotation of an intuitionistic linear sequent.

The services that a process uses and provides are described by session types A. For our fragment, the session types are the closed propositions generated by the grammar

$$A ::= \oplus_{\ell \in L}\{\ell{:}A_\ell\} \mid A_1 \otimes A_2 \mid \mathbf{1} \mid \mu X.A \mid X\,.$$

$\oplus_{\ell \in L}\{\ell{:}A_\ell\}$ is a labeled n-ary additive disjunction over a set L of labels; $A_1 \otimes A_2$ and $\mathbf{1}$ are multiplicative conjunction and its unit; $\mu X.A$ is the least fixed point of the equation $X = A$.

To handle mutually corecursive process definitions, we introduce a signature, Θ, of well-typed definitions $X = P$. We revise the session-typing judgment, indexing it by the signature: $x_1{:}A_1, \ldots, x_n{:}A_n \vdash_\Theta Q :: x{:}A$. The body P of a definition X may refer to X or any other process variable defined in Θ. We require that these definitions are productive in the sense that a definition's body may not immediately call another process variable Y, thereby ruling out definitions $X = Y$.

The session-typing rules for our fragment of interest are shown in Fig. 1. The judgmental rules CUT and ID serve to clarify the relationship between providing and using a service, whereas the left and right introduction rules for $\oplus$, $\otimes$, and $\mathbf{1}$ serve as session-typing rules that define those service combinators.

Judgmental rules. Just as the sequent calculus CUT rule composes a proof of lemma A with its use in a proof of theorem C, the corresponding session-typing rule becomes a typing rule for process composition: the $P \triangleright Q$ process composes a server P with its client Q. Dually, just as the sequent calculus ID rule says that a lemma A is enough to prove theorem A, the corresponding session-typing rule types a process that forwards communication between two channels. The PVAR rule uses a definition from the signature.

Additive disjunction as internal choice. With the type $\oplus_{\ell \in L}\{\ell{:}A_\ell\}$, a server can provide its choice of the possible services A_ℓ to the client. In the right rule ($\oplus$R), the server sends its selection, k, to the client and then continues by providing service A_k. In the matching left rule ($\oplus$L), the client must be prepared for any selection $\ell \in L$; the client process behaves like a `case`, waiting to receive the selection and then continuing accordingly.

Multiplicative conjunction as output. With the type $A_1 \otimes A_2$, a server sends the client the channel of a process that provides service A_1, and then continues by providing A_2 ($\otimes$R rule). The client waits to receive that channel and then continues as process Q ($\otimes$L rule).

Multiplicative unit as termination. Because $\mathbf{1}$ is the nullary form of $\otimes$, a server providing $\mathbf{1}$ neither sends a channel nor continues. Instead, it terminates communication ($\mathbf{1}$R rule). The client waits for the server to terminate and then continues as process Q ($\mathbf{1}$L rule).

Least fixed points as inductive types. Rather than giving right and left introduction rules for μ, we treat inductive types *equi-recursively*: we identify $\mu X.A$ with its unfolding $[(\mu X.A)/X]A$ and allow this unfolding to occur silently at any point in a typing derivation. Since all connectives in this fragment are covariant, inductive types satisfy strict positivity. We also require that inductive types are contractive [8], ruling out, for example, $\mu X.X$.

Proofs, or typing derivations, are built from the rules in Fig. 1, with process variables X to indicate circularities that make the proofs coinductive. Following Fortier and Santocanale [7], such proofs are well-defined if, within each cycle, a left rule is applied to an inductive type.

▶ **Example 1.** A process that emits the unary representation of a natural number can be described by the session type $\mathit{Nat} = \mu X.\, \oplus\{\${:}\mathbf{1}, s{:}X\}$, so that a natural number is a string of ss terminated by `$`. For instance, the process $\texttt{select}\, y\, s; \texttt{select}\, y\, \$; \texttt{close}\, y$ emits the unary representation of 1 by emitting s and `$` along channel y and then terminating communication.

The doubling function on natural numbers can then be expressed as the process of type $x{:}\mathit{Nat} \vdash y{:}\mathit{Nat}$ given by the following corecursive definition.

$$y \leftarrow \mathit{dbl} \leftarrow x = y.\texttt{case}(\$ \Rightarrow \texttt{wait}\, x; \texttt{select}\, y\, \$; \texttt{close}\, y \\ [] \; s \Rightarrow \texttt{select}\, y\, s; \texttt{select}\, y\, s; y \leftarrow \mathit{dbl} \leftarrow x)$$

If this process receives label `$` along x, it waits for x to terminate, emits `$`, and terminates. Otherwise, if this process receives s, it emits s twice and then calls itself corecursively.

Many sequents in this paper will have zero or exactly one antecedent; we call these subsingleton sequents. For proofs of subsingleton sequents, by extending Fortier and Santocanale's results [7] to include $\mathbf{1}$, we can eliminate cuts from proofs to construct cut-free proofs. The principal cut reductions are shown in Fig. 1; the $\longrightarrow$ relation also includes the standard commutative cut reductions, which we do not display.

For subsingleton sequents, we can also use a shorthand notation that elides channels and uses `L`- and `R`-variants of the usual process constructors. For example, we would write the

$$\frac{\Delta \vdash_\Theta P :: x{:}A \quad \Delta', x{:}A \vdash_\Theta Q :: z{:}C}{\Delta, \Delta' \vdash_\Theta P \triangleright_x Q :: z{:}C}\ \mathrm{CUT}_A \qquad \frac{}{x{:}A \vdash_\Theta z \leftrightarrow x :: z{:}A}\ \mathrm{ID}_A$$

$$\frac{X \in \mathrm{dom}\,\Theta}{\Delta \vdash_\Theta z \leftarrow X \leftarrow \Delta :: z{:}A}\ \mathrm{PVAR}$$

$$\frac{\Delta \vdash_\Theta P :: x{:}A_k \quad (k \in L)}{\Delta \vdash_\Theta \mathtt{select}\, x\, k; P :: x{:}\oplus_{\ell\in L}\{\ell{:}A_\ell\}}\ \oplus\mathrm{R} \qquad \frac{\forall \ell \in L{:}\quad \Delta', x{:}A_\ell \vdash_\Theta Q_\ell :: z{:}C}{\Delta', x{:}\oplus_{\ell\in L}\{\ell{:}A_\ell\} \vdash_\Theta x.\mathtt{case}_{\ell\in L}(\ell \Rightarrow Q_\ell) :: z{:}C}\ \oplus\mathrm{L}$$

$$\frac{\Delta_1 \vdash_\Theta P_1 :: y{:}A_1 \quad \Delta_2 \vdash_\Theta P_2 :: x{:}A_2}{\Delta_1, \Delta_2 \vdash_\Theta \mathtt{send}\, x\, P_1; P_2 :: x{:}A_1 \otimes A_2}\ \otimes\mathrm{R} \qquad \frac{\Delta', y{:}A_1, x{:}A_2 \vdash_\Theta Q :: z{:}C}{\Delta', x{:}A_1 \otimes A_2 \vdash_\Theta y \leftarrow \mathtt{recv}\, x; Q :: z{:}C}\ \otimes\mathrm{L}$$

$$\frac{}{\cdot \vdash_\Theta \mathtt{close}\, x :: x{:}\mathbf{1}}\ \mathbf{1}\mathrm{R} \qquad \frac{\Delta' \vdash_\Theta Q :: z{:}C}{\Delta', x{:}\mathbf{1} \vdash_\Theta \mathtt{wait}\, x; Q :: z{:}C}\ \mathbf{1}\mathrm{L}$$

$$(\mathtt{select}\, x\, k; P) \triangleright_x \left(x.\mathtt{case}_{\ell\in L}(\ell \Rightarrow Q_\ell)\right) \longrightarrow P \triangleright_x Q_k$$
$$(\mathtt{close}\, x) \triangleright_x (\mathtt{wait}\, x; Q) \longrightarrow Q$$
$$(x \leftrightarrow y) \triangleright_x Q \longrightarrow [y/x]Q \qquad P \triangleright_x (z \leftrightarrow x) \longrightarrow [z/x]P$$

Figure 1 Session-typing rules and some cut reductions for a fragment of intuitionistic linear logic

type of *dbl* as $Nat \vdash Nat$ and define *dbl* by

$$\begin{aligned} dbl = \mathtt{caseL}(\$ &\Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\$; \mathtt{closeR} \\ [] \ s &\Rightarrow \mathtt{selectR}\, s; \mathtt{selectR}\, s; dbl) \end{aligned}$$

3 Deterministic finite automata (DFAs)

This section reviews a standard mathematical model of deterministic finite automata. This specific presentation is derived from Sipser's formulation [16].

A *deterministic finite automaton* (DFA) M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet of input symbols a, $\delta\colon Q \times \Sigma \to Q$ is a (total) transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states. The set of all finite strings over alphabet Σ is written as Σ^*, with ϵ denoting the empty string.

A *computation* of M on the input string $w = a_1 a_2 \cdots a_n \in \Sigma^*$ is a sequence of states $q_0, q_1, \ldots, q_n$ such that $\delta(q_i, a_{i+1}) = q_{i+1}$ for each $0 \le i < n$. The DFA M is said to *accept* string w if there exists a computation on w that ends in an accepting state, $q_n \in F$; otherwise, M is said to *reject* string w. The *language recognized* by M is the set of all strings that are accepted by M. The *regular languages* are those that can be recognized by some DFA.

Alternatively, one may extend the transition function δ, which operates on symbols, to a cumulative function $\delta^*\colon Q \times \Sigma^* \to Q$ that operates on strings.

$$\begin{aligned} \delta^*(q, \epsilon) &= q \\ \delta^*(q, aw) &= \delta^*(\delta(q, a), w) \end{aligned}$$

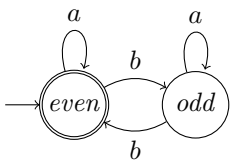


Figure 2 A DFA for strings over the alphabet $\Sigma = \{a, b\}$ that contain an even number of bs.

Intuitively, $\delta^*(q, w)$ is the state that the DFA reaches from state q upon reading input w. It's straightforward to prove, by induction on the length of the string w, that M accepts w if and only if $\delta^*(q_0, w) \in F$.

▶ **Example 2.** Figure 2 shows the state diagram for a DFA that recognizes those finite strings over the alphabet $\Sigma = \{a, b\}$ that contain an even number of bs. The initial state, as indicated by the unlabeled arrow, is *even*; it is also an accepting state, as depicted by its doubled outline. This DFA accepts the string *bab*, for example, because there is a path corresponding to *bab* from the initial state to an accepting state.

Although some definitions of DFAs allow δ to be a *partial* transition function [11], notice that we demand that δ is total. Totality ensures that a DFA never gets stuck while reading an input symbol, which will be crucial for the correspondence between DFA computations and cut reductions that we give in Section 4. In demanding totality, there is no loss of expressiveness: one can always introduce a distinguished rejecting state that acts as a sink for any transitions that would have been left undefined under a partial transition function.

4 DFAs are isomorphic to processes of a particular type

As described in the previous section, the computational behavior of a DFA derives from its transition function, $\delta \colon Q \times \Sigma \to Q$, which induces a cumulative function $\delta^* \colon Q \times \Sigma^* \to Q$ that can be used to characterize the language recognized by the DFA. Specifically, the DFA accepts string w if and only if $\delta^*(q_0, w)$ is an accepting state.

By currying, we may also think of δ^* as a finite family of functions $\delta^*_q \colon \Sigma^* \to Q$, one for each $q \in Q$, so that $\delta^*_q(w)$ is the state reached by the DFA from state q upon reading string w. Then, composing each δ^*_q with the indicator function $F \colon Q \to \mathbf{2}$ for the set of accepting states, we can see that the computational essence of each individual state is a Boolean-valued function on Σ^* that is closely coupled to the transition function δ.

This suggests a strategy for encoding DFAs — for each state q, define a process $[\![q]\!]$ of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$. If $\mathtt{String}_\Sigma$ is the type of finite strings over Σ and $\mathtt{Ans}$ is the type of accept/reject answers, then the process $[\![q]\!]$ is effectively a Boolean-valued function on Σ^*.

In the next three sections, we make this idea precise. Section 4.1 defines the type $\mathtt{String}_\Sigma$ and presents an isomorphism between finite strings over Σ and processes of type $\cdot \vdash \mathtt{String}_\Sigma$. Next, Section 4.2 presents an adequate encoding of DFA states as processes of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$ in which DFA transitions are matched by process reductions. Finally, Section 4.3 proves that, somewhat surprisingly, the encoding is an isomorphism.

4.1 Finite strings are isomorphic to processes of type $\cdot \vdash \mathtt{String}_\Sigma$

Strings $w \in \Sigma^*$ are finite lists — lists of symbols drawn from the finite alphabet Σ. In type theory, the type of finite lists with elements of type α is usually defined as the least solution

of the type equation $X_\alpha = (\alpha \times X_\alpha) + \mathbf{1}$, or, when the type α is finitely inhabited, the isomorphic $X_\alpha = X_\alpha + \cdots + X_\alpha + \mathbf{1}$.

By analogy, we will encode strings over Σ as cut-free processes of type $\cdot \vdash \mathtt{String}_\Sigma$, where the type $\mathtt{String}_\Sigma$ is the least solution of the type equation

$$X = \oplus_{a \in \Sigma \cup \{\$\}} \{a{:}A_a\} \quad \text{where} \quad A_a = \begin{cases} X & \text{if } a \in \Sigma \\ \mathbf{1} & \text{if } a = \$ \end{cases}$$

In other words, $\mathtt{String}_\Sigma$ is the inductive type $\mu X.\, \oplus_{a \in \Sigma \cup \{\$\}} \{a{:}A_a\}$. With a slight abuse of notation, in which we will indulge from here on, the type $\mathtt{String}_\Sigma$ may also be written as $\mu X.\, \oplus_{a \in \Sigma} \{a{:}X, \${:}\mathbf{1}\}$.

A cut-free process of type $\cdot \vdash \mathtt{String}_\Sigma$ emits a finite stream, or list, of symbols from Σ. By inversion on its typing derivation, such a process is either: $\mathtt{selectR}\,\$; \mathtt{closeR}$, which terminates the stream by emitting the terminal symbol, $\$$, to its right and then terminating communication; or $\mathtt{selectR}\,a; P$, which continues the stream by emitting some symbol a to its right and then behaving as a process P of type $\cdot \vdash \mathtt{String}_\Sigma$.

This suggests that we define the encoding $[\![-]\!] \colon \Sigma^* \to (\cdot \vdash \mathtt{String}_\Sigma)$ inductively by

$$[\![\epsilon]\!] = \mathtt{selectR}\,\$; \mathtt{closeR}$$
$$[\![aw]\!] = \mathtt{selectR}\,a; [\![w]\!]\,.$$

For example, with the alphabet $\Sigma = \{a, b\}$, the string ba is represented by the process $[\![ba]\!] = \mathtt{selectR}\,b; \mathtt{selectR}\,a; \mathtt{selectR}\,\$; \mathtt{closeR}$, which emits b, a, and $\$$ before terminating.

This representation of strings as processes of type $\cdot \vdash \mathtt{String}_\Sigma$ is adequate:

▶ **Theorem 3.** *Strings over the alphabet Σ are in bijective correspondence with cut-free processes of type $\cdot \vdash \mathtt{String}_\Sigma$.*

Proof. Define a function $-^\flat$ from cut-free processes of type $\cdot \vdash \mathtt{String}_\Sigma$ to strings over Σ:

$$(\mathtt{selectR}\,\$; \mathtt{closeR})^\flat = \epsilon$$
$$(\mathtt{selectR}\,a; P)^\flat = a(P)^\flat\,, \text{ if } a \neq \$\,.$$

The function $-^\flat$ is total because any cut-free process of type $\cdot \vdash \mathtt{String}_\Sigma$ has, by inversion, one of the forms on which $-^\flat$ is defined. It's also easy to show that $-^\flat$ is both a left and a right inverse of $[\![-]\!]$, thereby establishing that $[\![-]\!]$ is a bijection. ◀

4.2 Representing DFAs as processes of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$

Having now defined a type $\mathtt{String}_\Sigma$ and shown that Σ^* is isomorphic to cut-free processes of type $\cdot \vdash \mathtt{String}_\Sigma$, we can now turn to encoding DFAs as processes.

We first define the type $\mathtt{Ans}$ of accept/reject answers by $\mathtt{Ans} = \oplus\{\mathtt{acc}{:}\mathbf{1}, \mathtt{rej}{:}\mathbf{1}\}$. We'll encode each of the DFA's states as a process of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$ that emits $\mathtt{acc}$ or $\mathtt{rej}$ according to whether the DFA accepts or rejects the input string.

▶ **Example 4.** The DFA of Example 2 that recognizes strings over $\Sigma = \{a, b\}$ with an even number of bs would be encoded as a mutually corecursive pair of process definitions, one of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$ for each state:

$$\begin{aligned} \mathit{even} = \mathtt{caseL}(&a \Rightarrow \mathit{even} \parallel b \Rightarrow \mathit{odd} \\ &\parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\mathtt{acc}; \mathtt{closeR}) \end{aligned}$$

$$\begin{aligned} \mathit{odd} = \mathtt{caseL}(&a \Rightarrow \mathit{odd} \parallel b \Rightarrow \mathit{even} \\ &\parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\mathtt{rej}; \mathtt{closeR}) \end{aligned}$$

Under its definition, *even* is a process that waits to receive a label from its left and then case-analyzes it. If the label is a or b, then a corecursive call to the process *even* or *odd* is made, respectively. These calls serve to transition the DFA to the appropriate next state. Otherwise, if the label is the terminal symbol `$`, then the process waits for its input to terminate communication, emits `acc` to indicate that the input is accepted, and finally terminates. This describes *even* as an accepting state. The *odd* process can be read similarly.

Our encoding of DFAs follows the intuition behind the previous example. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary DFA. For each state $q \in Q$, we define a process named $[\![q]\!]_M$ of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$ by

$$[\![q]\!]_M = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow [\![\delta(q,a)]\!]_M \;[]\; \mathtt{\$} \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,[\![F(q)]\!]; \mathtt{closeR})$$

where $[\![F(q)]\!]$ is defined as the label `acc` if $q \in F$ (i.e., if q is an accepting state) and as the label `rej` otherwise. (The notation $[\![-]\!]$ is used for the encodings of both strings and DFA states, but the intended meaning is always clear from the context.)

When the process $[\![q]\!]_M$ receives a symbol $a \in \Sigma$, it makes a corecursive call to the process, $[\![\delta(q,a)]\!]_M$, that corresponds to the appropriate next state. Otherwise, when the process $[\![q]\!]_M$ receives `$`, it emits either `acc` or `rej` according to whether q is an accepting state.

This encoding of DFAs is adequate at a quite fine-grained level: every DFA transition is matched by a process reduction in the encoding.

▶ **Theorem 5.** *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Both of the following hold for all $q \in Q$.*

- *$[\![aw]\!] \triangleright [\![q]\!] \longrightarrow [\![w]\!] \triangleright [\![\delta(q,a)]\!]$ for all $a \in \Sigma$ and $w \in \Sigma^*$.*
- *$[\![\epsilon]\!] \triangleright [\![q]\!] \longrightarrow^2 \mathtt{selectR}\,[\![F(q)]\!]; \mathtt{closeR}$.*

Proof. Both parts follow by straightforward calculation. Taking the first part as an example, let $w \in \Sigma^*$ be an arbitrary string. By expanding definitions, the process $[\![aw]\!] \triangleright [\![q]\!]$ is

$$(\mathtt{selectR}\,a; [\![w]\!]) \triangleright (\mathtt{caseL}_{a\in\Sigma}(a \Rightarrow [\![\delta(q,a)]\!] \;[]\; \mathtt{\$} \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,[\![F(q)]\!]; \mathtt{closeR}))$$

which, thanks to the principal cut reduction that matches `selectR` with `caseL`, reduces to $[\![w]\!] \triangleright [\![\delta(q,a)]\!]$. The proof of the second claim is similar. ◀

This can be easily lifted, by induction, to full DFA computations; we elide the details.

4.3 Completing the isomorphism: From processes to DFAs

Although the encoding of DFAs as processes from the previous section is perhaps unsurprising, what is surprising is that the converse — that every process of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ corresponds to a DFA — is provable, as we now show.

Say that a signature of process definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ is in normal form if each definition has the form

$$X = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow X_a \;[]\; \mathtt{\$} \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\ell_X; \mathtt{closeR})\,,$$

where ℓ_X is an `acc`/`rej` label. It's easy to read off a DFA from this signature, as the following theorem shows.

▶ **Theorem 6.** *Let Σ be a finite alphabet and Θ be a signature of process definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ that are in normal form. For every $X_0 \in \operatorname{dom}\Theta$, there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $X_0 = [\![q_0]\!]_M$.*

Proof. The DFA can be directly read off of the signature Θ. Choose $Q = \operatorname{dom}\Theta$ as the set of states. Because signature Θ is in normal form, each process definition has the form $X = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow X_a \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\ell_X; \mathtt{closeR})$ where ℓ_X is an $\mathtt{acc}/\mathtt{rej}$ label. We read off the transition function δ by defining $\delta(X, a) = X_a$ for each X and $a \in \Sigma$. Similarly, include X in the set F of final states if and only if $\ell_X = \mathtt{acc}$.

Finally, choose an arbitrary $X_0 \in \operatorname{dom}\Theta$ and set $q_0 = X_0$ as the initial state. Thus, $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. By the definition of $[\![-]\!]_M$, it's immediate that $X_0 = [\![q_0]\!]_M$. ◀

Had we chosen polarized linear logic as our session-typing discipline [14], then focused proofs [1] of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ would necessarily be the normal form processes used in this theorem, making it the end of the story. However, we choose not to introduce polarization and focusing to keep the technical overhead lower. Thus, the following lemmas and theorem for normalizing a signature are necessary. (In this and subsequent proofs, we use $\equiv$ to denote equivalence up to standard commuting conversions and unfolding and folding of definitions; $=$ is reserved for syntactic equality.)

▶ **Lemma 7.** *Let Σ be a finite alphabet and Θ be a signature of cut-free process definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathbf{1}$. Given the definition $Y_0 = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow Y_0 \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{closeR})$, we have $Y \equiv Y_0$ for all $Y \in \operatorname{dom}\Theta$.*

Proof. First, repeatedly apply the following rewriting rule, introducing a new definition $Y = P$ as part of the rewriting. The rule applies only if P is not a process variable.

$$\begin{array}{lcl}\mathtt{caseL}_{b\in\Sigma-\{a\}}(a \Rightarrow P & \rightsquigarrow & \mathtt{caseL}_{b\in\Sigma-\{a\}}(a \Rightarrow Y \\ \quad \parallel b \Rightarrow Q_b \parallel \$ \Rightarrow R) & & \quad \parallel b \Rightarrow Q_b \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{closeR})\end{array}$$

Rewriting terminates because the rule decreases the number of rewriting sites. Upon termination, each definition has the form $Y = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow Y_a \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{closeR})$. By coinduction, each Y satisfies $Y \equiv Y_0$. Since rewriting adheres to $\equiv$, the lemma is proved. ◀

▶ **Lemma 8.** *Let Σ be a finite alphabet and Θ be a signature of cut-free process definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ and $\mathtt{String}_\Sigma \vdash_\Theta \mathbf{1}$. There exists a signature Θ' in normal form such that, for every $X \in \operatorname{dom}\Theta$, there exists an $X' \in \operatorname{dom}\Theta'$ such that $X \equiv X'$.*

Proof. First, repeatedly apply the following rewriting rules. Each rule applies only if P is not a process variable, and each rule introduces a new definition $X = P$ to the signature.

$$\mathtt{caseL}_{b\in\Sigma-\{a\}}(a \Rightarrow P \parallel b \Rightarrow Q_b \parallel \$ \Rightarrow R) \rightsquigarrow \mathtt{caseL}_{b\in\Sigma-\{a\}}(a \Rightarrow X \parallel b \Rightarrow Q_b \parallel \$ \Rightarrow R)$$
$$\mathtt{selectR}\,\ell; P \rightsquigarrow \mathtt{selectR}\,\ell; X$$

Rewriting terminates because it decreases the number of rewriting sites. Upon termination, all definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ have the forms $X = \mathtt{selectR}\,\ell; Y$ or $X = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow X_a \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\ell_X; \mathtt{closeR})$, and definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathbf{1}$ have the form $Y = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow Y_a \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{closeR})$.

Consider each definition $X = \mathtt{selectR}\,\ell_X; Y$. By Lemma 7, $X \equiv \mathtt{selectR}\,\ell_X; Y_0$. Expanding the definition of Y_0 and applying some commuting conversions, we have that $X \equiv \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow \mathtt{selectR}\,\ell_X; Y_0 \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\ell_X; \mathtt{closeR})$. We revise X's definition to $X = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow X \parallel \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\ell_X; \mathtt{closeR})$, since $X \equiv \mathtt{selectR}\,\ell_X; Y_0$. ◀

▶ **Theorem 9.** *Let Σ be a finite alphabet and Θ be a signature of process definitions of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$ and $\mathtt{String}_\Sigma \vdash_\Theta \mathbf{1}$. For every process P of type $\mathtt{String}_\Sigma \vdash_\Theta \mathtt{Ans}$, there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $P \equiv [\![q_0]\!]_M$.*

Proof. Introduce the definition $X_0 = P$ to signature Θ and then, after eliminating cuts (Section 2), appeal to Lemma 8 to normalize Θ to a signature Θ' such that there exists a $X_0' \in \operatorname{dom}\Theta'$ for which $X_0 \equiv X_0'$. By appealing to Theorem 6, we construct a DFA M such that $P = X_0 \equiv X_0' = [\![q_0]\!]_M$. ◀

This completes the proof that our process encoding of DFAs is an isomorphism.

5 Some closure properties of regular languages by cut elimination

The class of regular languages enjoys many closure properties, including closure under complementation, union, intersection, concatenation, and Kleene star. Traditionally, these properties are established by DFA constructions (sometimes using a nondeterministic finite automaton as an intermediate). Having seen that DFAs are isomorphic to processes of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$, it's natural to wonder how such constructions fit into this pleasing proof-theoretic picture.

The next two sections show that, perhaps surprisingly, closure of regular languages under complementation, union, and intersection can indeed be explained proof-theoretically in terms of cut elimination. Unfortunately, closure under concatenation and Kleene star are not as readily explainable proof-theoretically, primarily because the standard constructions rely heavily on nondeterminism; we leave their proof-theoretic investigation to future work.

5.1 Closure under complementation

Let L be an arbitrary regular language over the alphabet Σ. Because L is a regular language, there is a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes L. Using this DFA, there is a simple procedure for recognizing the language's complement, $\overline{L} = \Sigma^* - L$: simply run the input string through M and negate its answer. An input string is thus accepted if and only if M rejects that string.

As a process, this procedure can be expressed as $[\![q_0]\!]_M \triangleright \mathit{not}$, which composes the process, $[\![q_0]\!]_M$, that represents M's initial state with a process, not, that negates answers. We define not, of type $\mathtt{Ans} \vdash \mathtt{Ans}$, as follows; it case-analyzes an answer and then produces its negation.

$$\begin{aligned}\mathit{not} = \mathtt{caseL}(&\mathtt{acc} \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\mathtt{rej}; \mathtt{closeR}\\ \mid\; &\mathtt{rej} \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,\mathtt{acc}; \mathtt{closeR})\end{aligned}$$

Thus, the process $[\![q_0]\!]_M \triangleright \mathit{not}$ recognizes the complement language, $\overline{L}$. If recognizing $\overline{L}$ is all that we are interested in, then just using this process directly is certainly adequate. However, notice that $[\![q_0]\!]_M \triangleright \mathit{not}$ has type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$. If only this process were cut-free, then we could also apply Theorem 6 to extract a DFA that recognizes $\overline{L}$ and thereby prove closure of regular languages under complementation. The process $[\![q_0]\!]_M \triangleright \mathit{not}$ is not cut-free (because of $\triangleright$), but, as the following theorem shows, the cut can be eliminated.

▶ **Theorem 10.** *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $\overline{M} = (Q, \Sigma, \delta, q_0, Q - F)$. For all states $q \in Q$, there is an infinite reduction sequence $[\![q]\!]_M \triangleright \mathit{not} \longrightarrow^\omega [\![q]\!]_{\overline{M}}$.*

Proof. For each state $q \in Q$, we can reduce

$$\begin{aligned}&[\![q]\!]_M \triangleright \mathit{not}\\ &\quad\longrightarrow^+ \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow [\![\delta(q,a)]\!]_M \triangleright \mathit{not}\\ &\qquad\qquad\qquad \mid\; \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,[\![(Q-F)(q)]\!]; \mathtt{closeR})\\ &\quad\longrightarrow^\omega \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow [\![\delta(q,a)]\!]_{\overline{M}}\\ &\qquad\qquad\qquad \mid\; \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,[\![(Q-F)(q)]\!]; \mathtt{closeR})\end{aligned}$$

which, by definition of $[\![-]\!]_{\overline{M}}$, is $[\![q]\!]_{\overline{M}}$. The first step is justified by expanding definitions and carrying out a few reductions. The second step follows by appealing to the coinductive hypothesis, which gives an infinite reduction sequence $[\![\delta(q,a)]\!]_M \triangleright not \longrightarrow^\omega [\![\delta(q,a)]\!]_{\overline{M}}$. ◀

In other words, cut elimination from $[\![q_0]\!]_M \triangleright not$ converges to a process, and that process happens to be one which represents the DFA $\overline{M}$ that, by exchanging the accepting and rejecting states of M, is traditionally used to recognize the complement language.

Alternatively, we could shortcut the infinite reduction sequence that is used to eliminate the cut from $[\![q_0]\!]_M \triangleright not$. Since there are finitely many cuts $[\![q]\!]_M \triangleright not$ with $q \in Q$, rather than appealing to a coinductive hypothesis after the first few reductions, we could instead introduce, at that point, a definition of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$ in normal form for each state q:

$$X_q = \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow X_{\delta(q,a)} \;[]\; \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\,[\![(Q-F)(q)]\!]; \mathtt{closeR})\,.$$

The DFA that Theorem 6 extracts from this signature is exactly $\overline{M} = (Q, \Sigma, \delta, q_0, Q - F)$. So — not only does cut reduction converge (Theorem 10) — cut reduction, in fact, *constructs* the DFA $\overline{M}$ that is traditionally used to recognize the complement language!

5.2 Closure under intersection and union

The same basic idea can be used to show that closure of regular languages under intersection and union can also be understood proof-theoretically in terms of cut elimination.

Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be arbitrary DFAs over alphabet Σ. To recognize the intersection of the languages recognized by M_1 and M_2, we might use the process $(z_1 \leftarrow [\![q_1]\!]_{M_1} \leftarrow x) \triangleright_{z_1} (z_2 \leftarrow [\![q_2]\!]_{M_2} \leftarrow x) \triangleright_{z_2} (z \leftarrow and \leftarrow z_1, z_2)$, where $z_1{:}\mathtt{Ans}, z_2{:}\mathtt{Ans} \vdash and :: z{:}\mathtt{Ans}$, defined in the obvious way, computes conjunction of Booleans.

Unfortunately, this process is not well-typed because the linear channel x along which the input arrives is used by both $[\![q_1]\!]_{M_1}$ and $[\![q_2]\!]_{M_2}$, violating linearity. To maintain linearity, we need to define the following process $x{:}\mathtt{String}_\Sigma \vdash dup :: y{:}\mathtt{String}_\Sigma \otimes (\mathtt{String}_\Sigma \otimes \mathbf{1})$ that duplicates the input by taking advantage of multiplicative conjunction. Then, using *dup*, a process $isect_{q_1,q_2}$ of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$ that recognizes the intersection

$$\begin{array}{l}
y \leftarrow dup \leftarrow x = \\
\quad x.\mathtt{case}_{a\in\Sigma}(a \Rightarrow (y' \leftarrow dup \leftarrow x) \triangleright_{y'} \\
\qquad\qquad y'_1 \leftarrow \mathtt{recv}\, y'; y'_2 \leftarrow \mathtt{recv}\, y'; \mathtt{wait}\, y'; \\
\qquad\qquad \mathtt{send}\, y\, (\mathtt{select}\, y_1\, a; y_1 \leftrightarrow y'_1); \\
\qquad\qquad \mathtt{send}\, y\, (\mathtt{select}\, y_2\, a; y_2 \leftrightarrow y'_2); \\
\qquad\qquad \mathtt{close}\, y \\
\qquad\quad [] \; \$ \Rightarrow \mathtt{wait}\, x; \\
\qquad\qquad\quad \mathtt{send}\, y\, (\mathtt{select}\, y_1\, \$; \mathtt{close}\, y_1); \\
\qquad\qquad\quad \mathtt{send}\, y\, (\mathtt{select}\, y_2\, \$; \mathtt{close}\, y_2); \\
\qquad\qquad\quad \mathtt{close}\, y)
\end{array}$$

$$\begin{array}{l}
z \leftarrow isect_{q_1,q_2} \leftarrow x = \\
\quad (y \leftarrow dup \leftarrow x) \triangleright_y \\
\quad y_1 \leftarrow \mathtt{recv}\, y; y_2 \leftarrow \mathtt{recv}\, y; \mathtt{wait}\, y; \\
\quad (z_1 \leftarrow [\![q_1]\!]_{M_1} \leftarrow y_1) \triangleright_{z_1} \\
\quad (z_2 \leftarrow [\![q_2]\!]_{M_2} \leftarrow y_2) \triangleright_{z_2} \\
\quad z \leftarrow and \leftarrow z_1, z_2
\end{array}$$

Once again, if recognizing the intersection is all that we are interested in, then simply using this process directly is certainly adequate.

▶ **Theorem 11.** *Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be DFAs and let $M = (Q_1 \times Q_2, \Sigma, \delta, (q_1, q_2), F_1 \times F_2)$ be their product, where $\delta((q,q'),a) = (\delta_1(q,a), \delta_2(q',a))$ for all $(q,q') \in Q_1 \times Q_2$. For all $(q,q') \in Q_1 \times Q_2$, there is an infinite reduction sequence $(z \leftarrow isect_{q,q'} \leftarrow x) \longrightarrow^\omega (z \leftarrow [\![(q,q')]\!]_M \leftarrow x)$.*

Proof. By coinduction. The proof is similar to that of complementation (Theorem 10). ◀

As with complementation, we can alternatively shortcut the infinite reduction sequence by introducing definitions, one for the partially reduced form of each of the finitely many $z \leftarrow \mathit{isect}_{q,q'} \leftarrow x$. In this way, cut reduction can be seen as *constructing* the very same product DFA M that is typically used to prove closure of regular languages under intersection.

By a similar argument using a process $z_1{:}\mathtt{Ans}, z_2{:}\mathtt{Ans} \vdash \mathit{or} :: z{:}\mathtt{Ans}$ that computes the disjunction of two answers, it's possible to show that closure of regular languages under union follows from cut reduction. We omit the details because of space constraints.

6 An isomorphism between subsequential transducers and processes

Subsequential transducers [13, 15] are a generalization of DFAs which produce a full string — rather than a single accept/reject bit — as output. Having proved in Section 4 that DFAs are isomorphic to processes of type $\mathtt{String}_\Sigma \vdash \mathtt{Ans}$, it's natural to wonder if subsequential transducers are also isomorphic to processes of a particular type.

After reviewing the mathematical definition of subsequential transducers in Section 6.1, we present, in Section 6.2, an adequate encoding of transducers as processes of type $\mathtt{String}_\Sigma \vdash \mathtt{String}_\Gamma$. This encoding is very closely related to the encoding of DFAs from Section 4.2. Section 6.3 proves that this encoding is indeed an isomorphism.

6.1 Subsequential transducers

This presentation of subsequential transducers derives from a formulation by Mohri [13].

A (deterministic) *subsequential transducer* T is a 7-tuple $(Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ where Q is a finite set of states, Σ is a finite alphabet of input symbols, Γ is a finite alphabet of output symbols, $\delta\colon Q \times \Sigma \to Q$ is a transition function, $\sigma\colon Q \times \Sigma \to \Gamma^*$ is an output function, $\rho\colon Q \to \Gamma^*$ is a terminal output function, and $q_0 \in Q$ is the initial state. (The functions δ, σ, and ρ must be total functions.)

Once again, one may extend the transition function δ and the output function σ, which operate on input symbols, to cumulative functions $\delta^*\colon Q \times \Sigma^* \to Q$ and $\sigma^*\colon Q \times \Sigma^* \to \Gamma^*$ that operate on input strings. The function δ^* is defined as for DFAs, but is repeated here for convenience.

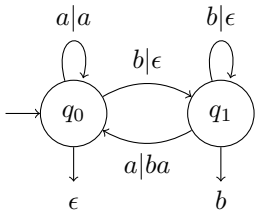
$$\begin{aligned} \delta^*(q, \epsilon) &= q & \qquad \sigma^*(q, \epsilon) &= \epsilon \\ \delta^*(q, aw) &= \delta^*(\delta(q, a), w) & \qquad \sigma^*(q, aw) &= \sigma(q, a)\ \sigma^*(\delta(q, a), w) \end{aligned}$$

The subsequential transducer T is said to *transduce* the input string $w \in \Sigma^*$ to the output string $\sigma^*(q, w)\ \rho(\delta^*(q, w)) \in \Gamma^*$.

Notice that, unlike some definitions, this definition does not include a set F of accepting states, nor corresponding notions of acceptance or rejection of input strings. This is because we are interested in transducers that induce a total transduction function, since such transductions turn out to compose more naturally in our proof-theoretic setting.

▶ **Example 12.** Let $\Sigma = \Gamma = \{a, b\}$ be an alphabet used for both input and output. The state diagram shown in Fig. 3 depicts a subsequential transducer that compresses each run of bs into a single b; the string *abbaabbb* is transduced to *abaab*, for example.

There are two states, q_0 and q_1, with the unlabeled arrow indicating q_0 as the initial state. The outgoing arrows labeled ϵ and b indicate the value of the terminal output function, ρ, for their respective states. The remaining edges are labeled to indicate an input symbol and the corresponding output string for that state and input symbol. For example, the edge from q_1 to q_0 that is labeled $a|ba$ indicates that $\delta(q_1, a) = q_0$ and $\sigma(q_1, a) = ba$. The output



$$\begin{aligned}
[\![q_0]\!] &= \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow \mathtt{selectR}\, a; [\![q_0]\!] \\
&\quad [] \; b \Rightarrow [\![q_1]\!] \\
&\quad [] \; \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\, \$; \mathtt{closeR}) \\
[\![q_1]\!] &= \mathtt{caseL}_{a\in\Sigma}(a \Rightarrow \mathtt{selectR}\, b; \mathtt{selectR}\, a; [\![q_0]\!] \\
&\quad [] \; b \Rightarrow [\![q_1]\!] \\
&\quad [] \; \$ \Rightarrow \mathtt{waitL}; \mathtt{selectR}\, b; \mathtt{selectR}\, \$; \mathtt{closeR})
\end{aligned}$$

Figure 3 The state diagram and process encoding (see Section 6.2) of a subsequential transducer over the alphabet $\Sigma = \Gamma = \{a, b\}$ that compresses each run of bs into a single b.

transduced from a given input can be obtained by concatenating the output labels along the path corresponding to the input.

6.2 Representing transducers as processes of type $\mathtt{String}_\Sigma \vdash \mathtt{String}_\Gamma$

Just as the computational essence of a DFA state is a Boolean-valued function on finite strings, we can see from the previous discussion that the computational essence of a transducer state q is a function $\sigma_q^* \colon \Sigma^* \to \Gamma^*$ from finite input strings to finite output strings. Thus, the strategy for encoding transducers is similar to the encoding of DFAs given in Section 4 — for each transducer state q, we will define a process $[\![q]\!]_T$ of type $\mathtt{String}_\Sigma \vdash \mathtt{String}_\Gamma$.

Before doing so, we need to slightly generalize the encoding of finite strings that was given in Section 4.1: we define a function $[-;-] \colon \Sigma^* \times (\Delta \vdash \mathtt{String}_\Sigma) \to (\Delta \vdash \mathtt{String}_\Sigma)$ that emits a string as a finite stream of symbols and then, unlike $[\![-]\!] \colon \Sigma^* \to (\cdot \vdash \mathtt{String}_\Sigma)$, continues as the given process rather than always terminating. The function $[-;-]$ is defined inductively by

$$\begin{aligned}
[\epsilon; P] &= P \\
[aw; P] &= \mathtt{selectR}\, a; [w; P]\,.
\end{aligned}$$

It's straightforward to prove that $[w_1; [\![w_2]\!]] = [\![w_1 w_2]\!]$, for all $w_1, w_2 \in \Sigma^*$, by induction on the length of w_1. In particular, $[w; (\mathtt{selectR}\, \$; \mathtt{closeR})] = [\![w]\!]$ for all $w \in \Sigma^*$.

With this generalization in hand, we are now ready to give a process encoding of transducers. Let $T = (Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ be an arbitrary subsequential transducer. Define a mutually corecursive family of process definitions $[\![q]\!]_T$, one of type $\mathtt{String}_\Sigma \vdash \mathtt{String}_\Gamma$ for each state $q \in Q$:

$$\begin{aligned}
[\![q]\!]_T = \mathtt{caseL}_{a\in\Sigma}(a &\Rightarrow [\sigma(q,a); [\![\delta(q,a)]\!]_T] \\
[] \; \$ &\Rightarrow \mathtt{waitL}; [\![\rho(q)]\!])
\end{aligned}$$

Under its definition, the process named $[\![q]\!]_T$ waits to receive a label from its left and then case-analyzes it. If the label is some $a \in \Sigma$, then the process emits the output string $\sigma(q, a)$ as a finite stream of symbols and then makes a corecursive call to the process $[\![\delta(q, a)]\!]_T$. Otherwise, if the label is the terminal symbol $, then the process waits for its input to terminate communication and then emits the terminal output string $\rho(q)$.

▶ **Example 13.** The transducer of Fig. 3 that compresses runs of bs can be encoded as a pair of process definitions of type $\mathtt{String}_\Sigma \vdash \mathtt{String}_\Gamma$, one for each state. The process definitions are shown in the second part of Fig. 3.

Like the encoding of DFAs, this encoding of transducers as processes of type $\texttt{String}_\Sigma \vdash \texttt{String}_\Gamma$ is adequate. Its adequacy is fine-grained: once again, each transducer transition is matched by a process reduction.

▶ **Theorem 14.** *Let $T = (Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ be a subsequential transducer. For all $q \in Q$:*
- *If $\Delta \vdash P : \texttt{String}_\Sigma$, then $(\texttt{selectR}\, a; P) \triangleright [\![q]\!] \longrightarrow P \triangleright [\sigma(q,a); [\![\delta(q,a)]\!]]$ for all $a \in \Sigma$.*
- *$[\![\epsilon]\!] \triangleright [\![q]\!] \longrightarrow^2 [\![\rho(q)]\!]$.*

Proof. By straightforward calculation, similar to the proof of Theorem 5 for DFAs. ◀

▶ **Corollary 15.** *Let $T = (Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ be a subsequential transducer. Then, for all $q \in Q$ and $w \in \Sigma^*$, there is a finite reduction sequence $[\![w]\!] \triangleright [\![q]\!] \longrightarrow^+ [\![\sigma^*(q,w)\ \rho(\delta^*(q,w))]\!]$.*

Proof. By induction on the length of w, appealing to Theorem 14. ◀

6.3 Completing the isomorphism: From processes to transducers

In this section, we prove the converse — that a subsequential transducer can be extracted from a proof of type $\texttt{String}_\Sigma \vdash \texttt{String}_\Gamma$. The general approach follows that of Section 4.3.

Say that a signature of process definitions of type $\texttt{String}_\Sigma \vdash_\Theta \texttt{String}_\Gamma$ is in normal form if each definition has the form

$$X = \texttt{caseL}_{a \in \Sigma}(a \Rightarrow [u_a; X_a] \,[]\, \$ \Rightarrow \texttt{waitL}; [\![v]\!])$$

for some strings $u_a \in \Gamma^*$ and $v \in \Gamma^*$. As with DFAs, it's easy to read off a transducer.

▶ **Theorem 16.** *Let Σ and Γ be finite alphabets and Θ be a signature of process definitions of type $\texttt{String}_\Sigma \vdash_\Theta \texttt{String}_\Gamma$ that are in normal form. For every $X_0 \in \operatorname{dom}\Theta$, there exists a subsequential transducer $T = (Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ such that $X_0 = [\![q_0]\!]_T$.*

Proof. Choose $Q = \operatorname{dom}\Theta$ as the set of states. Because signature Θ is in normal form, each process definition has the form $X = \texttt{caseL}_{a \in \Sigma}(a \Rightarrow [u_a; X_a] \,[]\, \$ \Rightarrow \texttt{waitL}; [\![v]\!])$ for some strings $u_a \in \Gamma^*$ and $v \in \Gamma^*$. We read off the functions δ, σ, and ρ by defining $\delta(X, a) = X_a$ and $\sigma(X, a) = u_a$ and $\rho(X) = v$ for each X and $a \in \Sigma$. Finally, choose an arbitrary $X_0 \in \operatorname{dom}\Theta$ and set $q_0 = X_0$ as the initial state. Thus, $T = (Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ is a subsequential transducer. By the definition of $[\![-]\!]_T$, it's immediate that $X_0 = [\![q_0]\!]_T$. ◀

Once again, with polarized linear logic as the session-typing discipline [14] and focusing [1], this would be the end of the story since only normal signatures would be allowed. Without those technical devices, we can use the following lemma that normalizes a signature.

▶ **Lemma 17.** *Let Σ and Γ be finite alphabets and Θ be a signature of cut-free process definitions of type $\texttt{String}_\Sigma \vdash_\Theta \texttt{String}_\Gamma$ and $\texttt{String}_\Sigma \vdash_\Theta \mathbf{1}$. There exists a signature Θ' in normal form such that, for every $X \in \operatorname{dom}\Theta$, there exists an $X' \in \operatorname{dom}\Theta'$ such that $X \equiv X'$.*

Proof. First, repeatedly apply the following rewriting rules. Unlike DFAs, each rule applies only if P is a `case`. Each rule introduces a new definition $X = P$ to the signature.

$$\texttt{caseL}_{b \in \Sigma - \{a\}}(a \Rightarrow P \,[]\, b \Rightarrow Q_b \,[]\, \$ \Rightarrow R) \rightsquigarrow \texttt{caseL}_{b \in \Sigma - \{a\}}(a \Rightarrow X \,[]\, b \Rightarrow Q_b \,[]\, \$ \Rightarrow R)$$
$$\texttt{selectR}\, \ell; P \rightsquigarrow \texttt{selectR}\, \ell; X$$

Also contract definitions $X_1 = [v_1; X_2]$ and $X_2 = [v_2; X_3]$ to $X_1 = [v_1 v_2; X_3]$. This procedure terminates because there are finitely many definitions and rewriting decreases the number

of rewriting sites. Upon termination, all definitions of type $\texttt{String}_\Sigma \vdash_\Theta \texttt{String}_\Gamma$ have the forms $X = [v; (\texttt{selectR}\,\$; Y)]$ or $X = \texttt{caseL}_{a\in\Sigma}(a \Rightarrow [u_a; X_a] \,[]\, \$ \Rightarrow \texttt{waitL}; [\![v]\!])$, and those of type $\texttt{String}_\Sigma \vdash_\Theta \mathbf{1}$ have the form $Y = \texttt{caseL}_{a\in\Sigma}(a \Rightarrow Y_a \,[]\, \$ \Rightarrow \texttt{waitL}; \texttt{closeR})$.

Consider each definition $X = [v; (\texttt{selectR}\,\$; Y)]$. By Lemma 7, $X \equiv [v; (\texttt{selectR}\,\$; Y_0)]$. Expanding the definition of Y_0 and applying some commuting conversions, we have that $X \equiv \texttt{caseL}_{a\in\Sigma}(a \Rightarrow [v; Y_0] \,[]\, \$ \Rightarrow \texttt{waitL}; [v; \texttt{selectR}\,\$; \texttt{closeR}])$. Thus, we form a new signature Θ' by revising X's definition to $X = \texttt{caseL}_{a\in\Sigma}(a \Rightarrow X \,[]\, \$ \Rightarrow \texttt{waitL}; [\![v]\!])$, since $X \equiv [v; (\texttt{selectR}\,\$; Y_0)]$. ◀

Using this lemma and the previous theorem, it's possible to complete the proof that the process representation of transducers is an isomorphism with the following theorem. We omit the easy details due to space constraints.

▶ **Theorem 18.** *Let Σ be a finite alphabet and Θ be a signature of process definitions of type $\texttt{String}_\Sigma \vdash_\Theta \texttt{String}_\Gamma$ and $\texttt{String}_\Sigma \vdash_\Theta \mathbf{1}$. For every process P of type $\texttt{String}_\Sigma \vdash_\Theta \texttt{String}_\Gamma$, there exists a transducer $T = (Q, \Sigma, \Gamma, \delta, \sigma, \rho, q_0)$ such that $P \equiv [\![q_0]\!]_T$.*

7 Comments on transducer compositions

Composing transducers. Composing two transducers $T_1 = (Q_1, \Sigma, \Gamma, \delta_1, \sigma_1, \rho_1, q_1)$ and $T_2 = (Q_2, \Gamma, \Omega, \delta_2, \sigma_2, \rho_2, q_2)$ is simple: just compose their process representations. Because $[\![q_1]\!]_{T_1}$ and $[\![q_2]\!]_{T_2}$ have types $\texttt{String}_\Sigma \vdash \texttt{String}_\Gamma$ and $\texttt{String}_\Gamma \vdash \texttt{String}_\Omega$, respectively, the composition $[\![q_1]\!]_{T_1} \triangleright [\![q_2]\!]_{T_2}$ is well-typed.

By composing transducers via their process representation, a concurrent semantics of transducer composition comes for free from the concurrent semantics of processes. As one example, in the transducer chain $[\![w]\!] \triangleright [\![q_1]\!]_{T_1} \triangleright [\![q_2]\!]_{T_2} \triangleright [\![q_3]\!]_{T_3} \triangleright \cdots \triangleright [\![q_n]\!]_{T_n}$, the transducer process for T_1 can react to the next symbol of input while T_2, having read T_1's first round of output, can supply output to T_3. Using an asynchronous concurrent semantics for session-typed processes based in proof theory [6], even more concurrency can be had: for example, the transducer process for T_1 can then react to the next symbol of input while T_2 is still absorbing T_1's first round of output.

If this kind of pipelined computation is one's only interest, then simply composing the transducers' process representations is suitable and very natural. Because subsequential functions are closed under composition, however, it's also possible to construct a subsequential transducer for the composition by eliminating the cut from $[\![q_1]\!]_{T_1} \triangleright [\![q_2]\!]_{T_2}$. As with closure of regular languages under complementation, intersection, and union, cut elimination yields the very transducer that is traditionally used (see [13]) to realize the composition. We omit the details due to space constraints.

Composing a transducer and a DFA. It's also possible to compose a transducer $T = (S, \Sigma, \Gamma, \delta_T, \sigma, \rho, s_0)$ and a DFA $M = (Q, \Gamma, \delta_M, q_0, F)$ by composing their process representations: $[\![s_0]\!]_T \triangleright [\![q_0]\!]_M$ is a well-typed process for their composition.

Incidentally, this kind of transducer–DFA composition can be used to prove closure of regular languages under inverse string homomorphism. For each string homomorphism $\varphi\colon \Sigma \to \Gamma^*$, we can construct a transducer T_φ with a single state s_0 that realizes φ. If M is a DFA with initial state q_0 that recognizes some language L, then $[\![s_0]\!]_{T_\varphi} \triangleright [\![q_0]\!]_M$ recognizes the language $\varphi^{-1}(L)$. As we did for closure under complementation, intersection, and union, we can again use cut reduction on $[\![s_0]\!]_{T_\varphi} \triangleright [\![q_0]\!]_M$ to *construct* the DFA that is traditionally used to prove closure under inverse string homomorphism, thereby showing that closure is again, proof-theoretically, cut elimination. We omit the details due to space constraints.

8 Conclusion

In this paper, we have established an isomorphism between DFAs and proofs of $\texttt{String}_\Sigma \vdash \texttt{Ans}$ in which DFA transitions correspond to cut reductions (Section 4). We've also established a related isomorphism between subsequential transducers and proofs of $\texttt{String}_\Sigma \vdash \texttt{String}_\Gamma$ (Section 6). Under these isomorphisms, closure of regular languages under complementation, intersection, union, and inverse homomorphism and closure of subsequential functions under composition can all be understood proof-theoretically in terms of cut elimination (Sections 5 and 7). Regular expression derivatives [2] may possibly be useful in similarly explaining closure under concatenation and Kleene star, but we leave that to future work. Another avenue for future work is to conduct a proof-theoretic investigation of deterministic pushdown automata and transducers; our preliminary results in this area appear promising.

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
2 Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
3 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st Int. Conf. on Concurrency Theory*, volume 6269 of *LNCS*, pages 222–236, 2010.
4 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.
5 Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, 2003.
6 Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In *21st Conf. on Computer Science Logic*, volume 16 of *LIPIcs*, pages 228–242, 2012.
7 Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: Semantics and cut elimination. In *22nd Conf. on Computer Science Logic*, volume 23 of *LIPIcs*, pages 248–262, 2013.
8 Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.
9 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
10 Kohei Honda. Types for dyadic interaction. In *4th Int. Conf. on Concurrency Theory*, volume 715 of *LNCS*, pages 509–523, 1993.
11 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 3rd edition, 2013.
12 William A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.
13 Mehryar Mohri. Finite-state transducers in language and speech processing. *Journal of Computational Linguistics*, 23(2):269–311, 1997.
14 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *18th Int. Conf. on Foundations of Software Science and Computation Structures*, 2015. Invited talk.
15 Marcel Paul Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, 1977.
16 Michael Sipser. *Introduction to the Theory of Computation*. Cengage, 3rd edition, 2013.