

A Linear Logic Model of State

(Preliminary Report)

Uday S. Reddy

University of Illinois at Urbana-Champaign
reddy@cs.uiuc.edu

July 6, 1992

Abstract

We propose an abstract formal model of state manipulation in the framework of Girard's linear logic. This work addresses twin issues: how to incorporate state manipulation in functional programming languages, and how to describe the semantics of higher-order imperative programming languages. In fact, it appears that, given the right model of state, the difference between functional and imperative programming becomes rather thin. Our model of state is based on a new “modality” type constructor for expressing “regenerative values” (values that reproduce themselves each time they are used). Just as Girard's “of course” modality allows him to express static values and intuitionistic logic within the framework of linear logic, our regenerative modality allows us to express states and state-dependent values within the same framework. We demonstrate the expressiveness of the model by showing that a higher-order Algol-like language can be embedded into it.

1 Introduction

State is ubiquitous in computing. Many important applications of computing—data bases, operating systems and window management, just to mention a few—as well as many important algorithms are based on manipulation of state. Finding formal models, programming notations, as well as reasoning methods for state manipulation are important problems facing programming language research. Unfortunately, the current state of affairs in this regard is far from pleasant. Imperative programming languages—the most able languages for carrying out state manipulation—do not yet have good formal models. See [MS88] for a discussion of the problems. Functional programming languages—arguably, the best understood of languages—do not yet have sufficient capabilities to manipulate state. In a sense, the two problems are twin sides of the same coin. For, if we had a good formal model of state, we could devise programming notations based on it and incorporate them into a functional programming language.

Girard's linear logic [Gir87a] has been perceived, almost from the beginning, as a formal system that deals with state manipulation. This perception is implicitly present in Girard's own explanation of linear logic [Gir87c, Gir87b], its applications to Petri nets [BG90, GG89, MM91], and linear functional programming [Abr90, Hol88, Laf88]. However, Wadler [Wad90b, Wad91a] seems to have been the first to propose it as an explicit tool for formulating state manipulation in

functional languages. It is true that a rudimentary notion of state is available in linear logic since its values (proofs) are not *static* but *consumable*. Computations progress by consuming values and producing new values. This “change of state” involved in computations seems to have led to the perception that state manipulation is involved in linear logic.

However, a case can be made that *any* computation involves this form of “change of state”. The fact that change of state is *effected* by computations does not entail that state can be manipulated *within* computations. (Consider the graph reduction model of functional program execution, for example). To support the latter, one must have a model of state within the programming language and operations to manipulate state.

States are not merely “linear”. They have additional structure. This has to do with the fact that states “regenerate” themselves into new states each time they are used. In other words, carrying out any operation on a state often transforms it into a new state. The additional structure needed for capturing this regenerative capability seems to have eluded us for a long time. In this paper, we propose a formal structure to capture this aspect of state. Linear logic provides a framework in which it can be expressed.

Values in traditional functional programming are “static” values. They exist for eternity and can be used any number of times. States are obviously not static and, so, cannot be satisfactorily expressed in such functional languages. In contrast, values in linear logic may be said to be “consumable” values. They exist for one-time use, much like signals on a wire, and, so, capture dynamic aspects of computations. The richness of this model allows even static values to be modeled within the same framework. This is done by Girard in terms of the “of course” modality (a special form of a type constructor). (Cf. Section 2). In a similar vein, we model regenerative values in terms of another modality. A regenerative value exists for one-time use, but generates another value of the same type whenever it is used. (Cf. Section 3). States are formulated as a special form of regenerative values with capabilities for explicit modification. (Cf. Section 4). Thus, a model of state is *built into* the language and terms in the language become capable of manipulating state. This forms the foundation for devising functional programming languages capable of state manipulation.

A question arises as to how much state manipulation can be expressed in this framework. To answer this, we must look to the languages with the most powerful facilities for state manipulation, *viz.*, imperative programming languages. In the second part of the paper (Section 5), we demonstrate that the regenerative type system embeds a higher-order Algol-like language based on Reynolds’s notion of syntactic control of interference. *Interference* is the higher-order analogue of the familiar “aliasing” phenomenon. It causes nondeterminism (loss of Church-Rosser property) and explicit sequencing of evaluation order would be necessary to regain determinacy (much as in Standard ML [HMM86]). So, only interference-free programs can be modeled in the regenerative type system. However, the embedding shows that first-class state variables, procedures as well as objects can be modeled in the system. Thus, the model is significantly richer than that based on pure linear logic. (Cf. [Wad90b, Wad91a]).

The structural correspondence between interference control and linear logic has already been studied by O’Hearn [OHe91] and our embedding builds on his work.

2 Static and consumable values

Traditional functional programming (and intuitionistic logic) deal with *static* values, *i.e.*, values that continue to exist for eternity, and, so, can be used arbitrary number of times within programs (or proofs). This is signified by the intuitionistic cut rule

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B} \text{Cut}$$

where the inputs Γ are used both in the production of A as well as in the consumption of A .

Linear logic deals with *consumable* values, *i.e.*, values that exist for one-time use. Computations, thus, “consume” values while producing other values. This is signified by the linear logic cut rule:

$$\frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B} \text{Cut}$$

Of the hypotheses available in the production of B , some collection Γ of them are consumed in producing A and the remainder Δ in the consumption of A .

Metaphorically, a value of functional programming akin to a *register* of read-only memory while a value of linear logic behaves like a *signal* on a wire. The interesting point is that the computational behavior of registers can be simulated in terms of signals using the “modality” type constructor $!$. Intuitively, the modality may be viewed as the recursive type:

$$!A = \mathbf{1} \& A \& (!A \otimes !A)$$

The three components of the $!A$ type support the structural rules of *weakening* (for discarding), *dereliction* (for using) and *contraction* (for duplication) respectively. Further, given a construction of A from hypotheses $!\Gamma$, one can also produce a construction of $!A$. This latter operation is often called “promotion”.

$$\frac{\Gamma \vdash C}{\Gamma, !A \vdash C} !\text{Weak} \quad \frac{\Gamma, A \vdash C}{\Gamma, !A \vdash C} !\text{Der} \quad \frac{\Gamma, !A, !A \vdash C}{\Gamma, !A \vdash C} !\text{Contr} \quad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A} !\mathcal{R}$$

Using the “!” modality, Girard [Gir87a] showed that the entire intuitionistic logic can be embedded in linear logic.

To show how these constructions operate, let us introduce term syntax for them:

$$\frac{\Gamma \vdash t : A \quad \Delta, x : A \vdash u : B}{\Gamma, \Delta \vdash \mathbf{let } t \mathbf{ be } x \mathbf{ in } u : B} \text{Cut}$$

$$\frac{\Gamma \vdash u : C}{\Gamma, x : !A \vdash \mathbf{let } x \mathbf{ be } _ \mathbf{ in } u : C} !\text{Weak} \quad \frac{\Gamma, x : A \vdash u : C}{\Gamma, z : !A \vdash \mathbf{let } z \mathbf{ be } !x \mathbf{ in } u : C} !\text{Der}$$

$$\frac{\Gamma, x : !A, y : !A \vdash u : C}{\Gamma, z : !A \vdash \mathbf{let } z \mathbf{ be } x@y \mathbf{ in } u : C} !\text{Contr} \quad \frac{\bar{x} : !\Gamma \vdash t : A}{\bar{z} : !\Gamma \vdash [\bar{z} \mathbf{ be } \bar{x}]t : !A} !\mathcal{R}$$

The notation used here is similar to that of [Abr90]. The term $\mathbf{let } t \mathbf{ be } p \mathbf{ in } u$ corresponds to what is normally written as $\mathbf{let } p = t \mathbf{ in } u$. A term is in general of the form

$$\mathbf{let } \theta \mathbf{ in } u$$

where u is a term and θ is a collections of *coequations* of the form $t_i \mathbf{be} p_i$ (for terms t_i and patterns p_i). We often place braces around the coequations, as in $\mathbf{let} \{\theta\} \mathbf{in} u$, to make terms more readable. Assuming all variables are renamed apart from each other, the order of coequations is immaterial. Further, we implicitly use the following equivalences on the term syntax:

$$\begin{aligned} \mathbf{let} t \mathbf{be} x \mathbf{in} u &\equiv u[t/x] \\ \mathbf{let} \{t \mathbf{be} p[x], x \mathbf{be} p'\} \mathbf{in} u &\equiv \mathbf{let} t \mathbf{be} p[p'] \mathbf{in} u \\ \mathbf{let} \theta \mathbf{in} \mathbf{let} \phi \mathbf{in} u &\equiv \mathbf{let} \{\theta, \phi\} \mathbf{in} u \\ \mathbf{let} \{(\mathbf{let} \theta \mathbf{in} t) \mathbf{be} p\} \mathbf{in} u &\equiv \mathbf{let} \{\theta, t \mathbf{be} p\} \mathbf{in} u \end{aligned}$$

Note that, by the second equivalence, $\mathbf{let} z \mathbf{be} !x@_ \mathbf{in} u$ means the same as

$$\mathbf{let} \{z \mathbf{be} z_1@z_2, z_1 \mathbf{be} !x, z_2 \mathbf{be} _ \mathbf{in} u.$$

Similarly, we allow patterns to be used in all variable binding positions, *e.g.*, $\lambda!x@_ . u$ means $\lambda z. \mathbf{let} z \mathbf{be} !x@_ \mathbf{in} u$.

See [Abr90] for a comprehensive review of intuitionistic linear logic together with a functional term assignment. Our term assignment differs from that of [Abr90] only in the treatment of the promotion rule.

The operation of the above constructs is captured by the following reduction rules:

$$\begin{aligned} (!W) \quad \mathbf{let} \{![\bar{v} \mathbf{be} \bar{x}]t \mathbf{be} _ \} \mathbf{in} u &\rightarrow \mathbf{let} \bar{v} \mathbf{be} _ \mathbf{in} u \\ (!D) \quad \mathbf{let} \{![\bar{v} \mathbf{be} \bar{x}]t \mathbf{be} !z\} \mathbf{in} u &\rightarrow \mathbf{let} t[\bar{v}/\bar{x}] \mathbf{be} z \mathbf{in} u \\ (!C) \quad \mathbf{let} \{![\bar{v} \mathbf{be} \bar{x}]t \mathbf{be} z_1@z_2\} \mathbf{in} u &\rightarrow \mathbf{let} \{\bar{v} \mathbf{be} \bar{x}_1@\bar{x}_2, ![\bar{x}_1 \mathbf{be} \bar{x}]t \mathbf{be} z_1, ![\bar{x}_2 \mathbf{be} \bar{x}]t \mathbf{be} z_2\} \mathbf{in} u \end{aligned}$$

Note that the contraction operation is essentially implemented by copying. We call a promoted term of the form $![\bar{v} \mathbf{be} \bar{x}]t$ a *boxed term* in analogy with Girard’s terminology for proof nets [Gir87a]. The reason for the complex form of this term is that promotion is an operation on the “function” from \bar{x} to t , not on t itself. If we wrote the term as $!(t[\bar{v}/\bar{x}])$, then it would mean that the terms \bar{v} are also promoted, which is not necessarily the case. So, we think of the term t , with free variables \bar{x} , as being in a “box” while the inputs \bar{v} are waiting outside the box. If any of these inputs is, in turn, a promoted term, it can enter the box using the following rule:

$$(!Comm) \quad ![\bar{v} \mathbf{be} \bar{x}, ![\bar{w} \mathbf{be} \bar{y}]v \mathbf{be} x] t \rightarrow ![\bar{v} \mathbf{be} \bar{x}, \bar{w} \mathbf{be} \bar{y}] t(![\bar{y} \mathbf{be} \bar{y}]v)/x$$

Note that only other boxed terms can enter the box. In particular, it is not permissible for *linear* computations to enter the box (because they may then get discarded or duplicated). See [OHe91] and [Wad91b] for a discussion of various problems that would result if these distinctions are blurred.

3 Regenerative values

To deal with the issues of state, we need a new kind of values called *regenerative* values. Whenever a component of the state is used, another value of the same type is manufactured and plugged back into the state. Thus, states, as well as their components, are regenerative.

To start with, we introduce regenerative values that can be only be read, not written. We use a modality operator \dagger to denote this type of values. Intuitively, the modality may be viewed as the recursive type:

$$\dagger A = \mathbf{1} \&(A \otimes \dagger A)$$

This differs from the $!$ modality in that, instead of free copying denoted by $!A \otimes !A$, we have sequential copying denoted by $A \otimes \dagger A$. A regenerative value may be “read” giving an ordinary value of type A and a replica of the original regenerative value of type $\dagger A$. Regenerative values support the following type rules:

$$\frac{\Gamma \vdash C}{\Gamma, \dagger A \vdash C} \dagger\text{Weak} \quad \frac{\Gamma, A, \dagger A \vdash C}{\Gamma, \dagger A \vdash C} \dagger\text{Read} \quad \frac{\dagger\Gamma \vdash A}{\dagger\Gamma \vdash \dagger A} \dagger\mathcal{R}$$

The weakening and reading rules essentially project the two components given in the recursive definition. In the promotion rule ($\dagger\mathcal{R}$), the context must have regenerative inputs. Each time the the $\dagger A$ -typed value is read, we may need to carry out some number of reads of the regenerative inputs $\dagger\Gamma$.

Note that contraction is conspicuously absent from the above rules. Regenerative values cannot be simply copied. However, a simulated form of contraction, called “threading”, can be made available.

$$\frac{\Gamma, \dagger A, \dagger A \vdash C}{\Gamma, \dagger A \vdash C} \dagger\text{Thread}$$

Given a value of type $\dagger A$, we would like to obtain two values $(\dagger A)_1$ and $(\dagger A)_2$. To do this, we first use the given $\dagger A$ as $(\dagger A)_1$. This value would be read some number of times and, finally, relinquished (through weakening). At this point, we use the relinquished value as $(\dagger A)_2$.

All this can be made precise. Choose a term assignment:

$$\frac{\Gamma \vdash u : C}{\Gamma, s : \dagger A \vdash \text{let } s \text{ be } _ \text{ in } u : C} \dagger\text{Weak} \quad \frac{\Gamma, x : A, s' : \dagger A \vdash u : C}{\Gamma, s : \dagger A \vdash \text{let } s \text{ be } x \wedge s' \text{ in } u : C} \dagger\text{Read}$$

$$\frac{\bar{x} : \dagger\Gamma \vdash t : A}{\bar{z} : \dagger\Gamma \vdash \dagger[\bar{z} \text{ be } \bar{x}]t : \dagger A} \dagger\mathcal{R}$$

$$\frac{\Gamma, s_1 : \dagger A, s_2 : \dagger A \vdash u : C}{\Gamma, s : \dagger A \vdash \text{let } s \text{ be } s_1 \sim s_2 \text{ in } u : C} \dagger\text{Thread}$$

The reduction rules for the terms are as follows:

$$\begin{aligned} (\dagger W) \quad & \text{let } \{\dagger[\bar{v} \text{ be } \bar{x}]t \text{ be } _ \} \text{ in } u \rightarrow \text{let } \{\bar{v} \text{ be } _ \} \text{ in } u \\ (\dagger R) \quad & \text{let } \{\dagger[\bar{v} \text{ be } \bar{x}]t \text{ be } z \wedge s' \} \text{ in } u \rightarrow \text{let } \{\bar{v} \text{ be } \bar{x}_1 \sim \bar{x}_2, t[\bar{x}_1/\bar{x}] \text{ be } z, \dagger[\bar{x}_2 \text{ be } \bar{x}]t \text{ be } s' \} \\ & \text{in } u \\ (TW) \quad & \text{let } \{t \text{ be } s_1 \sim s_2, s_1 \text{ be } _ \} \text{ in } u \rightarrow \text{let } \{t \text{ be } s_2 \} \text{ in } u \\ (TR) \quad & \text{let } \{t \text{ be } s_1 \sim s_2, s_1 \text{ be } y \wedge s'_1 \} \text{ in } u \rightarrow \text{let } \{t \text{ be } y \wedge s', s' \text{ be } s'_1 \sim s_2 \} \text{ in } u \\ (\dagger\text{Comm}) \quad & \dagger[\bar{v} \text{ be } \bar{x}, \dagger[\bar{w} \text{ be } \bar{y}]v \text{ be } x]t \rightarrow \dagger[\bar{v} \text{ be } \bar{x}, \bar{w} \text{ be } \bar{y}]t(\dagger[\bar{y} \text{ be } \bar{y}]v/x) \end{aligned}$$

In order to “read” a regenerative value $\dagger[\bar{v} \text{ be } \bar{x}]t$, (rule $\dagger R$), we first thread its regenerative inputs \bar{v} through two copies \bar{x}_1 and \bar{x}_2 , use one copy in the read value, and use the other in the regenerated replica. The rules (TW) and (TR) show how threading is achieved. The commutation ($\dagger\text{Comm}$) is the analogue of (!Comm).

Notice the close similarity between static (!) types and regenerative (\dagger) types, both in syntax and semantics. The reduction ($\dagger R$) works essentially the same way as the corresponding reduction

(!C), except that copying ($\bar{x}_1 @ \bar{x}_2$) is replaced by threading ($\bar{x}_1 \sim \bar{x}_2$). We can also make available dereliction by a derived rule:

$$(\dagger\text{Der}) \quad \frac{\frac{\Gamma, x : A \vdash u : C}{\Gamma, x : A, z' : \dagger A \vdash \mathbf{let } z' \mathbf{ be } _ \mathbf{ in } u : C} \dagger\text{Weak}}{\Gamma, z : \dagger A \vdash \mathbf{let } z \mathbf{ be } x \hat{_} \mathbf{ in } u : C} \dagger\text{Read}$$

Discussion It has been felt that states are *linear* objects. (Cf. [Wad91a]). The above treatment suggests that they are more than linear; they are regenerative. States *appear* to be linear in some ways when viewed from a conventional functional language. For one, the promotion rule does not permit states in the context. For another, free contraction is not available to states. These properties are shared by linear values. On the other hand, states support weakening and threading both of which are not available for linear values. O’Hearn [OHe91] notes a similarity between states and linear values, essentially in that they both lack contraction, but he does not go so far as identifying the two. Moreover, the threading construction described above is implicitly present in his Seq (sequencing) rule.

Combining static and regenerative types We can now construct a unified type system that incorporates both kinds of types. The static types remain unchanged because regenerative values cannot be used in constructing static values (for the lack of contraction). On the other hand, static values can be used in constructing regenerative values. So, the promotion rule for regenerative values is generalized to

$$\frac{\bar{x} : \dagger\Gamma, \bar{x}' : !\Delta \vdash t : A}{\bar{z} : \dagger\Gamma, \bar{z}' : !\Delta \vdash \dagger[\bar{z} \mathbf{ be } \bar{x}; \bar{z}' \mathbf{ be } \bar{x}']t : \dagger A} \dagger\mathcal{R}$$

We separately mark the regenerative and static inputs, \bar{x} and \bar{x}' , used in constructing the regenerative value. If no static inputs are involved, we abbreviate it to $\dagger[\bar{z} \mathbf{ be } \bar{x}]t$, and, if no inputs at all are involved, we write it simply as $\dagger t$. The reduction rules ($\dagger\text{W}$) and ($\dagger\text{R}$) are easily extended to handle the static inputs:

$$\begin{aligned} (\dagger\text{W}) \quad \mathbf{let } \dagger[\bar{v} \mathbf{ be } \bar{x}; \bar{v}' \mathbf{ be } \bar{x}']t \mathbf{ be } _ \mathbf{ in } u &\rightarrow \mathbf{let } \{\bar{v} \mathbf{ be } _, \bar{v}' \mathbf{ be } _ \} \mathbf{ in } u \\ (\dagger\text{R}) \quad \mathbf{let } \dagger[\bar{v} \mathbf{ be } \bar{x}; \bar{v}' \mathbf{ be } \bar{x}']t \mathbf{ be } z \hat{_} s' \mathbf{ in } u &\rightarrow \mathbf{let } \{\bar{v} \mathbf{ be } \bar{x}_1 \sim \bar{x}_2, \bar{v}' \mathbf{ be } \bar{x}'_1 @ \bar{x}'_2, \\ &\quad t[\bar{x}_1/\bar{x}, \bar{x}'_1/\bar{x}'] \mathbf{ be } z, \dagger[\bar{x}_2 \mathbf{ be } \bar{x}; \bar{x}'_2 \mathbf{ be } \bar{x}']t \mathbf{ be } s'\} \\ &\quad \mathbf{in } u \end{aligned}$$

4 States

States are regenerative values with an additional capability: they can be written to. They form the recursive type:

$$st A = \mathbf{1} \& (!A \otimes st A) \& (!A \multimap st A)$$

The second component allows the state to be read, just as for regenerative values, but here we allow a *static* value (!A) to be read. (This is a matter of convenience; it elides having to look up the state each time we want to use its value). The third component allows the state to be replaced by a new state with a new value.

The operations for states are those of regenerative values together with a new operation for writing:

$$\begin{array}{c}
\frac{\Gamma \vdash u : C}{\Gamma, s : st A \vdash \mathbf{let } s \mathbf{ be } _ \mathbf{ in } u : C} \textit{ st Weak} \\
\frac{\Gamma, x : !A, s' : st A \vdash u : C}{\Gamma, s : st A \vdash \mathbf{let } s \mathbf{ be } x \hat{\sim} s' \mathbf{ in } u : C} \textit{ st Read} \quad \frac{\Gamma \vdash t : !A \quad \Delta, s' : st A \vdash u : C}{\Gamma, \Delta, s : st A \vdash \mathbf{let } (s \leftarrow t) \mathbf{ be } s' \mathbf{ in } u : C} \textit{ st Write} \\
\frac{\Gamma \vdash t : !A}{\Gamma \vdash st t : st A} \textit{ st } \mathcal{R} \quad \frac{\Gamma, s_1 : st A, s_2 : st A \vdash u : C}{\Gamma, s : st A \vdash \mathbf{let } s \mathbf{ be } s_1 \hat{\sim} s_2 \mathbf{ in } u : C} \textit{ st Thread}
\end{array}$$

Note that a state of type $st A$, once constructed, supports all the operations of the regenerative type $\dagger(!A)$. That is to say, $st A$ is a “subtype” of $\dagger(!A)$. Therefore, we permit states to be used in constructing other regenerative values (via $\dagger\mathcal{R}$ promotion rule) as if they were values of type $\dagger(!A)$. This puts “teeth” into regenerative values. While, earlier, regenerative values could only be cumbersome variants of static values, they can now be state-dependent values (*closures*) which regenerate themselves after each use so that the regenerated versions can be used in a new state.

The reduction rules for the state types are as follows:

$$\begin{array}{l}
(\textit{stW}) \quad \mathbf{let } st t \mathbf{ be } _ \mathbf{ in } u \rightarrow \mathbf{let } \{t \mathbf{ be } _ \} \mathbf{ in } u \\
(\textit{stR}) \quad \mathbf{let } st t \mathbf{ be } z \hat{\sim} s' \mathbf{ in } u \rightarrow \mathbf{let } \{t \mathbf{ be } z @ z', st z' \mathbf{ be } s' \} \mathbf{ in } u \\
(\textit{stWr}) \quad \mathbf{let } (st t \leftarrow v) \mathbf{ be } s' \mathbf{ in } u \rightarrow \mathbf{let } \{t \mathbf{ be } _, st v \mathbf{ be } s' \} \mathbf{ in } u
\end{array}$$

Threading of states is handled with (TW) and (TR) reductions and, in addition, the following reduction to thread through a write:

$$(\textit{TWr}) \quad \mathbf{let } \{s \mathbf{ be } s_1 \hat{\sim} s_2, (s_1 \leftarrow t) \mathbf{ be } s'_1 \} \mathbf{ in } u \rightarrow \mathbf{let } \{(s \leftarrow t) \mathbf{ be } s', s' \mathbf{ be } s'_1 \hat{\sim} s_2 \} \mathbf{ in } u$$

There is a crucial structural change to the logic introduced by the threading construction. Standard linear logic, just as all other familiar logics, is *commutative*. In particular, the occurrences of types in a typing context Γ can be freely reordered. This is no longer possible in the presence of *st Thread*. A threading operation such as $s \mathbf{ be } s_1 \hat{\sim} s_2$ is meaningful only if s_1 can be relinquished before s_2 is used. If there is a dependence from the consumer of s_2 to the consumer of s_1 , a deadlock would result. For example, in

$$\frac{\frac{\Gamma, s_2 : st A \vdash t : !A \quad \Delta, s'_1 : st A \vdash u : C}{\Gamma, s_2 : st A, \Delta, s_1 : st A \vdash \mathbf{let } (s_1 \leftarrow t) \mathbf{ be } s'_1 \mathbf{ in } u : C} \textit{ st Write}}{\Gamma, \Delta, s_1 : st A, s_2 : st A \vdash \mathbf{let } (s_1 \leftarrow t) \mathbf{ be } s'_1 \mathbf{ in } u : C} \textit{ Exchange}}{\Gamma, \Delta, s : st A \vdash \mathbf{let } s \mathbf{ be } s_1 \hat{\sim} s_2, (s_1 \leftarrow t) \mathbf{ be } s'_1 \mathbf{ in } u : C} \textit{ st Thread}$$

the state variable is set to t , which is dependent on some future state of the same state variable, resulting in a circularity. So, the order of occurrence of the typing assumptions $s_1 : st A, s_2 : st A$ is significant and exchanging them does not give an isomorphic typing context. A proper treatment of these features would be a topic for a powerful *noncommutative* linear logic. Despite some efforts [Abr91, BG91, Yet90], this subject is still in its infancy. We finesse the issue in the body of the paper by assuming that one uses the **Exchange** rule appropriately. A somewhat preliminary treatment of the noncommutative features may be found in Appendix A.

Example To illustrate what can be accomplished through the type system developed so far, we present an example. We would like to allocate a state variable v for a counter, and associate, with it, an increment operation which reads and increments its value. Note that this is a “side effecting” operation. Nevertheless, it can be expressed in the linear logic-based type system:

$$\begin{aligned} & \text{let } \dagger[st \ !0] \ \text{be } v \ (\text{let } \{v \ \text{be } (x_1 @ x_2) \wedge p, (p \leftarrow \text{addone}(x_1)) \ \text{be } _ \} \ \text{in } x_2) \\ & \quad \text{be } \text{inc}, \\ & \quad \text{inc} \ \text{be } i \wedge \text{inc}', \\ & \quad \text{inc}' \ \text{be } j \wedge _, \\ & \text{in } (i, j) \end{aligned}$$

where $\text{addone}(x_1)$ is short for

$$![x_1 \ \text{be } !k](k + 1)$$

The counter is a state object, of type $st(\mathbf{int})$, and the increment operation is a regenerative object (closure), of type $\dagger(!\mathbf{int})$. Note that the dependence of the closure on the state variables v is marked. The closure reads the state variable, makes two copies of the value (x_1 and x_2), uses x_1 to replace the state variable, and returns x_2 . The modified state is relinquished, that is to say, this particular use of the closure has nothing more to do with the modified state. Since the state is often threaded through multiple uses of the closure, relinquishing the state has the effect of *handing* it to the next use of the closure.

Let us work through some of the reductions to see what is going on. The read operation $\text{inc} \ \text{be } i \wedge \text{inc}'$ reduces to the following collection of coequations (using reduction $\dagger R$):

$$\begin{aligned} st \ !0 & \ \text{be } v_1 \wedge v', \\ v_1 & \ \text{be } (x_1 @ x_2) \wedge p, \\ (p \leftarrow \text{addone}(x_1)) & \ \text{be } _, \\ x_2 & \ \text{be } i, \\ \dagger[v' \ \text{be } v] (\text{let } \{v \ \text{be } (x_1 @ x_2) \wedge p, (p \leftarrow \text{addone}(x_1)) \ \text{be } _ \} \ \text{in } x_2) & \ \text{be } \text{inc}' \end{aligned}$$

Interpreting the threading $v \ \text{be } v_1 \wedge v'$ (through reduction rules TW and TR) gives:

$$\begin{aligned} v & \ \text{be } (x_1 @ x_2) \wedge p, \\ (p \leftarrow \text{addone}(x_1)) & \ \text{be } v', \\ x_2 & \ \text{be } i, \\ \dagger[v' \ \text{be } v] (\text{let } \{v \ \text{be } (x_1 @ x_2) \wedge p, (p \leftarrow \text{addone}(x_1)) \ \text{be } _ \} \ \text{in } x_2) & \ \text{be } \text{inc}' \end{aligned}$$

The reader is invited to work through the remaining reductions. x_1 , x_2 and i become $!0$, and v' becomes $st \ !1$. It is this new state v' that is used with the regenerated closure inc' . So, the next read operation binds j to $!1$ and the over all term reduces to $(!0, !1)$.

As a variation on the theme, an “up down” counter with multiple operations would be defined as follows:

$$\begin{aligned} \dagger[st \ !0] \ \text{be } v \ (\text{let } \{v \ \text{be } (x_1 @ x_2) \wedge p, (p \leftarrow \text{addone}(x_1)) \ \text{be } _ \} \ \text{in } x_2, \\ \text{let } \{v \ \text{be } (x_1 @ x_2) \wedge p, (p \leftarrow \text{subone}(x_1)) \ \text{be } _ \} \ \text{in } x_2)) \end{aligned}$$

To use the increment operation of this counter, we would say:

$$\text{updown } \mathbf{be} \langle i, _ \rangle \hat{\ } \text{updown}'$$

and, to use the decrement operation:

$$\text{updown } \mathbf{be} \langle _, i \rangle \hat{\ } \text{updown}'$$

□

4.1 Discussion

This concludes the first part of the paper. The contribution of this part is a type system that extends linear logic with regenerative types so that state-oriented computation can be carried out. Henceforth, we will refer to this system as the *regenerative type system*. Apparently, the language of this system is a functional language and it goes considerably beyond the previous attempts to incorporate state into functional languages using linear logic alone [Abr90, Hol88, Laf88, Wad90b, Wad91a]. The question is how far does it go? An answer is provided in the second part of the paper.

The language presented here is a *formal* language rather than a *practical* language in that it is too cumbersome to use. To build a practical language, one must suitably constrain the type system and provide appropriate syntactic sugar so that weakening and contraction are implicit, and promotion is not “boxed” as in $![\bar{v} \ \mathbf{be} \ \bar{x}]t$. Some previous attempts to provide such sugared syntax for linear logic are [Abr90, LM92, Wad91a] as well as Girard’s own logical “unity” system [Gir91]. We believe similar forms of sugaring would apply to the regenerative type system. However, in providing such syntactic sugar, a crucial point is reached at which the language ceases to be “functional” and becomes “imperative”. This point occurs when *threading* is made implicit.

The structural similarity between contraction and threading is apparent. By making contraction implicit, one allows multiple occurrences of an $!A$ -typed value to be referred to by the same name. Similarly, making threading implicit means that multiple occurrences of a regenerative value can be referred to by the same name. But, the different occurrences of a regenerative value refer to different states! Therefore, such names refer to the “time lines” of states—what are called “variables” or “references” in imperative programming. Thus, the highest form of sugared syntax for the regenerative type system is imperative programming.

5 Semantics of imperative languages

In this section, we use the regenerative type system to give “semantics” to an Algol-like language with Reynolds’s notion of syntactic control of interference (SCI) [Rey78, Rey89]. This is not of course a semantics in the standard sense, but, rather a translation from SCI to the regenerative type system. We will also finesse the issue of conjunctive types, which play a prominent role in [Rey89] but seem orthogonal to the issues of this paper, and use a formulation that closely follows O’Hearn [OHe91].

The types of SCI are partitioned into two classes called *active* types and *passive* types. Active types involve values that (possibly) modify the state whereas values of passive types leave the state

$$\begin{array}{c}
\frac{}{x : \theta \vdash x : \theta} \text{Id} \quad \frac{\Theta, x : \theta_1, y : \theta_2, \Theta' \vdash p : \theta}{\Theta, y : \theta_2, x : \theta_1, \Theta' \vdash p : \theta} \text{Exchange} \\
\frac{\Theta \vdash p : \theta'}{\Theta, x : \theta \vdash p : \theta'} \text{Weak} \quad \frac{\Theta, x_1 : \phi, x_2 : \phi \vdash p : \theta'}{\Theta, x : \phi \vdash p[x/x_1, x/x_2] : \theta'} \text{Contr} \\
\frac{\Theta_1 \vdash p : \theta_1 \quad \Theta_2, x : \theta_1 \vdash q : \theta_2}{\Theta_1, \Theta_2 \vdash q[p/x] : \theta_2} \text{Cut} \\
\frac{\Theta, x : \delta\text{var} \vdash c : \text{comm}}{\Theta \vdash \text{new } \delta x. c : \text{comm}} \text{New} \quad \frac{\Theta, x : \delta\text{exp} \vdash q : \theta'}{\Theta, x : \delta\text{var} \vdash q : \theta'} \text{Deref} \quad \frac{\Theta \vdash e : \delta\text{exp} \quad \Theta \vdash v : \delta\text{var}}{\Theta \vdash v := e : \text{comm}} \text{Assign} \\
\frac{\Theta \vdash c_1 : \text{comm} \quad \Theta \vdash c_2 : \text{comm}}{\Theta \vdash c_1; c_2 : \text{comm}} \text{Seq} \quad \frac{\Theta_1 \vdash c_1 : \text{comm} \quad \Theta_2 \vdash c_2 : \text{comm}}{\Theta_1, \Theta_2 \vdash c_1 \parallel c_2 : \text{comm}} \text{Par} \\
\frac{\Phi, x : \theta_1 \vdash p : \theta_2}{\Phi \vdash \lambda x. p : \theta_1 \rightarrow_P \theta_2} \rightarrow_P \mathcal{R} \quad \frac{\Theta_1 \vdash p : \theta_1 \quad \Theta_2, z : \theta_2 \vdash q : \theta'}{\Theta_1, \Theta_2, f : \theta_1 \rightarrow_P \theta_2 \vdash q[fp/z] : \theta'} \rightarrow_P \mathcal{L} \\
\frac{\Theta, x : \theta \vdash p : \alpha}{\Theta \vdash \lambda x. p : \theta \rightarrow \alpha} \rightarrow \mathcal{R} \quad \frac{\Theta_1 \vdash p : \theta \quad \Theta_2, z : \alpha \vdash q : \theta'}{\Theta_1, \Theta_2, f : \theta \rightarrow \alpha \vdash q[fp/z] : \theta'} \rightarrow \mathcal{L}
\end{array}$$

Figure 1: Type rules for Syntactic Control of Interference (SCI)

unchanged. The syntax of type terms is as follows:

$$\begin{array}{ll}
\text{passive types} & \phi ::= \delta\text{exp} \mid \theta \rightarrow_P \theta' \\
\text{active types} & \alpha ::= \delta\text{var} \mid \text{comm} \mid \theta \rightarrow \alpha \\
\text{types} & \theta ::= \phi \mid \alpha
\end{array}$$

where δ ranges over *data types* such as **int**, **bool** etc. $\theta \rightarrow_P \theta'$ denotes “state-independent” functions which, when invoked, do not read or write global variables. Reynolds imposes an important restriction on such types: if θ' is passive then θ must be passive. This eventually ensures that passive phrases have only passive phrases as components, and allows cut to be represented by substitution.

A sample language based on these types is shown in Fig. 1. Its intuitive semantics is obvious, except for, may be, that of $c_1 \parallel c_2$ which denotes the parallel execution of c_1 and c_2 . Note that, in this case, the two commands use separate type contexts Θ_1 and Θ_2 whereas in the sequencing command, $c_1; c_2$, the two commands use the same type context. The difference between the sequential and parallel compositions is the essence of the language. Since contraction is only permitted for passive types, the separate type contexts Θ_1 and Θ_2 in the **Par** rule (as well as in $\rightarrow_P \mathcal{L}$ and $\rightarrow \mathcal{L}$ rules) cannot share any *active* components. So, control of interference is achieved merely by prohibiting contraction for active types. There is no explicit rule for threading. Instead, it is implicitly involved in **Seq** and **Assign**.

SCI types are translated to the regenerative type system as follows:

$$\begin{aligned}
(\delta\mathbf{exp})^* &= !\delta \\
(\theta_1 \rightarrow_P \theta_2)^* &= !(\theta_1^* \multimap \theta_2^*) \\
(\delta\mathbf{var})^* &= st \delta \\
\mathbf{comm}^* &= \dagger\mathbf{1} \\
(\theta \rightarrow \alpha)^* &= \dagger(\theta^* \multimap \alpha^*)
\end{aligned}$$

All passive types become static types in the linear setting and all active types become regenerative types.¹

A phrase with the typing $\Theta \vdash p : \theta$ is translated to a term with linear typing:

$$\Theta^* \vdash t : \theta^*$$

Stated in semantic terms, the meaning of the phrase p is a function from environments respecting Θ to the domain of θ .

The most striking difference between this semantics and the conventional one is the absence of a *state* parameter. Conventionally, the meaning of an expression maps an environment and a state to a value, that of a command maps an environment and a state to a state, and so on. This approach leads to endless difficulties. One is lead to ask “what are states?” If the answer is maps from locations to values then what are locations? Which locations are mapped? The original semantics documented in [MS76, Sto77] proved to be too concrete (read “too operational”). Reynolds and Oles [Ole85, Rey81] proposed a more abstract notion of state, essentially as tuples of values, and modeled variables as abstract entities with capabilities for producing a value from a state and plugging a value into a state. This is appealing, despite being highly technical. But, it still divides the context of a basic phrase into an environment and a state—one being “static” and the other “dynamic”—and the two do not match up. Suppose we define a procedure p in a static context Γ . The only objects the procedure can be manipulate are those in Γ . But, dynamically, the procedure can be called from a larger context Γ' . We are forced to define the meaning of p by its ability to operate in all such larger contexts. This leads to the so-called *possible world* semantics [OT92b, Ole85, Rey81, Ten89]. However, the procedure *cannot* touch anything outside the context of its definition. So, much technical sweat is wasted on utterly uninteresting activity.

Our approach, whose bare framework has already been established, is based on building regenerative capability directly into states and state-dependent quantities. We do not define the meaning of a procedure for contexts larger than that of its definition. The procedure simply ignores (by weakening) all objects outside its original context and these objects are carried to their appropriate destinations by threading. Consequently, there is no need for separate environment and state.

We now show the translation of the SCI phrases by induction on their type derivations. All the “logical” constructions have standard translations. So, we give only a brief sketch. The rules **Id**, **Weak**, **Contr** and **Cut** translate to their counterparts in the regenerative type system. (In case of **Weak**, it is one of **!Weak** and **†Weak** depending on θ^*). The function constructions $\rightarrow_P \mathcal{R}$ and $\rightarrow \mathcal{R}$ translate to a $\multimap \mathcal{R}$ followed by **!R** and **†R** respectively. The function application rules $\rightarrow_P \mathcal{L}$ and

¹A note for the linear logician: the passive part of the translation is the same as Girard’s “less satisfactory” $()^*$ translation of intuitionistic logic [Gir87a, Sec. 5.2]. A purer translation along the lines of $()^\circ$ translation also exists. See the notes at the end of the section. This and other features of the translation lead us to the belief that SCI is to the regenerative type system what intuitionistic logic is to linear logic.

$\rightarrow\mathcal{L}$ translate to $\multimap\mathcal{L}$ followed by $!Der$ and $\dagger Der$ respectively. For example, here is the translation of $\rightarrow\mathcal{L}$:

$$\frac{\frac{\Theta_1^* \vdash t : \theta_1^* \quad \Theta_2^*, z : \theta_2^* \vdash u : \theta'^*}{\Theta_1^*, \Theta_2^*, f' : \theta_1^* \multimap \theta_2^* \vdash \mathbf{let} f't \mathbf{be} z \mathbf{in} u : \theta'^*} \multimap\mathcal{L}}{\Theta_1^*, \Theta_2^*, f : \dagger(\theta_1^* \multimap \theta_2^*) \vdash \mathbf{let} \{f \mathbf{be} f' \wedge _, f't \mathbf{be} z\} \mathbf{in} u : \theta'^*} \dagger Der$$

Next, we look at the translation of imperative programming rules. If Θ is an SCI typing context, its translation Θ^* is a regenerative typing context $\dagger\Gamma, !\Delta$ where $\dagger\Gamma$ is the translation of active types in Θ and $!\Delta$ that of passive types in Θ . We use $\dagger\Gamma$ and $!\Delta$ below with this interpretation.

- The rule **New** translates to

$$\frac{\frac{\frac{\frac{}{\vdash \mathbf{undef}_\delta : \delta}}{\vdash !\mathbf{undef}_\delta : !\delta} !\mathcal{R}}{\vdash st (!\mathbf{undef}_\delta) : st \delta} st \mathcal{R}}{\dagger\Gamma, !\Delta \vdash \mathbf{let} st (!\mathbf{undef}_\delta) \mathbf{be} x \mathbf{in} t : \dagger\mathbf{1}} \text{Cut}}{\dagger\Gamma, !\Delta, x : st \delta \vdash t : \dagger\mathbf{1}}$$

where \mathbf{undef}_δ is a constant, in each data type δ , denoting the value of an uninitialized state variable.

- The rule **Deref** translates to

$$\frac{\dagger\Gamma, !\Delta, x : !\delta \vdash t : \theta'^*}{\dagger\Gamma, !\Delta, s : st \delta \vdash \mathbf{let} s \mathbf{be} x \wedge _ \mathbf{in} t : \theta'^*} st Der$$

- The translation of **Assign**, when Θ has only active types, is:

$$\frac{\frac{\frac{\frac{\frac{}{\vdash () : \mathbf{1}}{\vdash () : \mathbf{1}} st Weak}{\bar{x}_1 : \dagger\Gamma \vdash e : !\delta \quad z' : st \delta \vdash \mathbf{let} z' \mathbf{be} _ \mathbf{in} () : \mathbf{1}}{\bar{x}_1 : \dagger\Gamma, z : st \delta \vdash \mathbf{let} (z \leftarrow e) \mathbf{be} _ \mathbf{in} () : \mathbf{1}} st Write}}{\bar{x}_1 : \dagger\Gamma, \bar{x}_2 : \dagger\Gamma \vdash \mathbf{let} (v \leftarrow e) \mathbf{be} _ \mathbf{in} () : \mathbf{1}} \text{Cut}}{\bar{x}_1 : \dagger\Gamma, \bar{x}_2 : \dagger\Gamma \vdash \mathbf{let} (v \leftarrow e) \mathbf{be} _ \mathbf{in} () : \mathbf{1}} \text{Thread}}{\bar{x} : \dagger\Gamma \vdash \mathbf{let} \{\bar{x} \mathbf{be} \bar{x}_1 \sim \bar{x}_2, (v \leftarrow e) \mathbf{be} _ \} \mathbf{in} () : \mathbf{1}} \dagger\mathcal{R}}{\bar{x} : \dagger\Gamma \vdash \dagger[\bar{x} \mathbf{be} \bar{x}] \mathbf{let} \{\bar{x} \mathbf{be} \bar{x}_1 \sim \bar{x}_2, (v \leftarrow e) \mathbf{be} _ \} \mathbf{in} () : \dagger\mathbf{1}}$$

Note that this can be written more simply using the pattern cascading convention:

$$\dagger[\bar{x} \mathbf{be} \bar{x}_1 \sim \bar{x}_2] \mathbf{let} (v \leftarrow e) \mathbf{be} _ \mathbf{in} ()$$

If Θ has passive types as well, a similar derivation yields the translation:

$$\bar{x} : \dagger\Gamma, \bar{y} : !\Delta \vdash \dagger[\bar{x} \mathbf{be} \bar{x}_1 \sim \bar{x}_2, \bar{y} \mathbf{be} \bar{y}_1 @ \bar{y}_2] \mathbf{let} (v \leftarrow e) \mathbf{be} _ \mathbf{in} () : \dagger\mathbf{1}$$

Proposition 1 *Given a command $\Theta \vdash c : \mathbf{comm}$ and states $\sigma = \{\bar{v} \mapsto \bar{k}\}$ and $\sigma' = \{\bar{v} \mapsto \bar{k}'\}$, there exists a command c' such that $c \rightarrow^* c'$ and $(\sigma, c) \downarrow c'$ if and only if*

$$\dagger[st(\bar{k}) \mathbf{be} \bar{x}]c^* \rightarrow^* \dagger[st(\bar{k}') \mathbf{be} \bar{x}']()$$

where c^* is the translation of c .

The proof depends on the fact that all the state transitions of the imperative language are faithfully represented in the reductions of the regenerative type system.

Purer translation One of the problems with the $()^*$ translation is that it builds too many suspensions and closures (values of $!A$ and $\dagger A$ -typed values). Witness this in the translation of **Seq** and **Par** constructions which force closures only to wrap them in new closures. In [Gir87a, Sec. 5.2], Girard gives a $()^\circ$ translation from intuitionistic logic to linear logic which avoids this problem. A similar translation from SCI to regenerative type system would work as follows. The types have two translations $()^\circ$ and $()^*$ defined mutually recursively:

$$\begin{aligned} (\delta \mathbf{exp})^\circ &= \delta \\ (\theta_1 \rightarrow_P \theta_2)^\circ &= \theta_1^* \multimap \theta_2^\circ \\ (\delta \mathbf{var})^\circ &= st \delta \\ \mathbf{comm}^\circ &= \mathbf{1} \\ (\theta \rightarrow \alpha)^\circ &= \theta^* \multimap \alpha^\circ \\ \phi^* &= !\phi^\circ \\ \alpha^* &= \begin{cases} \alpha^\circ, & \text{if } \alpha = \delta \mathbf{var} \\ \dagger \alpha^\circ, & \text{otherwise} \end{cases} \end{aligned}$$

An SCI typing $\Theta \vdash p : \theta$ is then translated to $\Theta^* \vdash t : \theta^\circ$. The reader can work out for herself the translation of type rules and phrases. Notice, again, that the passive part of the translation coincides with Girard's translation of intuitionistic logic.

Product types Linear logic has two forms of product types and SCI needs the two as well. In SCI, the tensor product $\theta \otimes \theta'$ would denote pairs constructed from noninterfering inputs. Such pairs are useful for passing multiple arguments to procedures and also for building *record* data structures as in Pascal. However, types produced by tensor product do not seem to fall into the active-passive framework of SCI. Is $\alpha \otimes \phi$, for instance, an active type or a passive type? Even for a product such as $\alpha \otimes \alpha'$, we cannot readily see how to treat it as a regenerative type. So, it would not be right to classify it as an active type. This is a subject for further research.

The regular product types, also called *additive* products or *lazy* products, exist in SCI. Finding a convenient syntax for them, however, seems to require Reynolds's conjunctive types. While we do not know how to treat conjunctive types in linear logic, we can readily translate product types and type derivations (abstract syntax) into the regenerative type system. First, extend the syntax of SCI types as follows:

$$\begin{aligned} \phi &::= \dots \mid \phi_1 \times \phi_2 \\ \alpha &::= \dots \mid \alpha_1 \times \alpha_2 \mid \alpha \times \phi \mid \phi \times \alpha \end{aligned}$$

The translation to regenerative type system is as follows:

$$\begin{aligned}(\phi_1 \times \phi_2)^\circ &= \phi_1^\circ \& \phi_2^\circ \\(\alpha_1 \times \alpha_2)^\circ &= \alpha_1^\circ \& \alpha_2^\circ \\(\alpha \times \phi)^\circ &= \alpha^\circ \&! \phi^\circ \\(\phi \times \alpha)^\circ &= !\phi^\circ \& \alpha^\circ\end{aligned}$$

Active product types are useful for building “objects” that export multiple operations all of which operate on the same dynamic data. Witness this in the *updown* counter example of Section 4.

6 Conclusion

We have defined a formal system based on linear logic providing a model for state manipulation. The expressiveness of the system is demonstrated by showing that it embeds a higher-order Algol-like imperative programming language.

Much further work remains to be done. The foremost issue is the semantics of the regenerative type system. We would like concrete models such as coherent spaces as well as abstract definitions of models in a categorical setting. It appears that a coherent space for $\dagger A$ can be defined in terms of *sequences* of tokens of A . Such models would also lead to a better understanding of the noncommutative features involved in the regenerative types.

The relation between the regenerative semantics for imperative languages and the “possible world” semantics of [OHe92, Ole85, OT92a] must be investigated. Once the semantic issues regarding the present approach are worked out, it would be possible to study the full abstraction issues which currently plague the semantics of imperative languages.

Convenient concrete syntax must be found for functional languages based on regenerative types. Ideally, the syntax should hide the noncommutative features of the formal system. There is an interesting duality between the monad-based approaches to state [Mog91, Wad90a] and regenerative types. (The type $\dagger A$ is a comonad). It is very well possible that a reformulation of regenerative types in terms of monads gives a better syntax.

Appendix

A Noncommutative features of Regenerative types

The tensor product of standard linear logic is commutative, *i.e.*, $A \otimes B \cong B \otimes A$. For the regenerative type system, we need a noncommutative tensor product \star which has all the properties of \otimes except for the commutative property.

The minimal noncommutative support needed for the regenerative type system is a way to deal with the implicit use of \star in typing contexts. A standard typing context is a sequence of types A_1, \dots, A_n which is interpreted as the commutative tensor product $A_1 \otimes \dots \otimes A_n$. In contrast, a typing context of the regenerative type system may involve both \otimes and \star . So, define the following grammar for typing contexts:

$$\Gamma ::= A \mid \Gamma_1, \Gamma_2 \mid \Gamma_1; \Gamma_2$$

(We assume that “,” binds closer than “;”). Γ_1, Γ_2 is interpreted as the tensor product $\Gamma_1 \otimes \Gamma_2$ and the $\Gamma_1; \Gamma_2$ as noncommutative tensor product $\Gamma_1 \star \Gamma_2$.

All the typing rules operate on *occurrences* of types inside typing contexts. We show Cut and Contraction as examples:

$$\frac{\Gamma \vdash A \quad \Delta[A] \vdash B}{\Delta[\Gamma] \vdash B} \text{Cut} \qquad \frac{\Gamma[!A, !A] \vdash C}{\Gamma[!A] \vdash C} !\text{Contr}$$

All the other rules are modified similarly, except for weakening, threading and exchange. The weakening rules come in three forms:

$$\frac{\Gamma[\Delta] \vdash C}{\Gamma[\Delta, T] \vdash C} \qquad \frac{\Gamma[\Delta] \vdash C}{\Gamma[\Delta; T] \vdash C} \qquad \frac{\Gamma[\Delta] \vdash C}{\Gamma[T; \Delta] \vdash C}$$

for T of the form $!A$ or $\dagger A$. The threading rules eliminate occurrences of “;”.

$$\frac{\Gamma[(\Delta_1, \dagger A); (\Delta_2, \dagger A)] \vdash C}{\Gamma[\Delta_1; \Delta_2] \vdash C} \dagger\text{Thread}$$

Finally, the Exchange rule splits into many coherence transformations:

$$\frac{\Gamma[\Delta'] \vdash C}{\Gamma[\Delta] \vdash C} \text{Exchange} \qquad \text{where } \Delta \rightarrow \Delta'$$

$$\begin{aligned} \Delta_1, \Delta_2 &\rightleftharpoons \Delta_2, \Delta_1 \\ (\Delta_1, \Delta_2), \Delta_3 &\rightleftharpoons \Delta_1, (\Delta_2, \Delta_3) \\ (\Delta_1; \Delta_2); \Delta_3 &\rightleftharpoons \Delta_1; (\Delta_2; \Delta_3) \\ \Delta_2, (\Delta_2; \Delta_3) &\rightarrow (\Delta_1, \Delta_2); \Delta_3 \\ (\Delta_1; \Delta_2), \Delta_3 &\rightarrow \Delta_1; (\Delta_2, \Delta_3) \\ \Delta_1, \Delta_2 &\rightarrow \Delta_1; \Delta_2 \end{aligned}$$

References

- [Abr90] S. Abramsky. *Computational Interpretations of Linear Logic*. Research Report DOC 90/20, Imperial College, London, Oct 1990. (available by FTP from theory.doc.ic.ac.uk; to appear in *J. Logic and Computation*).
- [Abr91] V. M. Abrusci. Phase and semantics and sequent calculus for pure noncommutative classical linear propositional logic. *J. Symbolic Logic*, 56(4):1403–1451, Dec 1991.
- [BG90] C. Brown and D. Gurr. A categorical linear framework for petri nets. In *Fifth Ann. Symp. on Logic in Comp. Science*, pages 208–218, IEEE, June 1990.
- [BG91] C. Brown and D. Gurr. *Relations and Non-commutative Linear Logic*. Technical Report DAIMI PB-372, Aarhus University, Denmark, Nov 1991.
- [GG89] C. Gunter and V. Gehlot. *Nets as Tensor Theories*. Technical Report MS-CIS-89-68, University of Pennsylvania, Oct 1989.
- [Gir87a] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.

- [Gir87b] J.-Y. Girard. Linear logic and parallelism. In M. V. Zilli, editor, *Mathematical Models for the Semantics of Parallelism*, pages 166–182, Springer-Verlag, Berlin, 1987. (LNCS Vol. 280).
- [Gir87c] J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 69–108, American Mathematical Society, Boulder, Colorado, June 1987. (Contemporary Mathematics, Vol. 92).
- [Gir91] J.-Y. Girard. *On the Unity of Logic*. Prepublication 26, Equipe de Logique, University of Paris VII, June 1991.
- [HMM86] R. Harper, D. B. MacQueen, and R. Milner. *Standard ML*. Technical Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, Mar 1986.
- [Hol88] S. Holmstrom. A linear functional language. In T. Johnsson, S. Peyton Jones, and K. Karlsson, editors, *Proc. Workshop on Implementation of Lazy Functional Languages*, pages 13–32, Chalmers University, 1988.
- [Laf88] Y. Lafont. The linear abstract machine. *Theoretical Comp. Science*, 59:157–180, 1988.
- [LM92] P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *Symp. on Logic in Comp. Science*, IEEE, June 1992.
- [MM91] N. Marti-Oliet and J. Meseguer. From Petri nets to linear logic. *Math. Structures in Comp. Science*, 1:69–101, 1991.
- [Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, ():, 1991.
- [MS76] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976.
- [MS88] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 191–203, ACM, 1988.
- [OHe91] P. W. O’Hearn. Linear logic and interference control. In D. H. Pitt et. al., editor, *Category Theory and Computer Science*, pages 74–93, Springer-Verlag, Berlin, 1991. (LNCS Vol. 530).
- [OHe92] P. W. O’Hearn. *Semantics Aspects of Syntactic Control of Interference*. Manuscript, Syracuse University, Jan 1992.
- [Ole85] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573, Cambridge Univ. Press, Cambridge, U. K., 1985.
- [OT92a] P. W. O’Hearn and R. D. Tennent. Semantical analysis of specification logic, Part 2. *Information and Computation*, (to appear)():, 1992.

- [OT92b] P. W. O’Hearn and R. D. Tennent. Semantics of local variables. In *Proc. Durham Symposium on Applications of Categories in Computer Science*, page (to appear), London Math. Soc. Lecture Notes Series, 1992.
- [Rey78] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46, ACM, 1978.
- [Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, North-Holland, 1981.
- [Rey89] J. C. Reynolds. Syntactic control of interference, Part II. In , editor, *Intern. Colloq. Automata, Languages. and Programming*, pages 704–722, Springer-Verlag, 1989. (LNCS Vol. 372).
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Ten89] R. D. Tennent. Denotational semantics of Algol-like languages. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol II*, Oxford University Press, 1989. (to appear).
- [Wad90a] P. Wadler. Comprehending monads. In *ACM Symp. on LISP and Functional Programming*, 1990.
- [Wad90b] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, page , North-Holland, Amsterdam, 1990. (Proc. IFIP TC 2 Working Conf., Sea of Galilee, Israel).
- [Wad91a] P. Wadler. Is there a use for linear logic? In *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, ACM, 1991. (SIGPLAN Notices, Sep. 1991).
- [Wad91b] P. Wadler. *There’s no Substitute for Linear Logic*. Manuscript, Department of Computing Science, University of Glasgow, Dec 1991.
- [Yet90] D. Yetter. Quantales and non-commutative linear logic. *J. Symbolic Logic*, 55(I):41–64, 1990.