We have seen that logic and functional languages are the two sides of the same coin. Since we are programmers, not logicians, we will in these notes do away with the logic and focus directly on a “programming language” inspired by linear logic. The programming language will look odd at first since programs are given as graphs. Nevertheless it is a programming language and it is easy to make translations from standard programming languages into the graph notation.

It is often said that linear logic has never had any useful applications. While this might very well be true when looking strictly at linear logic, the idea of linear formulae has emerged as useful in a number of places: handling states in functional language [Wad90], reasoning about the resource bounds of programs [Hof00, Hof], proving correctness of garbage collection [BTSR03], and type inference [KWW02, MNM03].

We proceed in 3 steps: First we develop a graph notation based on our rules for linear logic. Second, we change the rules slightly to get a language (LAL) that can be evaluated in polynomial time. Finally, we show that this language is complete for polynomial time.

1 Linear logic

Let us first recall our inference system for linear logic. We deal only with so-call propositional formulae. This establishes what is true in all domains where we might apply our logic. We are thus only dealing with statements like “if some foo and bar are true, then foobar is also true”, but not the truth of concrete statements like “2 + 2 = 4”, “the moon is made of green cheese”, or “we can buy a pizza”.

If we wanted to deal with a concrete domain, we would need to define the domain syntactically and add domain specific rules. For instance, we could deal with addition by defining the numbers syntactically: \( N ::= 0 \mid s(N) \) and use domain specific rules like \( X + 0 = 0 \) and \( X + s(Y) = s(X + Y) \). We could then derive theorem like \( s(s(0)) + s(s(0)) = s(s(s(0))) \).

The distinction between propositional formulae and a concrete domain is similar to usual mathematics. We always take for granted modus ponens (if you know \( A \) and \( A \) implies \( B \) then you can conclude \( B \)). When studying pure logic we see what can be accomplished using just modus ponens. However, when we go to geometry, set theory, etc., we add more definitions before we start deriving lemmata, propositions, and theorems.
The syntax of our propositional formulae is:

\[
A, B, C ::= X \mid A \rightarrow B \mid A \otimes B \mid A \& B \mid A \oplus B \mid !A \mid \forall X.A
\]

where \(X\) is any propositional variables. The connectives are linear implication (\(\rightarrow\): consume \(A\) to produce \(B\)), a pair (\(\otimes\)), a choice (\&) between \(A\) and \(B\) (called with in linear logic lingo), a possibility (\(\oplus\)) for either \(A\) or \(B\) (where we cannot choose which), an unlimited supply (!—bang among friends and linear logicians), and a universal quantification (\(\forall\)). The latter corresponds to higher-order types in their most general form: \(X\) can be instantiated with any formula including the formula where the quantifier occurs, i.e., in the formula \(\forall X.X \rightarrow X\) we can replace \(X\) by \(\forall X.X \rightarrow X\) to obtain \((\forall X.X \rightarrow X) \rightarrow (\forall X.X \rightarrow X)\).

We are interested in establishing what formulae we can derive given a set of assumptions. Formally, this is captured in deriving so-called sequents of the form \(\Gamma \vdash A\) where \(\Gamma\) is a multiset of assumptions.\(^1\) The intuitive meaning is that we can conclude \(A\) when we assume \(\Gamma\); thus \(\Gamma \vdash A\) is something that is universally true. Our assumptions come in two flavors:

**Linear assumptions (written \(\langle \cdot \rangle\))**: these assumptions can (and should!) be used exactly once.

**Intuitionistic assumptions (written \([\cdot]\))**: these assumptions can be used an unlimited number of times or not at all. They are like the assumptions we know and love from usual mathematics.

We define an inference system with the following rules:

\[
\begin{align*}
\Gamma \vdash A & \quad (\text{Id}) & \Gamma \vdash A & \quad (\text{Id}) \\
\Gamma, [A], [A] \vdash B & \quad \text{Contraction} & \Gamma, [A] \vdash B & \quad \text{Weakening} \\
\Gamma, [A] \vdash B & \quad \Gamma \vdash !A & \quad \Delta, [A] \vdash B & \quad \vdash !E \\
\Gamma \vdash A \quad \Gamma \vdash !A & \quad \Delta \vdash B & \quad \Gamma, \Delta \vdash B \\
\Gamma \vdash A \rightarrow B & \quad \Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash A \rightarrow B \\
\Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash A \& B & \quad \&-I_1 \\
\Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash B \quad \Gamma \vdash A \& B & \quad \&-I_2 \\
\Delta \vdash C \quad \Delta \vdash C & \quad \Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash A \& B \quad \&-E \\
\Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash A \oplus B & \quad \oplus-1 \\
\Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash A \oplus B & \quad \oplus-2 \\
\Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash B \quad \Gamma \vdash A \oplus B & \quad \oplus-2 \\
\Delta \vdash C \quad \Delta \vdash C & \quad \Gamma \vdash A \quad \Gamma \vdash B & \quad \Gamma \vdash A \& B \\
\Gamma \vdash \forall X.A & \quad \forall-1 & \Gamma \vdash \forall X.A & \quad \forall-E \\
\end{align*}
\]

\(^1\)A multiset is a set where an element can occur multiple times. When using multisets we have \([A, A] \neq [A]\).
where in the rule $\forall$-I the variable $X$ must not occur free in any of the assumptions in $\Gamma$. The rule $\forall$-I might look mysterious at first, but it allows us for instance to derive:

$$
\frac{\langle X \rangle \vdash X}{\vdash X \rightarrow X} \\
\vdash \forall X. X \rightarrow X
$$

which is not so mysterious: no matter how we choose $X$, we can produce $X$ if we get an $X$. Formally we define the free variables $\text{fv}$ of a formula and an environment as follows:

$$
\text{fv}(X) = \{X\} \\
\text{fv}(A \rightarrow B) = \text{fv}(A \otimes B) = \\
\text{fv}(A \& B) = \text{fv}(A \lor B) = \text{fv}(A) \cup \text{fv}(B) \\
\text{fv}(\forall X.A) = \text{fv}(A) \setminus \{X\} \\
\text{fv}(\Gamma) = \bigcup \{A \mid [A] \in \Gamma \text{ or } [A] \in \Gamma\}.
$$

Let us take a look at a concrete derivation\(^2\): to fit the not too wide margins, we use $d = A \rightarrow A$ and $D = d \rightarrow d$:

$$
\begin{align*}
\langle D \rangle & \vdash D \\
[\langle D \rangle, |d|, |d|] & \vdash A \\
\langle D \rangle, |d|, |d| & \vdash A \\
\langle D \rangle, |d| & \vdash A \\
\langle D \rangle & \vdash A \rightarrow A
\end{align*}
$$

When looking at the above derivation, it is obvious that there is a good amount of redundancy: for instance, we keep copying $\langle D \rangle$ into every set of assumptions. With our stated goal of being economical is it natural to do away with the redundancy. This lead us to **proofnets**.

## 2 Proofnets

**Proofnets** represent proofs as a connected graph: the assumptions and conclusion become annotations of the edges (called *wires*) and the nodes correspond to an application of one of rules. As a concrete example consider the rule $\otimes$-E:

$$
\frac{\Gamma, \Delta \vdash B \quad \Delta, \langle A \rangle, \langle B \rangle \vdash C}{\Gamma, \Delta \vdash C}.
$$

\(^2\)Perhaps surprising, the derivation corresponds to a program doing addition . . .
We will represent it as the following proofnet

![Proofnet Diagram]

We will drop $\otimes$ and $\&$ for convenience. Furthermore, we drop the distinction between intuitionistic and linear assumptions; intuitionistic assumptions will simply be annotated with $!A$. We can thus turn our rules into graphs fairly straightforwardly: The $(\text{Id})$ rule becomes

![Graph Diagram]

Contraction and weakening become

![Graph Diagram]

where we have used shades to represent the nets built in the hypothesis of the inference. The rules for $\rightarrow$ become

![Graph Diagram]

---

3It is notoriously difficult to represent these connectives in proofnets and in light affine logic (which is our final goal) it can be recovered using the universal quantification and weakening.
And, as we saw, we get the following rules for \( \otimes \)

\[
\begin{align*}
\Pi & \\
C_1 & \cdots \nonumber
\end{align*}
\]

Finally, we are ready to the rules for bang. Since a bang can only be introduced when all the assumptions are intuitionistic, we need a structure that can span several nodes. We will use a box, which is called a global structure because it can span an arbitrary part of the proofnet. This is accompanied by a gadget called a croissant which works like a can opener on a box. Our 3 rules for bang become:

\[
\begin{align*}
\Pi \quad & \\
\Pi' \quad & \\
\Pi'' \quad & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma & \\
\Delta & \\
\end{align*}
\]

Similarly, the rules for \( \forall \) has constraints on the assumptions. We therefore introduce another kind of box (dashed):

\[
\begin{align*}
\Pi \quad & \\
\Pi' \quad & \\
\Pi'' \quad & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma & \\
\Gamma & \\
\end{align*}
\]
As can be seen, each rule only modify a tiny part of the graph. We can therefore define a proofnet as follows:

**Definition 2.1.** A proofnet is a directed graph meeting the following conditions:

1. The graph vertices (called *nodes*) are
   - free ports ($\bigcirc$) representing the free variables, and a single distinguished one (the root) representing the whole term,
   - weakening nodes ($\otimes$) marking unused function arguments,
   - application ($\odot$) and function nodes ($\lambda$) encoding functions,
   - sharing nodes ($\triangledown$) used when the same variable occurs in two subterms, and
   - croissant nodes ($\leftarrow$) marking variable occurrences.

2. It can contain boxes around a subgraph which meets all the criteria for being a proofnet.

3. Each edge is annotated with a formula from (1) such that the constraints in Fig. 1 hold around each node and on the border of each box.

4. Exactly one free port has its wire oriented toward it; this port is the root of the program.
5. Let a **switching** be the graph derived by removing all the boxes and replacing each $\lambda$-node with one of the configurations in Fig. 2. Every switching results in a forest of connected, acyclic graphs where each graph contains either the root of the net or a weakening node.

The last condition is called the **Danos-Regnier criterion**. That this definition is equivalent to our previous definition can be found for instance in [Laf95, Gir96].

### 3 A programming language

We have now carefully build up the tools to reason about resources. Our next task is to use them. We will do so by defining a typed functional language, light affine logic (LAL), which captures exactly polynomial time. First, however, let us look at the language embedded in linear logic.

![Figure 3: Reduction rules for proofnets. In the rule for quantifier-elimination we should also update all the wires in the box that mentions $X$.](image)

We saw earlier that the type system of a functional language can be read as the definition of logic. We will now go the other way: we will take the rules for
linear logic and read them as a programming language. The equivalence has already been suggested in the symbols we choose for our nodes: linear implication $\multimap$ corresponds to functions (like usual implication corresponds to the type of functions), $\otimes$ corresponds to pairing, and $\triangleleft$ corresponds to duplication of values. We use the rules in Fig. 3. We notice that each of the rules preserves the typing, i.e., if the proofnet has conclusion $A$ and assumptions $B_1, \ldots, B_k$ before reduction then so does the reduced proofnet. This property is called \textit{subject reduction}. Furthermore, using the Danos-Regnier criterion we can see that if we had a proofnet before reduction, the resulting graph is also a proofnet.

It is important to realize that non of the reduction rules depend on the type annotations. If we start with the with a proofnet the rules will always be reasonable—“well-typed programs do not go wrong”. So for a programming point of view we can simply remove the type annotations before we start reducing. This in particular makes life easier when we reduce one of the dashed quantifier boxes as we do not need to update the type annotation on the wires.

It also worth noticing how proofnets in many ways resembles (on the abstract level) what happens inside an interpreter for a functional language, e.g., each of the variable occurrences has a pointer to the value, we make copy of an argument for each occurrence of the formal parameter (even if we do not physical copy it, we still have to read the value several times).

![Figure 4: An “untyped proofnet” that can be reduced infinitely.](image)

Figure 4: An “untyped proofnet” that can be reduced infinitely.

To appreciate the expressive power of the system note that it allows an infinite chain of reductions if we ignore the types. This is illustrated in Fig. 4. It is an important fact there is no way to annotate the labels such that graph becomes a proofnet.

\textbf{Exercise 3.1.} Show that if you use the reduction rules for proofnets on the graph in Fig. 4, the reduction can be continued forever.

\textbf{Exercise 3.2.} Show that there it is not possible to type annotate the graph in Fig. 4.

Finally, it is worth noting that the rules can be completely localized thus
presenting an easy to parallelize algorithm for reduction. This is done with optimal reduction (which might not be so optimal after all, see [AM98], but also [ACM00])—this is described in [Mai02].

4 A polynomial time language

We will now finally see how we can turn the functional language just presented into a polynomial time language. To get there we need to understand how it gets its tremendous power in terms of running time.

We start out with a closer look at the example of a net leading to an infinite reduction. This is possible (in part) because boxes can be opened up. We will therefore change the language such that all nodes stay at fixed depth.

Our second observation is that it is only duplication that increases the size of the proofnet and thus pumps up the running time. We can make the observation stronger by noting that we get an exponential growth in the following way: make a chain of boxes where each box is shared (i.e., copied) and the two copies go into the next box. If the net inside the box somehow pairs the two copies, we get a net that is exponential in the size of the original net. We will therefore only allow boxes with at most one input to be copied. This is a bit like an oncologist curing cancer by killing the patient: we need boxes with more than one input. We therefore introduce a new modality \( \& \) on top of \(!\) such that boxes are either of type \( \& \) or \(!\) and allow the \( \&\)-boxes to have multiple input.

Finally, to make life easier for the working programmer, we allow weakening of an arbitrary net, not just stuff in a box. This results in the formation rules of Fig. 5 and reduction rules of Fig. 6. We will refer to the (fixed) number of boxes surrounding a node as its level or index.
Some notes on the rules: In the weakening rule $\Diamond$ represents any node. In the rules involving boxes (box copying and box weakening) $\Diamond$, $\triangle$, and $\Box_i$ represent either $!$ or $\exists$. It is an easy checkable fact that any choice of $!$ or $\exists$ for $\Diamond$, $\triangle$, and $\Box_i$ in the box rules make the rules sound when one remember that a $!$-box has at most one input wire. In the rule for quantifier-elimination we should also update all the wires in the box that mentions $X$. 

Figure 6: Reduction rules for light affine logic.
5 Expressiveness

It is surprisingly simple to figure out the running time of light affine logic:

1. All nodes stay at fixed level inside a given number of boxes. We therefore need only to figure out what happens at a single level.

2. Duplication is the only rule that increases the size of the net; all other rules reduce the number of nodes or the number of boxes and thus take only linear time to reduce.

The following strategy will completely reduce a level:

1. Reduce all linear redexes, i.e., redexes involving \( \rightarrow, \times \), or weakening.

2. Do all the duplication. As all the wires around a sharing node have type \( !A \) this cannot create new linear redexes. Furthermore due to the Danos-Regnier criterion, the sharing nodes are organized in a forest so this terminate when the nodes have copied all the leaves.

3. Do all the merging. This can only create new redexes at a level one deeper.

To figure out the cost assume that there are \( k \) sharing nodes at the level, \( m \) nodes inside boxes being copied, and \( n \) other nodes in the net. The size of the net before reducing the level is \( N = k + m + n \), while it is at \( N' = k \cdot m + n \leq (k + 1) \cdot (m + n) \). We note that \( k = N - (m + n) = N - x \cdot N \) for some \( x \) where \( 0 \leq x \leq 1 \). We therefore have

\[
N' \leq ((N - N \cdot x) + 1) \cdot x \cdot N = -x^2 \cdot N^2 + x(N + N^2) .
\]

This inequality bounds \( N' \) by a second degree polynomial and we find that the maximum occurs for \( x = 1/2 + 1/(2N) \) where we have \( N' = 1/4(N + 1)^2 \). This means for \( N \geq 2 \) we have \( N' \leq N^2 \), i.e., the size of the proofnet is at most squared when we reduce a level. As the proofnet has a fixed number of levels, say \( l \), we obtain:

**Theorem 5.1.** Let \( \Pi \) be a proofnet with \( N \) nodes and \( l \) levels. The size of the reduced proofnet is at most \( N^2 \). This can be computed in time \( O(N^2) \).

**Proof.** The bound of the size follows from our discussion above.

As for the time we first note that for each level the redexes in each of the three steps can be found by doing a depth-first search of the net. When we have done a reduction we check whether it created a new redex. The time of a depth-first search is linear to the size of the net. Consequently, the linear reductions can be done in linear time. The duplication takes at most time \( O(N^2) \) (the time to do the duplication), and the box merging takes time \( O(N^2) \) (because the net is bigger). This bound the time to handle one level as \( N^2 \) in the size of the level. With \( l \) levels we get the \( O(N^2) \) upper bound.

\[\square\]
Looking at the bound, you are bound to ask: this was supposed to give us polynomial time, but we have a double-exponential bound!? Polynomial time however arises in the following sense: Take any polynomial time Turing machine. Then there is fixed proofnet with depth $l$ that corresponds to the Turing machine. Since the proofnet has depth $l$ the evaluation will take place in polynomial time. The details will be the topic of our section.

6 Embedding polynomial time in LAL

As the final step we will show that any polynomial time Turing machine can be simulated in LAL. Our goal is to translate the Turing Machine into a function (represented as a proofnet). We then require that we can translate every input tape into a proofnet and get the result of running the Turing machine by applying the function to the translated tape.

We follow our by-now standard recipe for proving soundness:

1. choose a representation of tapes and other data types needed
2. establish a way to iterate the needed number of times
3. show how to do the transition function.

We will first see that we can elegantly implement lists using higher-order functions using just one level. This allow us to recycle the idea of representing a tape as two lists. Numbers can then also be represented easily as a list where the elements are empty. Furthermore, we are capable of making a copy of the tape of put it inside boxes as we need.

The next step is to realize that we can compute any polynomial $c \cdot x^k$ in $O(\log k)$ levels. Finally, we shall see that the transition function can be realized in just one level. This is not too surprising given that the transition function handles all data linear: data is moved from one list to the other or it simply reads or write the data from the tape. This allows us to make $c \cdot x^k$ iterations of the transition function at level $O(\log k)$

We then translate any Turing Machine as follows: given an input tape make a copy. Use the one copy to find the length and compute $c \cdot |x|^k$. The other copy is put inside $O(\log k)$ boxes and used as fodder for the iteration of the transition function.

6.1 Representing numbers and tapes

We know how lists can be represented for instance in Scheme using cons with the empty cell nil as terminator. This can be mimicked in LAL and other functional languages using higher-order functions. This is illustrated in Fig. 7 where a list with elements of type $A$ is represented as a function of type $\forall X.((A \to X \to X) \to \#(X \to X))$. One can look at the list as simply build using $\text{cons}(c)$ and $\text{nil}(n)$. When we supply functions $c$ and $n$ we replace
the \texttt{cons} and \texttt{nil} by the functions \texttt{c} and \texttt{n}; this becomes primitive recursion over the list: \texttt{n} represents the base case and \texttt{c} the iterative step.

This way of representing lists is fairly faithful: Looking at the proofnet notation it is clear that it closely resembles linked list which are the standard way to represent links. It is however a limiting factor in ease of use that we only have primitive recursion. This is illustrated by the function \texttt{tail} : \texttt{List} \( A \rightarrow \text{List} \ A \) in Fig. 8: as we have to build the result iteratively each node computes the pair consisting of \((\text{hd} \ L, \text{tl} \ L)\) and at the top we return only the second argument. This corresponds to the following program:

\[
\text{tail} \ l = \text{snd} \ l \\
\text{tail}' \ [] = ([],[]) \\
\text{tail}' \ (x : \text{xs}) = \text{let} \ (h',t') = \text{tail} \ \text{xs} \ \text{in} \ (x,(h' : t'))
\]

where \( ? \) is any element of \( A \).

With primitive recursion (called \texttt{foldr} (for fold right) by functional programmes) this looks like:

\[
\text{tail} \ l = \text{snd} (\text{foldr} (\text{(?,[]))) (\lambda x.\text{hd} \otimes \text{tl'},(x,(\text{cons} \ h' \ t'))) \ l)
\]

It is also worth noting that due to the type restrictions of LAL we cannot make a function \texttt{head} : \texttt{List} \( A \rightarrow A \), but only of type \texttt{List} \( A \rightarrow \{A\} \).

\footnote{Using second order quantification we could construct a so-called option-type where we explicitly mark that we do not have an element.}
Figure 8: Function tail : ListA \to ListA computing the tail of list. The question mark stands in the net stands for any net of type A.
Exercise 6.1.

1. Explain why it is not possible to make a function \( \text{head} : \text{List}A \rightarrow A \).

2. Show how to compute \( \text{head} : \text{List}A \rightarrow 8A \)

The same idea can be used for numbers. We represent them in unary with a list where nothing is stored in the cells. This is called Church numerals and results in the type \( \text{Int} = \forall X.!(X \rightarrow X) \rightarrow 8(X \rightarrow X) \).

Finally, we will represent the tape as two lists: one with the contents to the left of the head and one with the contents to the right of the head. For technical reasons we let the two tape parts share the same constructor and combine it with the program counter into a state, i.e., \( \text{State} = \forall X.X \otimes X \otimes X \otimes X \rightarrow X \) where \( X \otimes X \rightarrow \text{PgmCtr} \otimes X \otimes X \) represents a single square on the tape and \( \text{PgmCtr} \) the program counter. This type should be read as follow: when you provide the iterator \( !(X \rightarrow X) \) and the two base cases \( X \otimes X \) you get back a pair consisting of the two lists. We choose \( \text{Cell} = \forall X.X \otimes X \otimes X \rightarrow X \) which is just a 3-way choice. Similar, if there are \( m \) instructions in the program we choose \( \text{PgmCtr} = \forall X.X \otimes \cdots \otimes X \rightarrow X \) as \( m \)-ways choice.

6.2 How high can we go?

With the datatypes in place the scene is set to figure out how high we can count. This will take us through a couple of encodings of arithmetic on our funky Church numerals.

The first thing to notice is that we can do addition in LAL. This is shown in Fig. 9. This function has type \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \). Given addition we can also code multiplication of \( m \) with \( n \): simply add \( n \) to zero \( m \) times, see Fig. 10. As we are potentially using \( n \) more than once it must have a bang type. We therefore get:

\[ \times' : \text{Int} \rightarrow !\text{Int} \rightarrow !\text{Int} \]

We recover a multiplication \( \times : \text{Int} \rightarrow \text{Int} \rightarrow 8^2\text{Int} \) using a standard trick from the LAL programmers toolbox. Given an integer we can embed it in (almost) any number of boxes we want one by using the coercing function in Fig. 11. The function has type \( \text{Int} \rightarrow 8^p!q!\text{Int} \) for any \( p \geq 1, q \geq 0 \). Given the number \( n \), the function starts from zero embedded in \( p \) and \( q \) boxes and iterates successor \( n \) times. The result is \( n \) embedded in the given number of boxes. We thus get \( \times \) working on \( m \) and \( n \) from \( \times' \) as follows: coerce \( m \) to type \( 8!\text{Int} \), coerce \( n \) to type \( 8!!\text{Int} \), and then use \( \times' \) to obtain \( m \cdot n \) of type \( 8^3\text{Int} \).

Finally we can use \( \times \) to build the squaring widget in Fig. 12. The squaring proof-net has type \( !\text{Int} \rightarrow 8!!\text{Int} \). By combining \( m \) squaring widgets we get a function of type \( !\text{Int} \rightarrow 8!!m!!\text{Int} \) which computes \( n^{2^m} \) given a number \( n \). This function has depth \( 4k + 1 \) and we can thus compute an upper bound of any polynomial \( p(x) = c \cdot x^k \) in depth \( 4(\log k + 1) \) levels.

\[ \text{It would of course be a more natural choice to use } \forall X.X \& X \& X \rightarrow X, \text{ but even though } \& \text{ can be encoded in LAL using } \otimes \text{ and second order quantification it is not worth the trouble.} \]
Figure 9: Addition in LAL
Figure 10: Multiplication in LAL
Figure 11: The coercing function of type $\text{Int} \to \mathcal{P}^{p,q}\text{Int}$ for any $p \geq 1, q \geq 0$. 
6.3 Manipulating the tape

The core of the transition function $\delta$ is shown in Fig. 13. It is actually easier to understand that it appears at first sight: At the bottom we input the old state of type $\text{State} = \forall X.!(\text{Cell} \to X \to X) \to \#(X \otimes X \to \text{PgmCtr} \otimes X \otimes X)$ (hence the outermost quantifier box and the $\lambda$-node at the top). Reading the net bottom-up it proceeds as follows:

1. we split the two lists up into the first element and the rest. Conceptually it is done with the following functional program:

$$
\text{headtail} \ [] = (B, []) \\
\text{headtail} \ (x : xs) = \text{let} \ (h', t') = \text{headtail} \ xs \ \text{in} \ (x, (h' : t'))
$$

If the list is empty we simply define the head to be the blank symbol. With primitive recursion this looks like:

$$
\text{let} \ (h, t) = \text{foldr} \ (B, []) \ (
\lambda x. \lambda h'. \otimes t'. ((x, (\text{cons} \ h' \ t'))) \ l

$$

This is implemented by the $!$-box which is used as the constructor for the old Turing machine case. Note that we effectively add a blank space to each end of the tape.

Technically, we are required to do a couple of things: The old state has type $\text{State} = \forall X.!(\text{Cell} \to X \to X) \to \#(X \otimes X \to \text{PgmCtr} \otimes X \otimes X)$. We therefore first instantiate $X = \text{Cell} \otimes X$ where $X$ is the type variable used by the quantifier box. This matches with the $!$-box having type $!(\text{Cell} \to X \to X)$. So after applying the old state to the $!$-box we have a function of type $\#(X \otimes X \to \text{PgmCtr} \otimes X \otimes X)$ which given the base cases $((B, []))$ will
Figure 13: The Turing Machine transition function $\delta$. 
give us the program counter and the two pairs of a head and a tail. This has to be done inside the \$\$-box.

2. Having decomposed the old state, we feed the program counter to the function $\Delta$ which chooses the relevant command. As $\text{PgmCtr} = \forall X. X \otimes \cdots \otimes X \rightarrow X$ this is done by applying the program counter to a tuple consisting of functions implementing each of the lines in the Turing machine. Each of these functions have type $\text{Cell} \rightarrow \text{Cell} \rightarrow X \rightarrow X \rightarrow \text{State} \otimes X \otimes X$ and simply produce the new configuration given the two heads and two tails. It can use any fixed numbers of list constructors $c$ since we have copied them outside the box.

3. Finally we wrap up all the abstractions we have opened.

**Exercise 6.2.** Show how to translate each of the four Turing Machine commands `left`, `right`, `write X`, and `if X then l else l'` into proofnets. Each of the translations can use an arbitrary, but fixed number of list constructors $c$. *(Hint: For the `if`-command note that you can copy a tape symbol without using boxes by applying it to a tuple of $(B \otimes B) \otimes (0 \otimes 0) \otimes (1 \otimes 1)$.)

Combining the pieces from this and the two previous subsections we conclude:

**Theorem 6.3.** Let $M$ be a Turing Machine which terminates in $O(c \cdot |t|^k)$ for all input tapes $t$. Let $T$ be the encoding of any tape as a list in proofnets. There exists a proofnet $\Pi_M$ of type $\Omega \rightarrow \mathcal{O}(\log k) \Omega$ using $O(\log k)$ levels which simulates $M$ as follows: for any input $t$ applying $\Pi_M$ to $T(t)$ results in $T(M(t))$ embedded in log $k$-boxes.

**Proof.** When given the length of a tape we know how to compute $c \cdot |t|^k$ and iterate $\delta$ that number of times. So the two bits we are missing are:

1. How to get the tape to the appropriate depth;
2. How to compute the length.

The first step is done with a function similar to `coerce`. The second part is done by iterating over this list simultaneous building a copy and computing length. It corresponds to

$$
\text{lengthlist } [] = (0, [])
$$

$$
\text{lengthlist } (x : xs) = \text{let } (n, l) = \text{lengthlist } xs \text{ in } (n + 1, (h : l))
$$

which is rather similar to `tail` show in Fig. 8.

**6.4 An unpleasant twist**

After all this painstaking details, it is a little painful to realized that there is an unpleasant lack in our definitions. As discussed on p. 5 after Theorem 5.1 the bounds for LAL are double exponential. We can actually realize those bounds as...
follows by building the iterator when we encode the list. This results in a stack of squaring devices with a height that is exactly the length of the input tape. At the end of the translation we have to add the translated transition function.

The translations used in Theorem 6.3 can be done in $O(|M|)$ time and $O(\log |M|)$ space for $\Pi_M$ ($|M|$ is the number of lines in the Turing machine program) and $O(|t|)$ time and $O(1)$ for the tape. However, the construction sketched above can be done in exactly the same bounds because the transition function is fixed for all input tapes. It can be stated as follows:

**Theorem 6.4.** For any Turing machine $M$ which in $O(2^{2^{|t|}})$ steps accepts or rejects an input tape $t$, there exists families indexed by $n = |t|$ of LAL proofnets $\Pi_M^n : \text{Pre}^{4n+2}\text{Int} \rightarrow \text{State}^{4n+2}\text{State}$ and $\Pi_N^n : \text{Pre}^{4n+2}\text{Int}$ such that, for any $n$-length input $t \in \{B, 0, 1\}^n$ to $M$ there exists a term $\Pi^t : \text{Pre}^{4n+2}\text{State}$ with the property that $\Pi_M^n \Pi^t$ reduces to $\Pi_N^n \Pi^t$ which encodes the resulting tape. The proofnets $\Pi_M^n$ are all identical except in the number of surrounding boxes. The terms $\Pi_n, \Pi_N^n$, and $\Pi^t$ can be build from input $t$ in $O(1)$ space and $O(|t|)$ time.

This is proved in [MNM02].

7 Background information

Linear Logic was first developed by Girard for purely logical reasons; a good introduction aimed at programming language is [Wad93] while a very good introduction to the logical side is [Gir95]. Light linear logic was introduced by Girard [J.Y98], but made accessible by Asperti as light affine logic [Asp98, AR02]. The complexity proofs in these notes are developed by Møller Neergaard and Mairson [MNM02].

References


[Gir95] Jean-Yves Girard. Linear logic: its syntax and semantics. In Girard et al. [GLR95], pages 1–42. Referenced on pp. 22


[Laf95] Yves Lafont. From proof-nets to interaction nets. In Girard et al. [GLR95], pages 225–247. Referenced on pp. 7


