On Bunched Typing

Peter W. O'Hearn
Department of Computer Science
Queen Mary & Westfield College

July 10, 2000

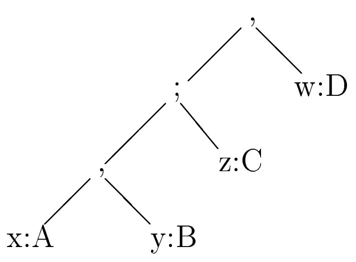
Abstract

We study a typing scheme derived from a semantic situation where a single category possesses several closed structures, or function types. In this scheme typing contexts are trees built from two (or more) binary combining operations, or in short, *bunches*.

The most basic system has two combining connectives, and the structural rules of Weakening and Contraction are allowed for one but not the other. This system includes a multiplicative, or substructural, function type $-\!\!*$ alongside the usual (additive) function type $\rightarrow$; it is dubbed the $\alpha\lambda$-calculus after its binders, α for the αdditive binder and λ for the multiplicative, or λinear, binder. We show that the features of this system are, in a sense, complementary to calculi based on linear logic. In particular, it is incompatible with an interpretation where a multiplicative function uses its argument once, but perfectly compatible with a reading based on sharing of resources. This *sharing interpretation* extrapolates from syntactic control of interference, a type-theoretic method of controlling sharing of storage, and we show how bunch-based management of Contraction can be used to provide a more flexible type system for interference control.

1 Introduction

In most type systems the context Γ in a typing judgement $\Gamma \vdash M : B$ is represented as a function from variables to types, or as a set or sequence of associations $x : A$ pairing identifiers with types. In this paper we study *bunched typing*, where the contexts are trees built from two or more combining operations. For instance, in the basic variant there are two combining operators, "," and ";", which allow us to form contexts Γ, Δ and $\Gamma; \Delta$. And we are able to nest "," and ";" arbitrarily deeply in a bunch, as in



The most intruiging possibility offered by bunched typing is that it gives us a flexible way to merge calculi that treat the structural rules of Exchange, Weakening and Contraction in different ways.

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

$$\frac{\Gamma, x : A, y : A \vdash M : C}{\Gamma, z : A \vdash M[z/x, z/y] : C}\ \text{Contraction.}$$

With bunches we can have several combining operations instead of only the comma, and it is entirely possible to arange matters so that one of the combining forms admits a structural rule when the other does not.

Interest in such substructural type systems has arisen mainly as a result of work on linear logic, which has provided a novel way of understanding the structural rules, particularly Contraction, in terms of *duplication, discarding and consumption* of data [16]. Bunched typing offers a markedly different viewpoint. The language we develop is even incompatible with a number-of-uses reading characteristic of linear typing. We will argue, instead, that it can be understood in terms of *sharing*, rather than duplication. Figuratively speaking, the bunch-based approach to structural rules is about who has access to what, rather than the number of times a piece of data is used.

Many variants on bunched typing are possible. The general case is to have a number of combining operations, each of which admits some subset of the structural rules, with perhaps some interaction between the different forms of combination. The pressing question is why one might consider bunched typing at all, rather than what the general situation is. So we will concentrate for the most part on a basic variant with two combining forms, where ";" admits all of the structural rules and "," admits Exchange only. There is no interaction between the two forms of combination. We obtain a calculus which combines simply-typed λ-calculus and a basic (multiplicative) linear λ-calculus: it is dubbed the $\alpha\lambda$-calculus after its binders, α for the αdditive binder and λ for the multiplicative, or λinear, binder.

Bunched typing may be understood from several theoretical perspectives – proof theoretic, category theoretic, and semantic – and can also be seen to arise from a specific application; sharing of storage in imperative programs. These sources serve to reinforce one another, and in next section we give an informal and leisurely survey of the perspectives offered by them. Readers who prefer a less leisurely approach can skip forward directly to the synopsis in Section 2.5.

Some of the material in this paper was presented, in preliminary form, at the 1999 TLCA conference [33]. Related material on the logic BI of bunched implications, the logical cousin of $\alpha\lambda$, may be found in [34, 38, 20]. See in particular Pym's forthcoming monograph for a treatment of some of the foundational issues avoided here [39].

2 Routes to Bunched Typing

2.1 Sharing and Contraction

The work reported in this paper arose originally from a failed attempt to reconcile two substructural type systems, systems where the structural rule of Contraction is restricted [26, 27].

The background is that in 1978 Reynolds proposed syntactic control of interference, or SCI, a type theoretic method of controlling aliasing and other shared variable interference in imperative programs [45]. In contemporary terminology, what Reynolds used was an affine λ-calculus, where Contraction is absent and where the typing rule for function application requires that a procedure and its argument have disjoint free identifiers:

$$\frac{\Gamma \vdash M : A \multimap\!\!* \, B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B} \,.$$

To understand how SCI works, it is crucial to draw a distinction between the notion of a variable, or identifier, and that of a strorage cell or location that it might denote. The central statement of imperative programming is the assignment $x := e$, which overwrites the contents of a cell denoted by x. For example, a sequence of assignment statements $x := 1; y := 2$ sets (the cell denoted by) x to 1 and y to 2 if they denote different cells.

$$\begin{array}{cc} x & y \\ \hline \multicolumn{1}{|c|}{1} & \multicolumn{1}{c|}{2} \\ \hline \end{array}$$

But if x and y are aliased, which is to say denote the same cell, then the assignment to y in $x := 1; y := 2$ destroys the value placed in the cell previously by assignment to x.

$$\begin{array}{c} x \;\; y \\ \boxed{\;2\;} \end{array}$$

To connect this discussion back to Contraction and function types, note that in

$$((\lambda x \lambda y \,.\, \cdots x := 1; y := 2 \cdots\cdots) z) z$$

if z denotes a cell, then that same cell will be passed to both x and y, resulting in aliasing. To enable this passing of the same cell to both y and z we have to have Contraction, either explicitly or as an admissible rule, in order to get two occurrences of z in an application $(Mz)z$. From this we can see that banishing Contraction abolishes aliasing, a basic example of sharing, at least in this example.

Meanwhile, in 1987 Girard described linear logic (or, LL), a logic that controls Contraction [16]. When its logical rules are used to type λ-terms, linear logic is evidently related to syntactic control. The typing rule for applying a linear function is

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

and there is no Contraction (or Weakening).

Furthermore, in SCI the rule of Contraction is not abolished altogether, but is allowed under restricted circumstances, for certain types that are labelled passive:

$$\frac{\Gamma, x : P, y : P \vdash M : C \quad P \text{ a passive type}}{\Gamma, z : P \vdash M[z/c, z/y] : C} \text{ SCI Contraction}$$

And in linear typing, Contraction is reintroduced under the control of a modality, "!".

$$\frac{\Gamma, x : !A, y : !A \vdash M : C}{\Gamma, z : !A \vdash M[z/c, z/y] : C} \text{ LL Contraction}$$

There is thus a tantalizing *formal* similarity between SCI and linear logic. But there is also a crucial *conceptual* difference: the reading of types, and resulting rationale for controlling contraction, is different in each case.

In SCI, the operative concept is *sharing*. This can be seen most clearly in the reading of the function type.

$A \multimap\!\!* \, B$: functions that don't share storage with their arguments.

Under this reading Contraction is understood as allowing shared access to storage. So limiting Contraction gives control over sharing.

In contrast, in LL the operative concepts are the *number of uses* of a datum, and *consumption*.

$A \multimap B$: functions that use their arguments exactly once.

The intuitive connection with consumption is that if a function uses its argument once, then it may never do so again; so one thinks of a linear function as consuming its argument. Under this reading Contraction is understood as duplicating a piece of data, rather than allowing sharing. So limiting Contraction gives control over the number of times a piece of data can be used.

This conceptual discrepancy between LL and SCI was clear to the author in 1990 [26, 27, 28], but there was a central question left unresolved then: is the difference merely one of having two semantic interpretations of the same system (say, LL), or is a separate formal structure appropriate to each? Stated another way, does the distinction between copying and sharing have type theoretic significance, or is this just a case of having two separate models, where the same type theory is appropriate for both?

A hint of a possible answer to this question is contained in a curious property of models that had been found for SCI, stated as follows in [29].

> "The semantic model presented here posesses two kinds of exponential, one for the monoidal closed structure, and another, adjoint to $\times$ for cartesian closed structure. This raises the question of whether interference control and uncontrolled Algol can coexist harmoniously in one system ... An interesting point to note is that here the two kinds of closed structure coexist in the *same* category, so there is no need to pass to a separate category, such as a Kleisli category, to interpret the intuitionistic (i.e., Algol's) function type."

Given the natural structure that exists in the models, we are lead to ask: what is a typed λ-calculus corresponding to a category that admits two closed structures? This leads us to bunched typing.

2.2 From Doubly-Closed Categories to Bunches

To see how bunches arise categorically, consider that an introduction rule for a function type typically corresponds to an adjunction. That is, a typing rule

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \Rightarrow B}$$

corresponds to an isomorphism of maps of the corresponding shape in a closed category

$$\frac{\Gamma \otimes A \longrightarrow B}{\Gamma \longrightarrow (A \Rightarrow B)} \;.$$

Now, suppose that we have a *doubly closed category*, i.e., a single category equipped with two monoidal closed structures instead of only one:

$$\frac{\Gamma \wedge A \longrightarrow B}{\Gamma \longrightarrow (A \to B)} \qquad \frac{\Gamma * A \longrightarrow B}{\Gamma \longrightarrow (A -\!\!* B)} \;.$$

To match this situation, we extend the syntax of typing contexts with an additional combining operation, semi-colon, which allows us to formulate introduction rules corresponding to the two adjunctions:

$$\frac{\Gamma ; x : A \vdash M : B}{\Gamma \vdash \alpha x \,.\, M : A \to B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A -\!\!* B} \;.$$

This leads directly to the use of tree-like structures, or bunches, for typing contexts.

We will consider sevaral variants. In each, ";" will admit structural rules of Weakening, Contraction, and Exchange and so $\rightarrow$ will behave like the function type in simply-typed λ-calculus or the implication in intuitionistic logic. The variants will arise by disallowing some of the structural rules for ",". In the basic case we consider "," will have Exchange, but not Weakening or Contraction, and this corresponds to the situation where $*$ is the product of a symmetric monoidal structure.

Even at this preliminary stage, the categorical perspective allows us to crisply state the formal difference between bunch-based control over structurals and that obtained from linear logic, or linear typing. In models of linear logic two closed categories are involved, where one is often presented as a Kleisli category [4, 5, 2]. For instance, in the original coherence space model there are indeed two function types, but $\multimap$ is closed structure in the category of linear maps, while the additive $\rightarrow$, which can be represented as $!A \multimap B$, is closed for the category of stable maps. The difference is that in a doubly closed category the two closed structures must reside in one and the same category, and the structural connectives correspond to products in the same category.

Although theoretically clear, this derivation of bunched structure is purely formal and does not, by itself, tell us much about the character of the resulting calculus; the view presented by categorical models is very abstract. More concretely, we have function types $A \mathrel{-\!\!*} B$ and $A \rightarrow B$, and we should ask: for what kinds of functions?

2.3 The Sharing Interpretation

The key to understanding the $\alpha\lambda$-calculus is what we call the *sharing interpretation*. The background idea is of functional programming data such as functions, pairs, etc, but with an additional, intensional, notion of resources that computational entities are allowed to access. By *resource* we mean physical resources in a computer system, such as files, storage, or external devices. The reading of function types is as follows.

> $A \mathrel{-\!\!*} B$: functions that have access to disjoint resources from their arguments.
>
> $A \rightarrow B$: functions that have access to the same resources as their arguments.

Of course, the explanation of $\mathrel{-\!\!*}$ is taken directly from SCI. The interesting point is that this can happily coexist with a direct reading of the additive function type, which does not rely on a decomposition in terms of a modality or other device.

Now, the bare statement of the interpretation is so direct that, at first glance, it may seem as if it must amount to the same thing as resource interpretations for other systems that control the structural rules. For, if we think of a context, roughly, as corresponding to a collection of resources, then the use of separate contexts in the elimination rule for the multiplicative function type $\mathrel{-\!\!*}$ directly expresses the disjointness mentioned in the informal interpretation, and the use of a common context in a rule for the additive corresponds to the sameness.

$$\frac{\Gamma \vdash M : A \mathrel{-\!\!*} B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B} \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M@N : B}$$

However, there is an important point to notice: the reading places no constraint on how many times a $\mathrel{-\!\!*}$-typed function uses its argument, it just cannot use arguments that share access to common resources.

This seems to throw a bit of light on our earlier question, of whether the distinction between copying and sharing has type-theoretic significance. For, although a sharing interpretation is possible, it turns out that a straightforward numbers-of-uses reading (which is intimately connected to copying) is incompatible with bunched typing. We will see in Section 3.2 a derivation of a multiplicative function

$$I \vdash \lambda x \,.\, \alpha f \,.\, (f@x)@x : A \mathrel{-\!\!*} ((A \rightarrow A \rightarrow B) \rightarrow B)$$

that uses its argument x multiple times (here, I is an empty context). This term contradicts the idea that a multiplicative function uses its argument once, but is perfectly compatible with the sharing interpretation: the application of $(f@x)$ to x is allowed because $f@x : A \to B$ is an additive function, and so is allowed to share with its argument.

Variations on the sharing interpretation are possible. For example, in the description of $-\!\!*$ we need not require absolute disjointness, but might allow sharing of certain kinds of resource. In the presence of passivity, in fact, the correct reading for SCI says that any storage shared by a procedure and its argument may only be accessed in a real-only fashion. For another example, in a non-commutative situation there are two multiplicative function types, and the interpretation works by appealing to "dependency" between procedure and argument.

> $A -\!\!\bullet\, B$: functions that cannot depend on the resources of their arguments.
>
> $A \bullet\!\!-\, B$: functions where an argument cannot depend upon the resources of the function.

The sharing interpretation for $-\!\!*$ stated earlier could be recast in terms of a symmetric notion of independence. In contrast, the notion of dependence here is non-symmetric; "dependency" could mean that a pointer can go from the area of the store referenced by one datum to an area of store referenced by another.

2.4 Bunches in Proof Theory

The discussion so far has charted a latter-day route to bunches, emphasizing an interplay between categorical properties and resource interpretations. Historically, however, bunches first arose in the 70s for completely different reasons, as a result of a problem in the proof theory of relevant logics [13]. Since then, bunches have been a standard device used by relevantists' [3, 40, 47].

The most famous property of relevant logics is their denial of Weakening

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \text{Weakening.}$$

This denial is done in a bid to ensure that the premisses in an entailment are actually relevant to the conclusion; for instance, from $A \vdash A$ we cannot infer $A, B \vdash A$, because B might be irrelevant. The technical problem is that, if one simply removes Weakening from standard sequent calculi, say for intuitionistic or classical logic, then some other intuitively reasonable laws are blocked as well. Principle among these is the law of distribution of disjunction over conjunction.

$$A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C).$$

A standard sequent calculus proof uses Weakening in the top steps:

$$\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{A, B \vdash A} \quad \dfrac{B \vdash B}{A, B \vdash B}}{A, B \vdash A \wedge B} \qquad \dfrac{\dfrac{A \vdash A}{A, C \vdash A} \quad \dfrac{C \vdash C}{A, C \vdash C}}{A, C \vdash A \wedge C}}{A, (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C).}$$

And there is no other proof of distribution, if Weakening is simply omitted from sequent calculi.

The most important relevant logics accept distribution for semantic reasons. Simply put, if one reads $\wedge$ as "and" and $\vee$ as "or", then distributiuon *must* follow.

To address the proof-theoretic problem, of how to get distribution while restricting Weakening, novel sequent calculi were formulated by Dunn and Minc (see [13]). In the notation of the present paper, the ";" form of combination admits Weakening, as in

$$\frac{\Delta(\Gamma) \vdash C}{\Delta(\Gamma; \Gamma') \vdash C} \text{ Weakening}$$

where Weakening for ";" can occur anywhere in a bunch. Then, the rules for $\vee$ and $\wedge$ mention ";" but not ",", and the proof just given for distribution goes through simply by replacing "," with ";".

The flexibility of the bunch-based approach to the structural rules comes about from the possibility of using one form of combination to describe rules for one collection of connectives, and the other combination for different connectives. Thus, using bunches the relevantists' we able to cater for the different requirements of *extensional*, or *additive*, connectives such as $\wedge$ and $\vee$, and *intensional*, or *multiplicative*, connectives such as fusion (multiplicative conjunction) and relevant implication. For example, the right rules for two conjunctions are

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma; \Delta \vdash A \wedge B} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B.}{\Gamma, \Delta \vdash A * B}$$

Although these rules are identical in form, the prohibition of Weakening for "," means that they have different ramifications.

The categorical derivation of bunched structure given earlier complements the prior discovery of the relevantists', but also provides a more theoretically cohesive rationale for it. Whereas bunches can be used to deal with technical problems (such as distribution), they are in a sense semantically inevitable from the point of view of doubly closed categories. Also, doubly closed categories emphasize the central role of implication or function types; this point of view is explored in the logic BI of bunched implications of the author and David Pym [34].

2.5 Synopsis

We begin our technical presentation in the next section with a a pared down form of bunched typing, concentrating on a system where the only type constructors are for function types. This function-only fragment is simple, but also displays the most important and unusual consequences of the approach. We make a comparison with linear typing, using a number of examples.

In Section 4 we do some basic work, verifying preservation of typing under substitution and reduction. We also spell out the interpretation of the system in categorical models.

We have a discussed the sharing interpretation above, and we use it to provide a rationale for the examples treated in Section 3.2. But this interpretation is stated informally, and it is important to know that it is consistent with bunched structure and doubly closed categories. We tackle this issue in Section 5 by presenting several models, whose formal description reflects the informal interpretation closely, while at the same time exhibiting doubly closed structure.

The central technique for linking the formal properties of bunched typing to sharing is the *spatial approach to possible world semantics* [43, 28, 29, 30]. In this approach, a world is viewed as corresponding to an area of memory (or, more generally, to resource), and the semantics of types and terms is parameterized by worlds. A semantic expression describing the meaning of any given term will have several occurrence of possible world parameters within it. The spatial intuition captured by this form of semantics is then that when two subexpressions of a semantic expression mention different worlds, the subexpressions refer to separate areas of storage, and consequently don't interfere. (A variation on this statement is needed for a non-commutative model we present.) The models in Section 5 are stripped down versions of spatial possible world models.

After describing further properties of the categorical models in Section 6, we move on in Sections 7-10 to SCI. We show how the affine variant of the calculus (which admits Weakening but not Contraction for ",") can be used to resolve problems with jumps and recursion in the original SCI. The key point in this work is the use of bunches to provide a flexible form of interference control, where sharing constraints can be switched on and off as one moves from more local to more global contexts.

We conclude the paper with a retrospective on resource and substructural typing, and directions for future work.

To proceed with a minimum of distraction we will avoid detailed questions about the relation between syntax and semantics, questions concerning coherence, completeness, and the like. We want to use the semantics mainly as a means to expose surprising or interesting properties of the system. On one level this seems fair enough, as the calculus is very close to the models, being derived from them. But a thorough analysis is crucial; this is provided by Pym separately [39]. In any case, our presentation will be self contained, and we refer to [39] for a more foundational treatment, which is largely complementary to that given in this paper.

3 Bunched Typing

3.1 The Basic System

The basic, symmetric, system is motivated by models as follows.

Definition 1 *A* cartesian doubly closed category*, or cartesian dcc in short, is a category equipped with two symmetric monoidal closed structures* $(I, *, -\!\!*)$ *and* $(1, \wedge, \rightarrow)$*, where* $1, \wedge$ *is cartesian.*

We assume an unspecified collection of primitive types.

TYPES

A	$::=$	ρ	primitive types
	$\mid$	$A -\!\!* A$	multiplicative function type
	$\mid$	$A \rightarrow A$	additive function type

BUNCHES

Γ	$::=$	$x : A$	identifier assumption
	$\mid$	I	multiplicative unit
	$\mid$	Γ, Γ	multiplicative combination
	$\mid$	1	additive unit
	$\mid$	$\Gamma ; \Gamma$	additive combination

Bunches are subject to the restriction that no identifier may occur twice in the tree. This restriction determines implicit side conditions on some of the rules below. We write $\Gamma(\Delta)$ to indicate a bunch in which Δ appears as a subtree, and $\Gamma(\Delta')$ for the similar tree where Δ' replaces Δ. To describe Contraction below, we use $i(\Gamma)$ to denote the list of identifiers encountered one after the other in an inorder traversal of the tree Γ. $\Gamma \cong \Delta$ indicates that Γ and Δ are isomorphic as trees; i.e., one can be obtained from the other by a suitable renaming of identifiers.

We won't try to come up with a more compact representation of bunches using, say, sets or sequences instead of binary operators; the real point of bunches is to let us get the α- and λ-abstractions right. We use an equivalence on trees instead of worrying about representation.

Coherent Equivalence: $\Gamma \equiv \Gamma'$.
$\equiv$ is the smallest equivalence relation on bunches satisfying

1 Commutative monoid equations for 1 and ;

2 Commutative monoid equations for I and ,

3 Congruence: if $\Delta \equiv \Delta'$ then $\Gamma(\Delta) \equiv \Gamma(\Delta')$.

Note that ";" and "," do not distribute over one another.

Typing Judgements
These are of the form

$$\Gamma \vdash M : A$$

where the terms M are defined in the following rules.

Identity and Structure

$$\frac{}{x : A \vdash x : A}\ Id \qquad \frac{\Gamma \vdash M : A}{\Delta \vdash M : A}\ \equiv\ (\text{where } \Delta\ \equiv\ \Gamma)$$

$$\frac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta; \Delta') \vdash M : A}\ W \qquad \frac{\Gamma(\Delta; \Delta') \vdash M : A}{\Gamma(\Delta) \vdash M[i(\Delta)/i(\Delta')] : A}\ C\ \ (\text{where } \Delta \cong \Delta')$$

Functions

$$\frac{\Gamma; x : A \vdash M : B}{\Gamma \vdash \alpha x \,.\, M : A \to B} \to I \qquad \frac{\Gamma \vdash M : A \to B \quad \Delta \vdash N : A}{\Gamma; \Delta \vdash MN : B} \to E$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \mathrel{-\!\!*} B} \mathrel{-\!\!*} I \qquad \frac{\Gamma \vdash M : A \mathrel{-\!\!*} B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash M@N : B} \mathrel{-\!\!*} E$$

The rules for $\mathrel{-\!\!*}$ and $\to$ are identical in form, but they behave differently because of the structural properties of "," and ";". For example, a rule for additive function application that shares contexts

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M@N : B}$$

is derivable using Contraction (see Section 3.4 for further discussion of this). The corresponding rule for $\mathrel{-\!\!*}$ is not derivable.

We have used "deep" formulations of Contraction and Weakening, where the contexts that the rules work on can appear arbitrarily deeply within a tree. These rules are justified by categorical concerns. Contraction corresponds to a doubling map $\delta : A \longrightarrow A \wedge A$, so induces a transformation

$$\frac{f : F(A) \longrightarrow C}{F(\delta); f : F(A \wedge A) \longrightarrow C}$$

if F is a functor. A bunch with a hold $\Gamma(\cdot)$ will determine such a functor, and if we have two isomorphic bunches then we will get a suitable doubling map. Similar considerations apply to Weakening, using the projection $A \wedge B \longrightarrow A$.

As a warming up example notice that, given a judgement $x : A \vdash x : A$, we cannot immediately use an introduction rule to type an identity function of type $A \mathrel{-\!\!*} A$ or $A \to A$, because to apply an introduction rule for a function type we must have a context of the form $\Gamma, x : A$ or $\Gamma; x : A$. So we need to use coherent equivalence first.

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}}{1; x : A \vdash x : A}}{1 \vdash \alpha x \,.\, x : A \to A} \qquad \dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}}{I, x : A \vdash x : A}}{I \vdash \lambda x \,.\, x : A \mathrel{-\!\!*} A}$$

As a second example, we can mimic the isomorphisms

$$[1, A \to B] \;\cong\; [A, B] \;\cong\; [I, A \mathrel{-\!\!*} B]$$

of hom sets in a dcc.

$$\dfrac{\dfrac{x : A \vdash M : B}{1; x : A \vdash M : B}}{1 \vdash \alpha x \,.\, M : A \to B} \qquad \dfrac{\dfrac{x : A \vdash M : B}{I, x : A \vdash M : B}}{I \vdash \lambda x \,.\, M : A \mathrel{-\!\!*} B}$$

$$\dfrac{\dfrac{1 \vdash M : A \to B \quad \dfrac{}{x : A \vdash x : A}}{1; x : A \vdash Mx : B}}{x : A \vdash Mx : B} \qquad \dfrac{\dfrac{I \vdash M : A \mathrel{-\!\!*} B \quad \dfrac{}{x : A \vdash x : A}}{I, x : A \vdash M@x : B}}{x : A \vdash M@x : B}$$

These derivations may make the difference between $\mathrel{-\!\!*}$ and $\to$ appear rather thin, but they only show that *closed* terms in different contexts of one functional type are convertible to the other. In $\alpha\lambda$ the putative judgements $A \mathrel{-\!\!*} B \vdash ? : A \to B$ and $A \to B \vdash ? : A \mathrel{-\!\!*} B$ are not inhabited by any terms. We confirm this claim in Example 16, Section 6.3, where we give a model in which there are no maps between $A \mathrel{-\!\!*} B$ and $A \to B$.

To see that Weakening for "," is not admissible in the calculus, simply note that $x : A \vdash x : A$ is derivable but $x : A, y : B \vdash x : A$ is not. To see that Contraction for "," is not admissible, note that $(f : A \mathrel{-\!\!*} B; x : A), (f' : A \mathrel{-\!\!*} B; x' : A) \vdash fx' : B$ is derivable, while $f : A \mathrel{-\!\!*} B; x : A \vdash fx : B$ is not. We will confirm the non-admissibility of these rules in Remark 10, Section 5.1.

3.2 Unusual Examples, and Comparison to Linear Typing

In $\alpha\lambda$ we can have a multiplicative function that uses its argument many times. For example, in the following, a variable abstracted using λ, the multiplicative abstraction, appears multiple times in the body of the term.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x; f \vdash f@x : A \to B} \qquad \dfrac{\vdots}{x; f \vdash x : A}}{x : A; f : A \to A \to B \vdash (f@x)@x : A \to B}\; C, \to E}{x : A \vdash \alpha f \,.\, (f@x)@x : ((A \to A \to B) \to B)}\; \to I}{I, x : A \vdash \alpha f \,.\, (f@x)@x : ((A \to A \to B) \to B)}\; \equiv}{I \vdash \lambda x \,.\, \alpha f \,.\, (f@x)@x : A \mathrel{-\!\!*} ((A \to A \to B) \to B)}\; \mathrel{-\!\!*} I$$

Here, in the key, top-pictured, step we use the admissible rule for $\to$ elimination.

This term seems wrong if one thinks of a number-of-uses reading. But it is justified by the sharing interpretation. To see why, consider that the subterm $f@x$ is of type $A \to B$. According to the sharing interpretation, it is allowed to share with its argument, in this case x, which is why $(f@x)@x$ is reasonable. (It is important to realize that this term really is "using" x twice. To see why, suppose that A is in fact a function type: if f is a function that accepts two functions, and applies them to different arguments, then $(f@x)@x$ would use x in two different ways.) The sharing interpretation would not support an application $(fx)x$ where f had type $A \mathrel{-\!\!*} A \mathrel{-\!\!*} B$.

Similarly, we can have a multiplicative function that doesn't use its argument at all.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\overline{y : B \vdash y : B}}{x : A; y : B \vdash y : B}\ W}{x : A \vdash (\alpha y.\, y) : B \to B}\ \to I}{I, x : A \vdash (\alpha y.\, y) : B \to B}\ \equiv}{I \vdash \lambda x.\, (\alpha y.\, y) : A \mathrel{-\!\!*} (B \to B)}\ \mathrel{-\!\!*} I
$$

It is instructive to compare with the corresponding types in linear type theory. For the first example, the type would be $A \multimap\, !(!A \multimap\, !A \multimap B) \multimap B$. In trying to derive a term we could λ-abstract on $x : A$ and function parameter f. But then, to apply (the dereliction of) f to x, we would need to convert x to something of type $!A$, and we cannot do a conversion from A to $!A$ in general. Similarly, for the type $A \multimap\, !B \multimap B$ we can abstract on $x : A$ and $y :!B$, but we cannot throw x away.

What is happening here can perhaps be seen more clearly by reference to Barber and Plotkin's DILL system [2], which is a particular formulation of linear typing that admits a pleasantly direct description of $\to$. In Barber and Plotkin's setup ";" is used as a marker, between intuitionistic and linear zones, and judgements are of the form $\Gamma; \Delta \vdash M : A$. Intuitionistic zone Γ and linear zone Δ are simply lists or sets, and there is no nesting of ";". Here are the rules for the function-only fragment.

DILL Typing

$$
\dfrac{}{\Gamma, x : A; \_ \vdash x : A}\ Id - Int \qquad\qquad \dfrac{}{\Gamma; x : A \vdash x : A}\ Id - Lin
$$

$$
\dfrac{\Gamma, x : A; \Delta \vdash M : B}{\Gamma; \Delta \vdash \alpha x.\, M : A \to B}\ \to I \qquad\qquad \dfrac{\Gamma; \Delta \vdash M : A \to B \quad \Gamma; \_ \vdash N : A}{\Gamma; \Delta \vdash M@N : B}\ \to E
$$

$$
\dfrac{\Gamma; \Delta, x : A \vdash M : B}{\Gamma; \Delta \vdash \lambda x.\, M : A \multimap B}\ \multimap I \qquad\qquad \dfrac{\Gamma; \Delta \vdash M : A \multimap B \quad \Gamma; \Delta' \vdash N : A}{\Gamma; \Delta, \Delta' \vdash MN : B}\ \multimap E
$$

For the first of our unusual examples, the key point in DILL is that the linear zone has to be empty in the argument of an additive application $M@N$. The reason, then, that $\lambda x.\, \alpha f.\, (f@x)@x$ is not typable in DILL is that x would have to be in the linear zone, as it is abstracted using λ, so the elimination rule for $\to$ could not be used to type $f@x$. For the second example, in DILL the linear zone must be empty when we introduce an identifier from the intuitionistic zone using $Id - Int$: as a result, $\lambda x\, (\alpha y.\, y)$ is not typable, because x would have to be in the linear zone when y is introduced in the body.

We have given terms for certain judgements in $\alpha\lambda$ whose corresponding judgement forms in linear λ-calculi are not inhabited. Next, we give a judgement form that is inhabited in linear type systems, but not in $\alpha\lambda$.

In linear logic $\multimap$ is convertible to $\to$: we always have $A \multimap B \vdash !A \multimap B$, using dereliction. In DILL we can represent this with the judgement

$$
\_; f : A \multimap B \vdash \alpha x.\, fx : A \to B.
$$

We use an empty intuitionistic context for comparison, as to use the intuitionistic zone would be tantamount to inserting an additional "!".

We remarked above that $\mathrel{-\!\!*}$ is not convertible to $\to$ under bunched typing, but it is useful to see why this is so. If we try to derive the corresponding judgement in

$\alpha\lambda$ we get stuck:

$$\dfrac{\dfrac{???}{f : A \multimap B; x : A \vdash fx : A \to B}}{f : A \multimap B \vdash \alpha x.\, fx : A \to B} \to I$$

The ??? canot be filled in, because under bunched typing f and x would have to be separated using "," in order to use the $\multimap E$ rule to apply the multiplicative function f. That there is no term inhabiting a judgement of this form will be verified in Example 16, Section 6.3.

The discussion in this section shows that, in terms of inhabitation, $\alpha\lambda$ and linear type systems are *incomparable* extensions of multiplicative and simply-typed λ-calcululi. (Stated logically, intuitionistic linear logic with "!" and BI are incomparable extensions of multiplicative, intuitionistic linear logic and intuitionistic logic.) More importantly, each system has a reasonable conceptual justification for the point of view it takes on these judgements where they differ. For linear typing, this is provided by the number-of-uses reading, and also by the linear type structure of domain theory. In $\alpha\lambda$, this rationale is provided by the sharing interpretation.

Further material on the relation to linear type theory may be found in Sections 6.1, 6.4, and 7.5. See also [34, 35] for a logical account.

3.3 Variants and Extensions

Many variants of the basic system are possible; the general case is to have a number of closed structures on a given category. There is no theoretical reason to stop at two, and neither is there a technical reason why one of these structures should be cartesian. But the main reason why bunches are interesting is that they give a particularly simple way to combine a substructural system, which on its own would be rather inexpressive, with a system of full strength additives. We briefly mention two of these variants, one where the multiplicative fragment is affine, and another that has both non-commutative and commutative substructural fragments. We also describe rules for products.

Adding Products. We have considered the function-only fragment. The rules for products, which internalize "," and ";", are as follows.

$$\frac{\Gamma \vdash M : A \quad \Delta \vdash N : B}{\Gamma ; \Delta \vdash \langle M, N \rangle : A \wedge B} \wedge I \qquad \frac{\Gamma \vdash M : A_1 \wedge A_2}{\Gamma \vdash \pi_i\, M : A_i} \wedge E \ \text{(where } i \text{ is 1 or 2)}$$

$$\frac{\Gamma \vdash M : A \quad \Delta \vdash N : B}{\Gamma, \Delta \vdash M * N : A * B} *I \qquad \frac{\Gamma(x : A, y : B) \vdash N : C \quad \Delta \vdash M : A * B}{\Gamma(\Delta) \vdash \mathbf{let}\,(x, y) = M \,\mathbf{in}\, N : C} *E$$

The full system of BI also contains units for these products, and coproducts [34]. See [39] for the corresponding term calculus rules.

The Affine Variant. The affine variant arises semantically by demanding that the units I and 1 be isomorphic. The models are *affine* dcc's, which are cartesian dcc's in which I is terminal.

The affine variant extends the basic calculus as follows.

Affine Coherent Equivalence adds

4 $I \equiv 1$

to Coherent Equivalence.

CONVERTABILITY OF "," TO ";"

$$\frac{\Gamma(\Delta;\Delta') \vdash M : A}{\Gamma(\Delta,\Delta') \vdash M : A}\ Conv$$

Weakening for "," is derivable in the affine variant.

$$\frac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta,\Delta') \vdash M : A}\ W,$$

$A \rightarrow B$ and $A \twoheadrightarrow B$ are not convertible to one another in basic $\alpha\lambda$, but in the affine variant we can go from the former to the latter.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{f : A \rightarrow B \vdash f : A \rightarrow B}}{f : A \rightarrow B, x : A \vdash f : A \rightarrow B}\ W, \quad \dfrac{\overline{x' : A \vdash x' : A}}{f' : A \rightarrow B, x' : A \vdash x' : A}\ W,}{(f : A \rightarrow B, x : A)\,;\,(f' : A \rightarrow B, x' : A) \vdash f x' : B}\ \rightarrow E}{f : A \rightarrow B, x : A \vdash f x : B}\ C}{f : A \rightarrow B \vdash \lambda x.\, f x : A \twoheadrightarrow B}\ \twoheadrightarrow I$$

A Non-commutative Variant. The non-commutative variant we consider combines non-commutative, commutative, and intuitionistic fragments. A model is a single category with: a monoidal biclosed structure; a symmetric monoidal closed structure; a cartesian closed structure. (We decline to formulate an acromym.) The biclosed part means that we have two function types $\bullet\!\!-$ and $-\!\!\bullet$ satisfying the isomorphisms

$$[B, A \bullet\!\!- C] \;\cong\; [A \bullet B, C] \;\cong\; [A, B -\!\!\bullet C].$$

where $\bullet$ is the product of a (not necessarily symmetric) monoidal structure.

Syntactically, the bunches from the basic system are augmented by adding a new unit and combination

$$\begin{array}{rcll} \Gamma & ::= & \vdots & \text{previous clauses} \\ & | & J & \text{non-commutative unit} \\ & | & \Gamma \bullet \Gamma & \text{non-commutative combination} \end{array}$$

where for coherent equivalence we require

4 Monoid equations for J and $\bullet$.

Notice that there is no commutativity. We can then add rules for the left-leaning and right-leaning function types.

$$\frac{x : A \bullet \Gamma \vdash M : B}{\Gamma \vdash \lambda_{\bullet\!-}\, x.\, M : A \bullet\!\!- B}\ \bullet\!\!- I \qquad \frac{\Gamma \vdash M : A \bullet\!\!- B \quad \Delta \vdash N : A}{\Delta \bullet \Gamma \vdash MN : B}\ \bullet\!\!- E$$

$$\frac{\Gamma \bullet x : A \vdash M : B}{\Gamma \vdash \lambda_{-\!\bullet}\, x.\, M : A -\!\!\bullet B}\ -\!\!\bullet I \qquad \frac{\Gamma \vdash M : A -\!\!\bullet B \quad \Delta \vdash N : A}{\Gamma \bullet \Delta \vdash MN : B}\ -\!\!\bullet E$$

(We will not attempt to syntactically disambiguate the various forms of application.)

The way that this system mixes its three fragments is different from Polakow and Pfenning's [37] three-zone, non-commutative variant of DILL, similarly to how DILL and $\alpha\lambda$ are different, as discussed above. It appears that models of their system can be given using three categories, with appropriate mappings between them, just as DILL arises from models based on a pair of categories.

The relationship to the system of Ruet and Fages [46] (also, [41]) is less obvious. Their system uses bunches to combine two multiplicative fragments, the non-commutative and the commutative. However, it does not use bunches to treat the additives, instead relying on a modality as in linear logic. Also, their multiplicatives are classical, in that there is a (dualizing) multiplicative negation. It appears that their system can be modelled using two categories, one a ccc and the other a category possessing simultaneously a monoidal biclosed structure and a (separate) symmetric monoidal closed structure, and a dualizing object (with additional properties).

3.4 Explicit versus Implicit Structural Rules

One might have expected a different formulation of $\alpha\lambda$, where the rules of Weakening and Contraction are removed, and the rules Id and $\to E$ are replaced with

$$\frac{}{\Gamma; x : A \vdash x : A}\ Id\text{, revised} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M @ N : B} \to E\text{, revised.}$$

These rules are derivable in the basic system. The revised rule for identifiers follows from Id, using the coherent equivalence $1; x : A \equiv x : A$. And the revised rule for additive function application follows from $\to E$ and Contraction.

Let us call this the *implicit system*. The simply-typed λ-calculus is usually presented in this implicit fashion, where Weakening is implicit in the rule for identifiers, and Contraction is implicit in the elimination rule for functions. However, this implicit approach does not work corectly in $\alpha\lambda$, owing to interactions between multiplicatives and additives.

To see the problem, consider the judgement

$$(f : A \mathrel{-\!\!*} B, x : A); z : C \vdash fa : B$$

This is derivable in the basic system: first we derive $f : A \mathrel{-\!\!*} B, x : A \vdash fa : B$, and then we use Weakening. But in the implicit system the judgement cannot be derived, because when we apply the elimination rule for $\mathrel{-\!\!*}$ we get a context of the form Γ, Δ. And, $(f : A \mathrel{-\!\!*} B, x : A); z : C$ is not equivalent to any bunch of the form Γ, Δ, where f appears in Γ and x in Δ. Since we can easily derive $f : A \mathrel{-\!\!*} B, x : A \vdash fa : B$ in the implicit system, this shows that it does not admit Weakening for ";".

There is a similar problem with $*$. For, if we add the rules for $*$ to the implicit system, then Contraction is not derivable. We may readily derive

$$(f : Z \mathrel{-\!\!*} A * B, z : Z); (f' : Z \mathrel{-\!\!*} A * B, z' : Z) \vdash \textbf{let}\,(a, b) \;=\; fz\,\textbf{in}\,f'z' : A * B$$

but not

$$f : Z \mathrel{-\!\!*} A * B, z : Z \vdash \textbf{let}\,(a, b) \;=\; fz\,\textbf{in}\,fz : A * B$$

There is the option of building Contraction into the elim rule for $*$, and Weakening into the elim rule for $\mathrel{-\!\!*}$, but these rules seem to be treating a symptom rather than a cause. In the absence of a less unsightly solution, the formulation with explicit structural rules is to be preferred.

4 Basic Properties

Although the main idea of the $\alpha\lambda$-calculus follows from doubly closed categorical structure, the detailed formulation does not. We even saw at the end of the previous section that it is entirely possible to formulate plausible-looking rules that are not quite right. In this section we examine some basic properties of $\alpha\lambda$, concentrating on the basic system from Section 3.1. (We will not be ambitious here; this material is of the initial sanity check variety, and one could go much further.)

We first validate properties relating typing to substitution and reduction. Typing and reduction are areas where substructural type systems, which are surprisingly delicate, have encountered problems in the past [45, 44, 27, 50], so it is appropriate that they be be explored early. We then spell out how the calculus can be interpreted in any cartesian dcc.

4.1 Substitution and Reduction

Before tackling reduction, we need that each of the introduction rules for function types is reversible. This is a property we expect given the isomorphisms

$$[A * B, C] \cong [A, B \mathrel{-\!\!*} C] \qquad\qquad [A \wedge B, C] \cong [A, B \to C]$$

in a dcc.

Lemma 2 (Reversibility) *The inverses of $\mathrel{-\!\!*} I$ and $\to I$ are admissible rules:*

$$\frac{\Gamma \vdash \alpha x.\, M : A \to B}{\Gamma ; x : A \vdash M : B} \qquad\qquad \frac{\Gamma \vdash \lambda x.\, M : A \mathrel{-\!\!*} B}{\Gamma , x : A \vdash M : B}$$

Proof Sketch: In each case a derivation of the premiss must end with an application of the corresponding intro rule, followed by a sequence of applications of the structural rules (W, C, $\equiv$). The proof goes by induction on the length of this last part of the derivation; all cases are straightforward. ■

Recall that "admissibility" means that if you can infer the judgement above the line, then you can also infer the judgement below. This does not mean that there is a generic derivation from one to the other, and admissibility is a property that is not preserved under extensions to a language.

The substitution lemma is formulated for identifiers appearing arbitrarily deeply in a bunch.

Lemma 3 (Substitution Lemma) *The following is an admissible rule.*

$$\frac{\Gamma(x : A) \vdash M : B \quad \Delta \vdash N : A}{\Gamma(\Delta) \vdash M[N/x] : B}$$

Proof Sketch: As usual, a multi-nary version is proven in order to get a strong enough induction hypothesis.

$$\frac{\Gamma(x_1 : A_1 \mid \cdots \mid x_m : A_m) \vdash M : B \quad \Delta_1 \vdash N_1 : A_1, ..., \Delta_m \vdash N_m : A_m}{\Gamma(\Delta_1 \mid \cdots \mid \Delta_m) \vdash M[N_1/x_1, ..., N_m/x_m] : B}$$

where $\Gamma(\Delta_1 \mid \cdots \mid \Delta_m)$ indicates a bunch with multiple distinct sub-bunches.

The proof goes by induction on the derivation of $\Gamma(x_1 : A_1 \mid \cdots \mid x_m : A_m) \vdash M : B$, where the multi-nary aspect is used to deal with the case of Contraction. ■

The two kinds of function in $\alpha\lambda$ come associated with the usual reductions.

β-reductions	η-reductions
$(\alpha x.\, M)@N \;\triangleright\; M[N/x]$	$(\alpha x.\, M@x) \;\triangleright\; M \; (x \notin \mathit{free}(M))$
$(\lambda x.\, M)@N \;\triangleright\; M[N/x]$	$(\lambda x.\, Mx) \;\triangleright\; M \; (x \notin \mathit{free}(M))$

Proposition 4 (Subject Reduction) *If $\Gamma \vdash M : A$ and $M \triangleright N$ then $N : A$.*

Proof: To prove the β case for λ, not that a derivation of $\Gamma \vdash (\lambda x.M)N : A$ must end in a use of $-\!\!*\, E$, followed by a number of applications of structural rules. The proof goes by induction on the length of this sequence.

In the base case we have

$$\frac{\Gamma \vdash (\lambda x.M) : A -\!\!* B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash (\lambda x.M)N : B}$$

as the last rule in the derivation. By the Reversibility Lemma we have $\Gamma, x : A \vdash M : B$ and we can then use the instance

$$\frac{\Gamma, x : A \vdash M : B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash M[N/x] : B}$$

of the substitution lemma.

All other cases $(C, \equiv, W)$ are straightforward, with C using a typical commutativity property of substitution.

The β law for $\rightarrow$ is similar, except that to handle the base case

$$\frac{\Gamma \vdash (\lambda x.M) : A \rightarrow B \quad \Delta \vdash N : A}{\Gamma ; \Delta \vdash (\alpha x.M)N : B}$$

we use the instance

$$\frac{\Gamma ; x : A \vdash M : B \quad \Delta \vdash N : A}{\Gamma ; \Delta \vdash M[N/x] : B}$$

of the Substitution Lemma. Notice how the single, nested, formulation of the the lemma covers both ";" and "," cases.

The proofs for η laws are straightforward, also relying on Reversibility. ■

Finally, a curious point is that the strong normalization theorem for $\alpha\lambda$ does not require any work on our part, because it follows at once from the corresponding result for simply-typed λ-calculus.

Proposition 5 (Strong Normalization) *There are no infinite reduction sequences, starting from any typable term.*

Proof Sketch: We can define a mapping of $\alpha\lambda$ into the simply-typed λ-calculus which sends both $\rightarrow$ and $-\!\!*$ to the function type $\rightarrow$ of λ-calculus. Any reduction in $\alpha\lambda$ then induces a reduction in λ-calculus, which is preserved by the mapping. There can therefore be no infinite reduction sequences in $\alpha\lambda$, or this would contradict the strong normalization theorem of λ-calculus. ■

4.2 Semantic Interpretation

Suppose we are given a cartesian dcc. An interpretation of $\alpha\lambda$ specifies an object $[\![\rho]\!]$ for each primitive type, which then extends to all types using the closed structures. A bunch is interpreted by mapping "," to $*$, ";" to $\wedge$, and similarly for the units. Then, for any proof π of a judgement $\Gamma \vdash M : A$ we can define a map

$$[\![M]\!]_\pi : [\![\Gamma]\!] \longrightarrow [\![A]\!]$$

by induction on π.

To describe the interpretation, if $\Gamma(\cdot)$ is a bunch with a hole, then it determines a functor $[\![\Gamma]\!](\cdot)$ on the dcc in question. When we write $[\![\Gamma]\!]([\![\Delta]\!])$ this indicates the action of the functor on objects, and the occurrences of $[\![\Gamma]\!](\delta)$ and $[\![\Gamma]\!](\pi_1)$ in the rules for Weakening and Contraction use the morphism part.

Semantic Interpretation

$$\frac{}{\mathrm{id} : [\![A]\!] \longrightarrow [\![A]\!]}\ Id \qquad \frac{m : [\![\Delta]\!] \longrightarrow [\![A]\!]}{i; m : [\![\Delta]\!] \longrightarrow [\![A]\!]} \equiv \ (\text{canonical } i : [\![\Delta]\!] \longrightarrow [\![\Gamma]\!])$$

$$\frac{m : [\![\Gamma]\!]([\![\Delta]\!]) \longrightarrow [\![A]\!]}{([\![\Gamma]\!](\pi_1)); m : [\![\Gamma]\!]([\![\Delta]\!] \wedge [\![\Delta']\!]) \longrightarrow [\![A]\!]}\ W$$

$$\frac{(m : [\![\Gamma]\!]([\![\Delta]\!] \wedge [\![\Delta']\!]) \longrightarrow [\![A]\!]}{([\![\Gamma]\!](\delta)); m : [\![\Gamma]\!]([\![\Delta]\!]) \longrightarrow [\![A]\!]}\ C \ \ (\text{where } \Delta \cong \Delta')$$

$$\frac{m : [\![\Gamma]\!] \wedge [\![A]\!] \longrightarrow [\![B]\!]}{m^* : [\![\Gamma]\!] \longrightarrow [\![A]\!] \to [\![B]\!]} \to I \qquad \frac{m : [\![\Gamma]\!] \longrightarrow [\![A]\!] \to [\![B]\!] \quad n : [\![\Gamma]\!] \longrightarrow [\![A]\!]}{m \wedge n; app_{\to} : [\![\Gamma]\!] \longrightarrow [\![B]\!]} \to E$$

$$\frac{m : [\![\Gamma]\!] * [\![A]\!] \longrightarrow [\![B]\!]}{m^\circ : \Gamma \longrightarrow [\![A]\!] \mathrel{-\!\!*} [\![B]\!]} \mathrel{-\!\!*} I \qquad \frac{m : [\![\Gamma]\!] \longrightarrow [\![A]\!] \mathrel{-\!\!*} [\![B]\!] \quad n : [\![\Delta]\!] \longrightarrow [\![A]\!]}{(m * n); app_{\mathrel{-\!\!*}} : [\![\Gamma]\!] * [\![\Delta]\!] \longrightarrow B} \mathrel{-\!\!*} E$$

In the interpretations of Weakening and Contraction, π_1 is the second projection and δ is the doubling map $\langle \pi_1, \pi_2 \rangle : [\![\Delta]\!] \longrightarrow [\![\Delta]\!] \wedge [\![\Delta]\!]$ associated with cartesian structure. Note that $[\![\Delta]\!]$ and $[\![\Delta']\!]$ are actually *equal* when Δ and Δ' are isomorphic; so this semantic clause is type correct.

For the other rules, i is a canonical isomorphism; $(\cdot)^*$ and $(\cdot)^\circ$ are the isomorphisms of hom sets obtained from the adjunctions for $\to$ and $\mathrel{-\!\!*}$; $app_{\to}$ and $app_{\mathrel{-\!\!*}}$ are the application maps obtained from the adjunctions; $\langle m, n \rangle$ is the pairing for the cartesian product; $m * n$ is the action of the $*$ functor on morphisms.

This interpretation can be extended to handle products in the obvious way.

There are two coherence issues worth mentioning. The first concerns the isomorphism $i : [\![\Delta]\!] \longrightarrow [\![\Gamma]\!]$. We intend that this is obtained from a proof that $\Delta \equiv \Gamma$ using commutative monoid laws: any such proof determines an isomorphism, using the coherent isomorphisms of symmetric monoidal categories [24]. We claim that applications of symmetry morphisms are explicitly disambiguated by the use of different identifiers for different types appearing in bunches, so that the morphism i is unique. (This appears to follow from the usual coherence results for symmetric monoidal categories, but we will not carry out a detailed proof.) In any case, this coherence issues can often be sidestepped in specific models by using an interpretation where equivalent bunches are semantically equal; an example will be given in Section 9.2.

The second coherence issue has to do with the order of application of the rules: there can be different proofs π and π' of the same judgement $\Gamma \vdash M : A$, for instance when a structural rule is applied before or after an elimination or intro rule. In this situation we would like to know that the two proofs determine the same map $[\![M]\!]_\pi = [\![M]\!]_{\pi'}$.

A detailed study of the connection between syntax and semantics would involve a careful proof of coherence, together with soundness and completeness results connecting syntactic equality with equality in the models. We avoid this here, and instead refer the reader to Pym's monograph for more information [39].

However, since we have established that any proof of a judgement $\Gamma \vdash M : A$ determines a map from $[\![\Gamma]\!]$ to $[\![A]\!]$ in a cartesian dcc, we can use the following fact with confidence (even prior to coherence or completeness issues).

Lemma 6 (Inhabitation Lemma) *If there exists a cartesian dcc and interpretation of primitive types in which the hom set $[[\![\Gamma]\!], [\![A]\!]]$ is empty, then there can be no M with $\Gamma \vdash M : A$ in $\alpha\lambda$.*

We will make arguments relying on this fact on several occasions, when probing some of the properties of $\alpha\lambda$.

5 Models for the Sharing Interpretation

Thus far the sharing interpretation has been stated informally; in this section we give three models corresponding to it. In the models for the symetric and non-commutative languages, we will give some somewhat lengthy examples, which show in some detail how the interpretations of types work. Our reason for doing this is that we are claiming that the sharing interpretation gives a consistent reading of bunched typing; to understand the sense in which is so, it is essential to do more than merely mention models in passing. At the same time these examples stop short of being "applications".

An affine model is described briefly. A variant of it is developed more fully in Sections 7-10, when we consider SCI.

The models in this section are instances of an abstract construction due to Day, recalled in Section 6.2, which shows how a monoidal structure on $\mathcal{C}$ induces a monoidal biclosed structure on the functor category $\mathbf{Set}^{\mathcal{C}}$. However, we present the models first in an elementary fashion, using direct descriptions of the dcc structure. It is not even necessary to know (or remember) the definitions of cartesian closed structure in functor categories. In two of the three cases the parameterizing category will just be a set and $\mathbf{Set}^{\mathcal{C}}$ thus a product category. In the other case $\mathcal{C}$ is a poset. Our intention in doing this is to use very concrete, and even simple-minded, models, in order to make the connection back to sharing in a straightforward way.

5.1 A Symmetric Model

Let $\mathcal{F}$ denote the collection of finite subsets of a given infinite set *Loc*. We think of an element $X \in \mathcal{F}$ as a possible world that determines a finite collection of resources or, more concretely, locations in computer memory. We will use the product category $\mathbf{Set}^{\mathcal{F}}$ as a model of $\alpha\lambda$. For an object A and element $a \in AX$, we regard a as a computational entity that has access to X.

The crucial operation on worlds is disjoint combination. It is a partial operation, because we only combine those finite sets that are disjoint; intuitively, $X * Y$ indicates separation, where the component worlds X and Y determine distinct resources.

$$\begin{array}{lcl} X * Y &=& X \cup Y, \text{ when } X \cap Y = \{\}; \\ X * Y &=& \textit{undefined}, \text{ when } X \cap Y \neq \{\}. \end{array}$$

This definition makes $(\mathcal{F}, \{\}, *)$ a partial commutative monoid. This means that the commutative monoid laws hold up to an equivalence $e \simeq e'$ on expressions built using $*$ which says that both sides are defined and equal or both undefined.

The cartesian closed structure is determined pointwise.

$$\begin{array}{rcl} (A \to B)X &=& AX \Rightarrow BX \\ (A \wedge B)X &=& AX \times BX \\ 1X &=& \{*\} \end{array}$$

Here, $\Rightarrow$ and $\times$ are function space and product in $\mathbf{Set}$.

Notice how the pointwise definition corresponds closely to the informal reading of $\to$ in the sharing interpretation, where an additive function and its argument have access to the same resources. The additive function type has a strongly local character, where an application of a function stays located at a given world.

To describe the multiplicative function type, we use "multiplicative" form of indexed product. If $A(X, Y)$ is an expression (in the metalanguage) containing parameters X, Y for distinct worlds, then

$$\Pi_{\#} Y.\, A(X, Y)$$

is the product, indexed over worlds disjoint from X. To be precise, a member of the product is a function that accepts a world Y disjoint from X and produces an element of $A(X,Y)$. We often refer to the Y parameter as being "fresh", to briefly indicate its disjointness from X.

The multiplicative function type then quantifies over fresh worlds.

$$(A \twoheadrightarrow B)X \quad = \quad \Pi_{\#}Y.\, A(Y) \Rightarrow B(X * Y)$$

Because of the disjointness requirement on $\Pi_{\#}Y$, the $X * Y$ component in this expression is always defined.

The absence of X in $A(Y)$ mirrors the informal description of multiplicative functions as disjoint from their arguments. An element $p \in (A \twoheadrightarrow B)X$ accepts fresh world Y and element $a \in AY$ as arguments, and produces $p[Y]a \in B(X * Y)$: The "resources" for p are X, while those for a are Y, and these are separate in the result type by virtue of their positions in the combined world $X * Y$. This exemplifies the spatial way of reading the semantic expressions.

We give a concrete example to further illustrate the sharing aspect. Consider the inclusion $L : \mathcal{F} \longrightarrow \mathbf{Set}$. For each finite set X, we think of $LX \;=\; X$ as a set of names, or locations. Let N be the constant object, which is the natural numbers at every component, and define

$$S \;=\; L \to (1 \vee (N \wedge L))$$

where $\vee$ is the pointwise-defined coproduct. Because of the pointwise definition of $\to$ we have that $SX \;=\; X \Rightarrow \{*\} + (N \times X)$. We regard an element $s \in SX$ as a representation of a portion of a computer store, where each $x \in X$ is a pointer to a linked list (possibly with loops).

Now consider any function $f \in ((S \wedge L) \to ((S \wedge L) \twoheadrightarrow S))X$. f accepts $(s, x) \in SX \times X$ and $(s', y) \in SY \times Y$, for fresh finite set Y, as arguments, and produces a state in $S(X * Y)$ as a final result. From the point of view of $S(X * Y)$, there is no overlap between x and y, or between the other pointers in the list pointed to by x and those pointed to by y. Thus, we can view f as a procedure that accepts two linked lists as arguments, with the proviso that the two input lists are defined using disjoint collections of pointers. This kind of proviso is often required in the statement of correctness of an algorithm that, say, removes the elements of one list that appear in the other.

On the other hand, consider the type $L \to (L \twoheadrightarrow (S \to S))$. A function of this type would accept two pointers to linked lists as arguments, and the two pointer arguments would again have to be distinct, but now they could point to lists that overlap in the store.

No particular practical significance is claimed for this example; it is offered just as a concrete illustration of how $\twoheadrightarrow$ and $\to$ can express sharing properties.

In this model the multiplicative unit is I where $I(\{\}) \;=\; \{*\}$, and $I(X) \;=\; \{\}$ for all other X. Before defining $*$ it is useful to observe that a multi-map characterization of maps out of $A * B$ is forced by the definition of $\twoheadrightarrow$. That is, if we are to have the isomorphism $\mathbf{Set}^{\mathcal{F}}[A * B, C] \cong \mathbf{Set}^{\mathcal{F}}[A, B \twoheadrightarrow C]$, then we must obtain the following.

> *Maps $p : A * B \longrightarrow C$ out of a tensor are in bijection with families of functions*
> $$\overline{p}[X][Y] : AX \times BY \longrightarrow C(X * Y)\ ,$$
> *indexed over disjoint finite sets X and Y.*

The idea in terms of sharing is that the components of $*$ are assigned different resources. This is in line with the form of semantics devised by Reynolds for syntactic control of interference [30].

Given this characterization, we can give a simple description of the application map $app_{-\!*} : (A \mathrel{-\!*} B) * A \longrightarrow B$. It is nothing other than

$$app_{-\!*}\,[X][Y]\langle p, a\rangle \;=\; p[Y]a.$$

Also, the exponential transpose,

$$\frac{m : A * B \longrightarrow C}{m^{\circ} : A \longrightarrow (B \mathrel{-\!*} C)}$$

is defined by

$$m^{\circ}[X]a[Y]b \;=\; m[X][Y]\langle a, b\rangle.$$

The actual definition of $*$ is straightforward:

$$(A * B)X \;=\; \{Y, Z, a \in AY, b \in BZ \mid Y * Z = X\}.$$

Here, the condition $Y * Z = X$ requires that $Y * Z$ be defined, so an element of $(A*B)X$ consists of a splitting of the current world, together with an element from each component of the splitting.

The application and exponentiation maps for $\to$ are immediate, given its pointwise definition. All told, we have all of the structure necessary to model $\alpha\lambda$.

Proposition 7 $\mathbf{Set}^{\mathcal{F}}$ *is a cartesian dcc.*

Remark 8 From the point of view of this model the judgement

$$I \vdash \lambda x.\, \alpha f.\, (f@x)@x : A \mathrel{-\!*} ((A \to A \to B) \to B)$$

from Section 3.2 is utterly unsurprising. It determines an element $p \in A \mathrel{-\!*} ((A \to A \to B) \to B)\{\}$ (where we indulge in a confusion between types and objects in $\mathbf{Set}^{\mathcal{F}}$). This function p accepts a fresh world X and $a \in AX$, and produces a function $p[X]a \in ((A \to A \to B) \to B)X$. By the pointwise definition of $\to$, this is a function of type $(AX \Rightarrow AX \Rightarrow BX) \Rightarrow BX$ in $\mathbf{Set}$, and it is the expected function that maps f to $(fa)a$. ■

Remark 9 A broadly similar development can also be carried out in $\mathbf{Set}^{\mathcal{B}}$, where $\mathcal{B}$ is the category of finite sets and bijections. In this category, we model disjointness using the (total) operation $+$ on finite sets which takes disjoint union by tagging its components. For this total operation to work the use of non-identity bijections is crucial: it gives rise to associativity, unity and symmetry isomorphisms that make $(\{\}, +)$ a symmetric monoidal structure on $\mathcal{B}$. We could not use $\mathcal{F}$ with $+$, because $\mathcal{F}$ is a discrete category, and does not have the morphisms needed to make $(\{\}, +)$ monoidal on that category. ■

Remark 10 It is important to see that there is no hidden Weakening or Contraction for "," lurking in the examples of terms that use their arguments two or zero times. In fact, we can see that these rules are absent $\mathbf{Set}^{\mathcal{F}}$ in a very strong sense; there are not even any candidate maps of the required types to model them, let alone maps with the proper properties.

To model Contraction we would need maps of shape $A \longrightarrow A * A$. But there are no maps $L \longrightarrow L * L$, where L is the inclusion from $\mathcal{F}$ to $\mathbf{Set}$. To see why, given $a \in L\{a\}$ we would have to produce an element in $(L * L)\{a\}$, but this set is empty. The reason is that there if $X * Y = \{a\}$ then either X or Y must be $\{\}$ and so, by the definition, a tuple in $(L * L)\{a\}$ would have to identify an element of $L\{\}$. But $L\{\}$ is empty so there can be no such tuple.

To model Weakening, we would need maps $A \longrightarrow I$, for all A. But there are no maps $1 \longrightarrow I$.

These remarks confirm the non-admissibility of Weakening and Contraction for "," referred to in Section 3.1. ■

5.2 An Affine Model

Strictly speaking, the sharing interpretation is stated is for the basic, symmetric version of $\alpha\lambda$. The reading for the *affine* variant, which admits Weakening for the multiplicative combination ",", is obtained by changing the interpretation of the additive function type to say that functions *may* share resources with their arguments.

Let $\mathcal{FS}$ (for $\mathcal{F}$-sub) denote the poset whose elements are the same as $\mathcal{F}$, and whose objects are ordered by subset inclusion. We will use the functor category $\mathbf{Set}^{\mathcal{FS}}$. The $*$ operation from the previous subsection satisfies the following monotonicity property: if $X * Y$ is defined, and $Z \subseteq Y$, then $Z * Y$ is defined and $X * Z \subseteq X * Y$. In this sense, $*$ gives $\mathcal{FS}$ the structure of an *ordered* partial commutative monoid.

The multiplicative function type has the same definition as before, with naturality added:

$$(A \mathrel{-\!\!*} B)X \quad = \quad \Pi_{\#} Y.\, A(Y) \Rightarrow B(X * Y), \text{ natural in } Y.$$

This formula is covariant in X, giving the morphism part.

The definition of $(A{*}B)X$ is more complex. It will be a set of equivalence classes, made up of tuples $\langle Y, Z, a \in AY, b \in BZ \rangle$ where $Y * Z$ is defined and $Y * Z \subseteq X$. Define the relation $\simeq$ by:

$$\begin{array}{l} \langle Y, Z, a \in AY, b \in BZ \rangle \simeq \langle Y', Z', a' \in AY', b' \in BZ' \rangle \\ \text{if } Y \subseteq Y',\ Z \subseteq Z',\ a' = A(Y \subseteq Y')a \text{ and } b' = B(Z \subseteq Z')b \end{array}$$

and consider the least equivalence relation that contains $\simeq$. The elements of $(A{*}B)Z$ are the equivalence classes $[\cdot]$ for this equicalence relation.

$$(A{*}B) = \{[\langle Y, Z, a \in AY, b \in BZ \rangle] \mid\ Y * Z \text{ is defined and } Y * Z \subseteq X\}.$$

(This is just a concrete description of Day's tensor product.)

Combined with the standard fact that $\mathbf{Set}^{\mathcal{FS}}$ is cartesian closed, we obtain

Proposition 11 $\mathbf{Set}^{\mathcal{FS}}$ *is an affine dcc.*

We can try to use the inclusion functor $L : \mathcal{FS} \longrightarrow \mathbf{Set}$ as a variant on the functor used to illustrate the linear sharing interpretation, but it has something of a different character in the affine model. It would not be as reasonable to think of $s \in (L \to (1 \vee (N \wedge L)))X$ as a state, because s would have to accept other worlds Y, and potentially $y \in LY$, as arguments. The reason is that the definition of $\to$ is no longer pointwise, and involves "future" worlds, as in

$$(A \to B)X = \mathbf{Set}^{\mathcal{FS}}[\mathcal{FS}[X, -] \wedge A, B].$$

So the development above, for the symmetric model, does not carry through well to the affine case. However, a thorough account of the sharing aspect of a variant of the affine model is given later, in the context of SCI.

5.3 A Non-commutative Model

In the non-commutative model the commutative multiplicatives $*$ and $\mathrel{-\!\!*}$ will continue to express absence of sharing. To this we add non-commutative operators $\bullet$, $\bullet\!\!-$ and $-\!\!\bullet$ which will be understood in terms of dependency.

Let $\mathcal{W}$ denote the set of binary relations $X \subseteq Loc \times Loc$, for a fixed set Loc. Intuitively, a relation describes a constraint on the shape of the computer store, where $\ell X \ell'$ means that there is allowed to be be a pointer from ℓ to ℓ'.

The non-commutative product of worlds, $X \bullet Y$, will describe a situation where there can be pointers from (the domain of) Y back into X, but not vice versa. To describe this we make two definitions:

$domX \;=\; \{\ell \mid \exists \ell'.\, \ell X \ell'\}$ is the domain of relation X;

$X \gg Y$ holds just if $domX \cap domY = \{\}$ and $\ell X \ell' \;\Rightarrow\; \ell' \notin domY$.

When $X \gg Y$ holds, the union of relations X and Y allows ℓ and ℓ' to be related, where $\ell \in domY$ and $\ell' \in domX$, but not the converse. This leads us to

$X \bullet Y \;=\; X \cup Y$, when $X \gg Y$;

$X \bullet Y \;=\;$ *undefined*, when $\neg(X \gg Y)$.

The definitions of $-\!\!*$ and $*$ in $\mathbf{Set}^{\mathcal{W}}$ are similar to the ones in the symmetric model, using the evident operation on worlds:

$X * Y \;=\; X \cup Y$, when $domX \cap domY \;=\; \{\}$;

$X * Y \;=$ undefined, otherwise.

To describe the non-commutative function types, we first define a non-commutative, multiplicative indexed product in the metalanguage. If $A(X, Y)$ is an expression containing parameters X, Y for worlds, where $X \gg Y$, then

$\Pi_{\gg} Y.\, A(X, Y)$

is the product, indexed over worlds $\gg$-related to Y. That is, an element is a function that accepts a world Y where $X \gg Y$ holds, and produces an element of $A(X, Y)$. Similarly, an element of

$\Pi_{\ll} X.\, A(X, Y)$

is a function that accepts a world X where $X \gg Y$ and produces an element of $A(X, Y)$.

Then the two function types are:

$$
\begin{array}{rcl}
(A \bullet\!\!- B)X & = & \Pi_{\ll} Y.\, A(Y) \Rightarrow B(Y \bullet X) \\
(A -\!\!\bullet B)X & = & \Pi_{\gg} Y.\, A(Y) \Rightarrow B(X \bullet Y).
\end{array}
$$

For $\bullet$,

$(A \bullet B)X \;=\; \{Y, Z, a \in AY, b \in BZ \mid Y \bullet Z = X\}$.

As in the symmetric model there multi-map characterization of maps out of $A \bullet B$, except that the characterization for $\bullet$ works with pairs of world subject to the constraint that $X \gg Y$; the application and transpose maps are also straightforward. So we state:

Proposition 12 $\mathbf{Set}^{\mathcal{W}}$ *is a model of the non-commutative variant: It is*

1. *monoidal biclosed* $(\bullet, \bullet\!\!-, -\!\!\bullet)$,
2. *symmetric monoidal closed* $(*, -\!\!*)$, *and*
3. *cartesian closed* $(\wedge, \rightarrow)$.

(The cartesian closed structure is inherited pointwise from **Set***.)*

The connection between the sharing interpretation and the definitions of $-\!\!*$ and $\rightarrow$ established in the discussion of the symmetric model go through just as well for the model of this section; we focus on the directional independence expressible using the non-commutative operators. (We stress that it is important that this reading does not invalidate that for the other connectives, especially the reading for the additive $\rightarrow$.)

First, we define states similarly as in the symmetric model, but with two differences.

$$SX = \{s \in domX \to (\{*\} \vee (N \wedge Loc)) \mid s\ell = \langle n, \ell' \rangle \text{ implies } \ell X \ell'\}.$$

The first difference is the use of the constraint $\ell X \ell'$ determined by the relation: the store must be compatible with the given store shape.

The second difference is that locations in the domain and range of a state s are treated differently, because the former must be in $domX$ while the latter are taken from all locations. We think of $domX$ as the collection of known or active locations at a given world. The use of Loc enables a situation where one location points to another, where that other's contents is unknown: we have dangling references. For example, in the world $\{\langle \ell, \ell' \rangle\}$ the store $[\ell \mapsto \langle 3, \ell' \rangle]$ is valid, where we do not know what ℓ' points to.

Dangling references play a crucial role in the treatment of $\bullet$. For example, in the composite world $Z = Y \bullet \{\langle \ell, \ell' \rangle\}$ we use the dangling reference ℓ' in $\{\langle \ell, \ell' \rangle\}$ to "reach back" to Y. In this composite world there cannot be any pairs of the form $\langle \ell'', \ell \rangle$, so in a state $s \in SZ$ there cannot be any pointers into ℓ. In particular, ℓ can be the head of a linked list, but not a non-head node.

Now we want to see how the non-commutative function types work in this model. Recall the sharing interpretation for $-\!\!\bullet$:

$A -\!\!\bullet B$: functions that cannot depend on the resources of their arguments.

To make this concrete we work with an object of cells, as well as with states. Since we regard the domain of a relation as the collection of known cells, we set $\mathbf{cell}X = domX$.

We are going to describe a map

$$stack\text{-}alloc : (\mathbf{cell} -\!\!\bullet (S \to S)) \longrightarrow (S \to S)$$

where $stack\text{-}alloc(\lambda_{-\!\bullet}\, x.C)$

- allocates a new location, initializes it to $*$, and binds it to x;
- executes C;
- de-allocates the new location on block exit.

The description of this operation may seem surprising, since in the presence of pointers (safe) stack allocation is not generally possible. For, if you allocate a pointer, and make some other pointer point to it, then the new pointer cannot be deallocated without creating a dangling reference.

Ultimately, $stack-alloc$ works because the right-leaning function type allows us to express a kind of dependency: x may point to other, older, pointers but not the reverse. Thus, deallocating x on block exit will not create any dangling pointers. And, if we know that C does not itself create a dangling pointer – say, if there are no facilities for freeing or disposing a pointer in the language – then this form of stack allocation of pointers will be completely safe.

To nail this down, first note that a function $f \in (\mathbf{cell} -\!\!\bullet (S \to S))X$ will accept a world Y and cell ℓ and then give us back

$$f[Y]\ell : S(X \bullet Y) \longrightarrow S(X \bullet Y)$$

What we need to do is choose ℓ to be a new location: then the definition of $\bullet$ will ensure that ℓ cannot be pointed to from X. We also need to choose the relational constraint for Y, and for this it makes sense to let ℓ point to anything in X.

To formalize this discussion we require

- a location $\mathit{newloc}(X) \notin \mathit{dom}X$.

This location can be chosen using some enumeration of *Loc*. Next, we define

- $\mathit{newworld}(X) \;=\; \{\langle \mathit{newloc}(X), \ell\rangle \mid \ell \in \mathit{dom}X \;\vee\; \ell = \mathit{newloc}(X)\}$.

The composite world $X \bullet \mathit{newworld}(X)$ describes a situation where a new location can point to locations in X, but not conversely. Then

$$\mathit{stack\text{-}alloc}[X]fs \;=\; \mathit{chop}\big(f[\mathit{newworld}(X)]\mathit{newloc}(X)\,[s \mid \mathit{newloc}(X) \mapsto *]\big)$$

where *chop* takes a state in $X \bullet \mathit{newworld}(X)$ and removes $\mathit{newloc}(X)$ from its domain, giving us a state in X. This chopping operation does not create any new dangling references. That is, if there is no dangling in s', then there will be none in $\mathit{chop}(s')$. The point is that the final state, just before chopping, must obey the constraint described by $X \bullet \mathit{newworld}(X)$. The directional sharing information expressed by $-\!\!\bullet$ and $\bullet$ should be clear from this example.

On the other hand, suppose we were to try the same thing with $\bullet\!\!-$

$$(\mathbf{cell} \bullet\!\!- (S \to S)) \longrightarrow (S \to S)$$

Then we could make the same definition as above, by redefining $\mathit{newworld}(X)$ and using $\mathit{newworld}(X) \bullet X$. But then we would no longer be guaranteed of safety of deallocation, because chopping the new location could create a dangling reference.

We took so long over this example because the resource reading of non-commutative features is delicate. But again, no particular practical significance is claimed. To go further, in particular, one would want to allow some form of heap allocation, or one might even regard the elements of worlds as *regions* of locations rather than single locations.

Remark 13 The definition of $*$ used here might be termed *shallow non-interference*. For, in $X * Y$ it is possible for the ranges of relations X and Y to overlap, or overlap with the domains. A deep notion of non-interference can be defined using

$$X * Y \;=\; X \cup Y, \text{ when } (\mathit{dom}X \cup \mathit{cod}X) \cap (\mathit{dom}Y \cup \mathit{cod}Y) = \{\}.$$

6 More on Categorical Models

In this section we look more closely at some properties of the categorical models. The reader who is more interested in seeing $\alpha\lambda$ in action can safely skip forward to the next section on SCI.

6.1 Non-Examples and an Obstruction

Notice first the difference between dcc's and models of linear logic. For example, the category of predomains (ω-complete partial orders) and continuous functions is cartesian closed, with $\to$ the continuous function space, whereas a different category, that of strict functions on pointed domains, has the strict function space $\multimap$ as the symmetric monoidal closed exponent. The lifting functor from predomains to domains has a right adjoint which, by composition, gives the lifting comonad on the category of strict maps. This gives us a model of linear logic, but not a dcc (except in the trivial sense where the two closed structures of a dcc are taken to be one and the same). The key point is that a dcc requires two closed structures on one category, where in domain theory $\multimap$ and $\to$ give closed structures for different categories. Other models of linear logic are similar non-examples, including the original coherence space model.

Although $-\!\!*$ and $\rightarrow$ are not in general convertable to one another, the dcc isomorphisms

$$[1, A \rightarrow B] \;\cong\; \mathcal{C}[A, B] \;\cong\; [I, A -\!\!* B]$$

do place a demand on non-degenerate models. We can make this precise by establishing an obstacle to the existence of non-degenerate dcc's, which rules out categories such as **Set** or the category of predomains. To state this, recall that a category with a terminal object is well pointed if for any parallel maps $f, g : X \rightarrow Y$ there is $e : 1 \rightarrow X$ such that $e; f \neq e; g$.

Proposition 14 *If a cartesian dcc is well pointed then it is degenerate in at least one of the following two senses:*

(a) it is a preorder (at most one map in any hom set), or

(b) the units 1 *and* I *of the monoidal structures coincide (up to isomorphism).*

Proof: Suppose $\mathcal{C}$ well pointed and not a preorder. We show that that $I \cong 1$.

To see that I is weakly terminal, since $\mathcal{C}$ is not a preorder there are two unequal maps $A \longrightarrow B$ for some A and B. By adjointness we obtain two maps $I \longrightarrow A -\!\!* B$ which, by well pointedness, can be distinguished by a map $1 \longrightarrow I$. For any object D we can compose that map with $D \longrightarrow 1$, thus showing that I is weakly terminal.

For uniqueness of the map $D \longrightarrow I$ we make use of the following two facts.

(i) [15] If $\mathcal{C}$ is a well pointed category then there is only one natural endomorphism on the identity functor $id_{\mathcal{C}} : \mathcal{C} \longrightarrow \mathcal{C}$.

(ii) [14] If $(\mathcal{C}, *, I)$ is a monoidal category then there is an injective function from $\mathcal{C}[I, I]$ into $\mathcal{C}^{\mathcal{C}}[id_{\mathcal{C}}, id_{\mathcal{C}}]$.

It follows that the identity is the only endomorphism on I in $\mathcal{C}$. Now, suppose (toward contradiction) that there are two maps $D \longrightarrow I$. Then there would be a map $1 \longrightarrow D$ distinguishing them by well pointedness, and composing gives us two maps $f, g : 1 \longrightarrow I$. By well pointedness we know that $\mathcal{C}$ is equivalent to a (not necessarily full) subcategory of **Set**, the category of its points (whose objects are hom sets $\mathcal{C}[1, A]$), where 1 corresponds to a one-point set. From this it is evident that composing on the left with the unique map $h : I \longrightarrow 1$ we obtain two different endomorphisms $(h; f) \neq (h; g) : I \longrightarrow I$. But we saw above that there could only be one such endomorphism, to we obtain a contradiction. So there can be at most one map $D \longrightarrow I$, and thus I is terminal and isomorphic to 1. ■

A preliminary version of this paper (from October, 1997) contained the erroneous claim that in a well-pointed dcc $*$ and $-\!\!*$ would collapse to $\wedge$ and $\rightarrow$ as well. However, Martin Hofmann has constructed well-pointed affine dcc's in which the products and function types are indeed distinct [19].

Nonetheless, the proposition does establish an obstacle to the search for models by a number of standard techniques. For example, realizability models are often given using partial equivalence relations over a partial combinatory algebra; the maps are those functions on equivalences classes that can be tracked by an element in the algebra. But such categories are well pointed, so this construction cannot be used to give non-degenerate realizability models of $\alpha\lambda$. Indeed, the problem of finding a convincing realizability interpretation of $\alpha\lambda$ remains open.

6.2 Day's Construction

In [11], Day shows how to obtain closed structures in functor categories. Day's definitions are phrased in the language of enriched category theory, and use what is termed a premonoidal structure on a category. A very particular case of his construction gives a monoidal biclosed structure on $\mathbf{Set}^{\mathcal{C}^{op}}$, given a (small) monoidal category $(\mathcal{C}, I, *)$.

$$\begin{array}{lcl}
(A \mathrel{-\!\!*} B)Z & = & \mathbf{Set}^{\mathcal{C}^{op}}[A, B(Z * -)] \\
(A \mathrel{*\!\!-} B)Z & = & \mathbf{Set}^{\mathcal{C}^{op}}[A, B(- * Z)] \\
(A * B)Z & = & \int^{X,Y} \; AX \times BY \times \mathcal{C}[Z, X * Y] \\
IZ & = & \mathcal{C}[Z, I].
\end{array}$$

The given formulas are all contravariant in Z, giving the morphism parts of the functors. Further, when $\mathcal{C}$ is symmetric monoidal so is the induced structure on $\mathbf{Set}^{\mathcal{C}^{op}}$, and similarly for affine structure. Combined with the standard fact that $\mathbf{Set}^{\mathcal{C}^{op}}$ is cartesian closed, this construction gives us a host of models for bunched typing.

The use of a coend in the formula for $*$ is ... daunting: it is generally calculated as a complicated quotient of a very large coproduct [24]. Some readers may find the following computational intuition helpful. Think of the coend as if it were an existential type

$$\exists X. \exists Y.\ AX \times BY \times (Z \to_{\mathcal{C}} X * Y).$$

That is, an element consists of representation types X and Y, and elements a, b, f of the indicated types, with a uniformity condition to factor out inessential differences between representations, like in abstract types. We warn that this is only a rough approximation, as the uniformity condition depends heavily on the morphisms in $\mathcal{C}$; but we have found it is a useful approximation nonetheless.

In specific examples it is sometimes possible to give an explicit description of $*$, without using the coend. Also, an analogue of the multi-map characterization, discussed in Section 5.1, works generally for Day's product, and this gives us a concrete way to understand maps out of a tensor.

Besides giving us a host of models, Day's construction enables us to make a remark about full and faithful embeddings. Faithfulness is the semantic counterpart of a syntactic conservativity result, while fullness says that adding such structure does not cause any new maps to added, when we focus on just ccc or smcc types being embedded.

Proposition 15 ([12]) *Suppose $\mathcal{C}$ is a (small) symmetric monoidal closed category. Then the Yoneda embedding takes the symmetric monoidal closed structure of $\mathcal{C}$ to that on $\mathbf{Set}^{\mathcal{C}^{op}}$, as given by Day's construction.*

Since $\mathbf{Set}^{\mathcal{C}^{op}}$ is a dcc, this can be read as showing the conservativity of $\alpha\lambda$ over multiplicative, intuitionistic linear λ-calculus. (At least, modulo an axiomatization of dcc equality [39].)

We can embed a ccc in a dcc in a trivial way, by regarding it as a dcc in which the two closed structures coincide. This implies the conservativity of the equality on $\alpha\lambda$-terms given by cartesian dcc's over that for simply-typed λ-calculus.

Finally, if we combine Proposition 15 with the well known fact that the Yoneda embedding preserves cartesian closed structure (when it exists) [48], then we can see that Day's construction covers essentially all models of the $(0, \vee)$-free fragment of $\alpha\lambda$: a cartesian dcc embeds into $\mathbf{Set}^{\mathcal{C}^{op}}$ in a way that preserves all the relevant structure. In this sense, it can truly be said that many of the semantic fundamentals of $\alpha\lambda$ had already been worked out in [11, 12].

However, one further point is worth making: if we consider coproduct types then there are many models that fall outside the scope of the construction, as explicitly stated. The reason is that Yoneda does not preserve coproducts. But it can be made to preserve them, by replacing $\mathbf{Set}^{\mathcal{C}^{op}}$ with a category of sheaves (this would take us outside the scope of the present paper, and is a subject examined further in [35]).

Day's construction as stated here starts with a monoidal category, where $*$ is a *total* operation on worlds. In constrast, in our example models $*$ was partial, and was defined only when the component worlds were subject to a constraint. The models in Section 5 can be described using Day's premonoidal structures, which is a generalization of monoidal structure that (among other things) allows for partiality. Rather that spell this notion out, we can connect the model description from Section 5 to that here by using an additional world, $\perp$, to represent undefinedness.

Recall the set $\mathcal{F}$ from Section 5.1. Let $\mathcal{F}_\perp$ denote the poset obtained from $\mathcal{F}$ by adjoining a new least element, $\perp$. The product category $\mathbf{Set}^{\mathcal{F}}$ is equivalent to the full subcategory of $\mathbf{Set}^{\mathcal{F}_\perp{}^{op}}$ whose objects map are such that $A\perp$ is a singleton. And adapting $*$ to the objects of $\mathcal{F}\perp$, we can define $X * Y = \perp$ when $\neg(X \ll Y)$, and $X * \perp = \perp * Y = \perp$. This gives us a monoidal structure on $\mathcal{F}_\perp$.

The concrete description of $*$ in Section 5.1 is just a representation of Day's product, tuned to the particularly simple structure of $\mathcal{F}$. To see this, consider the coend formula for $(A*B)X$, when $X \neq \perp$. An element of this set is (an equivalence class for) a tuple $\langle Y, Z, a \in AY, b \in BZ, f \in \mathcal{F}_\perp[X, Y * Z]\rangle$. Since X is not $\perp$, the definition of $\mathcal{F}_\perp$ requires that $Y*Z = X$. Further, this hom set has only one element, so f is uniquely determined by the information that $Y * Z = X$. Finally, because $\mathcal{F}$ is discrete, there are no dinaturality conditions to be met, and the equivalence relation on tuples generated by the coend forula is in fact trivial. In case $X = \perp$, the dinaturality condition causes all appropriately-typed tuples $\langle Y, Z, a, b, f\rangle$, to be equated, and $(A * B)\perp$ is a one-point set.

The multiplicative product $\Pi_{\#}Y$ used in the concrete description of $-\!\!*$ quantifies only over those worlds where $X\#Y$. This is equivalent to Day's formula, because in case $\neg(X\#Y)$ there will only be one element of $\mathbf{Set}[AY, B(X * Y)]$, if we assume that B sends $\perp$ to a singleton. So an element of $\mathbf{Set}\mathcal{F}_\perp{}^{op}[A, B(X * -)]$ is completely determined by its action on Y where $X\#Y$.

The connection of the affine and non-commutative models to Day's construction is similar. In the case of the affine model, the use of an equivalence relation in the description of $A * B$ is the uniformity induced by the coend formula.

6.3 Other Models

The first model is trivial, in that it is a poset (and even a boolean algebra).

Example 16 Consider the two-element boolean albebra $B = \{f, t\}$. It can be viewed as a degenerate (posetal) ccc, where $f \sqsubseteq t$, $1 = t$ and $\wedge$ and $\rightarrow$ are given by the truth tables for conjunction and implication. The product poset $B \times B$ inherits this ccc structure in a pointwise fashion, and it has symmetric monoidal closed structure given by

$$\begin{array}{rcl} I & = & (t, f) \\ (A_0, A_1) * (B_0, B_1) & = & ((A_0 \wedge B_0) \vee (A_1 \wedge B_1)\,, (A_0 \wedge B_1) \vee (A_1 \wedge B_0) \\ (A_0, A_1) -\!\!* (B_0, B_1) & = & ((A_0 \rightarrow B_0) \wedge (A_1 \rightarrow B_1)\,, \\ & & \quad (A_0 \rightarrow B_1) \wedge (A_1 \rightarrow B_0)) \end{array}$$

Coproduct structure is given by join in $B \times B$.

We can use this model to confirm the remark from Section 3.2 that $-\!\!*$ and $\rightarrow$ are not convertable to one another in the $\alpha\lambda$-calculus. To see this, note that $((f,t) \rightarrow (t,f)) = (t,f)$ and $((f,t) -\!\!* (t,f)) = (f,t)$. This, combined with the fact that there are no maps between (f,t) and (t,f) in either direction, implies that there are no maps from $((f,t) \rightarrow (t,f))$ to $((f,t) -\!\!* (t,f))$ or back. ■

A less trivial example example is given by the category **Cat** of small categories.

Example 17 **Cat** is cartesian closed, with product of categories and the one object category giving finite products and the functor category A^B giving the additive exponent. **Cat** also has another closed structure, where $A -\!\!* B$ is the category whose objects are functors and whose morphisms are "transformations," i.e. families of maps but without naturality constraints. The symmetric monoidal structure is given by Gray's tensor product [18] with the one object category as unit. So **Cat** is an affine, bicartesian dcc. These are the only symmetric monoidal closed structures on **Cat** [14]. ■

A number of other naturally occurring examples arise from Day's construction, including a model of higher-dimensional automata [17] and complexity models [19].

6.4 On Adding "!" to $\alpha\lambda$, or BI

In section 3.2 we showed how $\alpha\lambda$ and linear λ-calculi mix additive and multiplicative function types in a fundamentally different way. We probe this issue further by considering what happens when we add "!" to $\alpha\lambda$. To study this question we will not formulate explicit syntactic rules for "!", but rather will work exclusively at the semantic level. Also, we will consider models that include coproduct types, so will work with bicartesian dcc's (cartesian dcc's that have coproducts), which are models of the full system of BI.

We begin by noting an important fact.

Proposition 18 *There is model of $\alpha\lambda$ for which the decomposition $!A -\!\!* B \cong A \rightarrow B$ is impossible. That is, there is no functor "!" which can decompose the additive function type into the multiplicative one, in that specific model.*

Proof: Consider the affine model from Section 5.2. We claim that there is no functor (or even function on objects) $! : \mathbf{Set}^{\mathcal{FS}} \rightarrow \mathbf{Set}^{\mathcal{FS}}$ that gives us the isomorphism $!A -\!\!* B \cong A \rightarrow B$. Consider the constant functor 2 which delivers the two element set $\{t, f\}$. Then

$$(A -\!\!* 2)X \;=\; \Pi_{\#}Y.\, A(Y) \Rightarrow 2(X * Y) \;=\; \mathbf{Set}^{\mathcal{FS}}[A, 2]$$

is independent of X, and so $A -\!\!* 2$ is a constant functor. On the other hand,

$$(A \rightarrow 2)X \;=\; \mathbf{Set}^{\mathcal{FS}}[\mathcal{FS}[X, -] \wedge A\;,\; 2]$$

depends on X, and is not necessarily (isomorphic to) a constant functor. For instance, if L is the inclusion functor from $\mathcal{FS}$ into **Set**, then $(L \rightarrow 2)\{\}$ has two elements, corresponding to the two constant functions into $\{t, f\}$, and naturality can be used to show that these are the only two elements. But, $(L \rightarrow 2)\{a, b\}$ has elements that are not in the range of $(L \rightarrow 2)(f : \{\} \hookrightarrow \{a, b\})$. One such maps a to t and b to f (and all other inputs to, say, f). Therefore, no matter what "!" we try to pick, $!L -\!\!* 2$ will be a constant functor, while $L \rightarrow 2$ is not, so they cannot be isomorphic. ■

This result indicates that a dcc is not simply a model of linear logic in disguise. There is no obstacle to the existence of *some* "!" satisfying the required properties for linear logic. It just shows that, in general, we cannot expect such a "!" to decompose the additive function type that exists in the $\alpha\lambda$ model.

This leaves open the possibility, then, of having a category that is simultaneously a model of $\alpha\lambda$ and a model of linear logic. Here, we will take "model of linear logic" to mean a symmetric monoidal closed category, with products and coproducts, and equipped with a "monoidal comonod"; these are the models of intuitionistic linear logic, as presented in [6]. So we explicitly define:

Definition 19 *An* $\alpha\lambda$ with "!" model *is a bicartesian dcc with a monoidal comonad structure (where the monoidal structure used is that for* $(*, I)$*).*

(These are the minimum conditions we would expect from a model for a system combining $\alpha\lambda$ with "!".)

The affine model of Section 5.2 provides an example. There, for "!" we choose the functor where $!AX = A\{\}$. We omit the further data needed to describe a monoidal comonad, and simply state:

Proposition 20 $\mathbf{Set}^{\mathcal{FS}}$ *with the indicated structure is an* $\alpha\lambda$ *with "!" model.*

This gives us a model with an additive function type $A \to B$, the exponent in the functor category, together with an additional function type $!A \mathrel{-\!\!*} B$ gotten by decomposition. To see how different the decomposed function type is, consider $p \in (!A \mathrel{-\!\!*} B)X$. This gives, for any world Y, disjoint from X, a function from $A\{\}$ to $B(X * Y)$, which is completely determined by a function from $A\{\}$ to BX. So, such a function effectively accepts only "resource unconscious" arguments. In fact, we can see that this comonad essentially arises from an adjunction between $\mathbf{Set}^{\mathcal{FS}}$ and $\mathbf{Set}$. The left adjoint takes any functor A to $A\{\}$ and the right adjoint takes a set to the constant functor on it.

So, we see that it is possible to add "!" to $\alpha\lambda$, and that doing so does not completely collapse the system (at least, we have seen how to do so semantically). But if we add "!" we shoudld ask how the resulting system relates to linear logic. Clearly, the system is an extension of linear logic, but it is not a conservative extension of it because of the following fact.

Fact 21 *Any* $\alpha\lambda$ *with "!" model satisfies distribution:*

$$A \wedge (B \vee C) \;\cong\; (A \vee B) \wedge (A \vee C),$$

where $\wedge$ *is product and* $\vee$ *coproduct. However, distribution fails in models of linear logic (for instance, in the coherence space model).*

This fact follows at once from the requirement that $A \wedge (-)$ be a left adjoint, which hence must preserve coproducts.

We could go on and ask further technical questions, such as whether adding "!" is conservative over $\alpha\lambda$. However, a more important question is whether such an extension is of any interest. Fact 21 rules out the most important models of linear type theory that have been given in the literature. An exception is the model of passivity in syntactic control of interference, based on bireflective subcategories [29, 15]. But for that model additional properties, above and beyond those associated with "!", are crucial.

Here we have discussed models that consist of a single category, possessing all of the properties required to model both linear logic or type theory and BI or $\alpha\lambda$. There is another way that one might try to combine linear and bunched typing, based on a pair of categories. In the pair-of-category models of linear logic one asks for a symmetric monoidal category and a separate catesian (perhaps closed) category, with a monoidal adjunction between them [5, 2]. To obtain a combined linear/bunched type system one might start with a symmetric monoidal closed category and a separate cartesian dcc. (The details are by no means obvious, and this is a direction for future work.)

7 Interference Control

Now we switch gears and show a detailed use of $\alpha\lambda$. In this section we describe syntactic control of interference and Idealized Algol, two imperative languages defined by Reynolds in the late 70s and early 80s. The following section shows how SCI and IA can be combined into a single language, whose type system is based on the affine $\alpha\lambda$-calculus. This combined language overcomes a problem with recursion in the original SCI. After that, we indicate how $\alpha\lambda$ can be used to treat jumps, another problem area in the original SCI.

Experience shows that SCI can be very difficult to understand if presented too quickly. Therefore, we will include a number of small examples, and much informal discussion, in this section. The main focus, again, is on the connection between structural rules and sharing, and how this is different from the way of understanding these rules in terms of how many times a piece of data is used.

7.1 Basic SCI

We work with a version of SCI whose types are as follows.

$$\begin{array}{lcll} \rho & ::= & \mathbf{exp} \mid \mathbf{cell} \mid \mathbf{comm} & \text{primitive types} \\ \theta & ::= & \rho \mid \theta \wedge \theta' \mid \theta \to \theta' & \text{types} \end{array}$$

The primitive type **exp** is the type of natural number-valued expressions, **comm** is the type of commands, and **cell** is the type of storage cells, or locations.

Affine λ-calculus

$$\frac{}{x : \theta \vdash x : \theta}\ Id \qquad \frac{\Gamma \vdash M : \theta}{\Delta \vdash M : \theta}\ Ex \text{ (where } \Delta \text{ is a permutation of } \Gamma)$$

$$\frac{\Gamma \vdash M : \theta'}{\Gamma, x : \theta \vdash M : \theta'}\ W$$

$$\frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x : \theta.\, M : \theta \twoheadrightarrow \theta'}\ {\twoheadrightarrow} I \qquad \frac{\Gamma \vdash M : \theta \twoheadrightarrow \theta' \quad \Delta \vdash N : \theta}{\Gamma, \Delta \vdash MN : \theta'}\ {\twoheadrightarrow} E$$

$$\frac{\Gamma \vdash M : \theta \quad \Gamma \vdash N : \theta'}{\Gamma \vdash \langle M, N \rangle : \theta \wedge \theta'}\ \wedge I \qquad \frac{\Gamma \vdash M : \theta_1 \wedge \theta_2}{\Gamma \vdash \pi_i\, M : \theta_i}\ \wedge E \text{ (where } i \text{ is 1 or 2)}$$

A typing context Γ in SCI is a list of assumptions $x : \theta$ pairing identifiers with types, with the proviso that no identifier appears twice.

The crucial rule is $\to E$, where the use of distinct contexts Γ and Δ prevents the procedure and argument from sharing identifiers (the proviso that no identifier appears twice in a context puts an implicit constraint on Γ, Δ).

SCI-Specific Rules

$$\frac{\Gamma \vdash M : \mathbf{comm} \quad \Delta \vdash N : \mathbf{comm}}{\Gamma, \Delta \vdash M \parallel N : \mathbf{comm}} \qquad \frac{\Gamma \vdash M : \mathbf{comm} \quad \Gamma \vdash N : \mathbf{comm}}{\Gamma \vdash M; N : \mathbf{comm}}$$

$$\frac{}{\Gamma \vdash 17 : \mathbf{exp}} \qquad \frac{\Gamma \vdash N_1 : \mathbf{exp} \quad \Gamma \vdash N_i : \mathbf{comm}\ ,\ i = 2, 3}{\Gamma \vdash \mathbf{if}\ N_1 = 0\ \mathbf{then}\ N_2\ \mathbf{else}\ N_3 : \mathbf{comm}}$$

$$\frac{x : \theta \vdash M : \theta}{\vdash \mathbf{rec}\, x.\, M : \theta} \qquad \frac{\Gamma, x : \mathbf{cell} \vdash M : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, x.\, M\ : \mathbf{comm}}$$

$$\frac{\Gamma \vdash\ M : \mathbf{cell}}{\Gamma \vdash\ M : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{cell} \quad \Gamma \vdash N : \mathbf{exp}}{\Gamma \vdash M := N : \mathbf{comm}}$$

We have included a rule for impicit dereferencing, which converts a term of type **cell** to one of type **exp**. Most of the other rules should be familiar; we mention only that **new** allocates a fresh cell (which is put on the runtime stack). We have not listed typical arithmetic operations.

Now let us reconsider the example from the Introduction, which leads to aliasing:

$$((\lambda x \lambda y. \cdots x := 1; y := 2 \cdots\cdots) z) z.$$

This term does not typecheck in SCI because the function $((\lambda x \lambda y. \cdots x := 1; y := 2 \cdots\cdots) z)$ and argument z share the free identifier z: there is no way to apply the elimination rule for $\rightarrow$.

The parallel composition $M \parallel N$ is included alongside $M; N$ for contrast. If interference control is working properly then we would expect, because of the use of disjoint contexts, that the commands M and N refer to distinct areas of storage in $M \parallel N$. As a result, its overall effect should be determinate, and semantically equivalent to the sequential composition $M; N$ (when $M \parallel N$ typechecks). For example, $x := x + 1 \parallel y := 2$ is perfectly determinate, as long as x and y denote distinct cells. But $x := x + 1 \parallel x := 2$, which would be interminate, is ruled out by the typing rule for $\parallel$.

Conversely, if interference control is not working properly, then we would expect this to be seen in $M \parallel N$. For example, $x := 1 \parallel y := 2$ would be indeterminate if x and y were aliases.

The restriction on the rule for recursion is what one expects for affine typing. If M had free identifiers other than x then a fixed-point unwinding $\mathbf{rec}\, x.M \ \triangleright\ M[\mathbf{rec}\, x.M/x]$ would violate affine typing. This can be seen also with a fixed-point combinator $Y(M)$ where $M : A \rightarrow A$: an unwinding to $M(Y(M))$ would violate the disjointness property of procedure calls, if M was not closed.

As we mentioned in the Introduction, the original SCI allowed a restricted form of Contraction for *passive* types, which are types of values that may read from, but not write to, the store. Passivity is discussed briefly in Section 8.3.

7.2 The Sharing Interpretation of SCI

We saw above how abolishing Contraction eliminates one instance of aliasing. More generally, the absence of aliasing is subsumed under the

Disjointness Policy: distinct identifiers never interfere.

In Basic SCI we take "interfere" to mean "refer to common storage". (This must be modified to allow sharing of read access, when passivity is considered.)

The disjointness policy impacts the meaning of function types, while the meaning of products remains more standard:

$A \to B$: functions that don't interfere with their arguments;
$A \wedge B$: pairs that may interfere with one another.

SCI did not, originally, have a multiplicative product. The reading for it would be

$A * B$: pairs whose components don't interfere with one another.

But a form of this product is already present in the comma in typing contexts, in that in a judgement

$$x_1 : A_1, ..., x_n : A_n \vdash M : B$$

the disjointness policy expresses the same non-interference property as for $*$.

It is important to realize how the sharing interpretation is an unusual reading of the affine λ-calculus. Often, the idea in the affine calculus is that a function uses its argument at most once, so that for instance in a function of type $A \wedge B \to C$ either the A or the B component may be used, but not both. But according to SCI's reading, it is perfectly reasonable for a function p of such a type to use either or both components of a pair $\langle a, b\rangle$ supplied to it as an argument, and either of these elements could be used many times. The only constraint is that p doesn't interfere with $\langle a, b\rangle$.

For example, in SCI we can write a function

$$(\lambda c : \mathbf{comm} \wedge \mathbf{comm}.\, \pi_1 c\, ;\, \pi_2 c\, ;\, \pi_1 c) : \mathbf{comm} \wedge \mathbf{comm} \to \mathbf{comm}$$

that uses the first component of a pair twice and the second component once.

The reading also helps to understand the typing of **if**. In the number-of-uses reading, in $\mathbf{if}\, N_1 = 0\, \mathbf{then}\, N_2\, \mathbf{else}\, N_3$ one would expect to use one context for N_1, and a separate context for N_2 and N_3. But the conditional essentially corresponds to a constant of type $\mathbf{exp} \wedge \mathbf{comm} \wedge \mathbf{comm} \to \mathbf{comm}$ in SCI and there is no inconsistency if all the N_i's share the same context. And in imperative programming this sharing is often wanted, so that information can pass from the condition into the branches.

Now the affine λ-calculus certainly does not *force* the sharing reading. But it is consistent with it. The pure affine calculus is actually too small for this "many uses" aspect to be seen; the additional features of SCI are where it comes out. The pure $\alpha\lambda$-calculus, in contrast, already admits multiplicative functions that use their arguments many times, as we saw in Section 3.2.

7.3 IA

IA (for Idealized Algol) is similar to SCI, except that it uses the simply-typed λ-calculus in place of the affine λ-calculus. Formally, it is obtained from Basic SCI by removing the rule for $\|$ and adding Contraction and a new rule for recursion

$$\frac{\Gamma, x : \theta, y : \theta \vdash M : \theta'}{\Gamma, z : \theta \vdash M[z/x, z/y] : \theta'}\ \text{Contraction} \qquad \frac{\Gamma, x : \theta \vdash M : \theta}{\Gamma \vdash \mathbf{rec}\, x.\, M : \theta}$$

Instead of adding Contraction, we could equivalently banish the disjointness requirement in the rule $\to E$ rule. IA violates the disjointness policy, as now a term $((\lambda x \lambda y. \cdots x := 1; y := 2 \cdots\cdots) z) z$ where distinct identifiers x and y interfere is typable.

7.4 A Limitation

Many programs one would typically write (in a language, or language fragment, like IA without references or pointers) do in fact satisfy the disjointness policy of SCI.

But a problem with recursion was raised by Reynolds [45] and, indeed, one example from the book [42] that violates the policy illustrates the point well. For the record, it is the classic Towers of Hanoi program, where disks are moved between pegs.

$$
\begin{array}{l}
\textbf{procedure}\, movemany(k, a, b, c : \textbf{int}) \\
\qquad\qquad \textbf{if}\, k > 0\, \textbf{then} \\
\qquad\qquad\qquad\quad movemany(k-1, a, c, b); \\
\qquad\qquad\qquad\quad moveone(a, b); \\
\qquad\qquad\qquad\quad movemany(k-1, c, b, a)
\end{array}
$$

The procedure *moveone* can work by printing a message to the screen, or by recording a move in a global data structure. The point is that *moveone* and *movemany* interfere in the body of the procedure. Other examples of this form may be found in objects where one of the methods is recursive.

It is possible to write a recursive version of *movemany* in SCI by passing *moveone* as a dummy argument, and instantiating a curried version of the procedure with the actual *movemany*. But this seems unnecessarily complex, as the given definition of *movemany* is simple and clear enough as is; as a result, it does seem to be desirable to be able to turn off interference control in local contexts, as long as we can turn it back on again in a broader context. The "problem" would be exacerbated when programming an object that uses several cells to maintain a local state.

Note that this argument is exclusively concerned with how one constructs programs, not what program behaviours are definable. In fact, because of the embedding results from Section 6.2 we do not expect a language based on $\alpha\lambda$-calculus to allow more (up to semantic equivalence) programs to be expressed at IA or SCI types.

This program can be written in IA, because there is no restriction on the rule for recursion. But this comes at the price of destroying interference control.

Given this problem with recursion it is natural to ask: Can we combine IA and SCI in a single enveloping language, without destroying the essence of either? In the next section we provide an answer, based on the $\alpha\lambda$-calculus.

7.5 On Adding "!" to SCI

We digress for a moment to consider the following question: Since IA is based on the simply-typed λ-calculus and SCI on the affine calculus, do we not obtain the desired enveloping language just by adding "!" to SCI? After all, adding "!" to the affine calculus allows you to encode the simply-typed calculus.

The answer is no. For if we simply add "!", with Promotion, Dereliction and Contraction rules (say following the syntax of [4]), then we do get a language that contains the simply-typed λ-calculus, but not IA. In SCI+'!" the intuitionistic function type is $!A \to B$, so the IA **new** construct should determine a closed term of type $!(!\mathbf{cell} \to \mathbf{comm}) \to \mathbf{comm}$. But the SCI **new** is (essentially) of type $(\mathbf{cell} \to \mathbf{comm}) \to \mathbf{comm}$ and there is no way, in SCI+"!", to recover the Algol **new**.

Given that the answer is no, we might instead simply try to add the IA **new** as a constant, or with a typing rule

$$\frac{\Gamma, x : !\mathbf{cell} \vdash M : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, x.\, M\ : \mathbf{comm}}$$

But then we run into another problem, exemplified by the term

$$\mathbf{new}\, x.\, ((\lambda y\, \lambda z.\, y := y * 2 \parallel z := z + 1)(der(x))(der(x));\ \cdots$$

Here, $der(x)$ is of type **cell**, obtained by the Dereliction operation which takes $x :!A$ to $der(x) : A$. Since x has type !**cell** here, Contraction allows $der(x)$ to be sent as an argument to both y and z. Now, there are two ways to interpret this term. On one hand, if the two occurrences of $der(x)$ are to be the same storage cell, then the determinacy of SCI is lost. On the other hand, if Dereliction creates a copy of a storage cell then determinacy is maintained, but you would then appear not to have IA as a sublanguage: the interpretation of, say, $(\lambda yz. y := 1; z := y + 2)(der(x))(der(x))$ would be problematic. (This point would not be affected by adopting a more explicit syntax for Contraction.)

In summary, simply adding "!" to SCI does not give us the desired enveloping language, and adding additional constructs (such as IA **new**) does not appear to work either. It is worth stepping back for a moment to consider a more basic reason why adding "!" does not work well. We said above that SCI is based on an unusual interpretation of affine λ-calculus, where functions don't share with their arguments. In contrast, IA functions are allowed to share freely. This is the sticking point: It is difficult to see how you can have both the SCI interpretation of a multiplicative, or affine, function type and the IA intuitionistic type, in a way that is connected by a "!" modality. To see this, imagine that the contraction map $!A \rightarrow !A \otimes !A$ means that "!"-typed elements are allowed to share: then we lose the property that SCI functions can never share. And if, instead, we interpret the map as making independent, non-interfering, copies, then we lose the property that IA functions, gotten by prepending a "!" to an argument, are allowed to share.

This is not a tight argument, but we felt it an important one to make, since the question about adding "!" arises immediately. And more importantly, it highlights some of the key differences in the $\alpha\lambda$ approach to structural rules.

8 An Enveloping Language

8.1 SCI+

Now we consider a language, SCI+, that has primitive operations similar to those in IA and SCI, but which uses the affine $\alpha\lambda$-calculus as its type system.

The types are given by the following grammar.

$$\begin{array}{lll} \rho & ::= & \mathbf{exp} \mid \mathbf{cell} \mid \mathbf{comm} \qquad\qquad \text{primitive types} \\ \theta & ::= & \rho \mid \theta \wedge \theta' \mid \theta \to \theta' \mid \theta \mathrel{-\!\!*} \theta' \qquad\qquad \text{types} \end{array}$$

So, the primitive types are as in SCI and Idealized Algol, but we include the two function types of $\alpha\lambda$, with the rules from the affine variant as in Section 3.3. We also include the rules for cartesian products.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \wedge B} \wedge I \qquad \frac{\Gamma \vdash M : A_1 \wedge A_2}{\Gamma \vdash \pi_i\, M : A_i} \wedge E \ \text{(where } i \text{ is 1 or 2)}$$

SCI+-SPECIFIC TYPING RULES.

$$\frac{\Gamma \vdash M : \mathbf{comm} \quad \Delta \vdash N : \mathbf{comm}}{\Gamma, \Delta \vdash M \parallel N : \mathbf{comm}} \qquad \frac{\Gamma \vdash M : \mathbf{comm} \quad \Gamma \vdash N : \mathbf{comm}}{\Gamma \vdash M; N : \mathbf{comm}}$$

$$\frac{}{\Gamma \vdash 17 : \mathbf{exp}} \qquad \frac{\Gamma \vdash N_1 : \mathbf{exp} \quad \Gamma \vdash N_i : \mathbf{comm}\ ,\ i = 2, 3}{\Gamma \vdash \mathbf{if}\ N_1 = 0\ \mathbf{then}\ N_2\ \mathbf{else}\ N_3 : \mathbf{comm}}$$

$$\frac{\Gamma; x : \theta \vdash M : \theta}{\Gamma \vdash \mathbf{rec}\, x.\, M : \theta} \qquad \frac{\Gamma, x : \mathbf{cell} \vdash M : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, x.\, M\ : \mathbf{comm}}$$

$$\frac{\Gamma \vdash\ M : \mathbf{cell}}{\Gamma \vdash !M : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{cell} \quad \Gamma \vdash N : \mathbf{exp}}{\Gamma \vdash M := N : \mathbf{comm}}$$

All of these rules except for **rec** are, textually, exactly the same as rules in IA or SCI. The difference is that now the comma has a different meaning than in IA, in that it refers to the multiplicative combination. If we read the IA "," as ";" in SCI+, and the SCI "," as ",", then we have omitted the rule for recursion from SCI, and rule for **new** from IA. Let us see that the omitted rules are derivable.

The IA **new** would be

$$\frac{\Gamma; x : \mathbf{cell} \vdash M\ : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, x.\, M\ : \mathbf{comm}}.$$

We can derive this at once using the SCI+ rule for **new** and the inference

$$\frac{\Gamma; x : \mathbf{cell} \vdash M : \mathbf{comm}}{\Gamma, x : \mathbf{cell} \vdash M : \mathbf{comm}}$$

which follows from the Convertability rule in the affine variant.

The recursion rule we have given is the one appropriate to IA. It has the SCI rule as a special case, using $\Gamma = 1$ and a coherent equivalence. The use of ";" instead of "," in this rule is crucial.

Given these remarks it is not difficult to show the following.

Proposition 22 1. *SCI+ has IA as a sublanguage. That is, if*

$$x_1 : A_1, ..., x_n : A_n \vdash M : B$$

in IA then

$$x_1 : A_1^\star; ...; x_n : A_n^\star \vdash M^\star : B^\star$$

in SCI+, where $(\cdot)^\star$ *maps* $\rightarrow$ *to* $\rightarrow$, λ *to* α, MN *to* $M^\circ @ N^\circ$, *and everything else (inductively) to itself.*

2. *SCI+ has SCI as a sublanguage. That is, if*

$$x_1 : A_1, ..., x_n : A_n \vdash B$$

in SCI then

$$x_1 : A_1^\circ, ..., x_n : A_n^\circ \vdash M^\circ : B^\circ$$

where $(\cdot)^\circ$ *maps* $\rightarrow$ *to* $-\!\!*$ *and everything else (inductively) to itself.*

8.2 Sharing in SCI+

There is thus a syntactic sense in which SCI+ is an enveloping language, but this in itself is unremarkable. It is still conceivable that the larger language has features that are inconsistent with the essence of IA or SCI, destroying some crucial aspect of one of smaller languages.

The sharing interpretation of $\alpha\lambda$ (Section 2.3) describes the sense in which the larger language preserves the essence of SCI; the readings of $-\!\!*$ and $\wedge$ are exactly as in SCI. Also, the reading of $\rightarrow$ is one that is appropriate to IA. To sum up:

$A \wedge B$	pairs that may access a common portion of the store
$A -\!\!* B$	procedures that don't share store with arguments
$A \rightarrow B$	procedures that may share store with arguments

The resulting sense in which the $\alpha\lambda$-calculus allows detection of interference is that whenever we see a sequence $\alpha x\, \lambda y$ or $\lambda x\, \lambda y$ we know that x and y don't interfere. So, non-interference can be inferred (in a fail-safe manner) from a simple inspection of a context. The one difference is that in Basic SCI this determination is context free. It is context sensitive in SCI+ because when we see $\alpha x\, \alpha y$ or $\lambda x \alpha y$ we don't know if x and y interfere or not.

Let us revisit the Towers of Hanoi in light of this interpretation. Using the rule for recursion we can type (with a little syntactic sugar)

$$
\begin{array}{l}
\mathit{moveone} : \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{comm} \\
\vdash \ \mathbf{rec}\, \mathit{movemany}.\, \alpha k\, a\, b\, c : \mathbf{exp} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ k > 0\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad \mathit{movemany}(k-1, a, c, b); \\
\qquad\qquad\qquad\qquad\qquad \mathit{moveone}(a, b); \\
\qquad\qquad\qquad\qquad\qquad \mathit{movemany}(k-1, c, b, a) \\
: \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{comm}
\end{array}
$$

The crucial point is that, during the typing of the body, we turn interference control off by using the bunch

$$
\begin{array}{l}
\ \ \mathit{moveone} : \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{comm} \\
;\ \mathit{movemany} : \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{comm}
\end{array}
$$

which indicates that *moveone* and *movemany* might interfere. But more globally we can turn interference control back on and, for instance, run a call to the recursive procedure in parallel with another command, as long as that command doesn't interfere with *moveone*.

$$
\begin{array}{l}
\mathit{moveone} : \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{comm}\, ,\, c : \mathbf{comm} \\
\vdash\ ((\mathbf{rec}\, \mathit{movemany}.\, \cdots)7\,1\,2\,3) \parallel c : \mathbf{comm}
\end{array}
$$

8.3 Passivity

The language Basic SCI in Section 7.1 is in fact only a fragment of Syntactic Control of Interference, which includes typing rules for passivity [45, 44, 29]. A passive entity, such as a side-effect free expression, can safely be shared without causing interference, and a passive type is one whose elements are all passive. Bunches are compatible with the approach to passivity in the SCIR type system from [29]; we briefly indicate how this is so.

The SCIR type system uses judgements $\Pi \mid \Gamma \vdash M : A$, where the context is split into a passive zone Π and an active zone Γ. The three critical rules of the system are the permeability rules of Activation and Passification, for moving identifier across the $\mid$ separator, and Contraction in the passive zone.

$$\frac{\Pi \mid x : B, \Gamma \vdash M\colon A}{\Pi, x : B \mid \Gamma \vdash M\colon A}\ \text{Passif (where } A \text{ is passive)} \qquad \frac{\Pi, x : B \mid \Gamma \vdash M\colon A}{\Pi \mid x : B, \Gamma \vdash M\colon A}\ \text{Activ}$$

$$\frac{\Pi, y : B, z : B \mid \Gamma \vdash M : A}{\Pi, y : B \mid \Gamma \vdash M[y/z] : A}\ \text{Contr}$$

The zonal presentation does not work well with bunches, because we would want to be able to indicate that an identifier is passive without saying that a whole bunch is.

The solution is to allow "marked assumptions" $(x^* : A)$ alongside "normal assumptions" $(x : A)$. Then, with an extension of marking to contexts, the three rules are as follows:

$$\frac{\Gamma(\Delta) \vdash M\colon A}{\Gamma(\Delta^*) \vdash M\colon A}\ \text{Passif (where } A \text{ is passive)} \qquad \frac{\Gamma(\Delta^*) \vdash M\colon A}{\Gamma(\Delta) \vdash M\colon A}\ \text{Activ}$$

$$\frac{\Gamma(\Delta^*, \Delta'^*) \vdash M : A}{\Gamma(\Delta) \vdash M[i(\Delta^*)/i(\Delta'^*)] : A}\ \text{Contr (where } \Delta^* \cong \Delta'^*)$$

This gives us a limited form of Contraction for ",", in addition to the general Contraction for ";". These rules can all be interpreted using the *bireflective subcategory* structure found in Tennent's model [29, 15].

8.4 Remaining Limitations

The language uses call by name as its parameter passing mechanism. The extension of the approach to call by value does not appear to raise insuperable difficulties, but the typing rules required tend to become more complex when one wants to separate out the effects caused by evaluation to a value, from those that are latent. Latent effects occur only when using a value, not during the computation that produces it; for example, in $x := 1; (\lambda y.\, z := 2)$ the assignment to z is latent. In call by name, all effects at higher order are latent.

A more significant limitation is that it is not obvious how to incorporate higher-order store, where a reference may hold a procedure or another reference as its contents. This problem has been addressed using different methods, and with some success, by the effect systems of Gifford and Lucassen [23]. The types used in effect systes are, however, very detailed, and they seem more suitable to an intermediate language used in a compiler (where their complexity can be hidden from the programmer) than in a source language. It would be worthwhile to develop an abstract or high level form of control over higher-order refences, more along the lines of SCI.

In a recent development, Smith, Walker and Morrissett have devised a fascinating system for interference control [49, 53], which correctly handles higher-order store, and which is remarkably similar in structure to a program logic based on BI [20]. Despite the suggestive structural similarity, the relationship of their work to bunched typing is not yet clear.

9 A Model for SCI+

Our purpose in this section is to back up the informal interpretation of types from Section 8.2. So we will focus on describing the structure of the model, and how it relates back to the informal description.

9.1 Semantics of Types

We are going to use an affine model, similar to the one in Section 8.2. However, as explained in [31], in order for variable allocation to be natural it is necessary to allow for injective renaming of locations. Therefore, we work with the category $\mathcal{I}$ of finite sets and injective functions, rather than a category where the morphisms are just subset inclusions. Also, to treat recursion we use domains in place of sets.

Let **Predom** denote the category of predomains (ω-complete posets and continuous maps). We now describe several functors in $\mathbf{Predom}^{\mathcal{I}}$, beginning with functors for the primitive types of SCI+.

For X a finite set, we define

$$\begin{aligned} SX &= X \Rightarrow N \\ [\![\mathbf{comm}]\!]X &= SX \Rightarrow SX_{\perp} \\ [\![\mathbf{exp}]\!]X &= SX \Rightarrow N_{\perp} \\ [\![\mathbf{cell}]\!]X &= X_{\perp}. \end{aligned}$$

SX is the set of states at world X, and N is the set of natural numbers. Variations are possible. For example, we could allow side effects in expressions, using $SX \Rightarrow (N \times SX)_{\perp}$, or we could make cells dependent on the store (this allowing conditional storage cells).

The action of each primitive type on morphisms f in $\mathcal{I}$ is defined by renaming cells according to f and ignoring cells not in the range of f. The cases of **exp** and **cell** are simple:

$$\begin{aligned} [\![\mathbf{exp}]\!] f\, e\, s &= e(f; s), \\ [\![\mathbf{cell}]\!] f &= f_{\perp}. \end{aligned}$$

In the case of **comm**, when $f : X \to Y$,

$$[\![\mathbf{comm}]\!] f\, c\, s = \begin{cases} c(f;s)\perp & \text{if } c(f;s) = \perp \\ s' & \text{if } c(f;s) = s'' \text{ and} \\ & \forall \ell \in Y. \big(\ell = f\ell' \text{ implies } s'(\ell) = s''(\ell')\big) \text{ and} \\ & \qquad \ell \notin range(f) \text{ implies } s'(\ell) = s(\ell)\big). \end{cases}$$

S is not itself a functor: it is defined on objects, but not morphisms.

The functor category $\mathbf{Predom}^{\mathcal{I}}$ is cartesian closed, with finite products defined pointwise. The additive function type can be given a special representation, using the fact that any morphism in $\mathcal{I}$ factors into an injection $X \to X + Y$ into the left component of a disjoint union, followed by an isomorphism:

$$(A \to B)(X) = \mathbf{Predom}^{\mathcal{I}}[A(X + -), B(X + -)].$$

This accurately reflects the informal reading, in that the presence of X in the argument type $A(X + -)$ indicates how a function $p \in (A \to B)X$ may share access to X with its argument.

In this description of the function type the hom sets $\mathbf{Predom}^{\mathcal{I}}[A, B]$ are considered to be ordered pointwise. Also, $+$ is the evident functor on $\mathcal{I}$ given by disjoint union of finite sets. Notice that, in contrast to Section 5.2, the operation for composing worlds is *total*. (In $\mathcal{I}$ we have enough morphisms to give $(\{\}, +)$ a monoidal structure, where in $\mathcal{FS}$ we do not.)

The multiplicative function type once again expresses disjointness of a function from its argument:

$$(A \twoheadrightarrow B)X \quad = \quad \mathbf{Predom}^{\mathcal{I}}[A, B(X + -)]$$

To see how the semantics is working, consider the type $\mathbf{cell} \twoheadrightarrow \mathbf{cell} \twoheadrightarrow \mathbf{comm}$. Semantically, an element $p \in [\![\mathbf{cell} \twoheadrightarrow \mathbf{cell} \twoheadrightarrow \mathbf{comm}]\!]\{\}$ accepts

> two worlds Y and Z,
> cells $c \in Y_\perp$ and $e \in Z_\perp$

and produces (using $\{\} + Y + Z \cong Y + Z$)

> $p[Y]c[Z]e : S(Y + Z) \Rightarrow S(Y + Z)_\perp$

It is evident from this that the arguments c and e cannot be aliases, as they live in disjoint portions of the store at world $Y + Z$. Thus, the absence of X in $A(-)$ exactly mirrors the informal description of $\twoheadrightarrow$ in the sharing interpretation of SCI+.

To treat recursion in this model we must effect a transformation

$$\frac{p[X] : [\![\Gamma]\!]X \times [\![\theta]\!]X \longrightarrow [\![\theta]\!]X}{rec(p) : [\![\Gamma]\!]X \longrightarrow [\![\theta]\!]X} \quad .$$

Here, we uncurry p to obtain a map of type $[\![\Gamma]\!]X \longrightarrow ([\![\theta]\!]X \to [\![\theta]\!]X)$ and then compose on the right with the least fixed-point operator for pointed domains.

For this interpretation to exist we must have that each $[\![\theta]\!]X$ is pointed. And for it to be natural we require that each morphism part $[\![\theta]\!]f$ preserves least elements [36]. These properties are satisfied by all the typed in SCI+, and are part of the identification of subcategory of $\mathbf{Predom}^{\mathcal{I}}$ in the following section.

9.2 Bunches, Environments and Non-interference

[The presentation in this section will be more technical than the others, but it is mainly a pulling together of known results. Some readers may wish to skip forward, where several of the most important valuations are presented in Section 9.3, in a way that doesn't depend on the details of this section.]

To interpret bunches we could use Day's coend formula, which is calculated using colimits in **Predom**. However, colimits there are very complex to describe, and it will be simpler to identify a subcategory where the tensor product has a simple description.

The basic idea is to consider those functors that admit a notion of support. Intuitively,

> the support of a computational entity $a \in AX$ is the set $Y \subseteq X$ of locations on which it depends.

Then, we can define the tensor product as consisting of those pairs of elements whose summorts are disjoint.

Unfortunately, it is not possible to define support for all functors in $\mathbf{Predom}^{calI}$. For, it can be the case that an element $a \in AX$ "comes from" a smaller world (via an inclusion), but also that no unique smallest such world exists. The category that we work with is described as follows.

Definition 23 *$\mathcal{M}$ is the full subcategory of* $\mathbf{Predom}^{\mathcal{I}}$ *whose objects A have the following two properties:*

1. *each AX has a least element, and each $Af : AX \to AY$ preserves least elements;*
2. *A preserves pullbacks.*

The strictness part of the first condition is needed for naturality of the fixed-point operator, and the second condition enables a treatment of support. It is straightforward to verify that the manings of primitive types satisfy these conditions.

Pullbacks in $\mathcal{I}$ are similar to those in the category of sets. In particular,

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

is a pullback square, where the unlabeled arrows are inclusion functions. Because of this, for pullback-preserving functors there is always a smallest world that any $a \in AX$ comes from:

- we say that a comes from $Y \subseteq X$ iff $\exists a' \in A(Y).\, a = A(Y \hookrightarrow X)a'$, and
- we refer to the smallest world that a comes from as the support of a, written $\mathit{support}(a)$.

The smallest world in the second clause is guaranteed to exist by pullback preservation; see [28, 25] for more details.

Given this notion of support, we can define non-interference. If $a \in AX$ and $b \in BX$, then

- $a \# b \iff \mathit{support}(a) \cap \mathit{support}(b) = \{\}$.

Using this notion of non-interference we can finally describe the tensor product $*$ of functors:

$$\begin{array}{lcl} (A * B)X & = & \{(a, b) \in AX \times BX \mid a\#b\}, \\ (A * B)f(a, b) & = & (Afa, Bfb). \end{array}$$

For this definition to work correctly, it is important that $(A * B)f$ preserves non-interference and that $(A * B)X$ is complete; see [28].

Proposition 24 *$\mathcal{M}$ is an affine doubly closed category. That is, $1, \wedge, \to$ is cartesian closed structure and $1, *, -\!\!*$ monoidal closed structure.*

This gives us all the structure we need to interpret the affine $\alpha\lambda$-calculus, where we interpret a bunch by mapping "," to $*$ and ";" to $\wedge$. However, a more concrete semantics of bunches is useful. For this, we first define

$$\Gamma \vdash x\#y \iff x \text{ and } y \text{ have a "," as a common ancestor node in } \Gamma.$$

Bunches are then interpreted by relating the syntactic $\#$ to the semantic one:

$$[\![\Gamma]\!]W \;=\; \{u \in \prod_{x \in i\Gamma} \Gamma(x) \mid \Gamma \vdash x\#y \implies ux\#uy\}$$

where we are writing $\Gamma(x)$ for the type of x in Γ. This representation of environments allows us to ignore coherent equivalence, while still maintaining a relationship between "," and $*$ and between ";" and $\wedge$.

Proposition 25 *If $\Gamma \equiv \Delta$ then $[\![\Gamma]\!] = [\![\Delta]\!]$. Further, we have the following isomorphisms:*

$$[\![\Gamma, \Delta]\!] \cong [\![\Gamma]\!] * [\![\Delta]\!] \qquad [\![\Gamma; \Delta]\!] \cong [\![\Gamma]\!] \wedge [\![\Delta]\!]$$

The point of this concrete interpretation of bunches is that it makes clear that the use of the rule for $\equiv$ in $\alpha\lambda$-calculus is not in any way problematic.

9.3 Selected Valuations

We will make use of the multi-map characterization of maps out of $A * B$, where $p : A * B \longrightarrow C$ out of a tensor are in bijection with families of functions

$$\overline{p}[X][Y] : AX \times BY \longrightarrow C(X + Y) \ ,$$

natural in X and Y. Because of this characterization, we will expect maps out of a context Γ, Δ to be in bijection with families of maps where we use one world for Δ and another for Γ. Based on this assumption, we now give the semantics of several terms.

We begin with $\|$. First, we define a state transformation $c \parallel c'$, when c and c' are commands referring to disjoint store:

$$\frac{c : S(X) \longrightarrow S(X)_\perp \quad c' : S(Y) \longrightarrow S(Y)_\perp}{c \parallel c' = \lambda[s, s] : S(X+Y).\, [cs, c's'] : S(X+Y) \longrightarrow S(X+Y)_\perp}$$

where $[\cdot, \cdot] : S(X+Y) \to S(X) \times S(Y)$ is the evident isomorphism and $[cs, c's']$ is $\perp$ if either cs or $c's'$ is. Then we can interpret the term-formation rule for $\|$ as follows.

$$\frac{p[X] : [\![\Gamma]\!]X \longrightarrow [\![\mathbf{comm}]\!]X \quad q[Y] : [\![\Gamma]\!]Y \longrightarrow [\![\mathbf{comm}]\!]Y}{\Lambda X.\, \Lambda Y.\, \lambda\langle u, v\rangle.\, p[X]u \parallel q[Y]v : [\![\Gamma]\!]X \times [\![\Delta]\!]Y \longrightarrow [\![\mathbf{comm}]\!](X+Y).}$$

Here, we have used polymorphic λ-calculus notation to talk about families of maps in what should be a clear way [30].

This semantics makes obvious that different components of $\|$ act on disjoint portions of the store. In contrast, the rule for sequential composition uses the same context Γ for both commands, and so we use the transformation

$$\frac{p[X] : [\![\Gamma]\!]X \longrightarrow [\![\mathbf{comm}]\!]X \quad q[X] : [\![\Gamma]\!]X \longrightarrow [\![\mathbf{comm}]\!]X}{\Lambda X.\, \lambda u.\, (p[X]u); (q[X]u) : [\![\Gamma]\!]X \longrightarrow [\![\mathbf{comm}]\!]X}$$

where ";" is composition of partial functions. The common use of X by p and q makes clear that they access the same portion of store.

For **new** declarations we again appeal to disjointness, where the declared cell is disjoint from the store in use when a declaration begins execution. The semantic transformation is

$$\frac{p[X][Y] : [\![\Gamma]\!]X \times [\![\mathbf{cell}]\!]Y \longrightarrow [\![\mathbf{comm}]\!]X + Y}{\Lambda X.\, \lambda u.\, \lambda s.\, f(p[X][\{*\}] * (s \mid * \mapsto 0)) : [\![\Gamma]\!]X \longrightarrow [\![\mathbf{comm}]\!]X.}$$

where $f : S(X + \{*\})_\perp \to S(X)_\perp$ forgets the $\{*\}$ component.

Finally, for λ-abstraction a λ-bound variable abstracts over meanings defined in world that is separate from the world for other free identifiers.

$$\frac{p[X][Y] : [\![\Gamma]\!]X \times [\![\theta]\!]Y \longrightarrow [\![\theta]\!]X + Y}{\lambda u.\, \Lambda Y.\lambda x \in [\![\theta]\!]Y.\, p[X][Y]\langle u, x\rangle : [\![\Gamma]\!]X \longrightarrow [\![\theta \twoheadrightarrow \theta']\!]X.}$$

Thus, in $\lambda x.\, M$ the identifier x does not share storage with any other identifier free in M.

Previously, we showed that SCI+ has SCI and IA as syntactic sublanguages. The work in this section shows that

Proposition 26 *SCI+ has IA and SCI as* semantic *sublanguages.*

To be precise, what this means is

1. the semantics of SCI obtained by the translation $(\cdot)^\star$ stated in Proposition 22 is the semantics of SCI given in [28] (ignoring passivity),
2. the model obtained from $(\cdot)^\circ$ is a standard functor-category model of IA [31].

The only real differences in the various interpretations are the rules in SCI or IA that were left out of SCI+, but which were shown to be derivable. The most important case is **new**: the reason it does not present a difficulty is that, even in IA, a locally declared cell doesn't interfere with any other identifiers free in its defining block. This is why the use of "," in the SCI+ rule for **new** is semantically sufficient to capture IA's **new**.

10 Jumps

Jumps cause a problem broadly similar to the one with recursion in SCI. In this section we indicate how this problem can be overcome using $\alpha\lambda$.

To see the difficulty, consider a block **escape** x **in** M. This declares a new label which, when jumped to from within M, results in a transfer of control to the end of the block. From the point of view of continuation semantics, it binds x to the current continuation, which is a function from states to final answers that describes computation that will take place after the block is finished. This means that, if the computation associated with the current continuation changes any storage cell then x will interfere with that cell. So, in (**escape** x **in** M); $z := 4$ the identifiers z and x interfere, if z occurs within M.

Thus, from the point of view of continuation semantics, the **escape** statement violates the requirement that distinct identifiers never interfere (unless we put rather draconian conditions on identifiers appearing in or following an **escape** block). One might attempt to use a different form of semantics to give rise to a different notion of interference for labels; however, it will be simpler just to allow this interference, by arranging the typing rule so that x is set additively apart from other identifiers.

Following [36], we add a primitive type **compl** for completions (labels) and remove **comm**. We now regard **comm** as syntactic sugar for **compl** $\rightarrow$ **compl**. The semantics of the type of completions is given using a fixed domain A of answers.

$$\begin{aligned} [\![\mathbf{compl}]\!]X &= SX \Rightarrow A \\ [\![\mathbf{compl}]\!]f\,k\,s &= k(f;s) \end{aligned}$$

With this semantics, the language with the completion type has the same sense of non-interference as SCI+: the semantics of $-\!\!*$ ensures that whenever we see a sequence $\lambda x\,\lambda y$ or $\alpha x\,\lambda y$ we know that x and y access different portions of the store.

The central syntactic rules are

$$\frac{\Gamma; x : \mathbf{compl} \vdash M : \mathbf{comm}}{\Gamma \vdash \mathbf{escape}\ x\ \mathbf{in}\ M : \mathbf{comm}} \qquad \frac{\Gamma \vdash M : \mathbf{compl}}{\Gamma \vdash \mathbf{goto}\ M : \mathbf{comm}}$$

where the use of ";" in the rule for **escape** is the crucial point. These can be given a standard continuation semantics, exactly as was done by Oles [36]. For **escape**, we effect a transformation

$$\frac{[\![\Gamma; x : \mathbf{compl}]\!] \xrightarrow{f} [\![\mathbf{comm}]\!]}{[\![\Gamma]\!] \xrightarrow{f'} [\![\mathbf{comm}]\!]}$$

accomplished by binding x to the current continuation:

$$f'[X]\,u\,X'\,k\,s \;=\; f[X+X'](u' \mid x \mapsto k)\,k\,s$$

where

$$u \in [\![\Gamma]\!]X,$$

$$k \in [\![\mathbf{compl}]\!](X + X'),$$

$$i : X \to X + X' \text{ is the left injection,}$$

$$u' = [\![\Gamma]\!]iu,$$

$$s \in S(X + X').$$

Here, the extra parameter X' is occurring because $\mathbf{comm} = \mathbf{compl} \to \mathbf{compl}$ is a procedure type, and we are using the representation of $\to$ given in the last section.

goto is given by a map $g : [\![\mathbf{compl}]\!] \longrightarrow ([\![\mathbf{compl}]\!] \to [\![\mathbf{compl}]\!])$ from completions to commands, which ignores the current continuation:

$$g[X]\,k\,[X']\,k' = [\![\mathbf{compl}]\!]i\,k$$

where $i : X \to X + X'$ is again the left injection.

Finally, to illustrate the effect of interference constraints we define the parallel composition of completions.

$$\frac{\Gamma \vdash M : \mathbf{compl} \quad \Delta \vdash N : \mathbf{compl}}{\Gamma, \Delta \vdash M \parallel N : \mathbf{compl}}$$

Its semantics is given by a map $par : [\![\mathbf{compl}]\!] * [\![\mathbf{compl}]\!] \longrightarrow [\![\mathbf{compl}]\!]$ and is defined similarly to the parallel composition of commands. For this, we rely again on the multi-map characterization of maps out of $*$, and define

$$par[X][X']\,\langle k, k'\rangle\,[s, s'] = (ks \cdot k's')$$

where $(- \cdot -) : A \times A \to A$ is a function that puts together two final answers. For concreteness, we take A to be the two-point cpo $\{\top\}_\perp$ and $(-\cdot-)$ to be meet. Here, we regard an answer $\top$ as indicating termination. We admit that this use of a function on answers is *ad hoc*. It does, however, enable us to show a sense in which completions typed in contexts separated by "," do not interfere.

We have not included parallel composition for commands, because the right way to do so is not obvious. For, one of the commands in $M \parallel N$ might jump out, and ignore the current continuation. (It might be possible to use bunches to control the range of continuations; that is beyond the scope of this paper.)

11 Conclusion

11.1 On Resource and Substructural Typing

The $\alpha\lambda$-calculus and BI offer a new perspective on how control over structural rules translates into control over access to resources in a computer system. As we have suggested, the main point is the emphasis on sharing, supported by a spatial view of possible world semantics which has developed over a number of years [43, 26, 29, 30].

Section 2 charted the initial route that led us to bunched typing. This began with an analogy between syntactic control of interference and linear logic, where both systems limit the use of Contraction, followed by recalling a dilemma: although there is a formal similarity, there is also an important conceptual difference; control of Contraction in SCI is about sharing, while in LL it is primarily about duplication. Doubly-closed categories and, eventually, bunched typing were then seen as the stucture arising from Contraction-as-sharing as found in models of SCI.

The reader might feel that we are splitting hairs, as at first sight duplication versus sharing may appear to be six of one versus a half dozen of the other. But the distinction is actually critical in computer science. The number-of-uses explanation of linear logic calls to mind the notion of *temporary resources* in Operating Systems [7], the canonical example of which is a message produced by one process and consumed by another. The analogy with temporary resources is clear in several formal interpretations of linear logic, including the original coherence space model [16] and the concurrent semantics of proof expressions [1]. In contrast, the sharing interpretation of $\alpha\lambda$ concerns what is often labelled a *permanent resource*. Here, permanent does not literally mean permanent, but potentially long lived; examples include files, external devices, or portions of the store. For this kind of resource it is sharing, rather than copying, that is the prime concern.

The results of this paper give one answer to the question of whether the conceptual difference between sharing and duplication (or consumption) could be visible on the level of type-theoretic structure. We described a new calculus, the $\alpha\lambda$-calculus, which we showed differs from linear λ-calculi in several significant respects. And for each difference between the systems we were able to offer an explanation of $\alpha\lambda$'s stance by appeal to a sharing interpretation, where linear logic's stance can be understood in terms of the number of uses reading. Because separate formal systems fit each of these readings we claim that the differences are genuinely structural, and run deeper than merely having separate interpretations of the same system.

It is worth amplifying on these points, because there is much room for misinterpretation. First, we have shown that

> $\alpha\lambda$ admits a reading where the multiplicative type $-\!\!*$ disallows sharing between procedure and argument, where at the same time the additive type allows it.

Second, the existence of multiplicative functions that use their arguments multiple times or never shows that

> $\alpha\lambda$ does not admit a number-of-uses reading, where a multiplicative function uses its argument exactly once.

This second point is directly opposed to the very beginning work on linear logic, where it was established that

> linear λ-calculi do admit a number-of-uses reading, where a multiplicative function uses its argument exactly once.

What is more difficult to show is that a sharing interpretation of linear λ-calculi is impossible (for these purposes, "linear λ-calculi" only refers to systems with "!" or equivalent, and not to restricted or different systems). Nonetheless, we boldly state

> linear λ-calculi do not admit a reading where the multiplicative type $\multimap$ disallows sharing between procedure and argument, where *at the same time* the additive function type allows it.

The intuitive basis for this claim has to do with the linear decomposition $A \to B \cong\ !A \multimap B$, and its interaction with the operation of Contraction, of type $!A \multimap\ !A \otimes !A$. For, there seem to be two choices. First, if we interpret Contraction as allowing sharing, then we do obtain that $A \to B$ allows sharing, but we lose the property that $A \multimap B$ *disallows* it (in the case when A is of the form $!C$). Second, if we interpret Contraction as performing duplication, then we seem to lose the property that the additive function type allows sharing.

We therefore suggest that linear and bunched typing offer complementary type-theoretic treatments of resource.

Finally, it is worth mentioning that a related resource perspective on BI has recently been fleshed out [35]. Here we speak of the resources a function has access to which, when we erase λ-terms, corresponds to talking about proofs. A similar interpretation can be given on a purely logical level, where one views $-\!\!*$ and $\to$ as implications, and where the semantics is phrased in terms of truth conditions that appeal to a notion of resource; proofs are not mentioned. This view, which was advanced by Pym in 1997, is on one hand a straightforward adaptation of the functional interpretation derived from SCI, but on the other hand conceptually very important: there are a number of interesting models that make good sense from a truth-based perspective, independently of type-theoretic concerns [35, 20, 9]. So, although related to the interpretations given here, this logical reading should be considered genuinely different, because of its declarative character.

11.2 Further Directions

A number of questions arise as a result of the work in this paper.

First, it would be useful to have a close understanding of a wider range of concrete models. In particular, we have appealed to the notion of computer store in explicating the sharing interpretation; this was a natural choice, because the concept of store is so basic and immediately meaningful. But it could be especially valuable to have a resource model that worked without appeal to imeperative ideas, based perhaps on call graphs of λ-terms [52]. We would also like to have realizability models.

Second, we have made numerous comparisons between linear and bunched typing, but we wonder whether it is possible to have the best of both. Ideally, we would like a way to combine linear and bunched typing in a way that simultaneously accounts for sharing and consumption. (Examples of the consumptive aspect of linear typing include [22, 51, 30].) A hint of the way to proceed was given at the end of Section 6.4: we might start from a pair of categories, one to model $\multimap$ and the other to model $-\!\!*$ and $\to$. Then, we would expect to decompose $\to$ into $\multimap$, but not into $-\!\!*$. By observing such a separation, it might be possible to develop a calculus that supports number-of-uses and sharing interpretations at the same time, where the multiplicatives in the smcc have to do with consumption and those in the dcc with sharing. The main problem, beyond having convincing concrete models, is to determine the right conditions on the means of passing from one category to the other, and the corresponding syntactic rules.

Finally, a particularly important direction for future work is the extension of these ideas to dependent types, and logical frameworks.

It is increasingly being realized that substructural features in logical frameworks enable a more exact representation of the semantics of languages with state, and it stands to reason that the same should hold true for program logics as well. Important advances include LLF [10], which incorporates linear (non-dependent) function types, and the RLF system [21], which has even linear dependent function types. The problems in this area are difficult, but progress could prove especially useful in controlling sharing relationships between different parts of a specification or encoding.

Interestingly, in a precursor of bunched typing, RLF was formulated using two forms of context extension, one corresponding to linear assumptions and the other to non-linear [21]. It did this, however, with an elimination rule for additive function types that fits more closely with linear typing than the bunched typing considered here and in [34, 39]: there were required to be no linear assumptions in the the argument for additive function elimination (compare to the rule for $\to$ in DILL, as discussed in Section 3.2). An immediate challenge is to develop a type theory with both additive and multiplicative dependent function types which, when restricted to the appropriate fragment, coincides with the basic system of $\alpha\lambda$, as presented in Section 3.1.

Acknowledgements.
I am especially grateful to David Pym and John Reynolds, both for their original ideas and for numerous discussions, which provided much of the impetus for the work in this paper. Martin Hofmann and Guy McCusker pointed out glitches in an early version. This research was supported by the EPSRC and, during a one month visit to Carnegie Mellon in July, 1997, by the NSF.

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] A. Barber and G.D. Plotkin. Dual intuitionistic linear logic. Submitted, October 1997.

[3] N.D. Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–414, 1982.

[4] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezen and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90, Utrecht, The Netherlands, March 1993. Springer-Verlag, Berlin.

[5] P. N. Benton. A mixed linear and non-linear logic: proofs, terms and models. Proceedings of *Computer Science Logic '94*, Kazimierz, Poland. Springer-Verlag LNCS 933, 1995.

[6] G. Bierman. What is a categorical model of intuitionistic linear logic? Proceedings of TLCA'95, LNCS 902, 1995.

[7] P. Brinch Hansen. *Operating System Principles.* Prentice Hall, New Jersey, 1973. Series in Automatic Computation.

[8] S. Brookes, M. Main, A. Melton, and M. Mislove, editors. *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*, Tulane University, New Orleans, Louisiana, March 29–April 1 1995. Elsevier Science.

[9] L. Cardelli and A. D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, 2000. ACM, New York.

[10] I. Cervesato and F. Pfenning. A linear logical framework. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science — LICS'96*, pages 264–275. IEEE Computer Society Press, 1996.

[11] B. J. Day. On closed categories of functors. In S. Mac Lane, editor, *Reports of the Midwest Category Seminar*, volume 137 of *Lecture Notes in Mathematics*, pages 1–38. Springer-Verlag, Berlin-New York, 1970.

[12] B. J. Day. An embedding theorem for closed categories. In G.M. Kelly, editor, *Category Seminar, Sydney*, volume 420 of *Lecture Notes in Mathematics*, pages 55–64. Springer-Verlag, Berlin-New York, 1974.

[13] J. M. Dunn. Relevant logic and entailment. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, vol. III: Alternatives in Classical Logic*, number 166 in Synthese Library, pages 117–224. D. Reidel, Dordrecht, Holland, 1986.

[14] F. Foltz, C. Lair, and G. M. Kelly. Algebraic categories with few monoidal biclosed structures or none. *J. Pure and Applied Algebra*, 17:171–177, 1980.

[15] P.J. Freyd, P. W. O'Hearn, A. J. Power, R. Street, M. Takeyama, and R. D. Tennent. Bireflectivity. *Theoretical Computer Science*, 228(1-2):49–76, October 1999. Preliminary version in [8].

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[17] E. Goubault. *The Geometry of Concurrency.* Ph.D. thesis, Ecole Polytechnique, 1995.

[18] J. W. Gray. *Formal Category Theory – Adjointness for 2-Categories*, volume 391 of *Lecture Notes in Math.* Springer, 1974.

[19] M. Hofmann. *Type Systems for Polynomial Time Complexity.* Habilitation thesis, Darmstadt. Available as Edinburgh report ECS-LFCS-99-406, 1999.

[20] S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. Manuscript (March 2000) available on www.dcs.qmw.ac.uk/ ˜ ohearn.

[21] S.S. Ishtiaq and D.J. Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[22] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[23] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 1988.

[24] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[25] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic.* Springer-Verlag, 1992.

[26] P. W. O'Hearn. *The Semantics of Non-Interference: A Natural Approach.* Ph.D. thesis, Queen's University, Kingston, Canada, 1990.

[27] P. W. O'Hearn. Linear logic and interference control. In D. H. Pitt et al., editors, *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 74–93, Paris, France, September 1991. Springer-Verlag, Berlin.

[28] P. W. O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3(4):435–465, 1993.

[29] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228(1-2):211–252, October 1999. Preliminary version in [8] and in [32], vol 2.

[30] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, January 2000.

[31] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 217–238. Cambridge University Press, Cambridge, England, 1992.

[32] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages.* Two volumes, Birkhauser, Boston, 1997.

[33] P.W. O'Hearn. Resource interpretations, bunched implications and the $\alpha\lambda$-calculus (preliminary version). In *Typed λ-calculus and Applications*, J-Y Girard editor, LNCS 1581, 1999.

[34] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[35] P.W. O'Hearn, D.J. Pym, and H. Yang. Possible worlds and resources: the semantics of BI. Submitted, 2000.

[36] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages.* Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.

[37] J. Polakow and F. Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *Typed λ-calculus and Applications*, J-Y Girard editor, LNCS 1581., 1999.

[38] D. J. Pym. On bunched predicate logic. In *Proceedings, 14th Annual IEEE Symposium on Logic in Computer Science*, pages 183–192. IEEE Computer Society Press, 1999.

[39] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Monograph in preparation. Portions available on the web at http://www.dcs.qmw.ac.uk/ ˜ pym.

[40] S. Read. *Relevant Logic: A Philosophical Examination of Inference.* Basil Blackwell, 1987.

[41] C. Retoŕe. Pomset logic: A non-commutative extension of classical linear logic. In *Computer Science Logic*, Paderborn, 1995.

[42] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall International, London, 1981.

[43] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland, Amsterdam. Also in [32], vol 1, pages 67-88.

[44] J. C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722, Stresa, Italy, July 1989. Springer-Verlag, Berlin.

[45] J.C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM, New York. Also in [32], vol 1.

[46] P. Ruet and F. Fages. Concurrent constraint programming and noncommutative logic. In *Computer Science Logic'97*, LNCS. Springer-Verlag, 1998.

[47] P. Schroeder-Heister. Structural frameworks, substructural logics and the role of elimination inferences. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 385–403. Cambridge University Press, 1991.

[48] D. S. Scott. Relating theories of the lambda calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450. Academic Press, 1980.

[49] F. Smith, D. Walker, and G. Morrissett. Alias types. ESOP'00.

[50] P. Wadler. There's no substitute for linear logic. In *Proceedings of the 8th Symposium on Mathematical Foundations of Program Semantics*, 1982. Oxford.

[51] P. Wadler. Is there a use for linear logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991. Proceedings of the 1991 Conference.

[52] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* Ph.D. thesis, University of Oxford, 1971.

[53] D. Walker and G. Morrissett. Alias types for recursive data structures. Manuscript, April 2000.