**LAL Is Square:**

# Representation and Expressiveness in Light Affine Logic

Peter Møller Neergaard*[1] and Harry G. Mairson**[1]

School of Computer Science, Brandeis University, Waltham, Massachusetts 02454,
{turtle,mairson}@cs.brandeis.edu

**Abstract.** We focus on how the choice of input-output representation has a crucial impact on the expressiveness of so-called "logics of polynomial time." Our analysis illustrates this dependence in the context of *Light Affine Logic* (LAL), which is both a restricted version of Linear Logic, and a primitive functional programming language with restricted sharing of arguments. By slightly relaxing representation conventions, we derive doubly-exponential expressiveness bounds for this "logic of polynomial time." We emphasize that *squaring* is the unifying idea that relates upper bounds on cut elimination for LAL with lower bounds on representation. Representation issues arise in the simulation of DTIME$[2^{2^n}]$, where we construct a *uniform family* of proof-nets encoding a Turing Machine; specifically, the dependence on $n$ only affects the number of enclosing *boxes*. A related technical improvement is the simulation of DTIME$[n^k]$ in depth $O(\log k)$ LAL proof-nets. The resulting upper bounds on cut elimination then satisfy the properties of a *first-class* polynomial Turing Machine simulation, where there is a fixed polynomial slowdown in the simulation of any polynomial computation.

## 1 Introduction: Representation and Squaring Matters

Computer scientists are now defining the next 700 languages of polynomial time[1] [3, 13, 12, 8, 1, 18, 16]. These languages recast computation and complexity in settings familiar to programmers, rather than in terms of Turing Machines

* Supported by the Danish Research Agency grants 1999-114-0027 and 642-00-0062 and the NSF grant CCR-9806718.

** Supported by NSF Grants CCR-9619638, CDA-9806718, and the Tyson Foundation.

[1] A language is considered polynomial if all polynomial time algorithms can be expressed in the language (completeness) and any program can be evaluated in polynomial time (soundness). In neither case it is not necessarily in the standard way.

see [14] for an elaboration). Such studies have the potential of aiding compilers in providing target code with resource guarantees.

In this context, our paper has a two-fold purpose: on the philosophical level it will focus on the inherent difficulties of defining the "logic of polynomial time" or "a language of polynomial time". In particular we stress the importance of the conventions for representing input and output. Proving completeness of a polynomial time language typically employs an encoding of a polynomial time Turing Machine (TM) into the respective language. This naturally raises the question of what is considered a valid encoding. We demonstrate that slight variations in the conventions make the languages complete for super-polynomial time.

Specifically, we analyze Light Affine Logic (LAL) and derive doubly-exponential bounds on expressiveness. LAL [1, 2] (and its predecessor Light Linear Logic [8]) is primitive version of Linear Logic [5, 6] where duplication is restricted. Consequently, the paper contains some terse technical results which might only be of interest to LAL aficionados. In particular we make clear the essential and fundamentally naïve role of *squaring* for understanding the complexity of LAL.

In more technical detail, we start with the simple observation that normalizing a proof-net "level" can only *square* proof-net size. This is used to reprove the upper bounds on proof size via cut elimination. Next, we use iterated *squaring* in a technical lemma to construct a term of size and depth $O(n)$ which essentially has the LAL Church numeral for $2^{2^n}$ as the normal form. This lemma is extended to establish completeness for doubly exponential time using a family indexed by the tape size to encode a Turing Machine. In short, *squaring* is needed both for the upper and lower bound.

It should be noted that the LAL terms of this lemma differ slightly from the standard LAL representation of Church numerals by having type $\S^{4n}!\mathsf{Int}$ where $\mathsf{Int} = \Lambda\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. Each numeral can thus have a *different type*, but only in the number of surrounding boxes (the modality $\S^{4n}$). This kind of uniformity is reminiscent of the idea of *uniform circuits* in circuit complexity.

In full detail we use the aforementioned lemma to derive the following consequences:

- We resolve an open question recently posed by Terui [21] by providing a *first-class* [22] simulation of Turing Machines in LAL[2]. The standard way to represent a DTIME$[n^k]$ results in proof-nets of depth $k+7$. Computation via normalization is then bounded as $O(|X|^{2^{k+7}})$, which is not tight. Out technical lemma reduces the depth to $O(\log k)$, so that proof-net normalization is bounded more appropriately as $O(|X|^k)$.
- We *relax* the standard input/output conventions by allowing the representation of a TM to have a "uniform type" as outlined above. With this relaxation we can encode any Turing Machine with time bound $2^{2^n}$ on an input $x$ of length $n$. The transducer doing the encoding uses the same linear time and

[2] A machine simulation is first-class when running time of the simulation is a fixed polynomial of the original Turing Machine

constant space resources as the transducer used with the standard conventions, for example in [2].
- LAL has a "Statman theorem" [20]: given two LAL proof-nets, do they have the same normal form? This problem is complete for DTIME[$2^{2^n}$]. Even if LAL is the logic of polynomial time, reasoning about proofs in this logic certainly is not polynomial.

In following section we will give a brief tutorial on LAL proof-nets. In Sect. 3 we redo the cost analysis of LAL proof normalization with a simple geometrical argument. Sect. 4 provides the technical squaring lemma discussed above. Sect. 5 shows LAL's completeness for doubly exponential time.

## 2 What You Need to Know About LAL Proof-Nets

In this section we will briefly recall the central aspects of Linear Logic [5], proof-nets [7, 17], and LAL [2]. The section is not intended to be self-contained and readers are encouraged to consult the references for more details.

Like many other logical systems, Linear Logic can be seen both as a logic in itself and, through the Curry-Howard isomorphism, as a formalism for computation in the form of a typed $\lambda$-calculus. We will focus on the latter perspective where the notable feature is that duplication of values is made explicit. This is achieved through modal types where only values of type modality ! can be copied. Light Affine Logic takes this a step further by limiting duplication to restrict the expressiveness.

### 2.1 Building a Proof-Net

Linear Logic and its siblings can be described in the traditional way using inference rules. Perhaps more appealing to computer scientist is to consider it typing rules for a typed $\lambda$-calculus (this has the additional advantage of brevity when writing up conference papers [1]). Alternatively, one can use proof-nets which exhibit the two-dimensional structure of computation; we have provided proof-nets in an appendix.

Before introducing LAL proof-nets, we need the grammar of LAL-formulae:

$$F ::= V \mid V^\perp \mid F \otimes F \mid F \parr F \mid !F \mid ?F \mid \S F \mid \forall V.F \mid \exists V.F \ .$$

Here $V$ ranges over propositional variables, $\forall$ and $\exists$ are second-order quantification, $\otimes$ and $\parr$ are the conjunction and disjunction of the multiplicative fragment, and !, ?, and § are the modalities. We define negation $(-)^\perp$ as an involutive on literals ($(a^\perp)^\perp = a$) and on formulae by the de Morgan identities: $(A \otimes B)^\perp = A^\perp \parr B^\perp$, $(A \parr B)^\perp = A^\perp \otimes B^\perp$, $(!A)^\perp = ?(A^\perp)$, $(?A)^\perp = !(A^\perp)$, $(\S A)^\perp = \S(A^\perp)$, $(\forall X.B)^\perp = \exists X.B^\perp$, and $(\exists X.B)^\perp = \forall X.B^\perp$. We define *linear implication* $A \multimap B$ as $A^\perp \parr B$.

Each LAL proof-net is a graph where *proof-net ports*, *nodes*, and *boxes* are connected by *wires* (the edges). Nodes and boxes are considered to have one

*principal port* and a number of *auxiliary ports* depending on the kind of node or box. The wires are typed with LAL formulae in each direction with the constraint that if the wire has type $\varphi$ in one direction, it has type $\varphi^\perp$ in the other direction. The proof-net ports are only connected to one wire each and serve to define the input/output of a proof net. When looking at a wire connected to a proof-net port in the direction *into* the proof-net, we talk about *input*; when looking in the direction out of the proof-net, we talk about *output*. Note the *dual view* that a proof-net port can be input or output depending on the eye of the beholder.

LAL proof-nets are defined inductively: The basis is a single *wire* corresponding to the axiom $\varphi \vdash \varphi$. It has two proof-net ports with input (and output!) types $\varphi$ and $\varphi^\perp$. The *cut*-rule implements the *Cut*-rule of Sequent Calculus [4] and joints a proof-net with output $\varphi$ with a proof-net where one of the inputs is $\varphi$; this gives a proof-net with the remaining inputs as input and the second proof-net's output as output. The *multiplicative* nodes allow two edges from distinct proof-nets (with output wires typed $\varphi$ and $\psi$) to be jointed by a $\otimes$ node, constructing a graph where the new output edge has type $\varphi \otimes \psi$. Dually, two edges from a single proof-net (with output types $\varphi$ and $\psi$) can be joined by a $\parr$ node, constructing a graph where the new output edge has type $\varphi \parr \psi$.

The crucial aspect of LAL is how inputs can be shared and duplicated. As in linear logic, only input of type $!\varphi$ can be shared. This is done using a *sharing node* marking that the input comes from the same shared source of type $!\varphi$. The limitation comes in the creation of a sharable proof-net: only proof-nets with at most one input can be *boxed* to create a *sharable box* (a global structure). If the proof-net has input $\psi$ and output $\varphi$, the sharable box will have input $!\psi$ and output $!\varphi$. The "at most one input" restriction gives LAL proof nets the structure of a binary tree where the sharing nodes serves as the internal nodes.

The other important restriction of LAL is the lack of *dereliction*, allowing $!A \multimap A$. This has the consequence that all nodes stay at a fixed level (measured by the number of enclosing boxes) throughout a computation. This effectively gives a *stratification* of the computation. To allow computations to be aligned up at the same level, LAL uses the neutral modality $\S$. This allows us to make *neutral boxes* that cannot reproduce (like worker bees): a proof-net with output $\theta$ and inputs $\varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m$ can be boxed to produce a proof-net with output $\S\theta$ and inputs $\S\varphi_1, \ldots, \S\varphi_n, !\psi_1, \ldots, !\psi_m$.

These restrictions mean that a LAL proof-net can be described as a disjoint set $\mathcal{T}$ of binary trees, where every sharing node at level $j$ forms an *internal node* of some $T \in \mathcal{T}$. The formation rules for the logic assure that the trees indeed are acyclic: no leaf of the tree can be connected to the root. In addition:

- Each *internal edge* is a proof-net edge or a number of !-boxes connected in chain. The internal edge is attached to the principal port of a sharing node and auxiliary port of another sharing node.
- Each *leaf edge* of a tree is connected to either a proof-net port, the auxiliary port of a $\otimes$-, $\parr$-, $\exists$-, or $\forall$-node at level $j$, or the auxiliary port of an $\S$-box.
- Each *root* of a tree is connected to either the principal port of !-box, a proof-net port, or the auxiliary port of a $\otimes$-, $\parr$, $\exists$-, or $\forall$-node at level $j$.

### 2.2 Reducing a Proof-Net

LAL has three groups of cut elimination rules carried over from Linear Logic: the linear elimination rules, the shifting elimination rules, and the polynomial elimination rule. The rules are Church-Rosser and are shown in detail in App. A.

The *linear elimination rules* contain the annihilation rules for the quantifiers and the multiplicative fragment as well weakening (hence the logic is affine, not linear). We include $\lambda$-@-reduction for convenience even though it is just a special case of $\otimes$-$\parr$-reduction with a different orientation of the wires. We assume that the linear elimination rules have unit cost.

The *shifting elimination rules* consist of box elimination and box fusion. Due to the box fusion rule it can often be convenient to assume that every node have its level as an attribute and simply ignore the boxes. Finally, there is the *polynomial elimination rule*, box copying, where a sharing node copies the !-box at its principal port. Only this latter rule can increase the size of a proof-net. We assume that the shifting and polynomial elimination rules have cost linearly proportional to the size of the box involved.

## 3 Normalization of LAL Proof-Nets: Trivial Analysis

In this section we will revisit the proof of the complexity bound for LAL cut elimination. We will provide a simple geometrical proof of the double exponential bound in [8, 2]. We consider procedures which normalize proof-nets by level: first all nodes at level 0 are normalized, then those at level 1, etc. When we so normalize "by level", we follow a strategy where at level $j$ we first contract all linear redexes, then push sharing nodes to copy all boxes at level $j+1$, and finally do all shifting elimination reductions.

**Lemma 1.** *Normalization "by level" produces a normal form.*

**Lemma 2.** *Consider the normalization strategy "by level". Suppose levels $i < j$ have been normalized and normalization proceeds on a proof-net of size $n$. Then*

- *The increase in proof-net size is at most one of* squaring*: normalization at level $j$ increases the size to at most $O(n^2)$.*
- *The number of graph reduction operations is at most $n$.*
- *The cost of the graph reduction operations is at most $O(n^2)$.*

*As a consequence, the normal form of a proof-net of size $n$ and depth $d$ has size $O(n^{2^{d-1}})$. This bound also limits the number of graph operations and the cost.*

*Proof.* Recall from the introduction that a proof-net can be considered a collection of binary trees. Consider a tree $T_i$ with size $t_i$ whereof $t'_i$ comes from levels lower than $j$. We have at most $t_i - t'_i$ sharing nodes. The size of the reduced tree is thus at most $t'_i(t_i - t'_i) = O(n^2)$. The number of graph reduction steps and their cost are immediate from this analysis.

We notice that the bound can be even tighter if one replaces the depth $d$ by the number of different levels where there are sharing nodes.

Note that the exponent $d-1$ (rather than $d$) occurs because no sharing nodes exist at level $d$. It is straightforward to extend this trivial analysis to elementary affine logic (EAL), where there is no § modality, and !-boxes can have more than one auxiliary port. The binary trees in the above discussion then become directed acyclic graphs, and so duplication at each level can increase proof-net size by an exponential. The worst case is a chain of sharing nodes, where a box containing a trivial structure (say, a $\otimes$-node) connects the sharing nodes. A detailed picture can be found in App. A.

The above argument is easily extended to a bound, from above, of the total number of graph reduction operations in *any* normalization strategy. This provides a simplification of the analysis original done by Terui [21].

The above bounding argument fails when accounting for cost, because the *size* of a duplicated box matters. Consider duplication of a box $B$ reducible to $B'$ at cost $k$. Reducing before duplicating has cost $k + |B'|$, while duplicating before reducing has cost $2k+|B|$. Duplicating before reducing is not always more costly if $k \geq |B'| - |B|$. Counting reduction operations gives a uniform bound, since duplicating a box is a single operation regardless of box size.

## 4 Iterative Squaring: How High Can You Count?

In the last section we saw that the upper bound on LAL cut normalization is doubly exponential. In this section, we will turn the question around and see how we can realize that bound. Again *squaring* turns out to be crucial. By coding the squaring function on the LAL version of Church numerals, we come quite close, since we can type it as $!\mathsf{Int} \multimap \S^4!\mathsf{Int}$.[3] Since we only need a small constant depth to square a Church numeral, the construction can be iterated to construct a proof-net of size $n$ and depth $4n$ which normalizes to the numeral for $2^{2^n}$ of type $\S^{4n-1}!\mathsf{Int}$.

We recall the type $\mathsf{Int} = \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ for coding integers in LAL. We then have:

$$
\begin{aligned}
\overline{0} &= \Lambda\alpha.\lambda s : !(\alpha \multimap \alpha).\S(\lambda z : \alpha.z) : \mathsf{Int} \\
\overline{k} &= \Lambda\alpha.\lambda s : !(\alpha \multimap \alpha).\S(\lambda z : \alpha.w_1(w_2(\cdots(w_k z)\cdots))[s/w_1, \cdots, s/w_k] : \mathsf{Int} \\
\mathbf{succ} &= \lambda n : \mathsf{Int}.\Lambda\alpha.\lambda s : !(\alpha \multimap \alpha).\S(\lambda z : \alpha.u(vz))[s/u][n\alpha s/v] : \mathsf{Int} \multimap \mathsf{Int} \\
+ &= \lambda m : \mathsf{Int}.\lambda n : \mathsf{Int}.\Lambda\alpha.\lambda s : !(\alpha \multimap \alpha).\S(\lambda z : \alpha.u(vz))[m\alpha s/u, n\alpha s/v] \\
&\quad : \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int} \\
\times' &= \lambda m : !\mathsf{Int}.\lambda n : \mathsf{Int}.\Lambda\alpha.\S(w\,\overline{0})[((m\,\mathsf{Int})\,!(\lambda y : \mathsf{Int}.+ny))/w] : !\mathsf{Int} \multimap \mathsf{Int} \multimap \S\mathsf{Int} \\
\mathsf{coerce}_{p,k} &= \lambda m : \mathsf{Int}.\S^p(w\,!^k(\overline{0}))[m\,(!^k\mathsf{Int})\,!(\lambda n : !^k\mathsf{Int}.!^k(\mathbf{succ}\,n))/w] : \mathsf{Int} \multimap \S^p!^k\mathsf{Int} \\
\times &= \lambda m : \mathsf{Int}.\lambda n : \mathsf{Int}.\S(\times'(\mathsf{coerce}_{1,1}m)(\mathsf{coerce}_{1,0}n)) : \mathsf{Int} \multimap \mathsf{Int} \multimap \S^2\mathsf{Int}
\end{aligned}
$$

[3] In fact, we can reduce the range type to $\S^3\mathsf{Int}$, but further coercion is needed to iterate squaring.

where $p > 0$ and $k \geq 0$ in the definition of $\mathsf{coerce}_{p,k}$. The first five are familiar terms from $\lambda$-calculus with fancy new, light affine types; see [2]. The notation $!(\cdots)[\cdots]$ and $\S(\cdots)[\cdots]$ represent !- and §-boxes where each proof-net (term) or wire (variable) connected to the auxiliary ports is named explicitly in the substitution. The coercion functions $\mathsf{coerce}_{p,k} : \mathsf{Int} \multimap \S^p!^k\mathsf{Int}$, where $p > 0$ and $k \geq 0$, are a specialty of LAL which change the boxing around a Church numeral. This is done by plugging in a successor and zero at the relevant level for successor and zero of the Church numeral.[4] Coercion is used in $\times$ to "repair" the problem that the inputs have different levels in the immediate LAL-translation $\times'$ of multiplication. It is a cheap, static form of digging.

Using these building blocks, we can make the function $\mathsf{square}_1$ which squares an integer (the proof-net is provided in App. A). The cornerstone is $\times$ which is used for multiplying the integer by itself (hence the sharing). The output of $\times$ has only neutral modalities. We use coercion to make the output have an innermost ! which allows $\mathsf{square}_1$ to be used repeatedly. This construction leads us to the following generalized squaring function

$$\mathsf{square}_k = \lambda z_k : !\mathsf{Int}.\ \S^4\mathsf{square}_{k-1}(z_{k-1})[\mathsf{square}_1(z_k)/z_{k-1}] : !\mathsf{Int} \multimap \S^{4k}!\mathsf{Int}$$

**Lemma 3.** *Recall $\overline{k}$ to be the LAL Church numeral for integer $k$. Then the proof-net for $\mathsf{square}_n(!\overline{2})$ has type $\S^{4n}!\mathsf{Int}$, size and depth $O(n)$, and normalizes to $\S^{4n}!\overline{2^{2^n}}$.*

Notice that in this lemma, we get a big Church numeral whose proof-net description is just like that in proof-net codings of $\lambda$-calculus. the only difference is that the proof-net is enclosed in a linear number of boxes, which are conveniently erased in the $\lambda$-calculus analogue. By iterated squaring, we also derive the following:

**Corollary 1.** *For any polynomial $p(n) = n^k$ there is an LAL term $p_k : !\mathsf{Int} \multimap \S^{4\log k}!\mathsf{Int}$ where $p_k(!\overline{n})$ normalizes to $\S^{4\log k}!\overline{n^k}$.*

## 5 Representing Time-Bounded Computation

We now have the tools to show the main results: how to represent any double exponential time computation in LAL. This is done by showing how to represent any doubly exponential time TM as a family of LAL proof-nets. We will first show how to make the transition function; second we show how to translate an arbitrary input tape into a proof-net, and in the last subsection derive various Statman like corollaries from the main result. For the purpose of the exposition we will consider a TM with only 0 and 1 as the string alphabet. It is assumed to have an infinite tape in both directions and is initially filled with 0's except for the initial input. The input appears under and to the right of the head when the machine is started.

[4] Similar constructions can be made for other inductively defined data structures, but not for higher order types.

### 5.1 Transition Function

LAL provides weak-sister typings of many of the standard System F codings of inductive data types—"weak-sister" since their capacity to act as iterators is restricted. We have already seen the System F type $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$ reappear in LAL as $\mathsf{Int} = \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. The System F type of lists of elements of type $T$, $\forall\alpha.(T \to \alpha \to \alpha) \to \alpha \to \alpha$ is $\forall\alpha.!(T \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. For a compendium of such codings, see [10].

Following the intuitions behind the construction of lists, LAL is powerful enough to simulate the transition function of a TM. We represent the TM tape as two lists: one with the tape to the left of the head (with the symbol immediately to the left of the head as head of the list) and one with the tape under and to the right of the TM head. To get the full TM configuration we also need the control state. We therefore get the following type for a configuration:

$$ID \;=\; \forall\beta.!(\mathsf{Bool} \multimap \beta \multimap \beta) \multimap \S((\beta \multimap \beta) \otimes \mathit{State} \otimes (\beta \multimap \beta)) \;\; .$$

Here $\mathsf{Bool} = \forall\gamma.\gamma \otimes \gamma \multimap \gamma$ and $\mathit{State} = \forall\gamma.\gamma \otimes \gamma \otimes \cdots \otimes \gamma \multimap \gamma$ where the latter codes a type of constants whose cardinality is the number of control states in the TM. Note that the two lists share the same constructor (`cons`) of type $(\mathsf{Bool} \multimap \beta \multimap \beta)$. An initial configuration with $x_1 \cdots x_n$ as input and initial state $s_0$ is given by the term

$$\Lambda\beta.\lambda c : !(\mathsf{Bool} \multimap \beta \multimap \beta).\ \S((\lambda \mathsf{nil}' : \beta.\mathsf{nil}') \otimes S_0 \otimes (\lambda \mathsf{nil}'' : \beta.cx_1 \cdots (cx_n \mathsf{nil}'') \cdots)) \;\; . \tag{1}$$

We code the transition function by a term of type $ID \multimap ID$. The function just needs to take this structure apart, extract the cell under the read head, compute a finite function, and put the pieces back together appropriately. To split a tape into its head and tail we define a term $C$ that will be used for $c$:

$$C \equiv !(\lambda e : \mathsf{Bool}.\lambda g \otimes h : \mathsf{Bool} \otimes \alpha.\ e \otimes (c'gh)) :!(\mathsf{Bool} \multimap (\mathsf{Bool} \otimes \alpha) \multimap (\mathsf{Bool} \otimes \alpha)) \;\; . \tag{2}$$

Notice that $C$ has a free variable $c'$ of type $!(\mathsf{Bool} \multimap \alpha \multimap \alpha)$ which is used as constructor for the rest of the list. We note that for $x_i : \mathsf{Bool}$,

$$Cx_1(Cx_2(Cx_3 \cdots (Cx_n(0 \otimes \mathsf{nil})) \cdots)) \to^* x_1 \otimes (c'x_2(c'x_3(\cdots(c'x_n(c'0\,\mathsf{nil})) \cdots))) \;\; .$$

Given a term $L : ID$ representing a configuration, we now have

$$L[\mathsf{Bool} \otimes \alpha]C :\ \S(((\mathsf{Bool} \otimes \alpha) \multimap (\mathsf{Bool} \otimes \alpha)) \otimes \mathit{State} \otimes ((\mathsf{Bool} \otimes \alpha) \multimap (\mathsf{Bool} \otimes \alpha))) \;\; , \tag{3}$$

where we for each of the two tape halves can get the head and tail by providing a base case (we will use the pair of *false* and the empty list). Using the above constructions, we can define a transition function $\delta$. At the heart of $\delta$ is a function $\Delta : \mathit{State} \multimap \mathsf{Bool} \multimap \alpha \multimap \mathsf{Bool} \multimap \alpha \multimap \mathit{State}$ finding the new configuration from the current control state, the symbols to the left and under the head, and the tails of the left and right tapes. It is encoded as a simple table lookup indexed by the current state and the symbol under the tape. The details are given in the appendix.

### 5.2 Transforming the Input Tape

To circumvent issues with representation, we will consider a Turing machine transducer that translates the input directly into a proof net.[5]

**Definition 1.** *A* proof-net transducer *is a Turing machine with a read-only input tape and a number of work space proof-net nodes. The transducer can* link *nodes in the work space or* create *new nodes; when creating a new node an existing node is "pushed" out of the work space but kept as part of the output if it is linked to another work space node.*

We will show that any input tape can be transduced using only a constant number of work space nodes. To ensure this space bound, we use an output convention where boxes are not drawn as global structures. Rather wires are annotated with nodes indicating the need to change level, like an offset. Furthermore, the proof-nets are output without type annotation, since they can be reduced correctly without these annotations. This convention is familiar to programming language implementation, where types are used for static analysis, but thrown away at runtime.

Given an input tape of length $n$ the transducer builds and combines the following parts:

1. a representation $T$ of the input tape at level $4n+2$,
2. a proof-net $N$ normalizing to the Church numeral for $2^{2^n}$ of type $\S^{4n}!\mathsf{Int}$, and
3. an iterator $P$ at level $4n+1$ applied to the Turing machine.

The size of each part is linear in $n$ and we shall see that the transducer can build it stepwise for each character read.

The "tape" is the proof-net corresponding to the initial term of term of (1) embedded to level $n' = 4n+2$:

$$\S^{n'}\Lambda\beta.\lambda c : !(\mathsf{Bool} \multimap \beta \multimap \beta).\ \S((\lambda \mathsf{nil} : \beta.\mathsf{nil}) \otimes S_0 \otimes (\lambda \mathsf{nil}' : \beta.cx_1 \cdots (cx_n \mathsf{nil}') \cdots)) \ .$$

This is produced as follows: the transducer initially builds

$$\S^2\Lambda\beta.\lambda c : !(\mathsf{Bool} \multimap \beta \multimap \beta).\ \S((\lambda \mathsf{nil}' : \beta.\mathsf{nil}') \otimes S_0 \otimes (\lambda \mathsf{nil}'' : \beta.[]))$$

and then $[]$ is filled gradually while scanning the input tape. Since the initial term is a constant term, the initialization step takes constant time. The transducer has work space nodes holding the outermost $\S$ and the $\lambda$-node. For each input symbol read, 4 $\S$-nodes are added in front of the previously outermost $\S$ which is being pushed out. An new application is linked to the previous application node (or the $\lambda$-node for first symbol) building the proof-net in $[]$. This output only requires a constant number of work space nodes.

[5] The result of this subsection carries over even if one choses to output the proof-net in the syntax used in [1, 2]; in that case one just need to scan the input tape several times rather than one.

The "number" $N$ is simply the term $\mathsf{square}_n(!\overline{2})$ of type $\S^{4n}!\mathsf{Int}$ presented in Lem 3. Recall the construction from Sect. 4

$$\mathsf{square}_k \equiv \lambda z_k :!\mathsf{Int}.\ \S^4\mathsf{square}_{k-1}(z_{k-1})[\mathsf{square}_1(z_k)/z_{k-1}]\ .$$

The term $N$ can be build iteratively by adding a new squaring device inside the previous construction. One needs only two work space nodes to keep track of the output from $\mathsf{square}_k$ and the wire holding the result.

The proof-net $I$ is an application node embedded to level $4n+2$:

$$\lambda n : \S^{4n}!\mathsf{Int}\lambda t : \S^{4n+2}.\S(m\ t)[n\ ID\ !\delta/m]\ .$$

Note that the Turing machine transition function $\delta$ is implicitly embedded at level $4n+1$. This net is built by adding $\S^4$ for each input symbol. Detailed proof-nets can be found in App. A.

### 5.3 Main Results

We sum up the discussion of the previous subsections in the following theorem:

**Theorem 1.** *(Main theorem) For any TM $M$ which in $O(2^{2^n})$ steps accepts or rejects an input of length $n$, there exists a* family *of LAL proof-nets $P_n : \S^{4n}!\mathsf{Int} \multimap \S^{4n+2}ID \multimap \S^{4n+3}\mathsf{Bool}$ such that, for any $n$ input $x \in \{0,1\}^n$ to $M$, there exists terms $N_x : \S^{4n}!\mathsf{Int}$ and $T_x : \S^{4n+2}ID$, such that $M$ accepts $x$ iff $P_nN_xT_x$ reduces to $\S^{4n+3}true$, where true is the usual $\forall\beta.\lambda x : \beta.\lambda y : \beta.x$. The proof-nets $P_n$ are all identical except in the number of surrounding boxes. The terms $P_n$, $N_x$, and $T_x$ can be output by a TM transducer running on input $x$ in $O(1)$ space and $O(n)$ time.*

We can also state a theorem similar to [8, 2] where the TM is transformed once providing a function of type $\mathsf{List\ Bool} \multimap \S^\ell\mathsf{Bool}$ for some $\ell \geq 0$.

**Theorem 2.** *For any TM $M$ accepting or rejecting an input of length $n$ in $O(n^k)$ steps, there exists an LAL proof-net $P : (\mathsf{List\ Bool}) \multimap \S^{4(1+\log k)}\mathsf{Bool}$ such that for any input $x \in \{0,1\}^n$ to $M$, there exists a term $L : (\mathsf{List\ Bool})$, where $M$ accepts $x$ iff $PL$ reduces to $\S^{4(1+\log k)}true$. The term $L$ can be output by a TM transducer running on input $x$ in $O(1)$ space and $O(n)$ time.*

*Proof.* We functionally compose $p_x$ of Cor. 1 with $P_{\log k}$ to a net of type $!\mathsf{Int} \multimap \S^{4\log k+2}ID \multimap \S^{4\log k+3}\mathsf{Bool}$. We combine with the constructions found in [2, §12.5] to double an input tape and turn one of the copies into the length while embedding the other to level $4\log k+2$. This gives a function of type $(\mathsf{List\ Bool}) \multimap \S^{4\log k+3}\mathsf{Bool}$, satisfying the theorem.

Because this representation of DTIME$[n^k]$ is in depth $O(\log k)$, where only $\log k$ levels have occurrences of sharing nodes, we derive a corollary that is not implied by previous machine simulations via proof-nets. The corollary solves the problem raised by Terui [21] that the simulations in the literature are not first-class since DTIME$[n^k]$ is taken into DTIME$[n^{2^k}]$.

**Corollary 2.** *Consider the simulation, via a proof-net $\Pi$, of a TM $M$ accepting or rejecting an input of length $n$ in $O(n^k)$ steps. Then the bounds on normalization of $\Pi$ ensure a computation with cost bounded by a polynomial in $n^k$.*

These simulation results also allow a Statman-style theorem:

**Corollary 3.** *Given two LAL proof-nets of size and depth $O(n)$, deciding whether they have the same normal form is complete for* $\mathrm{DTIME}[2^{2^n}]$.

In turn, there is a watered-down version of Rice's theorem, which shows the weakness of polynomial-time reasoning about LAL programs:

**Corollary 4.** *Let $\mathcal{P}$ be a property of LAL proof-nets (or terms) that is preserved by reduction, where given a proof-net $\Pi$ with size and depth $O(n)$, $\mathcal{P}(\Pi)$ can be decided in $O(n^k)$ steps. Then $\mathcal{P}$ is* trivial *for proof-nets of depth $\Omega(\log k)$—there exists a constant $c$ independent of $k$, where for each type $\S^{c \log k}\sigma$, $\mathcal{P}$ contains all or none of the proof-nets of that type.*

One criticism of Thm. 1 is that in simulating the acceptance function $M(x)$ defined by TM $M$ on input $x$, *another* parameter is introduced: a *doubly-exponential iterator* for the computation. By contrast, the polynomial-time simulation only has one argument, because the input list can be coerced into also being the iterator. The doubly-exponential simulation is then rejected because a list cannot be coerced into a suitably powerful iterator. We reject this criticism: why can't a transducer generate multiple data structures to represent $x$?

Another criticism is that Thm. 1 does not represent TM $M$ by a proof-net, but by a *family* of proof-nets. Nonetheless, they are all identical, except for the number of enclosing boxes. By analogy, consider the simulation of a polynomial-time TM $M$ by a *family* of circuits $C_n$, one for each length input. The usual condition is that the $C_n$ be *uniform*: each can be generated by a transducer in $O(\log n)$-space from a representation of $n$. The family $P_n$ is certainly uniform in this sense. Furthermore, the transducers use the same time and space so as complexity black boxes they are not distinguishable. The construction presented here simply just gets more value for the money.

## 6 Conclusions

We have identified the crucial rôle of squaring to understand the complexity of LAL. Identifying squaring as the worst-case behavior when reducing proof net level, we could reprove the upper bound on LAL normalization using a simple geometric argument. We have also shown how to slightly relax the conventions to allow a uniform family of TM in proving the expressiveness of LAL. This makes the complexity bounds tight in the sense that the expressive power exploits the full power of normalization. This shows that the LAL as a logic has the power of exponential time rather than polynomial time.

We conjecture that other logics of polynomial time, for example Hofmann's modal calculus [12, 11] and Lafont's Soft Linear Logic [16], are prone to the

same game as illustrated here: to be able to handle the bounds of an arbitrary polynomial $n^k$ the systems must have the flexibility of letting $k$ grow arbitrarily. When we let a TM correspond to a family of encodings indexed by the size of the input tape, the encodings can exploit the possibility of varying $k$. It should thus be possible to show completeness for at least *exponential time* for each of the systems. They are, then, only logics of polynomial time in a restricted sense.

It is worth noting that the same issues are well recognized in the context of simply-typed $\lambda$-calculus: A well-known result of Schwichtenberg is the following. Let $\mathsf{Int}_\tau = (\tau \to \tau) \to (\tau \to \tau)$ be the type of the standard coding of integers as Church numerals, and let $\mathsf{Int} = \mathsf{Int}_o$; then the only representable functions of type $\mathsf{Int} \to \mathsf{Int} \to \cdots \to \mathsf{Int}$ are those functions called the "extended polynomials," defined by constants, addition, multiplication, and a conditional branch on zero [19]. One would imagine, similarly, that functions of type $\mathsf{Int}_{o\to o} \to \mathsf{Int}_{o\to o} \to \cdots \to \mathsf{Int}$ would characterize "extended exponentials," with similar generalizations to higher types. As the *order*[6] (degree of higher-order functionality) of these types grow, indicating their capacity for higher-order iteration, so should their expressive power.

This added power is identified in Statman's theorem: given an elementary function $K_i(n)$ (where $K_0(n) = n$ and $K_{i+1}(n) = 2^{K_i(n)}$), he showed how to write a short $\lambda$-term $E_{M,x}$ which normalized to a coding of "true" iff TM $M$ halted on $x$ in $K_i(|x|)$ steps [20]. We say "short" to mean that $E_{M,x}$ could be computed with inputs $x$ and $M$, where the transducer uses only $O(\log |x|)$ space. Were the decision problem to be solvable in $\mathrm{DTIME}[K_j(n)]$, then choosing a large enough $i$ would contradict the time hierarchy theorem. The details of the proof are to show how to construct a TM simulation and how to iterate an elementary number of times.

Statman's construction does not follow Schwichtenberg's fixed input/output convention in the following sense: his simulation of $\mathrm{DTIME}[K_i(n)]$ is realized by a family of $\lambda$-terms whose type and structure vary as a function of $i$. Nonetheless, they are *uniform* in that each such $\lambda$-term can be constructed by a TM transducer that runs in $O(\log i)$ space. The key difference with Schwichtenberg's conventions is that the input determines certain structure in the function—namely, the power of some *iterator*—which is explicitly coded as a $\lambda$-term.

In later research, Hillebrand, Kanellakis and Mairson [15] attempted to liberate Statman's expressiveness results from Schwichtenberg's bottleneck by merging simply-typed $\lambda$-calculus with simple database constructs, allowing $\lambda$-terms to compute functions between databases. They were able to relate the order of simply-typed terms to various complexity classes.

Exactly the same kind of scenario gets played out when we consider the expressive power of LAL as a logic and programming language. The normalization procedure is doubly exponential (instead of non-elementary), and is polynomial for fixed depth (elementary for fixed order).

---

[6] Recall the order function $o(t) = 1$ for a base type $t$, and $o(s \to t) = \max\{1 + o(s), o(t)\}$.

## References


1. Andrea Asperti. Light affine logic. In *Proc. 13th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 300–308, 1998.
2. Andrea Asperti and Luca Roversi. Intuitionistic light affine logic (proof-nets, normalization complexity, expressive power, programming notation). *ACM Transactions on Computational Logic*, 3(1):1–39, January 2002.
3. Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *Proc. 24th Symp. Theory of Computing*, pages 283–293, 1992.
4. Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.
5. J.-Y. Girard. Linear logic. *Theoret. Comput. Sci.*, 50:1–102, 1987.
6. Jean-Yves Girard. Linear logic: its syntax and semantics. In Girard et al. [9], pages 1–42.
7. Jean-Yves Girard. Proof-nets: The parallel syntax for proof-theory. In Ursini and Agliano, editors, *Logic and Algebra*. Marcel Dekker, Inc, New York, 1996.
8. Jean-Yves Girard. Light linear logic. *Inform. & Comput.*, 143:175–204, 1998.
9. J.-Y. Girard, Y. Lafont, and L. Regnier, editors. *Advances in Linear Logic, Proceedings of the 1993 Workshop on Linear Logic*, London Math. Soc. Lecture Note Series 222. Cambridge University Press, 1995.
10. Jean-Yves Girard, Yves Lafont, and P. Taylor. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989.
11. M. Hofmann. Linear type and non-size-increasing polynomial time computation. In *Proc. 14th Ann. IEEE Symp. Logic in Comput. Sci.*, July 1999.
12. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Theoret. Comput. Sci.* To appear.
13. Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, August 1987.
14. Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. Foundations of Computing. MIT Press, 1997.
15. P. C. Kanellakis, G. G. Hillebrand, and H. G. Mairson. An analysis of the Core-ML language: Expressive power and type reconstruction. In *Proc. 21st Int'l Coll. Automata, Languages, and Programming*, volume 820 of *LNCS*, pages 83–106, 1994. Invited paper.
16. Yves Lafont. Soft linear logic and polynomial time. *Theoret. Comput. Sci.* to appear.
17. Yves Lafont. From proof-nets to interaction nets. In Girard et al. [9], pages 225–247.
18. Luca Roversi. Light affine logic as a programming language: a first contribution. *Int'l J. Foundations Comput. Sci.*, 11(1):113–152, 2000.
19. Helmut Schwichtenberg. Definierbare Funktionen im $\lambda$–Kalkül mit Typen. *Arch. math. Logik*, 17:113–114, 1976.
20. R. Statman. The typed lambda-calculus is not elementary recursive. *Theoret. Comput. Sci.*, 9(1):73–81, July 1979.
21. Kazushige Terui. Light affine lambda calculus and polytime strong normalization. In *Proc. 16th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 209–220, June 2001.
22. P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*, pages 1–66. MIT Press, Cambridge, MA, 1990.

# A The Detailed Proof-nets

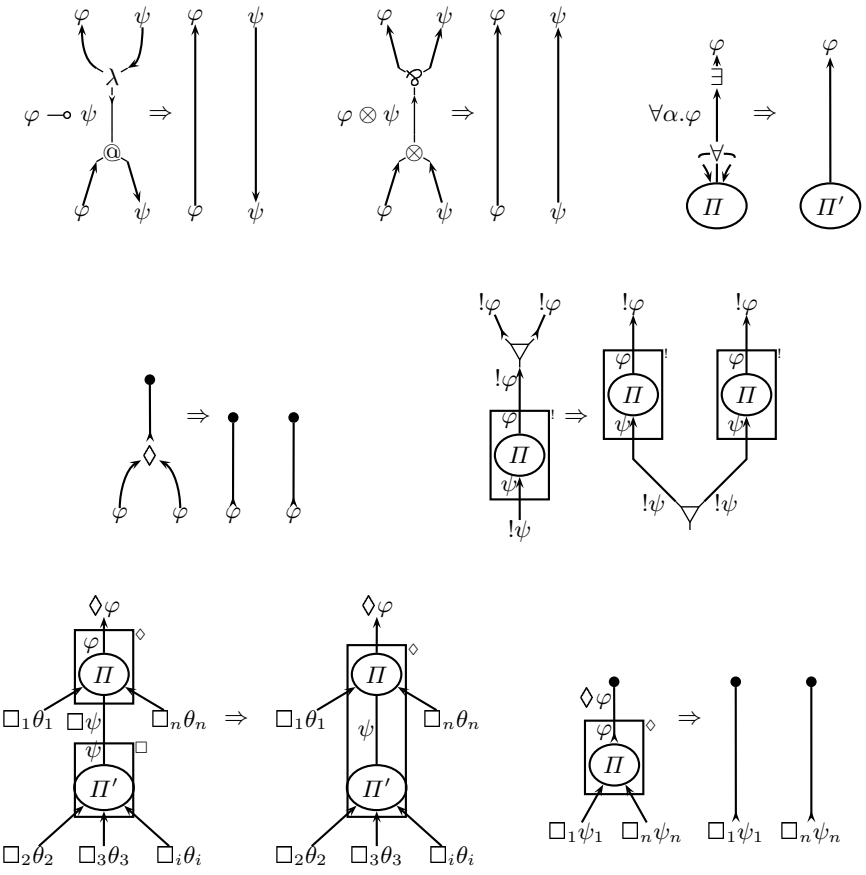


**Fig. 1.** Reductions in LAL. The top row contains the *linear elimination rules*: λ-@-reduction, ⊗-⅋-reduction, ∀-∃-reduction.
The middle row contains the last linear elimination rule, weakening, and the *polynomial elimination rule*, box copying. The latter is the only rule which increases the proof-net size. In the weakening rule ◊ represents any node, while ◊ and □ represent either ! or § in the box copying rule.
The bottom row contains the *shifting elimination rules*: box fusion and box elimination. The ◊ and □ represent either ! or §. Note that the type correctness of the box fusion rule does not require any side conditions, but is ensured by local typing constraints on the wire.

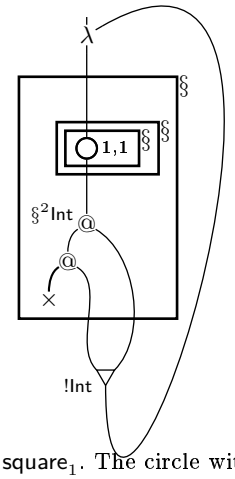


**Fig. 2.** The squaring proof-net $\mathsf{square}_1$. The circle with label $1,1$ is the coercion function $\mathsf{coerce}_{1,1}$.

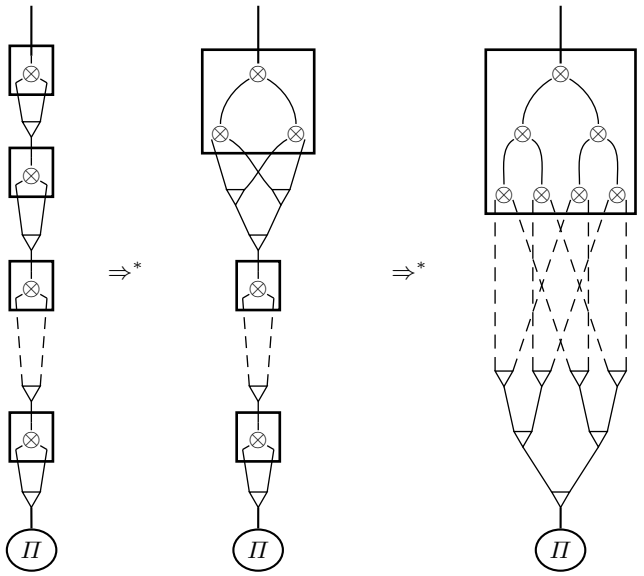


**Fig. 3.** Example of an Elementary Affine Logic "level" where reduction gives an exponential increase. The initial net has $k$ initial sharing nodes and normalizes to the net on the right with $2^k - 1$ $\otimes$-nodes and $2^k - 1$ sharing nodes.

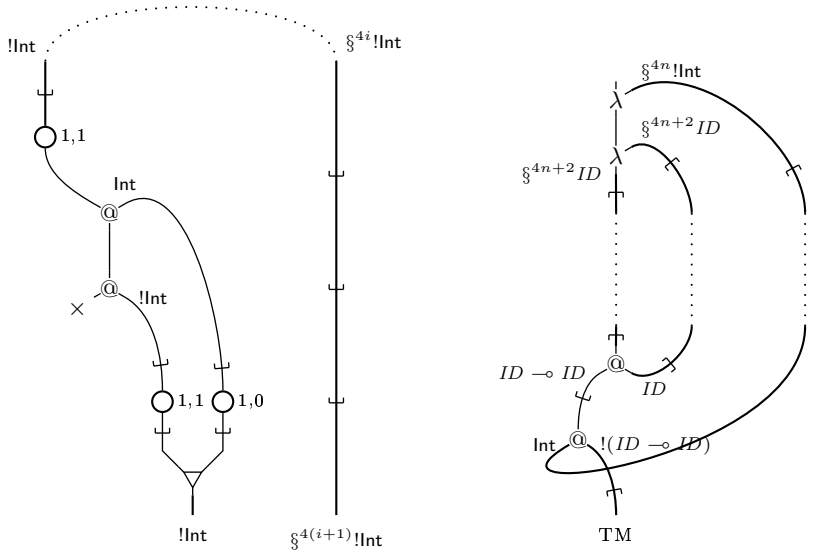


**Fig. 4.** Constructions added for each Turing machine input character. To the left is the addition to the proof-net $N$, The node $\times$ represents the proof net $\mathsf{square}_1$ of Sect. 4. To the right is the addition to the proof-net $I$ constructing an iterator at level $4n+1$. The node TM represents the Turing machine state transition function. In both subfigures the dotted lines are filled with four box builders for each input character.

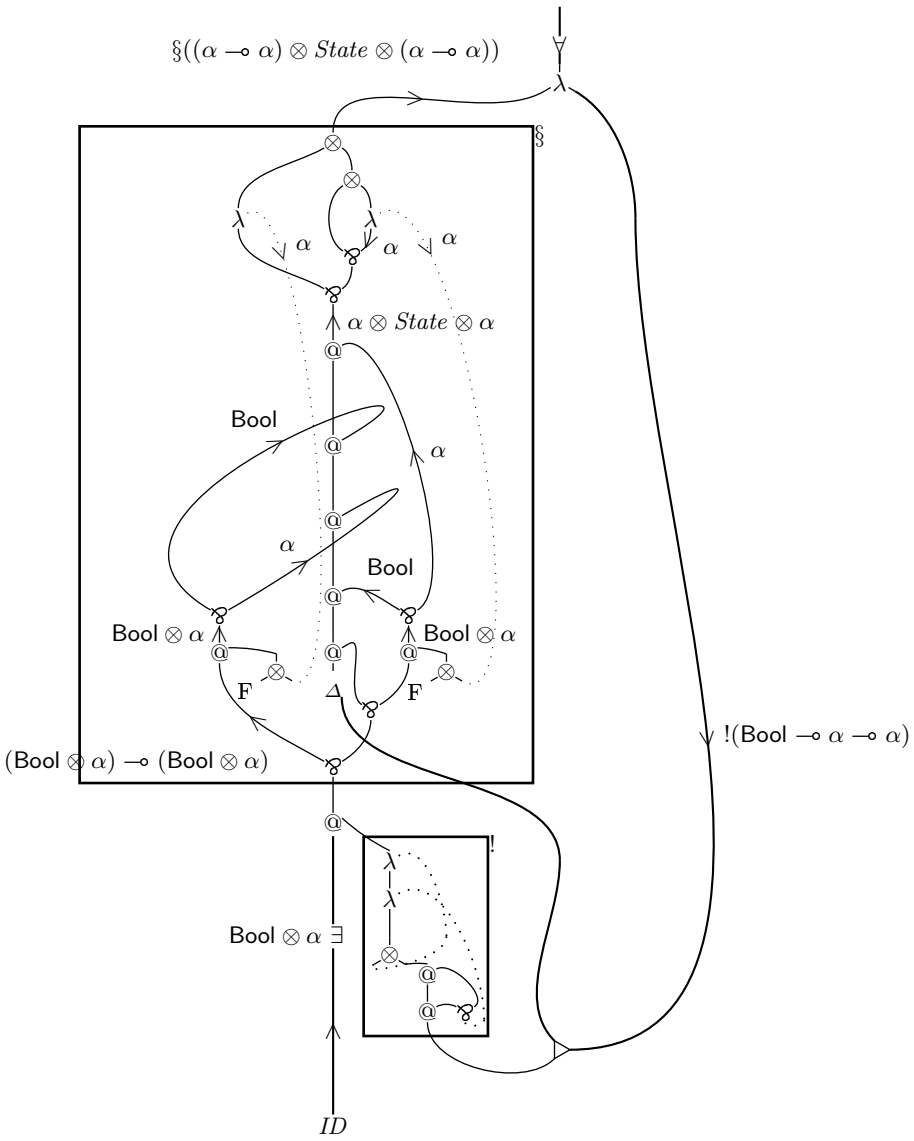


**Fig. 5.** The heart of the TM transition function $\delta$. The proof-net inputs type $ID$ at the bottom and outputs type $ID$ at the top. By abstracting it, we get $\delta$. The term $\Delta$ represents the state transition function. The !-box is the function $C$ of (2).
The proof-net is easiliest understood bottom-up. As in (3) we first instantiate $L$. The §-box then removes the modality § from the instantiation of $L$. The configuration is split into its three components. The current state is fed directly into $\Delta$. The two halves of the tape are applied to the base case consisting of a tuple of false (F in the figure) and the empty list. Afterwards the each half is split into head and tail and fed into $\Delta$. The new configuration as found by $\Delta$ is split in parts and combined with abstractions to get the type right. After leaving the box a $\lambda$ and $\forall$-node give the final abstraction needed for the function.