Turing Machines for Dummies
why representations do matter

Peter van Emde Boas

ILLC, FNWI, Universiteit van Amsterdam, PO Box 94242, 1090 GE Amsterdam
Bronstee.com Software & Services B.V., Heemstede
Dept. Comp. Sci. University of Petroleum, Chang Ping, Beijing, P.R. China
`peter@bronstee.com`

Abstract. Various methods exists in the literature for denoting the configuration of a Turing Machine. A key difference is whether the head position is indicated by some integer (mathematical representation) or is specified by writing the machine state next to the scanned tape symbol (intrinsic representation).
From a mathematical perspective this will make no difference. However, since Turing Machines are primarily used for proving undecidability and/or hardness results these representations do matter. Based on a number of applications we show that the intrinsic representation should be preferred[1].

1 The Turing Machine model

Given the nature of the meeting I expect that the dummies mentioned in my title will not be present in the audience. Still I believe that it is useful to start with a description of the Turing Machine model as we are supposed to know it.

The simplest version of the Turing machine is defined in mathematical terms by a tuple $M = \langle K, \Sigma, P, q_0, q_f, b, \Delta \rangle$. The machine has only a single one-dimensional tape, with tape alphabet Σ, and a set of internal states K. The program P of the finite control consists of a set of quintuples $\langle q, s, q', s', m \rangle \in K \times \Sigma \times K \times \Sigma \times \Delta$. Here the set $\Delta = \{L, 0, R\}$ denotes the set of possible head moves : Left, stay put or Right . The meaning of this quintuple is: if in state q the head is scanning symbol s then print symbol s', perform move m and proceed to state q'. The states q_0 and q_f are two special elements in K denoting the initial and the final state respectively. The symbol b is a special tape symbol called blank which represents the contents of a tape-cell which never has been scanned by the head.

In this single tape model there is no special input or output tape. The input is written on the unique tape in the initial configuration with the unique head scanning the leftmost input symbol. When started the computation will perform applicable instructions on the configuration up to the point in time where some termination condition is satisfied (if such a configuration arises at all). Various termination conditions are used in the literature. Absence of an applicable

[1] to appear in G. Gotlob & J Stuller eds., Proceedings SOFSEM 2012, Springer LNCS

instruction is a possible termination condition, but one can also use specially designed halting states to which one can ascribe a quality of being accepting or rejection. Another possibility is to modify the instruction format allowing the state from the machine to disappear, leaving a configuration consisting of tape symbols only.

If one wants the single tape model to produce output one obtains such output by an ad-hoc convention from the final configuration (for example, the output consists of all non-blank tape symbols written to the left of the head in the final configuration).

Note that if we denote configurations of the single tape machine in the format $\$\Sigma^* K \Sigma^* \$$, with the state symbol written in front of the currently scanned tape symbol, the transitions between two successive configurations are described by a very simple context sensitive grammar. In this grammar one includes for example for the instruction $\langle q, s, q', s', R \rangle$ the production rules $(qst, s'q't)$ for every $t \in \Sigma$, together with the rule $(qs\$, s'q'b\$)$ for the blank symbol b. Similar rules encode the behaviour of left-moving instructions or instructions where the head doesn't move.

Aside from this basic model there are various more extended models containing multiple tapes, multiple heads on a single tape, semi-infinite tapes, multi dimensional tapes, or extensions of the instruction repertoire by allowing heads on the same tape to jump to each other's position in constant time. In the multi tape version there can be special tapes reserved for the input or output, subject to special restrictions like right moving heads only, no printing on input and no rewriting of any previously printed symbol on the output.

Fact is that all these models are equivalent from the perspective of Theoretical Computer Science; they all satisfy the Invariance Thesis stating that they simulate each other with polynomial time and constant factor space overheads. For a more detailed discussion of the issues involved I refer to my chapter in the 1990 Handbook of Theoretical Computer Science [24] .

Turing and his model

Given the large number of variations of the Turing Machine model, one might consider to go back to the master himself, and accept the definition given in his 1936 paper [21] to be the official one. This approach, however, is not going to work.

Turing's original paper reads as the work of an engineer or programmer avant la lettre, rather than that of a mathematician or logician. He starts with an intuitive description and justification (which is expanded upon in a later section of the paper). The model he describes is more general than the standard models in the literature since he allows a finite sequence of atomic instructions rather than a single atomic step to be the action provoked by a state-tape symbol observation. The reader should observe here that for Turing a configuration means such a state-tape symbol pair; what we call a configuration (also called instantaneous description) is called a full configuration by Turing. In the paper Turing proceeds by outlining a macro-language allowing for iterated instructions in order to perform the required sweeps, copying actions, and searches for symbol occurrences which are found in all explicit programs for this device.

The model as presented is actually a linearisation of a two track single tape model where one track is used for the real data (digits 0 and 1) and the other track is used for auxiliary symbols called markers.

With all his flexibility there is an extension generally accepted today which is explicitly rejected: the use of nondeterminism. In the beginning of section 2 Turing considers *ambiguous* configurations where the machine has a choice but such a choice should be made by an external operator. As argued elsewhere [17] nondeterminism became an accepted notion only in the 1950-ies presumably in order to obtain the desired characterisations in Automata Theory.

The more restricted model which allows atomic actions only is introduced for the purpose of constructing the Universal Turing Machine. This construction then opens the way for the famous results on the unsolvability of the Halting problem and the undecidability of the Hilbert Calculus. These sections read almost like a research plan outline, the details of which can be found in later textbooks like Kleene [9]

Using the model

Why is the Turing Machine model so popular in Computer Science? It is not because of its close resemblance to real computers, since the Random Access Machine provides us with an idealised but more realistic model. It is also not popular because it is easy to write programs for Turing Machines. If you have ever taught a basic Theory class you will know that programming exercises on a Turing machine are very easy to state, but turn out to be rather cumbersome to solve, and boring to grade.

Instead I believe that the main reasons are that the model is very easy to understand and yet it is a model with universal computational power. Considering its conceptual simplicity it is hard to believe at first sight that the model is universal. But as it turns out it is not to hard to prove that (after having made the right choice of representation) one can simulate real computations as given by Kleene schemes of recursive functions, as well as alternative models of computation. The hardest part of proving the equivalence of the standard universal computational models after all is creating the coding mechanisms in Arithmetic (using Gödel numberings) in order to show that recursive function schemes can simulate symbol manipulating devices like Turing Machines.

There are a number of deep results on solving actual problems on Turing Machines. My favourite examples are the Hennie-Stearns oblivious simulation of multi tape machines on two tapes [6], Slisenko's algorithms for recognizing palindromes [16] and related string properties in real-time, and the the oblivious real-time minimal space simulation of a finite collection of counters by Vitányi [27]. However the dominant use of the Turing Machine model in Theory are negative results: Undecidability and/or Hardness results. Due to the undecidability of the Halting problem we can derive that every formal system capable of coding Turing machine computations in such a way that termination of these computations becomes expressible within the system will inherit this curse of undecidability. Similarly any formalism which can express the fact that some Turing Machine computation terminates within a given number of steps will have a satisfiability or validity problem which can't be solved in much less steps, provided this expression is sufficient succinct. It is in the context of the construction of this type of reductions that the advantage of the combinatorial simplicity of the Turing Machine model (particularly in its most simple form: the single tape version) become prominent.

Reductions as sketched above are a main topic in Theoretical Computer Science. They are used as a tool for measuring the complexity of various problems. A problem A is reduced to a problem B if there exists some explicit, efficient and sufficiently succinct procedure for transforming instances of problem A into one (or several) instances of problem B such that solving the obtained B-instances will provide an effective answer to the original A-instance. The standard use of such a reduction is to show that problem B is difficult, by reducing some problem A to B where it is already known that A is difficult. But how do we know already that problem A is hard? This knowledge arises from the fact that we (or someone else) has previously reduced some other hard problem to A. This chain of reductions must originate in some problem generally accepted to be hard, and that role is played by termination problems on Turing Machines. It is therefore useful to single out these reductions which start with problems on Turing Machines; I will call them *Master Reductions*.

Representing Machine Configurations

In this presentation I will illustrate that, in order to obtain really efficient Master Reductions, one more ingredient is required: the choice of the right representation of Machine configurations.

A Turing Machine configuration is fully described by its three components: the state of the machine, the complete contents of the machine tapes, and the positions of the reading heads on the various tapes. Hence for a Mathematician it is evident how to represent this configuration: a tuple containing these three components is all we need, so the representation will become an object like $< q, x_1x_2 \ldots x_k, i >$ with q denoting the state, $x_1x_2 \ldots x_k$ denoting the tape contents and i denoting the head position. However, we have already encountered the more graphical representation which has become standard in our community. It is a representation where the state is inserted in the string of tape symbols, preceding the scanned symbol, or alternatively, printed above or below the scanned symbol (but this will make the printing and/or typesetting much harder). In our example configuration the representation will become $x_1x_2 \ldots qx_i \ldots x_k$; moreover in order to make explicit where the used segment of the tape begins and ends this representation frequently is extended using endmarkers, resulting in a string like $\$x_1x_2 \ldots qx_i \ldots x_k\$$.

In the sequel I will call representations of the first kind *Mathematical Representations*, whereas those of the second kind will be called *Intrinsic Representations*.

Given a sequence of successive configurations which occur during some computation, one can represent this section of the computation simply by the sequence of their encodings. However, a much clearer representation is obtained by writing these configurations below each other thus giving rise to the so called *time-space diagram* representation of the computation. Both representations can be used for this purpose.

If the mathematical representation is used it is evident what a proper allignment of these configurations should be: the content of some tape square may change, but the square itself maintains its identity. So each column in the diagram corresponds to a single fixed tape cell. To the left of the diagram one writes the state symbol and an index of the position of the tape head, which index also indicates the unique region in the diagram where during the transition connecting the two configurations changes may occur.

If we use the intrinsic representation we face the problem that the state symbol requires an additional position which moreover wanders through the configuration during the computation. One can solve this problem by introducing extra empty columns in the time-space diagram used only for storing the state symbol if the head arrives at that position. The easiest solution is to combine the state symbol and the scanned tape symbol into a pair which becomes a new symbol in an extended alphabet. Now the tape cells remain within their column in the diagram and yet the effect of the transition becomes entirely local.

There exist also versions of the time-space diagram where the state symbol/head position remains in a fixed column, while the tape symbols are moving around. This diagram illustrates a version of the machine where the head remains fixed but the tape moves; something which is not very realistic given the fact that the tape is supposed to be infinite.

In the literature one finds also some intermediate representation which I will call the *Semi-Intrinsic Representation.* Here the state symbol is written before the sequence of tape symbols, but the scanned tape cell is indicated by some marker (an arrow preceding the scanned symbol, or the scanned symbol is underlined).

I believe that the first time the advantage of the use of an intrinsic representation was explicitly mentioned in the literature is in a simple lemma (2.14) on page 38 in the thesis of Larry Stockmeyer [19]. Stockmeyer introduces for his standard model (the nondeterministic multi tape version with input and output tape) a version of the mathematical representation (page 20). Later he introduces the single tape version as a "technical useful model", and for encoding configurations of the latter model he uses a version of the intrinsic representation (page 34-35).

The lemma states that there exists some compatibility relation between triplets of the symbols used in the time-space diagram such that one row represents a proper successor configuration of the row above it, if and only if all triplet pairs formed by three successive symbols in the same position in the two rows be-long to this compatibility relation[2]. So consider three successive symbols in some row and the three symbols below it; if it is always the case that these triplets are compatible then the entire diagram describes a segment of the computation of our machine. Moreover, this compatibility relation is completely determined by the program of our machine. Note that this locality condition does not hold for the semi-intrinsic representation, since the state symbol information is located at a distance.

I now can state the thesis I want to discuss in this presentation: *For the construction of Master Reductions the Intrinsic Representation is by far more useful than the Mathematical Representation.* Stated otherwise: if you are looking for a Master reduction, use the intrinsic representation and life will be easy.

In the sequel of this paper we will illustrate the advantage of the intrinsic representation in relation to the following topics. We first reconsider the relation between machine computations and grammar derivations on which the fundamental characterisation of the Chomsky Hierarchy in basic Automata theory is based. Next we consider the two most common versions of a master reduction for the class NP: the Cook-Levin reduction to Satisfiability and the reduction to tiling problems. We discuss how Stockmeyer used his locality lemma in order to prove hardness results in the theory of (Extended) Regular Expressions. The final part of the paper illustrates the importance of the intrinsic configuration for proving that various models for Parallel Computation satisfy the Parallel Computation Thesis which states that such models recognize in polynomial time exactly what sequential models recognize in polynomial space. I hope that these examples which seem harder if not impossible to perform using a mathematical representation will convince the audience of the validity of my thesis.

2 The Chomsky Hierarchy and the corresponding Automata

The core topic of an undergraduate course on Automata Theory is to provide a proof of the machine based characterisations of the four levels of the Chomsky Hierarchy: Regular Grammars vs. Finite Automata, Context Free Grammars vs. Push Down Machines, Context Sensitive Grammars vs. Linear Bounded Automata and finally Unrestricted Grammars vs. Turing Machines.

Proving these characterisations (once the required mathematical concepts have been introduced) requires a proof in two directions: one must show that the machine can simulate the grammar, and conversely that the machine computations can be simulated by grammar rules.

One may look therefore into the influence of the choice of representation of machine configurations on the proofs of these characterisations. It is evident that the intrinsic representation for this purpose is the right tool: individual transitions are fully described by context sensitive rules involving no more than

[2] note that Stockmeyer speaks in this lemma about a compatibility *function*, but in his language functions are partial and multivalued so he intends this to be a relation

three symbols on the left hand side (two symbols if the state symbol is paired with the scanned tape symbol - the second symbol is required for moving the head).

Given this insight the characterization of the type-0 languages becomes almost trivial. Turing machines are symbol manipulators, so it is not difficult - given some grammar - to write a Turing Machine program which starts out writing the start symbol, performing substitutions allowed by the grammar until the resulting string appears. The Turing machine can erase (or insert) symbols by shifting parts of the tape contents one square to the left (right). Conversely, given the fact that the machine configurations are derived by means of context sensitive rules, it is easy to construct a grammar which first generates an initial configuration and subsequently simulates the Turing Machine computation towards its accepting state. Since in this final configuration a substantial number of auxiliary symbols still may remain written on the tape, a final cleanup sweep where the undesired symbols are erased is required.

A similar proof will work for the context sensitive grammars vs. the linear bounded automata. However the prohibition of erasing rules requires a careful treatment of the boundary markers of the tape segment containing the input. These boundary markers are required since the machine must be capable of feeling the end of the tape, while on the other hand the machine is not allowed to leave the input string. This problem can be solved by pairing the end marker with the first(last) symbol and rewriting these marked symbols at the end of the production.

A comparable verbatim simulation between the machine configurations and the intermediate phrases of the derivation process is not possible for the two remaining cases of the context free and regular languages. The main reason is that in the grammar based world during the generation process only the initial part of the generated word is present, whereas the complete word exists already at the start of the machine computation.

One can however preserve the flavour of such a simulation. The problem is resolved by removing from the machine configuration the part of the input word which still has te be read in the future. The configuration consists of the part of the input already read (the part of the output already generated) followed by a machine state (nonterminal symbol). For the context free case this machine state is paired with the topmost stack symbol and the remaining stack symbols are concatenated in reverse order (paired up with the intermediate machine state attained when that stack symbol is eventually removed).

As is well known the choice freedom on the grammar side results in the machines becoming nondeterministic. This nondeterminism subsequently can be eliminated in the regular grammar case and for the unrestricted Turing Machines. For the context free grammar case nondeterminism has been shown to be required, whereas its necessity for the linear bounded machines is known as the famous LBA problem which still is unsolved.

We conclude that the intrinsic representation is used in Automata Theory as we know it today. This is not a formal proof that we can't build a version of Automata Theory based on the mathematical representation, but let me just observe that I have never encountered such a treatment in the literature.

3 Master reductions for NP

The two master reductions which I will investigate in this section are the Cook-Levin reduction to a version of the Satisfiability problem for Propositional Logic and the reduction based on Tilings.

Propositional logic is a language which is extremely flexible if you want to state properties of finite combinatorial structures, provided you are willing to introduce a sufficiently large collection of propositional variables. In the Cook-Levin reduction these variables encode the complete time-space diagram of an accepting Turing machine computation on the given input. The reduction is performed in such a way that it establishes a one-one correspondence between accepting computations and satisfying assignments of these propositional variables.

Let some language L in NP be accepted by some nondeterministic Turing Machine M in polynomial time. That means that for some input string x it holds that x belongs to L if and only if we can find a time-space diagram of size T by T which describes an accepting computation according to M where T is moreover bounded by $P(|x|)$ for some fixed polynomial P. The time-space diagram is encoded using propositional variables $p[i,j,k]$ expressing that *at position $< i,j >$ in the diagram symbol σ_k is written.*

The Cook-Levin formula is the conjunction of a collection of sub-formula's which express the required properties of the diagram like

1. At every position in the diagram some symbol is written
2. At every position in the diagram at most a single symbol is written
3. The diagram starts with the encoding of the initial configuration on the input x
4. The diagram terminates in some accepting configuration (which can be tweaked to be unique if one desires it to be so)
5. successive rows in the diagram are connected by legal transitions of the machine M

If the intrinsic representation is used we know (by Stockmeyer' lemma) that the last condition can be expressed by enforcing the local compatibility condition on all 3 by 2 sub-windows in the diagram. This can be expressed by writing some clause excluding an illegal combination of symbols within such a window for all illegal combinations and all proper positions of this window in the diagram (a nice way of expressing this condition if one aims at obtaining a Cook-Levin formula in Conjunctive Normal Form).

It is not difficult to design a Cook-Levin formula in case the Mathematical Representation is used. In this case the state and the head position are denoted outside the diagram but we can introduce additional variables $s[i,l]$ expressing *at time i the machine is in state q_l* and $h[i,j]$ expressing *at time i the head is* *located at position j*. The Cook-Levin formula now will include additional clauses expressing that at every time the state and head position are uniquely determined. The revised correctness conditions require that at some distance from the head position nothing changes and that the changes in the direct neighbourhood of the head positions conform to the given program. Details can be found in any textbook containing a full proof of the Cook-Levin result.

The question becomes whether there is an advantage here of using the intrinsic representation. I claim there is; it is recognized by a simple estimation of the size of the Cook-Levin formula's obtained.

For both representations the number of variables required is $O(T^2K)$ where K is some constant equal to the number of symbols which may occur in the time-space diagram. The number of additional state and head variables required for the Mathematical representation are of order $O(TK)$ and $O(T^2)$ respectively, and these numbers are small compared to the number of variables used for the diagram anyhow.

However if we consider the size of the various sub-formula's one observes that the five conditions in case we use the intrinsic representation are of sizes $O(T^2K)$, $O(T^2K^2)$,$O(T)$, $O(T)$ and $O(T^2K^6)$ respectively. However, when using the mathematical representation, the additional formula expressing the fact that the head always resides at a single position turns out to be of size $O(T^3)$ which is a factor T larger than all the other contributions and becomes the dominant term in the size estimate of the resulting formula in propositional logic(note that K is determined by the program only and is independent of the length of the input).

Hence the penalty for using the mathematical representation in the Cook-Levin result is that the size of the formula produced by the reduction becomes cubic rather than quadratic in the running time of the simulated machine. Yet, this unnecessary overhead has not prevented well known authors, including Cook [2] and Garey & Johnson [4] to use the mathematical representation for their proof of the Cook-Levin Theorem.

Tiling reductions

The tiling reduction, used for NP-reductions originally by Levin [10] and Harry Lewis [11, 12] is based on covering a region of the plane using square tiles which are divided in four triangles each being coloured. Tiles are to be selected from a fixed catalogue of tile types, and may not be rotated or reflected. When two tiles are placed adjacently (horizontally or vertically) the colours along a shared edge must be equal. Boundary conditions are enforced by fixing colours at the boundary of the region to be tiled; alternatively one can assign a first move to the devil by placing a single tile somewhere in the plane and demanding that the tiling must be extended to the full region.

Tilings allow a direct encoding of a time-space diagram if the Intrinsic Representation is used. The successive configurations appear encoded in colours along horizontal lines of the tiled region. We need tile types which express that a tape symbol is passed unchanged from one configuration to the next one. Other tile types express directly the instructions of the program. A third class of tile types allows some tape symbol to become scanned in the next configuration if the head enters from an adjacent column. One must however restrict the Turing Machine program in order to ensure that the machine when moving to some state q can't move in both directions, since this would allow the creation and/or annihilation of phantom pairs of heads in the time-space diagram simulated by the tiling.

A more detailed description of the construction and its use can be found in [22, 25]. The nice properties of the tiling reduction are that there is a complete separation between the encoding of the Turing Machine Program (which determines the catalogue of tile types) and the input (which is encoded in the boundary condition). If we allow the boundary condition to be specified in some more succinct form (I.E., if we can express the size of the boundary rather than listing all edge segments) the reduction shows hardness for higher complexity classes like PSPACE and NEXPTIME . Chlebus [3] has shown how alternating Turing Machines can be reduced to a two player game version of the Tiling problem.

As mentioned the tiling reduction works nicely for the intrinsic representation. It is not to difficult to design a tiling simulation for the semi-intrinsic representation (one uses signals transmitting the state information through a horizontal line) but I never have seen a simulation starting from the mathematical representation, which would require some internal mechanism for performing binary to unary conversion of numbers to start with.

Starting with the tiling reduction as a master reduction problems like Satisfiability but also Knapsack like problems are easily reached by further reductions [14]. But also a Hilbert 10 reduction can be obtained [22, 25] [3].

To my opinion the Tiling reduction is more suitable for educational use compared to the original Cook-Levin reduction. In my classes I have always used the example of a simple Turing Machine which increments a binary counter, a program of 6 instructions. The resulting catalogue of tile types contains 15 types. In 1992 my institute ordered the construction of a wooden demonstration model of the resulting puzzle to be used for educational events. I believe that it represents the most inefficient computer in the world which was ever built. After the move of the institute the puzzle was saved with my archives, but it is locked away in a storage room. The puzzle is available today in digital form on the web [26].

Note also that the combined reduction to Satisfiability using the tiling reduction as an intermediate step achieves the same $O(T^2)$ overhead which is obtained by the direct Cook-Levin reduction in case the intrinsic representation is used.

Our conclusion is that for NP-reductions the use of the Intrinsic Representation is not an absolute requirement, but the alternatives have some disadvantages.

[3] this reduction was originally constructed at a workshop in Paderborn in October 1982 in response and rebuttal to a presentation by J.P. Jones who presented with Yuri Matijasevič an improved version of the reduction of Machine termination to the solvability of exponential Diophantine Equations based on register machines, and claimed that such a reduction based on Turing Machines was not possible.

4 Stockmeyer and his work on regular expressions

The standard theory of Regular Expressions deals with expressions generated by a grammar based on three types of generators and three operations. The generators are:

1. 0 denoting the empty language
2. λ denoting the singleton language containing only the empty word
3. σ for each σ in the alphabet Σ under consideration, denoting the singleton language containing the single letter word σ.

The operators are the $+$ denoting union of languages, . for concatenation of languages and $*$ denoting the Kleene star iteration operation; the $*$ operator is monadic, whereas the $+$ and . are binary operators.

Beyond this standard language of regular expressions a number of additional operators are considered by Stockmeyer: ^{2} , denoting Squaring, I.E., concatenation of a language with itself, $\cap$ denoting intersection and $\sim$ denoting complementation. It is known that the family of regular languages is closed under these operators, hence, in principle, regular expressions involving such operators can be rewritten into standard expressions. However there is no direct algebraic method for doing so. The detour by construction of the corresponding automata and deriving the regular expressions corresponding to these automata will produce unmanageable large expressions.

In chapter 4 of his thesis (the largest chapter in this book) Stockmeyer investigates how these additional operators affect the complexity of decision problems on generalized Regular Expressions. Decision problems considered are:

1. $NEC(\phi, \Sigma)$: does the expression ϕ denote the set of all possible words over the alphabet Σ?
2. $EQ(\phi, \psi), INEC(\phi, \psi)$: do the two expressions denote the same (different) languages?

Evidently these problems are inter-reducible, provided operators like complementation and intersection are available, but since also languages without these operators are considered we need them all.

The hardness results in this chapter are obtained by a master reduction. Consider a rectangular time-space diagram of an accepting computation of some nondeterministic single tape Turing Machine. The correctness of such a diagram is expressed by the conjunction of a number of conditions expressing syntactic well-formedness (consisting of the right sort of symbols in the right positions), correct start (with the intended initial configuration on the input word), correct termination (in some final accepting configuration), and correct computation (enforced by application of Stockmeyer's 3 by 2 window compatibility check throughout the diagram).

The diagram is a two dimensional object, but it can be linearised into a string by printing all rows in the diagram behind each other, separated by a suitable extra marker. So one might look for some generalized regular expression describing precisely those strings which encode a correct time-space diagram. Note that we now must enforce the additional condition that the segments in the linearised diagram all should have the same length.

We need our expression to encode the conjunction of all the conditions which must be enforced. This is hard to express if we don't have the operator of intersection in our language. Therefore Stockmeyer migrates to the complementary world where he constructs an expression which intends to denote all strings which *fail* to encode a correct time-state diagram. The expression becomes a *Syllabus Errorum* stating all possible sources of an error. This explains the use of the decision problem NEC in his investigations: if the accepting time-space diagram exists there exists an error-free string, and therefore the described language will have a non-empty complement. Otherwise all strings are erroneous and the expression will be equivalent to the language Σ^*.

The hardest error type to be described is a violation of the 3 by 2 window compatibility relation. In the time-space diagram the symbols are written closely together but in the linearisation they are separated by a substring whose length is equal to the width of the diagram (up to a small additive constant). This explains the importance of yardsticks: sub-expressions of the form Σ^K for large values of K. Since the width of the time-space diagram equals the space consumed by the simulated computation it becomes relevant to invent succinct representations for these yardsticks: the more succinct such a representation becomes the higher the (nondeterministic) spacebound for which a hardness proof is obtained.

If we have no additional operators the size of the expression for a yardstick is linear in K. Thus hardness is obtained for linear bounded automata. Please keep in mind that at the time the thesis was written the Immerman-Szelepsényi result [7, 20] yielding closure under complementation of nondeterministic space bounded complexity classes had not yet been proven, whence Stockmeyer had to navigate carefully around issues involving complementation. Today we understand his result as a proof of hardness for PSPACE.

Adding the operation ^{2} of squaring reduces the size of the expression for a yardstick to $(O(log(K))$, and hardness for NEXPSPACE is obtained (by Savitch' result [15] NEXPSPACE = EXPSPACE). Removing the $*$ operator eliminates the possibility to talk about arbitrary long computations, and therefore hardness results are obtained for nondeterministic time classes (NP respectively NEXPTIME depending on whether squaring is available or not). The hardest part of the theory is section 4.2 where the impact of the complementation operator is shown: each increase by one in the complementation depth of the regular expressions allows for an exponential increase of the succinctness of the yardstick expression. Therefore the hardness results are raised to non-elementary space and/or time bounded complexity classes.

From our perspective the key ingredient in all constructions is the encoding of a compatibility violation in the diagram by an expression listing the violating pair connected by a yardstick expression. This simulation is made possible by the use of the intrinsic representation and it must be hard if not impossible to obtain a similar construction based on the mathematical representation.

5 The impact of the intrinsic representation on machine models in the Second Machine Class

The Second Machine Class [23, 24] consists of those models for machines supporting some form of parallel processing for which the *Parallel Computation Thesis*, expressed by the equalities //PTIME = //NPTIME = PSPACE is true: what the parallel model can do in polynomial time, deterministically or nondeterministically, is what can be achieved in the sequential world in Polynomial Space.

Machine models of this nature were investigated in the 1970-ies. There are various parallel versions of the Random Access Machines, and versions of Turing Machines supporting parallel branching. More surprising was the discovery that some sequential models which may operate on very large data objects also are second machine class members: typical examples are the Vector Machines introduced by Pratt and Stockmeyer [13] and the Random Access machine extended with multiplicative instructions described by Hartmanis and Simon [5]. Also the Alternating Turing Machine [1] belongs to this class, be it that there exists no nondeterministic version of this device.

Proving that some device indeed satisfies the above equalities uses some methods which by now have been well understood. The inclusion //NPTIME $\subseteq$ PSPACE is shown by guessing an accepting computation trace of the parallel device, and validating this trace using some recursive procedure which will evaluate the state of the elementary hardware components of this device at any time during the computation. A key argument is that the parameters of such a recursive procedure can be written down in polynomial space.

Such a proof can be given only when the parallel model is reasonable: it can activate in polynomial time an exponential amount of hardware (but not more) and the nondeterminism must be Uniform (the same choices are made on all parallel paths in the computation).

For the inclusion PSPACE $\subseteq$ //PTIME nowadays various strategies are avialable: one can show that the parallel device can simulate an Alternating machine, or one can construct a Polynomial Time algorithm for the PSPACE complete problem QBF [18]. However in the mid 1970-ies the Alternating machine had not yet been invented and nobody had proposed the idea of exploiting the QBF problem. The early proofs were all based on a master simulation of a PSPACE bounded Turing machine on the parallel machine.

The idea used in this master simulation is the reduction of the existence of an accepting computation to a connectivity problem on a huge (exponentially large) *Computation graph.* This graph has all possible configurations of the Turing Machine on the allowed amount of space as nodes, and the transitions between these configurations as edges. The initial configuration in the graph is just some special node, and so is the final accepting configuration (which may be assumed to be unique). Computations become paths in this computation graph. Hence the existence of an accepting computation is reduced to the existence of a path connecting the start node with the target node.

This connectivity problem can be solved by computing the transitive closure of the relation given by the edges (transitions). A convenient algorithm for computing this transitive closure uses the mathematical representation of the *Adjacency Matrix*: row and column indices represent nodes (configurations) and the presence of an edge from node i to node j is denoted by assigning the value 1 to matrix element at position $< i, j >$. The diagonal entries in the matrix obtain also value 1 (every node is reachable from itself by a path of length 0).

By iteratively squaring this matrix (over the Boolean algebra where $1.1 = 1 + 1 = 1$) one determines which pairs of nodes in the graph are connected: after t iterations all connections by some path of length $\leq 2^t$ are found. Since cycles don't contribute to connections and the number of the nodes is bounded by $2^{O(spacebound)}$ a polynomial number of iterations is sufficient. At the end of the computation the answer is found in the desired matrix element; the rest of the matrix is discarded.

The details of this simulation depend on the precise model considered. Generic tasks are the construction of some object representing a list of all integers in the range $0 \ldots 2^M$ for some large value of M. The entries in this list represent all possible configurations of the machine. Think of the numbers as being written down as binary numbers and consider the resulting bit-string to be the representation in binary of the string of symbols in the configuration. There is no guarantee that these numbers (digit strings) satisfy reasonably syntactic conditions like not containing more than one state symbol. However, getting rid of such junk configurations is not needed; they can't do any harm. In fact removing them may be harmful in the sequel of the proof, because it would create gaps in the sequence of configurations at positions which are hard to predict.

The next task is to construct the Cartesian product of this list with itself, yielding an object storing all configuration pairs.

Given this object we must determine for all these pairs of configurations whether they are equal or connected by a transition or not. Moreover this has to be done in parallel, given the exponential size of this object. This is precisely the point where it is crucial that these numbers are understood to encode configurations in the intrinsic representation. Equality is easy to test but the test for a transition requires that in the binary representation the digit block is identified where the two strings are different. This block represents the three symbols of Stockmeyer' 3 by 2 window. The contents of these blocks in the two configurations must obey the compatibility relation. Moreover, outside these blocks the two configurations must be identical.

Once this test has been performed the Adjacency matrix is obtained. The computation then can proceed by implementing the iterated multiplication of the matrix with itself. This is yet another complex task, but it is less model dependent.

The details of the above computation are different for parallel versions of the Turing Machine (which is symbol manipulation oriented) and the Parallel Random Access devices. For the RAM based models one must invoke some mechanism which will allow an efficient method for converting numbers into bitstrings. Inspection of the constructions proposed in the literature shows that for all RAM based parallel models some form of string manipulation or some mildly multiplicative operation like division by 2 is inserted in the instruction code. Such instructions are not available in the basic RAM model - the result by Hartmanis and Simon indicate that you can't add to much multiplicative power to the RAM without creating a model which is to powerful.

Our conclusion is that in these early simulations the fact that the numbers encode configurations in the intrinsic representation is a key ingredient for the correctness of the proof. It seems hard, if not impossible to find such a construction if the Mathematical representation is used.

6 Conclusion - is there a dragon out there ??

I hope that the examples in the preceding sections have convinced the reader that the use of the Intrinsic Representation of Turing Machine configurations has been the enabler for several fundamental results in Theoretical Computer Science. The question remains whether this observation should affect our behaviour as theoreticians. Stated otherwise: do we need to start a Crusade against the use of the Mathematical Representation? Is there a dragon out there which should be slayed?

While preparing this presentation I have searched the leftovers of what in the past used to be a well equipped Mathematical Library in my institute[4]. Inspection of some 25 textbooks on introduction in Computer Science or Computation Theory yielded the following results: Many authors give no formal definition of a configuration but informally they present something resembling the intrinsic or a semi-intrinsic representation. This also holds for the Wikipaedia page on Turing machines. I found a formal definition of the Mathematical representation for single tape machines only in the 1969 edition of Hopcroft and Ullman, and in the 1981 edition of Lewis and Papadimitriou. The later authors however immediately continue with a semi-intrinsic representation as an illustrative tool - no wonder, since in this textbook a master reduction based on tilings is presented. Other authors give the Mathematical representation for multi tape machines but move towards the intrinsic representation for the single tape model, and that is the model used in all the hardness and undecidability proofs. Turing himself uses an Intrinsic representation by way of illustration. So do Kleene and Davis.

Evidently in practice our colleagues have throughout the last 70 years followed their intuition and have made the right choice. But except for Stockmeyer I have not found anybody who explicitly has looked into the advantages of this decision.

The conclusion is that we can continue and live and work in peace. Dragons remain a rare species which should be protected rather than persecuted.

[4] victim of the curse of digitalisation

References

1. Chandra, A.K., Kozen, D.C. and Stockmeyer, L.J., *Alternation*, J. Assoc. Comput. Mach. 28 (1981) 114–133
2. Cook, S.A., *The complexity of theorem proving procedures*, Proc. ACM Symposium Theory of computing 3 (1971), 151–158
3. Chlebus, B.G., *Domino-tiling games*, J. Comput. Syst. Sci. 32 (1986) 374–392
4. M. Garey, D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co 1979
5. Hartmanis, J. and Simon, J., *On the power of multiplication in random access machines*, Proc. IEEE Switching and automata theory 15 (1974), 13–23
6. Hennie, F.C. and Stearns, R.E., *Two-way simulation of multi-tape Turing machines*, J. Assoc. Comput. Mach. 13 (1966) 533–546
7. Immerman, N., *Nondeterministic space is closed under complementation*, SIAM J. Comput 17 (1988) 935–938
8. Jones, J.P. & Matijasevič, Y.V., *Register machine proof of the theorem on exponential Diophantine representation of enumerable sets*, J. Symb. Logic 49 (1984) 818–829
9. Kleene, S.C., *Introduction to Metamathematics*, Noth Holland Publ. Cie 1952
10. Levin, L.A., *Universal'nie zadachi perebora*, Problemi Peredachi Informatsie IX (1973), pp. 115–116 (in Russian)
11. Lewis, H.R., *Complexity of solvable cases of the decision problem for the predicate calculus*, Proc. IEEE FOCS 19 (1978), pp. 35-47
12. Lewis, H.R. and Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall 1981
13. Pratt, V.R. and Stockmeyer, L.J., *A characterization of the power of vector machines*, J. Comput. Syst. Sci. 12 (1976) 198–221
14. Savelsberg, M.P.W. & van Emde Boas, P. *BOUNDED TILING, an alternative to SATISFIABILITY?*, in G. Wechsung (ed.) proc. 2nd Frege Memorial Conference, Schwerin, Sep 1984, Akademie Verlag, Mathematische Forschung vol. 20, 1984, pp. 401–407
15. Savitch, W. J., *Relations between Deterministic and Nondeterministic tape Complexities*, J Comput. Syst. Sci. 12 (1970) 177–192
16. Slisenko, A.O., *A simplified proof of the real-time recognizability of palindromes on Turing Machines*, J. Mathematical Sciences 5(1), 1981 , 68–77
17. Spaan, E., Torenvliet, L. & van Emde Boas, P., *Nondeterminism, fairness and a fundamental analogy*, EATCS Bulletin 37 (Feb. 1989) pp. 186–193
18. Stockmeyer, L.J. & Meyer, A.R., *Word problems requiring exponential time*, Proc. ACM STOC 5 (1973) pp.1–9
19. Stockmeyer, L., *The complexity of decision problems in automata theory and logic*, Report MAC-TR-133, MIT 1974
20. Szelepsényi, R., *The method of forcing for nondeterministic automata*, Bull. EATCS 33 (1987), 96–100
21. Turing, A.M., *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math. Soc. ser. 2, 42 (1936) 230–265
22. van Emde Boas, P., *Dominoes are forever* Proc. 1st GTI Workshop, Paderborn, Oct. 1982 L. Priese ed. Rheie Theoretische Informatik UGH Paderborn 13
23. van Emde Boas, P., *The second machine class 2, an encyclopedic view on the parallel computation thesis*, in H. Rasiowa, (ed.), Mathematical problems in computation theory, Banach Center Publications, 21 (1987) pp. 235–256

24. van Emde Boas, P., *Machine models and simulations*, in J. van Leeuwen (ed.), Handbook of theoretical computer science, North Holland Publ. Comp. 1990, vol A, pp. 3–66
25. van Emde Boas, P., *The convenience of tiling* in Andrea Sorbi, ed., Complexity, logic and recursion theory, Lect. Notes in Pure and Appled Math. 187, 1997, pp. 331–363
26. van Emde Boas, H., *Turing Tiles*, Web application located at `http://www.squaringthecircles.com/turingtiles/`
27. Vitányi, P.M.B., *An optimal simulation of counter machines*, SIAM J. Comput. 14 (1985) 1–33