P-time Completeness of Light Linear Logic and its Nondeterministic Extension

Satoshi Matsuoka, *National Institute of Advanced Industrial Science and Technology,*
1-1-1 Umezono
Tsukuba, Ibaraki
305-8561 Japan
matsuoka@ni.aist.go.jp

Abstract

In CSL'99 Roversi pointed out that the Turing machine encoding of Girard's seminal paper "Light Linear Logic" has a flaw. Moreover he presented a working version of the encoding in Light Affine Logic, but not in Light Linear Logic. In this paper we present a working version of the encoding in Light Linear Logic. The idea of the encoding is based on a remark of Girard's tutorial paper on Linear Logic. The encoding is also an example which shows usefulness of additive connectives.
Moreover we also consider a nondeterministic extension of Light Linear Logic. We show that the extended system is **NP**-complete in the same meaning as **P**-completeness of Light Linear Logic.

Keywords: Light Linear Logic, proof nets

1 Introduction

In [Rov99], Roversi pointed out that the Turing machine encoding of Girard's seminal paper [Gir98] has a flaw. The flaw is due to how to encode configurations of Turing machines: Girard chooses $\mathbf{list^p} \otimes \mathbf{list^p} \otimes \mathbf{bool^q}$ as the type of the configurations, where the first argument $\mathbf{list^p}$ represents the left parts of tapes, the second argument $\mathbf{list^p}$ the right parts, and the third argument $\mathbf{bool^q}$ states. But it is impossible to communicate data between the first and the second in this type: the communication is needed in transitions of configurations. Roversi changed the type of configurations in order to make the communication possible and showed that an encoding of Turing machines based on the type works in *Light Affine Logic*, which is Intuitionistic Light Linear Logic with unconstrained weakening and without additives. But he did not sufficiently discuss whether his encoding works in *Light Linear Logic*.
In this paper, we show an encoding of Turing machines in *Light Linear Logic*. This completes **P**-time completeness of Light Linear Logic with Girard's Theorem [Gir98] that states computations on proof nets with fixed depth in Light Linear Logic belong to class **P**. The idea of the encoding is based on a remark of Girard's tutorial paper on Linear Logic [Gir95]:

> *Affine linear logic* is the system of linear logic enriched (?) with weakening. There is no much use for this system since the affine implication between A

and B can be faithfully mimicked by $1\&A \multimap B$.

Roversi's encoding exploits weakening to discard some information after applications of iterations. Our encoding uses $A\&1$ as type of data that may be discarded. On the other hand Light Linear Logic retains principle $!A\otimes!B \multimap !(A\&B)$. Because of this principle, we can obtain a proof of $!1\otimes!A \multimap !B$ or $!1\otimes!A \multimap \S B$ from a proof of $1\&A \multimap B$ in Light Linear Logic. The obtained proof behaves like a function from $!A$ to $!B$ or $\S B$, not that of $!(1\&A)$: in other words, outside boxes we can hide additive connectives which are inside boxes. That is a reason why the encoding works in Light Linear Logic.
On the other hand we also try to simplify lazy cut elimination procedure of Light Linear Logic in [Gir98]. The attempt is based on the notion of chains of $\oplus$-links. The presentation of Girard's Light Linear Logic [Gir98] by sequent calculus has the *comma* delimiter, which implicitly denotes the $\oplus$-connective. The comma delimiter also appears in Girard's proof nets for Light Linear Logic. The introduction of two expressions for the same object complicates the presentation of Light Linear Logic. We try to exclude the comma delimiter from our proof nets.
Next, we consider a nondeterministic extension of the Light Linear Logic system. Our approach is to introduce a self-dual additive connective. The approach is also discussed in a recently appeared paper [Mau03]. But the approach was known to us seven years ago [Mat96]. Moreover, our approach is different from that of [Mau03], because we directly use the self-dual additive connective, not SUM rule in [Mau03] and we use a polymorphic encoding of nondeterminism. In particular, our approach does not bother us about commutative reduction between nondeterministic rule and other rules unlike [Mau03].

2 The System

In this section, we define a simplified version of the system of Light Linear Logic (for short LLL) [Gir98]. First we present the formulas in the LLL system. These formulas (F) are inductively constructed from literals (T) and logical connectives:

$$\begin{aligned} T &= \alpha \mid \beta \mid \gamma \mid \ldots \mid \alpha^{\perp} \mid \beta^{\perp} \mid \gamma \mid \ldots \\ F &= T \mid 1 \mid \perp \mid F \otimes F \mid F \parr F \mid F \& F \mid F \oplus F \mid !F \mid ?F \mid \$F \mid \forall\alpha.F \mid \exists\alpha.F. \end{aligned}$$

We say unary connective \$ is *neutral*. Girard [Gir98] used the symbol § for the connective. But we use \$ since this symbol is an ascii character.

Negations of formulas are defined as follows:

- $(\alpha)^{\perp} \equiv_{\mathrm{def}} \alpha^{\perp}, (\alpha^{\perp})^{\perp} \equiv_{\mathrm{def}} \alpha$
- $1^{\perp} \equiv_{\mathrm{def}} \perp, \perp^{\perp} \equiv_{\mathrm{def}} 1$
- $(A \otimes B)^{\perp} \equiv_{\mathrm{def}} A^{\perp} \parr B^{\perp}, (A \parr B)^{\perp} \equiv_{\mathrm{def}} A^{\perp} \otimes B^{\perp}$
- $(A\&B)^{\perp} \equiv_{\mathrm{def}} A^{\perp} \oplus B^{\perp}, (A \oplus B)^{\perp} \equiv_{\mathrm{def}} A^{\perp}\&B^{\perp}$
- $(\forall\alpha.A)^{\perp} \equiv_{\mathrm{def}} \exists\alpha.A^{\perp}, (\exists\alpha.A)^{\perp} \equiv_{\mathrm{def}} \forall\alpha.A^{\perp}$
- $(!A)^{\perp} \equiv_{\mathrm{def}} ?A^{\perp}, (?A)^{\perp} \equiv_{\mathrm{def}} !A^{\perp}$
- $(\$A)^{\perp} \equiv_{\mathrm{def}} \$A^{\perp}$

We also define linear implication $\multimap$ in terms of negation and $\parr$-connective:

$$A \multimap B \equiv_{\mathrm{def}} A^{\perp} \parr B$$

In this paper we do not present sequent calculus for Light Linear Logic. Instead of that, we present a subclass of Girard's proof nets for Light Linear Logic, *simple proof nets* (precisely, simple proof nets can be mapped into a subclass of Girard's proof nets). Although there is a proof net that is not simple in the sense of [Gir96], simple proof nets are sufficient for our purpose, encoding of Turing machines, because nonsimple proof nets never occur in our encoding. Moreover it is possible to translate proof nets in the sense of [Gir96] into simple proof nets although simple proof nets are generally more redundant than nonsimple proof nets.
A simple proof net consists of formulas and links. Figure 1 shows the *links* in LLL: $F_{\oplus}(A_1, \ldots, A_p)$ represents a formula that is generated from formulas $A_1, \ldots, A_p$ by using $\oplus$-connective and is called *general* $\oplus$*-formula*. $S_{\$}(A_1, \ldots, A_p)$ represents a list of several general $\oplus$-formulas that are generated from $A_1, \ldots, A_p$.

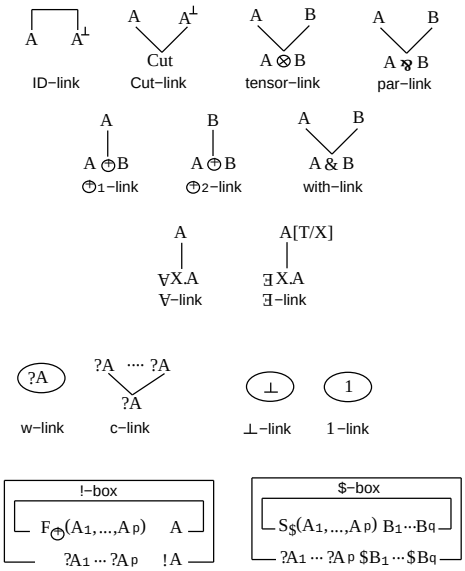

FIG. 1. the links in the LLL system

Figure 2 shows simple proof nets are defined inductively. The formulas and links in simple proof nets have weights. These weights are generated from eigenweights that are associated with &-links occurring in simple proof nets by using boolean product operator '$\cdot$'. If a formula or a link has the weight 1, then we omit the weight.

Moreover we must take care of the case of &-links. For example from two simple proof nets of Figure 3 we can construct a simple proof net with the conclusions $?A^{\perp}, ?B^{\perp}, !A \& \$A$ of Figure 4. As shown in the figure, the *context*-formulas must be shared.

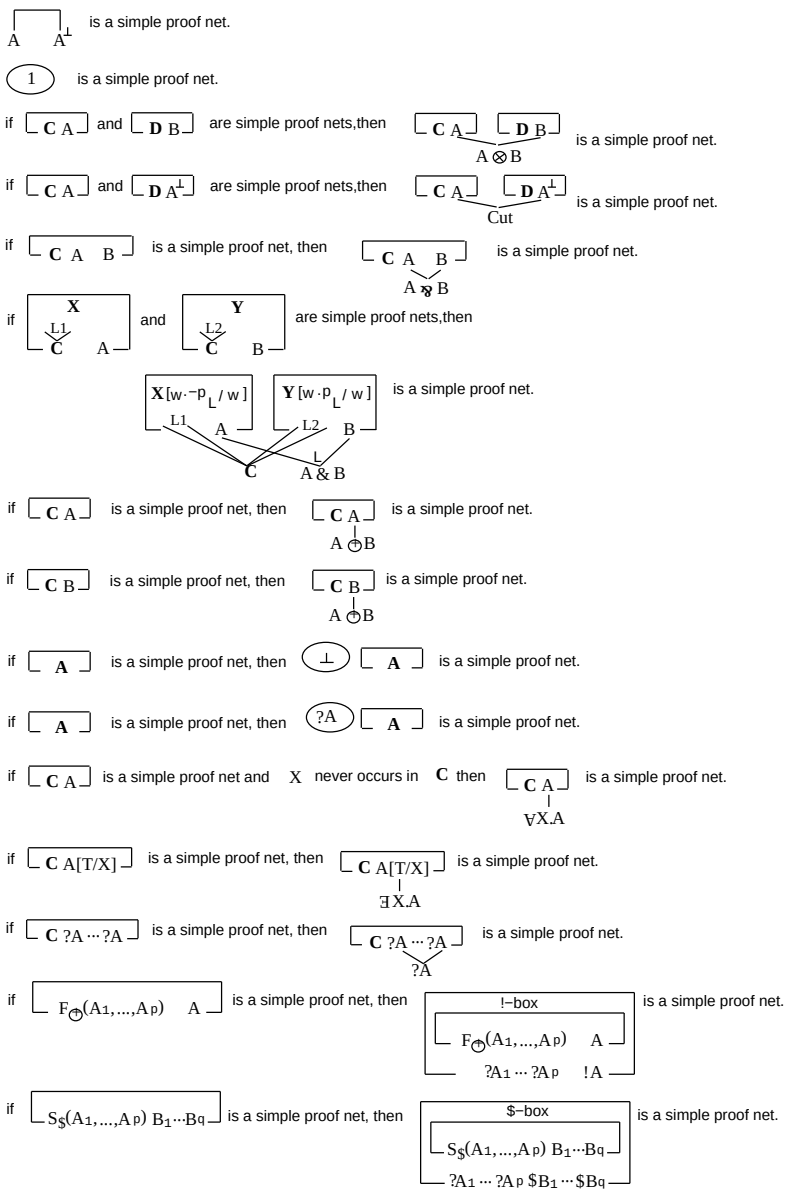

FIG. 2. the definition of simple proof nets

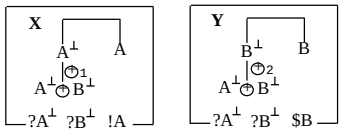

FIG. 3. two simple proof nets

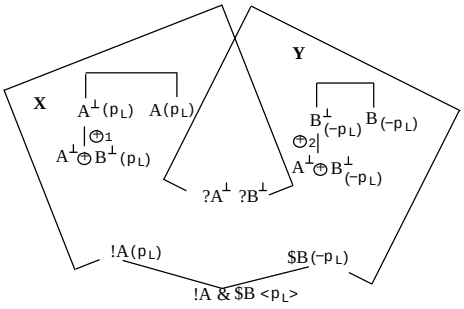

FIG. 4. An example of constructions of simple proof nets with &-links

Moreover sharing of context-formulas may be complex. For example, from two simple proof nets of Figure 5 we can construct a simple proof net with

$$!A\&\$D, ?B^{\perp}, ?C^{\perp}, ?D^{\perp}, \$C\otimes !A.$$

But it is difficult to write down this on a plane in a concise way. So we omit this.

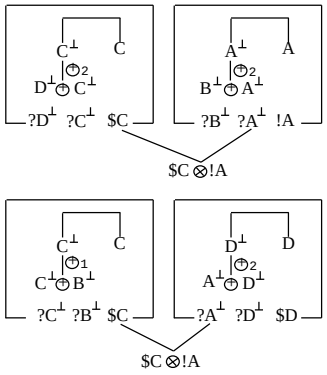

FIG. 5. more two simple proof nets

Figure 6 shows an example that is a proof net in the sense of [Gir96]. The proof net satisfies the correctness condition of [Gir96]. But it is not simple. However we can easily construct a simple proof net that has the same conclusions as the proof net. For example a simple proof net corresponding to that of Figure 6 is that of Figure 7. But such a simple proof net is not uniquely determined. For instance, Figure 8 shows another simple proof net corresponding to that of Figure 6. Besides, in the introduction rules of !-box and $-box when we replace $\oplus$-occurrences of generalized $\oplus$-formulas by comma delimiters, we can easily find that any modified simple proof net in this manner is a proof net of Girard by induction on derivations of simple proof nets.

Figure 9 shows the *rewrite rules* in the LLL system except for contraction, neutral, !-!, and !-$ rewrite rules. Fusion and c-w rewrite rules first appeared in [DK97]. The

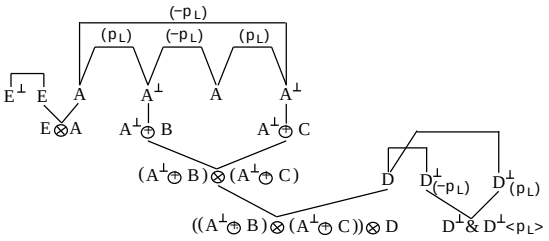

FIG. 6. An example of non-simple proof nets

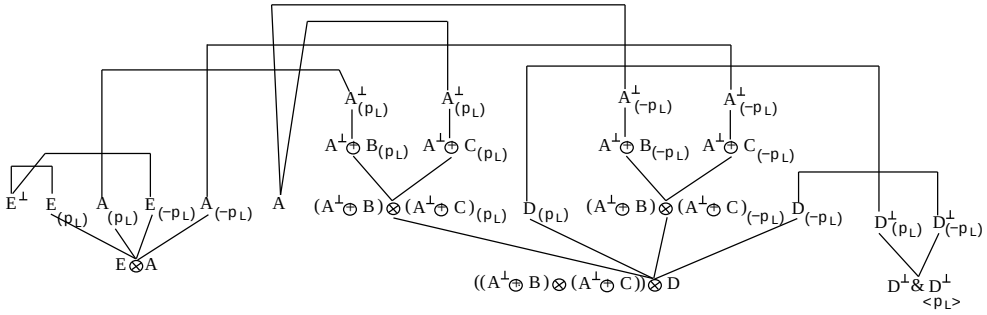

FIG. 7. A simple proof net corresponding to the above non-simple net

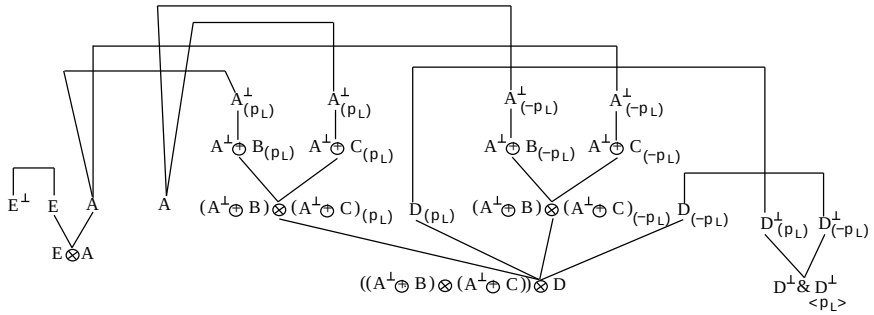

FIG. 8. Another simple proof net corresponding to the above non-simple net

other rewrite rules in Figure 9 are standard in Linear Logic. Figure 10 shows neutral rewrite rule. Figure 11 shows contraction rewrite rule, where $\$\mathbf{Y}$, $\$\mathbf{Y}'$, $!\mathbf{Z}$ and $!\mathbf{Z}'$ represent sequences of proof nets and '$\mathbf{w}$' a sequence of weakening links. The length of $\$\mathbf{Y}$ must be the same as that of $\$\mathbf{Y}'$ and the length of $!\mathbf{Z}$ the same as that of $!\mathbf{Z}'$. Let $\$\mathbf{Y} = Y_1, \ldots, Y_m$, $\$\mathbf{Y}' = Y'_1, \ldots, Y'_m$, $\$\mathbf{Z} = Z_1, \ldots, Z_n$, $\$\mathbf{Z}' = Z'_1, \ldots, Z'_n$. Each $Y_i\,(1 \leq i \leq m)$ and $Z_\ell\,(1 \leq \ell \leq n)$ must have the following conditions:

1. The conditions on Z_ℓ.
   Each Z_ℓ must have the form of the upper proof net of Figure 12 or that of Fig-

ure 13. In both proof nets, the first j arguments of $G_{\oplus}(A_1, \ldots, A_n)$ are all $A^{\perp}$ occurrences and all the links from A_i to $G_{\oplus}(A_1, \ldots, A_n)$ are $\oplus$-links that have weight 1 (therefore all the formulas from A_i to $G_{\oplus}(A_1, \ldots, A_n)$ are not conclusions of two links. We call such a sequence of $\oplus$-links $\oplus$-*chain*). In the former case A_i is equal to $A^{\perp}$ (in this case $i \leq j$) and in the latter case A_i not (in this case $i > j$). We call the former $\oplus$-chain *non-fake* and the latter *fake*.

2. The conditions on Y_i.
   Each Y_i must have the form of the upper proof net of Figure 14 or that of Figure 15. In the former case there are some non-fake chains, but in the latter case all the $\oplus$-chains are fake.

In other words each proof net in $\$\mathbf{Y}$ and $!\mathbf{Z}$ must have at least one $\oplus$-chain. Moreover each $Y_i'\,(1 \leq i \leq m)$ and $Z_\ell'\,(1 \leq \ell \leq n)$ must have the following forms according to Y_i and Z_ℓ:

1. The case where the $PLUS$-chain of Z_ℓ is non-fake:
   Then Z_ℓ' must be the lower proof net of Figure 12.
2. The case where the $\oplus$-chain of Z_ℓ is fake:
   Then Z_ℓ' must be the lower proof net of Figure 13.
3. The case where some $\oplus$-chains of Y_i are non-fake:
   Then Y_i' must be the lower proof net of Figure 14. The notation $?\hat{B}_{r_j}$ of the right side means that the weakening link with conclusion $?B_{r_j}$ is missing in the proof net.
4. The case where all the $\oplus$-chains of Y_i are fake:
   Then Y_i' must be the lower proof net of Figure 15.

Note that neither the left hand side nor the right hand side of Figure 11 is a simple proof net. If we find a pattern of the left hand side of Figure 11 in a simple proof net, we can apply the contraction rule to the simple proof net and replace the pattern by an appropriate instantiation of the right hand side of Figure 11.

Let us recall lazy cut elimination in [Gir96].

Definition 2.1
Let L be a Cut-link in an additive proof net. When two premises of L are A and $A^{\perp}$, L is *ready* if

1. L has the weight 1;
2. Both A and $A^{\perp}$ are the conclusion of exactly one link.

For example, in Figure 16, the right cut is ready, but the left not. After the right cut is rewritten, the left become ready.

Lazy cut elimination is a reduction procedure in which only ready cuts are redexes (of course, in the contraction rewrite rule the above mentioned conditions must be satisfied). The definition also applies to our rewrite rules. So we use the definition. By $\rightarrow_{\text{lazy}}$ we denote one step reduction of lazy cut elimination.

Theorem 2.2
Let Θ_1 be a simple proof net. If $\Theta_1 \rightarrow_{\text{lazy}} \Theta_2$, then Θ_2 is also a simple proof net.

Proof. Induction on the construction of simple proof net Θ_1 and an easy argument on permutations of links. ∎

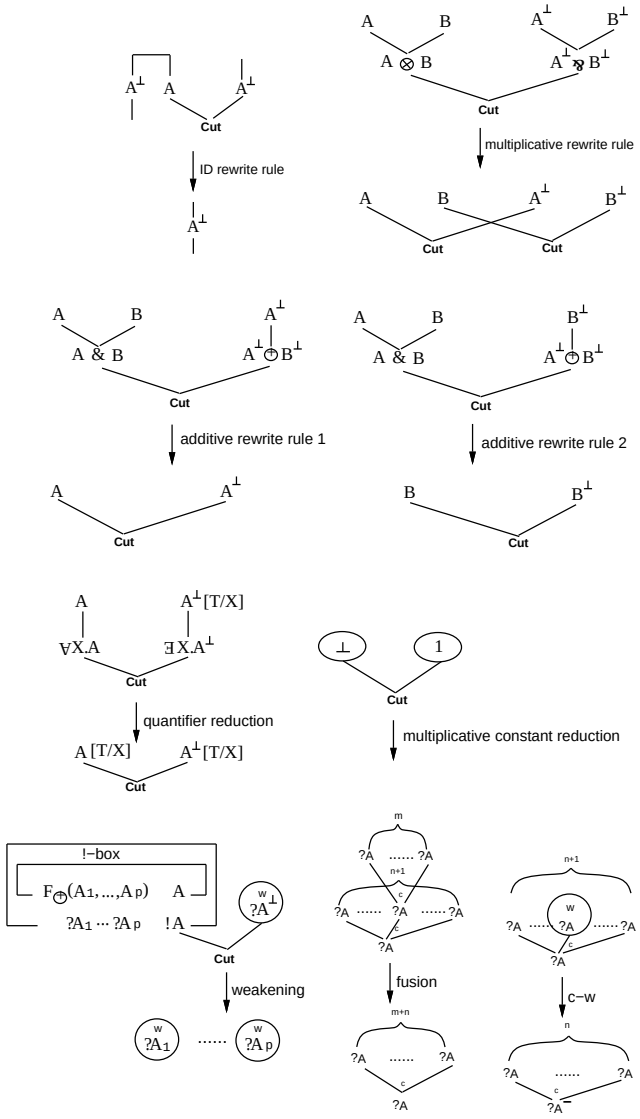

FIG. 9. the rewrite rules in the LLL system

Next, we relate lazy cut elimination of simple proof nets with that of Girard's proof nets.

PROPOSITION 2.3
One step of lazy cut elimination of simple proof nets can be simulated by several steps of that of Girard's proof nets.

We do not present the proof because in order to prove this we must rephrase the full details of Girard's proof nets. We just show the difference between them. The left cut of Figure 17 is a redex of Girard's lazy cut elimination, but not of that of simple

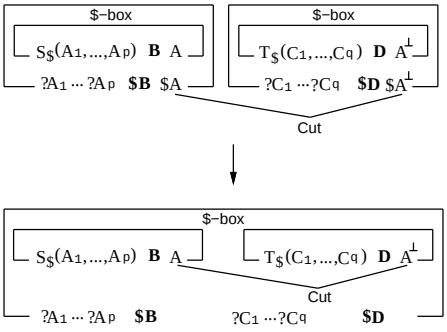

FIG. 10. neutral rewrite rule

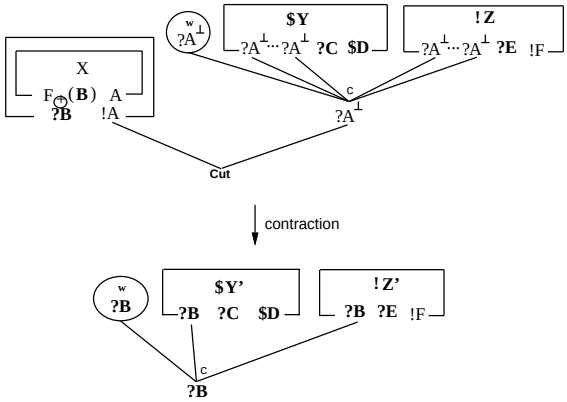

FIG. 11. contraction rewrite rule

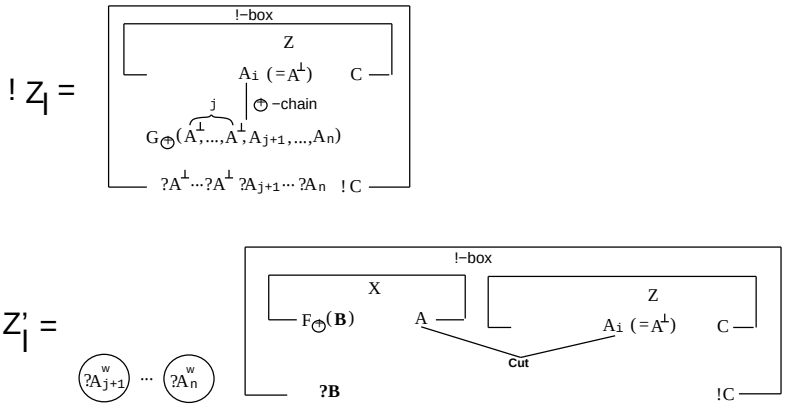

FIG. 12. The case where Z_ℓ has the non-fake $\oplus$-chain

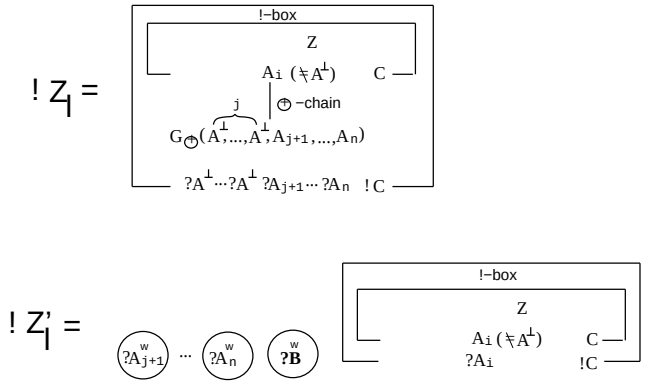

FIG. 13. The case where Z_ℓ has the fake $\oplus$-chain

proof nets. In Girard's lazy cut elimination, Figure 17 can be reduced to Figure 18. That does not happen to simple proof nets. Instead of that, in lazy cut elimination of simple proof nets the right cut of Figure 17 is ready and Figure 17 can be reduced to Figure 19. Then the residual left cut of Figure 19 become ready. In lazy cut elimination of simple proof nets, Figure 19 is reduced to Figure 20 by *one-step*. But in Girard's lazy cut elimination this reduction takes *two-steps*. For example we need an intermediate proof net like Figure 21.

It is obvious that there is a proof net that is reduced to a cut-free form in Girard's lazy cut elimination, but not in lazy cut elimination of simple proof nets. Hence, in this sense, our lazy cut elimination is weaker than that of Girard's proof nets. But, when we execute Theorem 3.2, that is, compute polynomial bounded functions on binary integers in proof nets, our lazy cut eliminations and Girard's always return the same result, since this is due to the following Girard's theorem and our binary integer encoding in simple proof nets does not have any &-occurrences.

THEOREM 2.4 ([Gir96])
Let Θ be a proof-net whose conclusions do not contain the connective & and $\exists X.$ and without ready cut; then Θ is cut-free.

3 A Turing Machine Encoding

Let M be a Turing machine and k be the number of the states of M. Without loss of generality, we can assume that only 0, 1, and $*$ occur in the tape of M, where $*$ is the blank symbol of M.

We use

$$\mathbf{bool^k} \equiv_{\mathrm{def}} \forall X.\overbrace{X\&(\cdots\&(X\&X)\cdots)}^{k} \multimap X$$

for the type of the states of M. In contrast to $\mathbf{bool^k}$ in [Gir98], $\mathbf{bool^k}$ in this paper does not include the neutral connective \$. Figure 22 shows an example of $\mathbf{bool^k}$ proofs. After 0 or 1 $\oplus_1$-link, $\oplus_2$-links follow $k-1$ or $i-1$ times.

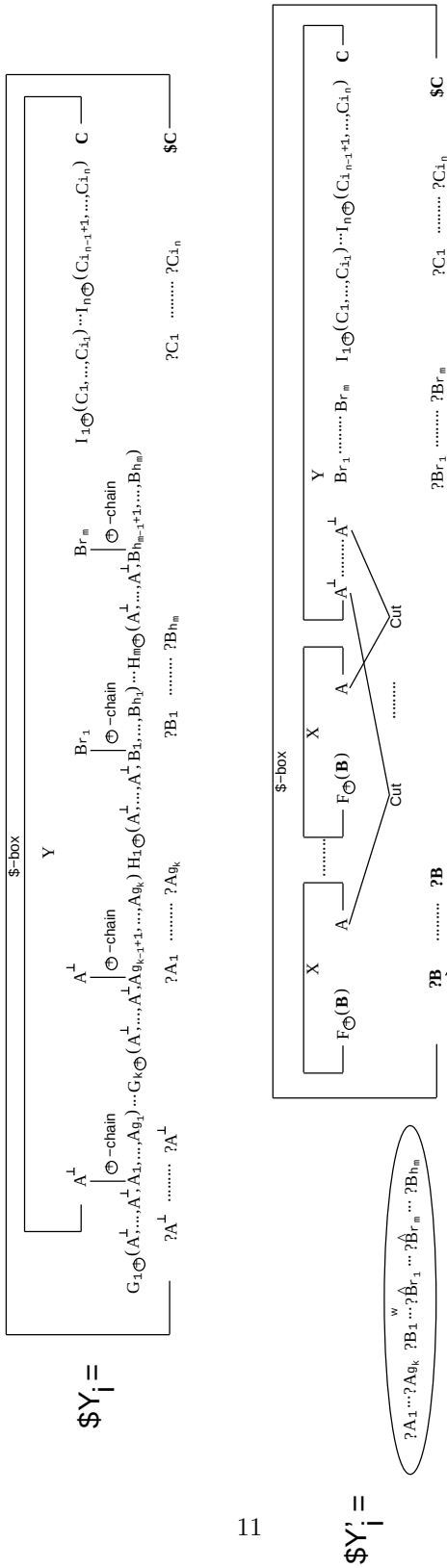

FIG. 14. The case where Y_i has non-fake $\oplus$-chains

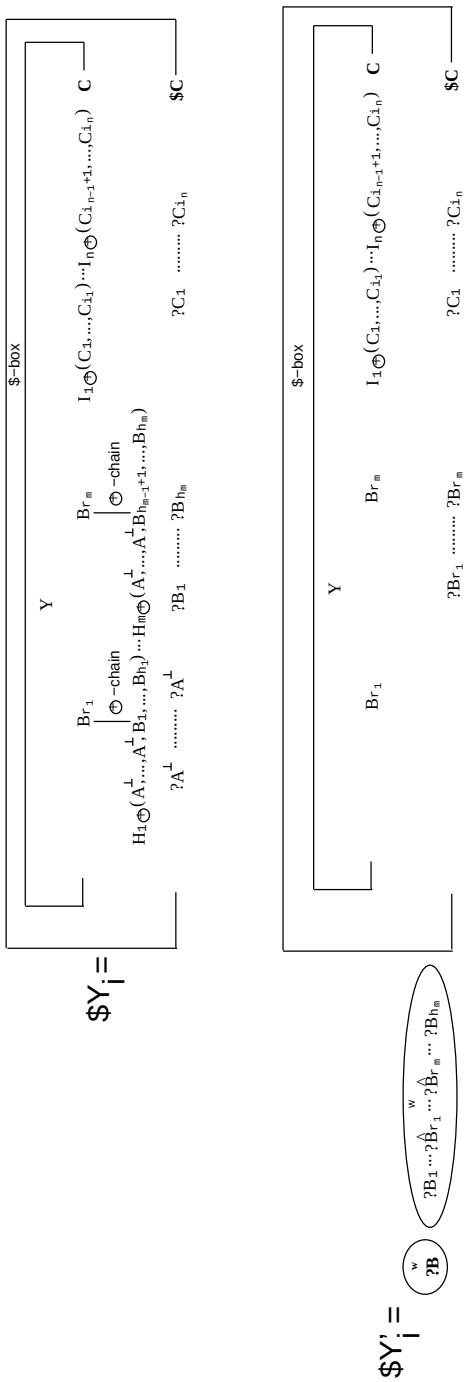

FIG. 15. The case where Y_i does not have any non-fake $\oplus$-chains

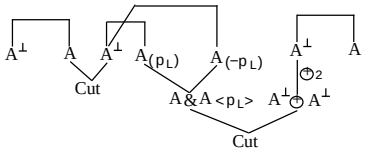

FIG. 16. An example of ready cuts and non-ready cuts

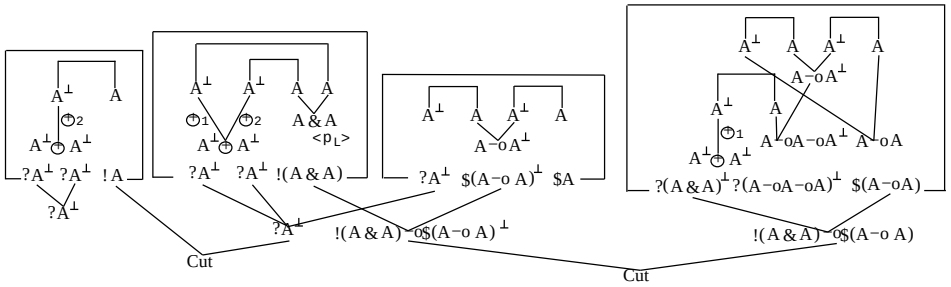

FIG. 17: an example that is a redex of Girard' proof nets but not that of simple proof nets

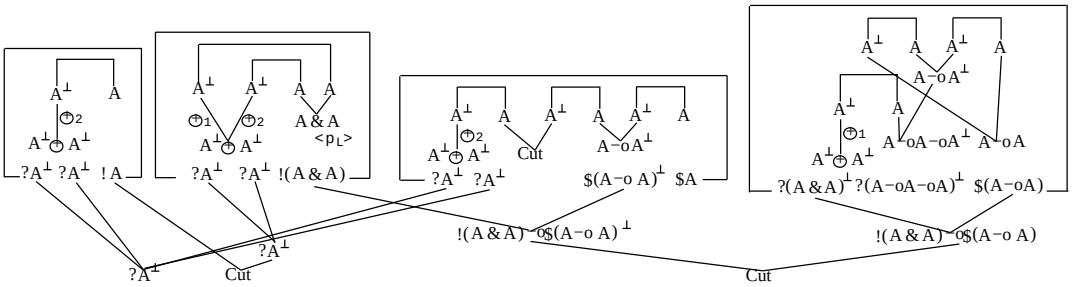

FIG. 18. a contractum of Girard's proof nets

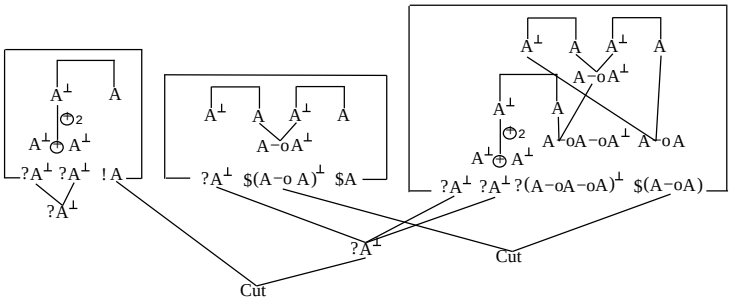

FIG. 19. an example that is a redex of simple proof nets

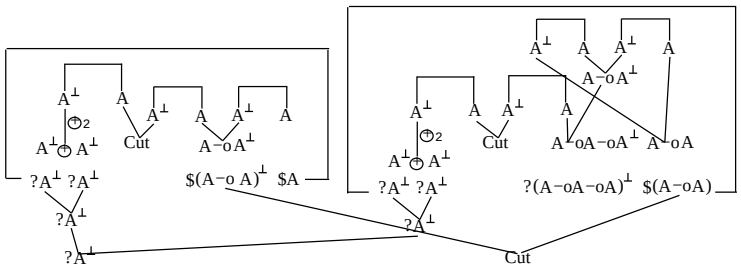

FIG. 20. a contractum of Figure 19

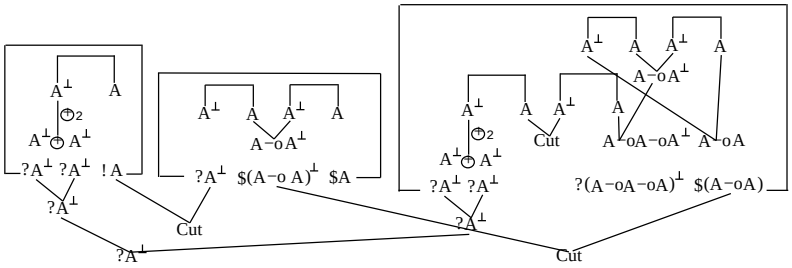

FIG. 21. an intermediate proof net

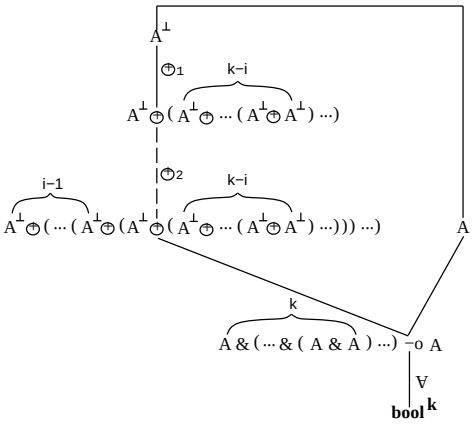

FIG. 22. an example of $\mathbf{bool^k}$ proofs

In addition we use

$$\mathbf{config} \equiv_{\mathrm{def}} \forall X.!(X \multimap X) \multimap !(X \multimap X) \multimap !(X \multimap X) \multimap \$(X \multimap X \multimap (X \otimes X) \otimes \mathbf{bool^k})$$

for the type of configurations of M. The type represents the current configuration of running M, that is, the 3-tuple of the left part of the current tape, the right part, and the current state. Figure 23 shows an example of **config** proofs. In the λ-notation, the example is $\lambda f_0.\lambda f_1.\lambda f_*.\lambda x.\lambda y.\langle f_0(f_1(x)), f_*(f_1(f_0(y))), b\rangle$, where b is a $\mathbf{bool^k}$-value. Hence the example denotes configuration $\langle 10, *10, b\rangle$.

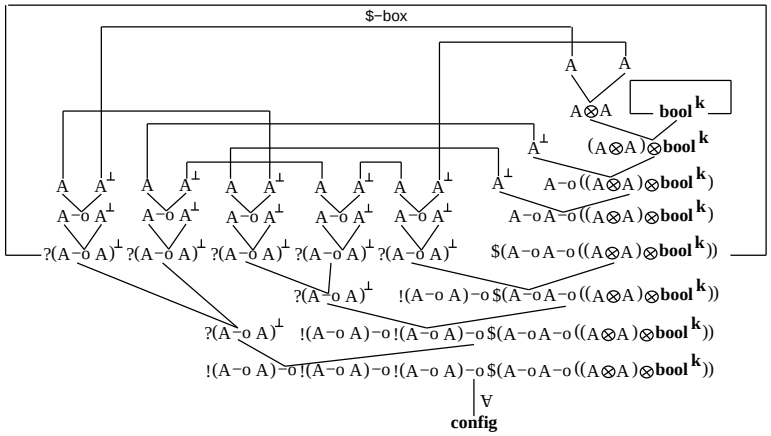

FIG. 23. an example of **config** proofs

For shorthand, we use $\mathbf{id}_A$ to represent $A \multimap A$. Then we write down the transition function of M in Light Linear Logic, which is the main task of the paper. Figure 24 shows our encoding of the transition function, where $\mathbf{trpl}_A$ is an abbreviation of $(\mathbf{bool^4}\&1) \otimes ((\mathbf{id}_A\&1) \otimes A)$. The formula $\mathbf{trpl}_A$ is fed to the second-order variable that is bound by the $\forall$-link in an input proof of **config**.

Three proof nets $\mathtt{apply\_step}(0^{\mathbf{bool^4}})$, $\mathtt{apply\_step}(1^{\mathbf{bool^4}})$, and $\mathtt{apply\_step}(*^{\mathbf{bool^4}})$ in Figure 24 are made up by giving a $\mathbf{bool^4}$ proof (0, 1, or $*$) to step of Figure 25 (see Figure 26), where $0^{\mathbf{bool^4}}$, $1^{\mathbf{bool^4}}$, and $*^{\mathbf{bool^4}}$ are different normal proof net of $\mathbf{bool^4}$. $0^{\mathbf{bool^4}}$, $1^{\mathbf{bool^4}}$, and $*^{\mathbf{bool^4}}$ represent the symbols 0, 1, and $*$ on the tape of M. The main purpose of these $\mathtt{apply\_step}(\Theta^{\mathbf{bool^4}})$ is to decompose the left or right part of the tape of a given configuration into data with type $\mathbf{trpl}_A = (\mathbf{bool^4}\&1)\otimes((\mathbf{id}_A\&1)\otimes A)$, where both $\mathbf{bool^4}\&1$ and $\mathbf{id}_A\&1$ represent the top symbol of the left or right part of the tape and A represents the rest except for the top symbol. The principle by which the encoding works is the same as that used in writing down the predecessor function. There is just one proof net of $\mathbf{bool^4}$ that are different from these three. Let the proof net be $\mathtt{empty}^{\mathbf{bool^4}}$. The proof net $\mathtt{empty}^{\mathbf{bool^4}}$ do not have any corresponding symbol on the tape of M: the proof net is used in `apply_base` of Figure 28 in order to make our encoding easy.

The proof net `main`\$ in the \$-box in Figure 24 is shown in Figure 27.

Proof net `apply_base` in Figure 27 are made up by giving a $\mathbf{bool^4}$ proof `empty` to base of Figure 28, where as we mentioned before, $\mathtt{empty}^{\mathbf{bool^4}}$ is a normal proof net of

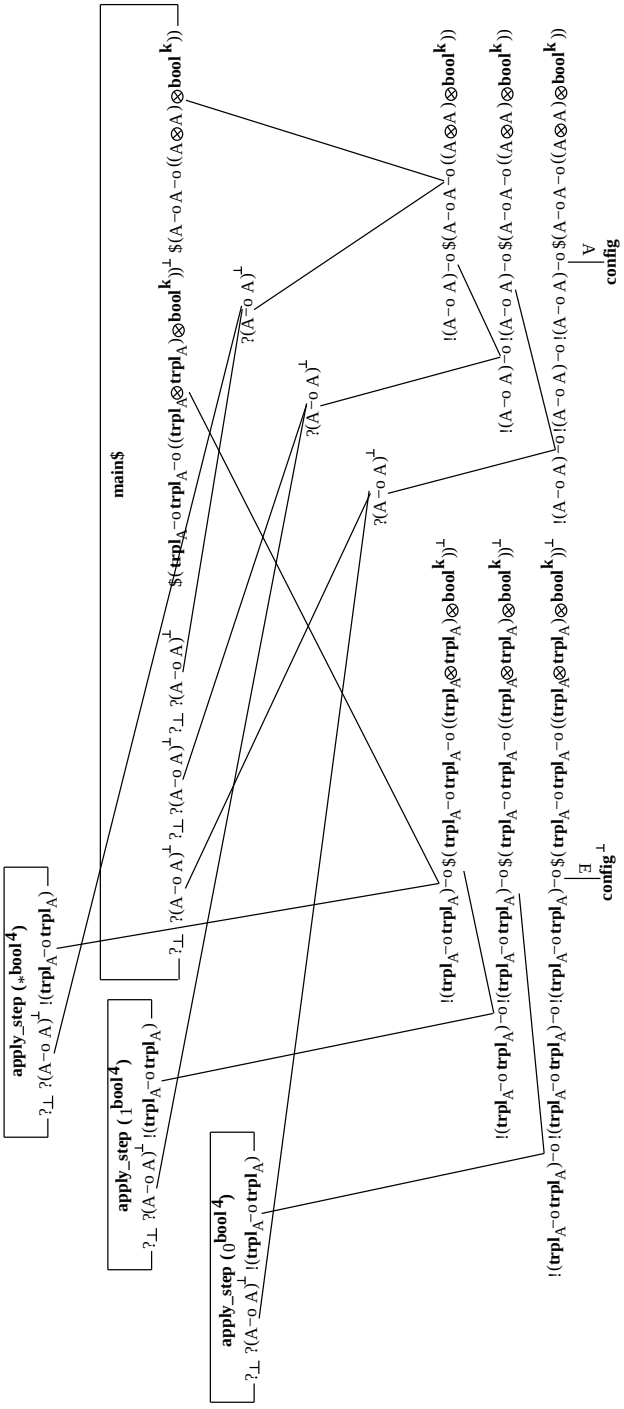

FIG. 24. `transition` function

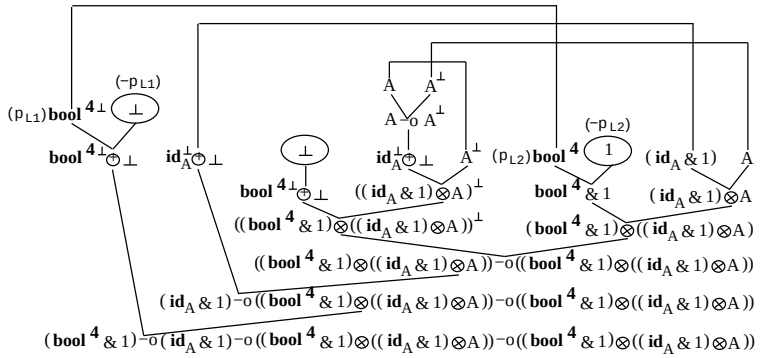

FIG. 25. `step` function

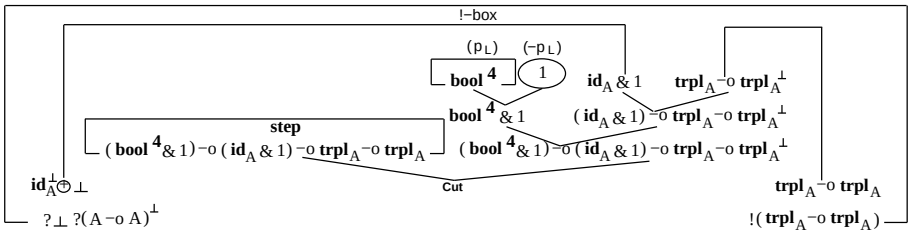

FIG. 26. `apply_step`

$\mathbf{bool^4}$, which is different from $0^{\mathbf{bool^4}}$, $1^{\mathbf{bool^4}}$, and $*^{\mathbf{bool^4}}$. The proof net `apply_base` is used in order to feed an initial value to $\texttt{apply\_step}(\Theta^{\mathbf{bool^4}})$.

Proof net `extract` in Figure 27 is shown in Figure 29. The intention of `extract` was to transform an input of the net $\langle\langle b_1, \langle f_1, a_1\rangle\rangle, \langle\langle b_2, \langle f_2, a_2\rangle\rangle, bk\rangle\rangle$ with type $(\mathbf{bool^4}\&1)\otimes((\mathbf{id}_A\&1)\otimes A)\otimes((\mathbf{bool^4}\&1)\otimes((\mathbf{id}_A\&1)\otimes A)\otimes\mathbf{bool^k})$ into $\langle\langle\langle\langle b_2, f_1\rangle, a_1\rangle, a_2\rangle, bk\rangle$ with type $(((\mathbf{bool^4}\otimes\mathbf{id}_A)\otimes A)\otimes A)\otimes\mathbf{bool^k}$. The top symbol of the left part of the current tape must be left with type $\mathbf{id}_A$ since this is used in order to be attached to the left or right part of the tape of the next configuration. The top symbol of the right part of the current tape, at which the head of M currently points, must be left with type $\mathbf{bool^4}$ since this is used in order to choose one of select functions (which are defined later). Note that to do this one must use additive connectives and multiplicative constants.

Proof net `comp` in Figure 27 is shown in Figure 30, where

$$\mathbf{shift} \equiv_{\text{def}} \forall X.(X \multimap X) \multimap X \multimap X \multimap((X \otimes X) \otimes \mathbf{bool^k})$$

and

$$\mathbf{row} \equiv_{\text{def}} \mathbf{shift}\&(\mathbf{shift}\&(\mathbf{shift}\&\mathbf{shift})).$$

The main purpose of `extract` is to transform an input of the net $\langle\langle\langle\langle b, f\rangle, a_1\rangle, a_2\rangle, bk\rangle$ with type $(((\mathbf{bool^4}\otimes\mathbf{id}_A)\otimes A)\otimes A)\otimes\mathbf{bool^k}$ into the next configuration $\langle\langle a_3, a_4\rangle, bk'\rangle$

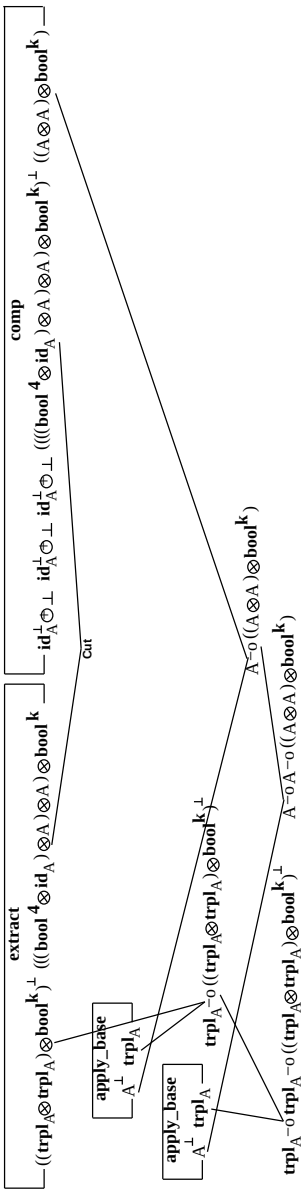
extract
((trpl_A⊗trpl_A)⊗bool^k)^⊥ (((bool^4⊗id_A)⊗A)⊗A)⊗bool^k
comp
id^⊥_A⊕⊥ id^⊥_A⊕⊥ id^⊥_A⊕⊥ ((((bool^4⊗id_A)⊗A)⊗A)⊗bool^k)^⊥ ((A⊗A)⊗bool^k)
cut
apply_base
A^⊥ trpl_A
apply_base
A^⊥ trpl_A
trpl_A⊸((trpl_A⊗trpl_A)⊗bool^k)^⊥
A⊸((A⊗A)⊗bool^k)
trpl_A⊸trpl_A⊸((trpl_A⊗trpl_A)⊗bool^k)^⊥
A⊸A⊸((A⊗A)⊗bool^k)

FIG. 27. `main$`

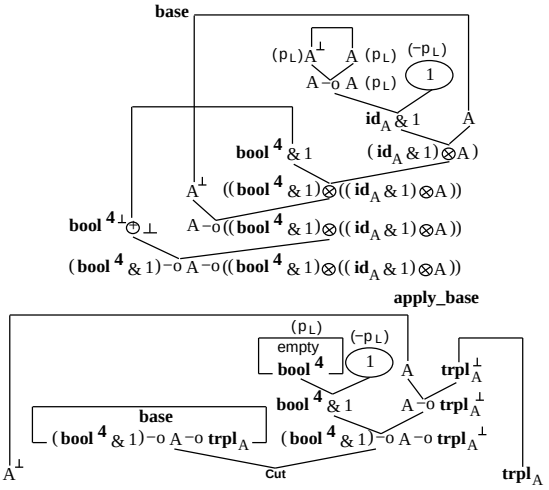

FIG. 28. `base` function and `apply_base`

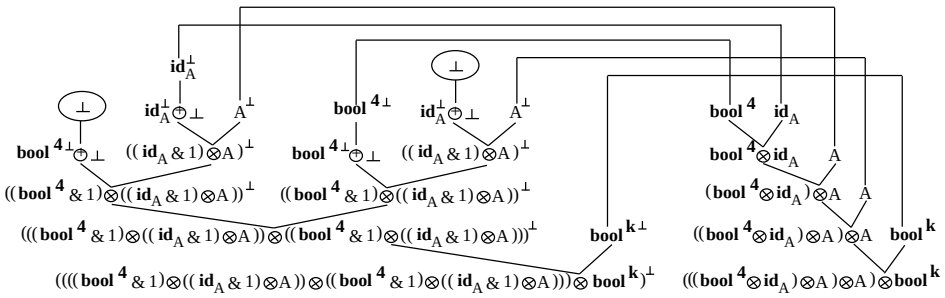

FIG. 29. `extract` function

with type $(A \otimes A) \otimes \mathbf{bool^k}$. Data b with type $\mathbf{bool^4}$ and bk with type $\mathbf{bool^k}$ in the input are used in order to choose one of select functions (which are defined later).

Proof net `matrix` in Figure 30 is shown in Figure 33, where $r_1, \ldots, r_{k-1}, r_k$ are proof nets that have the form of Figure 34. The main purpose of `matrix` is to retain k proof nets of the form of Figure 34. Proof nets s_1, s_2, s_3, and s_4 in Figure 34 have the form of Figure 31 or Figure 32. We call such proof nets *shift functions*. Proof nets that have the form of Figure 31 represent left moves of the head of M. On the other hand proof nets that have the form of Figure 32 represent right moves of the head of M.

From what precedes it is obvious that we can encode the transition function of M into a proof net with conclusions $?\bot, \mathbf{config}^\bot, \mathbf{config}$ of Light Linear Logic. By using the proof net, as shown in Appendix A, **P**-time Turing machines can be encoded. In other terms, we obtain the following theorem:

THEOREM 3.1
Let **bint** be $\forall X.!(X \multimap X) \multimap !(X \multimap X) \multimap \$(X \multimap X)$. Let M be a Turing machine

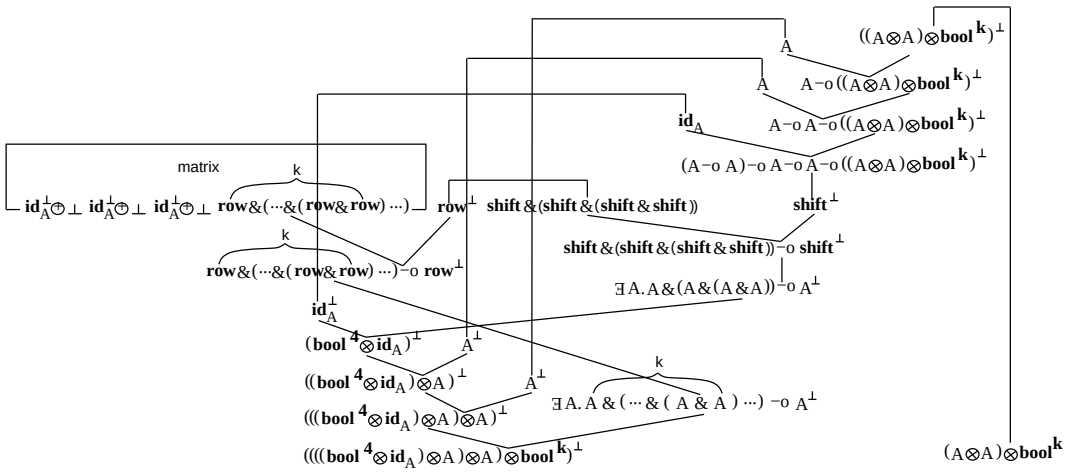

FIG. 30. `comp`

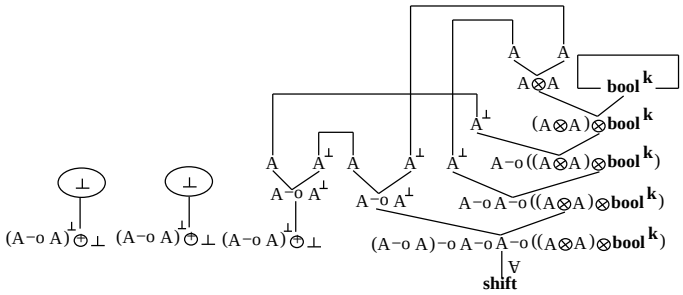

FIG. 31. an example of **shft** proofs (left move)

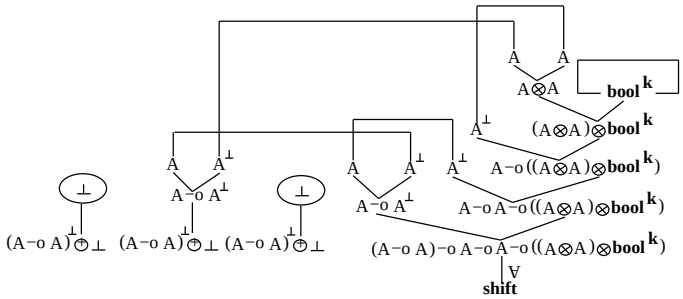

FIG. 32. an example of **shft** proofs (right move)

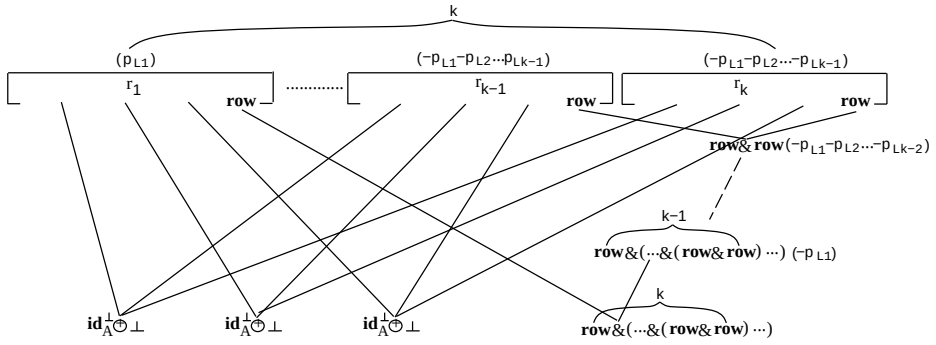

FIG. 33. `matrix`

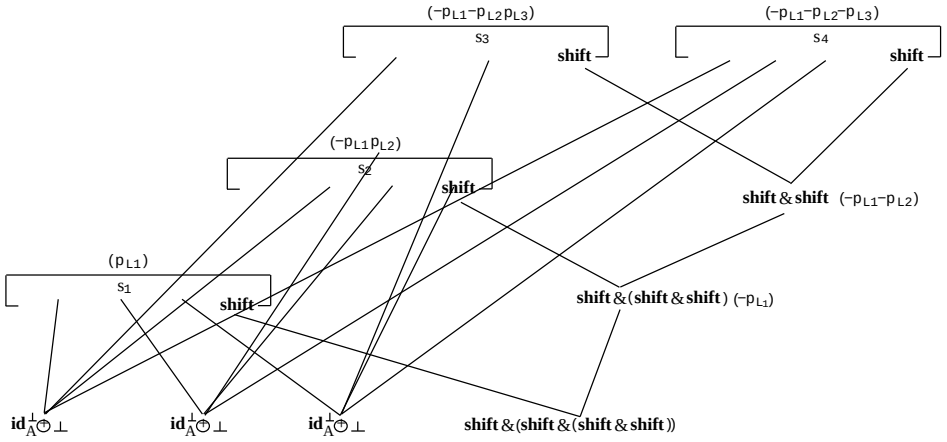

FIG. 34. `row_net`

with time bound of a polynomial with degree k. In Light Linear Logic M can be represented by a proof net with conclusions $\perp^{k+4}, \mathbf{bint}^{\perp}, \$^{k+3}\mathbf{config}$.

Furthermore, we can strengthen the above theorem as follows:

THEOREM 3.2
Let f be a polynomial-time function with degree k. In Light Linear Logic f can be represented by a proof net with conclusions $\perp^{k+6}, \mathbf{bint}^{\perp}, \$^{k+5}\mathbf{bint}$.

In order to prove the theorem, we need a proof net that transforms **config** into **bint**. In Appendix B, we show a proof net that performs the translation.

4 Our Nondeterministic Extension of the Light Linear Logic System

In this section we consider a nondeterministic extension of the LLL system called the *NDLLL* system. In this extended system we introduce a new self-dual additive

connective "nondeterministic with" ▲. Then the formulas of NDLLL are constructed by adding the following clause to that of LLL:

$$F = ...|F \blacktriangle F$$

Since ▲ is a self-dual connective, the negation of the formula $A \blacktriangle B$ is defined as follows:

- $(A \blacktriangle B)^{\perp} \equiv_{\text{def}} A^{\perp} \blacktriangle B^{\perp}$

The link newly introduced in NDLLL is the form of Figure 35. Proof nets for the NDLLL system are inductively defined from the rules of Figure 2 and in the middle of Figure 35. In a simple proof net for NDLLL a unique eigenweight is assigned to each ▲-link occurrence in the same manner as that of &-link.
Finally the rewrite rules of NDLLL are that of LLL plus the nondeterministic rewrite rule of Figure 35. In the rewrite rule for ▲ any of the two contractums is nondeterministically selected. If the left contractum (resp. the right contractum) of Figure 35 is selected, then all the occurrences of both eigenweights for $A \blacktriangle B$ and $A^{\perp} \blacktriangle B^{\perp}$ are assigned to 1 (resp. 0). In the next section we explain a usage for the ▲.

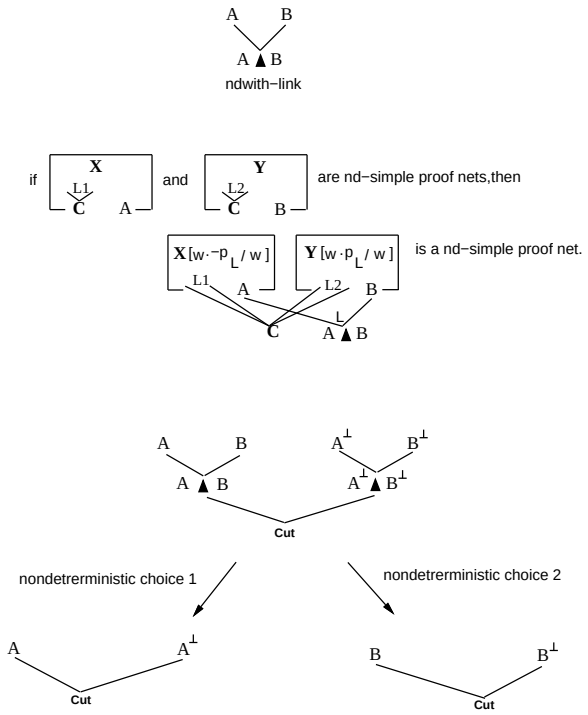

FIG. 35. Our nondeterministic extension of the LLL system

4.1 A Nondeterministic Turing Machine Encoding

Our usage of ▲-connective is to use ▲ in proof nets on datatypes like **bool** $\equiv_{\mathrm{def}}$ $\forall X.X\&X \multimap X$ which use the standard additive connectives & and $\oplus$. As an example, we consider cut-elimination of Figure 36, where note that the sub-proof net with conclusion **bool** $\multimap$ **bool** of the right premise of Cut is constructed by using ▲. From Figure 36 to Figure 40 the standard lazy cut elimination procedure is performed. In Figure 40 the nondeterministic cut elimination procedure defined in previous section is performed. Figure 41 is one choice and Figure 42 the other choice.

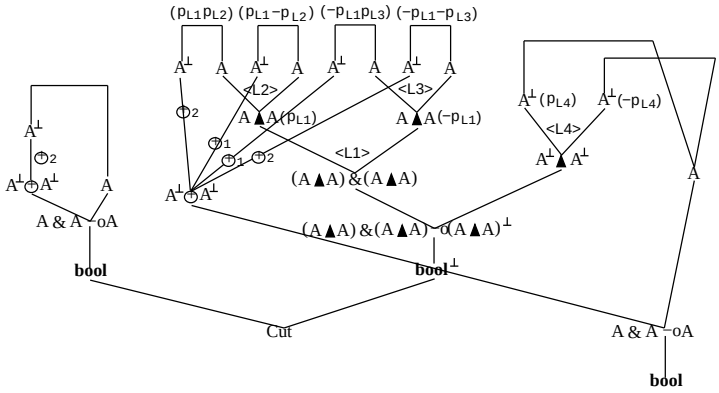

FIG. 36. An example: the starting point

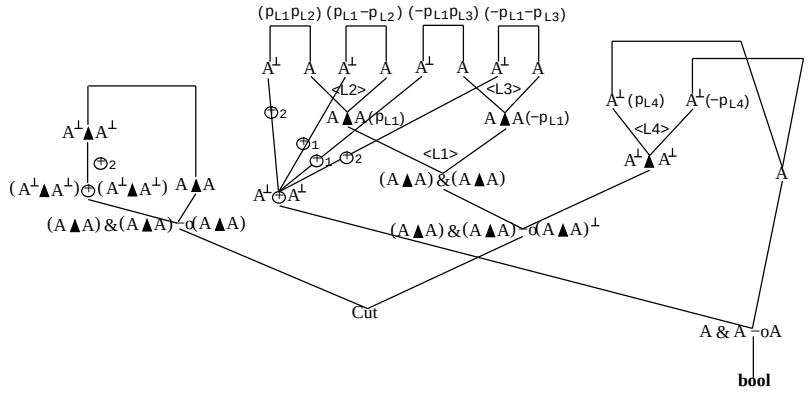

FIG. 37. An example: step 1

An encoding of a nondeterministic Turing machine into NDLLL uses the same idea. The encoding is the same as that of a deterministic Turing machine into LLL except for `comp` of Figure 30. The `comp` proof net is replaced by the `nd-comp` of Figure 43, where

ndrow $\equiv_{\mathrm{def}}$ (**shift**▲**shift**)&((**shift**▲**shift**)&((**shift**▲**shift**)&(**shift**▲**shift**))).

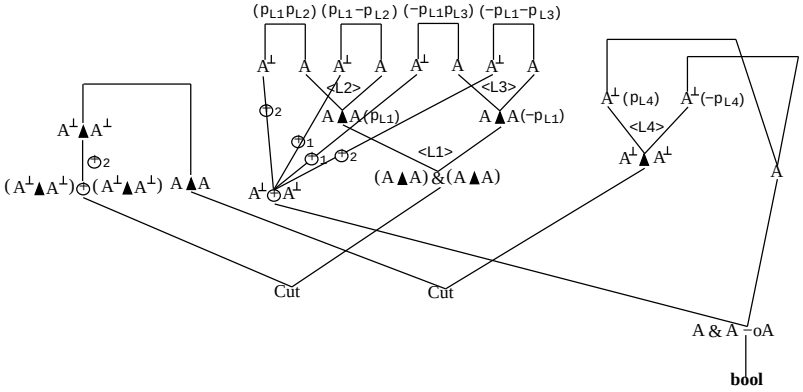

FIG. 38. An example: step 2

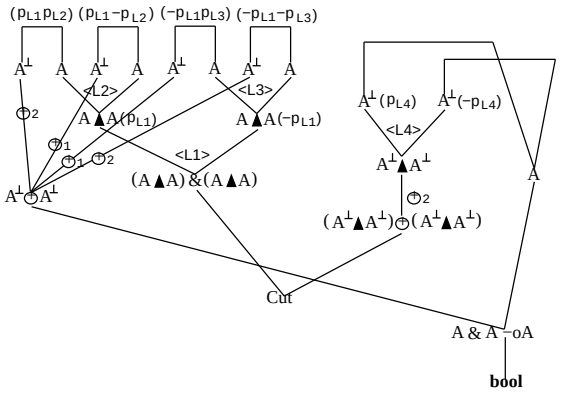

FIG. 39. An example: step 3

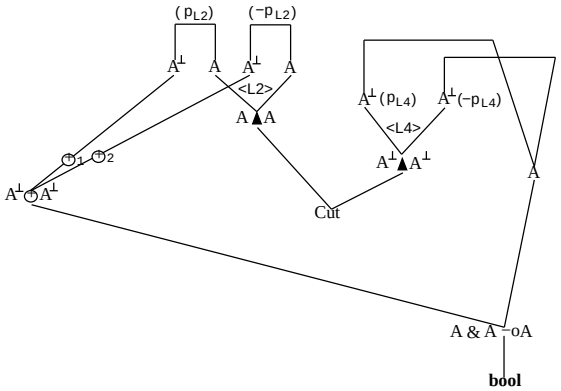

FIG. 40. An example: step 4

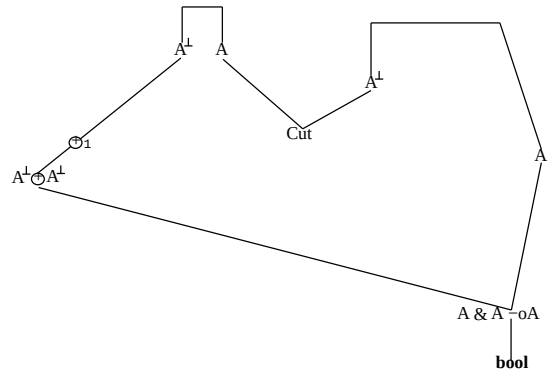

FIG. 41. An example: nondeterministic choice 1

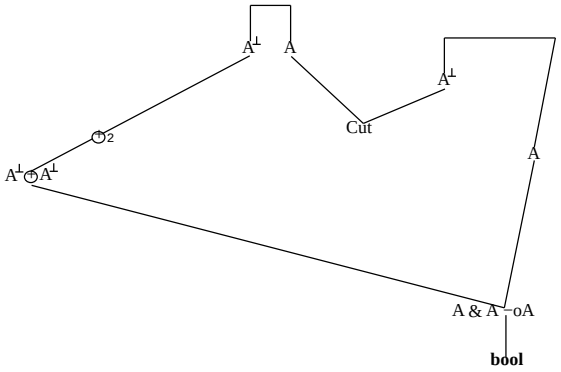

FIG. 42. An example: nondeterministic choice 2

The idea is completely the same as that of the above example. The information about nondeterministic transitions of a nondeterministic Turing machine is stored in a proof of

$$\overbrace{\mathbf{ndrow}\&(\cdots\&(\mathbf{ndrow}\&\mathbf{ndrow})\cdots)}^{k}.$$

By the completely same manner as Theorem 3.1 except for `nd-comp` the following theorem holds.

THEOREM 4.1
Let M be a nondeterministic Turing machine with time bound of a polynomial with degree k. In Nondeterministic Light Linear Logic M can be represented by a proof net with conclusions $\bot^{k+4}, \mathbf{bint}^{\bot}, \$^{k+3}\mathbf{config}$.

Usually a nondeterministic Turing machine characterizes a language accepted by the machine. Without loss of generality, we can assume that nondeterministic Turing machine has two special state symbols `yes` and `no` which judge whether a word is accepted by the machine. Moreover we can prove the following theorem.

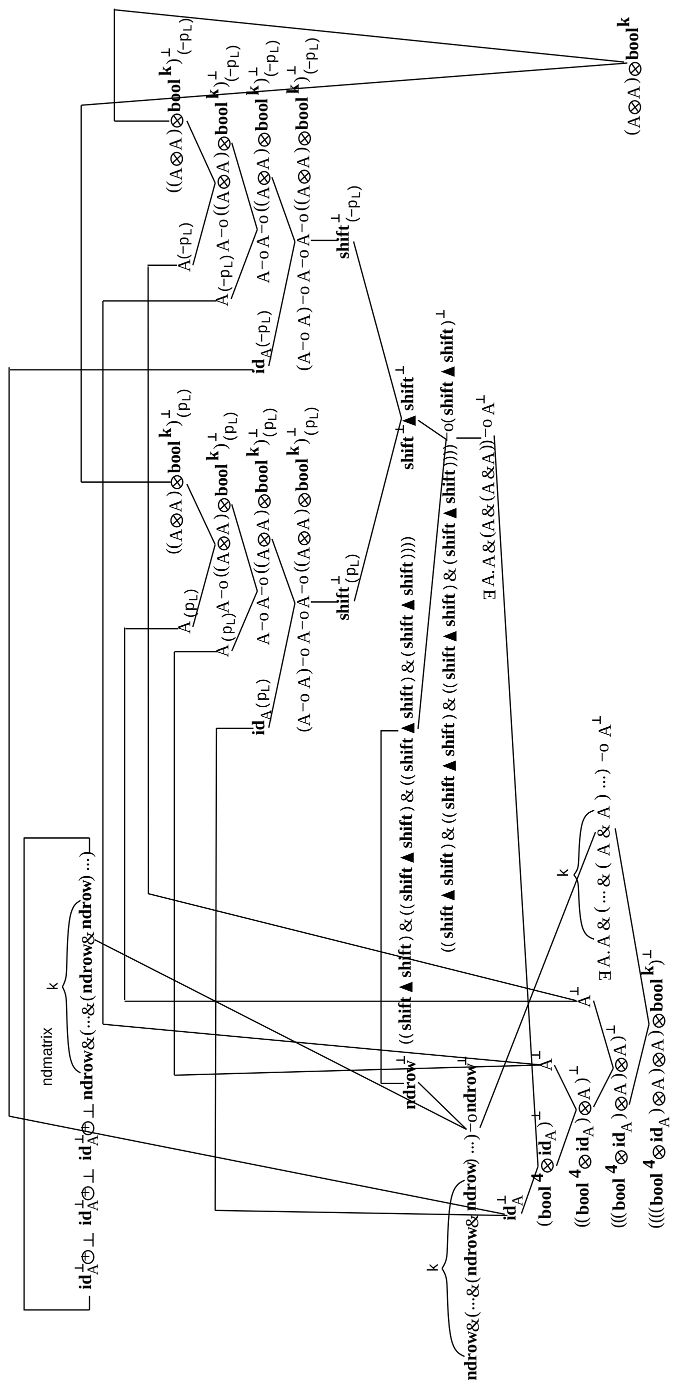

FIG. 43. nd-comp

THEOREM 4.2
Let $L \subseteq \{0,1\}^*$ be a language whose is accepted by a nondeterministic polynomial-time Turing machine M with degree k. In Nondeterministic Light Linear Logic the characterization function of L from $\{0,1\}$ to $\{0,1\}$ can be represented by a proof net with conclusions $\bot^{k+5}, \mathbf{bint}^{\bot}, \$^{k+4}\mathbf{bool}$.

In order to obtain such a proof net from the proof net with conclusions $\bot^{k+4}, \mathbf{bint}^{\bot}, \$^{k+3}\mathbf{config}$ constructed from Theorem 4.1, at first we construct a proof net which extracts a $\mathbf{bool^k}$ proof from a **config** proof. Figure 44 shows the proof net. Next we construct a proof net which maps a $\mathbf{bool^k}$ proof to a **bool** proof. The specification of proof net is that

1. if a given $\mathbf{bool^k}$ proof net represents `yes`, then the return value is a **bool** proof net that represents `yes`;
2. otherwise, the return value is a **bool** proof net that represents `no`.

We can easily construct such a proof net.

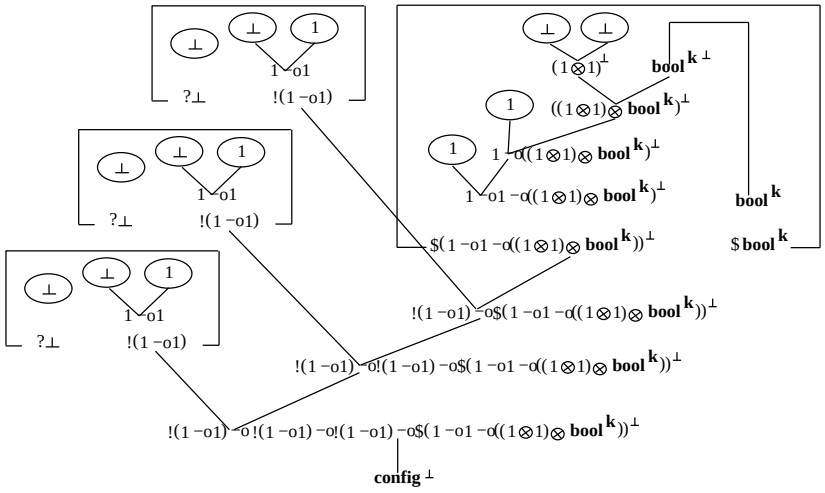

FIG. 44. `config2boolk`

4.2 Time Bound of Nondeterministic Light Linear Logic

Next we discuss the **P**-time bound of lazy cut elimination. We define the *size* of a link to be the number of the conclusions of the link. Moreover we define the *size* of a nd-simple proof net Θ (denoted by size(Θ) to be the sum of the sizes of the links in Θ. The *depth* of Θ (denoted by depth(Θ)) is defined to be the maxmal nesting number of the boxes (!-boxes or $-boxes) in Θ. As discussed in [Gir98], size(Θ) is quadratic w.r.t the encoded data generated from Θ. When $\Theta_1 \rightarrow_{\mathrm{lazy}} \Theta_2$ by nondeterministic choice, it is obvious that the size of Θ_2 is strictly less than that of Θ_1. We suppose that $\Theta \rightarrow^*_{\mathrm{lazy}} \Theta'$, where $\rightarrow^*_{\mathrm{lazy}}$ is the reflexive transitive closure of $\rightarrow_{\mathrm{lazy}}$. From the above observation and the discussion in [Gir98] on the LLL system, the following proposition is obvious.

PROPOSITION 4.3
In the NDLLL system, if $\Theta \rightarrow^*_{\mathrm{lazy}} \Theta'$, then the size of Θ' is bounded by $\mathrm{size}(\Theta)^{2^{\mathrm{depth}(\Theta)}}$.

It is easy to see that on the above proposition we can lazily reduce Θ to Θ' in a polynomial time w.r.t $\mathrm{size}(\Theta)^{2^{\mathrm{depth}(\Theta)}}$, because $\mathrm{size}(\Theta)$ is quadratic w.r.t the encoded data generated from Θ.

PROPOSITION 4.4
In the NDLLL system, if $\Theta \rightarrow^*_{\mathrm{lazy}} \Theta'$, then Θ is reduced to Θ' in a polynomial time w.r.t $\mathrm{size}(\Theta)^{2^{\mathrm{depth}(\Theta)}}$.

Let M be a nondeterministic polynomial-time Turing machine with degree k. Then by Theorem 4.1 we can construct a nd-simple proof net Θ_1 with conclusions $\bot^{k+4}, \mathbf{bint}^{\bot}, \$^{k+3}\mathbf{config}$. Then let Θ_2 be a simple proof net with the conclusion **bint** representing a binary integer with length n. Then from Proposition 4.4 we can see the proof net constructed by connecting Θ_1 and Θ_2 via Cut-link is lazily and nondeterministically reduced to a normal form in a polynomial time w.r.t n^{2^k}.

5 Concluding Remarks

It seems possible that a **P**-time Turing machine encoding in Light Affine Logic is mechanically translated into that in Light Linear Logic. A given proof of the **P**-time Turing machine encoding in Light Affine Logic, we replace all the formula occurrences A in the proof by $A\&1$ and then apply an extract function like Figure 29 to the resulting proof. But we did not adopt the method, since the simple transition makes a too redundant proof in Light Linear Logic. So we made some optimizations. For example, In [Rov99] $\forall X.X \otimes X \multimap X$ was used as the boolean type. The above mentioned translation makes $\forall X.((((X\&1) \otimes (X\&1))\&1) \multimap (X\&1))\&1$. The study to find optimal translations seems interesting.

Acknowledgements. The author thanks Luca Roversi for discussions at his visit to University of Torino.

References

[AR02] Asperti, A. and Roversi, L. Intuitionistic Light Affine Logic. *ACM Transactions on Computational Logic*, **3** (1),1–39, 2002.

[Asp98] Asperti, A. Light Affine Logic. In *LICS'98*, 1998.

[DK97] Di Cosmo, R. and Kesner, D. Strong Normalization of Explicit Substitutions via Cut Elimination in Proof Nets. In *LICS'97*, 1997.

[Gir87] Girard, J.-Y. Linear logic. *Theoretical Computer Science*, **50**,1–102, 1987.

[Gir95] Girard, J.-Y. Linear logic: its syntax and semantics. *Advances in Linear Logic, London Mathematical Society Lecture Notes Series* **222**, 1995.

[Gir96] Girard, J.-Y. (1996) Proof-nets: the parallel syntax for proof-theory. In Ursini and Agliano, editors, *Logic and Algebra, New York, Marcel Dekker*, 1996.

[Gir98] Girard, J.-Y. Light Linear Logic. *Information and Computation*, **143**, 175–204, 1998.

[Mat96] Matsuoka, S. Nondeterministic Linear Logic. IPSJ SIGNotes PROgramming No.12, 1996.

[MO00] Murawski, A. S. and Ong, C.-H. L. Can safe recursion be interpreted in light logic?. Available from Luke Ong's home page (`http://web.comlab.ox.ac.uk/oucl/work/luke.ong/`), 2000.

[Mau03] Maurel, F. Nondeterministic Light Logics and NP-Time. TLCA 2003, LNCS 2701, 2003.

[Pap94] Papadimitriou, C. Computational Complexity, Addison Wesley, 1994.

[Rov99] Roversi, L. A P-Time Completeness proof for light logics. In *Ninth Annual Conference of the EACSL (CSL'99), Lecture Notes in Computer Science*, **1683**, 469–483, 1999.

A Our encoding of Turing machines(continued)

For shorthand, we use $!^k A$ to represent $\overbrace{!(\cdots!(!}^{k} A)\cdots)$. We also use $?^k A \equiv_{\mathrm{def}} \overbrace{?(\cdots?(?}^{k} A)\cdots)$ and $\$^k A \equiv_{\mathrm{def}} \overbrace{\$(\cdots\$(\$}^{k} A)\cdots)$ In addition, we implicitly assume coercion for $\bot$. In other terms, we assume that by using a proof net with conclusions $?^{k_1}\bot, \ldots, ?^{k_p}\bot, \Gamma$ we can construct a proof net with conclusions $?^k\bot, \Gamma$ provided $k \geq k_1, \ldots, k_p$. This is done by using p proof nets that have the forms of Figure 45, Cut-links, and one contraction-link.

Unlike [Gir98], we do not use **int** $\equiv_{\mathrm{def}} \forall X.!(X \multimap X) \multimap \$(X \multimap X)$ for our Turing machine encoding:

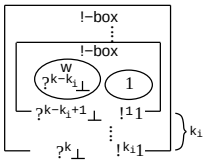

FIG. 45. coercion of $\bot$

we only use **bint** $\equiv_{\mathrm{def}} \forall X.!(X \multimap X) \multimap !(X \multimap X) \multimap \$(X \multimap X)$ instead of **int** since we would like to reduce the number of proof nets appearing in this paper. It is possible to construct a Turing machine encoding from our transition function encoding by using **int**. In the following we show three basic functions for **bint**: two successor function and addition. Figure 46 and Figure 47 show two successor functions for **bint**: we call these `suc0` and `suc1` respectively. Unlike **int**, **bint** has two successor functions.

Figure 48 shows the analogue in **bint** to the addition in **int**: we call the proof net `badd`. If we

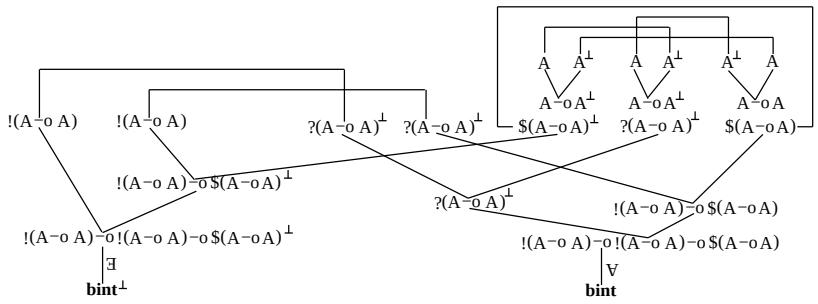

FIG. 46. `suc0`

regard two inputs proofs of **bint** of the proof net as two lists which only have 0 and 1, then we can regard `badd` as a concatenation function of two inputs.

Figure 49 shows a proof net called `bmul`. In the figure, `empty` is a proof net of **bint** that does not have exponential-links except for two weakening-links with $?(A \multimap A)^\bot$. Let Θ_1 be a **bint** proof that is supplied to $\mathbf{bint}^\bot$ port of `bmul` and Θ_2 be a proof net with $!\mathbf{bint}$ as one of conclusions that is supplied to $?\mathbf{bint}^\bot$ port of `bmul`. Let ℓ be the length of Θ_1. The evaluated result of `bmul` provided inputs Θ_1 and Θ_2 are given, is ℓ copies of Θ_2. Let m be the length of Θ_2. The length of the result is $\ell \times m$. The proof net `bmul` is analogous to multiplication of **int**.

The proof net shown in Figure 50 transform a **bint** proof into a **config** proof that is a initial configuration of Turing machines. We call the proof net `bint2config`. By using `bint2config` and `transition`, our encoding of the transition function of M, we can construct the engine part of Turing machines shown in Figure 51. But it is not sufficient for a proof of Theorem 3.1: besides we need

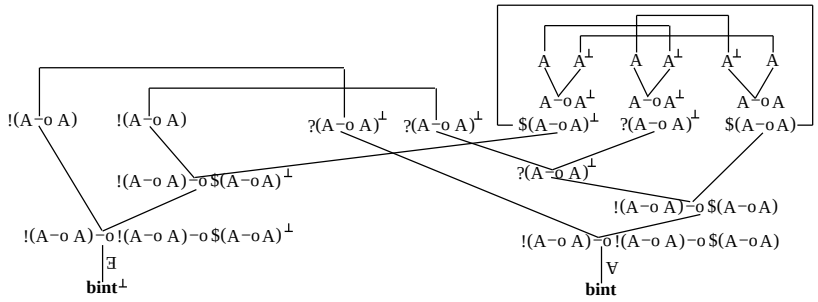

FIG. 47. `suc1`

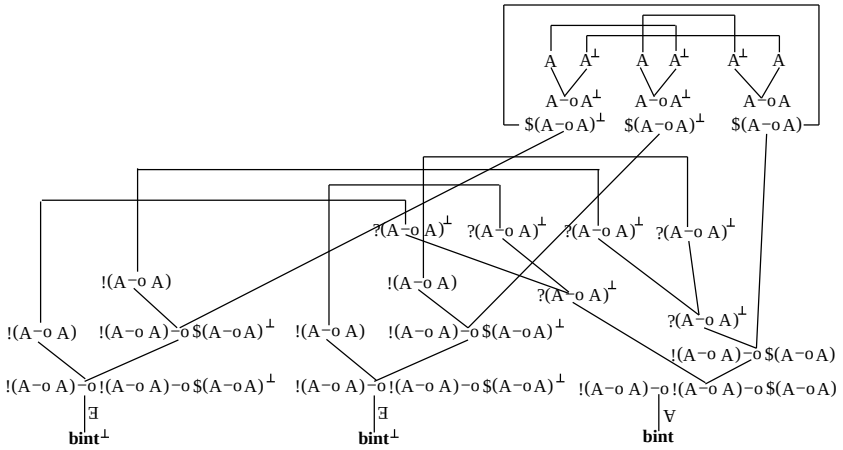

FIG. 48. `badd`

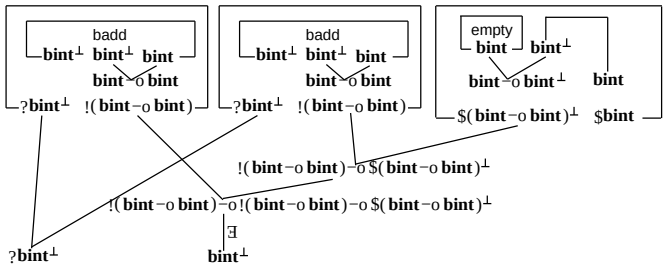

FIG. 49. `bmul`

constructions for polynomial time bound. To do this, we prepare several proof nets.

The proof net $\texttt{coer}^{p,q}$ of Figure 52 is the **bint** version of **int** coercion of [Gir98]. Then p must be

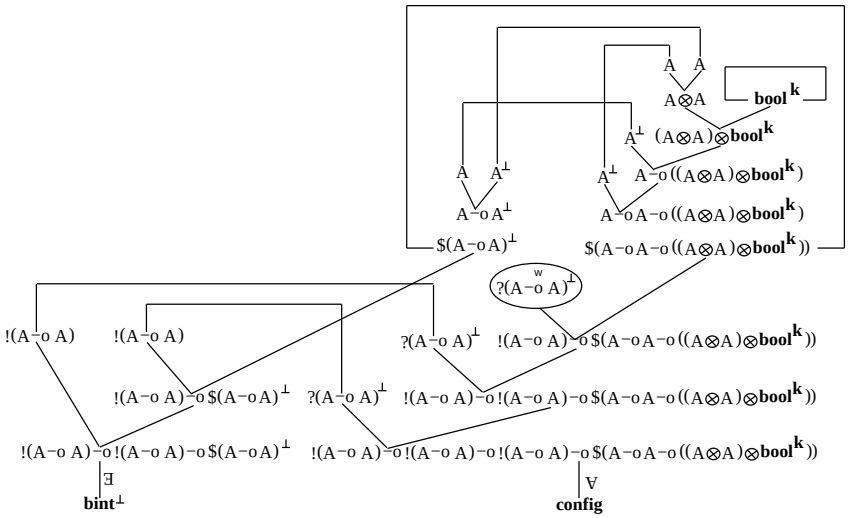

FIG. 50. `bint2config`

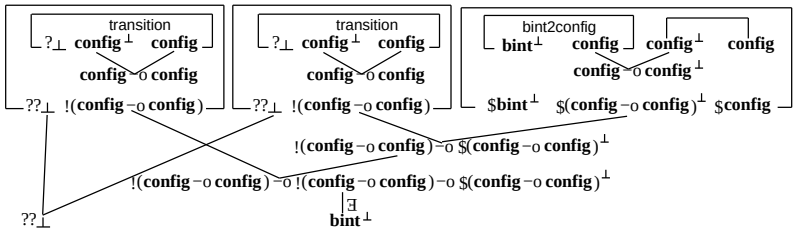

FIG. 51. `tm_engine`

greater than 0. This proof net is used in Figure 54 and Figure 56. The proof net `k-contraction` of Figure 53 is also the **bint** version of **int** contraction of [Gir98]. This proof net is used in Figure 54 and Figure 55.

The proof net in Figure 54 is used in Figure 55. This is basically k compositions of `bmul`. The **bint** proof c_{const} in Figure 54 is a constant that does not depend on the lengths of inputs of Turing machine M.

The proof net `kpolynomial` of Figure 55 is our polynomial construction with degree k. Let Θ be a proof net of **bint** and ℓ be the length of Θ. The evaluated result of `kpolynomial` provided an input Θ is given, is a nest of \$-boxes which has an inside proof net of **bint** with the length $c_{\text{const}} \times \ell^k$.

Finally we obtain our encoding of a Turing machine with polynomial time bound of Figure 56. This completes our proof of Theorem 3.1.

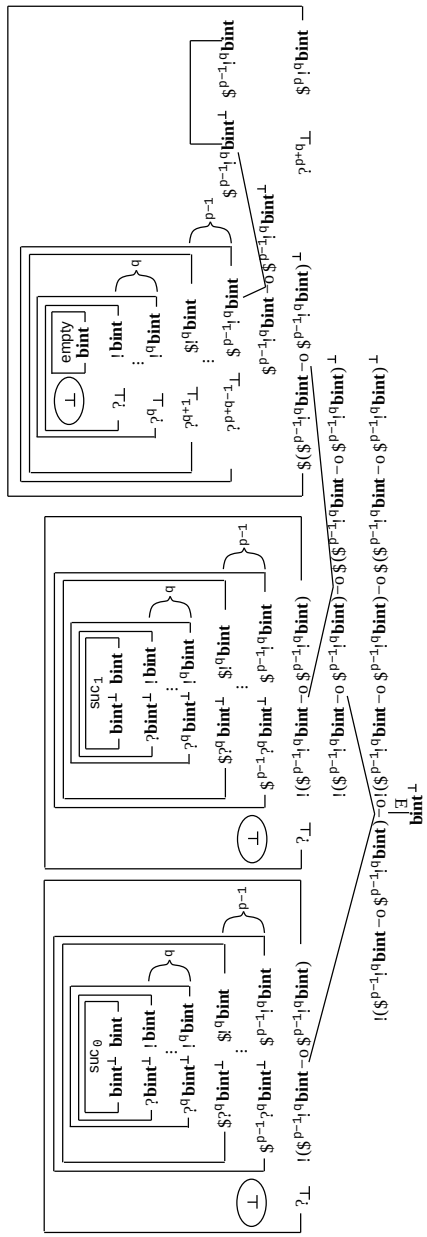

FIG. 52. $\mathtt{coer}^{p,q}$

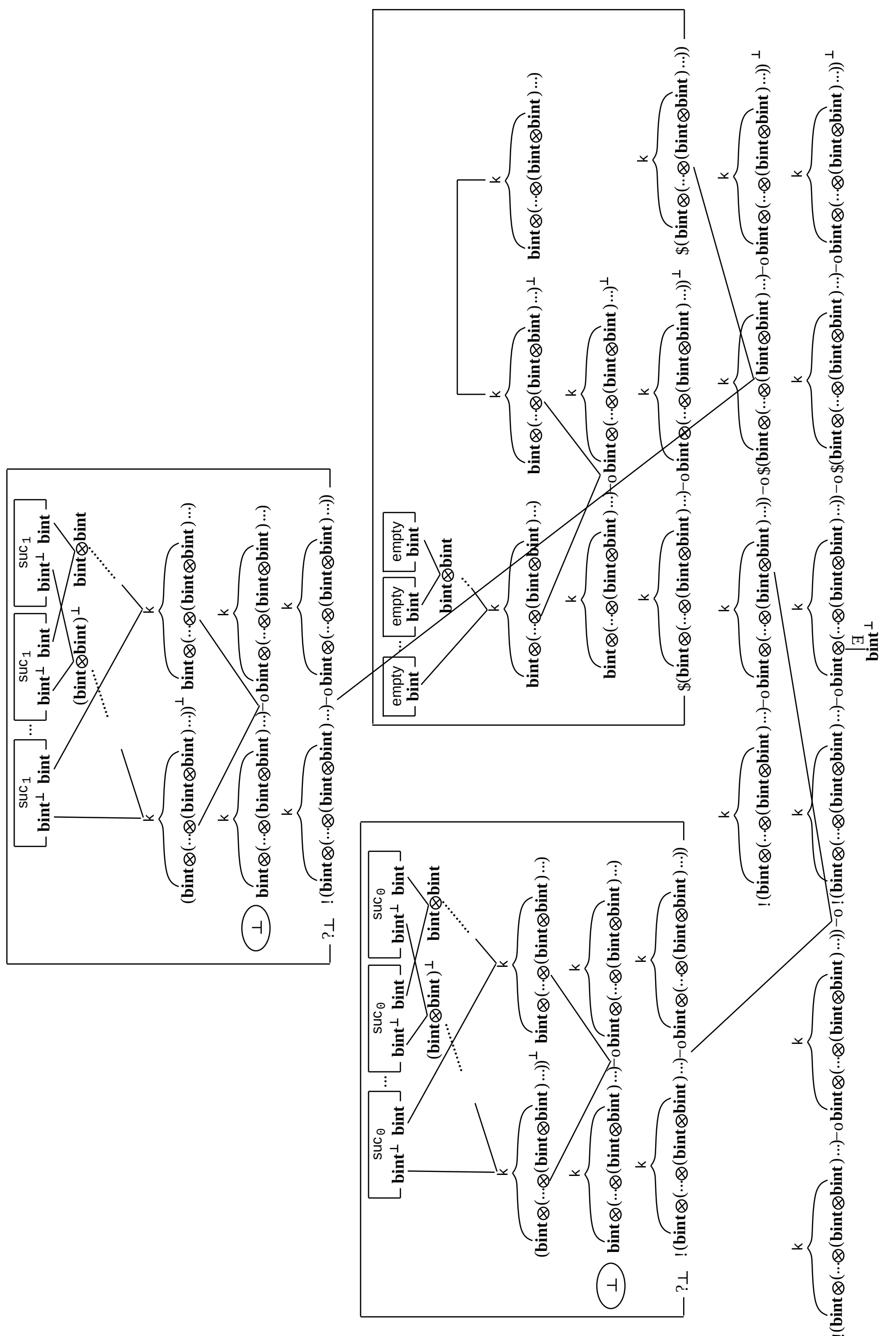

FIG. 53. k-contraction

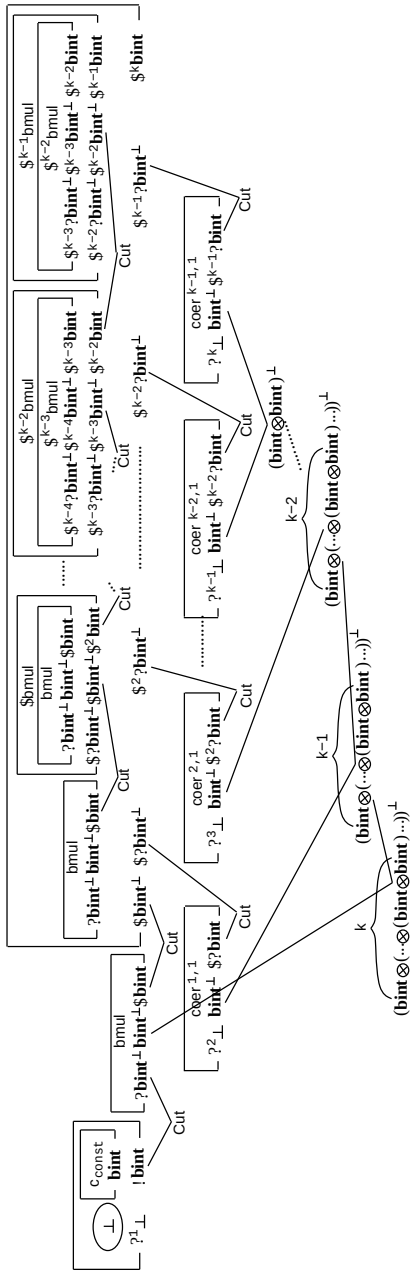

FIG. 54. `k-composition` of multiplications

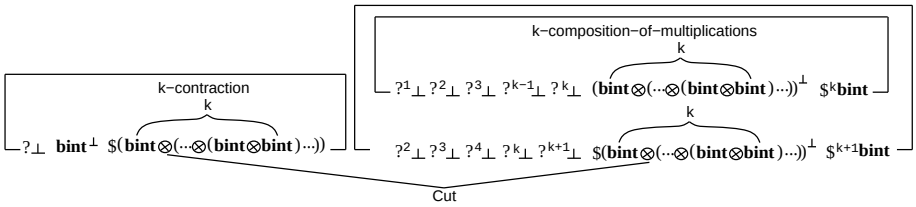

FIG. 55. `kpolynomial`

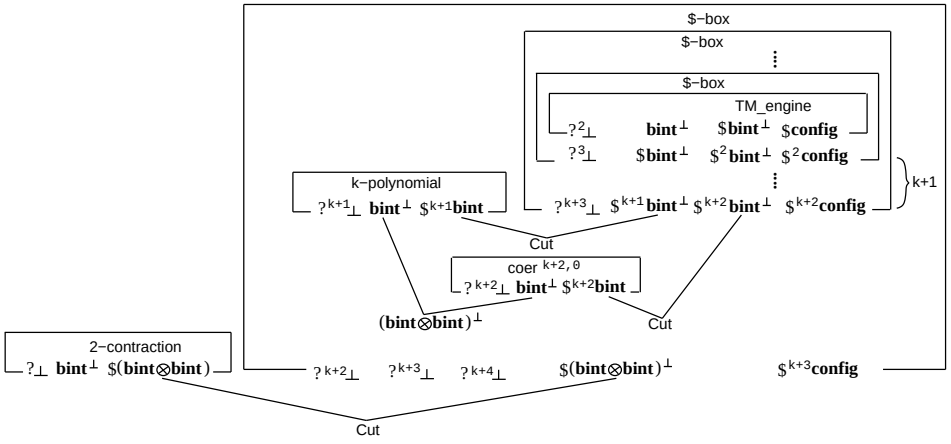

FIG. 56. `TM`

B A transformation from config proofs into bint proofs

At first we remark that when a given proof net with conclusion Γ, we can construct a proof net with $\mathbf{bool^2}^{\perp}, \Gamma$ as shown in Figure 57. It is easy to extend the remark to the general $\mathbf{bool^k}$ case for $k \leq 2$.
Then based on the above remark, as a derived rule, we introduce $\mathbf{bool^k}$-axiom as shown in Figure 58 in order to keep figures as simple as possible.

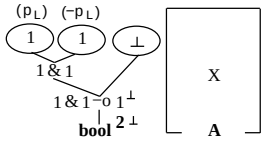

FIG. 57. $\mathbf{bool^2}$-weakening

FIG. 58. $\mathbf{bool^k}$-axiom

In order to translate **config** proofs into **bint** proofs, we introduce an immediate type **tint** $\equiv_{\mathrm{def}}$ $\forall X.!(X \multimap X) \multimap !(X \multimap X) \multimap !(X \multimap X) \multimap \$(X \multimap X)$.
Figure 59 shows our translator from **config** proofs into **bint** proofs. When a given **config** proof, at first we duplicate the proof by using `2-contraction-config` proof net. A construction of `2-contraction-config` proof net is not so easy as that of `2-contraction` for **bint**. Appendix C is devoted to the construction.

After that, each duplicated **config** proof net is projected into a `tint` proof by using `prj1` or `prj2` shown in Figure 60. The purpose of `prj1` is to extract the left parts of configurations of Turing machines and similarly that of `prj2` is to extract the right parts.
Proof net `prj1` has proof net `prj1sub` shown in Figure 61 as a sub-proof net and `prj2` has `prj2sub` shown in Figure 62. Proof net `prj1` also has proof nets $\texttt{tsuc0}^{\mathtt{r}}$, $\texttt{tsuc1}^{\mathtt{r}}$, and $\texttt{tsuc}*^{\mathtt{r}}$ as sub-proof nets and `prj2` has `tsuc0`, `tsuc1`, and `tsuc*`. Figure 63 and Figure 64 show proof net `tsuc0` and $\texttt{tsuc0}^{\mathtt{r}}$ respectively. We omit `tsuc1`, `tsuc*`, $\texttt{tsuc1}^{\mathtt{r}}$, and $\texttt{tsuc}*^{\mathtt{r}}$, since the constructions of these proof nets are easy exercise. Note that in order to recover tapes correctly we need to reverse the left parts of tapes. Next we concatenate obtained two **tint** proofs by `tadd` of Figure 65.

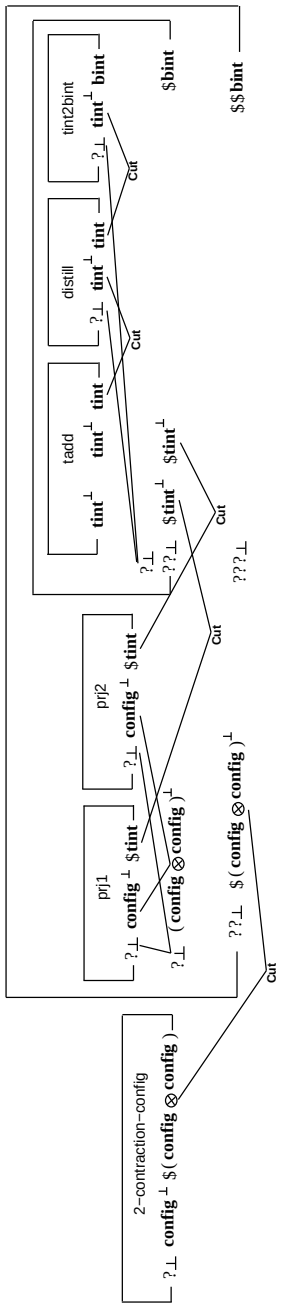

FIG. 59. `config2bint`

FIG. 60. `prj1` or `prj2`

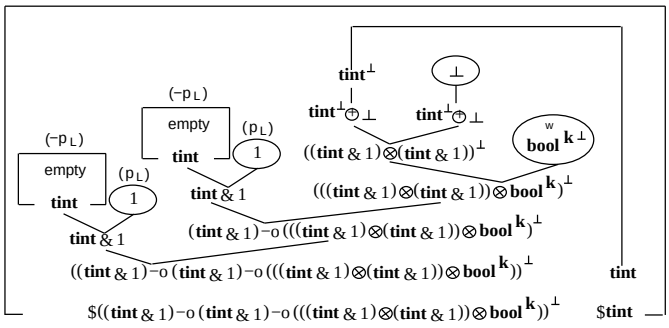

FIG. 61. `prj1sub`

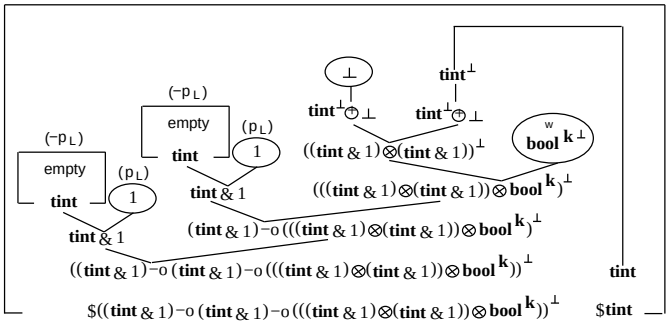

FIG. 62. `prj2sub`

We distinguish the two normal proofs of $\mathbf{bool}^{\mathbf{2}}$. One is called $\texttt{Pi}_{01}$. and the other $\texttt{Pi}_{*}$ (see Figure 66). Next we apply `distill` of Figure 67 to the obtained **tint** proof. The construction of the proof net `distill` is inspired by that of `strip` term in [MO00]. The intention of the `distill` proof net is to keep occurrences of 0 and 1 until the first $*$ occurrence is reached. After that, the rest are discarded. Figure 68 shows three sub-proof nets `distill_step_X` (`X`=1,2, and $*$) of the `distill` proof net. Moreover, two sub-proof nets $\texttt{distill\_step\_sub\_X}_{\texttt{L}}$ and $\texttt{distill\_step\_sub\_X}_{\texttt{R}}$ of `distill_step_X` have the forms of Figure 69 or Figure 70. Table 1 shows the correspondence.
Figure 71 shows `tint2bint` proof. The intention is to remove $*$-entry.

	D=L	D=R
X=0	distill_step_sub_join	distill_step_sub_discard
X=1	distill_step_sub_join	distill_step_sub_discard
X=$*$	distill_step_sub_discard	distill_step_sub_discard

TABLE 1. $\texttt{distill\_step\_sub\_X}_{\texttt{D}}$

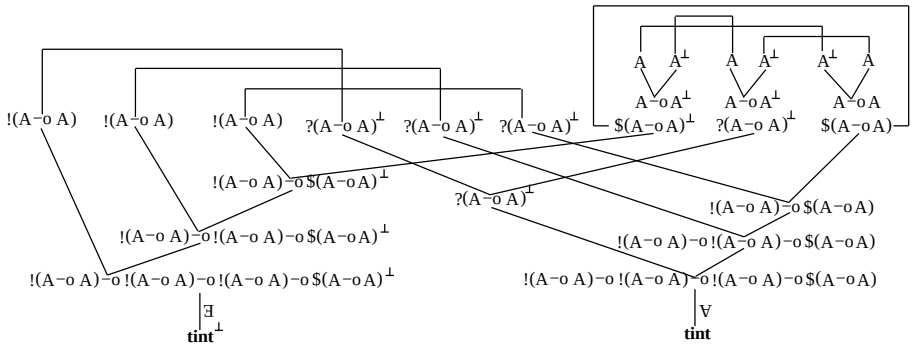

FIG. 63. `tsuc0`

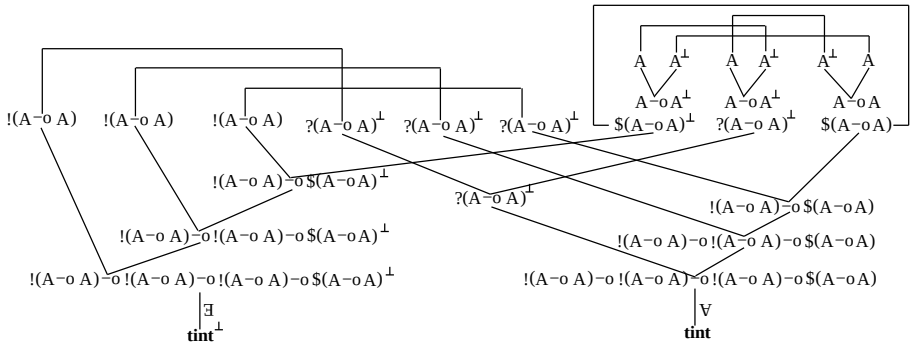

FIG. 64. `tsuc0`$^{\mathtt{r}}$

C Contraction on config

In this section we give how to construct `2-contraction-config` proof net. In `config2bint` of Appendix B we do not need the $\mathbf{book}^{\mathbf{k}}$-part of a given **config** proof. Hence in the construction of `2-contraction-config` proof net we could discard $\mathbf{book}^{\mathbf{k}}$-parts. But we give a general construction that duplicates $\mathbf{book}^{\mathbf{k}}$-parts here. Figure 72 shows `2-contraction-config` proof net. In this proof net,

1. when given a **config** proof net, `pre_config_dup` of Figure 73 outputs a quartet of **config** proof nets, where two **config** proofs are the same and only keep the left part and $\mathbf{book}^{\mathbf{k}}$ of the input, and the rest, which are two **config** proofs, are also the same and only keep the right part and $\mathbf{book}^{\mathbf{k}}$ of the input;
2. each `configadd` of Figure 80 concatenate two **config** proof nets in the quartet.

Type $\mathbf{dconfig}_{\mathbf{2}}^{\mathbf{k}}$ of proof net `pre_config_dup` is defined as follows:

$$\mathbf{config}_{\mathbf{2}}^{\mathbf{k}} \equiv_{\mathrm{def}} \overbrace{(\mathbf{config} \otimes \mathbf{config})\&(\cdots\&((\mathbf{config} \otimes \mathbf{config})\&(\mathbf{config} \otimes \mathbf{config}))\cdots)}^{k}$$

$$\mathbf{dconfig}_{\mathbf{2}}^{\mathbf{k}} \equiv_{\mathrm{def}} \mathbf{config}_{\mathbf{2}}^{\mathbf{k}} \oplus \mathbf{config}_{\mathbf{2}}^{\mathbf{k}}$$

In `pre_config_dup`, at first, we make $4k$ **config** proofs. Then according to the $\mathbf{book}^{\mathbf{k}}$-value of the input **config** proof, we choose 4 **config** proofs. That is why we use k-ary tuples by $\&$-connectives in $\mathbf{config}_{\mathbf{2}}^{\mathbf{k}}$. In addition we need to distinguish the left part and the right part of the input **config** proof. That is why we use one $\oplus$-connective in $\mathbf{dconfig}_{\mathbf{2}}^{\mathbf{k}}$.

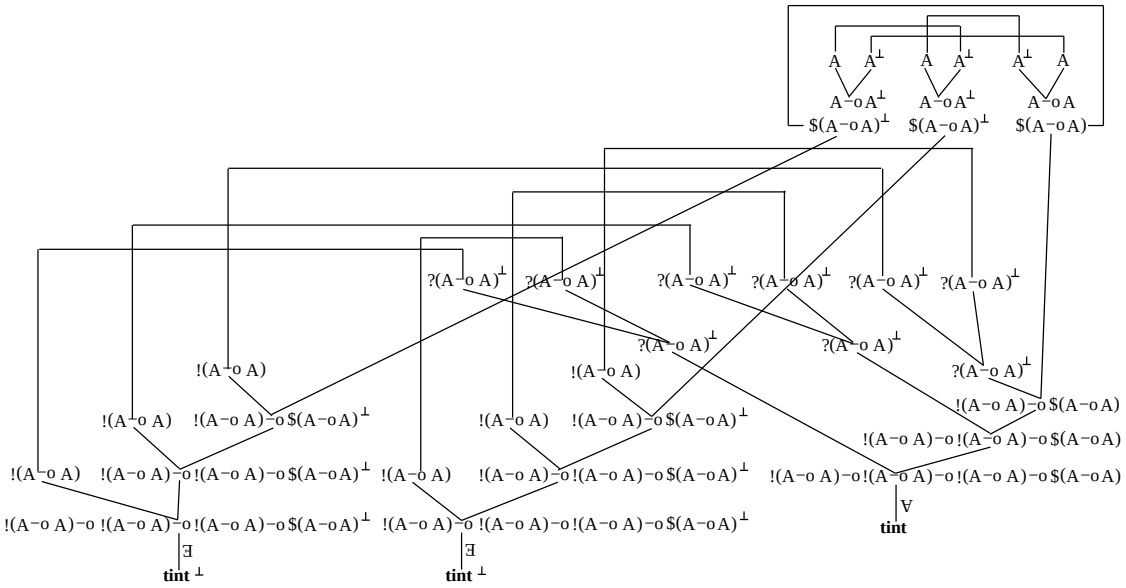

FIG. 65. `tadd`

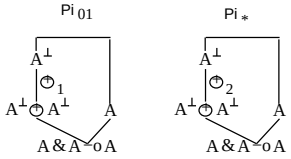

FIG. 66. $\texttt{Pi}_{01}$ and $\texttt{Pi}_{*}$

Figure 74 shows sub-proof net `pre_config_dup_main` of `pre_config_dup`. Note that as shown in Figure 75, we can duplicate $\mathbf{book^2}$ proof without using \$ (of course we can easily extend this construction to the $\mathbf{bool^k}$ case).
Proof nets `dconfig_sucX` (where $X = 1, 2,$ and $*$) shown in Figure 77 occur in
`pre_config_dup_main` as sub-proof nets. Figure 78 and Figure 79 show proof nets `suc0L` and `suc0R`. We omit `suc1L`, `suc1R`, `suc*L` and `suc*R` since the constructions of these proof nets are easy exercise.

Received 07/10/2003

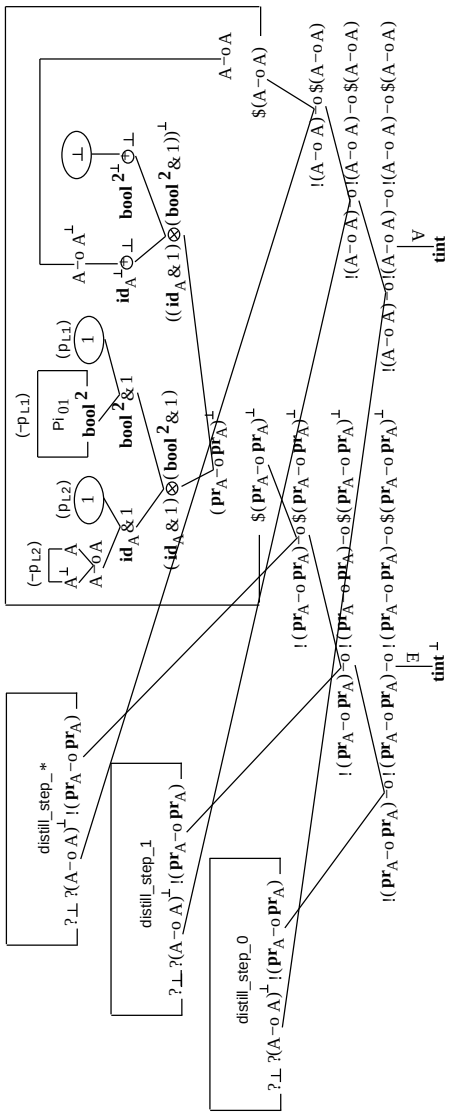

FIG. 67. `distill`

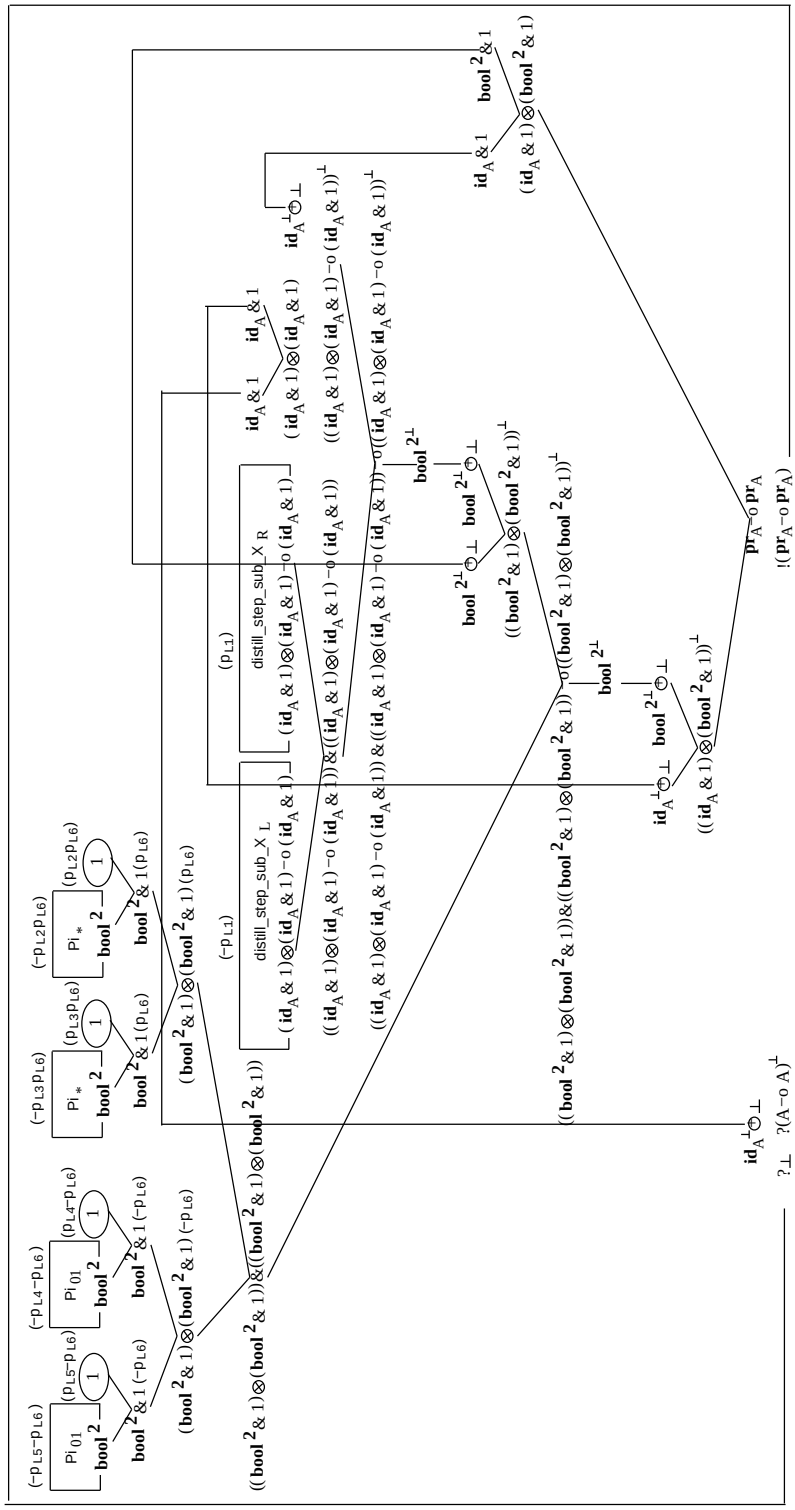

FIG. 68. distill_step_X

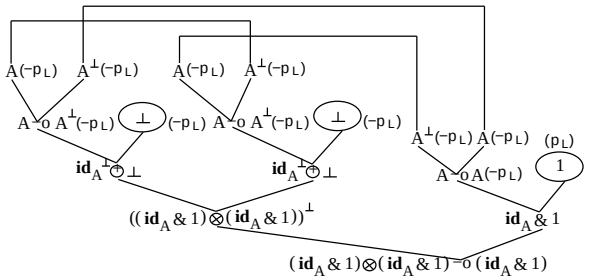

FIG. 69. `distill_step_sub_join`

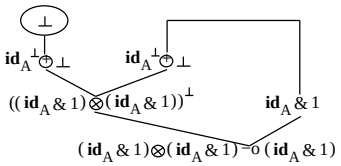

FIG. 70. `distill_step_sub_discard`

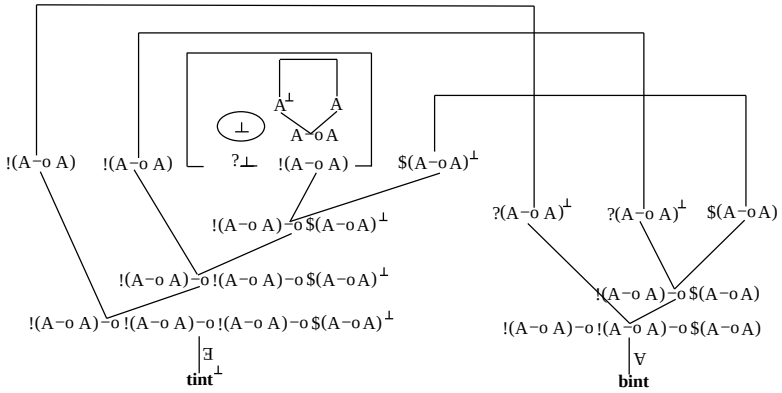

FIG. 71. `tint2bint`

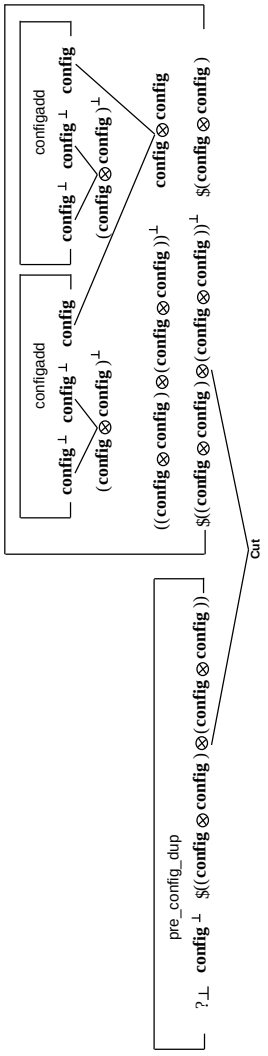

FIG. 72. 2-contraction-config

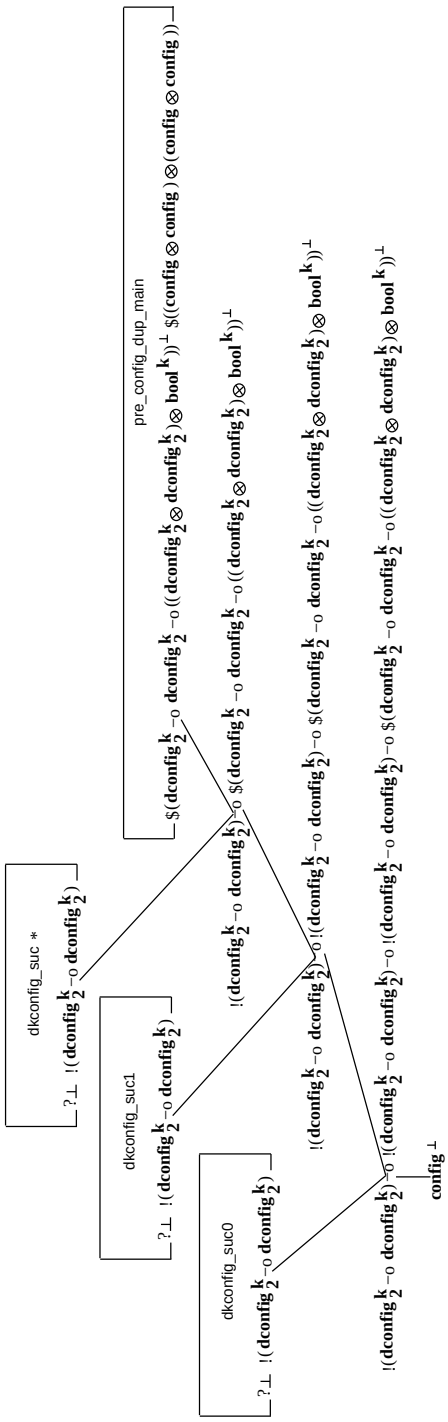

FIG. 73. `pre_config_dup`

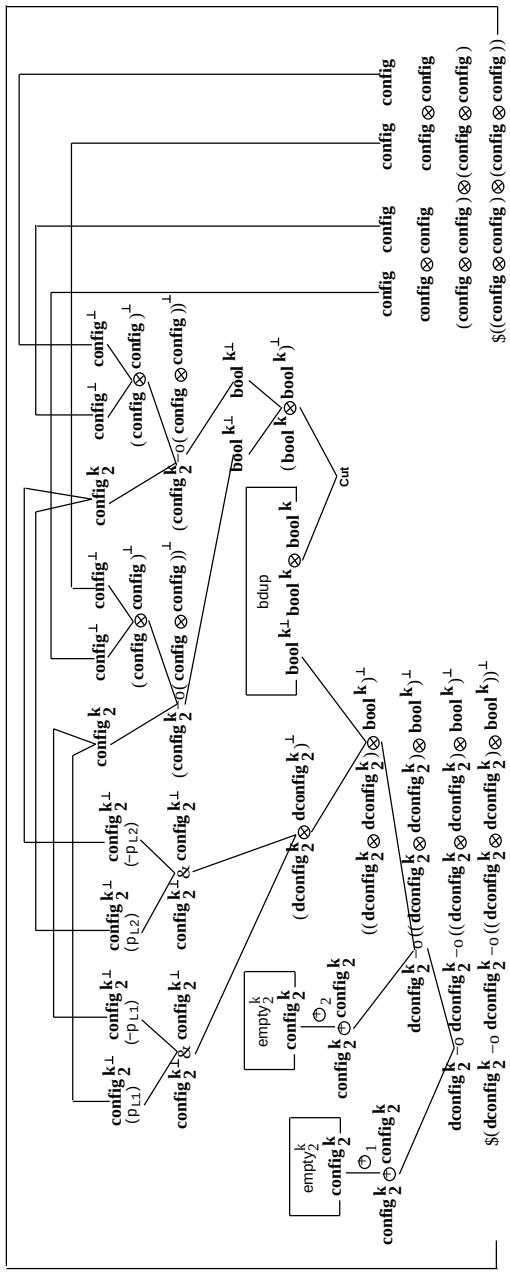

FIG. 74. pre_config_dup_main

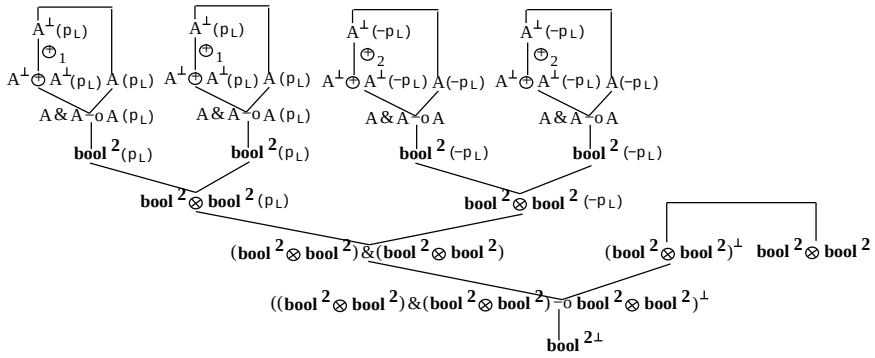

FIG. 75. bdup

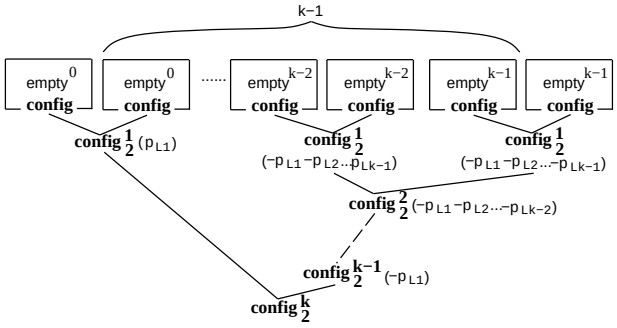

FIG. 76. $\mathtt{empty}_2^k$

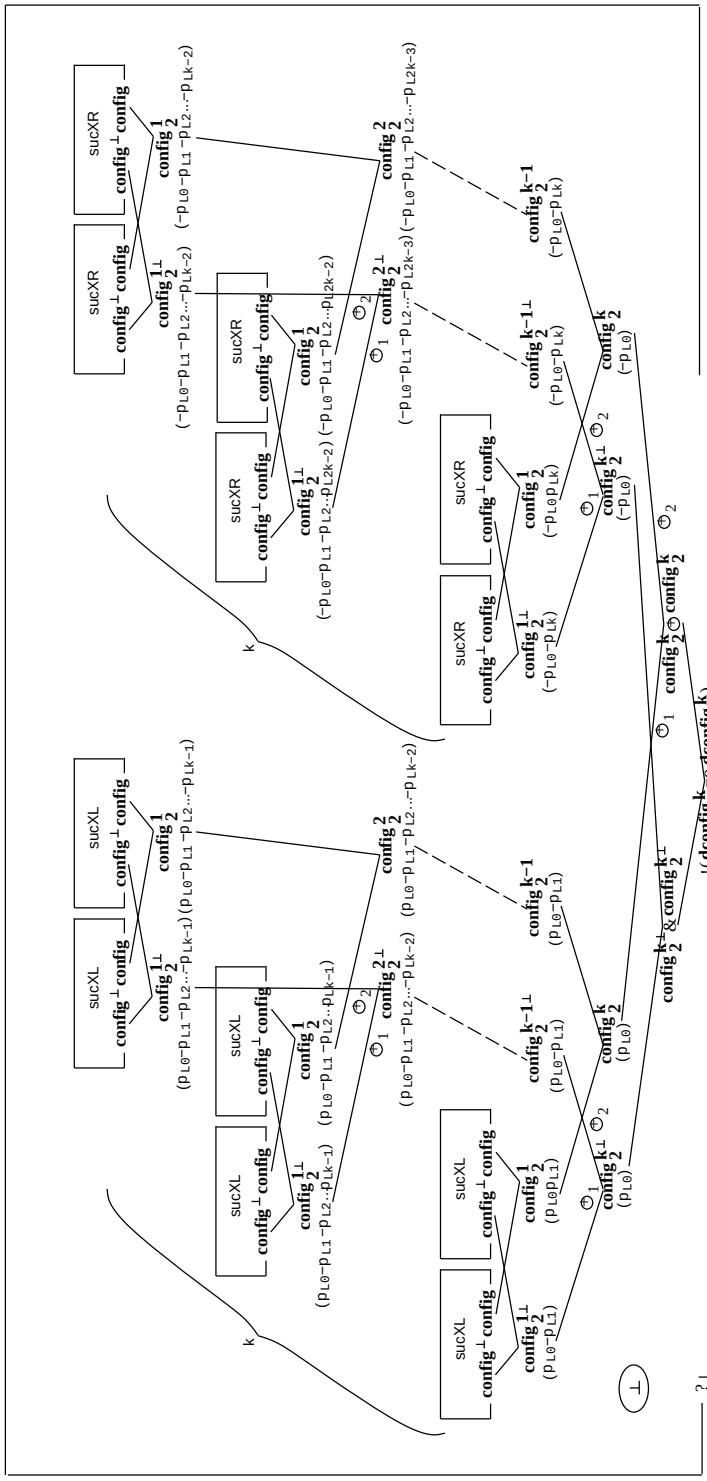

FIG. 77. dconfig_sucX

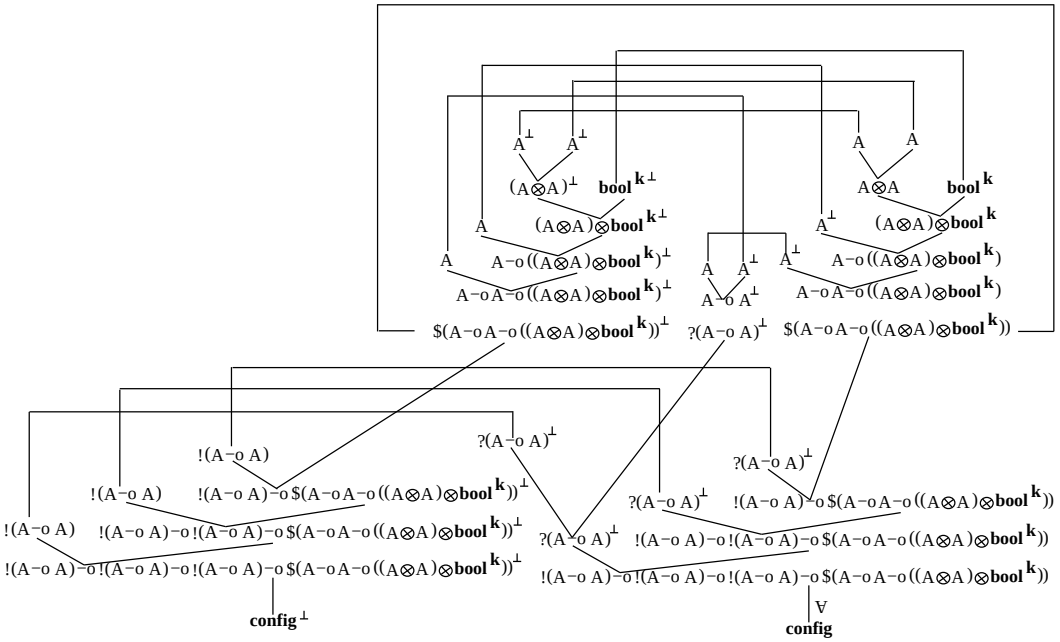

FIG. 78. `suc0L`

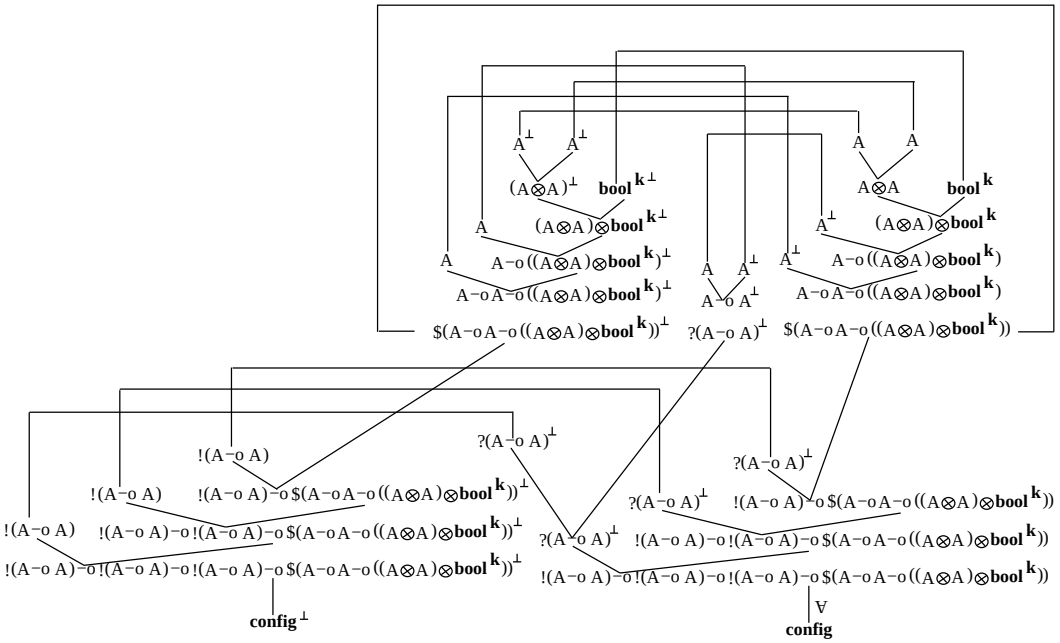

FIG. 79. `suc0R`

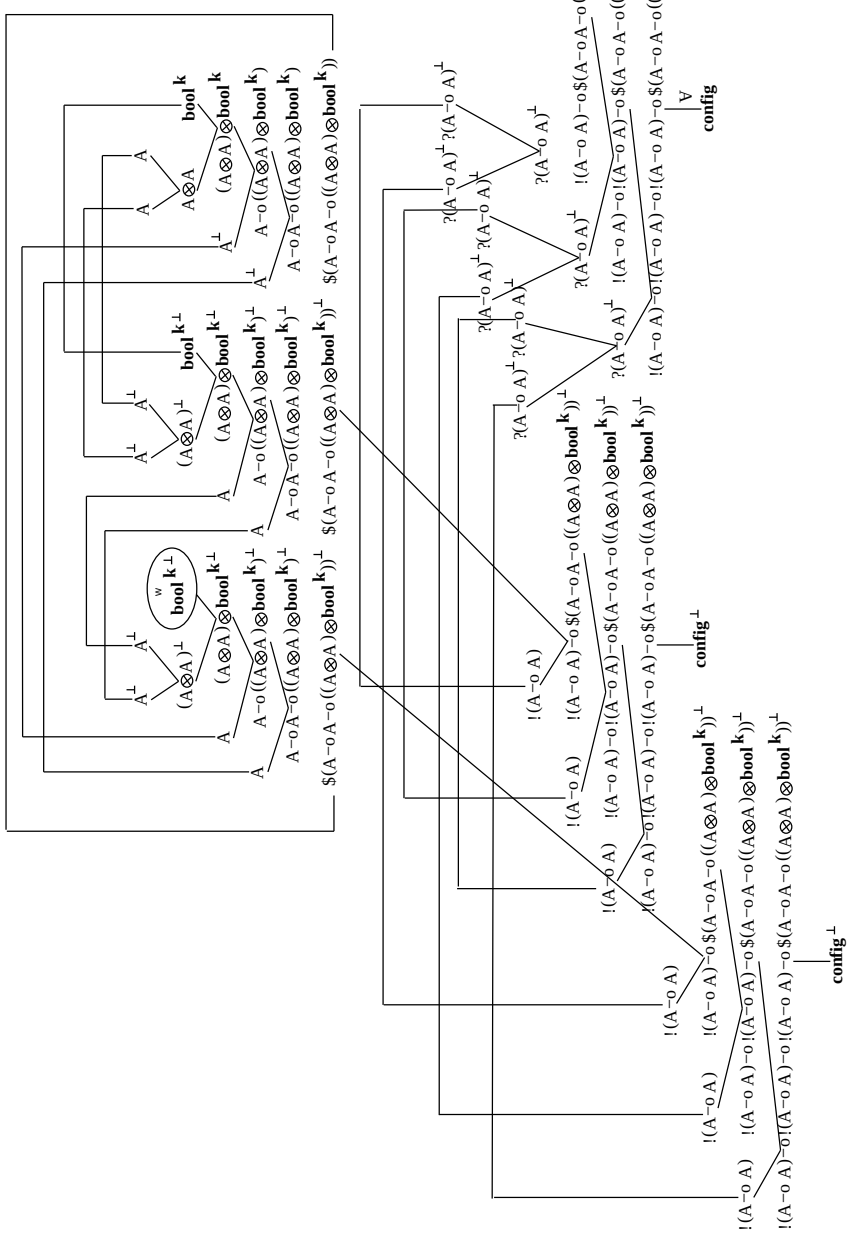

FIG. 80. configadd