Deriving Theorems in Implicational Linear Logic, Declaratively

Paul Tarau
University of North Texas
Texas, USA
paul.tarau@unt.edu

Valeria de Paiva
Topos Institute
California, USA
valeria.depaiva@gmail.com

The problem we want to solve is how to generate all theorems of a given size in the implicational fragment of propositional intuitionistic linear logic. We start by filtering for linearity the proof terms associated by our Prolog-based theorem prover for Implicational Intuitionistic Logic. This works, but using for each formula a PSPACE-complete algorithm limits it to very small formulas. We take a few walks back and forth over the bridge between proof terms and theorems, provided by the Curry-Howard isomorphism, and derive step-by-step an efficient algorithm requiring a low polynomial effort per generated theorem. The resulting Prolog program runs in $O(N)$ space for terms of size N and generates in a few hours 7,566,084,686 theorems in the implicational fragment of Linear Intuitionistic Logic together with their proof terms in normal form. As applications, we generate datasets for correctness and scalability testing of linear logic theorem provers and training data for neural networks working on theorem proving challenges. The results in the paper, organized as a literate Prolog program, are fully replicable.

Keywords: combinatorial generation of provable formulas of a given size, intuitionistic and linear logic theorem provers, theorems of the implicational fragment of propositional linear intuitionistic logic, Curry-Howard isomorphism, efficient generation of linear lambda terms in normal form, Prolog programs for lambda term generation and theorem proving.

1 Introduction

Linear Logic [12] as a resource-control mechanism constrains the use of formulas available as premises in a proof. In its full generality, a larger number of operators ensures on-demand (re)use of these resources, in a controlled way (e.g., with exponentials like "**!**"). While in full propositional form linear logic is already Turing complete, its *implicational fragment* is decidable and finding low polynomial algorithms for proving its theorems is especially interesting when large datasets of theorems need to be generated. Such datasets, combining tautologies and their proof terms can be useful for testing correctness and scalability of linear logic theorem provers (not necessarily restricted to the implicational fragment) and more importantly, for training deep learning networks focusing on *neuro-symbolic* computations, e.g., [9, 20, 23], an emerging research trend, motivated in part by the need for *explainable AI* in medical, legal or other industrial AI applications.

Of particular interest in the correspondence between computations and proofs is the Curry-Howard isomorphism [16, 32]. In its simplest form, it connects the implicational fragment of propositional intuitionistic logic with types in the *simply typed lambda calculus*. A low polynomial type inference algorithm associates a type (when it exists) to a lambda term. Harder (PSPACE-complete, see [27]) algorithms associate inhabitants to a given type expression with the resulting lambda term (typically in normal form) serving as a witness for the existence of a proof for the corresponding tautology in implicational propositional intuitionistic logic. In particular, when restricting linear logic to its implicational fragment

F. Ricca, A. Russo et al. (Eds.): Proc. 36th International Conference
on Logic Programming (Technical Communications) 2020 (ICLP 2020)
EPTCS 325, 2020, pp. 110–123, doi:10.4204/EPTCS.325.18

(syntactically, just binary trees with the "lollipop" operator "`-o`" and variables as leaves), it becomes interesting to find out how formulas relate to proof terms, seen as linear lambda terms (constrained to have exactly one variable associated to each lambda binder). Also, this is important because such formulas correspond to linear types, which can significantly optimize memory management by allowing reuse of single-threaded data structures as it has been implemented in Linear Haskell [5].

This singles out the usefulness of efficiently generating a dataset of linear types/linear logic tautologies, the focus of this paper, with at least three applications in mind:

- correctness and scalability tests for linear logic theorem provers, complementing the ones described in [22]
- a formula/proof term dataset for training neuro-symbolic systems with a likely to be learnable, PTIME-decidable set of problems
- a correctness and scalability test for systems implementing linear types (e.g., Linear Haskell)

We will proceed incrementally, with a step-by-step derivation process, starting with adapting an intuitionistic theorem prover to work as a prover for the implicational fragment of linear intuitionistic logic. From this solution, seen as an executable specification (correct but slow) we derive, after crossing the Curry-Howard "bridge", progressively more constrained lambda term generators, ending with one that not only generates efficiently closed linear lambda terms in normal form but it also infers their types, corresponding to theorems in the language of implicational linear intuitionistic logic. Moreover, we engineer the generation mechanism such that the lambda terms and their principal types have exactly the same size. Thus, without help of a theorem prover, we will uniformly generate all linear implicational tautologies of a given size [1]. As a result, our Prolog code defines constructively a size-preserving bijection between these two sets, on the opposite side of the Curry-Howard bridge. As a final step, we reengineer this bijection to work in reverse mode, as a theorem prover, that given an implicational formula, returns its proof term, if it exists.

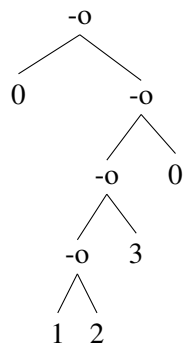
The rest of the paper is organized as follows. Section 2 introduces formula generators for (linear) implicational formulas of a given size. Section 3 describes the adaptation of an intuitionistic theorem prover to formulas of implicational propositional linear logic. Section 4 moves our effort to the other side of the Curry-Howard isomorphism, resulting in generation of linear lambda terms in normal form that are bijectively connected to their principal types corresponding to theorems in implicational linear logic. Section 5 discusses our results in the wider context of linear logic research. Section 6 overviews related work and section 7 concludes the paper. *As the paper is actually a literate Prolog program, it's code is also made available as a separate file*[2]*, in compliance with our commitment to fully replicable research results.*

2 The Formula Generators

We will first develop formula generators to cover all implicational formulas of a given size, measured as the number of internal nodes. With the "lollipop" operator "`-o`" labeling internal nodes and natural numbers starting with `0` as variables labeling the leaves, one such formula tree to be generated for N=4, is the following:

[1] A dataset containing the theorems generated and their proof-terms is available at `http://www.cse.unt.edu/~tarau/datasets/lltaut/` .

[2] `https://raw.githubusercontent.com/ptarau/TypesAndProofs/master/tlin.pro`



2.1 Generating Formula Trees

First, we generate all binary trees of size `N` with internal implication nodes "`-o/2`", while collecting their `N+1` distinct logic variable leaves to a list.

We define "`-o`" as an operator and we use `pred/2` to consume one unit of size on each internal node.

```
:-op(900,xfy,( '-o' )).

gen_tree(N,Tree,Leaves):-gen_tree(Tree,N,0,Leaves,[]).

gen_tree(V,N,N,[V|Vs],Vs).
gen_tree((A '-o' B),SN1,N3,Vs1,Vs3):-pred(SN1,N1),
  gen_tree(A,N1,N2,Vs1,Vs2),
  gen_tree(B,N2,N3,Vs2,Vs3).

pred(SN,N):-succ(N,SN).
```

The counts of generated trees match entry **A000108** in [25], representing the Catalan numbers [26], binary trees with *N* internal nodes.

2.2 Generating the variable labels

The next step toward generating the set of all type formulas is observing that logic variables define equivalence classes that correspond to *partitions of the set of variables*, simply by selectively unifying them.

The predicate `mpart_of/2` takes a list of distinct logic variables and generates partitions-as-equivalence-relations by unifying them "nondeterministically". It also collects the unique variables defining the equivalence classes, as a list given by its second argument.

```
mpart_of([],[]).
mpart_of([U|Xs],[U|Us]):-mcomplement_of(U,Xs,Rs),mpart_of(Rs,Us).
```

To implement a set-partition generator, we split a set repeatedly in subset+complement pairs with help from the predicate `mcomplement_of/2`.

```
mcomplement_of(_,[],[]).
mcomplement_of(U,[X|Xs],NewZs):-mcomplement_of(U,Xs,Zs),
  mplace_element(U,X,Zs,NewZs).

mplace_element(U,U,Zs,Zs).
mplace_element(_,X,Zs,[X|Zs]).
```

To generate all set partitions from a list of distinct variables of a given size, we build a list of fresh variables with Prolog's built-in predicate `length/2` and constrain `mpart_of/2` to use them as the set to be partitioned.

```
partitions(N,Ps):-length(Ps,N),mpart_of(Ps,_).
```

The counts of the resulting set-partitions (Bell numbers) `1, 1, 2, 5, 15, 52, 203,...` correspond to the entry **A000110** in [25].

Example 1 *Set partitions of size 3 expressed as variable equalities.*

```
?- partitions(3,P).
P = [A, A, A]; P = [A, B, A]; P = [A, A, B]; P = [A, B, B]; P = [A, B, C].
```

We next bind leaf variables of formula trees to our set partitions and encode distinct variables as consecutive natural numbers starting at `0`.

```
natpartitions(Vs):-mpart_of(Vs,Ns),length(Ns,SL),succ(L,SL),numlist(0,L,Ns).

gen_formula(N,T):-gen_tree(N,T,Vs), natpartitions(Vs).
```

This sequence corresponds to entry **A289679** in [25], with the first terms being `1, 1, 2, 10, 75, 728, 8526, 115764, 1776060`, computed as `a(N) = Catalan(N)*Bell(N+1)`.

Example 2 *Some formulas of size 2.*

```
?- gen_formula(2,T).
T =  (0 -o 0 -o 0) ; T =  (0 -o 1 -o 0) ;
...
T =  ((0 -o 1) -o 1) ; T =  ((0 -o 1)-o 2) .
```

3 Adapting a Prover for Implicational Linear Logic

We will derive a prover for the implicational fragment of Propositional Intuitionistic Linear Logic by adding linearity constraints to the intuitionistic prover described in [31], (also in Appendix A).

3.1 Ensuring the Proof Terms are Linear

We will constrain intuitionistic proofs to produce linear lambda terms as proof terms. Lambda terms are represented using `a/2` for application nodes and logic variables for binders in `l/2` nodes and the variables they bind.

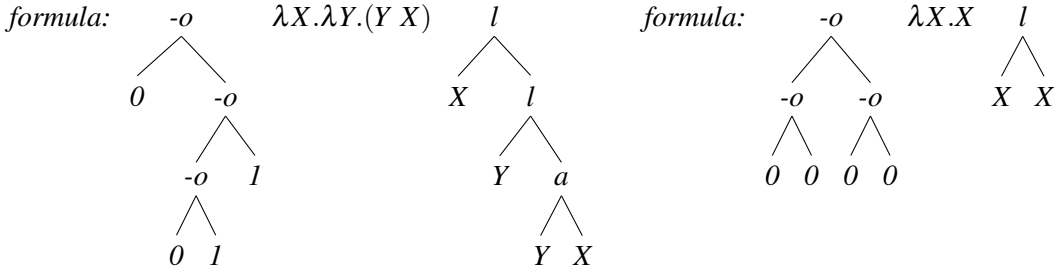
```
is_linear(X) :- \+ \+ is_linear1(X).

is_linear1(V):-var(V),!,V='$bound'.
is_linear1(l(X,E)):-is_linear1(E),nonvar(X).
is_linear1(a(A,B)):-is_linear1(A),is_linear1(B).
```

The predicate `is_linear/1` tests that each lambda binder corresponds to exactly one variable. Double negation is used to undo marking each variable with the atom "`$bound`".

A linear logic prover is now derived from the intuitionistic prover `prove_ipc` (see **Appendix**) by filtering *proof terms* with `is_linear`. The predicate `gen_taut` combines the implicational formula generator with the linear prover to obtain implicational linear logic tautologies of size `N`.

```
prove_lin(T,ProofTerm):-prove_ipc(T,ProofTerm),is_linear(ProofTerm).

gen_taut(N,T,ProofTerm):-gen_formula(N,T),prove_lin(T,ProofTerm).
```

Example 3 *Formulas of size 3 depicted as trees, together with their proof terms*



This is working but it is too slow, it takes 2203 seconds to generate the counts `0, 1, 0, 4, 0, 27, 0, 315, 0, 5565`. That's expected, not only because intuitionistic propositional logic proofs are PSPACE-complete even for the implicational fragment, but also because we are filtering through the super-exponential number of formulas counted by **A289679** in [25].

Besides performance issues, we are facing here three hurdles:

- proof terms are not necessarily in normal form
- multiple proof terms can result in the same provable formula
- sizes of formulas do not correlate in a simple way to the sizes of their proof terms

On the other hand, we know that type inference on lambda terms resulting in provable formulas can make the process much faster. This brings us to our next step.

4 Crossing the Curry-Howard Bridge: from Lambda Terms to Provable Formulas

It's time to look into the linear lambda terms corresponding to the formulas we want to generate.

While generators for linear lambda terms do exist (e.g., [19, 30] we are starting here with a clean design that propagates *size constraints* by keeping separate counts for lambda nodes and application nodes and then enforces linearity efficiently. This constraint reduces significantly the candidate trees to be decorated with lambda binders and variables as it is now like working with size N rather than size $2N+1$ in a super-exponentially growing set, while reducing the possible leaf labelings to much fewer than all possible combinations of variable names. Adding linearity constraints will further reduce the combinatorial explosion by ensuring that each lambda binder connects to a unique leaf variable.

4.1 A Generator for Linear Skeleton Motzkin Trees

First we use the fact that there are as many lambda binders as variables, given the one-to-one mapping required for the (completely) linear lambda calculus. Thus we will give N units to application nodes, corresponding to the $N+1$ variables in leaf position and $N+1$ units to lambda nodes, resulting in a total of $N+N+1 = 2N+1$ internal nodes. We define size by allocating one unit to each lambda node and one to each application node. This, for a given N, will produce lambda terms of size $2N+1$. But first we

will only generate term skeletons for which this constraint holds, with a dummy `leaf` node. As these are a special case of Motzkin trees (also called binary-unary trees, see **A001006** at [25]), we call them *linear Motzkin skeletons*. We will obtain linear lambda terms by decorating these trees with lambda binders and leaf variables in their scope.

```
linear_motzkin(N,E):-succ(N,N1),linear_motzkin(E,N,0,N1,0).
```

```
linear_motzkin(leaf,A,A,L,L).
linear_motzkin(l(E),A1,A2,L1,L3):-pred(L1,L2),linear_motzkin(E,A1,A2,L2,L3).
linear_motzkin(a(E,F),A1,A4,L1,L3):-pred(A1,A2),
  linear_motzkin(E,A2,A3,L1,L2),
  linear_motzkin(F,A3,A4,L2,L3).
```

Interestingly, they correspond to sequence **A024489** in [25], giving `1, 6, 70, 1050, 18018, 336336 ...`, which has a closed formula and originates from a geometric interpretation similar in terms of constraints on graph nodes, but it is not noted as related to lambda terms or their Motzkin skeletons.

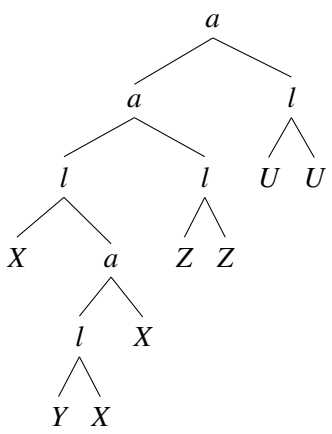
4.2 Decorating the Linear Skeletons

Next we decorate the Motzkin skeletons with lambda nodes and variables, while ensuring that we generate only closed terms. We push the lambda binder to a stack from which each variable will pick a binder having it in its scope. This ensures that we generate closed lambda terms.

The predicate `closed_almost_linear_term` initializes the counter `N` for application nodes. They are propagated down to `0` with variables `A1,..,An` through the recursive calls. The counter `N1 = N+1` for lambda binders is propagated with the variables L1,..,Ln. The stack of variables `Vs`, initially empty, makes available the lambda binders to the leaf variables. The stack grows when a lambda constructor `l/2` is introduced.

Example 4 *Almost linear lambda tree, having the same number of lambda nodes as leaves, but not paired two by two, 3 occurrences of X and one of Y being the exception).*

term: $(\lambda X.(\lambda Y.X\ X)\ \lambda Z.Z)\ \lambda U.U$ *tree:*




```
closed_almost_linear_term(N,E):-succ(N,N1),
  closed_almost_linear_term(E,N,0,N1,0,[]).

closed_almost_linear_term(X,A,A,L,L,Vs):-member(X,Vs).
closed_almost_linear_term(l(X,E),A1,A2,L1,L3,Vs):-pred(L1,L2),
```

```
  closed_almost_linear_term(E,A1,A2,L2,L3,[X|Vs]).
closed_almost_linear_term(a(E,F),A1,A4,L1,L3,Vs):-pred(A1,A2),
  closed_almost_linear_term(E,A2,A3,L1,L2,Vs),
  closed_almost_linear_term(F,A3,A4,L2,L3,Vs).
```

Note that linearity constraints are only half-way enforced so far: we only ensure that the number of lambda nodes is equal to the number of variables they bind.

4.3 Generating Closed Linear Lambda Terms

To ensure that terms are linear, we will mark each lambda binder when it reaches a variable. When exiting the expression in the scope of the binder we test that it has been indeed marked. Note, that without this test we would obtain *affine* lambda terms. The following predicates implement these operations.

```
bind_once(V,X):-var(V),V=v(X).

check_binding(V,X):-nonvar(V),V=v(X).
```

Otherwise, the predicate `linear_lambda_term` works like `closed_almost_linear_term`.

```
linear_lambda_term(N,E):-succ(N,N1),linear_lambda_term(E,N,0,N1,0,[]).

linear_lambda_term(X,A,A,L,L,Vs):-member(V,Vs),bind_once(V,X).
linear_lambda_term(l(X,E),A1,A2,L1,L3,Vs):-pred(L1,L2),
  linear_lambda_term(E,A1,A2,L2,L3,[V|Vs]),check_binding(V,X).
linear_lambda_term(a(E,F),A1,A4,L1,L3,Vs):-pred(A1,A2),
  linear_lambda_term(E,A2,A3,L1,L2,Vs),
  linear_lambda_term(F,A3,A4,L2,L3,Vs).
```

This gives us the sequence **A062980** in [25], starting as `1, 5, 60, 1105, 27120, 828250 ...`, confirming that they match results in [19, 30].

However, our goal is to generate unique theorems of a given size of linear implicational intuitionistic logic and that's on the other side of the Curry-Howard bridge. As otherwise the sizes of our lambda terms can be smaller or larger than the formulas and more than one term can correspond to the same formula, we will need to restrict ourselves to *lambda terms in normal form*, i.e., terms not having lambdas on the left side of application nodes that could be simplified using β-reduction.

4.4 Linear Normal Forms

Generation of normal forms relies on *neutral terms* that ensure that applications have as left nodes only variables or other application nodes. We ensure closedness and linearity constraints the same way as in the `linear_lambda_term` generator.

```
linear_normal_form(N,E):-succ(N,N1),linear_normal_form(E,N,0,N1,0,[]).

linear_normal_form(l(X,E),A1,A2,L1,L3,Vs):-pred(L1,L2),
  linear_normal_form(E,A1,A2,L2,L3,[V|Vs]),check_binding(V,X).
linear_normal_form(E,A1,A2,L1,L3,Vs):-
  linear_neutral_term(E,A1,A2,L1,L3,Vs).

linear_neutral_term(X,A,A,L,L,Vs):-member(V,Vs),bind_once(V,X).
linear_neutral_term(a(E,F),A1,A4,L1,L3,Vs):-pred(A1,A2),
```

```
  linear_neutral_term(E,A2,A3,L1,L2,Vs),
  linear_normal_form(F,A3,A4,L2,L3,Vs).
```

Again, this gives us sequence **A262301** in [25] starting with `1, 3, 26, 367, 7142, 176766, ...`, confirming the results in [19, 30]. Again, generating the lambda terms in normal form is essential as otherwise multiple terms that β-reduce to the normal form would correspond to the same formula.

4.5 Inferring the Types: Walking back Over the Curry-Howard Bridge

Finally, we will also annotate our lambda terms with their inferred types. This is quite easy as all linear terms are typable. Moreover, unlike in [30], unification does not require 'occurs check' as each lambda binds exactly one variable.

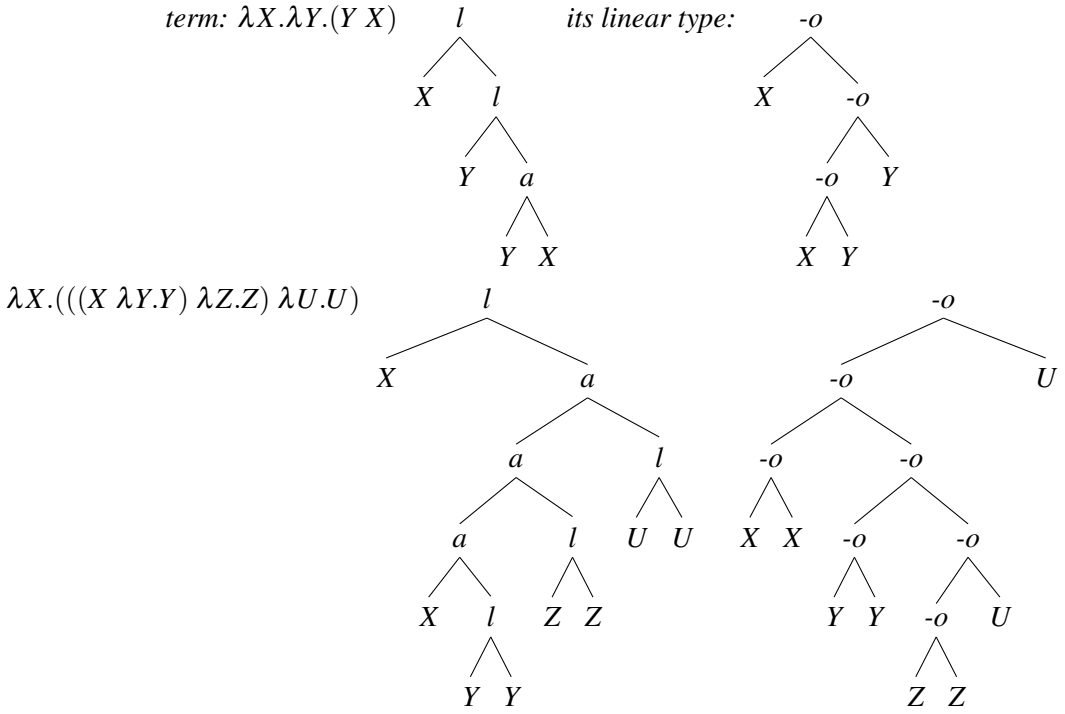
In fact, our type decoration mechanism can be seen as a simplified form of the usual Hindley-Milner type inference [15] used to derive the types of simply typed lambda terms, the main differences being that we do not need to use unification with occurs check and that we work exclusively on closed lambda terms.

```
linear_typed_normal_form(N,E,T):-succ(N,N1),
  linear_typed_normal_form(E,T,N,0,N1,0,[]).

linear_typed_normal_form(l(X,E),(S '-o' T),A1,A2,L1,L3,Vs):-pred(L1,L2),
  linear_typed_normal_form(E,T,A1,A2,L2,L3,[V:S|Vs]),
  check_binding(V,X).
linear_typed_normal_form(E,T,A1,A2,L1,L3,Vs):-
  linear_neutral_term(E,T,A1,A2,L1,L3,Vs).

linear_neutral_term(X,T,A,A,L,L,Vs):-member(V:TT,Vs),bind_once(V,X),T=TT.
linear_neutral_term(a(E,F),T,A1,A4,L1,L3,Vs):-pred(A1,A2),
  linear_neutral_term(E,(S '-o' T),A2,A3,L1,L2,Vs),
  linear_typed_normal_form(F,S,A3,A4,L2,L3,Vs).
```

The term counts, as expected, correspond to sequence **A262301** in [25], under the title *number of normal linear lambda terms of size n with no free variables*. Our Prolog program, runs the predicate `linear_typed_normal_form/3` in $O(N)$ space for terms of size N and it generates billions of terms and types in a few hours, e.g., `1, 3, 26, 367, 7142, 176766, 5304356, 186954535, 7566084686`. Note that in [19] a method to *count* linear terms analytically provides counts up to higher values of N for their super-exponentially growing terms, but the corresponding Haskell program[3] runs into memory problems after generating 5304356 terms, even if given 250GB of memory. The ability to go 3 orders of magnitude higher to 7566084686 actually generated terms (and even 4 if given longer time or by parallelizing the generators as in [3]), comes from the simple fact that Prolog recovers memory on backtracking after each generated term is written out to a file or used by another predicate. Thus we work in $O(N)$ space for terms and formulas of size N, with no need for a garbage collector invocation.

Example 5 *Normal forms and their corresponding linear types.*

[3] https://raw.githubusercontent.com/PierreLescanne/CountingGeneratingAfffineLinearClosedLambdaterms/master/LinearNormalFormSize0or1.hs



4.6 The Eureka Moment

After looking at the generated terms and their types we observe the following surprising facts:

- there are exactly two occurrences of each variable both in the theorems and their proof terms
- theorems and their proof terms have the same size, counted as number of internal nodes

Thus, we have solved the problem of generating all tautologies size N in the implicational fragment of propositional linear intuitionistic logic if the predicate `linear_typed_normal_form` *implements a generator of their proof-terms of size N, for which the tautologies can be seen each as their principal type*.

It turns out that there's a *size-preserving bijection between linear lambda terms in normal form and their principal types*. A proof of this follows immediately from [33] who attributes this observation to [21]. In [33] the bijection is proven by exhibiting a reversible transformation of oriented edges in the tree describing the linear lambda term in normal form, into corresponding oriented edges in the tree describing the linear implicational formula, acting as its principal type.

It follows that we have obtained a generator for all theorems of implicational linear intuitionistic propositional logic of a given size, as measured by the number of lollipops, without having to prove theorems, thus avoiding the need to call Turing-complete provers for linear logic or PSPACE-complete provers for propositional intuitionistic logic, simply by taking advantage of the existence of a size-preserving bijection *between theorems and their corresponding proof terms and the Curry-Howard correspondence*.

Clearly, this is a "Goldilocks" situation, that, in a way, points out the very special case that implicational formulas have in linear logic and equivalently, linear types have in type theory. We plan future work on extending this result to the case of propositional affine linear logic, also known to be decidable [18].

4.7 Applications

The dataset containing generated theorems and their proof-terms in postfix form (as well as their LaTeX tree representations marked as Prolog "%" comments) that we make available at `http://www.cse.unt.edu/~tarau/datasets/lltaut/` can be used for correctness, performance and scalability testing for linear logic theorem provers, in addition to the human made tests described in [22], as well as for providing similar tests for the Linear Haskell GHC compiler feature [5].

More importantly, the formula/proof-term pairs in the dataset are likely to be usable to test if deep-learning systems can perform a fairly interesting (and, in theory, learnable) theorem proving task: if trained via a `seq2seq` algorithm on encodings of theorems and their proof-terms, can the resulting model perform well on similar unseen formula/proof-term pairs? We have started work in that direction with promising initial results[4].

5 Discussion

Proofs of implications in intuitionistic logic have long been recognized as fundamental, as they correspond to (closed) programs in functional programming. The same cannot be said about "linear implications", as the tradition in linear logic tends to rewrite linear implications as multiplicative disjunctions ("pars" in proofnets) [12]. Proofnets, notwithstanding their logical appeal in simplifying proofs, are an "acquired taste", not shared by very many. One of our motivation for this research was using linear lambda-calculus [4] to investigate both logical proofs in Intuitionistic Linear Logic and, further down the line, to investigate translations between it and intuitionistic logic proofs. Girard produced two such translations in his original paper on linear logic [12]. Applying these translations to well-known proofs in intuitionistic logic (as for instance those described in the classic monograph [17]) was a main motivation of [22] leading to our initial interest for generating a benchmark of intuitionistic linear logic proofs, complementing the ones described in [22].

But the hope for intuitionistic Linear Logic has always been to discover where duplication of hypotheses and, respectively, their erasure is safe, as far as the meaning of the proofs/programs is concerned. Our long term goal is to improve on the already known translations of intuitionistic logic into intuitionistic linear logic. For that, we need to know more about the universe of existing linear proofs, like how many there are, their shapes, invariant properties, etc. Much work has already been done in this direction, see for example the work on "optimal reductions" [13] and on "linear decorations"[24]. However, it seems fair to suggest that this work has not produced all the expected benefits, yet. The work described here is supposed to help with both of these aims.

6 Related Work

The classic reference for lambda calculus is [1]. The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [14]. Distribution and density properties of random lambda terms are described in [8]. Asymptotic density properties of simple types (corresponding to tautologies in implicational intuitionistic logic) have been studied in [11] with the surprising result that "almost all" classical tautologies are also intuitionistic ones.

[4] Our (successful!) experiments with training Recurrent Neural Networks using our implicational linear logic theorem dataset are available at: `https://github.com/ptarau/neuralgs` .

The generation and counting of affine and linear lambda terms is extensively covered in [19], where, by using techniques from analytic combinatorics, much higher limits for *counting* (but not *generating*) linear lambda terms are derived, using efficient recurrence relations. By contrast, our focus here is on generation. While producing also the corresponding tautologies, our Prolog-based generators had actually go 3 orders of magnitude further than the Haskell program described in [19].

We have used extensively Prolog as a meta-language for the study of combinatorial and computational properties of lambda terms in papers like [2, 28] covering different families of terms and properties. The idea of using types inferred for lambda terms as formulas for testing theorem provers originates in [31]. The current paper extends this line of research to linear logic, specifically to the implicational fragment of linear intuitionistic propositional logic.

The closest work that we have used as starting point for the intuitionistic logic prover is [10] describing the **LJT** calculus. Asymptotic behavior of linear and affine lambda terms, in relation with the BCK and BCI combinator systems, as well as bijections to combinatorial maps are studied in [6]. In [7] analytic models are used to solve the problem of the asymptotic density of closable and uniquely closable skeletons, Motzkin trees that predetermine existence and uniqueness of the closed lambda terms decorating them.

The bijection between linear lambda terms in normal form and their principal types, first proven in [21] and explained also in terms of a geometric interpretation in [33], has been instrumental in deriving the optimal final form of our term/formula pair generator.

7 Conclusions

We have derived declaratively novel algorithms for the combinatorial generation of theorems in linear logic and their proof-terms. The ability to declaratively encode constraints on the structure and the content of Prolog terms has enabled us to produce a generator for billions of theorems and their proof-terms in an important sublanguage of linear logic and to collect them into a a dataset usable for testing linear logic provers and training deep-learning systems for theorem proving, an emerging new task in machine learning. By contrast to functional language implementations our algorithms fully recover space on backtracking, without even triggering Prolog's garbage collection. This makes Prolog the language of choice for work exploring synergies between combinatorial generation, type inference and theorem proving.

Acknowledgments

We thank the anonymous reviewers of ICLP'2020 for their constructive comments and suggestions.

References

[1] H. P. Barendregt (1984): *The Lambda Calculus Its Syntax and Semantics*, revised edition. 103, North Holland.

[2] Maciej Bendkowski, Katarzyna Grygiel & Paul Tarau (2017): *Boltzmann Samplers for Closed Simply-Typed Lambda Terms*. In Yuliya Lierler & Walid Taha, editors: *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, *Lecture Notes in Computer Science* 10137, Springer, pp. 120–135, doi:10.1007/978-3-319-51676-9. , Best student paper award.

[3] Maciej Bendkowski, Katarzyna Grygiel & Paul Tarau (2018): *Random generation of closed simply typed λ-terms: A synergy between logic programming and Boltzmann samplers*. *TPLP* 18(1), pp. 97–119. Available at https://doi.org/10.1017/S147106841700045X.

[4] Nick Benton, Gavin Bierman, Valeria De Paiva & Martin Hyland (1993): *A term calculus for intuitionistic linear logic*. In: *International Conference on Typed Lambda Calculi and Applications*, Springer, pp. 75–90, doi:10.1007/BFb0037099.

[5] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2018): *Linear Haskell: practical linearity in a higher-order polymorphic language*. *Proc. ACM Program. Lang.* 2(POPL), pp. 5:1–5:29, doi:10.1145/3158093.

[6] O. Bodini, D. Gardy & A. Jacquot (2013): *Asymptotics and random sampling for BCI and BCK lambda terms*. *Theoretical Computer Science* 502, pp. 227 – 238, doi:10.1016/j.tcs.2013.01.008.

[7] O. Bodini & P. Tarau (2017): *On Uniquely Closable and Uniquely Typable Skeletons of Lambda Terms*. *CoRR* abs/1709.04302. Available at http://arxiv.org/abs/1709.04302, doi:10.1007/978-3-319-94460-9_15.

[8] René David, Christophe Raffalli, Guillaume Theyssier, Katarzyna Grygiel, Jakub Kozik & Marek Zaionc (2009): *Some properties of random lambda terms*. *Logical Methods in Computer Science* 9(1).

[9] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li & Denny Zhou (2019): *Neural Logic Machines*. Available at https://openreview.net/pdf?id=B1xY-hRctX.

[10] Roy Dyckhoff (1992): *Contraction-free sequent calculi for intuitionistic logic*. *Journal of Symbolic Logic* 57(3), p. 795807, doi:10.2307/2275431.

[11] Antoine Genitrini, Jakub Kozik & Marek Zaionc (2007): *Intuitionistic vs. Classical Tautologies, Quantitative Comparison*. In Marino Miculan, Ivan Scagnetto & Furio Honsell, editors: *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, *Lecture Notes in Computer Science* 4941, Springer, pp. 100–109, doi:10.1007/978-3-540-68103-8_7.

[12] Jean-Yves Girard (1987): *Linear logic*. *Theoretical computer science* 50(1), pp. 1–101, doi:10.1016/0304-3975(87)90045-4.

[13] Georges Gonthier, Martín Abadi & Jean-Jacques Lévy (1992): *The geometry of optimal lambda reduction*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 15–26, doi:10.1145/143165.143172.

[14] Katarzyna Grygiel & Pierre Lescanne (2013): *Counting and generating lambda terms*. *J. Funct. Program.* 23(5), pp. 594–628, doi:10.1017/S0956796813000178.

[15] J. Roger Hindley (1997): *Basic Simple Type Theory*. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9780511608865.

[16] W.A. Howard (1980): *The Formulae-as-types Notion of Construction*. In J.P. Seldin & J.R. Hindley, editors: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London, pp. 479–490.

[17] S.C. Kleene (1952): *Introduction to Metamathematics*. Bibliotheca Mathematica, Wolters-Noordhoff. Available at https://books.google.com/books?id=O28-AQAAIAAJ.

[18] A.P. Kopylov (1995): *Decidability of Linear Affine Logic*. In Dexter Kozen, editor: *Proceedings of the Tenth Annual IEEE Symp. on Logic in Computer Science, LICS 1995*, IEEE Computer Society Press, pp. 496–504, doi:10.1006/inco.1999.2834.

[19] Pierre Lescanne (2018): *Quantitative Aspects of Linear and Affine Closed Lambda Terms*. *ACM Trans. Comput. Log.* 19(2), pp. 9:1–9:18, doi:10.1145/3173547.

[20] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester & Luc De Raedt (2018): *DeepProbLog: Neural Probabilistic Logic Programming*. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi & R. Garnett, editors: *Advances in Neural Information Processing Systems 31*, Curran Associates, Inc., pp. 3749–3759. Available at http://papers.nips.cc/paper/7632-deepproblog-neural-probabilistic-logic-programming.pdf.

[21] Grigorii E. Mints (1992): *Closed categories and the theory of proofs*. In: *Selected Papers in Proof Theory*, Bibliopolis, doi:10.1007/BF01404107.

[22] Carlos Olarte, Valeria de Paiva, Elaine Pimentel & Giselle Reis (2018): *The ILLTP Library for Intuitionistic Linear Logic*. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva & Lorenzo Tortora de Falco, editors: *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*, *EPTCS* 292, pp. 118–132, doi:10.4204/EPTCS.292.7.

[23] Tim Rocktäschel & Sebastian Riedel (2017): *End-to-end Differentiable Proving*. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan & R. Garnett, editors: *Advances in Neural Information Processing Systems 30*, Curran Associates, Inc., pp. 3788–3800. Available at `http://papers.nips.cc/paper/6969-end-to-end-differentiable-proving.pdf`.

[24] H Schellinx (1994): *The noble art of linear decorating. ILLC Dissertation Series, 1994-1*. Institute for Language, Logic and Computation, University of Amsterdam.

[25] N. J. A. Sloane (2020): *The On-Line Encyclopedia of Integer Sequences.* Published electronically at https://oeis.org/.

[26] R P Stanley (1986): *Enumerative Combinatorics*. Wadsworth Publ. Co., Belmont, CA, USA.

[27] Richard Statman (1979): *Intuitionistic Propositional Logic is Polynomial-Space Complete*. *Theor. Comput. Sci.* 9, pp. 67–72, doi:10.1016/0304-3975(79)90006-9.

[28] Paul Tarau (2015): *On a Uniform Representation of Combinators, Arithmetic, Lambda Terms and Types*. In Elvira Albert, editor: *PPDP'15: Proceedings of the 17th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ACM, New York, NY, USA, pp. 244–255, doi:10.1145/2790449.2790526.

[29] Paul Tarau (2017): *A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms*. In Manuel V Hermenegildo & Pedro Lopez-Garcia, editors: *Logic-Based Program Synthesis and Transformation: 26th International Symposium, LOPSTR 2016, Edinburgh, UK, Revised Selected Papers*, Springer LNCS, volume 10184, pp. 240–255, doi:10.1007/978-3-319-63139-4_14. , Best paper award.

[30] Paul Tarau (2018): *On k-colored Lambda Terms and their Skeletons*. In Francesco Calimeri, Kevin W. Hamlen & Nicola Leone, editors: *Practical Aspects of Declarative Languages - 20th International Symposium, PADL 2018, Los Angeles, CA, USA, January 8-9, 2018, Proceedings*, *Lecture Notes in Computer Science* 10702, Springer, pp. 116–131, doi:10.1007/978-3-319-73305-0_8.

[31] Paul Tarau (2019): *A Combinatorial Testing Framework for Intuitionistic Propositional Theorem Provers*. In José Júlio Alferes & Moa Johansson, editors: *Practical Aspects of Declarative Languages - 21th International Symposium, PADL 2019, Lisbon, Portugal, January 14-15, 2019, Proceedings*, *Lecture Notes in Computer Science* 11372, Springer, pp. 115–132, doi:10.1007/978-3-030-05998-9_8.

[32] Philip Wadler (2015): *Propositions as types*. *Commun. ACM* 58, pp. 75–84, doi:10.1145/2699407.

[33] Noam Zeilberger (2015): *Balanced polymorphism and linear lambda calculus, talk at TYPES'15*. `http://noamz.org/papers/linprin.pdf`.

A The Implicational Intuitionistic Theorem Prover

A.1 The LJT/G4ip Calculus

Motivated by problems related to loop avoidance when implementing Gentzen's **LJ** calculus, Roy Dyckhoff [10] introduces the following rules for his `LJT` calculus[5].

LJT_1 : $\dfrac{}{A,\Gamma \vdash A}$

[5] Also called the G4ip calculus. Restricted here to the implicational fragment.

LJT_2 : $\frac{A,\Gamma \vdash B}{\Gamma \vdash A \to B}$

LJT_3 : $\frac{B,A,\Gamma \vdash G}{A \to B,A,\Gamma \vdash G}$

LJT_4 : $\frac{D \to B,\Gamma \vdash C \to D \quad B,\Gamma \vdash G}{(C \to D) \to B,\Gamma \vdash G}$

The rules work with the context Γ being either a multiset or a set.

A.2 Extracting Proof Terms

We refer to [31] for the derivation steps leading from this calculus to Prolog-based theorem provers implementing it. We will focus here on extracting proof terms from a prover adapted to cover the implicational fragment of propositional linear intuitionistic logic.

Extracting the *proof terms* (lambda terms having the formulas we prove as types) is achieved by decorating the code with application nodes `a/2`, lambda nodes `l/2` (with first argument a logic variable) and leaf nodes (labeled with logic variables, same as the identically named ones in the first argument of the corresponding `l/2` nodes).

The fact that this is essentially the inverse of a type inference algorithm (e.g., the Prolog-based one in [29]) points out how the decoration mechanism works.

```
prove_ipc(T,ProofTerm):-prove_ipc(ProofTerm,T,[]).
prove_ipc(X,A,Vs):-memberchk(X:A,Vs),!. % leaf variable
prove_ipc(l(X,E),(A '-o' B),Vs):-!,prove_ipc(E,B,[X:A|Vs]).  % lambda term
prove_ipc(E,G,Vs1):-
  member(_:V,Vs1),head_of(V,G),!, % fast fail if non-tautology
  select(S:(A '-o' B),Vs1,Vs2),   % source of application
  prove_ipc_imp(T,A,B,Vs2),!,       % target of application
  prove_ipc(E,G,[a(S,T):B|Vs2]).  % application

prove_ipc_imp(l(X,E),(C '-o' D),B,Vs):-!,prove_ipc(E,(C '-o' D),[X:(D '-o' B)|Vs]).
prove_ipc_imp(E,A,_,Vs):-memberchk(E:A,Vs).

% optimization for quicker failure
head_of(_ '-o' B,G):-!,head_of(B,G).
head_of(G,G).
```

Thus, lambda nodes decorate *implication introductions* and application nodes decorate *modus ponens* reductions in the corresponding calculus. Note that the two clauses of `prove_ipc_imp` provide the target node T. When seen from the type inference side, T is the type resulting from cancelling the source type S and the application type $S \to T$.