A HIERARCHY OF TRANSDUCING OBSERVER SYSTEMS

Norbert Hundeshagen [A] Peter Leupold[B]

[A]Fachbereich Elektrotechnik/Informatik
Universität Kassel,
34109 Kassel, Germany
hundeshagen@theory.informatik.uni-kassel.de

[B]Institut für Informatik,
Universität Leipzig,
Leipzig, Germany
Peter.Leupold@web.de

Abstract
We mainly investigate the power of weight-reducing string-rewriting systems in the context of transducing observer systems. First we show that they can compute anything that a RRWW*-transducer can. Then we situate them between painter and length-reducing systems. Further we show that for every weight-reducing system there is an equivalent one that uses only weight-reducing painter rules. This result enables us to prove that the class of relations that is computed by transducing observer systems with weight-reducing rules is closed under intersection.*

1. Transducing by Observing

The paradigm of *Computing by Observing* was originally introduced for generating and accepting formal languages [3, 4]. The basic architecture is depicted in Figure 1. At the basis there is some system that evolves in discrete steps. Every configuration of this system is mapped to one single letter by the so-called observer. In this way a kind of protocol of the computation is built. This idea of observing and writing a protocol translates maybe even more naturally into transductions if we consider the input and the observation as a pair.

This approach was first investigated with painter string-rewriting systems as underlying systems [5]. They turned out to be quite powerful and compute classes of transductions that are beyond the best-known classes like rational and push-down transductions. This corresponds to the fact that they are equivalent to linear bounded automata, when we use them for accepting languages.

In the sequel, length-reducing string-rewriting systems were shown to compute a smaller class of transductions that is actually similar to the one computed by RRWW-transducers [6]. The main problem in showing these two classes to be equivalent was that a length-reducing system can make at most n reduction steps on a string of length n. In this respect weight-reducing

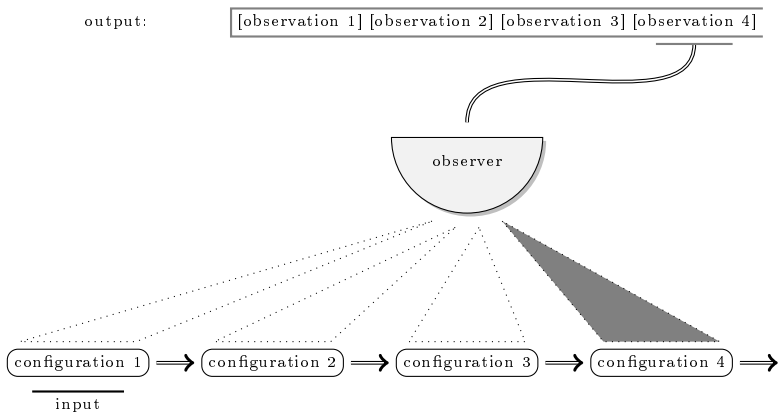

Figure 1: Schematic representation of a transducing observer system.

systems are slightly more permissive, because they can allow a number of steps that is linear in the length of the input string. For this reason they seem to be a good candidate for a comparison with RRWW-transducers.

We recall the definitions concerning transducing observer systems and establish a hierarchy between such systems according to the type of string-rewriting rules they use. Length-reducing rules lead to the smallest class of transductions, weight-reducing rules are at least as powerful and very probably compute more transductions. Finally, the relations that are computed with painter systems include both of the other classes. If we use painter systems whose rules always reduce weight we compute the same class of transductions as with general weight-reducing systems. This result enables us to prove that the class of relations that is computed by transducing observer systems with weight-reducing rules is closed under intersection. Intuitively, since the painter rules do not reduce the working space, it can be used twice, once for each of the intersected relations. Such a result can probably not be established for length-reducing systems, which substantiates our conjecture that the latter are of strictly lesser power.

2. Definitions and Examples

The reader is assumed familiar with standard terminology and notations from Formal Language Theory as they are exposed for example by Salomaa [10]. For the basic notions on transductions we refer to the book by Aho and Ullman [1]; in contrast to this standard approach, we exempt pairs with the empty word on the left hand side and some nonempty right hand side. Since the input is our computation space, such pairs might make classes different that are equal for all other pairs. We use *letter* and *symbol* synonymously for the elements of the alphabet.

2.1. Transducing Observer Systems

The observed systems in our architecture will be string-rewriting systems. Concerning these we follow notations and terminology as exposed by Book and Otto [2]. A *string-rewriting system* W on an alphabet Σ is a subset of $\Sigma^* \times \Sigma^*$. Its elements are called rewrite rules, and are written either as ordered pairs (ℓ, r) or as $\ell \to r$ for $\ell, r \in \Sigma^*$.

In the role of observers we use the devices that have become standard in this function: monadic transducers.

Definition 1 A *monadic transducer* (MT for short) is a tuple $\mathcal{O} = (Q, \Sigma, \Delta, \delta, q_0, \phi)$, where the set of states Q, the input alphabet Σ, the transition function δ, and the start state q_0 are the same as for deterministic finite automata. Δ is the output alphabet, and ϕ is the output function, a mapping $Q \mapsto \Delta \cup \{\varepsilon\}$ which assigns an output letter or the empty word to each state. The class of all monadic transducers is denoted by $\mathcal{MT}$.

The mode of operation is that the monadic transducer reads the input word, and then the image under ϕ of the state it stops in is the output. For length-reducing string-rewriting systems this imposes a very strict limit on the right-hand sides of relations. Since every computation step has to delete at least one symbol, the right-hand side can never be longer than the left-hand side.

This would make these relations incomparable even to simple classes like the one defined by GSMs; the difference, however could be eliminated by a simple morphism. Therefore we introduce a slightly modified device, *generalized monadic transducers*, which work just like monadic transducers except for the fact that they output strings instead of single symbols. The class of all these devices we denote by $g\mathcal{MT}$. In what follows, we will not always use the attribute *generalized* in the text, since from the output function it will be clear, which type of device we are using.

Now we combine the two components, a string-rewriting system and a monadic transducer in the way described in the introduction.

Definition 2 A *transducing observer system*, short T/O system is a triple $\Omega = (\Sigma, W, \mathcal{O})$, where Σ is the input alphabet, W is a string-rewriting system over an alphabet Γ such that $\Sigma \subseteq \Gamma$ which consists of all the symbols that occur in the rule set W, and $\mathcal{O}$ is a (generalized) monadic transducer, whose input alphabet is Γ.

The mode of operation of a transducing observer system $\Omega = (\Sigma, W, \mathcal{O})$ is as follows: the string-rewriting system starts to work on an input word u. After every reduction step the observer reads the new string and produces an output. The concatenation of all observations of a terminating derivation forms the output word v. The relation that Ω computes consists of all possible pairs (u, v). Note that already the input string is the first observation; thus there can be an output even if no rewriting rule can be applied to the first string.

Further, the observer is equipped with an important feature: By outputting the special symbol

$\perp$ it can abort a computation. In that case no output is produced. The other way in which no output might be produced is, if the string-rewriting system does not terminate. Formally, the relation computed is

$$\mathsf{Rel}(\Omega) = \{(u, v) \mid w \in W(u) \text{ and } v = \mathcal{O}(w)\},$$

where $W(u)$ denotes all sequences of words $(u, u_2, \ldots, u_k)$ that form terminating derivations $u \Rightarrow_W u_2 \Rightarrow_W \cdots \Rightarrow_W u_k$ of W. Thus $\mathsf{Rel}(\Omega)$ consists of all pairs of input words combined with the observations of possible terminating derivations on the given input word. Last but not least the class of relations defined by a special type of transducing observing system is denoted by $\mathcal{R}\mathsf{el}$; however, we will most often just use the name of the class if no confusion can arise.

Different types of string-rewriting rules result in different types of transducing observer systems. Here we will use the following three types of string-rewriting systems:

Painters: A string-rewriting system is called a *painter* system if for all its rules (ℓ, r), we have $|\ell| = |r| = 1$, that is, every rule just replaces one letter by another one.

Weight-reducing: A string-rewriting system over an alphabet Σ is called *weight-reducing* if there exists a weight function $\omega : \Sigma \mapsto \mathbb{N}_+$ from the alphabet to the set of all positive integers such that for all the rewrite rules (ℓ, r) we have $\omega(\ell) > \omega(r)$. Thus every rule reduces the string's weight by at least one.

Length-reducing: A string-rewriting system is called *length-reducing* if for all its rules (ℓ, r) we have $|\ell| > |r|$, that is, every rule shortens the string by at least one symbol.

The class of all string-rewriting systems that have painter, weight-reducing, and length-reducing rules are denoted by pnt-*SRS*, wr-*SRS*, and lr-*SRS*, respectively. The corresponding classes of transducing observer systems are denoted by pnt-T/O, wr-T/O, and lr-T/O.

In order to see how a transducing observer system works, we first look at an example that uses painter rules.

Example 3 We now construct a transducing observer system that computes the relation $\{(a^n, (a^n b^n)^n) : n \geq 2\}$. Note that the right-hand sides of the pairs are much longer than the left-hand sides; more precisely, a right-hand side of length n has a left-hand side of length $2n^2$.

The tactics our system follows to generate the right-hand side is the following: if the input consists only of letters a, then these are changed to A from left to right one by one in n steps. During each of these steps, the entire string is marked with underline from left to right; every time a letter is marked the output a is produced. Then the marks are removed and in every step b is output. When the entire input string has been converted to A this will have produced exactly one factor $a^n b^n$ for every input letter.

The string-rewriting rules that we employ are $a \to \alpha$ and $\underline{\alpha} \to A$ for converting the input letters in two steps to A; $a \to \underline{a}$, $\alpha \to \underline{\alpha}$, and $A \to \underline{A}$ for marking the letters; $\underline{a} \to a$ and $\underline{A} \to A$ for unmarking the letters.

Of course, these rules could be applied in arbitrary orders by the string-rewriting system. Thus the observer must be constructed in such a way that it rejects any derivation that deviates

from the sequence described above. We introduce two mappings that help us to specify the observer clauses in a more concise manner. For a string w we denote by $\underline{\mapsto}(w)$ the set of all strings that have the same letter sequence as w and have exactly one continuously underlined factor that starts in w's first letter. $\underline{\leftarrow\!\!\mapsto}(w)$ is the symmetric version denoting underlines that start somewhere in the string and are continuous till its end. Both apply also to sets of strings in the obvious way.

The observer realizes the following mapping:

$$\mathcal{O}(w) = \begin{cases} \varepsilon & \text{if } w \in a^+ \cup A^+a^* \cup A^+\alpha a^*, \\ a & \text{if } w \in \underline{\mapsto}(Aa^*) \cup \underline{\mapsto}(A^+\alpha a^*), \\ b & \text{if } w \in \underline{\leftarrow\!\!\mapsto}(A^+a^*), \\ \bot & \text{else} \end{cases}$$

In this way a derivation sequence

$$AAAaaaa \Rightarrow AAA\alpha aaa \Rightarrow \underline{AAA\alpha aaa} \Rightarrow \underline{AAAAaaa} \Rightarrow AAAAaaa$$

produces exactly the output a^7b^7. Further it is important to notice that the application of any rule out of this order leads to a string that is not treated in the first three clauses of $\mathcal{O}$ and thus results in the abortion of the computation. For example, if in the first string $AAAaaaa$ any symbol is underlined or an a different from the left-most one is converted to α, this leads to rejection. Other rule applications are not possible, and for the other configuration the situation is similar.

□

This relatively elaborate example already gives a good idea of the type o computation that can be done with painter rules.

3. The Relation to Restarting Transducers

One aim of the work on transducing observer systems has been the search for new characterizations of known classes of transductions. We have been especially interested in relations to *restarting transducers* which were drived from restarting automata that were introduced by Jancar et al. [7]. Equivalence results of this type would strongly connect the different computational models. So far, however, only close similarities have been found; most noteworthy the following relation between $\mathsf{lr\text{-}T/O}$ and $\mathcal{R}\mathsf{el}(\mathsf{RRWW\text{-}Td})$, the class of transductions computed by RRWW-transducers, via a morphism:

Theorem 4 *[6] For every relation $R \subseteq \Sigma^* \times \Delta^*$ and $R \in \mathcal{R}\mathsf{el}(\mathsf{RRWW\text{-}Td})$, there is a uniform morphism φ and a relation $S \in \mathcal{R}\mathsf{el}(\mathsf{lr\text{-}T/O})$ such that $R = \{(u, v) \mid (\varphi(u), v) \in S\}$.*

When we look at weight-reducing string-rewriting systems instead of length-reducing ones, then we see that the resulting class of relations contains $\mathcal{R}\mathsf{el}(\mathsf{RRWW\text{-}Td})$, but it is not clear whether

this inclusion is proper. Before we can prove this we need to recall the formal definition of a restarting transducer.

A *restarting transducer* (RRWW-Td for short) is a 9-tuple $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$ where Q is the finite set of states, Σ and Γ are the finite input and tape alphabet, Δ is the finite output alphabet, $¢, \$ \notin \Gamma$ are the markers for the left and right border of the tape, $q_0 \in Q$ is the initial state and $k \geq 1$ is the size of the read/write window. Additionally the transition function δ is defined by:

$$\delta : Q \times \mathcal{PC}^{(k)} \to \mathcal{P}(Q \times (\{\mathsf{MVR}\} \cup \mathcal{PC}^{\leq (k-1)}) \cup \{\mathsf{Restart}, \mathsf{Accept}\} \times \Delta^*),$$

where $\mathcal{PC}^{(k)}$ denotes the set of possible contents (over Γ) of the read/write window of T. The transducer works in cycles, where each cycle is a combination of a number of move-right-steps, one rewrite-step and a restart- or accept-step. In case of a tail computation the rewrite-step is optional.

Every rewrite step of the form $(q, v) \in \delta(p, u)$ shortens the tape, that is $|u| > |v|$. After a rewrite step is applied, the read/write window is placed immediately to the right of the string v. Further, the output of the transducer, that is a word in Δ^*, is produced during a restart-step at every end of a cycle or during an accept-step in a tail computation.

Such a transducer T defines a transduction such that every input word $w \in \Sigma^*$ is mapped onto the set of words $z \in \Delta^*$ for which there exists an accepting computation of T during which the output z is produced.

Meta-instructions were used to increase the readability of the behavior of restarting automata (see [9]). Therefore we briefly recall that a tuple of the form $(E_1, u \mapsto u', E_2)$ mirrors the cycle of an RRWW-automaton that reads across the tape content E_1, rewrites a subword u by a shorter subword u' and finally the automaton checks if the part of the tape unseen until then corresponds to E_2. As these meta-instruction describe the rewriting behavior of an automaton they can easily be extended to restarting transducers. Now

$$(E_1, u \mapsto u', E_2; v)$$

is a restarting transducer's meta-instruction, where E_1, E_2, u, u' are defined as for the corresponding automaton and v is the output word produced at the end of this cycle.

Theorem 5 $\mathcal{R}\mathsf{el}(\mathsf{RRWW\text{-}Td}) \subset \mathsf{wr\text{-}T/O}$.

Proof. As mentioned above that for every relation $R \in \mathcal{R}\mathsf{el}(\mathsf{RRWW\text{-}Td})$ there is a relation S realized by a length reducing system and a morphism φ such that $R = \{(u, v) \mid (\varphi(u), v) \in S\}$. Actually this morphism just transforms each input word into a redundant representation with a copy of each letter, and thus it provides some additional space to simulate one cycle of the restarting transducer in two steps of the lr-T/O-system. The latter was needed to "clean up" after applying a rule. Here we extend the idea to weight reducing systems.

Let $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$ be an RRWW-transducer. The weight function ω is defined for all $x \in \Gamma$ as $\omega(x) = 2$. The string-rewriting rules that the wr-T/O-system Ω uses are derived

from the meta-instructions of T. Let

$$t : (E_1, u \cdot x \to u', E_2; v)$$

be such a transition for $u, u' \in \Gamma^*$ and $x \in \Gamma$. We associate to each transition a unique label, here t. From this description we build a wr-T/O-system $\Omega = (\Sigma, W, \mathcal{O})$ such that for each of the meta-instructions above two rules are added to $W \subseteq \Gamma' \times \Gamma'$: $u \cdot x \to u' \cdot t$ and $t \to \varepsilon$. Note that Γ is a subset of Γ', where each label $t \in \Gamma' \backslash \Gamma$. Furthermore, the weight assigned to each t equals 1. Finally for every meta-instruction of T the observer's mapping includes the clause

$$\mathcal{O}(w) = v; \text{ if } w \in E_1 \cdot u' \cdot t \cdot E_2,$$

and $\mathcal{O}(w) = \varepsilon$ if no such label t is present in w. Observe that after the application of any rule from W the weight of a string is at least decreased by 1. Further note, that in this way the proof of correctness is a direct consequence of the simulation of restarting transducers by length reducing systems, cited above. Consequently, accepting meta-instructions of T are simulated by introducing a special symbol t_a, which can not be rewritten anymore.

Finally, T rejects an input word simply by getting stuck, that is, no transition is applicable in the current configuration. As the clauses of the observer mirror directly the move-right steps of T, it will also get stuck. In this situation we have to output $\perp$ to abort the computation of Ω. This can be done by making the observer "complete", that is, the transition function of $\mathcal{O}$ is extended such that every input word w, which is not described by the regular expressions above leads to the following output:

$$\mathcal{O}(w) = \perp.$$

This completes the proof. □

So for the moment we have the chain of inclusions lr-T/O $\subset \mathcal{R}$el(RRWW-Td) $\subset$ wr-T/O, but we do not know if either of them is proper. We suspect that this really is a chain and the three classes are different.

One indicator that wr-T/O is similar to $\mathcal{R}$el(RRWW-Td) is the fact that the class wr-T/O is subject to the same *length-bounded property* as the class $\mathcal{R}$el(RRWW-Td). This is a property that holds for every pair from relations from these classes; it bounds the length of the right-hand side in terms of the length of the left-hand side. In general, we call a relation R *length-bounded*, if and only if there is a constant c, such that for each pair $(u, v) \in R$ with $u \neq \varepsilon$, $|v| \leq c \cdot |u|$ holds. All relations in $\mathcal{R}$el(RRWW-Td) are length-bounded; this can be seen in a way similar to the following proof that this property also holds for wr-T/O.

Lemma 6 *For every relation R in* wr-T/O *there is a constant c such that for each pair $(u, v) \in R$ with $u \neq \varepsilon$, $|v| \leq c \cdot |u|$ holds.*

Proof. Let k be the sum of the weights of all of the symbols in a given input string w. Every rule application in a wr-T/O system must reduce the weight by at least one. Therefore there can be at most k computation steps. In every step the observer outputs one of its output strings. If n is the length of the longest string that can be output in one step, then the total length of the output for the string w cannot exceed $k \cdot n$. if m is the highest possible weight of

an input symbol, then k is at most $|w| \cdot m$. Thus setting $c = |w| \cdot m \cdot n$ makes the statement true. □

Notice that the transduction of Example 3, in contrast, is not length-bounded. Thus painter systems can compute relations that cannot be computed with weight-reducing systems.

4. A Hierarchy of Transductions by Observer Systems

Now we compare the computational power that transducing observer systems achieve with the three types of string-rewriting systems introduced above. From earlier work we know that the relations computed by length-reducing T/O-systems are included in those that are computed by painter T/O-systems [6]. The same is true for weight-reducing systems. In order to show this we first show that weight-reducing systems can be assumed to be in a kind of normal form.

Lemma 7 *Every transducing observer system with weight-reducing rules can be simulated by one with rules that do not increase the string's length.*

Proof. The key observation is the following: a symbol of a given weight k can only be rewritten to a maximum of k symbols. If the maximum weight of an input symbol is m, then a string w cannot increase in length beyond $m \cdot |w|$.

So we code m symbols into one with space symbols filling the unused spots. In the beginning, every letter is written into the first one of the m slots in its position. The main problems are then how to implement rule applications and when to delete empty space. If we just leave empty space there, the distance between different parts of the left-hand side of a rule could become arbitrarily large. If we delete too early, later expansions might become impossible.

For the latter problem we delete as late as possible. This means that a rewriting rule that has k compound symbols on its left-hand side will always write k new symbols unless the total of coded original symbols on its right-hand side is less than k. In that case, each symbol is written by itself into one compound symbol, the remaining positions are deleted. If there are more than k original symbols, say ℓ, then we distribute them in order from left to right in such a way that the first $\ell - \lfloor \frac{\ell}{k} \rfloor k$ original symbols positions contain $\lfloor \frac{\ell}{k} \rfloor + 1$ original symbols; the remaining positions contain $\lfloor \frac{\ell}{k} \rfloor$ original symbols. In this way the total weight of the original symbols can never exceed the number of available slots.

For a rule in the original system we need to introduce several new rules, because we do not know what the left-hand side will look like. The original symbols can be distributed in various ways over compound symbols with space slots inbetween, and the factor might even start and end inside compound symbols. So for an original rule $u \to v$ we need to implement all the possible left-hand sides starting with at least one original symbol in the left-most position and ending with at least one symbol in the right-most position. The symbols between the two ends might occur in just one compound symbol each or with several in one compound symbol. All possibilities must be implemented. Because we do not have completely empty compound symbols, there are only finitely many possibilities.

Note that at the right-most and left-most positions the distribution of symbols on the right-hand side described above might theoretically not work. If the rule's left-most symbol occupies the right-most slot in a compound symbol and more than one symbol should be written into that position, then this is not possible. However, if a symbol occupies the right-most slot, then this means that some rule before (or a sequence of rules) expanded a string roughly by the factor m; this in turn means that the weight of nearly all the original symbols in this string must be one. Thus we can adapt the distribution of original letters slightly to putting only one into the left-most compound symbol. We can proceed in a similar fashion for other small numbers greater than one and on the right end of the rewritten factor.

It is obvious how the observer of the new systems must work: it reads the compound symbols and after each one it changes its state as if it had read all the original symbols contained in the compound one. □

Theorem 8 $\mathsf{wr\text{-}T/O} \subsetneq \mathsf{pnt\text{-}T/O}$.

Proof. Transducing observer systems with weight-reducing rules fulfill the length-bounded property, see Lemma 6. Example 3 shows that with painter systems relations without such a linear bound can be computed. Thus the inclusion is proper.

The key for showing that the inclusion $\mathsf{wr\text{-}T/O} \subset \mathsf{pnt\text{-}T/O}$ holds is Lemma 7, which shows us that for every $\mathsf{wr\text{-}T/O}$-system there is an equivalent one that uses only non-increasing rules. On the other hand, it has been shown several times how general non-increasing (context-sensitive) rules can be simulated by painter systems. The technique was used for example in the proof of Theorem 4.1 and Corollary 4.1 on accepting observer systems [4]. Further, the proof that all relations of deterministic pushdown transducers can be computed via painter systems shows how to accommodate output in this context [5]. For technical details we refer to the referenced descriptions of these constructions. □

So this results follows in a straight-forward manner from earlier proofs. Now we take a look at what happens, if we put a further restriction on the painter systems in Theorem 8. Namely, we require the painter rules to be weight-reducing, too. Systems with this type of rules can still simulate all systems, whose rules are weight-reducing but not necessarily of size one. Thus the length bound of one on the size of the rules does not affect the computational power in this case.

Theorem 9 $\mathsf{wr\text{-}T/O} = \mathsf{wr\text{-}pnt\text{-}T/O}$.

Proof. Again, following Lemma 7 we can suppose that we are dealing with a weight-reducing system without length-increasing rules. The main problem in splitting a rule that rewrites more than one symbol into several painter rules is the following: a rule like $728 \to 555$ increases the weight in the second position, although the total weight is decreased. Here and in what follows the digits are symbols with the corresponding weight. So the painter rule resulting from the rewriting in the central position is $2 \to 5$ and would be weight-increasing. The basic idea to solve this is the introduction of new symbols. In this case, between the weights 2 and 1 we would use a symbol like 2_5. The index indicates the symbol that is in that position after application of the original rule. Unfortunately, there is no fixed bound on the number of indices

that might be necessary for one position. Therefore it will not always be possible to do this kind of book-keeping in just the position, where the weight is increased.

To work out the technical details of our simulation, we first enter into more detail about the simulation of non-increasing (context-sensitive) rules by painter systems, which we have only mentioned in the proof of Theorem 8. Let us look, for example, at such a rule $r : abc \to def$. The basic technique is to first mark all the letters of the left-hand side from left to right with the name of the rule resulting in $a_r b_r c_r$. The corresponding painter rules are $a \to a_r$ etc. In this moment the observer can verify that the entire left-hand side is present and that the rule can be applied. Then the letters are replaced from left to right by def. In our case the new, auxiliary symbols with index r get a weight between the weights of the original and the new symbol. So in our example $a > a_r > d$ if $a > d$.

The resulting system is already weight-reducing in all positions where the weight of the symbol is reduced. It is also easy to deal with letters that are not changed. Marking them and then returning to the original letter would not be weight-reducing; but we can simply ignore them. Since the entire rule is weight-reducing, there must be other symbols that are rewritten like in $abc \Rightarrow^* a_r bc_r \Rightarrow^* dbf$. Therefore some marking is done and thus the observer can produce the necessary output at some point.

Deletions are implemented by putting a space symbol $\square$ in that position. Since every position is rewritten individually, this does not pose a problem to later rule simulations. Instead of marking the consecutive abc we can mark the same letters with arbitrarily many space symbols between them, as for example in the string $a\square\square b\square\square\square c$; the result would be $a_r\square\square b_r\square\square\square c_r$ instead of the string $a_r b_r c_r$.

We do not present technical details on how the observer can guarantee that rules are simulated completely before the simulation of other rules is started etc. This has been treated in several places already and we refer the interested reader for example to the proofs of Corollary 4.1 and the preceding Theorem 4.1 from the work of Cavaliere and Leupold [4]; there, Turing Machine transitions are simulated on strings. So the only problem we have left to deal with are positions, where the weight is increased. We distinguish two cases.

The first and relatively simple case is when the weight is increased, but the position is never rewritten to the same symbol. For this we introduce new symbols: for every symbol a and all symbols b that have bigger weight, there is a new symbol ${}_b a$. This is treated just as b in further rule applications. While it is rewritten to symbols of weight bigger than that of a, we use the indexed version. As soon as the weight decreases beyond that of a we return to the original alphabet. We write the index in front of the symbol, because we will use the spot behind it for a different index further down.

The weight of all the indexed symbols is just between that of a and the next lighter symbol. Notice that a position's weight might increase several times in this way before it finally decreases below a again. Further there might be several possible sequences starting from a given symbol. However, there are only finitely many that do not repeat any symbol of weight bigger than that of a. Those that repeat a symbol are only treated until the first occurrence of this symbol; then a loop starts in this position, and we explain further down how these are treated. So we can

find the finitely many sequences that start with an increase of weight in symbol a by looking at the rules. For each one, we use an own set of indexed symbols. This avoids problems that we could run into with reductions like $1 \Rightarrow 3 \Rightarrow 2 \Rightarrow 5 \Rightarrow 0$ and $1 \Rightarrow 5 \Rightarrow 2 \Rightarrow 0$. Here for the first reduction we need ${}_2 1 >_5 1$, the second one requires ${}_2 1 <_5 1$. With different versions of these symbols for every sequence we can implement both orders.

The more difficult case is an increase in weight that might return to the same symbol in a loop. Such a loop can run several, actually an unbounded number of times in the same position. Therefore no static solution like the fixed index above can work. Instead, we make use of the fact that an increase in the weight in one position must be compensated by a decrease in some other position in the same rule; that position can be used to simulate the weight-increase by stretching its reduction into several steps. Since different runs of the same loop necessarily are compensated by increases in weight in different positions, we can use the latter to simulate the respective runs of the loop there. We illustrate the basic idea with an example. Instead of

$$19 \overset{19\to 54}{\Rightarrow} 54 \overset{5\to 3}{\Rightarrow} 34 \overset{34\to 12}{\Rightarrow} 12 \overset{1\to 0}{\Rightarrow} 02$$

with the loop $1 \Rightarrow 5 \Rightarrow 3 \Rightarrow 1$ in the first position we will do a derivation like

$$19 \Rightarrow^* 1^{pool}4_5 \Rightarrow^* 1^{pool}4_3 \Rightarrow^* 1^{pool}2_1 \Rightarrow^* 02.$$

Here the first position is marked by a specific name for this loop, in our case the word *pool* in the exponent. This position must not be rewritten while the loop is not finished. Instead the contents of the first position are now updated in the index of the second position. This can be done even if the second position is rewritten in the meantime. The weights of indexed symbols n_k are between the weights of n and $n-1$ with the weight in the index decreasing.

For every loop there is one fixed position from the corresponding first rule, where the loop actually runs in the index. This might be on either side of the position where the loop runs. The loop and the index might not be directly next to each other, for example if the first rule is $13337 \to 53332$ starting a loop in the first position. The only possible choice for the index is the fifth position, because it is the only one where the weight is decreased. It is important to chose one fixed position for every loop. Then the observer can know from the label where it has to look for the corresponding index.

However, there are two questions that require a clarification and refinement of this strategy: We must be clearer on what we call a loop; and a loop can run more than once in the same position.

To start with, we need to identify all possible loops that the original string-rewriting system can produce in one position. This can be done by an analysis of the rules. Since we use a different label and index for every loop, we can only treat finitely many distinct loops. For every loop one index is reserved in all of the possible symbols; we will call these the *slots* for the respective loops. For every rule that can start a loop in some position, we select one fixed position of the ones that are decreased in weight by that rule. Thus the symbol that is in that position is rewritten anyway. In this symbol's slot that corresponds to this loop we write the symbol that is next in the loop like in the example for the loop called *pool* above.

To clarify what we mean by a loop, first note that we need to distinguish loops with different sequences of rules in the original string-rewriting system. Thus the same sequence of symbols in one position might result from distinct loops like in

$$19 \overset{19\to 27}{\Rightarrow} 27 \overset{2\to 1}{\Rightarrow} 17 \overset{17\to 25}{\Rightarrow} 25 \overset{2\to 1}{\Rightarrow} 15 \overset{15\to 23}{\Rightarrow} 23 \overset{2\to 1}{\Rightarrow} 13 \overset{13\to 21}{\Rightarrow} 21 \overset{2\to 1}{\Rightarrow} 11.$$

Here we have the sequence $1{\Rightarrow}2{\Rightarrow}1{\Rightarrow}2{\Rightarrow}1{\Rightarrow}2{\Rightarrow}1$ in the first position. But this is actually produced by three distinct loops.

Further, we only consider minimal loops in the following sense: rule applications that change only neighboring positions while they leave the position in question unchanged are ignored. Otherwise such a loop could become arbitrarily long, and we would not only deal with a finite number of them. For example, the loops

$$19 \overset{19\to 27}{\Rightarrow} 27 \overset{2\to 1}{\Rightarrow} 17 \quad \text{and} \quad 19 \overset{19\to 27}{\Rightarrow} 27 \overset{27\to 2}{\Rightarrow} 2 \overset{2\to 1}{\Rightarrow} 1$$

are considered the same, since the derivation step $27 \overset{27\to 2}{\Rightarrow} 2$ does not change the first position, where actually have the loop. Note that formally the 2 is rewritten, but the result is again a 2. So the position just serves as context. As we have explained above, we do not rewrite the 2 in such a case. In the same way we ignore such a rewriting, when we analyze the different possible loops. If we did not identify these formally different loops, we would get infinitely many versions of this loop in strings from 197^*:

$$197^k \overset{19\to 27}{\Rightarrow} 27^{k+1} \overset{27\to 2}{\underset{k+1}{\Rightarrow}} 2 \overset{2\to 1}{\Rightarrow} 1$$

works for every natural number k. Only by considering all these loops the same we can ensure that there are only finitely many different loops.

Finally, we distinguish nested loops. If there is, for example, a derivation $1 \Rightarrow 3 \Rightarrow 5 \Rightarrow 3 \Rightarrow 1$ is treated as two different loops: $3 \Rightarrow 5 \Rightarrow 3$ and $1 \Rightarrow 3 \Rightarrow 1$. Of course, the latter one is not possible as such. But we stop its simulation at the first occurrence of 3 and resume it only when 3 is reached again. Otherwise we would again have infinitely many different loops, because the shorter one could run several times. So here the exponent for the second loop is put on a symbol in the index of another symbol. The example in Figure 2 shows in detail, how this works. In the same way we treat loops that start during a sequence with an increase in weight that is not a loop. Also there we add the exponent of the loop to the symbol in the index. Note that a sequence with an increase in weight that starts within a loop is just part of the loop and does not need to be treated separately.

Note that the system thus makes a decision beforehand whether an increase in weight will lead to a loop or to a reduction without loop and to which one of the possible several possibilities. In some cases both might be possible, but according to the choice taken in the first step only one path can be followed according to what the symbol is rewritten to. But for every computation of the original system there exists one that leads to the same result.

It remains to explain how we end a loop. Take for example the simple loop $\circlearrowleft$: $15 \Rightarrow 32 \Rightarrow 12$. After the simulation we arrive at the string $1^{\circlearrowleft}2_1$. The $1^{\circlearrowleft}$ remains unchanged until this position

is rewritten to a weight below 1. Everything else is done in the index of 2 just as we do it in the original position. So if the weight is increased without starting a loop, we obtain a string like $1^{\circlearrowleft}2_{7\,1}$. If another loop starts, then we obtain a string like $1^{\circlearrowleft}2_{1\circlearrowleft}$. In this way, it is clear at every moment, in which position a given marked position is simulated. In our case, $1^{\circlearrowleft}$ tells us to look one position to the right in the slot for the loop $\circlearrowleft$. There we find out that the next loop is $\circlearrowleft$ and thus we have to look at the respective slot in the position for this loop relative to the original position, not to the position where we have simulated the first loop.

When finally the weight is decreased in the loop's position, we can eliminate all the loop symbols. Further, all the indices that have been used are marked as used by a $\times$. Of course, they cannot be used again. Otherwise the resulting symbol would have the same weight again.

The weight of all these new symbols is ordered as follows: Suppose that a and b are two symbols such that there is no symbol with a weight between their weights and $a > b$. Then all new symbols indexing a receive a weight between that of a and that of b. The order of the weight of the slots for loops is as follows: the highest weight is for the index with all slots empty; then comes one slot active, all others empty; next is one slot marked by $\times$. The next series is for two slots that are active; first comes the configuration where both loops have just been started but have not made any further step; then come the two configurations where either one or the other has made exactly one step; next are all configurations where tow steps have been made and so on; for the same number of steps equal weight can be given, since we will never have to go from one configuration to the other. Now come the configurations with one slot active and one blocked, then two slots blocked, then three active and so on. Derivations corresponding to derivations of the original system will always decrease the weight in every step. In this way, we can do even several simulations of loops in the same position at the same time.

We now illustrate the operation of the system for two nested loops in Figure 2. The outer loop is $\circlearrowleft\colon 1 \Rightarrow 3(\Rightarrow 5 \Rightarrow 3) \Rightarrow 1$ and the inner one is $\circlearrowleft\colon 3 \Rightarrow 5 \Rightarrow 3$. The example also shows how the empty symbols at deleted positions continue to be used for the simulation of the loops.

So one position can use slots that are arbitrarily far from it. Now in principle it might not be possible to uniquely identify one loop running in a slot with the corresponding position that is marked by with the loop's name. For example, the same loop might run in positions one and three, and it chooses the slot just to the right of the position where it runs. If it runs several times in position one, it can use its slots also in position four. If now the same loop starts in position three, we have two pointers pointing to the same slot in position four.

And then there are also scenarios like the following:

$$913 \overset{13\to 21}{\Rightarrow} 921 \overset{2\to 1}{\Rightarrow} 911 \overset{911\to 613}{\Rightarrow} 613 \overset{13\to 21}{\Rightarrow} 621 \overset{2\to 1}{\Rightarrow} 611 \overset{611\to 313}{\Rightarrow}$$
$$313 \overset{13\to 21}{\Rightarrow} 321 \overset{2\to 1}{\Rightarrow} 311 \overset{111\to 013}{\Rightarrow} 013 \overset{13\to 21}{\Rightarrow} 021.$$

Such a loop can run more than n times for a string of length n. A definite upper bound on the number of times a loop can run is n times the maximum weight of any symbol; this is actually an upper bound on the total number of possible rule applications, because every rule must decrease the total weight by at least one. So far, however, we only have n slots for every loop.

rule	original string				string for simulation			
$19 \to 36$	1	9	6	6	1	9	6	6
	⏟							
$36 \to 5$	3	6	6	6	$1^{\circlearrowleft}$	$6_{3,\_}$	6	6
	⏟							
$5 \to 3$	5		6	6	$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},5}$	6	6
	⇓							
$36 \to 5$	3		6	6	$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},3}$	6	6
	⏟							
$5 \to 3$	5			6	$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},3^{\circlearrowleft}}$	$\square_{\_,5}$	6
	⇓							
$36 \to 5$	3			6	$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},3^{\circlearrowleft}}$	$\square_{\_,3}$	6
	⏟							
$5 \to 3$	5				$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},3^{\circlearrowleft}}$	$\square_{\_,3^{\circlearrowleft}}$	$\square_{\_,5}$
	⇓							
$3 \to 1$	3				$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},3^{\circlearrowleft}}$	$\square_{\_,3^{\circlearrowleft}}$	$\square_{\_,3}$
	⇓							
$1 \to 0$	1				$1^{\circlearrowleft}$	$\square_{3^{\circlearrowleft},3^{\circlearrowleft}}$	$\square_{\_,3^{\circlearrowleft}}$	$\square_{\_,1}$
	⇓							
	0				0	$\square_{\times,\times}$	$\square_{\_,\times}$	$\square_{\_,\times}$

Figure 2: Two nested loops: a loop $\circlearrowleft$: $3 \Rightarrow 5 \Rightarrow 3$ runs several times inside a loop $\circlearrowleft$: $1 \Rightarrow 3 \Rightarrow 5 \Rightarrow 3 \Rightarrow 1$. These are treated as two separate loops.

To remedy the two problems we have just described we modify our string-rewriting system in the following way: for every loop and the position where it is run in the index, we modify the rule from $a \to b$ to $a \to b\square$ if the position is to the right of the loop and to $a \to \square b$ if the position is to the left of the loop. In this way, for every time the loop runs there is one space symbol next to the position where the index runs. If the weight of all the other symbols compared to the space symbol is sufficiently big, then all these rules are still weight-reducing.

Thus the index never reaches a position where another instantiation of the same loop might be logged, and every time we use a slot we create a new one just next to it. In this way the first problem described cannot occur anymore, the second one is also solved, because additional slots are made available for loops that run many times. Of course, our system is not a painter system anymore. But from Lemma 7 we can see that there exists an equivalent painter system, since the rules there have the same length of the left-hand sides of the original rules – in our case length one.

We illustrate this last technique for the loop $\circlearrowleft$: $19 \overset{19\to 51}{\Rightarrow} 51 \overset{5\to 1}{\Rightarrow} 11$ and the following derivation:

$$1999 \overset{19\to 51}{\Rightarrow} 5199 \overset{5\to 1}{\Rightarrow} 1199 \overset{1\to \lambda}{\Rightarrow} 199 \overset{19\to 51}{\Rightarrow} 519 \overset{5\to 1}{\Rightarrow} 119 \overset{1\to \lambda}{\Rightarrow} 19 \overset{19\to 51}{\Rightarrow} 51 \overset{5\to 1}{\Rightarrow} 11 \overset{1\to \lambda}{\Rightarrow} 1.$$

This becomes

$$1999 \Rightarrow^* 1^{\circlearrowleft}1_5\Box 99 \Rightarrow^* 1^{\circlearrowleft}1_1\Box 99 \Rightarrow^* 1^{\circlearrowleft}\boxtimes_1\Box 99 \Rightarrow^* 1^{\circlearrowleft}\boxtimes_{1\circlearrowleft}\Box_5 1\Box 9 \Rightarrow^* 1^{\circlearrowleft}\boxtimes_{1\circlearrowleft}\Box_1 1\Box 9 \Rightarrow^*$$
$$1^{\circlearrowleft}\boxtimes_{1\circlearrowleft}\Box_1\boxtimes\Box 9 \Rightarrow^* 1^{\circlearrowleft}\boxtimes_{1\circlearrowleft}\Box_{1\circlearrowleft}\boxtimes_5\Box 1 \Rightarrow^* 1^{\circlearrowleft}\boxtimes_{1\circlearrowleft}\Box_{1\circlearrowleft}\boxtimes_1\Box 1 \Rightarrow^* 1^{\circlearrowleft}\boxtimes_{1\circlearrowleft}\Box_{1\circlearrowleft}\boxtimes_1\Box\boxtimes .$$

For better understandability we have used $\boxtimes$ for those spaces that are created by deleting the symbols 1 and the usual $\Box$ for those spaces that are introduced by initiating the loop. We see that the index in the originally third and fourth position of the string are never used by the loop in the first position. So even if the same loop starts in the second position, the corresponding slot in the originally third position will always be free.

We do not enter into details of how to guarantee that the rewritings in the different positions are coordinated. As we have stated above, in principle it works exactly like the simulation of context-sensitive rules by painter rules. The rewriting rules should be clear from our description, because painter rules always rewrite just one symbol. In conclusion, we have a pnt-T/O-system that simulates the original wr-T/O-system and produces the same output.

□

In the last part of this proof we use the reduction of weight at the start of a loop for the (temporary) creation of some additional space for our book-keeping. This is a technique similar to one that Jurdzinski and Otto, when they showed the equivalence between finite-change automata and shrinking restarting-automata [8].

From the last part of the proof or simply from Lemma 7 we can see that the use of context-free rules (left-hand side of length one, right-hand side arbitrarily long) does not increase the computational power compared to the use of painter rules as long as we use only weight-reducing rules. Denoting by wr-cf- the class of all weight-reducing context-free string-rewriting systems we can slightly generalize Theorem 9.

Corollary 10 wr-T/O = wr-pnt-T/O = wr-cf-T/O.

This is in contrast to the situation for general painter and context-free systems. There the former allow linearly bounded computations, while the additional use of just one context-free rule leads to computational completeness [4].

The next question concerns the relation between T/O systems with length-reducing and those with weight-reducing rules. A simple observation places weight-reducing T/O-systems above the latter class.

Theorem 11 lr-T/O $\subset$ wr-T/O.

Proof. Every length-reducing string-rewriting system is weight-reducing for the weight function that assigns weight one to every symbol. □

Actually, this result also follows directly from Theorem 5 and the fact that length-reducing T/O-systems can be simulated by RRWW-transducers. However, we are unable to determine

whether this inclusion is proper. Intuitively, we suspect that it is. A weight-reducing systems can in some sense use the input several times, while length-reducing rules necessarily consume it during the computation. To show what we mean by using the input several times, we sketch the proof of the following result.

Theorem 12 *The class* wr-T/O *is closed under intersection.*

Proof. After Theorem 9 we can assume that all relations in wr-T/O are computed by systems that only use painter rules. We only sketch how a wr-pnt-T/O can compute the intersection between two relations that are computed by systems from this class, let us call them Ω_1 and Ω_2. Further, let their observers be $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively. For both relations the length-bounded property holds, and let k be a constant that fulfills this property for both relations.

The system that computes the intersection works on four tapes that are simulated on the symbols of the input string. First, two copies of the input string w are created. Then the computation of Ω_1 on w is simulated and the output is written on the third tape. Here we need to write k symbols into one. Then the output of Ω_2 on w is computed and written on the fourth tape.

It remains to compare the two outputs. Only if they are equal we now output exactly this string. During the steps before no output has been produced. Since Ω_1 and Ω_2 are both weight-reducing, appropriate weights can be chosen such that the two successive computations are simulated by weight-reducing rules. Also storing the outputs can be done in a weight-reducing way, since symbols that have been written are not changed any more; thus the empty storage should produce the highest weight, every further symbols from left to right should decrease the weight. □

The same tactics does not work in a direct way for length-reducing rules, just as it would not work for general weight-reducing rules. Already the computation of the first output could reduce the string to just one symbol, and no copy can be stored on a second track. It also does not seem to be possible to simulate the two computations in parallel, since the rewritings might take place at different positions. However, it is not clear how to formalize this. Therefore the properness of the inclusion remains an open problem.

This parallels the situation for RRWW automata. These are like RRWW transducers, but do not produce output; rather they accept or reject a word. There automata with length-reducing rules can be simulated by those with weight-reducing rules (which are called shrinking restarting automata) [8]. But it remains unknown whether the weight-reducing rules lead to bigger computational power.

5. Conclusions

The results of the preceding sections are summarized in Figure 3. Lines indicate inclusions from bottom to top, arrows indicate proper inclusions. Dotted arrows show the relation via morphisms in the style of Theorem 4. what we have not treated explicitly in the relation of

the two classes on top, but it follows from results on accepting observer systems. Painters are equally powerful as linear bounded automata, context-free rules allow the simulation of arbitrary turing Machines [4]. Thus pnt-T/O is the class of transduction that can be computed by Turing Machines with a linear space bound, while cf-T/O are all computable relations; these two classes are obviously not equal.

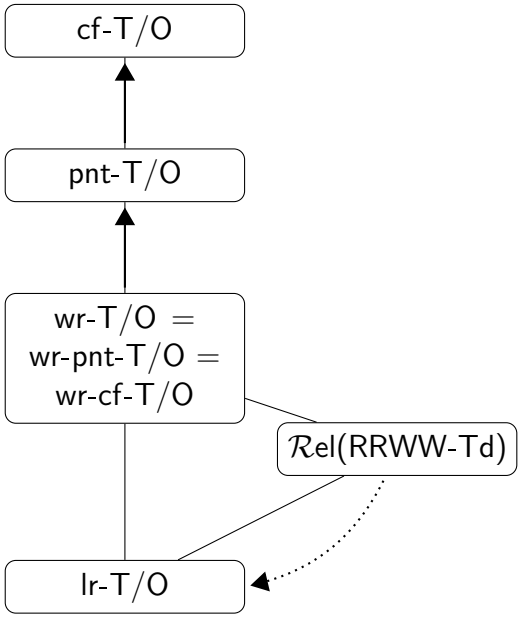

Figure 3: The hierarchy of transducing observer systems with the relation to RRWW-tranducers.

The main open questions regard the properness of the inclusions lr-T/O $\subset$ wr-T/O, lr-T/O $\subset$ $\mathcal{R}$el(RRWW-Td), and $\mathcal{R}$el(RRWW-Td) $\subset$ wr-T/O. As argued above and in earlier work [6], we expect all three of these inclusions to be proper. Theorem 12 might be a hint that $\mathcal{R}$el(RRWW-Td) and wr-T/O are different, because the former might not be able to compute intersections. We also doubt that the morphism in Theorem 4 can be eliminated, which would mean that also lr-T/O and $\mathcal{R}$el(RRWW-Td) are different.

Note that it does not make much sense to look at lr-pnt-T/O systems like we have looked at wr-pnt-T/O systems here: length-reducing rules of length one are simply deleting rules for single symbols.

References

[1] A. AHO, J. ULLMAN, *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Upper Saddle River, NJ, 1972.

[2] R. BOOK, F. OTTO, *String-Rewriting Systems*. Springer, Berlin, 1993.

[3] M. CAVALIERE, P. LEUPOLD, Evolution and Observation — A Non-Standard Way to Generate Formal Languages. *Theoretical Computer Science* 321 (2004), 233–248.

[4] M. CAVALIERE, P. LEUPOLD, Observation of String-Rewriting Systems. *Fundamenta Informaticae* 74 (2006) 4, 447–462.

[5] N. HUNDESHAGEN, P. LEUPOLD, Transducing by Observing. In: H. BORDIHN, R. FREUND, T. HINZE, M. HOLZER, M. KUTRIB, F. OTTO (eds.), *NCMA*. books@ocg.at 263, Österreichische Computergesellschaft, 2010, 85–98.

[6] N. HUNDESHAGEN, P. LEUPOLD, Transducing by Observing and Restarting Transducers. In: R. FREUND, M. HOLZER, B. TRUTHE, U. ULTES-NITSCHE (eds.), *NCMA*. books@ocg.at 290, Österreichische Computer Gesellschaft, 2012, 93–106.

[7] P. JANCAR, F. MRÁZ, M. PLÁTEK, J. VOGEL, Restarting Automata. In: H. REICHEL (ed.), *FCT*. Lecture Notes in Computer Science 965, Springer, 1995, 283–292.

[8] T. JURDZINSKI, F. OTTO, Shrinking Restarting Automata. *Int. J. Found. Comput. Sci.* 18 (2007) 2, 361–385.

[9] F. OTTO, Restarting Automata. In: Z. ÉSIK, C. MARTÍN-VIDE, V. MITRANA (eds.), *Recent Advances in Formal Languages and Applications*. Studies in Computational Intelligence 25, Springer, 2006, 269–303.

[10] A. SALOMAA, *Formal Languages*. Academic Press, New York, 1973.