15-819K: Logic Programming

Lecture 15

Resource Management

Frank Pfenning

October 17, 2006

In this lecture we address the resource management problem in linear logic programming. We also give some small but instructive examples of linear logic programming, supplementing the earlier peg solitaire code.

15.1 Input/Output Interpretation of Resources

Reconsider the rule for simultaneous conjunction as a goal.

$$\frac{\Delta = (\Delta_1, \Delta_2) \quad \Delta_1 \Vdash G_1 \quad \Delta_2 \Vdash G_2}{\Delta \Vdash G_1 \otimes G_2} \otimes R$$

The difficulty in using this rule in proof search is that, as written, we have to "guess" the right way to split the resources Δ into two. Clearly, enumerating all possible ways to split Δ will be very inefficient, and also difficult for the programmer to predict.

Instead, we pass all resources to the goal G_1 and then pass the ones that were not consumed in the derivation of G_1 to G_2. We write

$$\Delta_I \setminus \Delta_O \Vdash G$$

where Δ_I is the input context and Δ_O is the output context generated by the proof search for G. The invariant we preserve is that

$$\Delta_I \setminus \Delta_O \Vdash G \text{ iff } \Delta_I - \Delta_O \Vdash G$$

where $\Delta_I - \Delta_O$ subtracts the resources in Δ_O from the resources in Δ_I. It is convenient to keep Δ_I and Δ_O ordered, so that this difference can be computed component by component (see below).

Now the rule for simultaneous conjunction is

$$\frac{\Delta_I \setminus \Delta_M \Vdash G_1 \quad \Delta_M \setminus \Delta_O \Vdash G_2}{\Delta_I \setminus \Delta_O \Vdash G_1 \otimes G_2} \otimes R.$$

It is easy to verify that the above invariant holds for this rule. More formally, this would be part of a soundness and completeness proof for the input/output interpretation of resources.

15.2 Slack Resources

The simple input/output interpretation for resources breaks down for consumptive truth ($\top$). Recall the rule

$$\frac{}{\Delta \Vdash \top} \top R$$

which translates to

$$\frac{\Delta_I \supseteq \Delta_O}{\Delta_I \setminus \Delta_O \Vdash \top} \top R$$

because $\top$ *may* consume any of its input but does *not need* to. Now we are back at the original problem, since we certainly do not want to enumerate all possible subsets of Δ_I blindly.

Instead, we pass on all the input resources, but also a flag to indicate that all of these resources could have been consumed. That means if they are left over at the end, we can succeed instead of having to fail. We write the judgment as

$$\Delta_I \setminus \Delta_O \Vdash_v G$$

where $v = 0$ means G used exactly the resources in $\Delta_I - \Delta_O$, while $v = 1$ means G could also have consumed additional resources from Δ_O. Now our invariants are:

i. $\Delta_I \setminus \Delta_O \Vdash_0 G$ iff $\Delta_I - \Delta_O \Vdash G$

ii. $\Delta_I \setminus \Delta_O \Vdash_1 G$ iff $\Delta_I - \Delta_O, \Delta' \Vdash G$ for any $\Delta_O \supseteq \Delta'$.

The right rule for $\top$ with slack is just

$$\frac{}{\Delta_I \setminus \Delta_I \Vdash_1 \top} \top R.$$

In contrast, the rule for equality which requires the linear context to be empty is

$$\frac{}{\Delta_I \setminus \Delta_I \Vdash_0 t \doteq t} \doteq R.$$

As far as resources are concerned, the only difference is whether slack is allowed ($v = 1$) or not ($v = 0$).

We now briefly return to simultaneous conjunction. There is slack in the deduction for $G_1 \otimes G_2$ if there is slack on either side: any remaining resources could be pushed up into either of the two subderivations as long as there is slack in at least one.

$$\frac{\Delta_I \setminus \Delta_M \Vdash_v G_1 \quad \Delta_M \setminus \Delta_O \Vdash_w G_2}{\Delta_I \setminus \Delta_O \Vdash_{v \vee w} G_1 \otimes G_2} \otimes R.$$

Here $v \vee w$ is the usual Boolean disjunction between the two flags: it is 0 if both disjuncts are 0 and 1 otherwise.

15.3 Strict Resources

Unfortunately, there is still an issue in that resource management does not take into account all information it should. There are examples in the literature [2], but they are not particularly natural. For an informal explanation, consider the overall query $\Delta \Vdash G$. We run this as $\Delta \setminus \Delta_O \Vdash_v G$, where Δ_O and v are returned. We then have to check that all input has indeed be consumed by verifying that Δ_O is empty. If Δ_O is not empty, we have to fail this attempt and backtrack.

We would like to achieve that we fail as soon as possible when no proof can exist due to resource management issues. In the present system we may sometimes run to completion only to note at that point that we failed to consume all resources. We can avoid this issue by introducing yet one more distinction into our resource management judgment by separating out *strict resources*. Unlike Δ_I, which represents resources which *may* be used, Ξ represent resources which *must* be used in a proof.

$$\Xi; \Delta_I \setminus \Delta_O \Vdash_v G$$

The invariant does not get significantly more complicated.

i. $\Xi; \Delta_I \setminus \Delta_O \Vdash_0 G$ iff $\Xi, \Delta_I - \Delta_O \Vdash G$

ii. $\Xi; \Delta_I \setminus \Delta_O \Vdash_1 G$ iff $\Xi, \Delta_I - \Delta_O, \Delta' \Vdash G$ for all $\Delta_O \supseteq \Delta'$.

When reading the rules please remember that no resource in Ξ is ever passed on: it *must* be consumed in the proof of $\Xi; \Delta_I \setminus \Delta_O \Vdash_v G$. The unrestricted context Γ remains implicit and is always passed from conclusion to all premisses.

We will enforce as an additional invariant that input and output context have the same length and structure, except that some inputs have been consumed. Such consumed resources are noted as underscores '_' in a context. We use $\Delta_I \supseteq \Delta_O$ and $\Delta_I - \Delta_O$ with the following definitions:

$$\frac{}{(\cdot) \supseteq (\cdot)} \qquad \frac{\Delta_I \supseteq \Delta_O}{(\Delta_I, \_) \supseteq (\Delta_O, \_)} \qquad \frac{\Delta_I \supseteq \Delta_O}{(\Delta_I, A) \supseteq (\Delta_O, \_)} \qquad \frac{\Delta_I \supseteq \Delta_O}{(\Delta_I, A) \supseteq (\Delta_O, A)}$$

and, for $\Delta_I \supseteq \Delta_O$,

$$\frac{}{(\cdot) - (\cdot) = (\cdot)} \qquad \frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, \_) - (\Delta_O, \_) = (\Delta, \_)}$$

$$\frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, A) - (\Delta_O, \_) = (\Delta, A)} \qquad \frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, A) - (\Delta_O, A) = (\Delta, \_)}$$

Atomic Goals. When a goal is atomic, we focus on an assumption, residuate, and solve the resulting subgoal. There are three possibilities for using an assumption: from Γ, from Ξ, or from Δ_I. Because we use residuation, resource management is straightforward here: we just have to replace the assumption with the token '_' to indicate that the resource has been consumed.

$$\frac{D \in \Gamma \quad D \Vdash P > G \quad \Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v P} \text{ resid!}$$

$$\frac{D \Vdash P > G \quad \Gamma; (\Xi_1, \_, \Xi_2); \Delta_I \setminus \Delta_O \Vdash_v G}{\Gamma; (\Xi_1, D, \Xi_2); \Delta_I \setminus \Delta_O \Vdash_v P} \text{ resid}_1$$

$$\frac{D \Vdash P > G \quad \Gamma; \Xi; (\Delta_I', \_, \Delta_I'') \setminus \Delta_O \Vdash_v G}{\Gamma; \Xi; (\Delta_I', D, \Delta_I'') \setminus \Delta_O \Vdash_v P} \text{ resid}_2$$

Asynchronous Multiplicative Connective. There is only one multiplicative asynchronous connective, $D \multimap G$ which introduces a new linear assumption. Since D *must* be consumed in the proof of G, we add it to the

strict context Ξ.

$$\frac{(\Xi, D); \Delta_I \setminus \Delta_O \Vdash_v G}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v D \multimap G} \multimap R$$

Synchronous Multiplicative Connectives. In the linear hereditary Harrop fragment as we have constructed it here, there are only two multiplicative connectives that are synchronous as goals: equality and simultaneous conjunction. The multiplicative unit **1** is equivalent to $P \doteq P$ and does not explicitly arise in residuation. For equality, we just need to check that Ξ is empty and pass on all input resources to the output, indicating that there is no slack ($v = 0$).

$$\frac{\Xi = (\_, \ldots, \_)}{\Xi; \Delta_I \setminus \Delta_I \Vdash_0 P \doteq P} \doteq R$$

For simultaneous conjunction, we distinguish two cases, depending on whether the first subgoal has slack. Either way, we turn all strict resources from Ξ into lax resources for the first subgoal, since the second subgoal is waiting, and may potentially consume some of the formulas in Ξ that remain unconsumed in the first subgoal.

$$\frac{.\,; \Xi, \Delta_I \setminus \Xi', \Delta'_I \Vdash_0 G_1 \quad \Xi'; \Delta'_I \setminus \Delta_O \Vdash_v G_2 \quad (\Xi \supseteq \Xi')}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \otimes G_2} \otimes R_0$$

If the first subgoal is slack, then it could consume the leftover resources in Ξ', so they do not necessarily need to be consumed in the second subgoal. Originally strict resources that remain after the second subgoal are then dropped, noting that the first subgoal must have (implicitly) consumed them.

$$\frac{.\,; \Xi, \Delta_I \setminus \Xi', \Delta'_I \Vdash_1 G_1 \quad .\,; \Xi', \Delta'_I \setminus \Xi'', \Delta_O \Vdash_* G_2 \quad (\Xi \supseteq \Xi' \supseteq \Xi'')}{\Xi; \Delta_I \setminus \Delta_O \Vdash_1 G_1 \otimes G_2} \otimes R_1$$

It does not matter whether the second subgoal is strict or lax, since the disjunction is already known to be 1. We indicate this with an asterisk '$*$'.

Asynchronous Additive Connectives. The additive connectives that are asynchronous as goals are alternative conjunction ($G_1 \,\&\, G_2$) and consumptive truth ($\top$). First $\top$, which motivated the slack indicator v. It consumes all of Ξ and passes the remaining inputs on without consuming them.

$$\frac{}{\Xi; \Delta_I \setminus \Delta_I \Vdash_1 \top} \top R$$

For alternative conjunction, we distinguish two subcases, depending on whether the first subgoal turns out to have slack or not. If not, then the second subgoal must consume exactly what the first subgoal consumed, namely Ξ and $\Delta_I - \Delta_O$. We therefore add this to the strict context. The lax context is empty, and we do not need to check the output (it must also be empty, since it is a subcontext of the empty context). Again, we indicate this with a '$*$' to denote an output we ignore.

$$\frac{\Xi; \Delta_I \setminus \Delta_O \Vdash_0 G_1 \quad \Xi, \Delta_I - \Delta_O; \cdot \setminus * \Vdash_* G_2}{\Xi; \Delta_I \setminus \Delta_O \Vdash_0 G_1 \,\&\, G_2}\ \&R_0$$

If the first subgoal has slack, with still must consume everything that was consumed in the first subgoal. In addition, we may consume anything that was left.

$$\frac{\Xi; \Delta_I \setminus \Delta_M \Vdash_1 G_1 \quad \Xi, \Delta_I - \Delta_M; \Delta_M \setminus \Delta_O \Vdash_v G_2}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \,\&\, G_2}\ \&R_1$$

Synchronous Additive Connectives. Disjunction is easy, because it involves a choice among alternatives, but not resources, which are just passed on.

$$\frac{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \oplus G_2}\ \oplus R_1 \qquad \frac{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_2}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \oplus G_2}\ \oplus R_2$$

Falsehood is even easier, because it represents failure and therefore has no right rule.

$$\frac{\text{no rule } \mathbf{0}R}{\Xi; \Delta_I \setminus * \Vdash_* \mathbf{0}}$$

Exponential Connectives. Unrestricted implication is quite simple, since we just add the new assumption to the unrestricted context.

$$\frac{\Gamma, D; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v D \supset G}\ \supset R$$

The (asymmetric) exponential conjunction passes all resources to the first subgoal, since the second cannot use any resources. We do not care if the exponential subgoal is strict or lax, since it does not receive or return any resources anyway.

$$\frac{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \quad \Gamma; \cdot; \cdot \setminus * \Vdash_* G_2}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \otimes! G_2}\ \otimes! R$$

Quantifiers. Quantifiers are resource neutral.

$$\frac{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G \quad x \notin \mathrm{FV}(\Gamma; \Xi; \Delta_I)}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v \forall x.\, G}\ \forall R$$

$$\frac{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G(t/x)}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v \exists x.\, G}\ \exists R$$

This completes the connectives for the linear hereditary Harrop formulas. The proof that these are sound and complete amounts to showing the invariants stated at the beginning of this section. A crucial lemma states that resources can be added to the lax input context without affecting provability.

If $\Xi; \Delta_I \setminus \Delta_O \Vdash_v G$ *then* $\Xi; (\Delta_I, \Delta') \setminus (\Delta_O, \Delta') \Vdash_v G$ *for all* Δ'.

This provides a measure of modularity to proofs using consumable resources, at least as far as the existence of proofs is concerned. During proof search, however, it is clear that Δ' could interfere with with the proof if added to the input.

At the top level, we solve $\Delta \Vdash G$ by invoking $\Delta; \cdot \setminus * \Vdash_* G$. We do not need to check the output context (which will be empty) or the slack indicator, because Δ is passed in as strict context.

15.4 Sample Program: Permutation

To illustrate linear logic programming we give a few small programs. The first computes all permutations of a list. It does so by adding the elements to the linear context and then reading them out. Since the linear context is not ordered, this allows all permutations.

```
perm([X|Xs],Ys) ⟜ (elem(X) ⊸ perm(Xs,Ys)).
perm([],[Y|Ys]) ⟜ elem(Y) ⊗ perm([],Ys).
perm([],[]).
```

The last clause can only apply if the context is empty, so any order of these clauses will work. However, putting the third before the second will cause more backtracking especially if permutation is embedded multiplicatively in a larger program.

15.5 Sample Program: Depth-First Search

Imperative algorithms for depth-first search mark nodes that have been visited to prevent looping. We can model this in a linear logic program by starting with a linear assumption `node(`x`)` for every node x and consuming this assumption when visiting a node. This means that a node cannot be used more than once, preventing looping.

We assume a predicate `edge(`x`,`y`)` which holds whenever there is a directed edge from x to y.

```
dfs(X,Y) ⟜ edge(X,Y).
dfs(X,Y) ⟜ edge(X,Z) ⊗ node(Z) ⊗ dfs(Z,Y).
```

This by itself is note quite enough because not all nodes might be visited. We can allow this with the following top-level call

```
path(X,Y) ⟜ node(X) ⊗ dfs(X,Y) ⊗ ⊤.
```

15.6 Sample Program: Stateful Queues

In Lecture 11 we have seen how to implement a queue with a difference list, but we had to pass the queue around as an argument to any predicate wanting to use it. We can also maintain the queue in the linear context. Recall that we used a list of instructions `enq(`x`)` and `deq(`x`)`, and that at the end the queue must be empty.

```
queue(Is) ⟜ (front(B) ⊗ back(B) ⊸ q(Is)).

q([enq(X)|Is]) ⟜ back([X|B]) ⊗ (back(B) ⊸ q(Is)).
q([deq(X)|Is]) ⟜ front([X|Xs]) ⊗ (front(Xs) ⊸ q(Is)).
q([]) ⟜ front([]) ⊗ back([]).
```

In this version, the dequeuing may borrow against future enqueue operations (see Exercise 15.2).

It is tempting to think we might use the linear context itself as a kind of queue, similar to the permutation program, but using cut '!' to avoid getting all solution. This actually does not work, since the linear context is maintained as a stack: most recently made assumptions are tried first.

15.7 Historical Notes

Resource management for linear logic programming was first considered by Hodas and Miller in the design of Lolli [7, 8]. The original design underestimated the importance of consumptive truth, which was later repaired by adding the slack indicator [5] and then the strict context [1, 2]. Primitive operations on the contexts are still quite expensive, which was addressed subsequently through so-called *tag frames* which implement the context management system presented here in an efficient way [6, 9].

An alternative approach to resource management is to use Boolean constraints to connect resources in different branches of the proof tree, developed by Harland and Pym [3, 4]. This is more general, because one is not committed to depth-first search, but also potentially more expensive.

An implementation of the linear logic programming language Lolli in Standard ML can be found at `http://www.cs.cmu.edu/~fp/lolli`. The code presented in this course, additional puzzles, a propositional theorem prover, and more example can be found in the distribution.

15.8 Exercises

Exercise 15.1 *Prove carefully that the* `perm` *predicate does implement permutation when invoked in the empty context. You will need to generalize this statement to account for intermediate states in the computation.*

Exercise 15.2 *Modify the stateful queue program so it fails if an element is dequeued before it is enqueued.*

Exercise 15.3 *Give an implementation of a double-ended queue in linear logic programming, where elements can be both enqueued and dequeued at both ends.*

Exercise 15.4 *Sometimes, linear logic appears excessively pedantic in that all resource must be used. In* affine logic *resources may be used at most once. Develop the connectives of affine logic, its asynchronous fragment, residuation, and resource management for affine logic. Discuss the simplifications in comparison with linear logic (if any).*

Exercise 15.5 *Philosophers have developed* relevance logic *in order to capture that in a proof of A implies B, some use of A should be made in the proof of B.* Strict logic *is a variant of relevance logic where we think of strict assumptions as resources which must be used at least once in a proof. Develop the connectives of strict logic, its asynchronous fragment, residuation, and resource management for strict logic. Discuss the simplifications in comparison with linear logic (if any).*

Exercise 15.6 *In the resource managment system we employed a relatively high-level logical system, without goal stack, failure continuation, or explicit unification. Extend the abstract machine which uses these mechanisms by adding resource management as given in this lecture.*

Because we can make linear or unrestricted assumptions in the course of search, not all information associated with a predicate symbol p is static, in the global program. This means the rule for atomic goals must change. You should fix the order in which assumptions are tried by using most recently made assumptions first and fall back on the static program when all dynamic possibilities have been exhausted.

15.9 References

[1] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, March 1996. Springer-Verlag LNAI 1050.

[2] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

[3] James Harland and David J. Pym. Resource distribution via Boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer Verlag LNCS 1249.

[4] James Harland and David J. Pym. Resource distribution via Boolean constraints. *ACM Transactions on Computational Logic*, 4(1):56–90, 2003.

[5] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[6] Joshua S. Hodas, Pablo López, Jeffrey Polakow, Lubomira Stoilova, and Ernesto Pimentel. A tag-frame system of resource management for proof search in linear-logic programming. In J. Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL'02)*, pages 167–182, Edinburgh, Scotland, September 2002. Springer Verlag LNCS 2471.

[7] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 32–42, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[8] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[9] Pablo López and Jeffrey Polakow. Implementing efficient resource management for linear logic programming. In Franz Baader and Andrei Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, pages 528–543, Montevideo, Uruguay, March 2005. Springer Verlag LNCS 3452.