# A λ-Calculus For Resource Separation

Robert Atkey

LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
bob.atkey@ed.ac.uk

**Abstract.** We present a system of typed $\lambda$-calculus, named $\lambda_{\text{sep}}$, which has a semantics based on the relative separation of the resources used by its objects. $\lambda_{\text{sep}}$ is an extension of the affine $\alpha\lambda$-calculus of O'Hearn and Pym that allows finer grained expression of separation constraints. We describe the syntax and typing rules of the system; give a categorical semantics which is coherent, sound and complete; and give a functor-category semantics which treats as distinct the combination of objects and their relationships, showing how the system can represent constraints on resources.

## 1 Introduction

Functional programming languages present the programmer with the neat abstraction that they are always dealing with values. The programmer is lead into the comfortable illusion that these values have no physical presence, that they may be created and discarded as one creates and discards thoughts. However, on a real computer these values take up memory space. Different values may share sections of memory, potentially inhibiting code transformations which speed up functional code by using imperative techniques.

This paper presents a system of typed $\lambda$-calculus, $\lambda_{\text{sep}}$, which attempts to record in the typing judgements the relationships between the resources used by the objects of the system. We adapt the techniques used by other substructural type systems such as the linear $\lambda$-calculus [BBdPH93] and the $\alpha\lambda$-calculus [O'H03], [Pym02] to record and enforce relationships between the objects.

The relationship that we are concerned with here is *separation*; we record the relative separation between resources used by objects. In an application based on memory regions this would record the fact that two objects occupy separate regions of memory. In other applications it could record the fact that two objects are derived from independent statistical data; or that they refer to geographically separate locations. The notion of separation of resources has particular properties that are reflected in the type system; it is a symmetric binary relation on "resources", it is not in general reflexive, nor in general transitive. It has the distributivity property that if an object is separate from a collection of objects then it is also separate from them individually, and vice versa.

To incorporate such relationships in the typing judgements we adopt a strategy, inspired by that of the $\alpha\lambda$-calculus, of introducing new ways of forming contexts. We no longer think of the context as a list or set of type assignments.

Rather, we now regard the context as an undirected graph of type assignments, with edges recording the relationships between members. We also consider subgraphs that have uniform relationships to the rest of the context; representing collections that may be treated as one. The allowable manipulations on contexts, the structural rules, correspond to the properties of separation.

The $\alpha\lambda$-calculus uses two different context formers, represented by the comma and semicolon. Both are binary constructors used to construct contexts from nested "bunches" of type assignments. The two constructors obey different structural rules; the comma disallowing everything except reordering (exchange), and the semicolon allowing the full range of intuitionistic structural rules. The two constructors may then given different semantics; a common one is that the comma combines two objects which use separate resources, the semicolon combines two objects which may use overlapping resources. In this way the system can express relationships between objects. The system presented here, $\lambda_{\text{sep}}$, generalises this situation to $n$ places with attached binary relations expressing separation constraints between members. An example typing judgement is:

$$[1\#2, 1\#3](\mathtt{x} : A, \mathtt{y} : B, \mathtt{z} : B) \vdash \mathtt{e} : C$$

This judgement asserts that $\mathtt{e}$ has type $C$ in a context which ensures that $\mathtt{x}$, of type $A$, is separate from both $\mathtt{y}$ and from $\mathtt{z}$, both of type $B$. The symbol $\#$ indicates separation between positions. By the properties of separation listed above, it is valid to rewrite this context to group the variables $\mathtt{y}$ and $\mathtt{z}$ into a sub-context:

$$[1\#2](\mathtt{x} : A, [](\mathtt{y} : B, \mathtt{z} : B)) \vdash \mathtt{e} : C$$

Since $\mathtt{y}$ and $\mathtt{z}$ are not required to be separate we may identify them. This gives a restricted rule of contraction, allowing one to derive:

$$[1\#2](\mathtt{x} : A, \mathtt{y} : B) \vdash \mathtt{e}[\mathtt{y}/\mathtt{z}] : C$$

If $\mathtt{y}$ and $\mathtt{z}$ had been required to be separate then this inference step would not be valid; in the case of memory regions for example, we would be essentially using a reference to a single region of memory twice and expect the two instances to point to separate regions of memory. Contraction is disallowed as a direct result of the fact that separation is not reflexive (except inthe special case of the "empty" resource).

The complexity of these manipulations on contexts, and their lack of term syntax, motivates the formulation of a categorical semantics which allow us to state clearly the conditions required of models to be coherent.

As a simple example showing how $\lambda_{\text{sep}}$ captures separation constraints, consider a set of primitives for constructing jobs for processing. Each of these jobs consists of two items of data, which are operated on in parallel; they must occupy separate regions of memory (to allow for temporary in-place mutation, for example). This constraint can be captured by a job construction operation:

$$\mathtt{consJob} : [1\#2](D, D) \to J$$

This construction is also possible in the $\alpha\lambda$-calculus.

Now consider a collection of such jobs to be run in sequence. Since they are to be run in sequence it does not matter if any of the jobs overlap in memory. A collection of three such jobs, over four items of data, can be represented as:

$$(\mathtt{consJob}(\mathtt{a},\mathtt{b}),\mathtt{consJob}(\mathtt{b},\mathtt{c}),\mathtt{consJob}(\mathtt{c},\mathtt{d})) : [](J,J,J)$$

Now a context for this term should represent the constraints here as accurately as possible; it should constrain sharing where required, but allow sharing as often as possible. In $\lambda_{\text{sep}}$ the context can be given as:

$$[\mathtt{a}\#\mathtt{b},\mathtt{b}\#\mathtt{c},\mathtt{c}\#\mathtt{d}](\mathtt{a}:D,\mathtt{b}:D,\mathtt{c}:D,\mathtt{d}:D)$$

Here the only constraints are between members of the context whose separation is forced by the construction of the term. In contrast, the affine $\alpha\lambda$-calculus cannot express this configuration. The restriction to binary combinations for expressing separation forces a context where there is extraneous separation enforced. One can get close using a context such as (where ';' represents possible sharing, and ',' no sharing):

$$((\mathtt{a}:D;\mathtt{d}:D),\mathtt{b}:D,\mathtt{c}:D)$$

However, this requires that `a` and `c` be separate, whereas $\lambda_{\text{sep}}$ does not require this. We also claim that the Separation Typing scheme of separating the separation constraints from the body of the context allows for a clearer and more intuitive expression of constraints.

The rest of this paper is laid out as follows. The following section introduces the formal type system and states some simple syntactic results. The following sections consider the semantics of the system. Section 3 gives a categorical semantics. Section 4 gives a resource aware semantics using functor categories, formalising the idea of relationships between resources described in this introduction. Section 5 concludes with a brief discussion of related and further work.

## 2 The Type System

We first define separation relations and an operation of flattening. Separation relations represent the graphs of relationships between objects.

**Definition 1 (Separation Relation).** *A* Separation Relation*, $S$, (of size $n$) is a binary, symmetric, non-reflexive relation on the set $\{1,\ldots,n\}$. $|S|$ denotes the size of a separation relation. For separation relations $S$, $S'$, with $|S| = |S'|$, $S \subseteq S'$ if for all $i,j$ we have that $iSj$ implies $iS'j$.*

*For a list of separation relations $S_1,\ldots,S_n$, define two functions based on the concatenation of this list:*

$$\text{base}(j) = \Sigma_{1\leq i<j}|S_i|, \textit{ the } 0\textit{th index of the } j\textit{th part}$$

$$\text{part}(j) = \textit{the } k \textit{ such that } \text{base}(k) < j \leq \text{base}(k+1)$$

*For a separation relation $S$, of size $n$, we define an operation of flattening on the list:*

$$(i,j) \in S[S_1,\ldots,S_n] \text{ iff } \begin{cases} (i-\text{base}(k), j-\text{base}(k)) \in S_k & \text{if } k = \text{part}(i) = \text{part}(j) \\ (\text{part}(i),\text{part}(j)) \in S & \text{otherwise} \end{cases}$$

We write specific separation relations as lists of related pairs $[r_1,\ldots,r_k]_n$, where each $r_l$ is of the form $i\#j, i<j$, denoting the related pairs of the relation, and $n$ is the size of the relation. We will write $S[S'/i]$ or sometimes just $S[S']$ for $S[[]_1,\ldots,[]_1,S',[]_1,\ldots,[]_1]$ when $S'$ is at position $i$. Observe that the empty separation relation $[]_0$ acts similarly to a unit under flattening. Flattening formalises the distributivity property mentioned above.

The types of the calculus is generated by the following grammar, given some primitive types $X_1, X_2, ...$:

$$A ::= \quad X_i \quad | \quad A_1,\ldots,A_n \xrightarrow{S} B \quad | \quad S(A_1,\ldots,A_n)$$

where $S$ is a separation relation of size $n+1$ for function types and size $n$ for tuple types. The extra place for function types represents the resources used by the function itself. The types then generate the contexts:

$$\Gamma ::= \quad x : A \quad | \quad S(\Gamma_1,\ldots,\Gamma_n)$$

where $S$ is a separation relation, $A$ is a type and no identifier appears more than once. We define $\mathrm{V}(\Gamma)$ to be the list of identifiers in $\Gamma$ built from a depth first, left-to-right traversal.

We also define two forms of equivalence for contexts. The first is normal $\alpha$-equivalence; the second encodes some of the properties of separation, via the flattening operations.

**Definition 2 (Equivalences for Contexts).**

1. *$\Gamma \cong \Gamma'$ when $\Gamma$ and $\Gamma'$ are identical up to renaming of the identifiers.*
2. *$\equiv$ is the smallest equivalence relation which is also a congruence satisfying the following equations:*

$$S(\overrightarrow{\Gamma}, S'(\overrightarrow{\Delta}), \overrightarrow{\Gamma'}) \equiv S[S'](\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Gamma'}) \qquad S(\overrightarrow{\Gamma}) \equiv \sigma S(\sigma\overrightarrow{\Gamma})$$

*where $\sigma$ is a permutation of the set $\{1,\ldots,|S|\}$; $\overrightarrow{\Gamma}$, $\overrightarrow{\Delta}$ and $\overrightarrow{\Gamma'}$ represent sequences of contexts.*

The terms of the calculus are generated from a collection of primitive operations $\Sigma = \{f_1 : A_1 \to B_1, ...\}$:

$$\begin{aligned} e ::= x \quad &| \quad S(e_1,\ldots,e_n) \quad | \quad \lambda^S(x_1,\ldots,x_n).e \quad | \quad f_i e \\ &| \quad \text{let } S(x_1,\ldots,x_n) = e_1 \text{ in } e_2 \quad | \quad e@_S(e_1,\ldots,e_n) \end{aligned}$$

The typing rules are shown in figure 1. They have been split into structural rules and the introduction and elimination rules.

**Structural rules**

$$\frac{}{x : A \vdash x : A}\ (\text{ID}) \qquad \frac{S(\overrightarrow{\Gamma}, [](\Delta, \Delta')) \vdash e : A \qquad \Delta \cong \Delta'}{S(\overrightarrow{\Gamma}, \Delta) \vdash e[\mathrm{V}(\Delta)/\mathrm{V}(\Delta')] : A}\ (\text{CONTR})$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \equiv \Gamma'}{\Gamma' \vdash e : A}\ (\text{EQUIV}) \qquad \frac{S(\overrightarrow{\Gamma}) \vdash e : A \qquad S \subseteq S'}{S'(\overrightarrow{\Gamma}) \vdash e : A}\ (\text{S-WEAKEN})$$

$$\frac{S(\overrightarrow{\Gamma}, []_0()) \vdash e : A}{S(\overrightarrow{\Gamma}, \Delta) \vdash e : A}\ (\text{WEAKEN})$$

**Connective rules**

$$\frac{\Gamma_1 \vdash e_1 : A_1 \qquad \ldots \qquad \Gamma_n \vdash e_n : A_n}{S(\Gamma_1, \ldots, \Gamma_n) \vdash S(e_1, \ldots, e_n) : S(A_1, \ldots, A_n)}\ (S\text{-I})$$

$$\frac{\Gamma \vdash e_1 : S(A_1, \ldots, A_n) \qquad S'(\overrightarrow{\Delta}, S(x_1 : A_1, \ldots, x_n : A_n)) \vdash e_2 : B}{S'(\overrightarrow{\Delta}, \Gamma) \vdash \text{let } S(x_1, \ldots, x_n) = e_1 \text{ in } e_2 : B}\ (S\text{-E})$$

$$\frac{S(\Gamma, x_1 : A_1, \ldots, x_n : A_n) \vdash e : B}{\Gamma \vdash \lambda^S(x_1, \ldots, x_n).e : A_1, \ldots, A_n \xrightarrow{S} B}\ (\to\text{-I})$$

$$\frac{\Gamma \vdash f : A_1, \ldots, A_n \xrightarrow{S} B \qquad \text{for } 1 \leq i \leq n. \quad \Delta_i \vdash a_i : A_i}{S(\Gamma, \Delta_1, \ldots, \Delta_n) \vdash f@_S(a_1, \ldots, a_n) : B}\ (\to\text{-E})$$

$$\frac{\Gamma \vdash e : A \qquad f : A \longrightarrow B \in \Sigma}{\Gamma \vdash fe : B}\ (\text{PRIM})$$

**Fig. 1.** Typing Rules

The judgement is of the form $\Gamma \vdash e : A$. Contexts $\Gamma$ fill a purpose beyond listing a superset of the free variables in the term; they also represent the resources used by the term $e$. That is, the resources occupied by the free variables of a term are also (a superset of) the resources used by the term. This reading provides the informal justification for the rules concerning the type constructors. For example, the rule $S$-I uses the same relationship between the contexts on the left as the terms on the right; so if the free variables of the terms obey the correct separation, then so will the corresponding terms.

The rules $\to$-I and $\to$-E can be understood similarly. In the introduction rule we have the sub-context $\Gamma$ representing the resources used by the function itself, treated as a single block. The function's required separation between the arguments and its own resources is then recorded in its type. This separation is then reconstituted in the elimination rule.

$$\frac{(\Gamma \vdash e = e' : A) \in \Phi}{\Gamma \vdash e = e' : A}\ (\text{AXIOM}) \qquad \frac{\Gamma \vdash e = e' : A \qquad \Gamma \equiv \Gamma'}{\Gamma' \vdash e = e' : A}\ (\text{EQ-EQUIV})$$

$$\frac{S(\overrightarrow{\Gamma}) \vdash e = e' : A \qquad S \subseteq S'}{S'(\overrightarrow{\Gamma}) \vdash e = e' : A}\ (\text{EQ-S-WEAKEN})$$

$$\frac{S(\overrightarrow{\Gamma}, [](\Delta, \Delta')) \vdash e = e' : A \qquad \Delta \cong \Delta'}{S(\overrightarrow{\Gamma}, \Delta) \vdash e[\mathrm{V}(\Delta)/\mathrm{V}(\Delta')] = e'[\mathrm{V}(\Delta)/\mathrm{V}(\Delta')] : A}\ (\text{EQ-CONTR})$$

$$\frac{S(\overrightarrow{\Gamma}, []_0) \vdash e = e' : A}{S(\overrightarrow{\Gamma}, \Delta) \vdash e = e' : A}\ (\text{EQ-WEAKEN})$$

Plus congruence, reflexivity, symmetry and transitivity rules.

**Fig. 2.** Structural Equational Rules

We can provide informal justification for the structural rules by appealing to the properties of separation. The fact that separation obeys a distributivity-like property explains the flattening and unflattening parts of the $\equiv$ equivalence. Permutation is explained by observing that it does not matter in which order we combine resources, it is the separation that matters. This then explains the rule EQUIV, which allows us to replace $\equiv$-equivalent contexts. We can understand S-WEAKEN by observing that once a term is constructed then it does not matter if we enforce more separation than is actually required, since all the required separation will still be present. In the rule CONTR it is possible to duplicate objects as long as we do not require their underlying resources to be separate.

**Lemma 1 (Substitution).** *The following rule is admissible:*

$$\frac{\Gamma(x_1 : A_1, \ldots, x_n : A_n) \vdash e : B \qquad \textit{for all } i \leq n,\ \Delta_i \vdash e_i : A_i}{\Gamma(\Delta_1, \ldots, \Delta_n) \vdash e[e_1/x_1, \ldots, e_n/x_n] : B}$$

Given a collection of axioms, $\Phi$, of the form $\Gamma \vdash e = e' : A$ where $\Gamma \vdash e : A$ and $\Gamma \vdash e' : A$, we define the equational theory of the calculus to be generated by the rules in figures 2 and 3. We define the set of term contexts for the commuting conversions rule (CC) to be:

$$\begin{aligned} C[-] ::= {} & - \quad | \quad S(\ldots, C[-], \ldots) \quad | \quad \lambda^S(x_1, \ldots, x_n).C[-] \quad | \quad fC[-] \\ & | \quad \text{let } S(x_1, \ldots, x_n) = C[-] \text{ in } e_2 \quad | \quad \text{let } S(x_1, \ldots, x_n) = e_1 \text{ in } C[-] \\ & | \quad C[-]@_S(e_1, \ldots, e_n) \quad | \quad e@_S(\ldots, C[-], \ldots) \end{aligned}$$

The equational rules follow the usual form for a typed $\lambda$-calculus with a tensor product, extended to the $n$-place case. We also include two extra rules $\eta$-$\times$ and $\eta$-0 to ensure that the two place product $[](A, B)$ acts as a normal product type and $[]_0()$ as a unit type. To ensure the well-definedness of these rules we must prove that the definition always gives rise to well-typed terms in context.

$$\frac{S(\Gamma, \overrightarrow{\overline{x} : A}) \vdash e : B \qquad \Delta_1 \vdash a_1 : A_1 \qquad \ldots \qquad \Delta_n \vdash a_n : A_n}{S(\Gamma, \overrightarrow{\Delta}) \vdash (\lambda^S(\overrightarrow{\overline{x}}).e)@_S(\overrightarrow{\overline{a}}) = e[\overrightarrow{\overline{a}}/\overrightarrow{\overline{x}}] : B} \ (\beta\text{-}\lambda)$$

$$\frac{\Gamma \vdash e : A_1, \ldots, A_n \xrightarrow{S} B \qquad \overrightarrow{\overline{x}} \notin \Gamma}{\Gamma \vdash (\lambda^S(\overrightarrow{\overline{x}}).e@_S(\overrightarrow{\overline{x}})) = e : A_1, \ldots, A_n \xrightarrow{S} B} \ (\eta\text{-}\lambda)$$

$$\frac{\Gamma \vdash (\overrightarrow{\overline{e}}) : S(\overrightarrow{A}) \qquad S'(\overrightarrow{\Delta}, S(\overrightarrow{\overline{x}} : \overrightarrow{A})) \vdash e' : B}{S'(\overrightarrow{\Delta}, \Gamma) \vdash (\text{let } S(\overrightarrow{\overline{x}}) = (\overrightarrow{\overline{e}}) \text{ in } e') = e'[\overrightarrow{\overline{e}}/\overrightarrow{\overline{x}}] : B} \ (\beta\text{-S})$$

$$\frac{\Gamma \vdash e : S(\overrightarrow{A})}{\Gamma \vdash (\text{let } S(\overrightarrow{\overline{x}}) = e \text{ in } S(\overrightarrow{\overline{x}})) = e : S(\overrightarrow{A})} \ (\eta\text{-S})$$

$$\frac{}{c : []( A, B) \vdash (\text{let } []( a, b) = c \text{ in } a, \text{let } []( a, b) = c \text{ in } b) = c : []( A, B)} \ (\eta\text{-}\times)$$

$$\frac{\Gamma \vdash e : []_0()}{\Gamma \vdash e = []_0() : []_0()} \ (\eta\text{-}0)$$

$$\frac{\Gamma \vdash \text{let } S(\overrightarrow{\overline{x}}) = e_1 \text{ in } C[e_2] : A \qquad C \text{ does not bind or contain } \overrightarrow{\overline{x}}}{\Gamma \vdash (\text{let } S(\overrightarrow{\overline{x}}) = e_1 \text{ in } C[e_2]) = (C[\text{let } S(\overrightarrow{\overline{x}}) = e_1 \text{ in } e_2]) : A} \ (\text{CC})$$

**Fig. 3.** Connective Equational Rules

**Lemma 2.** *If $\Gamma \vdash e = e' : A$ is derivable, then $\Gamma \vdash e : A$ and $\Gamma \vdash e' : A$.*

### 2.1 Relation to Other Calculi

We define translations $(-)^*$ from other systems of typed $\lambda$-calculus into $\lambda_{\text{sep}}$:

**Simply-Typed** On contexts: $(x_1 : A_1, \ldots, x_n : A_n)^* = []( x_1 : A_1^*, \ldots, x_n : A_n^*)$. On types $(-)^*$ replaces $\times$ with $[](-,-)$ and $\to$ with $\xrightarrow{[]}$.

**Affine !-free linear** On contexts: $(x_1 : A_1, \ldots, x_n : A_n)^* = S(x_1 : A_1^*, \ldots, x_n : A_n^*)$, where $S$ has all possible separation. On types $(-)^*$ replaces $\otimes$ with $[1\#2](-,-)$ and $\multimap$ with $\xrightarrow{[1\#2]}$.

**Affine $\alpha\lambda$-calculus** On contexts: $(x : A)^* = x : A$; $(\Gamma, \Delta)^* = [1\#2](\Gamma^*, \Delta^*)$ and $(\Gamma; \Delta)^* = []( \Gamma^*, \Delta^*)$. On types $(-)^*$ replaces $*$ with $[1\#2](-,-)$;$\times$ with $[](-,-)$; $-\!\!*$ with $\xrightarrow{[1\#2]}$; and $\to$ with $\xrightarrow{[]}$.

**Lemma 3.** *For the above three translations $(-)^*$ (with the evident actions on terms) from typed $\lambda$-calculi, it is the case that if $\Gamma \vdash e_1 = e_2 : A$ in the source system then $\Gamma^* \vdash e_1^* = e_2^* : A^*$ in $\lambda_{\text{sep}}$.*

## 3 Categorical Semantics

In this section we describe the categorical structure required to model the syntax of $\lambda_{\text{sep}}$. We require a generalisation of symmetric monoidal operations to model the contexts and tuple types. As with monoidal categories, we prove a coherence result for these categories. The rest of the section then deals with the interpretation of $\lambda_{\text{sep}}$ in categories with the required structure.

**Definition 3 (Separation Category).** *A separation category is a category $\mathcal{C}$ with a family of* separation functors $S : \mathcal{C}^n \to \mathcal{C}$*, for each separation relation $S$ of size $n$, such that $[]_1$ is the identity functor and there exists a family of natural isomorphisms:*

$$\alpha_{S(A,S'(B),C)} : S(\overrightarrow{A}, S'(\overrightarrow{B}), \overrightarrow{C}) \to S[S'](\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C})$$

*These natural isomorphisms must satisfy the following commutative diagrams. First, it does not matter in which order we flatten sibling applications:*

$$\begin{array}{ccc} S(\overrightarrow{A}, S'(\overrightarrow{B}), \overrightarrow{C}, S''(\overrightarrow{D}), \overrightarrow{E}) & \xrightarrow{\alpha} & S[S'](\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}, S''(\overrightarrow{D}), \overrightarrow{E}) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\ S[S''](\overrightarrow{A}, S'(\overrightarrow{B}), \overrightarrow{C}, \overrightarrow{D}, \overrightarrow{E}) & \xrightarrow{\alpha} & S[S', S''](\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}, \overrightarrow{D}, \overrightarrow{E}) \end{array}$$

*Second, it does not matter in which order we flatten nested applications:*

$$\begin{array}{ccc} S(\overrightarrow{A}, S'(\overrightarrow{B}, S''(\overrightarrow{C}), \overrightarrow{D}), \overrightarrow{E}) & \xrightarrow{\alpha} & S[S'](\overrightarrow{A}, \overrightarrow{B}, S''(\overrightarrow{C}), \overrightarrow{D}, \overrightarrow{E}) \\ \downarrow{\scriptstyle S(id^{|A|}, \alpha, id^{|E|})} & & \downarrow{\scriptstyle \alpha} \\ S(\overrightarrow{A}, S'[S''](\overrightarrow{B}, \overrightarrow{C}, \overrightarrow{D}), \overrightarrow{E}) & \xrightarrow{\alpha} & S[S'[S'']](\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}, \overrightarrow{D}, \overrightarrow{E}) \end{array}$$

*A* Closed Separation Category *is a separation category in which each functor $S(-, A_1, \ldots, A_n) : \mathcal{C} \to \mathcal{C}$ has a specified right adjoint $[A_1, \ldots, A_n \xrightarrow{S} -] : \mathcal{C} \to \mathcal{C}$.*

Coherence for a Separation Category ensures that all paths with the same start and end objects constructed from the $\alpha$s and their combinations are equal. Following Mac Lane [Mac98], this result is stated in terms of *words*:

**Definition 4 (Words).** *For every separation relation $S$, there is a set of* words *$w^S$, defined by the grammar:*

$$w^{[]_0} ::= []_0() \qquad\qquad w^{[]_1} ::= []_1(-)$$

$$w^S ::= S'(w^{S_1}, ..., w^{S_n}), \text{ where } S = S'[S_1, ..., S_n]$$

*For each word $u \in w^S$ define a functor $[u] : \mathcal{C}^{|S|} \to \mathcal{C}$, defined by induction on the structure of $w$ as:*

$$[[]_0()] = \star \mapsto []_0 : \mathbf{1} \to \mathcal{C} \qquad [[]_1(-)] = x \mapsto x : \mathcal{C} \to \mathcal{C}$$

$$[S(u_1, \ldots, u_n)] = S \circ ([u_1] \times \ldots \times [u_n]) : \mathcal{C}^{|S|} \to \mathcal{C}$$

*with the evident action on arrows.*

Note that words are grouped by the overall separation relation they possess; the internal structure of the word, corresponding to the different ways of forming the same context in the type system, is irrelevant. Therefore, there is a canonical *fully flattened* word, consisting of just a top-level application of a separation functor with only applications of $[]_1(-)$ beneath it. This is analogous to regarding iterated monoidal products abstractly as a natural number, and is the crux of the proof of coherence. The statement follows that of Mac Lane for monodial categories:

**Theorem 1.** *If $\mathcal{C}$ is a separation category then for any pair of words, $u, v \in w^S$, there is a unique natural isomorphism*

$$\mathrm{can}(u, v) : [u] \Rightarrow [v] : \mathcal{C}^{|S|} \to \mathcal{C}$$

*called the canonical map from $[u]$ to $[v]$, such that the identity arrow $[]_0 \to []_0$ is canonical, the identity $\mathrm{id}_{\mathcal{C}} : []_1 \Rightarrow []_1$ is canonical, all $\alpha$ and $\alpha^{-1}$ are canonical and the composition and $S$ separation combination of canonical arrows are canonical.*

*Proof.* Adaptation of the proof of coherence for monoidal categories as presented by Mac Lane [Mac98].

Given the basic definition of a Separation Category, we now add natural transformations to model the permutation and constraint weakening (S-WEAKEN) of contexts. Permutation is viewed as a typed group of transitions on separation relations, and weakening as a typed monoid of such transitions. We will consider them together.

First, we define the action of a permutation on a separation relation $S$ as giving a new separation relation such that $\sigma S = \{(i, j) : \sigma^{-1}(i) S \sigma^{-1}(j)\}$.

For each separation relation $S$ and each permutation on the set $\{1, \ldots, |S|\}$, $\sigma$, we define a *permutation transition*, $\sigma_S$, between $S$ and $\sigma S$. These are composable, invertible and have an identity. Each such transition has an action on indices, defined as $\sigma_S(i) = \sigma(i)$. Likewise, for each $S$ and $S'$ such that $S' \subseteq S$ we define a *weakening transition*, labelled $[S, S']$. These are composable and have an identity. Each such transition has an action on indices, defined as $[S, S'](i) = i$.

We now state the structure required as two collections of natural transformations; one parameterised by permutation transitions, $\gamma[\sigma_S] : S(\overrightarrow{I}) \to \sigma S(\sigma \overrightarrow{I})$; the other by reverse inclusions, $\zeta[S, S'] : S(\overrightarrow{I}) \to S'(\overrightarrow{I})$. The original actions obey some algebraic laws, and we require that they are honoured by the natural transformations:

$$\gamma[\sigma'_{\sigma S}] \circ \gamma[\sigma_S] = \gamma[\sigma'_{\sigma S} \circ \sigma_S] \qquad \zeta[S', S''] \circ \zeta[S, S'] = \zeta[S, S'']$$
$$\gamma[id_S] = id \qquad \zeta[S, S] = id$$
$$\gamma[\sigma_S]^{-1} = \gamma[\sigma^{-1}_{\sigma S}]$$

In order to coherently model the syntax, we also require that these natural transformations commute with the flattening and unflattening operations above, and also with each other. This corresponds to the commuting of the syntax-free structural rules in typing derivations. To state the commutativity with flattening, we must define the effect of flattening on both permutation and weakening transitions. This takes the form of: (a) substituting a transition into a separation relation; and (b) substituting a separation relation into a transition. These deal with the cases when, prior to flattening, the transition is acting on the inner or outer separation relation respectively.

**Definition 5 (Substitution of transitions).** *Given a transition $\Box : S_1 \to S_2$, with action $\Box(-)$, define:*

- *$S[\Box/i] : S[S_1/i] \to S[S_2/i]$, substituting $\Box$ into position $i$ of $S$. Set:*

$$S[\Box/i](k) = \begin{cases} \Box(k - \mathrm{base}(i)) + \mathrm{base}(i) & \text{if } \mathrm{part}(k) = i \\ k & \text{otherwise} \end{cases}$$

- *$\Box[S/i] : S_1[S/i] \to S_2[S/\Box(i)]$, substituting $S$ into position $i$ of $\Box$. Set:*

$$\Box[S/i](k) = k - \mathrm{base}(\mathrm{part}(k)) + \mathrm{base}(\Box(\mathrm{part}(k)))$$

The first operation defines a small area of the new separation relations where the transition has been substituted into and only applies the operation within that area; the second operation treats the newly flattened part of the separation relation as a block with respect to the original transition, applying the transition as if no substitution had taken place. Note that a permutation/weakening transition that has been through either of these operations is still a permutation or weakening transition. We can now state the commutativity requirement with flattening, where $\tau$ is $\gamma$ or $\zeta$, and $\Box$ is a transition of the appropriate type:

$$\begin{array}{ccc} S(\overrightarrow{\Gamma}, S'(\overrightarrow{\Delta}), \overrightarrow{\Gamma'}) & \xrightarrow{S(\ldots,\tau[\Box],\ldots)} & S(\overrightarrow{\Gamma}, \Box S'(\Box\overrightarrow{\Delta}), \overrightarrow{\Gamma'}) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\ S[S'/i](\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Gamma'}) & \xrightarrow{\tau[S[\Box/i]]} & S[\Box S'/i](\overrightarrow{\Gamma}, \Box\overrightarrow{\Delta}, \overrightarrow{\Gamma'}) \end{array}$$

$$\begin{array}{ccc} S(\overrightarrow{\Gamma}, S'(\overrightarrow{\Delta}), \overrightarrow{\Gamma'}) & \xrightarrow{\tau[\Box]} & \Box S(\Box(\overrightarrow{\Gamma}, S'(\overrightarrow{\Delta}), \overrightarrow{\Gamma'})) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\ S[S'/i](\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Gamma'}) & \xrightarrow{\tau[\Box[S'/i]]} & \Box S[S'/\Box(i)](\Box[S'/i](\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Gamma'})) \end{array}$$

We also require that permutations and weakenings commute with each other:

$$\begin{array}{ccc} S(A_1,\ldots,A_n) & \xrightarrow{\gamma[\sigma_S]} & \sigma S(A_{\sigma(1)},\ldots,A_{\sigma(n)}) \\ \downarrow{\scriptstyle \zeta[S,S']} & & \downarrow{\scriptstyle \zeta[\sigma S,\sigma S']} \\ S'(A_1,\ldots,A_n) & \xrightarrow{\gamma[\sigma_{S'}]} & \sigma S'(A_{\sigma(1)},\ldots,A_{\sigma(n)}) \end{array}$$

With these requirements, it is possible to prove an extended coherence result for separation categories with permutation and weakening. The proof relies on the fact that we can use the group/monoid and commutativity properties to rewrite any path containing flattenings, unflattenings, permutations, weakenings and their combinations into a canonical form. We do not formally state the coherence property here, but we will state it in an extended form below for the main coherence result for the interpretation of typing derivations.

The final piece of structure we require is for modelling the WEAKEN and CONTR typing rules. Rule WEAKEN just requires that the object $[]_0()$ is the terminal object with $!_A$ denoting the unique arrow from $A$ to $[]_0$. Rule CONTR is more complicated; before introducing the structure required, we define *Separation Category Functors* and natural transformations between them, by analogy with monoidal functors:

**Definition 6 (Separation Category Functor).** *A* Separation Category Functor *consists of a functor $F : \mathcal{C} \to \mathcal{C}'$ between separation categories $\mathcal{C}$ and $\mathcal{C}'$, and a family of natural transformations:*

$$f_{S(A_1,\ldots,A_n)} : S(FA_1,\ldots,FA_n) \to F(S(A_1,\ldots,A_n))$$

*such that the following diagram commutes:*

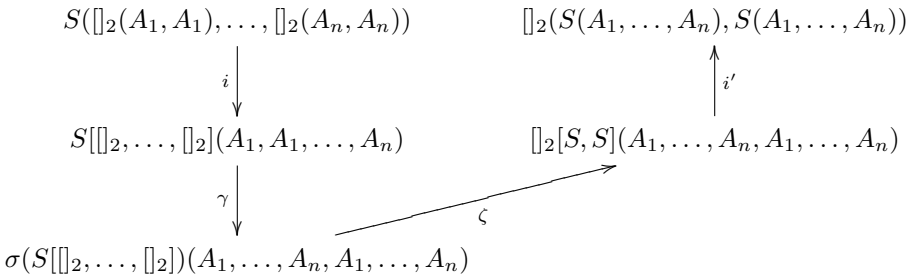
$$\begin{array}{ccc} S(FA_1,\ldots,S'(FB_1,\ldots),\ldots) & \xrightarrow{\alpha} & S[S'/i](FA_1,\ldots,FB_1,\ldots) \\ \downarrow{\scriptstyle S(\ldots,f,\ldots)} & & \\ S(FA_1,\ldots,F(S'(B_1,\ldots)),\ldots) & & \downarrow{\scriptstyle f} \\ \downarrow{\scriptstyle f} & & \\ F(S(A_1,\ldots,S'(B_1,\ldots),\ldots)) & \xrightarrow{F(\alpha)} & F(S[S'/i](A_1,\ldots,B_1,\ldots)) \end{array}$$

*If $\mathcal{C}$ and $\mathcal{C}'$ support permutation (weakening) transitions, then the permutations (weakenings) should be preserved by $F$ via $f$ in the evident way.*

*A* Separation Natural Transformation *is a natural transformation $\tau$ between two Separation Category Functors $(F, f)$ and $(G, g)$ such that all instances of the following diagram commute:*

$$\begin{array}{ccc} S(FA_1,\ldots,FA_n) & \xrightarrow{f} & F(S(A_1,\ldots,A_n)) \\ \downarrow{\scriptstyle S(\tau_{A_1},\ldots,\tau_{A_n})} & & \downarrow{\scriptstyle \tau_{S(A_1,\ldots,A_n)}} \\ S(GA_1,\ldots,GA_n) & \xrightarrow{g} & G(S(A_1,\ldots,A_n)) \end{array}$$

To model CONTR we need a separation natural transformation $dup_A : A \to []( A, A)$, where the functor $A \mapsto []( A, A)$ is made into a separation category functor by the following composite giving the required family of transformations:




where $i$ and $i'$ are the canonical maps between the applications of separation functors, consisting of combinations of flattening arrows. This ensures that we may coherently model the commuting of CONTR applications with other structural rules in the syntax.

To coherently and soundly model the syntax we require some further conditions. Firstly, for each object $A$ we require that the triple $(A, dup_A, !_A)$ is a commutative comonoid. Secondly, we require an additional equation involving $!_A$ and $dup_A$, to soundly model the $\eta$-$\times$ rule of the syntax:

$$[](\alpha \circ [](id_A, !_B), \alpha \circ [](!_A, id_B)) \circ dup_{[](A,B)} = id_{[](A,B)}$$

We now pause to remark that the definition of separation category also gives rise to two monoidal structures on the category, which are also symmetric monoidal if the category has permutation. These are constructed from the two possible 2-place separation functors $[](-,-)$ and $[1\#2](-,-)$, with identity in both cases given by the 0-place separation functor $[]_0()$. Associativity and identity natural transformations are given by the evident combination of flattenings and unflattenings; by the coherence theorem above they satisfy the coherence conditions for a monoidal category. Symmetry is given by the permutation natural transformations, and the coherence diagrams hold again. Given the extra structure described above for the *dup* natural transformation it is easy to see that $[](-,-)$ is the categorical product. Therefore a category with the above structure is also a Doubly Closed Category (DCC) and hence a model of the $\alpha\lambda$-calculus [Pym02].

### 3.1 Interpretation of the Typing Rules

Given the above definitions and an interpretation for the primitive types and operations of the calculus, we have enough categorical structure to model $\lambda_{\text{sep}}$. The interpretation of the typing rules, which we elide for space reasons, is straightforward. We follow the standard approach for modelling linear $\lambda$-calculi in symmetric monoidal closed categories, using the separation functors to model the context and tuple types, and the closed structure to model the function types. This interpretation defines a map $[\![-]\!]$ from type derivations to arrows in $\mathcal{C}$.

Due to the syntax-free structural rules in the type system there is the possibility of having two judgements $\Gamma \vdash e : A$ with different derivations, and hence potentially different meanings in $\mathcal{C}$. However, as a result of the coherence conditions this is not possible:

**Theorem 2 (Coherence).** *If $\pi_1$ and $\pi_2$ are two derivations of the judgement $\Gamma \vdash e : A$ then $[\![\pi_1]\!] = [\![\pi_2]\!]$.*

*Proof.* Adaptation of coherence of SCIR by O'Hearn *et. al.* [OPTT99].

Now, given such a category, along with the requirement that for each axiom $\Gamma \vdash e = e' : A$ the (unique) arrows $[\![e]\!], [\![e']\!] : [\![\Gamma]\!] \to [\![A]\!]$ are such that $[\![e]\!] = [\![e']\!]$, we have a soundness and completeness result:

**Theorem 3 (Soundness and Completeness).** *$\Gamma \vdash e_1 = e_2 : A$ if and only if in all categorical models $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.*

Since any cartesian closed category also enjoys the correct structure then we also have this extension to Lemma 3.

**Lemma 4.** *Using the translation above, we have: $\Gamma \vdash e_1 = e_2 : A$ in the simply-typed $\lambda$-calculus if and only if $\Gamma^* \vdash e_1^* = e_2 * : A$ in $\lambda_{\text{sep}}$.*

## 4 Functor Category Semantics

We now give an example of a family of categories with the structure described above, using functor categories. This semantics gives a direct demonstration of the resource awareness of the typing rules.

We require a small category, $\mathcal{R}$, with finite co-products $(r_1 + \ldots + r_n)$. The objects of $\mathcal{R}$ are intended to model resources, and the co-product the combination of two resources to make a new resource. To model separation constraints we require a bi-functor $\# : \mathcal{R}^{op} \times \mathcal{R}^{op} \to \mathbf{2}$, where $\mathbf{2}$ is the category with objects $t$ and $f$ and a single non-identity arrow $f \to t$. The functor $\#$ must take co-products in $\mathcal{R}$ to products in $\mathbf{2}$ (i.e. it preserves products on $\mathcal{R}^{op}$). Also, we require that the bi-functor is symmetric: $r\#r' = r'\#r$ (and also for arrows). The intention is that $r\#r' = t$ when $r$ and $r'$ are separate resources.

We now model each type as a functor from $\mathcal{R}$ to **Set**, so that each type is a collection of sets indexed by the resources available. Using the relation $\#$, we

can give $\mathbf{Set}^{\mathcal{R}}$ the separation category structure described in the last section. The separation functors are given by the following definition, for $S(A_1, \ldots, A_n)$ at resource $r$:

$$\begin{array}{l}\{(a_1, \ldots, a_n) \in A_1 r \times \ldots \times A_n r : \\ \quad \exists f_1 : r_1 \to r, \ldots, f_n : r_n \to r, a'_1 \in A_1 r_1, \ldots, a'_n \in A_n r_n \text{ s.t.} \\ \quad\quad \forall i. a_i = A_i f_i a'_i \text{ and } \forall i, j. iSj \Rightarrow r_i \# r_j = t\}\end{array}$$

The resources used by the members of the tuple must be related in such a way that satisfies the separation relation. The action of these functors on arrows $f : r \to r'$ and natural transformations $h_i : A_i \Rightarrow B_i$ is defined point-wise.

The function space construction, $[A_1, \ldots, A_n \xrightarrow{S} B]$, is defined at resource $r_0$ as the family of functions:

$$\Pi_{(r_1, \ldots, r_n) \in R}. A_1 r_1 \times \ldots \times A_n r_n \Rightarrow B(r_0 + r_1 + \ldots + r_n)$$

where $R = \{(r_1, \ldots, r_n) \in \mathrm{Ob}(\mathcal{R})^n : \forall i, j \in \{0, \ldots, n\}. iSj \Rightarrow r_i \# r_j\}$. Note the extra 0th place for the function's resources.

**Theorem 4.** *The above definitions give a closed Separation Category structure on* $\mathbf{Set}^{\mathcal{R}}$ *with permutation, weakening and a duplication natural transformation.*

*Proof.* The flattening, unflattening, permutation and weakening maps are given by the evident manipulations on tuples. That they preserve the required separation constraints follows from the conditions on the functor #. The fact that the function space forms the right adjoint to the separation functors is shown by calculation of the required properties. Duplication is given by the obvious map duplicating the elements.

An example of the above construction is given by the memory region example from the introduction. Starting from some set of memory locations $L$, we take our category of resources to have objects from the powerset of $L$ and inclusions as arrows. The relation is then defined as $r_1 \# r_2 = t \Leftrightarrow r_1 \cap r_2 = \emptyset$. Hence, two regions of memory are separate if they do not share any memory locations. It is easy to see that this relation obeys the required properties, and so $\mathbf{Set}^{\mathcal{P}(L)}$ is a separation category.

An interesting point about this semantics is the distinction between the combination of resources and the separation of resources, represented by the coproduct and the functors $-\#-$ respectively. This is in contrast to the common models of Bunched Implications and the $\alpha\lambda$-calculus, such as those given by Pym, O'Hearn and Yang [POY03], where no explicit distinction is made. The partial monoid and Grothendieck possible world semantics do admit a model for BI based on this distinction, but not explicitly.

The fact that this semantics is based on a binary relation is reminiscent of possible world semantics of modal logics, and there could possibly be connections between generalizations of this semantics to non-symmetric relations and modal logics. In particular the natural deduction presentations with zoned contexts such as those of Pfenning and Davies [PD01].

## 5 Conclusions and Further Work

We have presented the $\lambda_{\text{sep}}$ system along with a categorical semantics and have shown how the judgements of the calculus can be interpreted as making statements about resources. The system as it stands, while offering perhaps small increase in flexibility over the $\alpha\lambda$ system, does suggest interesting paths for extension. These include resource insensitive types and more general classes of relations than separation, such as non-symmetric relations. The exact relationship to the affine $\alpha\lambda$-calculus also requires more investigation, in particular whether $\lambda_{\text{sep}}$ is a conservative extension. Also, questions of completeness for the functor category semantics need to be studied.

There have been many other systems of typed $\lambda$-calculus and related logics for resource constraints. Apart from Bunched Implications and the $\alpha\lambda$-calculus which have already been mentioned [OP99], [O'H03]; we also mention Reynolds' Syntactic Control of Interference (SCI) [Rey78], which disallows almost all contraction, preserving the invariant that distinct identifiers refer to distinct resources. Various spatial logics have been developed which have interpretations based on resources, such as the Ambient Logic of Cardelli and Gordon [CG03].

## References


[BBdPH93] N. Benton, G. Bierman, V. de Paiva, and H. Hyland. A term calculus for intuitionistic linear logic. In *Proceedings of International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag, 1993. LNCS 664.

[CG03] Luca Cardelli and Andrew D. Gordon. Ambient logic. WWW: http://www.luca.demon.co.uk/, 2003.

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 2nd edition, 1998.

[O'H03] P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

[OP99] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–243, 1999.

[OPTT99] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228:211–252, 1999.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[POY03] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible Worlds and Resources: The Semantics of BI. To appear in Theoretical Computer Science. WWW: http://www.cs.bath.ac.uk/~pym/resource.ps, 2003.

[Pym02] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 39–46. ACM Press, 1978.