

A Linear Logic Model of State (Extended Abstract)

Uday S. Reddy*

Abstract

We propose an abstract formal model of state manipulation in the framework of Girard’s linear logic. Two issues motivate this work: how to describe the semantics of higher-order imperative programming languages and how to incorporate state manipulation in functional programming languages. The central idea is that a state is linear and “regenerative”, where the latter is the property of a value that generates a new value upon each use. Based on this, we define a type constructor for states and a “modality” type constructor for regenerative values. 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 dynamic values and imperative programs within the same framework. We demonstrate the expressiveness of the model by showing that a higher-order Algol-like language can be embedded in it.

1 Introduction

The question addressed in this paper is “What is state?” The interest in the question is twofold: to build semantic models of imperative programming languages and, secondly, to find ways to incorporate state manipulation in functional languages.

The traditional models [Gor79, MS76, Sto77] are based on the notion of *global state*. In other words, “state” is a state of the entire store. A reference (or state “variable”) is viewed as an index into the global state. Commands take global states to global states, even though an individual command only transforms a small portion of the global state. This view point leads to the problem that Backus labelled most colorfully as the “von Neumann bottleneck”. On the theoretical front, it leads to the problem of the semantics of “local variables”. See [HMT83, MS88, OT92] for a discussion of the latter.

On the other hand, the notion of a “local state” comes naturally to the programmer. For instance, in the object-oriented tradition, an object is viewed as an encapsulated local state together with operations on it. Similarly, every command and every procedure can be viewed as acting on its own local state. The traditional semantics is unable to support this view point.

An alternative approach to state called “possible world semantics” was formulated by Reynolds and Oles [Ole85, Rey81]. (Recent work [OT92] relates it to [MS88]). Here, all the semantic domains are indexed by stores (or “worlds”). Commands (as well as all other phrases) have meanings in *each* store. These meanings must satisfy certain naturality relationships so as to form the meaning of the same command. (Mathematically, the semantic “domains” are *functors* from the category of worlds to a semantic category and meanings are natural transformations between such functors).

*Address for correspondence: Department of Computer Science, 1304 W. Springfield Avenue, University of Illinois at Urbana-Champaign, Urbana, IL 61801. Email: reddey@cs.uiuc.edu.

While this approach is elegant and appealing, it remains a technical trepeze act where local states are obtained only indirectly by quantifying over *all* states.

In this paper, we propose a new analysis of states afforded by Girard’s linear logic [Gir87, GLT89, Laf88]. We model states by linear values with certain additional structure. The additional structure has to do with the fact that states are *regenerative*: each use of a state generates a new state which then replaces the old state. Thus, to have access to a state is to have access to an entire “thread” of states. A command takes an initial state, develops a certain thread of states—using up all the intermediate states in the process—and, finally, discards the last state in the thread. References are such threads of states.

Two points must be noted. First, what we call a state may be either a “small” state, such as individual reference, or a “large” state, such as an array or the state of the store. The model is indifferent to the “size” of the state. In this paper, we only focus on small states because that is where the novelty lies.

The second point is that a command *discards* the final state. This is in sharp contrast to the traditional model where a command *returns* the final state. How, then, are the effects of the command to be observed? This is achieved by a combinator on states called *threading*. Given a state s , we can pretend that we have two “copies” of the state, say s_1 and s_2 , and use them in two separate commands c_1 and c_2 respectively. At the same time, we remember which copy has *priority* (say s_1). The threading combinator combines the copies s_1 and s_2 into a single state s as follows. Whatever read/write operations c_1 carries out on s_1 are transmitted to s . When c_1 eventually discards some regenerated version of s_1 , the corresponding regenerated version of s is used as s_2 (*i.e.*, all the read/write operations on s_2 are transmitted to this regenerated version of s). Semantically speaking, the threading combinator concatenates the state threads s_1 and s_2 into the state thread s . Operationally speaking, when the command c_1 discards its private copy of the state (s_1), it effectively *releases* the original state s for use by c_2 . In the process, it communicates its effects on s to c_2 .

While the traditional model views commands as functions from states to states and sequencing as function composition, we view commands as computations on regenerative states and sequencing as threading of states. This change of view point accrues many benefits. Foremost among them is the fact that expanding a meaning from a small state to a larger state is as simple as *weakening* on a state-typed input (discarding a state-typed input). This same operation works for all types of meanings (expressions, commands, procedures and other higher-order phrases). Secondly, it opens the door to a genuinely “logical” view of state-oriented computation. In previous work, jointly with V. Swarup and E. Ireland, we presented a type-theoretic calculus for imperative computations [SRI91, SR91]. Many curious features and difficulties encountered there seem explainable using the present framework.

Unfortunately, the logical model of state presented here is not without some cost. It seems to require some form of a *noncommutative* linear logic rather than the standard commutative linear logic. When we assume, using the threading combinator, two copies of a given state, the copies must be used in a *sequential* fashion and cannot be arbitrarily exchanged. Despite some efforts [Abr91, BG91, Retb, Yet90], the subject of noncommutative linear logic is still in its infancy. We use the minimal amount of noncommutative features in this paper awaiting further developments in this subject.

To summarize the results of this paper, we present a logical system (with propositions as types and proofs as programs) that models state-oriented computation (Section 3). This is called the *regenerative type system*. The system has a semantics in the framework of Girard’s coherent spaces (Section 4). We demonstrate the efficacy of the system as a logical-semantical model by giving

semantics to a higher-order Algol-like language based on Reynolds’s syntactic control of interference (Section 5). Section 2 gives a brief explanation of linear logic and the notation we use.

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 (a derived rule based on a natural deduction presentation):

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

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 *tokens* on a data flow arc. The interesting point is that the computational behavior of registers can be simulated in terms of tokens 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 [Gir87] 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:

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

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

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

The notation used here is similar to that of [Abr90]. 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 $p_i = t_i$ (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:

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

Note that, by the second equivalence, $\mathbf{let} !x@_ = z \mathbf{in} u$ means the same as $\mathbf{let} \{z_1@z_2 = z, !x = z_1, _ = z_2\} \mathbf{in} u$. Similarly, we allow patterns to be used in all variable binding positions, *e.g.*, $\lambda !x@_ . u$ means $\lambda z. \mathbf{let} !x@_ = z \mathbf{in} u$. The reader is referred to [Abr90] for a comprehensive review of intuitionistic linear logic and its term assignment. (Our syntax mixes the two syntaxes given in his Sec. 3 and Sec. 6).

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

$$\begin{aligned} (!W) \quad \mathbf{let} \{_ = ![x = \bar{v}] t\} \mathbf{in} u &\rightarrow \mathbf{let} _ = \bar{v} \mathbf{in} u \\ (!D) \quad \mathbf{let} \{!z = ![x = \bar{v}] t\} \mathbf{in} u &\rightarrow \mathbf{let} z = t[\bar{v}/x] \mathbf{in} u \\ (!C) \quad \mathbf{let} \{z_1@z_2 = ![x = \bar{v}] t\} \mathbf{in} u &\rightarrow \mathbf{let} \{\bar{x}_1@x_2 = \bar{v}, z_1 = ![x = \bar{x}_1] t, z_2 = ![x = \bar{x}_2] t\} \\ &\quad \mathbf{in} u \end{aligned}$$

Note that the contraction operation is essentially implemented by copying. We call a promoted term of the form $![x = \bar{v}] t$ a *boxed term* in analogy with Girard’s terminology for proof nets [Gir87]. 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 $!(t[\bar{v}/x])$, 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 equivalence:

$$(!Comm) \quad ![x = \bar{v}, x = ![y = \bar{w}] v] t \equiv ![x = \bar{v}, y = \bar{w}](t([y = \bar{y}] v)/x)$$

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 [BBdPH92, O’H91, Wad91] for a discussion of various problems that would result if these distinctions are blurred.¹

3 Regenerative Type system

Let us start with a simple notion of states: $st A$ is the type of states that hold *static* values of type A , *i.e.*, values of type $!A$. (We cannot store a linear value in the state because it may be read multiple times). The following operations on states suggest themselves:

$$\begin{aligned} &\frac{\Gamma, x : !A, s' : st A \vdash u : C}{\Gamma, s : st A \vdash \mathbf{let} x \hat{=} s' = s \mathbf{in} u : C} \text{ st Read} \\ &\frac{\Gamma \vdash t : !A \quad \Delta, s' : st A \vdash u : C}{\Gamma, \Delta, s : st A \vdash \mathbf{let} s' = (s \leftarrow t) \mathbf{in} u : C} \text{ st Write} \\ &\frac{\Gamma \vdash t : !A}{\Gamma \vdash st t : st A} \text{ st } \mathcal{R} \end{aligned}$$

¹The boxed syntax comes to us from [Abr90, Sec. 6].

Notice that both the read and write operations “regenerate” states, *i.e.*, they produce a new state from the state that was used. (The same feature was observed in [SRI91] as “linearity” of observers). In addition to these operations, it seems necessary to provide two structural rules: *weakening* to discard a state and *threading* to assume two copies of a state:

$$\frac{\Gamma \vdash u : C}{\Gamma, s : st A \vdash \mathbf{let} _ = s \mathbf{in} u : C} \text{ st Weak}$$

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

The need for weakening is easy to see. The threading operation (discussed in Introduction) is necessary for sequencing. For example, the imperative programming construction

$$\frac{v : A \mathbf{var} \vdash c_1 : \mathbf{comm} \quad v : A \mathbf{var} \vdash c_2 : \mathbf{comm}}{v : A \mathbf{var} \vdash (c_1; c_2) : \mathbf{comm}} \text{ Seq}$$

requires that we split the state thread of v into two parts, use the earlier part in c_1 and use the later part in c_2 . The coequation $s_1 \sim s_2 = s$ captures this form of splitting. Notice that the threading construction is superficially similar to contraction, but its semantics is quite different.

The rule *st Thread*, as written above, leads to problems. In splitting the state thread of s into two parts, we have an understanding that the first part must be used *before* the second part. This notion of “before” is not expressed in the rule. If one were to ignore this aspect, one could write a computation such as

$$\mathbf{let} \{s_1 \sim s_2 = s, _ = (s_1 \leftarrow t[x]), x \hat{=} s'_2 = s_2, y \hat{=} _ = s'_2\} \mathbf{in} y$$

which attempts to write, into the state, the term $t[x]$ where x is a future value of the same state! The recursion-free fragment of the language would not be normalizing.

To get around this problem, we introduce *sequential* linear logic, a variant of intuitionistic linear logic which allows sequencing constraints of the above form to be expressed. We then return to the state type constructor and give its correct type rules.

3.1 Sequential linear logic

A type assignment of sequential linear logic is given by the following syntax:

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

Γ_1, Γ_2 is interpreted as the tensor product $\Gamma_1 \otimes \Gamma_2$ and $\Gamma_1; \Gamma_2$ as a noncommutative tensor product $\Gamma_1 \triangleright \Gamma_2$. (“ \triangleright ” is read “before”). We use the convention that “,” binds closer than “;”. “ ϵ ” is the empty type assignment which is a unit for both “,” and “;”. Associated with a type assignment Γ is a partial order \leq_Γ on the typings in Γ such that $(x : A) <_\Gamma (y : B)$ whenever $(x : A)$ and $(y : B)$ occur in the left and right arguments of a “;” in Γ .

The salient type rules of sequential linear logic are as follows (we omit the term assignment

which remains the same as before):

$$\begin{array}{c}
\frac{\Gamma' \vdash C}{\Gamma \vdash C} \text{Exchange} \quad \text{if } \Gamma \text{ and } \Gamma' \text{ have the same typings and } <_{\Gamma} \subseteq <_{\Gamma'} \\
\\
\frac{}{A \vdash A} \text{Id} \quad \frac{\Gamma \vdash A \quad \Delta[A] \vdash B}{\Delta[\Gamma] \vdash B} \text{Cut} \\
\\
\frac{\Gamma[\epsilon] \vdash C}{\Gamma[!A] \vdash C} !\text{Weak} \quad \frac{\Gamma[A] \vdash C}{\Gamma[!A] \vdash C} !\text{Der} \quad \frac{\Gamma[!A, !A] \vdash C}{\Gamma[!A] \vdash C} !\text{Contr} \quad \frac{! \Gamma \vdash A}{! \Gamma \vdash !A} !\mathcal{R}
\end{array}$$

The other type rules are modified similarly. Note that the exchange rule allows permutations like $(\Gamma_1, \Gamma_2) \rightarrow (\Gamma_2, \Gamma_1)$ and $(\Gamma_1, \Gamma_2); \Gamma_3 \rightarrow \Gamma_1, (\Gamma_2; \Gamma_3)$ etc.

We introduce only a minimal amount of sequential features in our system in the interest of simplicity. (For instance, we don't add \triangleright as a type constructor). C. Retore [Ret92, Reta] has done an extensive study of this system. We await the publication of these results.

3.2 State types

Using the framework of sequential linear logic, the type rules of state types are expressed as follows:

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

Note that the “;” connective is introduced in the *st Write* and eliminated in *st Thread*.

The reduction rules for the state types are as follows:

$$\begin{array}{l}
(stW) \quad \mathbf{let } _ = st t \mathbf{ in } u \rightarrow \mathbf{let } _ = t \mathbf{ in } u \\
(stR) \quad \mathbf{let } z \hat{\sim} s' = st t \mathbf{ in } u \rightarrow \mathbf{let } \{z @ z' = t, s' = st z'\} \mathbf{ in } u \\
(stWr) \quad \mathbf{let } s' = (st t \leftarrow v) \mathbf{ in } u \rightarrow \mathbf{let } \{_ = t, s' = st v\} \mathbf{ in } u \\
(TW) \quad \mathbf{let } \{s_1 \hat{\sim} s_2 = t, _ = s_1\} \mathbf{ in } u \rightarrow \mathbf{let } \{s_2 = t\} \mathbf{ in } u \\
(TR) \quad \mathbf{let } \{s_1 \hat{\sim} s_2 = t, y \hat{\sim} s'_1 = s_1\} \mathbf{ in } u \rightarrow \mathbf{let } \{y \hat{\sim} s' = t, s'_1 \hat{\sim} s_2 = s'\} \mathbf{ in } u \\
(TWr) \quad \mathbf{let } \{s_1 \hat{\sim} s_2 = t, s'_1 = (s_1 \leftarrow v)\} \mathbf{ in } u \rightarrow \mathbf{let } \{s' = (t \leftarrow v), s'_1 \hat{\sim} s_2 = s'\} \mathbf{ in } u
\end{array}$$

The reductions (TW, TR, TWr) show how threading operates. When the first copy of the state is discarded, the state is identified with the second copy. Otherwise, the read and write operations on the first copy are propagated to the state.

3.3 Regenerative modality

Just as the modality “!” allows contraction, we can introduce another modality “†” that allows threading in place of contraction. We call this the *regenerative modality*. Its type rules are:

$$\frac{\Gamma[\epsilon] \vdash C}{\Gamma[\dagger A] \vdash C} \dagger\text{Weak} \quad \frac{\Gamma[A] \vdash C}{\Gamma[\dagger A] \vdash C} \dagger\text{Der} \quad \frac{\Gamma[\dagger A; \dagger A] \vdash C}{\Gamma[\dagger A] \vdash C} \dagger\text{Thread}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \dagger A} \dagger\mathcal{R} \quad \text{if } \Gamma \text{ contains only } !, \dagger \text{ or } st \text{ types}$$

Notice that a value of type $\dagger A$ can be built from other regenerative values (including states). *Dynamic* values like commands, procedures and objects are examples of such values. All such dynamic values can be used multiple times, sequentially. The behavior obtained in one use of a dynamic value is not necessarily consistent with that obtained in another use. In this respect, the regenerative modality is very different from “of course”. However, the proof-theoretic behavior is very similar.

The term assignment for the regenerative modality closely parallels that of “of course”:

$$\frac{\Gamma[\epsilon] \vdash u : C}{\Gamma[s : \dagger A] \vdash \mathbf{let} _ = s \mathbf{in} u : C} \dagger\text{Weak} \quad \frac{\Gamma[x : A] \vdash u : C}{\Gamma[s : \dagger A] \vdash \mathbf{let} \dagger x = s \mathbf{in} u : C} \dagger\text{Der}$$

$$\frac{\Gamma[s_1 : \dagger A; s_2 : \dagger A] \vdash u : C}{\Gamma[s : \dagger A] \vdash \mathbf{let} s_1 \sim s_2 = s \mathbf{in} u : C} \dagger\text{Thread}$$

$$\frac{\bar{x} : \Gamma \vdash t : A}{\bar{z} : \Gamma \vdash \dagger[\bar{x} = \bar{z}] t : \dagger A} \dagger\mathcal{R} \quad \text{if } \Gamma \text{ contains only } !, \dagger \text{ or } st \text{ types}$$

The reduction rules are also similar.

$$\begin{aligned} (\dagger W) \quad & \mathbf{let} \{ _ = \dagger[\bar{x} = \bar{v}] t \} \mathbf{in} u \rightarrow \mathbf{let} \{ _ = \bar{v} \} \mathbf{in} u \\ (\dagger D) \quad & \mathbf{let} \{ \dagger z = \dagger[\bar{x} = \bar{v}] t \} \mathbf{in} u \rightarrow \mathbf{let} \{ z = t[\bar{v}/\bar{x}] \} \mathbf{in} u \\ & \quad \{ \bar{x}_1 \sim \bar{x}_2 = \bar{v}, \\ (\dagger C) \quad & \mathbf{let} \{ s_1 \sim s_2 = \dagger[\bar{x} = \bar{v}] t \} \mathbf{in} u \rightarrow \mathbf{let} \{ s_1 = \dagger[\bar{x} = \bar{x}_1] t, \\ & \quad s_2 = \dagger[\bar{x} = \bar{x}_2] t \} \\ & \quad \mathbf{in} u \end{aligned}$$

In the above, we assume that all the inputs of the boxed term are regenerative. If there are also static inputs, the reduction ($\dagger C$) must introduce threading or contraction, as appropriate.

$$\begin{aligned} (\dagger C) \quad & \mathbf{let} s_1 \sim s_2 = \dagger[\bar{x} = \bar{v}, \bar{x}' = \bar{v}'] t \mathbf{in} u \rightarrow \mathbf{let} \{ \bar{x}_1 \sim \bar{x}_2 = \bar{v}, \bar{x}'_1 @ \bar{x}'_2 = \bar{v}', \\ & \quad s_1 = \dagger[\bar{x} = \bar{x}_1, \bar{x}' = \bar{x}'_1] t, \\ & \quad s_2 = \dagger[\bar{x} = \bar{x}_2, \bar{x}' = \bar{x}'_2] t \} \\ & \quad \mathbf{in} u \end{aligned}$$

We also have the various commutation equivalences:

$$\begin{aligned} (\dagger\dagger\text{Comm}) \quad & \dagger[\bar{x} = \bar{v}, x = \dagger[\bar{y} = \bar{w}] v] t \equiv \dagger[\bar{x} = \bar{v}, \bar{y} = \bar{w}] (t[(\dagger[\bar{y} = \bar{y}] v)/x]) \\ (\dagger!\text{Comm}) \quad & \dagger[\bar{x} = \bar{v}, x = ![\bar{y} = \bar{w}] v] t \equiv \dagger[\bar{x} = \bar{v}, \bar{y} = \bar{w}] (t[![\bar{y} = \bar{y}] v]/x]) \\ (\dagger st \text{ Comm}) \quad & \dagger[\bar{x} = \bar{v}, x = st ![\bar{y} = \bar{w}] v] t \equiv \dagger[\bar{x} = \bar{v}, \bar{y} = \bar{w}] (t[(st ![\bar{y} = \bar{y}] v)/x]) \end{aligned}$$

3.4 Examples

Suppose we wish to define a counter object which maintains an internal state variable and increments it on each use of the counter. We define it as a regenerative value as follows:

$$\mathit{counter} = \dagger[v = \mathit{st} (!0)] \mathbf{let} \{x \hat{\sim} s = v, x_1 @ x_2 = x, _ = (s \leftarrow \mathit{addone}(x_1))\} \mathbf{in} x_2$$

where $\mathit{addone}(x_1)$ is short for $![k = x_1] (k + 1)$. The state variable v is of type $\mathit{st} (\mathbf{int})$, and the counter 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. For example, if we use the counter as

$$\mathbf{let} \{\dagger i \hat{\sim} c' = \mathit{counter}, \dagger j \hat{\sim} _ = c'\} \mathbf{in} (i, j)$$

we obtain $(!0, !1)$.

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

$$\begin{aligned} \mathit{updown} = \dagger[v = \mathit{st} (!0)] & \langle \mathbf{let} \{(x_1 @ x_2) \hat{\sim} s = v, _ = (s \leftarrow \mathit{addone}(x_1))\} \mathbf{in} x_2, \\ & \mathbf{let} \{(x_1 @ x_2) \hat{\sim} s = v, _ = (s \leftarrow \mathit{subone}(x_1))\} \mathbf{in} x_2 \rangle \end{aligned}$$

This is of type $\dagger(!\mathit{int} \& !\mathit{int})$. More interestingly, here is an up-counter that also provides an operation to set its current state:

$$\begin{aligned} \mathit{settable} = \dagger[v = \mathit{st} (!0)] & \langle \mathbf{let} \{(x_1 @ x_2) \hat{\sim} s = v, _ = (s \leftarrow \mathit{addone}(x_1))\} \mathbf{in} x_2, \\ & \lambda x. \mathbf{let} _ = (v \leftarrow x) \mathbf{in} () \rangle \end{aligned}$$

whose type is $\dagger(!\mathit{int} \& (!\mathit{int} \multimap \mathbf{1}))$.

Commands are modelled by the type $\dagger\mathbf{1}$. They act on their internal state without returning a result of any consequence. A “doubling” combinator that takes a command and executes it twice is defined as follows:

$$\begin{aligned} \mathit{double} : \dagger\mathbf{1} \multimap \dagger\mathbf{1} \\ \mathit{double} c = \dagger[\dagger e_1 \hat{\sim} \dagger e_2 = c] \mathbf{let} \{() = e_1, () = e_2\} \mathbf{in} () \end{aligned}$$

A general sequencing combinator is

$$\begin{aligned} \mathit{seq} : \dagger(\mathbf{1} \& \mathbf{1}) \multimap \dagger\mathbf{1} \\ \mathit{seq} d = \dagger[\dagger p_1 \hat{\sim} \dagger p_2 = d] \mathbf{let} \{() = \pi_1 p_1, () = \pi_2 p_2\} \mathbf{in} () \end{aligned}$$

Note that the argument of seq is of type $\dagger(\mathbf{1} \& \mathbf{1})$ rather than $\dagger\mathbf{1} \otimes \dagger\mathbf{1}$. The latter type would not allow the two commands to act on the same state. If the two commands act on separate states, they can be combined using a parallel composition combinator:

$$\begin{aligned} \mathit{par} : \dagger\mathbf{1} \otimes \dagger\mathbf{1} \multimap \dagger\mathbf{1} \\ \mathit{par}(c_1, c_2) = \dagger[\dagger e_1 = c_1, \dagger e_2 = c_2] \mathbf{let} \{() = e_1, () = e_2\} \mathbf{in} () \end{aligned}$$

A “Kleisli-style” sequencing operator, used in monad approaches to state [Wad92], is also easy to define:

$$\begin{aligned} \mathit{bind} : \dagger(A \& (A \multimap B)) \multimap \dagger B \\ \mathit{bind} d = \dagger[\dagger p_1 \hat{\sim} \dagger p_2 = d] \mathbf{let} \{x = \pi_1 p_1, y = \pi_2 p_2 x\} \mathbf{in} y \end{aligned}$$

Our final example is a combinator for “while loops” defined using recursion:

$$\begin{aligned} \text{while} & : \dagger(\text{bool} \ \& \ \mathbf{1}) \multimap \dagger \mathbf{1} \\ \text{while } d & = \dagger[\dagger p_1 \rightsquigarrow d_1 = d] \text{ if } \pi_1 p_1 \text{ then let } \{\dagger p_2 \rightsquigarrow d_2 = d_1, () = \pi_2 p_2\} \text{ in while } d_2 \\ & \quad \text{else let } _ = d_1 \text{ in } () \end{aligned}$$

4 Coherent semantics

Recall, from [GLT89, Gir87], that the “web” of a coherent space A is a reflexive undirected graph given by the data:

- a set $|A|$ of vertices called “tokens”, and
- a set of arcs determined by a binary relation called “coherence”. If there is an arc between $a, a' \in |A|$, we write $a \circlearrowleft a' \text{ [mod } A]$.

We also use the notation:

$$\begin{aligned} a \frown a' \text{ [mod } A] & \Leftrightarrow a \circlearrowleft a' \text{ [mod } A] \wedge a \neq a' \\ a \smile a' \text{ [mod } A] & \Leftrightarrow \neg(a \circlearrowleft a' \text{ [mod } A]) \\ a \asymp a' \text{ [mod } A] & \Leftrightarrow a \smile a' \text{ [mod } A] \vee a = a' \end{aligned}$$

The coherent space is the set of cliques of the web, made into a cpo using inclusion as the order. Since coherent spaces and webs determine each other, we loosely refer to the webs themselves as coherent spaces.

The coherent spaces for the various types are defined as follows. All except the last three clauses are standard:

$$\begin{aligned} |\mathbf{1}| & = \{*\} \\ |A \otimes B| & = |A| \times |B| & (a_1, b_1) \circlearrowleft (a_2, b_2) & \Leftrightarrow a_1 \circlearrowleft a_2 \wedge b_1 \circlearrowleft b_2 \\ |A \multimap B| & = |A| \times |B| & (a_1, b_1) \frown (a_2, b_2) & \Leftrightarrow a_1 \circlearrowleft a_2 \Rightarrow b_1 \frown b_2 \\ |!A| & = A_{\text{fin}} & X \circlearrowleft Y \text{ [mod } !A] & \Leftrightarrow \forall a \in X. \forall b \in Y. a \circlearrowleft b \\ |A \triangleright B| & = |A| \times |B| & (a_1, b_1) \frown (a_2, b_2) & \Leftrightarrow a_1 \frown a_2 \vee a_1 = a_2 \wedge b_1 \frown b_2 \\ |\dagger A| & = |A|^* & s \frown r \text{ [mod } \dagger A] & \Leftrightarrow (s = \epsilon \vee_{\text{excl}} r = \epsilon) \vee \\ & & & s = tas' \wedge r = tbr' \wedge a \frown b \text{ [mod } A] \\ & & & \text{for some } t, s', r' \in |A|^*, \text{ and } a, b \in |A| \\ |st A| & = A_{\text{state}} & s \frown r \text{ [mod } st A] & \Leftrightarrow (s = \epsilon \vee_{\text{excl}} r = \epsilon) \vee \\ & & & s = t(i, X)s' \wedge r = t(j, Y)r' \wedge (i, X) \frown (j, Y) \\ & & & \text{for some } t, s', r' \in A_{\text{state}}, X, Y \in |!A|, \\ & & & \text{and } (i, j) \in \{0, 1\} \end{aligned}$$

In the above, A_{fin} is the set of finite coherent sets over $|A|$. A_{state} is a subset of $(|!A| + |!A|)^*$ called the set of *state sequences*. A state sequence for an *initialization* $X \in |!A|$ is either the empty sequence, a sequence $(0, X')s$ where $X' \subseteq X$ and s is a state sequence for initialization X , or a sequence $(1, X')s$ where s is a state sequence for initialization X' . A member of A_{state} is a state sequence for some initialization $X \in |!A|$. Evidently, $(0, X')$ denotes a read operation that extracts the information X' from the state, and $(1, X')$ denotes a write operation that places the information X' in the state. The coherence relation on such pairs is defined as:

$$\begin{aligned} (0, X) \frown (1, Y) & \\ (0, X) \frown (0, Y) & \Leftrightarrow X \frown Y \text{ [mod } !A] \\ (1, X) \frown (0, Y) & \\ (1, X) \frown (1, Y) & \Leftrightarrow X \smile Y \text{ [mod } !A] \end{aligned}$$

The semantics of a typing assertion $\Gamma \vdash t : A$ is a linear map $\Gamma \multimap A$. We represent the tokens of $|\Gamma|$ as *environments*: *i.e.*, finite maps from variables in Γ to tokens of appropriate types. The symbol η is used to range over environments. η_\perp denotes the empty environment and $\eta \downarrow \Gamma$ denotes the restriction of η to the variables in Γ . The coherence relation of environments is defined by:

$$\begin{aligned} \eta_\perp \circ \eta_\perp & [\text{mod } \epsilon] \\ \eta \circ \eta' & [\text{mod } x : A] \Leftrightarrow \eta(x) \circ \eta'(x) [\text{mod } A] \\ \eta \circ \eta' & [\text{mod } \Gamma_1, \Gamma_2] \Leftrightarrow \eta \downarrow \Gamma_1 \circ \eta' \downarrow \Gamma_1 [\text{mod } \Gamma_1] \wedge \eta \downarrow \Gamma_2 \circ \eta' \downarrow \Gamma_2 [\text{mod } \Gamma_2] \\ \eta \frown \eta' & [\text{mod } \Gamma_1; \Gamma_2] \Leftrightarrow \eta \downarrow \Gamma_1 \frown \eta' \downarrow \Gamma_1 [\text{mod } \Gamma_1] \vee \\ & \eta \downarrow \Gamma_1 = \eta' \downarrow \Gamma_1 \wedge \eta \downarrow \Gamma_2 \frown \eta' \downarrow \Gamma_2 [\text{mod } \Gamma_2] \end{aligned}$$

We also extend the coherence relation for type assignments with holes: $\eta_1 \circ \eta_2$ $[\text{mod } \Delta[\]]$ if, for all type assignments $\Delta[\Gamma]$ and all extensions $\eta'_1, \eta'_2 \in |\Delta[\Gamma]|$ of η_1 and η_2 , we have $\eta'_1 \circ \eta'_2$ $[\text{mod } \Delta[\Gamma]]$. Similarly, $\eta_1 \frown \eta_2$ and other relations can be extended.

We write the pairs of a map $\Gamma \multimap A$ using the suggestive notation $\eta \vdash a$ instead of (η, a) . The meaning of derivable typing assertions is defined, inductively, as follows.

Id $[[x : A \vdash x : A]]$ is $\{(\eta_\perp[x \mapsto a] \vdash a) : a \in |A|\}$.

Cut $[[\Delta[\Gamma] \vdash \text{let } x = t \text{ in } u : C]]$ is the set of $\eta \vdash c$ (for $\eta \in |\Delta[\Gamma]|$) where

$$\begin{aligned} \eta \downarrow \Gamma \vdash a & \in [[\Gamma \vdash t : A]] \\ (\eta \downarrow \Delta)[x \mapsto a] \vdash c & \in [[\Delta[x : A] \vdash u : C]] \end{aligned}$$

for some $a \in |A|$.

!R $[[z : !C \vdash ! [x = z] t : !A]]$ is the set of $\eta_\perp[z \mapsto \bigcup_i X_i] \vdash \{a_i\}_i$ where $\bigcup_i X_i$ is coherent and

$$\eta_\perp[x \mapsto X_i] \vdash a_i \in [[x : !C \vdash t : A]]$$

!Weak $[[\Gamma[z : !A] \vdash \text{let } _ = z \text{ in } u : C]]$ is the set of $\eta[z \mapsto \emptyset] \vdash c$ where

$$\eta \vdash c \in [[\Gamma[\] \vdash u : C]]$$

!Der $[[\Gamma[z : !A] \vdash \text{let } !x = z \text{ in } u : C]]$ is the set of $\eta[z \mapsto \{a\}] \vdash c$ where

$$\eta[x \mapsto a] \vdash c \in [[\Gamma[x : A] \vdash u : C]]$$

!Contr $[[\Gamma[z : !A] \vdash \text{let } x_1 @ x_2 = z \text{ in } u : C]]$ is the set of $\eta[z \mapsto X_1 \cup X_2] \vdash c$ where

$$\eta[x_1 \mapsto X_1, x_2 \mapsto X_2] \vdash c \in [[\Gamma[x_1 : !A, x_2 : !A] \vdash u : C]]$$

†R $[[z : \dagger B, z' : st C, z'' : !D \vdash \dagger [x = z, x' = z', x'' = z''] t : \dagger A]]$ is the set of

$$\eta_\perp[z \mapsto s_1 \dots s_k, z' \mapsto s'_1 \dots s'_k, z'' \mapsto X_1 \cup \dots \cup X_k] \vdash a_1 \dots a_k$$

where $s'_1 \dots s'_k$ is a state sequence, $X_1 \cup \dots \cup X_k$ is coherent and

$$\eta_\perp[x \mapsto s_i, x' \mapsto s'_i, x \mapsto X_i] \vdash a_i \in [[x : \dagger B, x' : st C, x'' : !D \vdash t : A]]$$

\dagger Weak $\llbracket \Gamma[z : \dagger A] \vdash \mathbf{let} _ = z \mathbf{in} u : C \rrbracket$ is the set of $\eta[z \rightarrow \epsilon] \vdash c$ where

$$\eta \vdash c \in \llbracket \Gamma[] \vdash u : C \rrbracket$$

\dagger Der $\llbracket \Gamma[z : \dagger A] \vdash \mathbf{let} \dagger x = z \mathbf{in} u : C \rrbracket$ is the set of $\eta[z \rightarrow (a)] \vdash c$ where

$$\eta[x \rightarrow a] \vdash c \in \llbracket \Gamma[x : A] \vdash u : C \rrbracket$$

\dagger Thread $\llbracket \Gamma[z : \dagger A] \vdash \mathbf{let} x_1 \rightsquigarrow x_2 = z \mathbf{in} u : C \rrbracket$ is the set of $\eta[z \rightarrow s_1 s_2] \vdash c$ where

$$\eta[x_1 \rightarrow s_1, x_2 \rightarrow s_2] \vdash c \in \llbracket \Gamma[x_1 : \dagger A; x_2 : \dagger A] \vdash u : C \rrbracket$$

$st \mathcal{R}$ $\llbracket \Gamma \vdash st t : st A \rrbracket$ is the set of $\eta \vdash s$ where s is a state sequence for initialization X and

$$\eta \vdash X \in \llbracket \Gamma \vdash t : !A \rrbracket$$

st Weak and st Thread are similar to \dagger Weak and \dagger Thread respectively, except that in st Thread, the sequences $s_1 s_2$ are required to be state sequences.

st Read $\llbracket \Gamma[z : st A] \vdash \mathbf{let} x \rightsquigarrow y = z \mathbf{in} u : C \rrbracket$ is the set of $\eta[z \rightarrow (0, X)s] \vdash c$ where $(0, X)s$ is a state sequence and

$$\eta[x \rightarrow X, y \rightarrow s] \vdash c \in \llbracket \Gamma[x : !A; y : st A] \vdash u : C \rrbracket$$

st Write $\llbracket \Delta[\Gamma; z : st A] \vdash \mathbf{let} x = (z := t) \mathbf{in} u : C \rrbracket$ is the set of $\eta[z \rightarrow (1, X)s] \vdash c$ where $(1, X)s$ is a state sequence and

$$\begin{aligned} (\eta \downarrow \Delta)[x \rightarrow s] \vdash c &\in \llbracket \Delta[x : st A] \vdash u : C \rrbracket \\ \eta \downarrow \Gamma \vdash X &\in \llbracket \Gamma \vdash t : !A \rrbracket \end{aligned}$$

Theorem 1 For all derivable typing assertions, $\llbracket \Gamma \vdash t : A \rrbracket$ is a linear map $\Gamma \multimap A$.

Proof: The requirement is that for all distinct $\eta \vdash a$ and $\eta' \vdash a'$ in $\llbracket \Gamma \vdash t : A \rrbracket$, $\eta \circ \eta' \Rightarrow a \frown a'$. The proof is by induction on the derivation of t . We show a few crucial cases.

Id Let a, a' be distinct tokens of $|A|$. Then, obviously, $a \circ a' \Rightarrow a \frown a'$.

Cut Suppose $\eta \vdash c$ and $\eta' \vdash c'$ are distinct pairs in $\llbracket \Delta[\Gamma] \vdash u[t/x] : C \rrbracket$. Then, there are a and a' in $|A|$ such that

$$\begin{aligned} \eta \downarrow \Gamma \vdash a, \quad \eta' \downarrow \Gamma \vdash a' &\in \llbracket \Gamma \vdash t : A \rrbracket \\ (\eta \downarrow \Delta)[x \rightarrow a] \vdash c, \quad (\eta' \downarrow \Delta)[x \rightarrow a'] \vdash c' &\in \llbracket \Delta[x : A] \vdash u : C \rrbracket \end{aligned}$$

By inductive hypothesis,

$$(\eta \downarrow \Gamma) \circ (\eta' \downarrow \Gamma) \text{ [mod } \Gamma] \Rightarrow a \frown a' \text{ [mod } A] \quad (1)$$

and

$$(\eta \downarrow \Delta)[x \rightarrow a] \circ (\eta' \downarrow \Delta)[x \rightarrow a'] \text{ [mod } \Delta[x : A]] \Rightarrow c \frown c' \text{ [mod } C] \quad (2)$$

We need to show $\eta \circ \eta' \text{ [mod } \Delta[\Gamma]] \Rightarrow c \frown c' \text{ [mod } C]$.

We have either $a \asymp a'$ or $a \frown a'$. If $a \asymp a'$ then, by (1), $\eta \downarrow \Gamma \smile \eta' \downarrow \Gamma$. So, the assumption $\eta \circ \eta' \text{ [mod } \Delta[\Gamma]]$ means $(\eta \downarrow \Delta) \circ (\eta' \downarrow \Delta) \text{ [mod } \Delta[]]$, and, by (2), $c \frown c'$. If $a \frown a'$ then, by (2), either $c \frown c'$ or $(\eta \downarrow \Delta) \smile (\eta' \downarrow \Delta) \text{ [mod } \Delta[]]$. In both cases, we have the conclusion.

□

Theorem 2 The reduction semantics is sound with respect to coherent semantics. That is, $\Gamma \vdash t : A$ and $t \longrightarrow t'$ then $\llbracket \Gamma \vdash t : A \rrbracket = \llbracket \Gamma \vdash t' : A \rrbracket$.

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]. We will finesse the issue of conjunctive types, which play a role in [Rey89], but seem orthogonal to the issues of this paper, and use a formulation that closely follows O’Hearn [O’H91].

Interference is the higher-order analogue of the familiar “aliasing” phenomenon. In general, it occurs when two apparently independent phrases (such as a function and its argument) act on the “same” state. In such a case, the two phrases affect the state in an arbitrarily interleaved fashion and the resulting behavior is difficult to model. For this reason, we restrict attention to noninterfering programs defined using SCI. O’Hearn [O’H91] has already noted that Reynolds’s restrictions in SCI amount to prohibiting contraction for “active” types. In other words, he showed a correspondence between the notion of active types in SCI and the notion of linear types in linear logic. Our semantics builds on this work and translates active types to regenerative types.

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 unchanged. The syntax of type terms is as follows:

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

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:²

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

All passive types become static types in the linear setting and all active types become regenerative types. A phrase with the typing $\Theta \vdash p : \theta$ is translated to a term with linear typing:

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

²For simplicity, we illustrate the translation that models “call-by-value” reduction semantics. A translation that models “call-by-name” semantics is indicated in the sequel.

$$\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} \\
\frac{\Theta, x : \delta \text{exp} \vdash q : \theta'}{\Theta, z : \delta \text{var} \vdash q[z/x] : \theta'} \text{Deref} \quad \frac{\bar{x}_1 : \Theta \vdash v : \delta \text{var} \quad \bar{x}_2 : \Theta \vdash e : \delta \text{exp}}{\bar{x} : \Theta \vdash v[\bar{x}/\bar{x}_1] := e[\bar{x}/\bar{x}_2] : \text{comm}} \text{Assign} \\
\frac{\bar{x}_1 : \Theta \vdash c_1 : \text{comm} \quad \bar{x}_2 : \Theta \vdash c_2 : \text{comm}}{\bar{x} : \Theta \vdash (c_1[\bar{x}/\bar{x}_1]; c_2[\bar{x}/\bar{x}_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)

where Θ^* is the translation of the typings of Θ combined using the “;” connective. The “;” connective never occurs in the translation of a phrase.

We now show the translation of the SCI phrases by induction on their type derivations. The rules **Id**, **Exchange**, **Weak** and **Contr** have obvious translations. For the others:

$$\begin{array}{ll}
\text{Cut} & q[p/x]^* = \mathbf{let } x = p^* \mathbf{ in } q^* \\
\text{New} & (\text{new } \delta x. c)^* = \mathbf{let } x = st \text{ (!undef}_\delta) \mathbf{ in } c^* \\
\text{Deref} & (q[z/x])^* = \mathbf{let } x \hat{_} = z \mathbf{ in } q^* \\
\text{Assign} & (v[\bar{x}/\bar{x}_1] := e[\bar{x}/\bar{x}_2])^* = \dagger[\bar{x}_2 \rightsquigarrow \bar{x}_1 = \bar{x}] \mathbf{let } _ = (v^* \leftarrow e^*) \mathbf{ in } () \\
\text{Seq} & (c_1[\bar{x}/\bar{x}_1]; c_2[\bar{x}/\bar{x}_2])^* = \dagger[\bar{x}_1 \rightsquigarrow \bar{x}_2 = \bar{x}] \mathbf{let } \dagger() = c_1^*, \dagger() = c_2^* \mathbf{ in } () \\
\text{Par} & (c_1 \parallel c_2)^* = \dagger[\bar{x} = \bar{x}] \mathbf{let } \dagger() = c_1^*, \dagger() = c_2^* \mathbf{ in } () \\
\rightarrow_P \mathcal{R} & (\lambda x. p)^* = \mathbf{!}[\bar{x} = \bar{x}] \lambda x. p^* \\
\rightarrow_P \mathcal{L} & q[fp/z]^* = \mathbf{let } !x = f, z = xp^* \mathbf{ in } q^* \\
\rightarrow \mathcal{R} & (\lambda x. p)^* = \dagger[\bar{x} = \bar{x}] \lambda x. p^* \\
\rightarrow \mathcal{L} & q[fp/z]^* = \mathbf{let } \dagger x = f, z = xp^* \mathbf{ in } q^*
\end{array}$$

In the translation of **New**, we assume that there is a constant undef_δ , for each data type δ , denoting the value of an uninitialized state variable. The bindings $\bar{x} = \bar{x}$ in the translation of **Par** etc. denote the mandatory renaming of all the free variables required for boxed terms.

The correctness of the translation is established as follows. For space limitations, we merely indicate the method. The semantics of Algol-like language is defined in two parts: a reduction system for the applicative aspects and a state transition system for the command execution. The latter is best expressed in the natural semantics style with judgements of the form $(\sigma, c) \downarrow \sigma'$ where

σ and σ' are the initial and final states. A state $\{\bar{x} \mapsto \bar{k}\}$ is represented in the regenerative type system as a collection of coequations of the form $\bar{x} = st (!\bar{k})$.

Proposition 3 Given a command $\Theta \vdash c : \mathbf{comm}$ and states $\sigma = \{\bar{x} \mapsto \bar{k}\}$ and $\sigma' = \{\bar{x} = \bar{k}'\}$, there exists a command d such that $c \longrightarrow d$ and $(\sigma, d) \downarrow \sigma'$ if and only if

$$\dagger[\bar{x} = st (!\bar{k})] c^* \longrightarrow \dagger[_ = st (!\bar{k}')] ()$$

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.

To model call-by-name semantics, we define can two translations for types denoted $()^\circ$ and $()^*$ respectively.

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

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. Other extensions such as passive and active product types etc. are similarly definable. We omit the details from this summary.

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. While the coherent semantics of Section 4 is a beginning, we need more intuitive semantics such as game semantics [AJ92]. A general characterization in terms of categorical models would also be of interest.

The relation between the regenerative semantics of imperative languages and the “possible world” semantics of Reynolds and Oles needs to be investigated. It is also important to know whether the regenerative type system can model interference (apparent interference disambiguated by explicit evaluation order). Once the semantic issues regarding the present approach are worked out, it would be possible to study the full abstraction issues regarding the semantics of local variables etc.

Convenient concrete syntax must be found for functional languages based on the regenerative type system. 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, Wad92] and regenerative types. (The functor \dagger forms a comonad). It is very well possible that a reformulation of regenerative types in terms of monads gives a better syntax.

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.
- [AJ92] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. Manuscript, Imperial College, 1992. (available by ftp from theory.doc.ic.ac.uk, directory /theory/papers/Abramsky).
- [BBdPH92] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. Term assignment for intuitionistic linear logic. Technical Report 262, University of Cambridge, Computer Laboratory, Aug 1992. (available by ftp from theory.doc.ic.ac.uk:/theory/papers/dePaival/taill.dvi).
- [BG91] C. Brown and D. Gurr. Relations and non-commutative linear logic. Technical Report DAIMI PB-372, Aarhus University, Denmark, Nov 1991.
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.
- [GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Univ. Press, Cambridge, 1989.
- [Gor79] M. J. C. Gordon. *The Denotational Description of programming languages*. Springer-Verlag, New York, 1979.
- [HMT83] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free list free? In *Tenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 245–257. ACM, 1983.
- [Laf88] Y. Lafont. The linear abstract machine. *Theoretical Comp. Science*, 59:157–180, 1988.
- [Mog91] E. Moggi. Notions of computations 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.
- [O’H91] P. W. O’Hearn. Linear logic and interference control. In D. H. Pitt et. al., editor, *Category Theory and Computer Science*, volume 350 of (*LNCS*), pages 74–93. Springer-Verlag, Berlin, 1991. (LNCS Vol. 530).
- [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.

- [OT92] P. W. O’Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, pages 217–238. Cambridge University Press (London Math. Soc. Lecture Notes Series), Cambridge, England, 1992.
- [Reta] C. Retore. Forthcoming ph. d. thesis. Equipe de Logique, Department de Mathematiques, Universite Paris 7, Paris.
- [Retb] C. Retore. Private communication regarding noncommutative tensor. Sep, 1992.
- [Ret92] C. Retore. Graph theory from linear logic. Manuscript, Equipe de Logique, Department de Mathematiques, Universite Paris 7, Paris, Oct 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 *Intern. Colloq. Automata, Languages. and Programming*, pages 704–722. Springer-Verlag, 1989. (LNCS Vol. 372).
- [SR91] V. Swarup and U. S. Reddy. A logical view of assignments. In M. J. O’Donnell, editor, *Constructivity in Computer Science*, pages 127–141, San Antonio, Texas, June 1991. Trinity University.
- [SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, pages 192–214. Springer-Verlag, Berlin, 1991. (LNCS Vol. 523).
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Wad91] P. Wadler. There’s no substitute for linear logic. Manuscript, Department of Computing Science, University of Glasgow, Dec 1991.
- [Wad92] P. Wadler. The essence of functional programming. In *ACM Symp. on Princ. of Program. Lang.*, 1992.
- [Yet90] D. Yetter. Quantales and non-commutative linear logic. *J. Symbolic Logic*, 55(I):41–64, 1990.