Programming in Lygon: a system demonstration

James Harland[1] David Pym[2] Michael Winikoff[3]

[1] RMIT, GPO Box 2476V, Melbourne 3001, Australia
[2] Queen Mary & Westfield College, University of London, UK
[3] University of Melbourne, Parkville 3052, Australia

Introduction. The application of algebraic and logical techniques to programming is facilitated by the use of *declarative* languages. Such languages have simple semantics and programs in them are easy to reason about. However, declarative languages are often weak at expressing such important notions as *resource-allocation*, *state-manipulation* and *concurrency*.

Girard's *linear logic*,[4] provides an elegant framework for expressing these notions. For this and other reasons there has been significant interest recently in the design of logic programming languages based on linear logic.

Our demonstration is of the linear logic programming language *Lygon*. Intuitively, Lygon can be thought of as an extension of Prolog with (i) a linear context, and (ii) logical connectives manipulating this context.

A paper describing Lygon's design, together with example programs illustrating its expressive power in general, and its ability to handle state, resources and concurrency in particular, can be found elsewhere in this volume.[5]

On the interpreter. The operational rules of linear sequent calculus divide into *additive* and *multiplicative* groups. Typical of these groups, respectively, are the rules for $\&$ and $\otimes$:

$$\& \frac{\vdash \phi_1, \Delta \quad \vdash \phi_2, \Delta}{\vdash \phi_1 \& \phi_2, \Delta}; \qquad \otimes \frac{\vdash \phi_1, \Delta_1 \quad \vdash \phi_2, \Delta_2}{\vdash \phi_1 \otimes \phi_2, \Delta_1, \Delta_2}.$$

For simplicity, the Δs here should be read as the linear context only. (Note that the interpreter is based on one-sided sequents.)

The interpreter must read these rules as reduction operators, from conclusion to premisses. Under this interpretation, the conjunction `&` (ASCII for $\&$, pronounced "with") makes a separate copy of the linear context for each of its two arguments. Thus, for example, given the context (*i.e.*, the negation of Δ) `a(1)` and program `a(2)`, part of the unbounded context, the goal `a(X) & a(Y)` has the single solution `X=1,Y=1`. Note that neither variable can be 2 since each call to `a` gets a copy of the context and the formula in the context must be used.

The conjunction `*` (ASCII for $\otimes$, pronounced "tensor") *divides* the context between its arguments. Thus, given the same program and context, the goal `a(X) * a(Y)` succeeds with the two possible solutions being `X=1, Y=2` and `X=2, Y=1`. Note that both variables cannot be 1 since there is only a single copy of `a(1)` in the context and we should need two copies (one for each call to `a`). Both variables cannot be 2 since formulae in the linear context cannot be arbitrarily deleted.

If the linear context is not used, then `&` and `*` both reduce to Prolog's conjunction (`,`).

The division of the context associated with a tensor product is, at the level of the logical definition of Lygon, *non-deterministic*. However, the interpreter must determine this dynamically. The solution adopted in the current implementation (see below) is a non-trivial development of the *input/output model of resource consumption* first introduced by Hodas and Miller.[6] The basic idea is that all resources are passed first to the left branch; upon completion of the left branch, the remaining resources are passed to the right branch. The (substantial) details have been presented by Winikoff and Harland.[7]

System availability. The current implementation of Lygon (Version 0.4) is an interpreter written in BinProlog and is available by email from the authors or via the World Wide Web from the URL `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`. The system includes a debugger which extends the standard Prolog four port model with the ability to view the context.

One of the strengths of declarative languages is the support for formal reasoning tools: future work involves developing advanced debugging tools. Plans are afoot for a system which combines Lygon and Mercury to give a strongly typed and moded, compiled language.

[4] J.-Y. Girard, Linear Logic, *Theoret. Comp. Sci.*, 50:1–102, 1987.

[5] J. Harland, D. Pym, M. Winikoff. Programming in Lygon: an overview. This volume.

[6] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Inform. and Computat.* 110:2:327-365, 1994.

[7] M. Winikoff and J. Harland. Implementing the linear logic programming language Lygon. In: J. Lloyd (ed.), *Proc. ILPS'95*, 66–80, MIT Press, 1995.