Math. Struct. in Comp. Science (199?), *vol.* ?, *pp.* ?–?

A Model for Syntactic Control of Interference

P. W. O'Hearn
School of Computer and Information Science
Syracuse University, Syracuse, NY, USA 13224-4100

Received ????

Two imperative programming language phrases *interfere* when one writes to a storage variable that the other reads from or writes to. Reynolds has described an elegant linguistic approach to controlling interference in which a refinement of typed λ-calculus is used to limit sharing of storage variables; in particular, different identifiers are required never to interfere. This paper examines semantic foundations of the approach.

We describe a category that has (an abstraction of) interference information built into all objects and maps. This information is used to define a "tensor" product whose components are required never to interfere. Environments are defined using the tensor, and procedure types are obtained via a suitable adjunction. The category is a model of intuitionistic linear logic. Reynolds' concept of passive type – i.e. types for phrases that don't write to any storage variables – is shown to be closely related, in this model, to Girard's "of course" modality.

1. Introduction

The ability to update the state is the source of much of the flexibility and efficiency of imperative programming, but also many of its difficulties. Undisciplined sharing of storage variables can lead to subtle program errors that are difficult to detect and trace, and just the possibility of this kind of interference, even when absent, can have a significant detrimental impact on ease of reading and reasoning about programs (e.g. Hoare, 1974b; Reynolds, 1978). The well-known dishevelment caused by variable aliasing – when different identifiers name the same storage variable – in Hoare-style proof systems for reasoning about assignment and procedures (Hoare, 1971) is a kind of theoretical symptom of the problems brought on by interference.

There are related, and perhaps even more vivid, problems in the presence of concurrency, where uncontrolled interference can be a serious obstacle to program predictability. As a result, a number of authors (e.g. Hoare (1974a) and Brinch Hansen (1973)) have argued that all interference between concurrent processes should be mediated by, say, monitors or communication primitives.

Reynolds' (1978,1989) syntactic control of interference approaches these issues from a linguistic viewpoint, by using syntactic constraints that limit interaction between different program parts. The aim is not to eliminate interference entirely, but is to make it more manageable by arranging matters so that (a conservative approximation to) non-interference is easy for the programmer or compiler to recognize. Programs can still use state, but there is a tighter control over sharing of storage variables.

The purpose of this paper is to show that the interference constraints that form the basis of the approach have good semantic properties. This is done in two steps. First, we describe a category in which interference properties of semantic entities, as well as types, are made explicit. Then, using this category we examine interference control principles from a semantic perspective, and familiar category-theoretic structure falls out quite directly. The structure we obtain amounts to a model of (intuitionistic) linear logic (Girard, 1987), and a bit more.

A model for syntactic control of interference was previously proposed in (Tennent, 1983). This was basically an untyped model, in the "Curry" style. While this kind of interpretation can be useful for proving properties of interference constraints, we believe that the "Church-style" model presented here gives a more satisfactory semantic account of the type-theoretic basis of the approach (especially in the close relationship between categorical structure and syntactic constraints).

Controlling interference is an old problem in programming languages. It dates as far back as Fortran and Concurrent Pascal, with their anti-aliasing restrictions (Brinch Hansen, 1973; ANSI, 1978; see also Hoare, 1971), and plays an important role in such languages as Euclid, Turing and occam (Cordy, 1984; Popek et. al., 1977; Holt et. al., 1987; INMOS, 1988). We will not attempt to survey here the growing body of recent work on interference control and related topics. The reader is referred to the papers by Lucassen and Gifford (1988), Wadler (1990,1992), Guzmàn and Hudak (1990), Swarup et. al. (1991), and their references for further discussion of this.

Syntactic control of interference represents an important step toward our understanding of how a programming language could provide the benefits of state, while avoiding many of the difficulties it causes in present-day languages. It should be mentioned, however, that the approach has not yet been perfected. In particular, there are presently difficulties with recursion and jumps (Reynolds, 1989); this will be discussed briefly in Section 9. Nevertheless, we feel it remarkable that the syntactic restrictions at the core of the approach are semantically so well-behaved, especially since interference is often regarded as a low-level operational concept. This encourages our belief in the possibility of clean, and yet practical, methods for harnessing the power of assignment.

We will outline the main features of our model later in this introductory section, after discussing background on interference control.

1.1. Background on Interference Control

Syntactic control of interference is based on a refinement of typed λ-calculus, where typing constraints are used to limit the manner in which interference can arise. The constraints are motivated by a number of "principles of interference control" described by Reynolds, which are chosen so as to ensure that interference is easily detectable.

The first principle is

I. *if no identifier free in phrase P interferes with any identifier free in phrase Q, then P and Q don't interfere.*

This is an assumption about the nature of the language which says, in effect, that all "channels" of interference must be named by identifiers. In particular, closed terms don't interfere with any other terms. (We use "phrase" and "term" interchangeably.)

The second principle is what necessitates syntactic constraints.

II. *distinct identifiers never interfere.*

Combined with I, this provides the programmer with a particularly simple method of predicting non-interference, and meets head on problems caused by such phenomena as aliasing of storage variables. For example, Principle II implies that running the assignment statements $x := 1$ and $y := 2$ in any order, or in parallel, is determinate at an appropriate level of abstraction. This would not be the case if aliasing between x and y was allowed, because the same storage variable would be destructively altered by each statement.

Principle II may seem overly restrictive at first, but interference is not forbidden altogether. For example, an abstract data structure can be represented by a collection of interfering procedures that are different qualifications of the *same* identifier, such as different components of a "record" or "object" (Reynolds, 1978; Dahl, 1972). In effect, interference is treated as an exceptional case, which requires effort from the programmer to indicate explicitly. In contrast, most imperative languages have (the possibility of) interference as the default case, with determination of non-interference requiring effort.

The final principle allows a limited amount of sharing.

III. *passive phrases, which don't write to any (global) variables, don't interfere with one another.*

For example, if y is a "read-only" expression then, according to all of the Principles I-III, the assignment statements $x := y + 1$ and $z := y + 2$ won't interfere. Note that sharing of read access is consistent with Principle II: two identifiers can have read access to the same storage variable, as long as neither has write access to it.

Notice that these principles do not attempt to predict *all* interference relationships between program phrases, such as whether different uses of the same non-passive identifier interfere. More "fine-grained" interference detection is often used in parallel program optimization, e.g. to determine if different uses of an array identifier don't interfere

(Padua and Wolfe, 1989). Of course, linguistic interference control and algorithmic interference detection have different aims (and should be considered complementary), with simplicity being of the upmost importance in the former, enabling a programmer to recognize interference *easily* in many cases.

1.2. Semantic Aspects of State Dependence

The backbone of our analysis of interference control is a notion of the *support* of a semantic entity a as a pair R, W of finite sets of locations (storage variables). Intuitively, R consists of the locations that a reads from, and W consists of the locations that a writes to. Once support is defined it is then straightforward to formulate a semantic counterpart of non-interference, which we will call *independence*, by comparing the supports of semantic entities to determine whether one writes to a location that another reads from or writes to.

This notion of support is inspired in part by earlier work of Halpern et. al. (1983) and Meyer and Sieber (1988). There the support of a semantic entity is identified as, intuitively, the set of locations upon which it depends. Our formulation is different in two ways. The first is that we separate the read and write capabilities of locations; this will turn out to be crucial for the treatment of passivity. The second is that our formulation uses the functor-category approach to program semantics initiated by Reynolds (1981) and Oles (1982, 1985). In fact, the treatment of support given by Meyer and Sieber can be considered to have functors at its core. Making this explicit leads to a cleaner and simpler treatment. We refer to the expository article (O'Hearn and Tennent, 1992) for further discussion of this. See also (Tennent, 1990,1991; O'Hearn and Tennent 1992,1993) for other related work on functors and non-interference.

The reader may wonder why we are invoking this category-theoretic machinery. The reason is that the notion of support is surprisingly difficult to make precise in standard cpo semantics. There is no trouble defining it for meanings of basic program types, such as *commands* (i.e. state transformations) or *expressions*, but getting a definition that works well at higher types is problematic (try it!). As in the just-mentioned work on the semantics of state, the key role of functors here will be to explicitly include support information in the *definition* of what count as meanings at higher types.

The basic idea of our semantics is to have a category of "possible worlds" in which the objects are supports. The worlds are used to partition semantic entities according to the variables that they read or write, and semantic maps are required to respect this structure. This is done by interpreting types as functors from this category to a category of domains, and terms as natural transformations between these functors. So types come with interference information "built-in." This opens up the possibility of defining type constructors, as operations on a functor category, in a way that takes interference information into account.

1.3. Overview of the Model

A main aim of our model will be to make Principle II evident. This is an an assumption about the nature of the *environment*, one that we shall take as our starting point. We do this by defining the notion of environment so that the *only* environments are ones in which distinct identifiers denote non-interfering (independent) meanings. The principle is thus regarded as a prior assumption about the semantic character of the language, present in the structure of all semantic maps (definable or not), rather than as a property to be proved about valuations.

Environments will be built up using a product-like operation as usual. But Principle II tells us right away that a cartesian (categorical) product would not be appropriate in this context. The reason is that we do not want a "diagonal" map that takes an environment and forms a bigger environment in which there are two copies (b, b), denoted by different identifiers, of a single component b from the smaller environment: if b interferes with itself then this would violate II.

We will instead use a restricted form of product which, while not cartesian, satisfies the requirements for symmetric monoidal categories. The intuition about this "tensor" product $A \otimes B$ is that its elements are pairs whose components don't interfere with one another. A term t with, say, two free identifiers of types A and B is interpreted as a map of the form

$$A \otimes B \xrightarrow{\quad [\![t]\!] \quad} C$$

thus achieving our basic aim of making Principle II evident. Procedure types are then obtained via a suitable exponential adjunction:

$$Hom(A \otimes B, C) \;\cong\; Hom(A, B \multimap C).$$

This gives a very satisfactory explanation of the interaction of procedures and environments in the presence of interference constraints.

For Principle III, we describe a notion of passive object in our category, where a type is passive iff none of its "elements" writes to any storage variables. If A and B are passive objects then the key property is that $A \otimes B$ is isomorphic to the (categorical) product $A \times B$. In particular, for such a passive A there is a diagonal map from A to $A \otimes A$, corresponding to the intuition that sharing of read access is permissible. In categorical terminology, for each passive type (and only for passive types) there is a canonical commutative comonoid structure. This semantic treatment of passivity amounts to a categorical interpretation of the modality "!" from linear logic (Lafont, 1988; Seeley, 1989).

This categorical structure suggests a semantic view of the typing rules of interference control that is independent of the interference-specific aspects of the underlying category. This abstraction is especially useful because the details of the semantics of non-interference that we present are, in truth, quite complicated in certain respects. However, we believe that it is important to see how this categorical structure *arises* from an account of non-interference in a specific case. It is the interplay between the concrete model and categorical principles that is important here, with the former giving assurance that the abstraction provided by the latter is indeed faithful to more concrete programming intuitions. Our semantic analysis would be incomplete if it did not include both of these perspectives.

1.4. Outline

The remainder of the paper is organized as follows.

We begin in Section 2 by describing a simple language that incorporates Principles I and II. It should be mentioned that the type system used in this study is actually a stripped-down version of those described by Reynolds. In particular, we do not consider record types or intersection types, which were used by Reynolds (1989) to cope with subtleties in the syntactic treatment of passivity. These omissions are made mainly to simplify the exposition.

The use of functors in explicating certain aspects of state dependence is the topic of Sections 3-5. Section 3 begins with the definition of the category of worlds, and then provides an illustrative example of the category at work. Sections 4 and 5 are devoted to defining and exposing basic properties of support and independence.

In Section 6 we use the independence predicate to describe the tensor $\otimes$ and its corresponding exponential adjunction. An interpretation of the typing rules from Section 2 is given in Section 7. This follows fairly directly from categorical considerations. We do not carry out a detailed study of properties of the interpretation itself, but we do look at some semantic equivalences that illustrate reasoning principles that are sound in the presence of interference constraints, principles that would be unsound otherwise.

Sections 8 and 9 are concerned with passivity. Section 8 is devoted to categorical matters, including the connection to intuitionistic linear logic. Section 9 gives interpretations for typing rules that incorporate Principle III into the language.

2. Interference Constraints

We will use A, B to range over types in our language:

$$A, B ::= \mathbf{exp} \mid \mathbf{comm} \mid \mathbf{var} \mid A \rightarrow B.$$

var, **comm** and **exp** are the types of (storage) variables, commands and expressions, respectively. Commands denote state transformations, while expressions denote functions from states to values. The language is Algol-like in the sense of (Reynolds, 1981): all side-effects are encompassed in the type **comm**, expressions are side-effect-free (but state-dependent), and procedures are called by-name. A procedure can cause side-effects only indirectly, when used within a phrase of type **comm**.

A typing context Γ is a (possibly empty) list $x_1 : A_1, ..., x_n : A_n$, where the x_i are distinct identifiers and the A_i are types. (We assume an infinite, but otherwise unspecified,

$$\frac{}{x : A \vdash x : A}\ \text{Id}$$

$$\frac{\Gamma, y : B, x : A, \Delta \vdash t : A}{\Gamma, x : A, y : B, \Delta \vdash t : A}\ \text{Exchange} \qquad \frac{\Gamma \vdash t : A}{\Gamma, x : A \vdash t : A}\ \text{Weakening}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B}\ \rightarrow I \qquad \frac{\Gamma \vdash p : A \rightarrow B \qquad \Delta \vdash q : A}{\Gamma, \Delta \vdash p(q) : B}\ \rightarrow E$$

$$\frac{\Gamma \vdash p : \mathbf{comm} \quad \Gamma \vdash q : \mathbf{comm}}{\Gamma \vdash p; q : \mathbf{comm}}\ \text{Sequencing}$$

$$\frac{\Gamma \vdash p : \mathbf{comm} \quad \Delta \vdash q : \mathbf{comm}}{\Gamma, \Delta \vdash p \,||\, q : \mathbf{comm}}\ \text{Par}$$

$$\frac{\Gamma \vdash V : \mathbf{var} \quad \Gamma \vdash E : \mathbf{exp}}{\Gamma \vdash V := E : \mathbf{comm}}\ \text{Assignment}$$

Table 1 *Typing Rules*

collection of identifiers.) We write Γ, Δ for the concatenation of Γ and Δ. This assumes that identifiers in Γ and Δ are disjoint, which is the implicit assumption whenever we write Γ, Δ. Typing judgements are of the form $\Gamma \vdash t : A$, where t is a term, A is a type, and Γ is a typing context. Some typing rules are in Table 1. Other rules are in Sections 7 and 9.

The use of different contexts for the procedure and the argument in the elimination rule $\rightarrow E$ plays a central role in the approach. An application $p(q)$ can be typed using this rule only if the free identifiers in p and q are disjoint. This ensures that binding using λ preserves the property that distinct identifiers don't interfere. For example, in

$$(\lambda x\,.\ \cdots x \cdots y \cdots)q$$

if we assume that no identifier free in q interferes with y then, using II, we can conclude by Principle I that q doesn't interfere with y. Thus, the bound identifier x won't interfere with y either, preserving Principle II. (So $A \rightarrow B$ is the type of procedures that never interfere with their arguments.)

Put another way, this constraint on application prevents the typing of

$$(\lambda\, x\,.\ \cdots x \cdots y := 2 \cdots)y$$

because y is free in both the argument and procedure. This is how aliasing between x and y in the body of the procedure is avoided.

As in (Reynolds, 1978, 1989), for illustrative purposes we consider a simple form of parallel composition, whereby $p \,||\, q$ is well-formed only when p and q don't interfere. The intention is that this will ensure that their parallel execution is determinate. This is achieved with the different contexts for p and q in the rule Par. For example, we cannot type something like $x := 1 \parallel x := 2$.

Principle III has not yet been taken into account. It will allow a limited form of sharing between concurrent commands, and between procedures and their arguments. This is deferred to Section 9.

3. Functors and State Dependence

In the possible-world approach to program semantics, types are interpreted as functors from a small category $\mathbf{W}$ of "possible worlds" to a category $\mathbf{D}$ of domains, and terms are given by natural transformations. Instead of a single set or domain, types are *families* of suitably related domains.

This approach was used initially by Reynolds and Oles in the semantics of local-variable declarations. Their idea was that allocation of a local variable in a block such as $\mathbf{new}\, x \,.\, C$ induces a change in the "shape" of the store, because it results in there being an extra storage variable (bound to x) that was not available previously. Thus, the collection of possible states *varies* as storage variables get allocated and de-allocated. This variation was modeled using a category of worlds in which the objects were abstract store shapes and the morphisms were "expansions" that allocated additional variables.

We will use possible-world structure to partition semantic entities according to the storage variables (or "locations") that they read from or write to. This will be accomplished with a suitable choice of $\mathbf{W}$. The worlds will be pairs (R, W) of finite sets of locations, with R representing "readable" locations and W "writable" ones. The underlying intuition is that, for a functor $[\![A]\!] : \mathbf{W} \longrightarrow \mathbf{D}$ corresponding to a type A,

> $[\![A]\!](R, W)$ is a domain of meanings appropriate to A which may read from (only) locations in R, and write to (only) locations in W.

Notation: We write $A \in \mathbf{C}$ to indicate that A is an object of a category $\mathbf{C}$. $\mathbf{D}$ is the category of directed-complete posets (predomains) and continuous functions. Semicolon denotes composition in diagrammatic order in various categories. Our model will be a full subcategory of the functor category $\mathbf{D}^{\mathbf{W}}$.

3.1. The Category of Worlds

We assume a fixed infinite set *Loc* (of locations). The category $\mathbf{W}$ has

Objects: The $\mathbf{W}$-objects are pairs (R, W) of finite subsets of *Loc*.
Morphisms: A $\mathbf{W}$-morphism $f : (R, W) \to (R', W')$ is an injective function from $R \cup W$ to $R' \cup W'$ such that $f(R) \subseteq R'$ and $f(W) \subseteq W'$. (Though we won't be explicit on this point, to be completely precise we should label each morphism with its domain and co-domain in $\mathbf{W}$, to disambiguate cases when a single function can serve as a number of $\mathbf{W}$-morphisms.)

Here, $f(R)$ is the image of f on locations in R, and similarly for $f(W)$. We will also write $f(X)$ for the world $(f(R), f(W))$, when $X = (R, W)$. Composition is just composition of set-theoretic functions. We denote the identity **W**-morphism on world X by id_X.

We extend set-theoretic notion for subset inclusion ($\subseteq$), union ($\cup$), intersection ($\cap$) and the inclusion function $L \hookrightarrow L'$ between sets L and L' (when $L \subseteq L'$) to worlds as follows:

$$\begin{array}{l}
(R,W) \subseteq (R',W') \iff R \subseteq R' \wedge W \subseteq W' \\
(R,W) \cup (R',W') \;=\; (R \cup R', W \cup W') \\
(R,W) \cap (R',W') \;=\; (R \cap R', W \cap W') \\
(R,W) \hookrightarrow (R',W') \;=\; R \cup W \hookrightarrow R' \cup W'.
\end{array}$$

We call $(R, W) \hookrightarrow (R', W')$ an *inclusion* morphism.

W-morphisms will be used to convert semantic entities at one world to another world where possibly additional locations are readable or writable. The requirements $f(R) \subseteq R'$ and $f(W) \subseteq W'$ mean that, in general, we cannot do such a conversion to a world where fewer locations are readable or writable. The injectivity requirement ensures that distinct locations do not get identified when passing from one world to another, a restriction that is crucial for the definition of morphism parts of various functors for program types.

W is similar in some respects to the category of finite sets and injective functions, a category that can be used to treat local-variable declarations (Moggi, 1989; O'Hearn and Tennent, 1992). The part of our development having mainly to do with principles I and II, i.e. up until the end of Section 7, could in fact be carried out using this simpler category. However, the separation of read and write aspects of locations will prove to be important for the treatment of passivity.

3.2. Command Meanings

We now illustrate the use of state-dependence information in **W** by considering a functor $[\![\mathbf{comm}]\!]$ of command meanings. For any world (R, W), $[\![\mathbf{comm}]\!](R, W)$ will consist of state transformations for commands that, intuitively, only read from locations in R and write to locations in W. Accordingly, two conditions on state transformations will be imposed, one concerned with locations that are not writable and the other with locations that are not readable.

Suppose c is a state transformation. If c doesn't write to a location, then the location should have the same value in the output state as in the input state. For read access, if applying c to state s doesn't read location ℓ then the result should be independent of the value of ℓ. This "independence" splits into two cases: (i) the value of ℓ is left unchanged, and this does not depend on the particular value that ℓ has, or (ii) ℓ is set to a specific value that does not depend on the initial value of ℓ. (i) corresponds to the case when c neither reads nor writes ℓ, and (ii) is the case when c writes but does not read ℓ. These points are incorporated into the definition of $[\![\mathbf{comm}]\!]$ below.

If $L \subseteq Loc$ then we define the states for L as a set of functions

$$S(L) = L \rightarrow \mathit{Values}$$

where *Values* is some set of storable data values (e.g. integers). For simplicity, we have assumed that there is only one kind of value that can be stored as the contents of a location. To cope with more than one kind of storable value, we could tag each location with a type, indicating the kind of value it can hold, and then require that states and **W**-morphisms respect these tags.

We now define $[\![\mathbf{comm}]\!]$ on **W**-objects:

$$\begin{array}{rl} [\![\mathbf{comm}]\!](R,W) = \Big\{ & c \in S(R\cup W) \rightsquigarrow S(R \cup W) \mid \text{if } c(s) = s' \text{ then} \\ & \forall \ell \in R \cup W, \\ & \quad \ell \notin W \implies s(\ell) = s'(\ell), \text{ and} \\ & \quad \ell \notin R \implies \Big(\forall v \in \text{Values}.\, c(s \mid \ell \mapsto v) = (s' \mid \ell \mapsto v) \\ & \qquad\qquad\quad \vee\ \forall v \in \text{Values}.\, c(s \mid \ell \mapsto v) = s' \Big) \\ & \Big\}, \text{ ordered by graph inclusion.} \end{array}$$

Here $\rightsquigarrow$ is the partial-function space and $(s \mid \ell \mapsto v)$ is the state like s except that ℓ maps to v.

In the case that $W = R = L$ this definition reduces to $S(L) \rightsquigarrow S(L)$, which is similar to the notion of command meaning that we would use if finite sets and injective functions was to serve as the category of worlds (O'Hearn and Tennent, 1992; Oles, 1985). The extra conditions in the definition correspond to the points (i) and (ii) discussed above, and are due to our separation of read and write aspects of locations.

To make $[\![\mathbf{comm}]\!]$ into a functor, for $f : (R, W) \rightarrow (R', W')$ we must define

$$[\![\mathbf{comm}]\!] f \;:\; [\![\mathbf{comm}]\!](R,W) \longrightarrow [\![\mathbf{comm}]\!](R',W').$$

Note that if $f : (R, W) \rightarrow (R', W')$ and $s \in S(R' \cup W')$ then $(f; s) \in S(R \cup W)$.

For $c \in [\![\mathbf{comm}]\!](R,W)$ and $s \in S(R' \cup W')$, $[\![\mathbf{comm}]\!] f\, c\, s$ is undefined if $c(f; s)$ is undefined (notation $c(f; s)\!\uparrow$). If $c(f; s) = s_1 \in S(R \cup W)$, then we define $[\![\mathbf{comm}]\!] f\, c\, s$ to be $s_2 \in S(R' \cup W')$, where for $\ell \in R' \cup W'$

$$s_2(\ell) \;=\; \begin{cases} s(\ell) & \text{if } \ell \notin f(R \cup W) \\ s_1(\ell') & \text{if } f(\ell') = \ell. \end{cases}$$

That is, up to location-renaming by f, $[\![\mathbf{comm}]\!](f)c$ extends c by the identity on extra locations in world (R', W') that are not in the image of f.

To illustrate this definition, consider the meaning $c0 \in [\![\mathbf{comm}]\!](\{x, y\}, \{x, y\})$ corresponding to the assignment statement $x := 1$. (For notational simplicity, in examples we will use $x, y, \ldots$ for identifiers as well as the locations that they denote.) $c0$ takes an input state $s \in S(\{x, y\})$ and produces as output $s' \in S(\{x, y\})$, where $s'(x) = 1$ and $s'(y) = s(y)$.

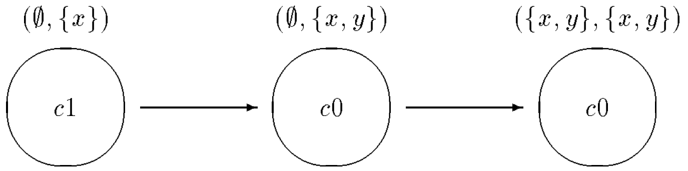
Clearly, $x := 1$ doesn't read from x or y, so we should be able to remove x and y from the read component of the world. Recalling our two cases about read access, we note that (i) c acts like the identity on y, and the final value of x doesn't depend on y, and (ii) it sets x to 1, irrespective of what values are in the initial state. Therefore, from the definition of $[\![\mathbf{comm}]\!]$ it follows that $c0$ is also in $[\![\mathbf{comm}]\!](\emptyset, \{x, y\})$ and

$$c0 \;=\; [\![\mathbf{comm}]\!] \Big((\emptyset, \{x, y\}) \hookrightarrow (\{x, y\}, \{x, y\}) \Big)\; c0.$$

On the other hand, since $x := 1$ doesn't write to y we should be able to remove y from the write component. The result is a meaning $c1 \in [\![\mathbf{comm}]\!](\emptyset, \{x\})$ that maps $s \in S(\{x\})$ to s', where $s'(x)$ is 1. $[\![\mathbf{comm}]\!]$ is defined on morphisms so that, for a morphism $f : X \to Y$ and $c' \in [\![\mathbf{comm}]\!]X$, $[\![\mathbf{comm}]\!]fc'$ is the identity on any locations not in the image of f. Therefore, for our $c0$ and $c1$,

$$c0 \;=\; [\![\mathbf{comm}]\!] \Big((\emptyset, \{x\}) \hookrightarrow (\{x, y\}, \{x, y\}) \Big)\; c1.$$

This example illustrates an important intuition: the meaning $c0$ "lives at" the world $(\{x, y\}, \{x, y\})$, but it "comes from" each of the smaller worlds by using the appropriate component of the morphism part of $[\![\mathbf{comm}]\!]$.



Furthermore, $(\emptyset, \{x\})$ is the "smallest" world that $c0$ comes from in this sense (via an inclusion); we say that $(\emptyset, \{x\})$ is the *support* of $c0$. Notice how well this accords with intuitions about support. The absence of y in the write component of $(\emptyset, \{x\})$ indicates that $c0$ (or the assignment statement $x := 1$) doesn't write to y, and the empty read component indicates that no locations are read from.

4. Support

In this section we define a notion of support that, intuitively, identifies the locations that a semantic entity reads from or writes to. The strategy will be to generalize the treatment of support for command meanings sketched at the end of the previous section. This notion of support will be applicable to all functors in a full subcategory of the functor category $\mathbf{D}^{\mathbf{W}}$.

4.1. Pullbacks and Support

Since types in our model will be functors from $\mathbf{W}$ to $\mathbf{D}$, the support of a semantic entity will be a concept defined *relative to a possible world*, i.e. a $\mathbf{W}$-object. For such a functor A and element $a \in A(X)$, where X is a world, we want to identify the "smallest" world that a "comes from." As a prelude to the definition of support we make this "comes from" intuition precise.

Suppose $A : \mathbf{W} \longrightarrow \mathbf{D}$, $X \in \mathbf{W}$, and $a \in A(X)$. When $Y \subseteq X$, we define

$$Y \models a \iff \exists\, a' \in A(Y) \,.\, A(Y \hookrightarrow X)\, a' \;=\; a.$$

$Y \models a$ is read "a comes from Y." (The notation $Y \models a$ does not indicate the relevant functor (A) or world (X), but no ambiguity is likely to arise as these will always be clear from context. Similar remarks apply to the notations *support*(a), $a\triangle b$ and *passive*(a) to be defined later.)

We are going to define the *support* of $a \in A(X)$ as the smallest $Y \subseteq X$ such that $Y \models a$. Such a smallest world is not guaranteed to exist for arbitrary functors A. However, we can get a satisfactory definition when the following property holds

($*$) for all worlds X, $Y, Z \subseteq Y$, and $a \in A(X)$,

$$Y \models a \;\wedge\; Z \models a \;\Longrightarrow\; Y \cap Z \models a.$$

When ($*$) holds, the intersection of the (finite) collection of Y's such that $Y \models a$ will be the support of a.

We can ensure property ($*$) by making use of a standard connection between intersections and pullbacks. Recall that, in the usual category of sets, if $Y, Z \subseteq X$, then

$$\begin{tikzcd} & X & \\ Y \arrow[ur] & & Z \arrow[ul] \\ & Y \cap Z \arrow[ul] \arrow[ur] & \end{tikzcd}$$

is a pullback square, where the unlabeled arrows are inclusion functions. This is also a pullback square in $\mathbf{W}$, when the unlabeled arrows are what we called the inclusion morphisms and $\cap$ is the componentwise intersection of $\mathbf{W}$-objects.

Note also that the category $\mathbf{W}$ has all pullbacks. Given $\mathbf{W}$-morphisms $j : Y \to X$ and $k : Z \to X$, a pullback square
$$\begin{tikzcd} & X & \\ Y \arrow[ur, "j"] & & Z \arrow[ul, "k"'] \\ & X' \arrow[ul, "h"] \arrow[ur, "i"'] & \end{tikzcd}$$
is obtained by setting $X' = j(Y) \cap k(Z)$ and defining h and i in the evident fashion so that $h(\ell) = j^{-1}(\ell)$ and $i(\ell) = k^{-1}(\ell)$.

Now, property ($*$) can be guaranteed to hold whenever the functor $A : \mathbf{W} \longrightarrow \mathbf{D}$ preserves pullbacks.

Proof of ($*$). If $Y \models a$ and $Z \models a$ then there are $a_1 \in A(Y)$ and $a_2 \in A(Z)$ such that $A(Y \hookrightarrow X)a_1 = a$ and $A(Z \hookrightarrow X)a_2 = a$. Let $\{*\}$ be a one-point domain, and define $f : \{*\} \to A(Y)$ and $g : \{*\} \to A(Z)$ by $f(*) = a_1$ and $g(*) = a_2$. Consider the following diagram, where the unlabeled arrows are $A(i)$ for the appropriate inclusions i.

$$\begin{tikzcd}
 & A(X) & \\
A(Y) \arrow[ur] & & A(Z) \arrow[ul] \\
 & A(Y \cap Z) \arrow[ul] \arrow[ur] & \\
 & \{*\} \arrow[uul, "f"] \arrow[uur, "g"'] \arrow[u, "k"] &
\end{tikzcd}$$

The outer diagram commutes by the definitions of f and g so, since A preserves pullbacks, there is a unique k making the whole diagram commute. In particular, $k(*) \in A(Y \cap Z)$ must be such that $A(Y \cap Z \hookrightarrow Z)\, k(*) = a_2$, and this implies $Y \cap Z \models a$. $\square$

Thus, if $A : \mathbf{W} \longrightarrow \mathbf{D}$ preserves pullbacks, X is a world, and $a \in A(X)$, we define *support*(a) to be the unique world such that

$$\begin{array}{l}
\mathit{support}(a) \models a, \text{ and} \\
\forall Y \subseteq X \,.\, Y \models a \implies \mathit{support}(a) \subseteq Y .
\end{array}$$

That is, *support*(a) is the "smallest" world that a comes from.

Example. Define $c2, c3 \in [\![\mathbf{comm}]\!](\{x\}, \{x\})$ by

$$\begin{array}{l}
c2\, s \;=\; s' \;\text{ where } s'(x) = s(x) + 1, \\
c3\, s \;=\; \begin{cases} \text{undefined} & \text{if } s(x) = 1 \\ s & \text{otherwise.} \end{cases}
\end{array}$$

These could be denotations of commands $x := x + 1$ and **if** $x = 1$ **then diverge**. Recalling also $c0$ from the previous section, we calculate

$$\mathit{support}(c2) = (\{x\}, \{x\}),\;\; \mathit{support}(c0) = (\emptyset, \{x\}), \text{ and }\; \mathit{support}(c3) = (\{x\}, \emptyset).$$

If we let $c4$ denote the everywhere-undefined partial function and $c5$ the identity function in $[\![\mathbf{comm}]\!](\{x\}, \{x\})$ then $\mathit{support}(c4) = \mathit{support}(c5) = (\emptyset, \emptyset)$.

4.2. The Semantic Category

We define the $\mathbf{K}$ to be the category whose objects are pullback-preserving functors from $\mathbf{W}$ to $\mathbf{D}$ and whose morphisms are all natural transformations between such functors, with the usual (vertical) componentwise composition. One can easily verify that $[\![\mathbf{comm}]\!]$ preserves pullbacks.

$\mathbf{K}$ has finite products, which are calculated pointwise as is typical in functor categories. A terminal object $\mathbf{1} \in \mathbf{K}$ can be defined as $\mathbf{1}(X) = \{*\}$ and $\mathbf{1}(f) =$ the identity function on $\{*\}$, for some singleton domain $\{*\}$ and any world X and $f : X \to Y$. If $A, B \in \mathbf{K}$, $X \in \mathbf{W}$ and f is a $\mathbf{W}$-morphism then

$$\begin{aligned}
(A\times B)(X) &= A(X)\times B(X), \text{ ordered componentwise} \\
(A\times B)(f) &= A(f)\times B(f)
\end{aligned}$$

where $\times$ on the right-hand side is the (categorical) product in $\mathbf{D}$. For maps $\eta : A \dot{\to} A'$ and $\mu : B \dot{\to} B'$ in $\mathbf{K}$, the natural transformation $\eta\times\mu : A\times B \dot{\to} A'\times B'$ is defined by setting $\eta\times\mu\, X = \eta(X)\times\mu(X)$. Pullback preservation for $A\times B$ follows straightforwardly from the definition of the $\mathbf{W}$-morphism part of $A\times B$.

A consequence of the pullback-preservation requirement is that the morphism parts of functors in $\mathbf{K}$ are automatically *order-reflecting*, where a map $m : D \to E$ of predomains is order-reflecting iff $\forall d, d' \in D \,.\, m(d) \leq_E m(d') \Rightarrow d \leq_D d'$. This result (which was pointed out by A. Pitts) will play an important role in our development, ensuring that domain-theoretic structure is respected when we define the tensor product, exponential, and passive types in $\mathbf{K}$.

Lemma 1 *Suppose $A \in \mathbf{K}$ and $f : X \to Y$ is a $\mathbf{W}$-morphism. Then the map $A(f)$ is order-reflecting. As a result, it is injective and its image is directed-complete.*

Proof. First, note that for any map $f : X \to Y$ in $\mathbf{W}$ there are maps g and h such that

$$\begin{tikzcd} & \cdot & \\ Y \arrow[ur, "g"] & & Y \arrow[ul, "h"'] \\ & X \arrow[ul, "f"] \arrow[ur, "f"'] & \end{tikzcd}$$

is a pullback square. Second, note that if

$$\begin{tikzcd} & \cdot & \\ E \arrow[ur, "\eta"] & & E \arrow[ul, "o"'] \\ & D \arrow[ul, "m"] \arrow[ur, "m"'] & \end{tikzcd}$$

is a pullback square in $\mathbf{D}$ then m is necessarily order-reflecting; this can be shown by a straightforward calculation using the isomorphism between D and the standard construction of the pullback object as a suitable sub-poset of $E\times E$. That $A(f)$ is order-reflecting then follows because A preserves pullbacks.

Since $A(f)$ is order-reflecting, injectivity follows from monotonicity, and directed-completeness of the image follows from continuity. $\Box$

The next result shows that supports of semantic entities are respected by all natural transformations and by the morphism parts of functors in $\mathbf{K}$. These properties form the technical underpinning for almost all of what follows.

Lemma 2 *Suppose $A, B \in \mathbf{K}$, X is a world, $a \in A(X)$, $f : X \to Z$, and $\eta : A \dot{\to} B$. Then*

$$\begin{array}{ll}
(i) & \mathit{support}(\eta X\, a) \subseteq \mathit{support}(a) \\
(ii) & f(\mathit{support}(a)) = \mathit{support}(A(f)a)
\end{array}$$

Proof. (i). Suppose $Y \models a$. Then $A(Y \hookrightarrow X)a' = a$ for some $a' \in A(Y)$. Naturality of η then guarantees that $B(Y \hookrightarrow X)\Big(\eta Y\, a'\Big) = \eta X\, a$, and the result follows.

(ii). We show $\forall Y \subseteq X \,.\, Y \models a \iff f(Y) \models A(f)a$; the desired result follows from this. Suppose $Y \subseteq X$ and $f : X \to Z$. One can easily verify that the left-hand diagram below commutes in $\mathbf{W}$ – where $f^* : Y \to f(Y)$ is the evident map induced by the restricting f – and the right-hand diagram commutes by the functoriality of A.

$$\begin{tikzcd} Y \arrow[r, "Y \hookrightarrow X"] \arrow[d, "f^*"'] & X \arrow[d, "f"] \\ f(Y) \arrow[r, "f(Y) \hookrightarrow Z"'] & Z \end{tikzcd} \qquad \begin{tikzcd} A(Y) \arrow[r, "A(Y \hookrightarrow X)"] \arrow[d, "A(f^*)"'] & A(X) \arrow[d, "A(f)"] \\ A(f(Y)) \arrow[r, "A(f(Y) \hookrightarrow Z)"'] & A(Z) \end{tikzcd}$$

$\Longrightarrow$: Assume $Y \models a$. This means that $A(Y \hookrightarrow X)a' = a$ for some a'. The right diagram then gives $A(f(Y) \hookrightarrow Z)\Big(A(f^*)a'\Big) = A(f)a$, and so $f(Y) \models A(f)a$.

$\Longleftarrow$: Assume $f(Y) \models A(f)a$. Then the right-hand diagram implies $A(Y \hookrightarrow X; f)a' = A(f)a$ for some a', because $A(f^*)$ is an isomorphism in $\mathbf{D}$ (since f^* is iso in $\mathbf{W}$). $Y \models a$ follows from the injectivity of $A(Y \hookrightarrow X)$ (Lemma 1). □

4.3. Discussion: On the Level of Granularity

With the definition of command meanings as state transformations, the words "a command writes (reads) a location ℓ" must be understood at a level of abstraction where one ignores intermediate states. For example, the "do-nothing" command **skip** and the composite $x := x+1; x := x-1$ would be semantically equal. However, our *mathematical* notion of support does not depend on the particulars of this treatment of commands because it applies to any functor in $\mathbf{K}$, and so this frees us to study properties of support in relative isolation from details of how commands or other types are interpreted. Indeed, it would possible to interpret commands on a level of abstraction where intermediate states are made visible, and the relevant parts of our semantic theory would still apply.

But one important consequence of interference control is that, in the absence of controlled interaction (e.g. through monitors), it is consistent to view concurrent commands on the level of abstraction of state transformations. We believe that it is simpler and more informative to formulate the model in a way that makes this clear.

5. Independence

We can now use the support predicate to define *independence*, a semantic counterpart of non-interference. (We have chosen to use a somewhat neutral term, "independence," to avoid confusion that may arise from possible operational, or implementation oriented, predispositions with regard to the concept of non-interference; c.f. the comments at the end of the last section.)

We begin by defining a relation $\triangle$ of independence between possible worlds. If (R_1, W_1) and (R_2, W_2) are $\mathbf{W}$-objects then

$$(R_1, W_1) \triangle (R_2, W_2) \iff (W_1 \cup R_1) \cap W_2 = \emptyset \ \wedge\ (W_2 \cup R_2) \cap W_1 = \emptyset$$

If two worlds are independent then any writable location in one is not in the other.

Independence between semantic entities is defined in terms of their supports. This is again a notion that is relative to a possible world. If A, B are $\mathbf{K}$-objects, $a \in A(X)$, and $b \in B(X)$, then

$$a \triangle b \iff \mathit{support}(a) \triangle \mathit{support}(b).$$

Example. Suppose $c_1, c_2 \in [\![\mathbf{comm}]\!](R, W)$ and $c_1 \triangle c_2$. We define a state transformation $c_1 \parallel c_2 \in [\![\mathbf{comm}]\!](R, W)$ that represents the joint, parallel, capabilities of c_1 and c_2.

Suppose $\mathit{support}(c_1) = (R_1, W_1)$, $\mathit{support}(c_2) = (R_2, W_2)$, and $s \in S(R \cup W)$. If $c_1(s)\uparrow$ or $c_2(s)\uparrow$ then $(c_1 \parallel c_2)s\uparrow$. Otherwise,

$$(c_1 \parallel c_2)\, s\, \ell \quad = \quad \begin{cases} c_1(s)\,\ell & \text{if } \ell \in W_1 \\ c_2(s)\,\ell & \text{if } \ell \in W_2 \\ s(\ell) & \text{otherwise} \end{cases}$$

Note that W_1 and W_2 are disjoint since $c_1 \triangle c_2$, and so $c_1 \parallel c_2$ is well-defined. From the definitions of $[\![\mathbf{comm}]\!]$ and $\triangle$ one can easily show that $c_1 \parallel c_2 = c_1; c_2 = c_2; c_1$ when $c_1 \triangle c_2$, where semicolon here is composition of partial functions. $\square$

The next lemma states basic properties independence, as it relates to the functor-category structure in $\mathbf{K}$. Part (i) says that independence is preserved and reflected by the morphism parts of functors in $\mathbf{K}$; as a result, changing worlds does not alter independence relationships between semantic entities. (The $\Longrightarrow$ direction is essentially the usual "Kripke monotonicity" property that intuitionistic predicates in presheaf toposes $\mathbf{Sets}^{\mathbf{X}}$ must satisfy.) Part (ii) states that independence is preserved, though not necessarily reflected, by all maps in our category. From the programming perspective, this says that if you apply a closed term of procedural type to two non-interfering entities, then the two resulting terms must still be non-interfering. (To see why the converse should fail, consider a constant procedure that takes an argument and simply returns the numeral 1: 1 doesn't interfere with itself, but this does not imply that arguments to different calls of the constant procedure do not interfere.)

Lemma 3 *Suppose $A, B \in \mathbf{K}$, $a \in A(X)$, $b \in B(X)$, $f : X \to Y$, and $\eta : A \dot{\to} A'$.*

- *(i)* $a \triangle b \iff A(f)a \triangle B(f)b$
- *(ii)* $a \triangle b \Longrightarrow (\eta\, X\, a) \triangle b$
- *(iii)* $\{(a, b) \in A(X) \times B(X) \mid a \triangle b\}$ *is directed-complete.*

Proof. (i) and (ii) follow from Lemma 2. For (iii) suppose $D \subseteq (A \otimes B)X$ is non-empty directed. Since the maps $(a, b) \mapsto \mathit{support}(a)$ and $(a, b) \mapsto \mathit{support}(b)$ from D to the set of worlds have finite image, there is a cofinal subset $D' \subseteq D$ on which they are constant, with values, say, Y and Z. (i.e. $\forall d \in D\, \exists d' \in D' \,.\, d \leq d'$, and $\forall (a, b) \in D' \,.\, \mathit{support}(a) = Y \wedge \mathit{support}(b) = Z$) Clearly $Y \triangle Z$, since $a \triangle b$ whenever $(a, b) \in D'$. Furthermore, if $(a', b') = \bigsqcup D'$ (taking limits in $(A \times B)X$) then $\mathit{support}(a') \subseteq Y$ and

$support(b') \subseteq Z$, because of the componentwise calculation of limits in the product and because the morphism parts of A and B preserve and reflect order (Lemma 1). So this limit is in $(A \otimes B)X$, and the result follows since $\bigsqcup D = \bigsqcup D'$ (as D' is cofinal). □

Finally, we consider how $\triangle$ interacts with finite (categorical) products. This will prove to be important when we define the tensor product in the next section.

Lemma 4 *Suppose $A, B, C \in \mathbf{K}$, $a \in A(X)$, $b \in B(X)$, $c \in C(X)$, and $*$ is the unique element of $\mathbf{1}(X)$. (In (iii) (b, c) is considered as an element of $(B \times C)X$, and similarly for (iv).)*

(i) $* \triangle a$

(ii) $b \triangle a \iff a \triangle b$

(iii) $a \triangle (b, c) \iff a \triangle b \wedge a \triangle c$

(iv) $(a, b) \triangle c \wedge a \triangle b \iff a \triangle (b, c) \wedge b \triangle c$

Proof. (i): Since $support(*) = (\emptyset, \emptyset)$ it follows that $support(*) \triangle support(a)$ no matter what $support(a)$ is.

(ii): Immediate.

(iii) $\Longrightarrow$: If $Z \models (b, c)$ then the definition of the morphism part of $B \times C$ implies $Z \models b$ and $Z \models c$, and so $support(b) \subseteq support(b, c)$ and $support(c) \subseteq support(b, c)$. Thus, if $support(a) \triangle support(b, c)$ then we may conclude that $support(a) \triangle support(b)$ and $support(a) \triangle support(c)$ since, for arbitrary worlds W_1, W_2, W_3, it is clear that

$$W_1 \subseteq W_2 \;\wedge\; W_2 \triangle W_3 \implies W_1 \triangle W_3.$$

(iii) $\Longleftarrow$: Clearly $support(b, c) = support(b) \cup support(c)$, by the definition of $B \times C$ on morphisms. If $a \triangle b$ and $a \triangle c$ then $support(a) \triangle support(b) \cup support(c)$ since, for arbitrary worlds W_1, W_2, W_3,

$$W_1 \triangle W_2 \;\wedge\; W_1 \triangle W_3 \implies W_1 \triangle W_2 \cup W_3.$$

The result follows.

(iv): Immediate from (iii) and (ii). □

6. The Symmetric Monoidal Closed Structure

We now use the independence predicate to define a tensor product $\otimes$ on $\mathbf{K}$. This will be used to interpret contexts on the left-hand side of $\vdash$ in typing judgements. Then we construct the corresponding exponential adjunction $\multimap$ that will model procedure types.

6.1. The Tensor Product

The bifunctor $\otimes$ on $\mathbf{K}$ is a subfunctor of the categorical product $\times$, restricted so that different components are independent of one another.

If A, B are $\mathbf{K}$-objects then

$$\begin{aligned}(A\otimes B)X &= \{(a,b) \in A(X)\times B(X) \mid a\triangle b\}, \text{ ordered componentwise}\\ (A\otimes B)\, f\,(a,b) &= (A(f)\,a, B(f)\,b)\end{aligned}$$

for worlds X and $\mathbf{W}$-morphisms $f : X \to Y$. $(A\otimes B)X$ is directed-complete by Lemma 3(iii), and $(A\otimes B)f$ is well-defined – i.e. $(A(f)a, B(f)b) \in (A\otimes B)Y$ – by the Kripke monotonicity property for $\triangle$ (Lemma 3(i)). Pullback-preservation is immediate from the definition of $A\otimes B$ on $\mathbf{W}$-morphisms.

If $\mu : A \dot{\to} A'$ and $\eta : B \dot{\to} B'$ are maps in $\mathbf{K}$ then the natural transformation $\mu\otimes\eta : A\otimes B \dot{\to} A'\otimes B'$ is such that

$$(\mu\otimes\eta)\, X\,(a,b) = (\mu\, X\, a, \eta\, X\, b)$$

when $(a,b) \in (A\otimes B)X$. This is well-defined by Lemmas 3(ii) and 4(ii). Preservation of identities and composites for $A\otimes B$ and $-\otimes-$ is straightforward.

We can obtain "projections" $pr_i : A_1\otimes A_2 \dot{\to} A_i$, $i = 1,2$, by using the evident inclusion map from $\otimes$ to $\times$. The direct definition is $pr_1\, X\,(a,b) = a$ and $pr_2\, X\,(a,b) = b$.

$\otimes$ is not a categorical product because there is no pairing (or diagonal). However, it does have symmetric monoidal structure. That is, there are symmetry, associativity, and unity isomorphisms that commute in an appropriately coherent fashion. (The "projections" for $\otimes$ can also explained by the fact that the terminal object $\mathbf{1}$ is the unit of this monoidal structure.)

Proposition 5 *There are isomorphisms*

$$\begin{aligned}(A\otimes B)\otimes C &\cong A\otimes(B\otimes C)\\ \mathbf{1}\otimes A \cong A &\cong A\otimes\mathbf{1}\\ A\otimes B &\cong B\otimes A\end{aligned}$$

satisfying the Mac Lane-Kelly equations for symmetric monoidal categories.

Proof. Straightforward using Lemma 4(i) for unity, (ii) for symmetry, and (iv) for associativity. □

6.2. The Exponential

The description of $\multimap$ will follow the standard definition of exponentiation in functor categories, with some alterations to reflect the request that $B\multimap -$ be adjoint to $-\otimes B$, instead of $-\times B$.

First, we recall how (the object part) of the exponentiation $A \Rightarrow B$ in a presheaf category $\mathbf{Sets}^{\mathbf{C}}$ is typically defined (e.g. Lambek and Scott, 1986). For each $X \in \mathbf{C}$, there is a *representable functor* $h^X = Hom_{\mathbf{C}}(X, -)$ from $\mathbf{C}$ to $\mathbf{Sets}$, and the Yoneda lemma tells us that, no matter how exponentiation $\Rightarrow$ is defined, we must have that

$$(A \Rightarrow B)X \;\cong\; Hom_{\mathbf{Sets}^{\mathbf{C}}}(h^X, A \Rightarrow B).$$

Thus, if $A \Rightarrow -$ is to be right adjoint to $- \times A$ then (using Currying) we *must* have that $(A \Rightarrow B)X$ is isomorphic to $Hom_{\mathbf{Sets}^{\mathbf{C}}}(h^X \times A, B)$, and we can simply take this last Hom set to be the *definition* of $(A \to B)X$. Our case will be treated similarly, using $\otimes$ in place of $\times$.

If X is a $\mathbf{W}$-object then the functor $h^X : \mathbf{W} \longrightarrow \mathbf{D}$ is defined by

$$h^X \;=\; Hom_{\mathbf{W}}(X, -)\,;\, F$$

where F is the embedding functor from the category of sets and functions to $\mathbf{D}$ that equips a set with the discrete order. An element of $h^X(Y)$ is a $\mathbf{W}$-morphism $f : X \to Y$. The morphism part of h^X is such that if $f : X \to Y$ and $g : Y \to Z$ then $(h^X\, g\, f) \in h^X(Z)$ is just the composite $f; g$. Pullback preservation is a consequence of the standard fact that representable functors preserve limits (Mac Lane, 1971). So h^X is in fact a $\mathbf{K}$-object.

If A, B are $\mathbf{K}$-objects and X is a world then we define

$$(A \multimap B)X \;=\; Hom_{\mathbf{K}}(h^X \otimes A, B), \text{ ordered pointwise.}$$

Here, by the pointwise order we mean that, for $p_1, p_2 \in (A \multimap B)X$,

$$p_1 \leq p_2 \iff \forall Y \in \mathbf{W} \,.\, \forall (g, a) \in (h^X \otimes A)Y \,.\; p_1\, Y\, (g, a) \leq p_2\, Y\, (g, a)\,.$$

A result of this use of $\otimes$ in place of $\times$ is that procedure meanings can *only* be applied to arguments that they are independent of, as will become evident below when we consider the application map.

Now we define the morphism parts of $A \multimap B$ and $- \multimap -$. If $f : X \to Y$, $(g, a) \in (h^X \otimes A)Z$ and $m \in (A \multimap B)X$ then

$$\Big((A \multimap B)\, f\, m\Big)\, Z\, (g, a) \;=\; m\, Z\, \Big((f; g), a\Big)$$

Notice that $(f; g, a) \in (h^X \otimes A)Z$ because $g \triangle a$. One can show by straightforward calculations that $A \multimap B$ preserves pullbacks. If $\mu : A' \stackrel{.}{\to} A$, $\eta : B \stackrel{.}{\to} B'$ and $p \in (A \multimap B)X$ then $(\mu \multimap \eta)X\, p$ is the bottom of the following diagram.

$$\begin{tikzcd}
h^X \otimes A \arrow[r, "p"] & B \arrow[d, "\eta"] \\
h^X \otimes A' \arrow[u, "id \otimes \mu"] \arrow[r, "(\mu \multimap \eta)Xp"] & B'
\end{tikzcd}$$

The currying map

$$Hom_{\mathbf{K}}(A \otimes B, C) \xrightarrow{\;\mathbf{curry}\;} Hom_{\mathbf{K}}(A, B \multimap C)$$

is given by the equation

$$\Big((\mathbf{curry}\, m)\, X\, a\Big)\, Y\, (f, b) \;=\; m\, Y\Big(A(f)a, b\Big).$$

Note that $A(f)a \triangle b$ by the assumption that $(f, b) \in (h^X \otimes B)Y$, so the argument $(A(f)a, b)$ is of the right type.

These definitions are very similar to the usual ones associated with exponentiation (as adjoint to $\times$) in functor categories. The application map

$$(A \multimap B) \otimes A \xrightarrow{\quad \mathbf{app} \quad} B$$

is more subtle, however, because of the use of $\otimes$ in the definition of $\multimap$. Application for presheaf exponentiation is given using identity morphisms: $\mathbf{app}\, X\, (p, a) \ = \ p\, X\, (id_X, a)$. We cannot use this equation here, because the definition of $\multimap$ would require that $id_X \triangle a$, and this is not always the case.

However, if $p \in (A \multimap B)X$ then, by injectivity of the morphism part of $A \multimap B$ (Lemma 1), there is a unique element $\lceil p \rceil \in (A \multimap B) support(p)$ such that

$$\Big(support(p) \hookrightarrow X \Big) \lceil p \rceil \ = \ p.$$

Furthermore, we clearly have that $(support(p) \hookrightarrow X) \triangle a$ whenever $p \triangle a$, and so the pair $((support(p) \hookrightarrow X), a)$ is in $(h^{support(p)} \otimes A)X$. These observations lead to the following definition of application:

$$\mathbf{app}\, X\, (p, a) \ = \ \lceil p \rceil\, X\, (support(p) \hookrightarrow X, a).$$

With these definitions it is then routine to show that, for $m : C \otimes A \stackrel{\cdot}{\rightarrow} B$, $\mathbf{curry}(m)$ is the unique map making

$$\begin{tikzcd}
C \otimes A \arrow[d, "\mathbf{curry}(m) \otimes id"'] \arrow[dr, "m"] & \\
(A \multimap B) \otimes A \arrow[r, "\mathbf{app}"] & B
\end{tikzcd}$$

commute.

Proposition 6 *For all $B \in \mathbf{K}$, $- \otimes B$ is left adjoint to $B \multimap -$.*

7. Interpretation of Typing Rules

In this section we define a semantics for the language from Section 2. The meaning of a term will be given by a natural transformation between functors in $\mathbf{K}$. More specifically, each derivation of a typing judgement $\Gamma \vdash t : A$ will determine a natural transformation $[\![t]\!]$ from a functor $[\![\Gamma]\!] \in \mathbf{K}$ of environments appropriate to Γ to a functor $[\![A]\!] \in \mathbf{K}$ of meanings appropriate to A.

(To be completely precise we would decorate these meanings $[\![t]\!]$ with data indicating a derivation, and then prove a *coherence* result stating that different derivations of a judgement always lead to the same meaning. See Breazu-Tannen et. al. (1989) for discussion of coherence in this type-theoretic sense.)

7.1. Types and Environments

Now we define suitable functors $[\![A]\!]$ and $[\![\Gamma]\!]$ for types A and typing contexts Γ. The functor $[\![\mathbf{comm}]\!]$ of command meanings has already been specified in Section 3.

$[\![\mathbf{var}]\!]$ is defined on $\mathbf{W}$-objects by

$$[\![\mathbf{var}]\!](R, W) \;=\; R \cap W, \quad \text{discretely ordered.}$$

Variables are locations that are both readable and writable. We have opted for a "simple" semantics here that cannot handle, e.g., state-dependent variables such as conditional variables. On $\mathbf{W}$-morphisms $[\![\mathbf{var}]\!]$ is defined by

$$[\![\mathbf{var}]\!]\, f\, \ell \;=\; f(\ell).$$

The functor $[\![\mathbf{exp}]\!]$ of expression meanings is

$$[\![\mathbf{exp}]\!](R, W) \;=\; S(R) \rightsquigarrow \text{Values}, \quad \text{ordered by graph inclusion}$$
$$[\![\mathbf{exp}]\!]\, f\, e\, s \;=\; e(f^R; s).$$

where $f^R : R \to R'$ is the evident function obtained by restricting the $\mathbf{W}$-morphism $f : (R, W) \to (R', W')$. Procedure types are interpreted as $[\![A \to B]\!] = [\![A]\!] \multimap [\![B]\!]$.

For simplicity, we will regard products of the form $A{\otimes}(B{\otimes}C)$ and $(A{\otimes}B){\otimes}C$ as being identical (in light of Proposition 5), and write $A{\otimes}B{\otimes}C$.

The environment functors are

$$[\![x_1 : A_1, ..., x_n : A_n]\!] \;=\; [\![A]\!]_1 \otimes \cdots \otimes [\![A_n]\!] \qquad\qquad [\![[]]\!] \;=\; \mathbf{1}$$

where $[]$ is the empty typing context. Intuitively, an environment $u \in [\![\Gamma]\!]X$ at world X is a tuple $(u_1, ..., u_n)$ of meanings, the components of which don't interfere with one another.

Example: Suppose that $\ell_1, \ell_2 \in [\![\mathbf{var}]\!](R, W)$ and $\ell_1 \triangle \ell_2$. Since both of these locations are in $R \cap W$, the definition of independence between worlds means that $\ell_1 \neq \ell_2$. Thus, the definition of environments using $\otimes$ ensures that there is no aliasing.

7.2. λ-Calculus Rules

The pure λ-calculus rules from Table 1 are interpreted as follows, where id, exch and proj are appropriate identity, exchange and projection maps (recall that $\otimes$ has "projections").

$$
\begin{array}{rl}
\text{Id} & [\![A]\!] \xrightarrow{\quad\text{id}\quad} [\![A]\!] \\[1em]
\text{Exchange} & [\![\Gamma]\!]\otimes[\![A]\!]\otimes[\![B]\!]\otimes[\![\Delta]\!] \xrightarrow{\text{id}\otimes\text{exch}\otimes\text{id}} [\![\Gamma]\!]\otimes[\![A]\!]\otimes[\![B]\!]\otimes[\![\Delta]\!] \xrightarrow{[\![t]\!]} [\![C]\!] \\[1em]
\text{Weakening} & [\![\Gamma]\!]\otimes[\![A]\!] \xrightarrow{\quad\text{proj}\quad} [\![\Gamma]\!] \xrightarrow{\quad[\![t]\!]\quad} [\![B]\!] \\[1em]
\rightarrow E & [\![\Gamma]\!]\otimes[\![\Delta]\!] \xrightarrow{[\![p]\!]\otimes[\![q]\!]} ([\![A]\!]\multimap[\![B]\!])\otimes[\![A]\!] \xrightarrow{\quad\mathbf{app}\quad} [\![B]\!] \\[1em]
\rightarrow I & [\![\Gamma]\!] \xrightarrow{\quad\mathbf{curry}[\![t]\!]\quad} [\![A]\!]\multimap[\![B]\!]
\end{array}
$$

The reader will see that we have suppressed some trivial applications of unity isomorphisms in the interpretations of these rules.

The placement of $\otimes$ in the interpretation of $\rightarrow E$ is the semantic counterpart of the syntactic requirement that a procedure and its argument don't interfere.

The usual β and η laws of λ-calculus are valid according to this interpretation, because of the adjunction between $-\otimes B$ and $B\multimap -$. The validity of β reflects the call-by-name nature of the language.

Principle II is evident from the definition of environments. As for Principle I, that closed terms don't interfere with any other terms can be explained semantically as follows. A closed term should correspond to a map of the form $m : \mathbf{1} \stackrel{\cdot}{\rightarrow} A$, for some A. Given any world X, $B \in \mathbf{K}$, and $b \in B(X)$, Lemma 4(i) guarantees that $*\triangle b$, and since maps in $\mathbf{K}$ preserve independence (Lemma 3(ii)) it follows that $m(X)*\triangle b$. Thus the meanings of closed terms are independent (in the $\triangle$ sense) of the meanings of other terms. (The principle can be explained similarly for open terms, using Lemma 4(iii) and 4(i) to show that an environment doesn't interfere with a semantic entity if its components don't, and then using the fact that the meaning of a term, as a map in $\mathbf{K}$, must preserve independence.)

7.3. Selected Algol-like Rules

The other rules in Table 1 are interpreted as follows.

$$
\begin{array}{rl}
\text{Par} & [\![\Gamma]\!]\otimes[\![\Delta]\!] \xrightarrow{[\![p]\!]\otimes[\![q]\!]} [\![\mathbf{comm}]\!]\otimes[\![\mathbf{comm}]\!] \xrightarrow{\quad par\quad} [\![\mathbf{comm}]\!] \\[1em]
\text{Sequencing} & [\![\Gamma]\!] \xrightarrow{\quad\langle[\![p]\!],[\![q]\!]\rangle\quad} [\![\mathbf{comm}]\!]\times[\![\mathbf{comm}]\!] \xrightarrow{\quad seq\quad} [\![\mathbf{comm}]\!] \\[1em]
\text{Assignment} & [\![\Gamma]\!] \xrightarrow{\qquad\langle[\![v]\!],[\![e]\!]\rangle\qquad} [\![\mathbf{var}]\!]\times[\![\mathbf{exp}]\!] \xrightarrow{\quad ass\quad} [\![\mathbf{comm}]\!]
\end{array}
$$

Here, $\langle\cdot,\cdot\rangle$ is pairing for the product $\times$ in $\mathbf{K}$, and $par : [\![\mathbf{comm}]\!]\otimes[\![\mathbf{comm}]\!] \stackrel{\cdot}{\rightarrow} [\![\mathbf{comm}]\!]$, $seq : [\![\mathbf{comm}]\!]\times[\![\mathbf{comm}]\!] \stackrel{\cdot}{\rightarrow} [\![\mathbf{comm}]\!]$, and $ass : [\![\mathbf{var}]\!]\times[\![\mathbf{exp}]\!] \stackrel{\cdot}{\rightarrow} [\![\mathbf{comm}]\!]$ are defined as follows, where $c_1 \parallel c_2$ is as in Section 5, $c_1; c_2$ is composition of partial functions, and

$s|_R$ is the restriction of state $s \in S(R \cup W)$ to R:

$$par\, X\,(c_1, c_2) \;=\; c_1 \,\|\, c_2$$

$$seq\, X\,(c_1, c_2) \;=\; c_1; c_2$$

$$ass\,(R, W)\,(\ell, e)\,s \;=\; \begin{cases} (s \mid \ell \mapsto e(s|_R) & \text{if } e(s_R)\downarrow \\ \text{undefined} & \text{if } e(s_R)\uparrow \end{cases}$$

Notice the roles of $\otimes$ in Par and $\times$ in Seq: concurrent commands may not interfere with one another, while sequentially-composed commands may.

A dereferencing coercion that converts a variable to an expression can be given by the map $j : [\![\mathbf{var}]\!] \dot{\rightarrow} [\![\mathbf{exp}]\!]$ such that $j(X)\,\ell\, s \;=\; s(\ell)$.

Two "global" commands are

$$\mathbf{skip} : \mathbf{comm} \qquad\qquad \mathbf{diverge} : \mathbf{comm}$$

They are interpreted by maps $skip$, $diverge : \mathbf{1} \dot{\rightarrow} [\![\mathbf{comm}]\!]$ such that

$$\begin{array}{lcl} skip(R, W) & = & \text{the identity function on } S(R \cup W) \\ diverge(R, W) & = & \text{the everywhere-undefined partial function} \end{array}$$

These are the only maps from $\mathbf{1}$ to $[\![\mathbf{comm}]\!]$.

Now we consider variable declarations. (This is a good test case for our $\triangle$.) To be consistent with Principle II, we will need to ensure that, in a block of the form $\mathbf{new}\, x\,.\,C$, the meaning of the locally-declared identifier x is independent of the meanings of other identifiers. We would certainly expect this to be the case, since the intention is that x denotes a newly allocated variable that is inaccessible by non-local entities.

Matters are simplified if we regard $\mathbf{new}\, x\,.\,C$ as sugar for $\mathbf{new}(\lambda x.C)$, where $\mathbf{new}$ is a combinator of type $(\mathbf{var} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$. For the semantics of $\mathbf{new}$ we define a map $new : ([\![\mathbf{var}]\!] \multimap [\![\mathbf{comm}]\!]) \dot{\rightarrow} [\![\mathbf{comm}]\!]$. By adjointness (Proposition 6) this determines a map from $\mathbf{1}$ to $[\![(\mathbf{var} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}]\!]$. If $p \in ([\![\mathbf{var}]\!] \multimap [\![\mathbf{comm}]\!])(R, W)$ and $s \in S(R \cup W)$, then

$$new\,(R, W)\,p\, s \;=\; \begin{cases} f; s_2 & \text{if } p\,Y\,(f, \ell)\, s_1 \;=\; s_2 \\ \text{undefined} & \text{if } (p\,Y\,(f, \ell)\, s_1)\uparrow \end{cases}$$

where

- $\ell \notin R \cup W$ is any fresh location,
- $f = (R, W) \hookrightarrow Y$, where $(R \cup \{\ell\}, W \cup \{\ell\}) = Y$,
- $s_1 = (s \mid \ell \mapsto 0)$ (0 is the initial value).

The idea here is that the morphism f connects the non-local world to the expanded world with the additional variable. The procedure p is executed in this expanded world with the fresh location ℓ passed as an argument, and this location is de-allocated on termination. The de-allocation is performed using f, obtaining the state $f; s_2 \in S(R, W)$ from the state $s_2 \in S(Y)$ at the expanded world. Notice that $f \triangle \ell$, which is necessary for the argument

(f, ℓ) to be of the right semantic type; one might say that the principle that non-local entities don't interfere with local variables is *forced* on us by the use of $\multimap$ in the semantic type of **new**. We refer to (Oles, 1982,1985; O'Hearn and Tennent, 1992; Tennent, 1991) for further discussion of this form of local-variable semantics. (We mention only that a specific choice of fresh location ℓ need not be given because of the naturality of p: any $\ell \notin R \cup W$ will do.)

Other valuations, e.g. for conditionals and while loops, are as usual.

7.4. Discussion

There are simple equivalences, valid in the model, that illustrate reasoning principles that are sound in the presence of interference constraints. For example,

$$x := 1; y := 2 \quad \equiv \quad y := 2; x := 1$$

when x, y : **var** are different identifiers. Because of the use of $\otimes$ in environments, x and y must denote independent locations, so assigning to one won't affect the other. This equivalence would not hold in a language that allowed aliasing.

Principle II applies to types other than **var**, so it is more than just a statement about aliasing. For example, (assuming the obvious interpretation of **if**) the following equivalence is valid

$$\begin{array}{l} \mathbf{if}\ e = 0\ \mathbf{then} \\ \quad (\,c\,;\ \mathbf{if}\ e = 0\ \mathbf{then\ diverge}\,) \\ \mathbf{else}\quad \mathbf{diverge} \end{array} \qquad \equiv \qquad \mathbf{diverge}$$

for identifiers c : **comm** and e : **exp**. The intuition that is captured here is that execution of c won't change the value of e because c and e are different identifiers.

It is straightforward to prove an adequacy correspondence with a suitable operational semantics (Lent, 1992). However, the model is not fully abstract. Some of the difficult test equivalences for local variables described by Meyer and Sieber (1988) are not valid here (specifically, their Examples 5 and 7).

8. Semantical Passivity

The presentation thus far has not dealt with typing rules that permit *any* sharing between identifiers. In this section and the next we extend our analysis to account for Principle III from the Introduction. This principle allows for a limited amount of sharing, where read, but not write, access is involved. The main semantic concept that must be explained is that of *passivity*, a property of types and phrases that amounts to the absence of write-access capabilities.

This section is concerned with an analysis of basic semantic properties of passivity. Typing rules are considered in Section 9.

8.1. Passive Elements

A program phrase is passive if it doesn't write to any (global) locations. We wish to explain this semantically by saying when an "element" of a semantic domain is passive. As with the concept of independence, this will be relative to a possible world.

If A is a **K**-object and $a \in A(R, W)$ then we define

$$passive(a) \iff (R, \emptyset) \models a.$$

a is passive if comes from a world in which there are no writable locations.

Example. Returning to the command meanings from Sections 3 and 4, examining their support shows

$passive(c3)$, $passive(c4)$, and $passive(c5)$, while
$\neg passive(c2)$ and $\neg passive(c0)$.

The commands **diverge**, **skip**, and **if** $x = 1$ **then diverge** are passive, while $x := x + 1$ and $x := 1$ are not. □

The following result describes basic properties of passivity. Part (i) says essentially that closed terms, given by maps out of **1**, are passive. (ii) relates passivity to products, and (iii) is Principle III. (iv) connects passivity and independence, and in particular implies that passivity is preserved and reflected by morphism parts of **K**-objects, and preserved by **K**-maps (as in Lemma 3).

Lemma 7 *Suppose A, B are **K**-objects, $a \in A(X)$, $b \in B(X)$, $\eta : A \dot{\rightarrow} B$, $f : X \rightarrow Y$, and $*$ is the unique element of $\mathbf{1}(X)$.*

(i) $passive(*)$
(ii) $passive(a) \wedge passive(b) \iff passive(a, b)$
(iii) $passive(a) \wedge passive(b) \implies a \triangle b$
(iv) $passive(a) \iff a \triangle a$

Proof. (i) and (ii) are immediate from Lemma 4. (iii) and (iv) follow from the definitions of $support(\cdot)$ and $\triangle$, and the functoriality of A and B. □

8.2. Passive Objects

The passivity predicate says when an "element" is passive. We call a **K**-object A passive if all of its elements are:

$$A \in \mathbf{K} \text{ is passive} \iff \forall X \in \mathbf{W} \,.\, \forall a \in A(X) \,.\, passive(a).$$

The functor $[\![\mathbf{exp}]\!]$ is easily seen to be passive, because its definition does not mention the write components of worlds at all. $[\![\mathbf{comm}]\!]$ and $[\![\mathbf{var}]\!]$ are not passive.

Passive objects are manufactured by an endofunctor ! on **K**:

$$
\begin{aligned}
!A\,X &= \{a \in A(X) \mid passive(a)\}, \text{ with ordering inherited from } A(X)\\
!A\,f\,a &= A(f)\,a\\
!\eta\,X\,a &= \eta\,X\,a
\end{aligned}
$$

where f is a $\mathbf{W}$-morphism and $\eta : A \dot{\rightarrow} A'$ is a map in $\mathbf{K}$. $!A\,X$ is directed-complete by Lemmas 7(iv) and 3(iii). $!A\,f$ and $!\eta\,X$ are well-defined by Lemmas 7(iv) and 3(i) and (ii). The functoriality of ! and $!A$ are straightforward, and pullback preservation follows directly from the definition of $!A\,f$ and pullback preservation for A.

Proposition 8 *(i)* $A \in \mathbf{K}$ *is passive iff* $!A = A$
(ii) $!^2 = !$
(iii) $!(A{\times}B) = !(A{\otimes}B) = !A{\otimes}!B = !A{\times}!B$
(iv) $!\mathbf{1} = \mathbf{1}$.

Proof. (i) and (ii) are obvious. (iii) follows from Lemma 7(ii) and (iii) and the definitions of $\otimes$ and !. (iv) follows from Lemma 7(i). $\square$.

We now consider the relationship with the ! modality from linear logic. We do this by interpreting the usual logical rules for !.

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}\ \text{Dereliction} \qquad\qquad \frac{!\Gamma \vdash B}{!\Gamma \vdash !B}\ R!$$

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ \text{Contraction}$$

(We don't need to consider the Weakening rule for !, because it is already covered by the general Weakening for $\otimes$.)

Dereliction is given semantically by the map $in_A : !A \dot{\rightarrow} A$ that simply includes the "passive subset" of A into A. Contraction is given by the diagonal map $diag_{!A} : !A \dot{\rightarrow} !A{\otimes}!A$. This exists since $!A{\otimes}!B = !A{\times}!B$, and so we can in fact just use diagonal from $!A$ to $!A{\times}!A$. For $R!$, given a map $m : !A \dot{\rightarrow} B$ we can form the composite

$$!A \xrightarrow{\quad id \quad} !!A \xrightarrow{\quad !m \quad} !B$$

where id is the identity (since $!^2 = !$). In the next section we will use the Dereliction map to interpret application for passive procedures, the diagonal map for Contraction for passive types, and $R!$ for λ-abstraction for passive procedures.

Thus, there are maps of the right functionality for interpreting Dereliction, Contraction, and $R!$. These maps also satisfy the usual categorical axioms for !, amounting on the logical level to equivalences between proofs (e.g. Seely, 1989).

Proposition 9

(i) There are natural transformations $\eta : ! \dot{\rightarrow} I$, $\mu : ! \dot{\rightarrow} !^2$ *making* $(!, \eta, \mu)$ *a comonad, where* I *is the identity functor on* $\mathbf{K}$.
(ii) ! carries the canonical commutative comonoid structure for $\times$ *to a commutative comonoid structure for* $\otimes$.

Proof. (i). The comonad structure is given by defining $\eta_A X : !A X \to A X$ as the evident inclusion map and taking μ as the identity (since $!^2 = !$).

(ii). The canonical comonoid structure (wrt $\times$) on an object A is given by the diagonal $\mathrm{diag}_A : A \dot{\to} A \times A$ and the unique map $\sigma : A \dot{\to} \mathbf{1}$. ! takes the diagonal to $!\mathrm{diag}_A : !A \dot{\to} !(A \times A)$, and this is just the diagonal map $\mathit{diag}_{!A} : !A \dot{\to} !A \times !A$ (note the equality $!(A \times A) = !A \times !A$). Also, since $!\mathbf{1} = \mathbf{1}$, $!m : !A \dot{\to} \mathbf{1}$ is the unique map, and so $(!A, !\mathrm{diag}_A, !m)$ is the canonical commutative comonoid structure (wrt $\times$) for $!A$. Finally, observing that $!A \otimes !A = !A \times !A$ and recalling that $\mathbf{1}$ is the unit of $\otimes$, we get that it is a commutative comonoid wrt $(\otimes, \mathbf{1})$ as well. □

To sum up, the structure on the category **K** that has been found is that of a symmetric monoidal closed category $(\mathbf{1}, \otimes, \multimap)$ with finite products $(\mathbf{1}, \times)$ and a functor ! satisfying the conditions of Proposition 9.

Theorem 10 *Our category is a model of intuitionistic linear logic.*

(Since Weakening is valid, we actually have a model of *affine* logic with "of course" types. There are also additional properties satisfied by our ! that are not valid in all intuistionic linear models, such as the isomorphism $!A \otimes !B = !A \times !B$ and the stronger condition of Lafont (1988) that $!A$ is the *cofree* commutative comonoid over A (the $\Leftarrow$ direction of 7(iv) is important for this).)

This relation to linear logic is interesting. There is in fact a striking similarity in the goals of syntactic control of interference and linear functional programming, as set out in (Lafont, 1988; Holmström, 1988; Wadler, 1990; Abramsky, 1993). These might be considered as two heads of the same coin. One aims to make imperative programming more elegant, by limiting difficulties caused by aliasing and interference, while the other aims to make functional programming more efficient, by permitting destructive updating in a purely functional context and by limiting the need for garbage collection. That they have similar *formal* structure is perhaps more than coincidence. (A preliminary, not entirely satisfactory, syntactic study of this relationship has been attempted in (O'Hearn, 1991).)

9. Passive Types

This section considers syntax rules that take Principle III into account. The most important addition will be a restricted form of the structural rule of Contraction, which was conspicuously absent in Section 2. Contraction is the source of sharing in λ-calculus, so to maintain Principle II we will allow it only for passive types.

It should be mentioned that the presentation in this section departs somewhat from (Reynolds, 1989). One difference is that we have chosen to use explicit structural rules in our formulation, while Reynolds' systems are in a more familiar format where these rules are left implicit. This is for the most part a minor point, though focusing on structural rules perhaps more clearly illustrates the logical flavour of the approach (e.g. the restricted Contraction). A more significant departure is that we do not consider the use of intersection types. We will comment briefly on this at the end of the section.

$$\frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, z : A \vdash t[z/x, z/y] : B}\ \text{Contraction} \quad (A \text{ is passive})$$

$$\frac{\Gamma \vdash p : A \underset{P}{\rightarrow} B \qquad \Delta \vdash q : A}{\Gamma, \Delta \vdash p(q) : B} \underset{P}{\rightarrow} E \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \underset{P}{\rightarrow} B} \underset{P}{\rightarrow} I \quad (\Gamma \text{ is passive})$$

Table 2 *Rules for Passive Types*

9.1. Typing Rules and their Interpretations

The grammar of types is extended to include types for passive procedures

$$A, B \ ::= \ \cdots \ A \underset{P}{\rightarrow} B.$$

The intention is that a procedure of type $A \underset{P}{\rightarrow} B$ must not write to any (global) variables. For example, $\lambda x.x := y$ is of type $\mathbf{var} \underset{P}{\rightarrow} \mathbf{comm}$, when $y : \mathbf{exp}$, because the only free identifier y is in a read-only position. On the other hand, $\lambda y \,.\, x := y$ is not of type $\mathbf{exp} \underset{P}{\rightarrow} \mathbf{comm}$ when $x : \mathbf{var}$, because the procedure has write access to the global variable denoted by x.

$[\![A \underset{P}{\rightarrow} B]\!]$ is defined as $!([\![A]\!] \multimap [\![B]\!])$. We call types of the form $\mathbf{exp}$ and $A \underset{P}{\rightarrow} B$ passive. (Incidentally, if B is a passive type then $[\![A \rightarrow B]\!]$ and $[\![A \underset{P}{\rightarrow} B]\!]$ are isomorphic, so there is a certain amount of redundancy in the types; Reynolds (1989) in fact disallows types of the form $A \rightarrow B$ when B is passive.) A context $x_1 : A_1, ..., x_n : A_n$ is termed passive if each A_i is a passive type. The empty context is considered passive. Some typing rules are in Table 2. In Contraction, $t[z/x, z/y]$ is t with z substituted for x and y.

Lemma 11 *If A is a passive type then $[\![A]\!]$ is a passive $\mathbf{K}$-object. If Γ is a passive typing context then $[\![\Gamma]\!]$ is a passive $\mathbf{K}$-object.*

Proof. $[\![\mathbf{exp}]\!]$ is passive, and $[\![A \underset{P}{\rightarrow} B]\!]$ is passive by Proposition 8(i) and (ii). The result for $[\![\Gamma]\!]$ then follows from Proposition 8(iii) and (iv) □

Using Contraction, passive identifiers can be shared between a procedure and its argument, or between concurrent commands. For example, assuming typical rules for $+$ and 1, we can type $x := z \parallel y := z + 1$:

$$\cfrac{\cfrac{\begin{array}{c}\vdots\\ x : \mathbf{var}, z_1 : \mathbf{exp} \vdash x := z_1 : \mathbf{comm}\end{array} \qquad \begin{array}{c}\vdots\\ y : \mathbf{var}, z_2 : \mathbf{exp} \vdash y := z_2 + 1 : \mathbf{comm}\end{array}}{x : \mathbf{var}, y : \mathbf{var}, z_1 : \mathbf{exp}, z_2 : \mathbf{exp} \vdash x := z_1 \parallel y := z_2 + 1 : \mathbf{comm}}\ \text{Par}}{x : \mathbf{var}, y : \mathbf{var}, z : \mathbf{exp} \vdash x := z \parallel y := z + 1 : \mathbf{comm}}\ \text{Contraction}$$

The restriction of Contraction to passive types is essential. If it were allowed for **var** then we could type $x := 1 \parallel x := 2$, or a procedure call like

$$(\lambda x \,.\, \cdots y \cdots x := 1 \cdots)\, y$$

which would lead to variable aliasing.

Contraction is interpreted by the diagonal map $\mathrm{diag} : [\![A]\!] \dot{\rightarrow} [\![A]\!] \otimes [\![A]\!]$, which exists by Proposition 9 and Lemma 11:

$$\text{Contraction} \qquad [\![\Gamma]\!] \otimes [\![A]\!] \xrightarrow{\mathrm{diag}} [\![\Gamma]\!] \otimes [\![A]\!] \otimes [\![A]\!] \xrightarrow{[\![t]\!]} [\![B]\!]$$

For $\underset{P}{\rightarrow} I$, we obtain

$$[\![\Gamma]\!] \xrightarrow{\mathbf{curry}[\![t]\!]} [\![A]\!] \multimap [\![B]\!]$$

as usual, and then apply ! to get

$$![\![\Gamma]\!] \xrightarrow{!\mathbf{curry}[\![t]\!]} !([\![A]\!] \multimap [\![B]\!])$$

Since Γ is a passive context, $[\![\Gamma]\!]$ is a passive $\mathbf{K}$-object. Thus, $![\![\Gamma]\!] = [\![\Gamma]\!]$ and the map $!\mathbf{curry}[\![t]\!]$ is of the right functionality for the $\underset{P}{\rightarrow} I$ rule.

$\underset{P}{\rightarrow} E$ is given by

$$[\![\Gamma]\!] \otimes [\![\Delta]\!] \xrightarrow{([\![p]\!];\hookrightarrow) \otimes [\![q]\!]} ([\![A]\!] \multimap [\![B]\!]) \otimes [\![A]\!] \xrightarrow{\mathbf{app}} [\![B]\!]$$

where $\hookrightarrow : !([\![A]\!] \multimap [\![B]\!]) \dot{\rightarrow} ([\![A]\!] \multimap [\![B]\!])$ is the evident inclusion.

Finally, we remark that the interpretation of passive procedure types using $!(A \multimap B)$ can be characterized via an adjunction. Let $- \otimes_p B$ be the the restriction of $- \otimes B$ to the subcategory **Pass** of passive objects in $\mathbf{K}$ (B need not be passive here). Then $!(B \multimap -)$, as a functor from $\mathbf{K}$ to **Pass**, is right adjoint to $- \otimes_p B$. (This follows straightforwardly from Proposition 6 and the fact that maps in $\mathbf{K}$ preserve passivity.) Thus, we have an isomorphism of hom sets

$$Hom_{\mathbf{K}}(A \otimes B, C) \cong Hom_{\mathbf{K}}(A, !(B \multimap C))$$

which holds in general only when A is passive. This means also that

$$Hom_{\mathbf{K}}(A, B) \cong Hom_{\mathbf{K}}(\mathbf{1}, !(A \multimap B))$$

since $\mathbf{1}$ is passive. (We will use this last isomorphism implicitly when interpreting block expressions below.)

9.2. Block Expressions

We illustrate passive procedure types with a form of block expression:

$$\mathbf{blkexp} : (\mathbf{var} \underset{P}{\rightarrow} \mathbf{comm}) \underset{P}{\rightarrow} \mathbf{exp}.$$

Intuitively, execution of $\mathbf{blkexp}(t)$ proceeds by first allocating a new (local) location ℓ, then executing $t(\ell)$ in an extended state in which ℓ is initialized to some value, and on termination returning the final value of ℓ as the value of the expression block. The intention is that passivity of t should ensure that there are no changes to non-local variables, and so the use of side-effects in the body of a block expression should be invisible outside its scope. The treatment of block expressions here is inspired by (Tennent, 1991).

As an example block expression, if $n : \mathbf{exp}$ then

$$
\begin{array}{l}
\mathbf{blkexp}\Big(\lambda\, fact\,.\\
\qquad\qquad \mathbf{new}\,(\lambda\, k.\\
\qquad\qquad\qquad fact := 1\ ;\ k := n\ ;\\
\qquad\qquad\qquad \mathbf{while}\ k \neq 0\ \mathbf{do}\\
\qquad\qquad\qquad\quad fact := fact \times k\ ;\\
\qquad\qquad\qquad\quad k := k - 1\\
\qquad\qquad)\ \Big)
\end{array}
$$

calculates the factorial of a non-negative integer n in a side-effect-free fashion.

We give the semantics by defining a map $blkexp : !([\![\mathbf{var}]\!] \multimap [\![\mathbf{comm}]\!]) \dot{\rightarrow} [\![\mathbf{exp}]\!]$. If $t \in !([\![\mathbf{var}]\!] \multimap [\![\mathbf{comm}]\!])(R, W)$ and $s \in S(R)$, then

$$
blkexp\,(R, W)\, t\, s \;=\; \begin{cases} s_2(\ell) & \text{if } \lceil t \rceil\, Y\, (f, \ell)\, s_1 = s_2 \\ \text{undefined} & \text{if } \lceil t \rceil\, Y\, (f, \ell)\, s_1 \uparrow \end{cases}
$$

where

- $\ell \notin R \cup W$ is any fresh location,
- $f = (R, \emptyset) \hookrightarrow Y$, where $Y = (R \cup \{\ell\}, \{\ell\})$,
- $[\![\mathbf{var} \underset{P}{\rightarrow} \mathbf{comm}]\!]((R, \emptyset) \hookrightarrow (R, W))\lceil t \rceil = t$, and
- $s_1 = (s \mid \ell \mapsto 0)$.

$\lceil t \rceil$ exists because t is passive, and is unique by Lemma 1. Since the command meaning $\lceil t \rceil\, Y\, (f, \ell)$ lives at the world $(R \cup \{\ell\}, \{\ell\})$, by the definition of $[\![\mathbf{comm}]\!]$ this means that the values of global variables in $R \cup W$ are not altered. That is, the fresh location ℓ is the only location that can have a different value in s_2 than in s_1. Thus, the passivity of t ensures that the expression block is side-effect-free when viewed from outside the scope of the declaration, where changes to local variables aren't visible.

9.3. Recursion

As stated in (Reynolds, 1978,1989), it is not possible to include a general fixed-point combinator in syntactic control of interference as it presently stands. If $F = \lambda f.t$ assigns to a global variable denoted by a free identifier, then f and this identifier will interfere in a fixed-point definition $\mathbf{Y}F$, violating Principle II. Another way to see the problem is to notice that the right-hand side of the fixed-point equation $\mathbf{Y}F = F(\mathbf{Y}F)$ violates the restriction that a procedure never interfere with its argument. This difficulty is mitigated somewhat by the fact that we can define fixed-points of *passive* procedures. If $\lambda f.t$ is passive then there will be no assignments (to global variables) in the body that could cause interference with f. Similarly, there is no problem with the fixed-point equation.

Jumps cause related problems. If we take the position that a "label" denotes a continuation, then it interferes with any variables that are assigned to "later." This seems difficult to reconcile with the principle that distinct identifiers don't interfere, without relaxing the principle or introducing a naming convention that groups interfering continuations and variables together into a common collection.

These problems are the subject of current research. Here we are going to simply indicate that the relevant fixed-points for passive procedures do exist in our model.

Fixed-points are calculated in the full subcategory $\mathbf{K}'$ of $\mathbf{K}$ whose objects A are such that

- $A(X)$ has a least element, for each $\mathbf{W}$-object X, and
- $A(f)$ is strict, for each $\mathbf{W}$-morphism f.

The strictness requirement applies only to the objects of $\mathbf{K}'$; a component $\eta(X)$ of a natural transformation η in $\mathbf{K}'$ need not be strict. $\mathbf{K}$ is an analogue of the category of "predomains," while $\mathbf{K}'$ is a category of "domains."

Notice that the "simple" $[\![\mathbf{var}]\!]$ that we have opted for does not lie in $\mathbf{K}$', though $[\![\mathbf{comm}]\!]$ and $[\![\mathbf{exp}]\!]$ do. If A is any $\mathbf{K}$-object and B is a $\mathbf{K}'$-object then $A \multimap B$ is in $\mathbf{K}'$. In fact, all of the structure ($\otimes$, $\multimap$, !) cuts down to this smaller category, including the exponential adjunction and the comonoid structure for !.

The strictness requirement has two (related) purposes. First, "global" least elements are needed in B for $(A \multimap B)X$ to have a least element (Oles, 1982). Second, for the fixed-point combinator to be natural the calculation of fixed-points must be preserved by the morphism parts of functors, and strictness is essential for this.

Now we can define the fixed-point map $Y_A :!(A \multimap A) \dot{\rightarrow} A$ for objects A in $\mathbf{K}'$. If $m \in !(A \multimap A)(R, \emptyset)$ then

$$\mathrm{fix}(m) \;=\; \bigsqcup\{(F^i \perp \mid i \text{ is a natural number}\}$$

where $F^0(d) = d$ and $F^{i+1}(d) = m\,(R, \emptyset)\,(id_{(R,\emptyset)}, F^i(d))$, for $d \in A(R, \emptyset)$.

Notice that $id_{(R,\emptyset)} \triangle d$ for such a d since both are passive, and $F^i(d) \in A(R, \emptyset)$ because m is passive. Notice also that $id_{(R,\emptyset)} \triangle \perp$ because $\mathit{support}(\perp) = (\emptyset, \emptyset)$, so $\perp \in [\![A]\!](R, \emptyset)$ and $\{F^i \perp\}$ is in fact have a chain in $[\![A]\!](R, \emptyset)$. We then define

$$Y_A(X)m \;=\; A(\mathit{support}(m) \hookrightarrow X)(\mathrm{fix}\lceil m \rceil).$$

9.4. Discussion

The syntactic treatment of passivity in this section is not entirely satisfactory. As in (Reynolds, 1978), β-reduction does not preserve typings. For example, it is easy to derive

$p : \mathbf{comm} \underset{P}{\rightarrow} \mathbf{exp}, c : \mathbf{comm} \vdash p(c) : \mathbf{exp}$, and
$\vdash \lambda x.\lambda y.x : \mathbf{exp} \underset{P}{\rightarrow} (\mathbf{comm} \underset{P}{\rightarrow} \mathbf{exp})$,

and therefore

$$p : \mathbf{comm} \underset{P}{\rightarrow} \mathbf{exp}, c : \mathbf{comm} \vdash (\lambda x.\lambda y.x)(p(c)) : \mathbf{comm} \underset{P}{\rightarrow} \mathbf{exp}.$$

But we cannot derive the judgement that results from β-reduction

$$p : \mathbf{comm} \underset{P}{\rightarrow} \mathbf{exp}, c : \mathbf{comm} \vdash \lambda y.p(c) : \mathbf{comm} \underset{P}{\rightarrow} \mathbf{exp}$$

because the identifier c is non-passive, and so we cannot use the rule $\underset{P}{\rightarrow} I$ to infer that the λ-abstraction has passive type. Reynolds (1989) has shown how this difficulty can be overcome very neatly using a variant of the intersection type discipline of Coppo and Dezani (1978). An elegant category-theoretic interpretation of intersection types has been discussed in (Reynolds, 1987,1991)

Acknowledgements. It is a pleasure to acknowledge the influence on this work of my thesis advisor Bob Tennent, and to express my thanks for his enthusiastic guidance and encouragement throughout. I am also grateful to Steve Brookes, Frank Oles, Andy Pitts, John Reynolds, Edmund Robinson, and an anonymous referree for helpful discussions and comments; and to Samson Abramsky and Prakash Panangaden for suggesting I explore the relation with linear logic. Diagrams were drawn with John Reynolds' macros. This research was partially supported by the Information Technology Research Center of Ontario.

References

Abramsky, S. (1993) Computational interpretations of linear logic. To appear in *Theoretical Computer Science*.

ANSI (1978) American National Standards Institute, Fortran Standard, ANSI X3.9.

Breazu-Tannen, V., Coquand, T., Gunter, C. A. and Scedrov, A. (1991) Inheritance and explicit coercion. *Information and Computation* 93:172–221.

Brinch Hansen, P. (1973) *Operating Systems Principles.* Prentice Hall.

Coppo M. and Dezani, M. (1978) A new type-assignment for λ-terms. *Archiv. Math. Logik.*, 19, 139–156.

Dahl, O. J. (1972) Hierarchical program structures. In *Structured Programming*, Academic Press, London

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science*, 50, 1–102.

Guzmàn, J. and Hudak, P. (1990) Single-threaded polymorphic lambda calculus. *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, 333–345.

Halpern, J. Y., Meyer, A. R. and Trakhtenbrot, B. A. (1983) The semantics of local storage, or what makes the free list free? *Conference Record of the 11th ACM Symposium on Principles of Programming Languages*, 245-257.

Hoare, C. A. R. (1971) Procedures and parameters: an axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, E. Engeler ed., Lecture Notes in Mathematics 188, Springer Verlag, 102–116.

—— (1974a) Monitors: an operating system concept. *Communications of the ACM*, 17, 549–557.

—— (1974b) Hints on programming language design. *Technical Report CS-74-403*, Stanford University.

Holmström, S. (1988) Linear functional programming. *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, Chalmers University.

Holt et. al. (1987) *The Turing Programming Language. Design and Definition.* Prentice Hall.

INMOS LTD. (1988) *occam 2 Reference Manual.* Prentice Hall.

Lafont, Y. (1988) The linear abstract machine. *Theoretical Computer Science*, 59, 157–180.

Lambek, J. and Scott, P. J. (1986) *Introduction to Higher-Order Categorical Logic.* Cambridge University Press.

Lent, A. F. (1992) *The category of functors from state shapes to bottomless CPOs is adequate for block structure.* Master's thesis, MIT.

Lucassen, J. M. and Gifford, D. K. (1988) Polymorphic effect systems. *Conference Record of the 15th ACM Symposium on Principles of Programming Languages.*

Mac Lane, S. (1971) *Categories for the Working Mathematician*, Springer-Verlag.

Meyer, A.R. and Sieber, K. (1988) Towards fully abstract semantics for local variables. *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, 191-203.

Moggi, E. (1989) An abstract view of programming languages. Course Notes.

O'Hearn, P. W. (1991) Linear logic and interference control (preliminary report). *4'th Conference on Category Theory and Computer Science*, LNCS 530, 74–93.

O'Hearn, P. W. and Tennent, R. D. (1992) Semantics of local variables. In Fourman, Johnstone and Pitts editors, *Applications of Categories in Computer Science*, London Math. Soc. Lecture Notes Series 177. Cambridge Univ. Press.

—— (1993) Semantical analysis of specification logic, part 2. To appear in *Information and Computation.*

Oles, F. J. (1982) *A Category-Theoretic Approach to the Semantics of Programming Languages.* Syracuse University.

—— (1985) Type Algebras, Functor Categories and Block Structure. In M. Nivat and J. C. Reynolds editors, Algebraic Methods in Semantics, Cambridge University Press, 543–573.

Padua, D. A. and Wolfe, M. J. (1986) Advanced compiler optimizations for supercomputers. *Communications of the ACM* 29(12).

Popek et. al. (1977) Notes on the design of EUCLID. *SIGPLAN Notices*, 12(3), 11–18.

Reynolds, J. C. (1978) Syntactic control of interference. *Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, 39-46.

—— (1981) The essence of Algol. In J. W. de Bakker and J. C. van Vliet editors, *Algorithmic Languages*, 345–372, North-Holland, Amsterdam.

—— (1987) Conjunctive types and Algol-like languages (abstract of invited lecture). *Proc. 2nd IEEE Symposium on Logic in Computer Science*, Ithaca.

—— (1989) Syntactic control of interference, part II. *ICALP 89 Proceedings*, LNCS 372, 704–722.

—— (1991) The coherence of languages with intersection types. *International Conference on Theoretical Aspects of Computer Software*, Sendai, Japan

Seeley, R. A. G. (1989) Linear logic, $*$-autonomous categories and cofree coalgebras. *Contemporary Mathematics 92: Categories in Computer Science and Logic.* 371–382.

Swarup, V., Reddy, U. S. and Ireland, E. (1991) Assignments for applicative languages. *Proc. Conference on Functional Programming Languages and Computer Architecture.*

Tennent, R. D. (1983) Semantics of interference control. *Theoretical Computer Science*, 27, 297–310.

—— (1986) Functor-category semantics of programming languages and logics, *Category Theory and Computer Programming*, LNCS 240, 206–224

—— (1990) Semantical analysis of specification logic, *Information and Computation* 85(2), 135–162.

—— (1991) *Semantics of Programming Languages.* Prentice Hall International, London.

Wadler, P. (1990) Linear types can change the world!. In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, North Holland.

—— (1992) The essence of functional programming. *Conference Record of the 19th ACM Symposium on Principles of Programming Languages.*