A λ-Calculus for Resource Separation

Robert Atkey

LFCS, School of Informatics, University of Edinburgh
Mayfield Rd, Edinburgh EH9 3JZ, UK
`bob.atkey@ed.ac.uk`

Abstract. We present a typed λ-calculus for recording resource separation constraints between terms. The calculus contains a novel way of manipulating nested multi-place contexts augmented with constraints, allowing a concise presentation of the typing rules. It is an extension of the affine $\alpha\lambda$-calculus. We give a semantics based on sets indexed by resources, and show how the calculus may be extended to handle non-symmetric relations with application to allowable information flow. Finally, we mention some future directions and questions we have about the calculus.

1 Introduction

Functional programming languages present the programmer with the neat abstraction that they are dealing with pure 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 occupy memory space. Different values may share sections of memory, potentially inhibiting techniques which speed up functional code by using imperative techniques [5, 7], see also the benign sharing condition in [6]. The same information can often be useful for the programmer in imperative languages for reasoning about aliasing [15, 10].

This paper presents a system of typed λ-calculus, λ_{sep}, which attempts to record in the typing judgements the separation between the resources used by the values of the system. We adapt the techniques used by other substructural type systems such as the $\alpha\lambda$-calculus [9, 12] and Reynolds' SCI [15, 16] to record and enforce the separation required between the values of the system.

The recording of separation constraints allows us to give the basic operations of the language types that enforce the constraints. For example, consider an application where we construct jobs to be run on two items of data in parallel, but we require that the items of data occupy separate regions of memory to allow for temporarily destructive operations:

$$\texttt{mkJob} : [1\#2](D, D) \longrightarrow J$$

The notation [1#2] expresses that `mkJob` takes two arguments which must occupy separate regions of memory. We call such specifications *separation relations.*

To incorporate separation relations 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 required separation between members. To allow the piecemeal construction of larger contexts, we also consider sub-contexts that have a uniform relationship to the rest of the context. The allowable manipulations on contexts, the structural rules, correspond to separation constraint preserving manipulations. The context also represents the union of all the resources occupied by the term.

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, 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 contexts which use separate resources, the semicolon combines two contexts which may use overlapping resources. In this way, the system can express relationships between objects. The system presented here, λ_{sep}, generalises this situation to n places with attached binary relations expressing separation constraints between members. An example typing judgement is:

$$[1\#2, 1\#3](a : D, b : D, c : D) \vdash (\mathtt{mkJob}(a, b), \mathtt{mkJob}(a, c)) : (J, J) \qquad (1)$$

The separation between a and b and a and c recorded in the context is induced by the separation required by `mkJob`. This separation configuration can be expressed in the $\alpha\lambda$-calculus as $(a : D, (b : D; c : D))$.

Our extension to n-place separation relations on contexts, rather than binary bunches of contexts, is justified by looking at the possible graphs of separation between members expressible by both schemes. Obviously, any bunched context may be translated into an n-place context by writing out all the induced separations in full. Conversely, however, binary context formers only allow the expression of the series-parallel graphs; graphs that are constructed recursively from the one point graph by two operations of either complete non-separation, or complete separation. The following fact [2, 18] shows that this does not cover all graphs:

Fact 1 (SP-graph Characterisation) *A graph is series-parallel iff its restriction to any four vertices is not equal to* $(\{a, b, c, d\}, \{(a, b), (b, c), (c, d)\})$.

To see an example of how this separation relation may occur, consider the context required for typing the construction of 3 jobs in sequence over 4 items of data:

$$[1\#2, 2\#3, 3\#4](a : D, b : D, c : D, d : D) \vdash (\mathtt{mkJob}(a, b), \mathtt{mkJob}(b, c), \mathtt{mkJob}(c, d))$$

Another example of this configuration is shown in section 4.

As mentioned above, the structural rules of the calculus correspond to separation preserving manipulations of the context. To allow concise presentation of complex separation relations we permit nested contexts. For example, judgement (1) may also be written as:

$$[1\#2](a : D, [](b : D, c : D)) \vdash (\texttt{mkJob}(a, b), \texttt{mkJob}(a, c)) : (J, J) \tag{2}$$

Since a was required to be separate from both b and c, we may group b and c into a nested context with uniform separation from a. The equivalence of these two contexts is justified by considering that an object is separate from a group of objects if and only if it is separate from them all individually. This operation is encoded in the type system by the formalisation of substitution of separation relations into separation relations.

To preserve the correctness of separation relations when variables are used multiple times in a term we restrict the use of the rule of contraction. We only allow contraction between two variables when they are not required to be separate. Hence, given judgement (2) we may infer:

$$[1\#2](a : D, b : D) \vdash (\texttt{mkJob}(a, b), \texttt{mkJob}(a, b)) : (J, J) \tag{3}$$

This step is justified by thinking of contraction as semantically inducing the duplication of references to resources. Obviously two references to the same resource cannot be considered separate, so the typing judgement may not rely on them being so.

2 The Type System

Separation Relations. We first introduce separation relations; these represent the relationships of relative separation between objects.

Definition 1. *A* separation relation *of size n is a binary, irreflexive, symmetric relation on the set $\{0, \ldots, n-1\}$.*

We write $|\mathrm{S}|$ for the size of a separation relation. Define the relation $\mathrm{S} \subseteq \mathrm{S}'$ between two separation relations to hold if and only if $|\mathrm{S}| = |\mathrm{S}'|$ and, for all x and y, $x\mathrm{S}y$ implies $x\mathrm{S}'y$. We write specific relationship specifiers as lists of related pairs $[r_1, \ldots, r_k]_n$, where each r_i is of the form $x\#y$, denoting the related pairs of the relation, and n is the size.

Definition 2 (Relation Substitution). *For separation relations* S *and* S′*, with sizes n and n' respectively, define the operation of substitution* $\mathrm{S}\{\mathrm{S}'/i\}$*, where $0 \leq i < n$, as:*

$$(x, y) \in \mathrm{S}\{\mathrm{S}'/i\} \textit{ iff } \begin{cases} (x - i, y - i) \in \mathrm{S}' & \textit{if } \mathrm{norm}^i_{n'}(x) = \mathrm{norm}^i_{n'}(y) = i \\ (\mathrm{norm}^i_{n'}(x), \mathrm{norm}^i_{n'}(y)) \in \mathrm{S} & \textit{otherwise} \end{cases}$$

where:

$$\mathrm{norm}^i_{n'}(x) = \begin{cases} x & \textit{if } x < i \\ i & \textit{if } i \leq x < i + n' \\ x - n' + 1 & \textit{if } x \geq i + n' \end{cases}$$

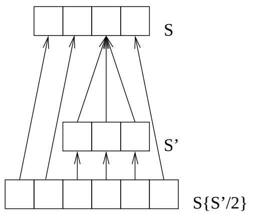

Fig. 1. Substitution of Separation Relations

Substitution of relations may be visualised as in figure 1. For a pair of positions x and y in $\mathrm{S}\{\mathrm{S}'/i\}$, either both x and y are in the range of S', or at least one of them is in the range of S. In the first case we use the relation S'; otherwise, we map the positions back to S (up the diagram) and use S to judge whether x and y are related. The function norm_n^i does the mapping back to S. Note that if a member of S is related to any member of S' then it is related to all of them.

Lemma 1. *The following properties hold of substitution:*

1. $S\{S'/i\}$ *is a separation relation;*
2. $S\{[]_1/i\} = S$*;*
3. $S\{S_1/i\}\{S_2/j + n_1 - 1\} = S\{S_2/j\}\{S_1/i\}$*, where* $i < j$ *and* $|S_1| = n_1$*;*
4. $S\{S_1\{S_2/j\}/i\} = S\{S_1/i\}\{S_2/i + j\}$.

Properties 3 and 4 ensure that if we perform two non-interfering substitutions in two different orders then we always finish in the same state. This is useful for reasoning about the allowable manipulations of contexts, since a nested context may always be substituted out to a single flat context. The 0-place separation relation $[]_0$ acts as a unit under substitution, since substituting $[]_0$ into a position i effectively causes that position to be removed.

Types and Contexts. The types of the calculus are generated by the following grammar, given a set of primitive types $\mathcal{T}$:

$$A, B \quad ::= \quad X \in \mathcal{T} \quad | \quad A_1, \dots, A_n \xrightarrow{\mathrm{S}} B \quad | \quad \mathrm{S}(A_1, \dots, A_n)$$

where S is a separation relation of size $n + 1$ for function types and size n for tuple types. The extra place in the function types represents the resources used by the body of the function. The types then generate the contexts, using the nested structure described in the introduction:

$$\Gamma, \Delta \quad ::= \quad x : A \quad | \quad \mathrm{S}(\Gamma_1, \dots, \Gamma_n)$$

where S is a separation relation of size n, A is a type and no variable x appears more than once in a context. We identify the context $x : A$ with the one-place context $[]_1(x : A)$. We define $\mathrm{v}(\Gamma)$ to be the list of variables in Γ built from a depth-first, left-to-right traversal. The notation $\Gamma(-)$ represents a context with a

$$\frac{}{x : A \vdash x : A}\ (\textsc{Id}) \qquad \frac{\Gamma(\Delta') \vdash e : A \qquad \Delta \overset{\phi}{\Rightarrow} \Delta'}{\Gamma(\Delta) \vdash \phi(e) : A}\ (\textsc{Struct})$$

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

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

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

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

$$\frac{\Gamma \vdash e : A \qquad f : A \longrightarrow B \in \mathcal{F}}{\Gamma \vdash fe : B}\ (\textsc{Prim})$$

Fig. 2. Typing Rules

"hole" at some position in the tree, and $\Gamma(\Delta)$ is the context with Δ substituted for the hole. This will be used for selecting nested sub-contexts for particular attention. We write $\Gamma \equiv_\alpha \Gamma'$ to denote equivalence of contexts up to renaming of variables.

A context Γ determines a separation relation, S_Γ, on all the variables it contains, by substituting out all the nested separation relations. It is unique by Lemma 1.

Structural Rules. We give the structural rules of the calculus in a uniform fashion, collected in the typing rule Struct in figure 2. This rule is parameterised by labelled transitions $\Delta \overset{\phi}{\Rightarrow} \Delta'$, where Δ and Δ' are contexts and ϕ is an operation on terms. The allowable transitions are shown in figure 3.

Transition	Effect on the context	$\phi(e)$
Flatten	$\mathrm{S}(\overrightarrow{\Gamma}, \mathrm{S}'(\overrightarrow{\Delta}), \overrightarrow{\Gamma'}) \Leftrightarrow \mathrm{S}\{\mathrm{S}'/\mathrm{i}\}(\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Gamma'})$	e
S-Weak ($\mathrm{S}' \subseteq \mathrm{S}$)	$\mathrm{S}(\overrightarrow{\Gamma}) \Rightarrow \mathrm{S}'(\overrightarrow{\Gamma})$	e
Contr ($\Gamma \equiv_\alpha \Gamma'$)	$\Gamma \Rightarrow []_2(\Gamma, \Gamma')$	$e[\mathrm{v}(\Gamma)/\mathrm{v}(\Gamma')]$
Weak	$\Gamma \Rightarrow []_0()$	e
Perm	$\mathrm{S}(\overrightarrow{\Gamma}) \Leftrightarrow \sigma\mathrm{S}(\sigma\overrightarrow{\Gamma})$	e

Fig. 3. Structural Transitions

We can justify the structural rules by appeal to the properties of separation. The FLATTEN and CONTR transitions were justified in the introduction, here they appear in their general form. The transition S-WEAK is justified by observing that (reading the STRUCT rule from bottom to top) if we have a context which promises more separation than we require, then we may forget about the extra separation. Transitions WEAK and PERM are justified by the fact that we consider the underlying combination of values to be given by a normal product type.

Using these contexts and structural rules we can simulate the bunches of the $\alpha\lambda$-calculus. If we replace the context former "," with $[1\#2](-,-)$ and ";" with $[](-,-)$ we can rewrite an $\alpha\lambda$ context into a λ_{sep} context. The associativity of the two context formers is then a two-way derived rule formed from two applications of STRUCT with FLATTEN and its inverse ($S = []_2$ or $[1\#2]$):

$$\frac{\varGamma(S(\varDelta_1, S(\varDelta_2, \varDelta_3))) \vdash e : A}{\varGamma(S(S(\varDelta_1, \varDelta_2), \varDelta_3)) \vdash e : A}$$

Since we have S-WEAK and WEAK we are simulating the affine $\alpha\lambda$-calculus. Another useful derived rule is a generalised form of contraction:

$$\frac{\varGamma(x : A)(y : A) \vdash e : B \qquad \neg(xS_\varGamma y) \qquad \forall z. xS_\varGamma z \Leftrightarrow yS_\varGamma z}{\varGamma(x : A)() \vdash e[x/y] : B}$$

This is derived by completely flattening out the context $\varGamma(x : A)(y : A)$ and then grouping the pair x, y. This is possible since they have exactly the same relationship to all other members of the context. Then, since x and y are not required to be separate then we may apply CONTR. Another sequence of FLATTEN applications reconstructs the original context's structure.

Connective Rules. The typing rules for tuple and function types are also shown in figure 2. By reading the contexts as representing the resources used by the term we obtain an informal justification of the typing rules. The rule S-I uses the same relationship between the contexts on the left as for the terms on the right; therefore, if the free variables of the terms obey the required separation then so will the corresponding terms. The elimination rule for tuples, S-E, exploits the structure of the contexts. The position of the hole in $\varDelta(-)$ indicates the relationships that the resources used by the variables x_i must have with the rest of $\varDelta$; by substituting $\varGamma$ directly into this hole we are maintaining the same relationships.

The rules $\rightarrow$-I and $\rightarrow$-E can be understood similarly; in the introduction rule we have the nested sub-context $\varGamma$ representing the resources used by the function body, treated as a single block. The required separation between the function's arguments and the function itself are recorded in S, which becomes part of the function's type. The relations are then reconstituted in the elimination rule. Using these function types we may simulate the function types of the $\alpha\lambda$-calculus: $A \rightarrow B$ becomes $A \xrightarrow{[]} B$ and $A \multimap\!\!* \, B$ becomes $A \xrightarrow{[1\#2]} B$.

The Prim rule incorporates a set of primitive operations $\mathcal{F}$ of the form $f : A \to B$ where A, B are types. We assume that primitive operations consume no resources themselves.

We have the usual admissible substitution rule for the calculus. We consider n-ary substitution both in order to have a strong enough induction hypothesis to handle the Contr structural rule and because it is needed for the well-definedness of the equational theory.

Lemma 2 (Substitution). *The following rule is admissible:*

$$\frac{\Gamma(x_1 : A_1)\dots(x_n : A_n) \vdash e : B \qquad \Delta_1 \vdash e_1 : A_1 \qquad \dots \qquad \Delta_n \vdash e_n : A_n}{\Gamma(\Delta_1)\dots(\Delta_n) \vdash e[e_1/x_1, \dots, e_n/x_n] : B}$$

Definition 3 (Equational Theory). *Given a set of axioms of the form $\Gamma \vdash e_1 = e_2 : A$ such that $\Gamma \vdash e_i : A$, $i \in \{1, 2\}$, define the judgement of equation-in-context ($\Gamma \vdash e_1 = e_2 : A$) to be given by the axioms, plus: extensions of the usual $\beta\eta$ rules for linear tuple and function types; commuting conversions for tuple elimination; uniqueness for terms of type $[]_0()$; surjective pairing for $[]_2(-,-)$ types; and congruence, symmetry and transitivity rules.*

Proposition 1. *If $\Gamma \vdash e_1 = e_2 : A$ is derivable in the equational theory then $\Gamma \vdash e_i : A$, $i \in \{1, 2\}$.*

The proof of this proposition is mostly straightforward, given the substitution lemma, apart from the commuting conversion rules which, due to the syntax free structural rules, are treated using a variant of the structural extensions of [10].

3 Semantics

This section describes the semantics of the calculus. We first briefly mention the categorical semantics of the system. This fixes the structure we require and provides generic coherence and soundness results for models. The main body of the section covers a resource-indexed sets semantics which shows how the calculus models resources and their separation.

Categorical Semantics. We give the syntax a categorical semantics by requiring a category $\mathcal{C}$ with endofunctors $\underline{\mathrm{S}} : \mathcal{C}^{|\mathrm{S}|} \to \mathcal{C}$ for each separation relation S. These model the product types and contexts of the syntax. The structural rules are natural transformations between these functors, subject to several commuting conditions, corresponding to the (term syntax invisible) reordering of structural rules. Function types are modelled as functors, $[A_1, \dots, A_n \xrightarrow{S} -] : \mathcal{C} \to \mathcal{C}$, right adjoint to the $\underline{\mathrm{S}}$ functors. The semantics extends a map $\mathcal{I}$ of primitive types and operators to objects and arrows in $\mathcal{C}$ to a map $[\![-]\!]_{\mathcal{I}}$ from derivations to arrows in $\mathcal{C}$. We have the following results for the categorical semantics:

Theorem 1 (Coherence). *If π_1 and π_2 are two derivations of the judgement $\Gamma \vdash e : A$ then $[\![\pi_1]\!]_{\mathcal{I}} = [\![\pi_2]\!]_{\mathcal{I}}$.*

Theorem 2 (Soundness and Completeness). *$\Gamma \vdash e_1 = e_2 : A$ if and only if in all categorical models $[\![\Gamma \vdash e_1 : A]\!]_{\mathcal{I}} = [\![\Gamma \vdash e_2 : A]\!]_{\mathcal{I}}$.*

Resource indexed sets. We will model the types of the system as sets indexed by a partially ordered set of "resources". A binary relation on the resources provides our semantical interpretation of separation.

We start with a partially ordered set R that has finite joins $r_1 \vee \ldots \vee r_n$. The ordering represents the inclusion of small resources inside larger ones, and joins represent the combination of resources. To model separation we require a relation $-\#- \subseteq R \times R$ with the following properties:

1. $r_1 \# r_2$ iff $r_2 \# r_1$;
2. If $r_1 \# r_2$ and $r_1' \sqsubseteq r_1$ and $r_2' \sqsubseteq r_2$ then $r_1' \# r_2'$;
3. $r \# (r_1 \vee \ldots \vee r_n)$ iff $r \# r_1, \ldots, r \# r_n$.

Intuitively these properties are those of separation: separation is symmetric; if two resources are separate and we have two other resources contained in them then the two smaller resources are separate; and if a collection of resources is separate from a resource then they are all separate individually.

Types are modelled as functors from R to **Set**, the category of sets and functions. Terms are modelled as natural transformations between these functors. The constructions for tuple and function types are instances of Day's general construction for monoidal closed structures in functor categories [4].

Tuple and contexts $\mathrm{S}(A_1, \ldots, A_n)$ are modelled by the sets, at resource r:

$$\begin{aligned}&\{(a_1, \ldots, a_n) \in A_1 r \times \ldots \times A_n r : \\ &\quad \exists r_1 \sqsubseteq r, \ldots, r_n \sqsubseteq r, a_1' \in A_1 r_1, \ldots, a_n' \in A_n r_n \text{ s.t.} \\ &\quad\quad \forall i. a_i = A_i(r_i \sqsubseteq r) a_i' \text{ and } \forall i, j. i \mathrm{S} j \Rightarrow r_i \# r_j\}\end{aligned}$$

An element of a tuple is a tuple of elements $(a_1, \ldots, a_n)$; each one represents a value in its own resource r_i, projected forward into the containing resource r. The resources for each of the elements must be related as dictated by the separation relation. For natural transformations modelling terms $f_1, \ldots, f_n$, $\mathrm{S}(f_1, \ldots, f_n)$ is defined pointwise in the evident way.

The function types $A_1, \ldots, A_n \xrightarrow{\mathrm{S}} B$ are modelled at resource r_0 as the family of functions:

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

where $R = \{(r_1, \ldots, r_n) \in R : \forall i, j. i \mathrm{R} j \Rightarrow r_i \# r_j\}$ and $\Rightarrow$ is the set-theoretic function space. The resource r_0 represents the resources that the function itself occupies. Given resources $r_1, \ldots, r_n$ for the argument positions that satisfy the separation relation we get a function from arguments at these resources to the result using the combined resources of the function and all its arguments. This matches the justification of the typing rule using the contexts to represent the resources and their relationships.

Theorem 3. *The above definitions on the category $[R, \mathbf{Set}]$ for tuple and function types, together with the evident natural transformations for the structural rules, give a sound class of models for λ_{sep}.*

Memory Regions. Our main example of the above construction is given by memory regions. Starting from some set of memory locations L, we take our set of resources to be the powerset of L and inclusion as the order. The relation is then defined as $r_1 \# r_2 \Leftrightarrow r_1 \cap r_2 = \emptyset$. 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 $[\mathcal{P}(L), \mathbf{Set}]$ is a model of λ_{sep}.

Representing Resources. This semantics suggests a useful way to extend the calculus via a simple application of the Yoneda embedding [8]. The Yoneda embedding allows us to represent resources $r \in R$ directly in the calculus as types Y_r. Separation with a value of type Y_r indicates separation from the fixed resource r. Following Yoneda we define:

$$Y_r(r') = \begin{cases} \{*\} & \text{if } r \sqsubseteq r' \\ \emptyset & \text{otherwise} \end{cases}$$

Thus Y_r is the empty type at resources inadequate for r and the singleton type at resources containing r.

Following on from the memory regions example, consider an example where we have a region k representing kernel memory in an operating system. Calls from the operating system kernel to user programs must not pass references to kernel memory, since it is inaccessible to user programs. This constraint may be typed as follows:

$$\texttt{callUserProgram} : [1\#2](Y_k, \mathit{Message}) \to \mathit{Result}$$

The representation of named resources in the calculus has a precedent in the *nominals* of hybrid logic. See, e.g. [1].

4 Non-symmetric Relations

A potentially useful variation of λ_{sep} is to allow non-symmetric relations in place of the separation relations. We can then model such constraints as allowable information flow or temporal ordering. We outline the changes required to the calculus and its semantics after a short example.

Take the base types to be *Src*, *Sink* and *Integer*, with primitive operations:

$$\texttt{read} : \mathit{Src} \to [](\mathit{Src}, \mathit{Integer}) \qquad \texttt{write} : [1 \triangleright 2](\mathit{Integer}, \mathit{Sink}) \to \mathit{Sink}$$

Thus, `read` takes a *Src* element and returns the next integer in that source, with the new state of the source; `write` takes a *Sink* and an integer, with the guarantee that the information in the integer may flow to the sink and returns the new state of the sink. We write the relations with $\triangleright$s to indicate the lack of symmetry. A simple example judgement is:

$$\Gamma \vdash \text{let } [](a', i) = \texttt{read}(a) \text{ in } [](a', \texttt{write}[1 \triangleright 2](i, x)) : [](\mathit{Src}, \mathit{Sink})$$

This types under $\Gamma = [1 \rhd 2](a : \mathit{Src}, x : \mathit{Sink})$, but not $\Gamma = [](a : \mathit{Src}, x : \mathit{Sink})$ since the typing of `write` requires information to be able to flow from (the source of) the integer to the sink.

By manipulating the typing context we may set up networks of allowable information flow between sources and sinks. For example, programs that satisfy the typing judgement cannot pass data from a source to a sink that is not explicitly allowed by the context. This context sets up a network that allows information to flow from a to x and b to x and y, but nothing else:

$$[1 \rhd 3, 2 \rhd 3, 2 \rhd 4](a : \mathit{Src}, b : \mathit{Src}, x : \mathit{Sink}, y : \mathit{Sink}) \vdash e : [](\mathit{Src}, \mathit{Src}, \mathit{Sink}, \mathit{Sink})$$

Note that this network is not expressible using binary constraints since it takes the shape not expressible in series-parallel graphs (Fact 1).

The relaxation to non-symmetric constraints has almost no effect on the typing rules. However, our suggested applications of non-symmetric relations indicate an additional structural transition for the calculus; transitive closure of relations ($TC(S)$ is the transitive closure of S):

$$\text{TC}\textsc{losure} \quad \Big| \quad \mathrm{S} \Rightarrow TC(\mathrm{S}) \quad \Big| \quad \phi(e) = e$$

The resource indexed semantics is easily extended to the non-symmetric case by dropping the requirement of symmetry on the relation # between resources and adding a requirement of transitive closure.

An instance of the resource indexed sets semantics may be constructed as follows. Take a set Id of "identities" – people's names for instance – with a binary relation $\rhd \subseteq Id \times Id$, denoting which people are allowed to talk to other people. This relation should be transitively closed. Extend the relation $\rhd$ to sets of identities by $I \rhd I'$ iff $\forall (i, i') \in I \times I'.(i \rhd i')$. Now take $\mathcal{P}(Id)$ as the set of "resources" with this relation and union as the combining operation. By a variation of the above theorem, $[\mathcal{P}(Id), \mathbf{Set}]$ has the correct structure to model the calculus with non-symmetric relations and transitive closure.

To take this example further we wish to compare the effectiveness of this extension with type systems designed for secure information flow such as [17]. In particular, we have not considered the effect of control flow on the information flow; a large factor in type systems for security.

5 Conclusions

We have presented the calculus λ_{sep} and shown how the separation relations in the syntax may be interpreted as separation constraints on the resources used by values. We have shown how the calculus may be extended to deal with non-symmetric relations between values and how it can model properties such as allowable information flow.

The idea of augmenting contexts with a relation on the members has also been used in Retoré's Pomset Logic [14], an extension of linear logic. Pomset (Partially Ordered Multiset) logic extends linear logic by adding a "before" connective

$A < B$, such that $A \otimes B \vdash A < B \vdash A \wp B$. This is interpreted, via proof nets, as being possible uni-directional communication, with $\otimes$ as no communication and $\wp$ as possible bi-directional commmunication. Retoré gives a coherence space semantics and a sequent calculus, but does not define n-ary tuple or implication formulae, nor does he consider nesting as a way of managing contexts. We believe that Retoré's coherence space semantics also works for a cut down variant of λ_{sep} with non-symmetric relations, but without contraction, weakening or function types. The "before" connective has also been considered by Reddy [13], da S. Corrêa, Haeusler and de Paiva [3] as a way of modelling temporal ordering in languages with state. da S. Corrêa *et. al.* also describe a semantics based on Dialectica Categories.

Reynolds' Syntactic Control of Interference (SCI) [15, 16], a variant of Idealized Algol, also controls the aliasing of values in the system by disallowing contraction, except in the case of *passive* variables that do not update shared storage. O'Hearn [9] describes a form of SCI based on the $\alpha\lambda$-calculus.

We plan further work with this calculus and its semantics to answer some questions we have not yet been able to solve and to attempt to extend and apply this calculus in other areas. The first question is that of completeness for the resource-indexed sets semantics. It is easy to generalise the semantics to use a category of resources, and attempt to build such a category from the syntax. The primary problem is that there does not seem to be an obvious way to build the relation $-\#-$ from the syntax. The second question is that of conservativity over the $\alpha\lambda$-calculus; we conjecture this to be true, but have not yet found a proof.

We are also investigating several extensions of λ_{sep}. Following SCI's passive types, we want to extend the calculus with *resource-insensitive* types. That is, types whose values do not occupy any resources and so separation constraints involving them have no meaning and can be added and removed arbitrarily.

We expect that the nested context structure of λ_{sep} will be useful for adapting the calculus to other uses. Having separation relations rather than binary context formers means that we maintain a linear ordering on the context where the $\alpha\lambda$-calculus would have to reorder context members to express some separation configurations. This means that the system should easily combine with ordered type systems for memory layout such as that of Petersen *et. al.* [11].

Lastly, we mention that the initial motivation for this calculus was to devise a higher-order version of the in-place update type system of Hofmann [5]. We have deviated from this goal in that it is not possible to directly express the separation of an object from *everything*, which is required to support in-place update. However, we note that one can express this using λ_{sep} using continuation passing style:

$$\texttt{cons} : [1\#2, 1\#3, 2\#3, 3\#4](A, \mathit{List}(A), \diamond, \mathit{List}(A) \to R) \to R$$

where $\diamond$s represent memory locations.

Acknowledgements I would like to thank David Aspinall and Michal Konečný for helpful discussions on the work presented here.

References

1. Carlos Areces and Patrick Blackburn. Bringing them all together. *Logic and Computation*, 11(5), 2001. Editorial of special issue on Hybrid Logics.
2. Denis Bechet, Philippe de Groote, and Christian Retoré. A complete axiomatisation for the inclusion of series-parallel partial orders. In *Proceedings of RTA'97*, volume 1232 of *Lecture Notes in Computer Science*, pages 230–240, 1997.
3. Marcelo da S. Corrêa, Edward H. Haeusler, and Valeria C. V. de Paiva. A dialectica model of state. In *CATS'96, Computing: The Australian Theory Symposium Proceedings*, January 1996.
4. 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, 1970.
5. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
6. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 185–197. ACM Press, 2003.
7. Michal Konečný. Functional in-place update with layered datatype sharing. In *Proceedings of TLCA 2003*, pages 195–210, 2003. LNCS 2701.
8. Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 2nd edition, 1998.
9. P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.
10. 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.
11. Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In G. Morrisett, editor, *Conference Record of the 30th Annual Symposium on Principles of Programming Languages (POPL'03)*, pages 172–184, January 2003. ACM Press.
12. D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series.* Kluwer Academic Publishers, 2002.
13. Uday Reddy. A linear logic model of state. Electronic manuscript: `http://www.cs.bham.ac.uk/~udr/`, October 1993.
14. Christian Retoré. Pomset logic: a non-commutative extension of classical linear logic. In *In proceedings of TLCA'97*, volume 1210 of *Lecture Notes in Computer Science*, pages 300–318, 1997.
15. 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.
16. John 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*, pages 704–722. Springer-Verlag, 1989. Lecture Notes in Computer Science 372.
17. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. Special issue on Formal Methods for Security.
18. J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series-parallel digraphs. *SIAM Journal of Computing*, 11(2):298–313, May 1982.