Substructural Simple Type Theories for Separation and In-place Update

Robert Atkey

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2006

Abstract

This thesis studies two substructural simple type theories, extending the "separation" and "number-of-uses" readings of the basic substructural simply typed λ-calculus with exchange.

The first calculus, λ_{sep}, extends the $\alpha\lambda$-calculus of O'Hearn and Pym by directly considering the representation of separation in a type system. We define type contexts with separation relations and introduce new type constructors of separated products and separated functions. We describe the basic metatheory of the calculus, including a sound and complete type-checking algorithm. We then give new categorical structure for interpreting the type judgements, and prove that it coherently, soundly and completely interprets the type theory. To show how the structure models separation we extend Day's construction of closed symmetric monoidal structure on functor categories to our categorical structure, and describe two instances dealing with the global and local separation.

The second system, λ_{inplc}, is a re-presentation a of substructural calculus for in-place update with linear and non-linear values, based on Wadler's Linear typed system with non-linear types and Hofmann's LFPL. We identify some problems with the metatheory of the calculus, in particular the failure of the substitution rule to hold due to the call-by-value interpretation inherent in the type rules. To resolve this issue, we turn to categorical models of call-by-value computation, namely Moggi's Computational Monads and Power and Robinson's Freyd-Categories. We extend both of these to include additional information about the current state of the computation, defining *Parameterised Freyd-categories* and *Parameterised Strong Monads*. These definitions are equivalent in the closed case. We prove that by adding a commutativity condition they are a sound class of models for λ_{inplc}. To obtain a complete class of models for λ_{inplc} we refine the structure to better match the syntax. We also give a direct syntactic presentation of Parameterised Freyd-categories and prove that it is soundly and completely modelled by the syntax. We give a concrete model based on Day's construction, demonstrating how the categorical structure can be used to model call-by-value computation with in-place update and bounded heaps.

Acknowledgements

I would like to thank my supervisor, David Aspinall, for his help and guidance in academic matters and especially his encouraging me to go to conferences and workshops to speak about my work as well as his skill in finding funding for me during my final year.

Many thanks are also due to my second supervisor, Ian Stark, as well as Alex Simpson and John Longley who sat on my progress review panels for my second and third years and first year respectively. I also wish to thank the other members of the Mobile Resource Guarantees project for providing the original motivation for this thesis.

I would also like to thank my office mates in Edinburgh, Uli Schöpp and Jan Obdrzalek, and my office mates in Warsaw, Artur Zawłocki and Piotr Hofmann, for putting up with my erratic working schedule. Michal Konečný was a great help in the initial stages and I greatly enjoyed working with him. Also I would like to thank Irwin Kennedy for many conversations on doing a PhD and many other things. The many people I met in Warsaw, to numerous to mention, greatly enriched my visit there.

My parents, Richard and Rosemary Atkey, have been constantly encouraging and always there when I needed them.

Lastly, but not leastly, my partner, Lauren Gerrard, has been a bottomless source of fun and reassurement over the course of my studies, especially during the final frantic days before submission.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Robert Atkey*)

Table of Contents

Chapter 1

Introduction

This thesis is about substructural simple type theories and their application to expressing separation and safe in-place update in programming languages. We extend O'Hearn and Pym's $\alpha\lambda$-calculus [Pym02, O'H03] to a new calculus λ_{sep} that allows finer control over separation. We also re-present Wadler's linear type system [Wad90] with non-linear types as the system λ_{inplc} and give it an equational theory. We investigate its semantics and discover connections with categorical models of call-by-value augmented with state information. The investigation of the semantics of linear typing leads to a third calculus, the *Typed Command Calculus*, a simply typed version of other type systems presented in the literature for typed memory management, such as Alias Types [SWM00] and the Capability Calculus [WCM00].

The motivation for both these investigations is a desire for more detailed control of memory within programs. We use ideas from substructural logics to formulate type systems that can express separation and memory access permission control, both of which have been identified as useful for the safe expression of memory management.

Control of separation is essential to prevent bugs arising from unintended *aliasing*. Aliasing is a long standing problem in computer science, and has been address by many researchers [Rey78, Pym02, Bak92, IO01, Red94, BNR01, BS93]. Aliasing arises when a single piece of the computer's store has two or more references. Each reference in a program has assumptions connected to it, either for-

mally by the type system or informally in the mind of the programmer. Aliased references that are not explicitly connected can become out of date with respect to each other, and this can lead to hard to track down memory errors. This thesis presents a foundational calculus for controlling separation to prevent aliasing, λ_{sep}. We introduce the ideas behind this calculus in Section 1.2 below.

Permission management is a way of stating the assumptions connected to a piece of memory. Strict control of permissions is required to make sure that the programmer is not able to treat a piece of memory using out-of-date assumptions. The distribution of permissions must be tightly controlled if they are not to become worthless. We use a substructural type system to accomplish this in our calculus λ_{inplc}. Our investigations of λ_{inplc} will lead us to develop extensions of categorical models of call-by-value programming languages that directly include a notion of permission management. We introduce λ_{inplc} and our models in Section 1.3.

Before we introduce the main work of this thesis, we first describe the general definition of substructural type theories, based on substructural logics.

1.1 Substructural Type Theories

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \to B}\ (\to\text{I})$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 e_2 : B}\ (\to\text{E})$$

Contexts, no variable x may appear more than once:

$$\Gamma \quad ::= \quad \epsilon \quad | \quad \Gamma, x : A$$

Figure 1.1: The Simply-Typed λ-calculus

Substructural Type Theories arise by the removal of the structural rules implicit in normal type theories. They are the Curry-Howard image of Substructural Logics [Res00, SHD93].

Structural rules determine the manipulations that may be applied to contexts. Given the normal typing rules of the simply-typed λ-calculus (Figure 1.1), the following structural rules are admissible:

$$\frac{\Gamma, x : A, y : A, \Gamma' \vdash e : B}{\Gamma, x : A, \Gamma' \vdash e[x/y] : B}\ (\textsc{Contraction}) \qquad \frac{\Gamma, \Gamma' \vdash e : B}{\Gamma, x : A, \Gamma' \vdash e : B}\ (\textsc{Weakening})$$

$$\frac{\Gamma, x : A, y : B, \Gamma' \vdash e : C}{\Gamma, y : B, x : A, \Gamma' \vdash e : C}\ (\textsc{Exchange})$$

We explain each of the rules in turn, and why they are admissible. The rule Contraction permits the identification of two previously independent variables in a typing derivation, allowing x to be used twice. It is admissible because the context Γ is shared between the two premises in the rule $\to$E. The rule Weakening shows that any derivation may have additional variables in the context without affecting the typability of the term. It is admissible because the rule Var permits Γ to contain an arbitrary list of variables, as long as it contains the one required. The final rule, Exchange, permits the reordering of variables in the context. It is admissible due to the form of the rule Var, which does not specify any particular order on the variables.

The type theories in this thesis are formed by restricting the application of the structural rules, in particular Contraction. We reformulate the rules in Figure 1.1 to explicitly state the application of structural rules, removing their implicit presence in the other rules. This reformulation is shown in Figure 1.2. We have altered the definition of contexts to be inductively constructed from variable/type pairs, a constant I and a binary constructor ",". We have also added a rule Struct, parameterised by "valid" transitions $\Gamma_1 \overset{\rho}{\Rightarrow} \Gamma_2$ between such contexts, where ρ is a map of variables in Γ_2 to variables in Γ_1, applied as a renaming in the Struct rule.

The restatement of contexts as constructed from type assignments, constants

$$\frac{}{x : A \vdash x : A}\ (\textsc{Id}) \qquad \frac{\Gamma_2 \vdash e : A \qquad \Gamma_1 \overset{\rho}{\Rightarrow} \Gamma_2 \ \mathit{valid}}{\Gamma_1 \vdash \rho(e) : A}\ (\textsc{Struct})$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \multimap B}\ (\multimap\text{I}) \qquad \frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : B}\ (\multimap\text{E})$$

Contexts, no variable x may appear more than once:

$$\Gamma \quad ::= \quad x : A \quad | \quad I \quad | \quad \Gamma_1, \Gamma_2$$

Figure 1.2: Basic Substructural λ-calculus

and constructors will allow us to generate more complex contexts with different structural rules applicable to the different constructors. The construction of contexts in this way is inspired by the $\alpha\lambda$-calculus (described below in Section 1.2) [O'H03] and is used to express the logic of Bunched Implications [OP99, Pym02] and also to uniformly present substructural logics in [Res00].

Along with this restatement of contexts as trees rather than lists, the restatement of structural rules in terms of transitions also allows greater control[1]. We have already mentioned the concept of valid structural transitions. These are defined inductively by a collection of rules. All of our systems include the following as valid structural transitions:

$$\frac{\rho(\Gamma') = \Gamma}{\Gamma \overset{\rho}{\Rightarrow} \Gamma' \ \mathit{valid}} \qquad \frac{\Gamma_1 \overset{\rho_1}{\Rightarrow} \Gamma_2 \ \mathit{valid} \qquad \Gamma_2 \overset{\rho_2}{\Rightarrow} \Gamma_3 \ \mathit{valid}}{\Gamma_1 \overset{\rho_1;\rho_2}{\Rightarrow} \Gamma_3 \ \mathit{valid}}$$

$$\frac{\Gamma_1 \overset{\rho}{\Rightarrow} \Gamma_2 \ \mathit{valid} \qquad \Gamma'_1 \overset{\rho'}{\Rightarrow} \Gamma'_2 \ \mathit{valid}}{\Gamma_1, \Gamma'_1 \overset{\rho;\rho'}{\Rightarrow} \Gamma_2, \Gamma'_2 \ \mathit{valid}} \qquad \frac{}{I \Rightarrow I \ \mathit{valid}}$$

These four rules express: identity up to renaming (α-equivalence), sequential composition, and two rules for congruence with the constants and constructors for

[1] The idea of representing structural rules as abstract transitions was inspired by [OPTT99]'s use of structural extensions which are sequences of structural rules.

contexts. In the third rule, note that $\rho; \rho' = \rho'; \rho$ by the disjoint variable condition on contexts. The empty mapping is not written in a transition: $\Gamma \Rightarrow \Gamma$ *valid* holds from the rules above for any Γ. We also wish to regard the comma as associative and I as its unit, so we include the following as valid transitions:

$$\begin{aligned} (\Gamma_1, \Gamma_2), \Gamma_3 &\Leftrightarrow \Gamma_1, (\Gamma_2, \Gamma_3) \ \textit{valid} \\ I, \Gamma &\Leftrightarrow \Gamma \ \textit{valid} \\ \Gamma, I &\Leftrightarrow \Gamma \ \textit{valid} \end{aligned}$$

The double headed arrows $\Leftrightarrow$ indicate that these transitions may be applied in both directions.

The valid structural transitions we have present so far are all implicit in the system of Figure 1.1, by construction of contexts as lists. The three structural rules identified above may also be expressed as structural transitions:

$$\frac{\Gamma \equiv_\alpha \Gamma'}{\Gamma \overset{[v(\Gamma') \mapsto v(\Gamma)]}{\Rightarrow} \Gamma, \Gamma' \ \textit{valid}} \ (\textsc{Contraction}) \qquad \frac{}{\Gamma \Rightarrow I \ \textit{valid}} \ (\textsc{Weakening})$$

$$\frac{}{\Gamma_1, \Gamma_2 \Leftrightarrow \Gamma_2, \Gamma_1 \ \textit{valid}} \ (\textsc{Exchange})$$

In the Contraction rule, the premise $\Gamma \equiv_\alpha \Gamma'$ states that the two contexts should be α equivalent, i.e. equivalent up to renaming of variables. The notation $v(\Gamma)$ denotes to the list of variable names in Γ, taken in depth-first left-to-right order.

The rules in Figure 1.2, when we consider all of the named structural transitions as valid, have the same power as the original system. The rule Var is derivable from the rule Id and repeated applications of the rules Weakening and Exchange; the rule $\multimap$I is identical in form to $\to$I; and $\to$E is derivable from $\multimap$E and repeated applications of the rules Contraction and Exchange. Conversely, the rule Id is a special case of Var; the rules $\to$I and $\multimap$I are identical in form; and $\multimap$E is admissible by repeated uses of Weakening and Contraction to make the context in both premises equal to Γ_1, Γ_2 and an application of $\to$E.

Removing admissible structural rules from the basic system of Figure 1.1 changes the meaning of terms and the function type. Removing Contraction

means that a function body may not use a variable more than once, and removing Weakening means that a function must use a variable at least once. Note that this basic reading of the removal of the structural rules does not hold when we move to more complex substructural systems. See the description of the $\alpha\lambda$-calculus below in Section 1.2.

The system may now seem to be overly complicated; we have added an extra rule and a complex system of contexts and valid structural transitions to produce a system with exactly the same typable terms. However, the new complexity brings more flexibility. We can alter the definitions of contexts and valid structural transitions to get new systems. By reading the structural rules, and their omission, as having computational significance, we can develop new type systems that have useful properties. In this thesis we will consider the removal of Contraction and Weakening and their application to the expression of separation and in-place update. We introduce these in Sections 1.2 and 1.3 respectively.

Before we do that, in the rest of this section we introduce some more issues common to all substructural type systems. We first discuss the introduction of new type constructors given by varying the structural rules, then the effect of these constructors on the equations of a type theory, and finally we discuss the categorical semantics of substructural systems.

1.1.1 New type constructors

Varying the valid structural transitions allows the introduction of a new binary type constructor, the tensor product: $A \otimes B$. This has the following introduction and elimination rules:

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash e_1 \otimes e_2 : A \otimes B} \ (\otimes\mathrm{I})$$

$$\frac{\Gamma_1 \vdash e_1 : A \otimes B \qquad \Gamma_2(x : A, y : B) \vdash e_2 : C}{\Gamma_2(\Gamma_1) \vdash \text{let } x \otimes y = e_1 \text{ in } e_2 : C} \ (\otimes\mathrm{E})$$

The introduction rule places the contexts in the same order as the terms; the comma between the contexts mirrors the $\otimes$ between the terms and types. In the

$\otimes$E rule, the context $\Gamma_2(-)$ represents a context with a missing sub-term, filled in with $x : A, y : B$ in the premise and Γ_1 in the conclusion. The free variables of e_1 are placed into the same subcontext as x and y. Note that the two variables x and y are separated by a single comma in the context, matching the $\otimes$ in the pair being split. When new context constructors are introduced below, there will be new tensor products connected to them in the same way.

Due to the close correspondence between the terms and contexts in the introduction and elimination rules, the presence or absence of the structural rules has an effect on the way in which we interpret the tensor type.

When all the structural rules are present, the tensor type behaves similarly at the type judgement derivation level to the normal product type $A \times B$. The following judgements are derivable:

$$x : A \vdash (x, x) : A \otimes A \qquad\qquad x : A \vdash \star_I : I$$

$$x : A \otimes B \vdash \text{let } (y, z) = x \text{ in } (z, y) : B \otimes A$$

In terms of structural rules, the judgements correspond to a use of the rules Contraction, Weakening and Exchange respectively. If a rule is missing, then the corresponding judgement is not derivable. Hence, the operations permissible on the type $A \otimes B$ match the valid structural transitions applicable to the corresponding context constructor.

Removing the structural rules does not prevent the system being extended with the traditional product type. It can be introduced using shared contexts and eliminated using projections:

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A \times B}\ (\times\text{I}) \qquad\qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i e : A_i}\ (\times\text{E-}i)$$

1.1.2 Type Theories

To be a type *theory*, there must be some notion of equality between terms. In the case of the basic calculi presented in Figures 1.1 and 1.2, we have the following two equations:

$$(\lambda x : A.e_1)e_2 \quad = \quad e_1[e_2/x]$$

$$\lambda x : A.ex \quad = \quad e$$

These are the equational versions of the normal β and η reduction rules respectively. They are the same for the non-substructural and substructural systems. It is easy to see that if the left hand side of each equation is typable, then so is the right hand side, under the same context and result type: for the β rule this is by substitution, and for the η rule this is by inversion of the typing rules. When we introduce the tensor product type, however, complications arise. We want the usual β and η equational rules:

$$\begin{aligned} \text{let } (x, y) = (e_1, e_2) \text{ in } e_3 &= e_3[e_1/x, e_2/y] \\ \text{let } (x, y) = e \text{ in } (x, y) &= e \end{aligned}$$

Both sides of these rules are obviously typable, by direct application of the type rules and by the substitution property. However, they do not cover all the possible equations that we desire between terms. Due to the presence of "parasitic types" in the elimination rules for products, we must be able to permute these eliminations with the other rules. For example, these two terms should be equal when x' and y' are not free in e_3.

$$\begin{gathered} (\text{let } (x, y) = (\text{let } (x', y') = e_1 \text{ in } e_2) \text{ in } e_3) \\ = \\ (\text{let } (x', y') = e_1 \text{ in let } (x, y) = e_2 \text{ in } e_3) \end{gathered}$$

The traditional approach is to require a collection of *commuting conversion* rules that state all the required permutations explicitly. However, this leads to unwieldy systems of rules, particularly when we must consider their interaction with the syntax-free structural rules.

To avoid this problem, we adopt the approach of Ghani [Gha95] and use generalised η expansion:

$$\text{let } (x, y) = e_1 \text{ in } e_2[(x, y)/z] \quad = \quad e_2[e_1/z]$$

This rule subsumes the commuting conversion rules, and when all the named structural rules above are present, ensures that the product obeys the surjective pairing property.

Ghani's rule is applicable in the λ_{sep} and λ_{inplc} calculi presented in this thesis, but we are forced to use explicit commuting conversions in the Typed Command Calculus. See Chapter 6 for details.

1.1.3 Categorical Semantics of Substructural Type Theories

In order to give the precise requirements for models of our substructural type theories, we formulate categorical semantics for them. We do this in the style of the cartesian closed category semantics for the simply typed λ-calculus [LS88, Cro94] by interpreting contexts and types by objects of a category $\mathcal{C}$ and judgements $\Gamma \vdash e : A$ by arrows $[\![\Gamma]\!] \to [\![A]\!]$. Equality of typed terms is modelled by equality of arrows.

We interpret context constructors and constants by functors $\mathcal{C}^n \to \mathcal{C}$, where n is the arity of the constructor. Valid structural transitions are interpreted as natural transformations between multiple applications of the context constructor functors. We define the interpretations by induction on the structure of contexts and the derivation of validity, respectively. Tensor product types are interpreted using the same functors as the context constructors to which they correspond. Function types are interpreted using the usual right adjoints to the context constructors from which they are formed.

There are many ways of deriving valid structural transitions and, due to the fact that the Struct rule introduces no term syntax, we must ensure that all derivations of the same valid transition $\Gamma_1 \stackrel{\rho}{\Rightarrow} \Gamma_2$ have the same interpretation. In the case of the basic system with a single binary context constructor and unit, this is exactly the coherence of symmetric monoidal structure on a category [Mac98]. When we consider the semantics of λ_{sep} in Chapter 3 and the direct semantics of λ_{inplc} in Chapter 8 we will have to prove this for ourselves.

Because Struct introduces no term syntax, there may be many derivations of a single typing judgement $\Gamma \vdash e : A$. We must ensure that each of these derivations has the same interpretation. We do this following O'Hearn *et al*'s proof of coherence of the interpretation of Syntactic Control of Interference Revisited [OPTT99], effectively by rewriting the derivation tree to a normal form

and proving that the process of rewriting preserves the interpretation. To do this we require an abstract characterisation of valid structural transitions and a way of "factorising" them over applications of introduction and elimination rules that preserves their interpretation. We do this for λ_{inplc} in Section 3.3.1. The other substructural type theory in this thesis, λ_{inplc}, also requires a proof of coherence, which is completed in the same way.

1.2 The Separation Reading : $\alpha\lambda$ and λ_{sep}

In this section we describe a way to express separation in typing contexts via the $\alpha\lambda$-calculus and our extension, λ_{sep}.

The $\alpha\lambda$-calculus [O'H03, Pym02] is the typed λ-calculus corresponding via the Curry-Howard isomorphism to the Logic of Bunched Implications (BI) [OP99]. The $\alpha\lambda$-calculus combines a substructural type system without Contraction or Weakening with an intuitionistic type system. The contexts of the two systems are mixed by allowing contexts to be constructed from two context formers "," and ";". The comma only allows Exchange, while the semicolon allows all the structural rules. The contexts are given by the following grammar:

$$\Gamma \;::=\; x : A \;\mid\; I \;\mid\; 1 \;\mid\; \Gamma_1, \Gamma_2 \;\mid\; \Gamma_1; \Gamma_2$$

with the usual condition that no variable x may appear more than once in a context.

The $\alpha\lambda$-calculus has associativity and left and right unit transitions for both the comma and I and the semicolon and 1, as well as congruence, identity and composition. The interesting part lies in the fact that the comma and I have only Exchange:

$$\Gamma_1, \Gamma_2 \Leftrightarrow \Gamma_2, \Gamma_1 \;\; \textit{valid}$$

While the semicolon and 1 have all the structural rules:

$$\frac{\Gamma \equiv_\alpha \Gamma'}{\Gamma \overset{[v(\Gamma') \mapsto v(\Gamma)]}{\Rightarrow} \Gamma; \Gamma' \;\; \textit{valid}} \qquad \frac{}{\Gamma \Rightarrow 1 \;\; \textit{valid}} \qquad \frac{}{\Gamma_1; \Gamma_2 \Leftrightarrow \Gamma_2; \Gamma_1 \;\; \textit{valid}}$$

The introduction rule for the product type is altered from the formulation at the end of Section 1.1.1 to make it clear that it is related to the semicolon:

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2 \vdash e_2 : B}{\Gamma_1; \Gamma_2 \vdash \langle e_1, e_2 \rangle : A \times B} \ (\times\text{I})$$

By the Contraction structural transition, the rule for product introduction from Section 1.1.1 is derivable from this one. We could also use an elimination rule for products similar to the let $(x, y) = e_1$ in e_2 construct for eliminating tensor products. Such a construct is required for the more flexible products of λ_{sep} introduced below and developed in Chapter 2, however.

The substructural function type $A \mathrel{-\!\!*} B$ is expressed in the same way as in Figure 1.2, albeit with a different symbol to emphasise the difference from presentations of the linear λ-calculus.

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \mathrel{-\!\!*} B} \ (\mathrel{-\!\!*}\text{I}) \qquad \frac{\Gamma_1 \vdash e_1 : A \mathrel{-\!\!*} B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1 @_* e_1 : B} \ (\mathrel{-\!\!*}\text{E})$$

Note the use of the comma in the premise of the introduction rule and the conclusion of the elimination rule. The same pattern may be applied with the semicolon to get the simply-typed function space:

$$\frac{\Gamma; x : A \vdash e : B}{\Gamma \vdash \alpha x.e : A \to B} \ (\to\text{I}) \qquad \frac{\Gamma_1 \vdash e_1 : A \to B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1; \Gamma_2 \vdash e_1 e_2 : B} \ (\to\text{E})$$

Note that whereas we could have included the product type $A \times B$ without introducing the context constructor ";" by the rules in Section 1.1.1, the introduction of the non-substructural function type $A \to B$ requires the use of ";" to formulate the rule $\to$I.

The inclusion of two kinds of function type alters the behaviour of the substructural function type $A \mathrel{-\!\!*} B$. Despite the lack of Weakening and Contraction we cannot regard it as a type of functions that use their argument only once, as in the linear λ-calculus [Wad93b]. Consider the following example due to O'Hearn

[O'H03]:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : A; f : A \to A \to B \vdash fxx : B}}{x : A \vdash \alpha f.fxx : (A \to A \to B) \to B}}{I, x : A \vdash \alpha f.fxx : (A \to A \to B) \to B}}{I \vdash \lambda x.\alpha f.fxx : A \multimap (A \to A \to B) \to B}$$

Despite the fact that λx is a substructural function abstraction, its body mentions x twice. The function argument bound to f has no notion of any connection between its two arguments, as it would if they had been combined in an argument of type $A \times B$, and may use both of them.

The $\alpha\lambda$-calculus has a strong reading in terms of separation. Reading the comma context constructor as stating that the two halves must be separate – due to the lack of Contraction – and the semicolon as stating that the two sides may share, we can give meanings to the two kinds of products and the two kinds of function abstraction.

Since the two product types match the context constructors, we can immediately read the tensor $A \otimes B$ as a separated, or non-sharing pairing and the product $A \times B$ as a pairing of values that may share resources. Due to the use of a comma between the abstracted variable and the rest of the context, the function type $A \multimap B$ can be read as the type of functions whose arguments are separate from the function – the resources of the function itself being represented by the free variables. Likewise, the use of a semicolon in the introduction rule of the type $A \to B$ allows sharing between the function and its arguments.

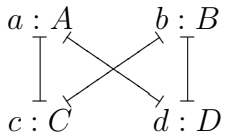
For the sharing interpretation of the calculus, we may also allow weakening for the comma. The simplest way to allow this is to identify the two units I and 1 with a single unit 1. This variant is called the *affine* $\alpha\lambda$-calculus. In this variant, we also have a derived valid structural transition called Dereliction:

$$\Gamma_1, \Gamma_2 \Rightarrow (\Gamma_1, \Gamma_2); (\Gamma_1', \Gamma_2') \Rightarrow (\Gamma_1, 1); (1, \Gamma_2') \Rightarrow \Gamma_1; \Gamma_2' \equiv_\alpha \Gamma_1; \Gamma_2$$

This rule states that we may forget the fact that Γ_1 and Γ_2 are separate and just regard them as a pair that may share.

There are other variants of the $\alpha\lambda$-calculus that replace or add binary context formers and alter the structural rules that are applicable over them, in particular the removal of EXCHANGE. This allows the expression of pointers from one heap to another, but not in the opposite direction. See [O'H03] for more details

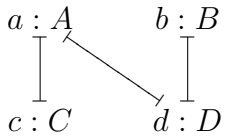
The context in the $\alpha\lambda$-calculus represents the permissible sharing relationships between the individual free variables used by the term. For example, the context $(a : A; b : B), (c : C; d : D)$ represents the following separation relationships:



From the diagram it is easy to see that the two pairs a, b and c, d are not required to be separate internally, and that there must be total separation between them. This is the pattern of separation given by the structure of the context.

An obvious question is whether every possible pattern of separation may be represented by an $\alpha\lambda$-calculus-style context. This is not possible. The use of two contexts formers "," and ";" means that only so-called *series-parallel* separation graphs may be constructed; graphs constructed from pairwise combination of graphs with no separation or total separation. The following fact [BdGR97, VTL82] states that not all graphs are series-parallel:

Fact 1.2.1 (SP-graph Characterisation) A graph is series-parallel iff its restriction to any four vertices is not equal to the graph:



The type theory for separation that we define and investigate in this thesis, λ_{sep}, overcomes this problem by directly expressing an arbitrary pattern of separation in the context. We will describe λ_{sep} in depth in Chapter 2. Here we give a quick overview, based on the general plan for substructural type systems we laid out in Section 1.1. The contexts of λ_{sep} are defined by the following grammar:

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

where S is a *separation relation*, defined in Definition 2.1.1 as a binary symmetric relation on the contexts $\overrightarrow{\Gamma}$. The intended meaning of the separation relation is that the variables in the contexts $\overrightarrow{\Gamma}$ refer to data which is separated according to the relation. The context with the empty, zero-place separation relation gives a representation of an empty context.

The valid structural transitions for λ_{sep} include the identity, composition and congruence transitions, plus the following transitions:

$$\frac{}{\mathrm{S}(\overrightarrow{\Gamma}, \mathrm{S}'(\overrightarrow{\Delta}), \overrightarrow{\Gamma'}) \Leftrightarrow \mathrm{S}\{\mathrm{S}'/\mathrm{i}\}(\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Gamma'})} \text{ (Flatten)} \qquad \frac{}{[]_1(\Gamma) \Leftrightarrow \Gamma} \text{ (Single)}$$

$$\frac{\mathrm{S}' \subseteq \mathrm{S}}{\mathrm{S}(\overrightarrow{\Gamma}) \Rightarrow \mathrm{S}'(\overrightarrow{\Gamma})} \text{ (S-Weak)} \qquad \frac{\Gamma \equiv_\alpha \Gamma'}{\Gamma \overset{v(\Gamma') \mapsto v(\Gamma)}{\Rightarrow} []_2(\Gamma, \Gamma')} \text{ (Contraction)}$$

$$\frac{}{\Gamma \Rightarrow []_0()} \text{ (Weakening)} \qquad \frac{\sigma \text{ a permutation}}{\mathrm{S}(\overrightarrow{\Gamma}) \Leftrightarrow \sigma\mathrm{S}(\sigma\overrightarrow{\Gamma})} \text{ (Permutation)}$$

All of these transitions can be justified in terms of separation, which we do in Section 2.2. The grammar of contexts also gives rise to two new type constructors, separation products and separation functions:

$$\mathrm{S}(A_1, \ldots, A_n) \qquad\qquad A_1, \ldots A_n \xrightarrow{\mathrm{S}} B$$

In the case when S is a two place relation then depending on the particular relation used, these are equivalent to the affine $\alpha\lambda$-calculus product types and function types. The translation from the $\alpha\lambda$-calculus into λ_{sep} is given in Section 2.4.1.

1.2.1 Semantics of Separation

There are categorical semantics for the $\alpha\lambda$-calculus and λ_{sep} matching the general scheme described in Section 1.1.3 above. For the $\alpha\lambda$-calculus, this is given in [O'H03, Pym02] as a category $\mathcal{C}$ with finite products for interpreting ";"/$\times$ and 1, and symmetric monoidal products for interpreting ","/$*$ and I. The two function types are interpreted by assuming that both product functors have right adjoints.

We define a categorical semantics for λ_{sep} following the same scheme. We assume functors $\underline{\text{S}} : \mathcal{C}^{|\text{S}|} \to \mathcal{C}$ for each separation relation S, and natural transformations to interpret the valid structural transitions. We describe this semantics in Chapter 3, carefully ensuring that we have enough coherence conditions to coherently interpret valid structural transitions and typing derivations.

This semantics gives the abstract structure required to coherently and soundly interpret λ_{sep}, but does not explicate its meaning in terms of separation of resources. To do this, in Chapter 4, we construct categories with the required structure from categories of functors from some category with structure, $\mathcal{C}$, to the category of sets and functions, **Set**. The basic idea is that the objects of $\mathcal{C}$ represent abstract "resources" and functors $\mathcal{C} \to \mathbf{Set}$ interpret types as sets indexed by the resources available. This is a generalisation of the frame, or possible world, semantics of substructural logics [Res00].

O'Hearn, Pym and Yang [O'H03, Pym02, POY04] use Day's construction of symmetric monoidal structure in functor categories [Day70] to give a semantics for the $\alpha\lambda$-calculus. Day defines symmetric monoidal structure on $[\mathcal{C}, \mathbf{Set}]$ by a co-end formula[2]:

$$(A \otimes B)X = \int^{Y,Z} AY \times BZ \times P(Y, Z, X)$$

where $P : \mathcal{C}^{\text{op}} \times \mathcal{C}^{\text{op}} \times \mathcal{C} \to \mathbf{Set}$ is a functor equipped with several natural transformations. This construction also makes the given monoidal structure closed. Since $[\mathcal{C}, \mathbf{Set}]$ is always cartesian closed, this gives a model for the $\alpha\lambda$-calculus.

We extend Day's construction in the **Set** case by considering functors $P_{\text{S}} : (\mathcal{C}^{\text{op}})^{|\text{S}|} \times \mathcal{C} \to \mathbf{Set}$, with a different collection of natural transformations. We prove in Chapter 4 that this gives the structure required to coherently and soundly model λ_{sep}.

We connect this semantics to resources and separation in Sections 4.3.2 and 4.3.3. In Section 4.3.2 we consider "global separation", where we treat each object of the domain category as an individual resource. Resources may be combined using coproducts and there is a predicate, modelled as a binary contravariant

[2]Day [Day70] also handles the case for codomain categories other than **Set** by using enriched categories, but we only consider the case for **Set** in this thesis.

functor to **2**, for expressing separation. The second model, in Section 4.3.3, models "local separation" where we consider embeddings of sub-resources into a larger resource, and their separation and combination as embeddings.

1.3 In-place Update

In the second half of this thesis we investigate another application of the restriction of the CONTRACTION rule – *in-place update* – by constructing a type theory λ_{inplc}, derived from Wadler's linear/non-linear type system [Wad90] and Hofmann's LFPL [Hof00]. In-place update refers to the updating of memory cells with new values, and possibly with different types to the original values. Changing the type of the data stored in a memory cell means that we have to careful not to access the memory cell with out-of-date information about its type. This would lead to run-time type errors.

In-place update is potentially useful because it allows the expression of explicit memory management in a safe way. Explicit memory management means that the precise use of memory within a program is controlled by the programmer, who has to handle all allocation and deallocation and reuse of memory. This is already present in languages such as C [ISO99], but the programmer is given no help in making sure that they do not mistakenly use memory that has been de-allocated or access memory with mistaken assumptions about its type. Another advantage of safe in-place update is that the expression of low-level memory operations in the typing rules gives programming language designers a handle with which to control memory consumption and thus develop type systems for resource limited programming. Hofmann's LFPL (Linear Functional Programming Language) [Hof00] is designed with this application in mind. We use LFPL as a guide for the design of λ_{inplc}.

Another potential use of in-place update is eased reasoning about side-effecting programs. By making the types of the memory cells operated on by the program explicit, the requirements on the initial and final states of the memory required and delivered by the program are treated as normal inputs and outputs. This is

in contrast to the normal situation where the state of the memory is implicit and the programmer receives little help from the type system.

Also, we hope to be able to forget about the imperative interpretation of the language and treat programs as a restricted subset of functional programs and hence apply functional reasoning techniques such as β-reduction and substitution. It will turn out, however, that the equational theory of λ_{inplc} is affected by the type system itself. The rule of substitution is no longer admissible. This is due to a call-by-value interpretation being required by the typing rules.

We illustrate how the removal of Contraction allows us to type safe in-place update. Let the following be primitive operations in a typed λ-calculus:

$$\texttt{store}_A : \Diamond \otimes A \to [A] \qquad \texttt{retrieve}_A : [A] \to [A] \otimes A \qquad \texttt{forget}_A : [A] \to \Diamond$$

The operation $\texttt{store}_A$ takes a memory cell, represented by a value of type $\Diamond$[3], and a value of type A and returns a pointer to the same memory cell, now containing that value of type A. The operation $\texttt{retrieve}_A$ takes a memory cell containing a value of type A and returns a pair of the memory cell and the contained value. The operation $\texttt{forget}_A$ is a no-op coercion that takes a memory cell and "forgets" the type of the data it contains.

If we were to allow Contraction in the calculus, then it would be possible to type programs such as (where the free variable d has type $\Diamond$):

$$\begin{aligned}&\text{let } x = \texttt{store}_{Int}(d, \texttt{2}) \text{ in}\\ &\text{let } y = \texttt{store}_{Bool}(d, \texttt{true}) \text{ in}\\ &\text{let } (d', i) = \texttt{retrieve}_{Int} x \text{ in}\\ &\quad i + \texttt{4}\end{aligned}$$

This program fragment stores the integer 2 in the memory cell d, obtaining an integer view of the memory cell, stored in x. It then stores the boolean `true` in d, obtaining a boolean view of the same memory, stored in y. The third line then attempts to read out the integer stored in the first line. However, this attempt will fail since it has been overwritten by a boolean in the second line. This line,

[3]The type $\Diamond$ to represent unused memory is taken from LFPL [Hof00]. Diamonds are precious, and so is unused memory.

or the line following will cause a run-time error when it tries to use the boolean value as an integer. Removing Contraction makes the program above untypable since we would not be able to use d twice.

The system without contraction is far too restrictive. There are types, such as that of integers and booleans, whose values do not occupy any space in the computer's mutable store. Values of these types cannot alias one another, and permission to use integers or booleans can be duplicated at will because they are just values.

To fix this we follow the approach of Wadler [Wad90] and Hofmann [Hof00] (see also [Wal05]) and introduce a distinction between *stateful* and *state-free* types. Stateful types covers types that represent memory cells, such as $[A]$ or $\Diamond$ above, or linked data structures that reside in the store, such as lists in LFPL. State-free types are those such as integers and booleans that are manipulated in machine registers or on the stack. State-free types may have the rules of Contraction and Weakening applied, as shown by these conditional valid structural transitions:

$$\frac{\mathsf{sf}(\Gamma)}{\Gamma \Rightarrow I \ \mathit{valid}} \qquad\qquad \frac{\mathsf{sf}(\Gamma) \qquad \Gamma \equiv_\alpha \Gamma'}{\Gamma \overset{[v(\Gamma)/v(\Gamma')]}{\Rightarrow} \Gamma, \Gamma' \ \mathit{valid}}$$

where $\mathsf{sf}(\Gamma)$ is true if Γ consists of only variables of state-free type.

The tensor product of two state-free types is considered state-free. There are two function types $A \to B$ and $A \multimap B$ which are state-free and stateful respectively. The difference between them is in their introduction rules:

$$\frac{\Gamma, x : A \vdash e : B \qquad \mathsf{sf}(\Gamma)}{\Gamma \vdash \lambda_{\to} x : A.e : A \to B} \qquad\qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda_{\multimap} x.e : A \multimap B}$$

All of the free variables of a function of type $A \to B$ must be state-free (the premise $\mathsf{sf}(\Gamma)$). Therefore, values of type $A \to B$ contain no stateful part and so are state-free. Values of type $A \multimap B$ may have stateful variables free in them, so they are stateful.

The calculus that we have just sketched, λ_{inplc}, is defined in detail in Chapter 8. It is capable of typing safe memory management operations such as in-place

update and de-allocation. To use it for reasoning about programs as well, we must establish an equational theory for its terms. An immediate obstacle is that the rule of substitution is not admissible for this calculus:

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2(x : A) \vdash e_2 : B}{\Gamma_2(\Gamma_1) \vdash e_2[e_1/x] : B}$$

This rule is the basis of reasoning about terms of the calculus by β-reduction. The failure of the substitution rule is shown by the following judgement:

$$x : \mathrm{Int} \vdash (x, x) : \mathrm{Int} \otimes \mathrm{Int}$$

This judgement is valid if we assume that Int is state-free. Now consider another judgement:

$$d : \mathrm{Cell} \vdash \mathtt{location}(d) : \mathrm{Int}$$

where the type Cell is not state-free. An attempt to substitute the term $\mathtt{location}(d)$ for x in the first judgement would result in an untypable term:

$$(\mathtt{location}(d), \mathtt{location}(d))$$

The variable d appears twice in the term, but since d is of a non-state-free type this is not possible.

We resolve this problem by noticing that all the operational semantics of this calculus [Wal05, Hof00] have been *call-by-value*. That is, in the execution of a term like let $x = \mathtt{location}(d)$ in (x, x), the sub-term $\mathtt{location}(d)$ is reduced to a value (i.e. the location), before being used in the body of the let expression. Direct substitution does not take this into account.

We study the semantics of λ_{inplc} to make clear the call-by-value execution strategy and also to explicate the meaning of the removal of Contraction. There are two ways to read the removal of Contraction. One is to follow the reading of the $\alpha\lambda$-calculus and λ_{sep} in Section 1.2 and view it as the control of aliasing. Memory cells are *aliased* when they have more than one reference to them. In the above program, the single memory cell d becomes aliased by using the variable d twice. The prohibition of Contraction prevents this. In terms of separation,

a non-aliased memory cell has a single reference that is separate from all other references. Non-aliasing is taken as primary in Reynolds' Syntactic Control of Interference [Rey78, Rey89] and the Separation Logic of O'Hearn, Ishtiaq and Reynolds [Rey02, IO01].

Another view is that of *permission control*. If we view a value of type $\Diamond$ as a permission to store data in the memory cell it represents, then the lack of Contraction is the inability to duplicate a permission into two independent permissions. Once a permission has been "used up", by the $\texttt{store}_A$ operation, it is no longer available and we get a new permission $[A]$ in return. In this reading, the lack of Weakening means that we cannot discard a permission. This is the "number-of-uses" reading of the lack of Contraction and Weakening in Linear Logic [Gir87, Wad93b, Tro93]. The view as permissions has also been emphasised in recent work on type systems for imperative object-oriented programming [BNR01] and verifying concurrent programs in Separation Logic [O'H05].

We investigate the permissions view by extending categorical models of call-by-value programming languages with attached sets of permissions. Types of λ_{inplc} will be interpreted as pairs of value types and permissions to operate on the state.

1.3.1 Semantics of In-place Update

In this section we will briefly describe the ideas behind our categorical semantics of λ_{inplc} to be studied in Chapters 5 and 8. We extend two categorical models of call-by-value programming languages, *Freyd categories* and *Computational Monads*, described in detail in Section 5.1. We introduce our extension of them to *Parameterised* Freyd categories and *Parameterised* Monads. We will use these to give models of λ_{inplc}. Since λ_{inplc} only corresponds to such structures that have an additional commutativity property, we give a calculus directly related to them, the Typed Command Calculus, in Chapter 6.

Both Freyd categories [PR97, PT99] and Computational Monads [Mog91] model call-by-value by splitting it into two parts. First, they separate values and computations (though values here are not exactly the values used in operational

semantics). Freyd categories accomplish this by considering identity-on-objects functors $J : \mathcal{C} \to \mathcal{K}$, where $\mathcal{C}$ is category of "values" and $\mathcal{K}$ is a category of "computations". The computational monad approach considers a category $\mathcal{C}$ and takes all arrows of the form $A \to TB$ to be "computations", where T is the functor part of the monad. Secondly, both approaches consider computations-in-context, building on some symmetric monoidal structure of the value category $\mathcal{C}$. Freyd categories do this by requiring *premonoidal* structure on $\mathcal{K}$, consisting of two functors $A \mathbin{\circledcirc} - : \mathcal{K} \to \mathcal{K}$ and $- \mathbin{\circledcirc} A : \mathcal{K} \to \mathcal{K}$ that agree on objects, with some additional natural isomorphisms, such that everything is strictly preserved by J. Computational monads are required to have a *strength* $\tau_{A,B} : A \otimes TB \to T(A \otimes B)$, obeying some axioms. Power and Robinson [PR97] prove that, when J has a right adjoint, these two notions are equivalent.

We augment these definitions to handle explicit typing of the state by adding a parameter category $\mathcal{S}$. Objects of $\mathcal{S}$ are intended to represent permissions to access parts of the state. We extend the definition of Freyd category by considering identity-on-objects functors $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$. Thus, every arrow of $\mathcal{K}$ is tagged with a start and finish permissions object taken from $\mathcal{S}$. We form composite permissions by requiring monoidal structure on $\mathcal{S}$. As well as the premonoidal structure on $\mathcal{K}$ with respect to $\mathcal{C}$, we also require premonoidal structure with respect to $\mathcal{S}$, allowing us to lift arrows in $\mathcal{K}$ to larger permissions contexts:

$$\frac{(A, S_1) \xrightarrow{f} (B, S_2)}{(A, S_1 \otimes S) \xrightarrow{f \mathbin{\circledcirc}_{\mathcal{S}} S} (B, S_2 \otimes S)}$$

We also extend the definition of monads with strength to add the parameter category by considering functors $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$ with some natural transformations. Lifting of parameterised computations to larger contexts is handled by a pair of natural transformations:

$$\mu_{\otimes S, S_1, S_2, A} : T(S_1, S_2, A) \to T(S_1 \otimes S, S_2 \otimes S, A)$$

$$\mu_{S \otimes, S_1, S_2, A} : T(S_1, S_2, A) \to T(S \otimes S_1, S \otimes S_2, A)$$

As well as interpreting λ_{inplc} in categories with closed commutative versions of this structure, we also give a direct calculus, the Typed Command Calculus.

This calculus directly matches the structure of the model and does not suppose commutativity. Judgements of the Typed Command Calculus have the form:

$$\Gamma; \Delta \vdash c : A; S$$

where Γ and A are a value context and type respectively, and Δ and S are a permission, or state, context and type. Permission, or state, contexts are disallowed from using Weakening and Contraction, but value contexts have access to all the substructural rules. The Typed Command Calculus is defined and proven complete for our categorical models in Chapter 6.

We also give a direct semantics of λ_{inplc} in order to get a complete class of models for the calculus. The structure required, which we will call In-place Update Categories, is introduced in Chapter 8. The structures described in the previous section will be instances of the structure. In-place Update Categories make no mention of the permissions component of λ_{inplc} types, they just embed value (state-free) types in a larger category of potentially stateful types.

We also give an instance of an In-place Update category based on a possible worlds semantics, using Day's construction, that demonstrates how the imperative semantics relies on non-aliasing and also how it enables heap-bounded, resource constrained execution, thus giving a semantics of LFPL.

1.4 Thesis Outline

The order of presentation in this thesis mostly follows the order of this introductory chapter. Our type theory for separation, λ_{sep}, is presented with its models in Chapters 2, 3 and 4. Some of the work in these three chapters has been previously published in [Atk04]. The presentation of work on the in-place update calculus, λ_{inplc}, is presented in a slightly different order. We present our extensions of Freyd categories and Computational monads in Chapter 5, the calculus corresponding directly to these constructions in Chapter 6 and the presentation of λ_{inplc} itself and its semantics in Chapter 8.

A short synopsis of each chapter is given here:

Chapter 2: Syntax and Typing of λ_{sep} This chapter formally introduces the syntax and typing of λ_{sep}. We describe the formal representation of separation via separation relations and use them to give the typing rules. The notion of valid structural transition is defined which simplifies the manipulation of typing derivations containing term-syntax-free structural rules. A substitution lemma is proven along with several other admissible and derived rules. Translations from the simply-typed λ-calculus and the $\alpha\lambda$-calculus are given. The equational theory is also described, and proven to be well-defined. Finally, we give a type-checking algorithm for explicitly typed terms, demonstrating the use of structural transitions.

Chapter 3: Categorical Semantics of λ_{sep} This chapter describes the categorical structure required to model λ_{sep}. After recalling the definition of symmetric monoidal closed structure, used to model the $\alpha\lambda$-calculus, we then define the structure required to model the contexts and types of λ_{sep}, introducing the structure required for each of the structural transitions in turn and proving it coherent. We then show that the calculus as a whole is coherent with respect to the interpretation, as well as being sound and complete.

Chapter 4: Day's construction and Presheaf models We extend Day's construction of monoidal products on functor categories to the structure required for λ_{sep} described in the previous chapter. We then describe several instances of this construction: an abstract one starting from any category with all the necessary structure apart from closure, and then two constructions which illustrate the modelling of resources and their separation.

Chapter 5: Typed Computational Effects We recall the definitions of Freyd category and Strong Monad and give examples of how they model computational effects in call-by-value languages. We then present our extension, Parameterised Freyd categories and Parameterised Strong Monads. We first give the basic definitions and show they are equivalent in the closed case and then extend them to the state context lifting operations introduced

above. These extensions are again equivalent. We also describe a second form of closure that closes over contexts containing state.

Chapter 6: Typed Command Calculus We take the definitions of the previous chapter and define the Typed Command Calculus based directly upon them. We prove that we can coherently, soundly and completely model the calculus. We also describe the Alias Types system of Smith, Walker and Morrisett [SWM00] and relate it to the Typed Command Calculus.

Chapter 7: Heap Bounded State We give a concrete model of the Typed Command Calculus based on Day's construction that demonstrates non-aliasing and heap-size-bounded computation.

Chapter 8: An In-place Update Calculus: λ_{inplc} We present λ_{inplc}, sketched above, based on Wadler's system with linear and non-linear types. We give a well-defined equational theory and show that the constructions of Chapter 5 coherently and soundly model the equational theory. We also give a direct semantics of the calculus and show that it forms a complete class of models for the calculus.

Chapter 9: Conclusions We finish the thesis with a summary of the achievements and a discussion of related and future work.

There are also two appendices. Appendix A contains the details of some of the proofs of Chapter 5. Appendix B contains a theoretical justification of our definition of parameterised monad by relating them to adjunctions with parameters. We define a notion of parameterised algebra and Eilenberg-Moore category and prove that, in the case of typed side-effects, the Eilenberg-Moore category is equivalent to a natural category of typed side-effect algebras.

Chapter 2

Syntax and Typing of λ_{sep}

In this chapter we describe the syntax and typing of λ_{sep}. We start with the separation relations that will be used to record the separation constraints in the typing rules, followed by the typing rules and the equational theory. We also introduce the rules generating the valid structural transitions for λ_{sep}. They will be crucial in helping prove both syntactic properties of the calculus in this chapter and the connection between the syntax and the categorical semantics in the next. We also present translations of some other typed λ-calculi into λ_{sep} and a type checking algorithm. We finish the chapter by describing a syntax directed type-checking algorithm for explicitly typed terms.

2.1 Separation Relations

Separation relations formalise separation relationships between members of the context. We define the substitution of separation relations into one another, and state some properties of this operation. These properties establish the well-behavedness of substitution on separation relations and underlie much of the reasoning we will do with separation relations.

Definition 2.1.1 (Separation Relation) A *separation relation* of arity n is a binary, symmetric, non-reflexive relation on the set $\{0, ..., n-1\}$.

For a separation relation S, we write $|\mathrm{S}|$ for the arity of the separation relation. We 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 $x\mathrm{S}'y$. A list $[r_1, \ldots, r_k]_n$, where each r_i is of the form $x\#y$ denotes the smallest separation relation of arity n containing the pairs r_i. We will often omit the arity when the intended size is clear from the context. For a separation relation S and a permutation σ on the set $\{0, \ldots, |\mathrm{S}| - 1\}$ the notation $\sigma\mathrm{S}$ denotes the separation relation with $i\sigma\mathrm{S}j$ iff $\sigma^{-1}(i)\mathrm{S}\sigma^{-1}(j)$.

Definition 2.1.2 (Substitution of Separation Relations) For separation relations S and S′, with sizes n and n' respectively, define the operation of substitution $\mathrm{S}\{\mathrm{S}'/i\}$, where $0 \leq i \leq n-1$ as:

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

where:

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

Substitution of relations may be visualised as in Figure 2.1(a). 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^i_n 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.

A special case is that of substituting in the zero-arity relation. As shown in Figure 2.1(b), this removes the substituted-for position from the relation.

The following lemma establishes some basic properties of substitution. In particular, properties 3 and 4 ensure that if we perform two non-interfering (non-overlapping) substitutions in two different orders then we always finish in the

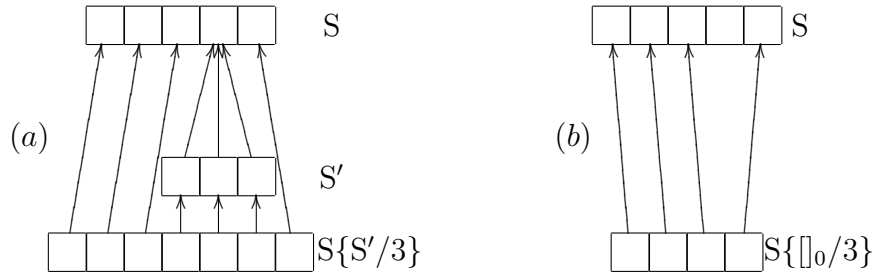


Figure 2.1: Substitution of separation relations

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. These properties are also required for the categorical coherence axioms (Definition 3.2.3) to make sense.

Lemma 2.1.3 The following properties hold, where S, S_1, S_2 are separation relations.

1. $\mathrm{S}\{\mathrm{S}_1/i\}$ is a separation relation.
2. $\mathrm{S}\{[]_1/i\} = \mathrm{S}$
3. $\mathrm{S}\{\mathrm{S}_1/i\}\{\mathrm{S}_2/j + n_1 - 1\} = \mathrm{S}\{\mathrm{S}_2/j\}\{\mathrm{S}_1/i\}$, where $i < j$
4. $\mathrm{S}\{\mathrm{S}_1\{\mathrm{S}_2/j\}/i\} = \mathrm{S}\{\mathrm{S}_1/i\}\{\mathrm{S}_2/i + j\}$

Proof Routine calculation with the definitions of substitution and norm_n^i. □

2.2 Typing Rules: λ_{sep} Systems

We describe the definition of a λ_{sep} system in two parts. First, we describe the types, contexts and valid structural transitions built from a set of primitive types. We then define a λ_{sep} system as a set of primitive types and a set of primitive typed operations that generates a typing judgement. The remainder of the section then establishes some simple meta-theoretical properties of the system and presents some derived rules, relating the calculus to similar calculi.

2.2.1 Types, Contexts and Structural Transitions

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

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

where S is a separation relation of arity $n+1$ for function types and arity n for tuple types. The extra place in the function types represents the resources used by the body of the function. Assuming a countably infinite set of variable names, ranged over by x, y, etc., the types then generate the contexts, as an instance of the structured contexts described in the introduction:

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

where the separation relation S is of arity n, and no variable x appears more than once in a context. When writing the separation relations associated with contexts we will often use variable names in place of numerical positions, where appropriate.

We also consider contexts with holes by adding the following production to the grammar:

$$\Gamma, \Delta, \Theta \quad ::= \quad -_a$$

where a is a name for the hole, no hole name may appear more than once in a context. We will write a context with a hole as $\Gamma(-)_a$, explicitly naming the hole, or as $\Gamma(-)$ when there is a single hole. The notation $\Gamma(\Delta)$ denotes a context with a hole filled in with another context Δ. We will also use the notation $\Gamma()$ to indicate the "removal" of a hole by substituting the empty context $[]()$ into the hole's position in the parent context in the tree, or by replacing the hole by the empty context when there is no parent context.

We define $v(\Gamma)$ to be the list of variable and hole names in Γ built from a depth-first, left-to-right traversal. We write $\Gamma \equiv_\alpha \Gamma'$ when Γ and Γ' are equivalent up to renaming of variables. Also, we define $a(\Gamma)$ to be the set of type assignments in Γ: that is, all the $x : A$ pairs in Γ. We define $\Gamma[x]$ to be the type of the variable x in Γ, if it exists.

A context Γ determines a separation relation S_Γ on the variables it contains by substituting out all the nested separation relations. When Γ is $x : A$ this is defined to be $[]_1$. By Lemma 2.1.3, it is easy to see that the order of substitutions used to obtain S_Γ does not matter; we will always get the same relation. We will often use S_Γ as a relation between variables; the connection between variable names and positions is determined by the $v(-)$ function above.

A structural transition is a triple $\Gamma \overset{\rho}{\Rightarrow} \Delta$, where Γ and Δ are contexts (possibly with holes) and ρ is a mapping of variables in Δ to variables in Γ (and possibly holes in Δ to holes in Γ). If ρ is omitted it stands for the identity renaming. The rules in Figure 2.2 define a judgement $\Gamma \overset{\rho}{\Rightarrow} \Delta$ *valid* which identifies a subset of the structural transitions which are valid for λ_{sep}.

$$\frac{\rho(\Gamma') = \Gamma}{\Gamma \overset{\rho}{\Rightarrow} \Gamma' \ \mathit{valid}}\ (\textsc{Renaming}) \qquad \frac{\Gamma_1 \overset{\rho_1}{\Rightarrow} \Gamma_2 \ \mathit{valid} \qquad \Gamma_2 \overset{\rho_2}{\Rightarrow} \Gamma_3 \ \mathit{valid}}{\Gamma_1 \overset{\rho_1;\rho_2}{\Rightarrow} \Gamma_3 \ \mathit{valid}}\ (\textsc{Composition})$$

$$\frac{\overrightarrow{\Gamma \overset{\rho}{\Rightarrow} \Delta \ \mathit{valid}}}{\mathrm{S}(\overrightarrow{\Gamma}) \overset{\overrightarrow{\rho}}{\Rightarrow} \mathrm{S}(\overrightarrow{\Delta}) \ \mathit{valid}}\ (\textsc{Congruence})$$

$$\frac{}{\mathrm{S}(\overrightarrow{\Gamma}, \mathrm{S}'(\overrightarrow{\Delta}), \overrightarrow{\Theta}) \Leftrightarrow \mathrm{S}\{\mathrm{S}'/i\}(\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Theta}) \ \mathit{valid}}\ ((\textsc{Un})\textsc{Flattening})$$

$$\frac{}{[](\Gamma) \Leftrightarrow \Gamma \ \mathit{valid}}\ (\textsc{Single}) \qquad \frac{\mathrm{S}' \subseteq \mathrm{S}}{\mathrm{S}(\overrightarrow{\Gamma}) \Rightarrow \mathrm{S}'(\overrightarrow{\Gamma}) \ \mathit{valid}}\ (\text{S-}\textsc{Weakening})$$

$$\frac{\sigma \text{ a permutation on } \{0, ..., |\mathrm{S}| - 1\}}{\mathrm{S}(\overrightarrow{\Gamma}) \Leftrightarrow \sigma\mathrm{S}(\sigma\overrightarrow{\Gamma}) \ \mathit{valid}}\ (\textsc{Permutation})$$

$$\frac{\Gamma \equiv_\alpha \Gamma'}{\Gamma \overset{[v(\Gamma') \mapsto v(\Gamma)]}{\Rightarrow} [](\Gamma, \Gamma') \ \mathit{valid}}\ (\textsc{Contraction}) \qquad \frac{}{\Gamma \Rightarrow [] \ \mathit{valid}}\ (\textsc{Weakening})$$

Figure 2.2: Valid Structural Transitions for λ_{sep}

The rules Renaming, Composition and Congruence are the standard ones for substructural type systems identified in Section 1.1.

We can justify the valid structural transition rules specific to λ_{sep} by appeal to the properties of separation. The rule (Un)Flattening expresses the fact that if a variable is declared to be separate from a group of variables, it is also separate from them all individually. The duplication of variables by Contraction is valid since the right hand side has no separation: we may duplicate values as long as we do not mind that they will share resources. The rule S-Weakening is justified by observing that if we have a context which promises more separation than we require, then we may forget about the extra separation. Transitions Weakening and Permutation are justified by the fact that we consider the underlying combination of values to be given by a normal product type. The Single transition fulfils an administrative role mediating between bare contexts and the trivial separated context. Many times it may be replaced by instances of (Un)Flattening.

The following lemma brings all these justifications together. It characterises valid structural transitions as transitions which preserve separation and types. As well as showing that our choice of rules is complete, this characterisation will be enormously useful in proving meta-theoretic properties of λ_{sep} in this chapter and proving the connection between the calculus and the categorical structure defined in the next chapter.

Lemma 2.2.1 A structural transition $\Gamma \overset{\rho}{\Rightarrow} \Delta$ is valid if and only if:

1. If $x \mathrm{S}_\Delta y$ then $\rho(x) \mathrm{S}_\Gamma \rho(y)$, and the same for holes; and
2. For all variables $x \in v(\Delta)$, $\Delta[x] = \Gamma[\rho(x)]$.

Proof We prove the forward implication by induction over the derivation of a valid structural transition. Note that all the rules preserve the types of variables, and the Contraction rule, the only way of introducing maps that take two variables to a single variable ensures that they are not required to be separate.

For the converse, we construct a canonical derivation of validity for a structural transition $\Gamma \overset{\rho}{\Rightarrow} \Delta$ that matches the specification given in the lemma statement.

We do this in stages, composed with Composition and Congruence:

1. For each variable x in Γ generate, in place of x, a sub-context depending on $\rho^{-1}(x)$: if $\rho^{-1}(x) = \emptyset$ then use Weakening to generate $[]_0()$; otherwise repeatedly use Contraction (and possibly Renaming) to generate a context of the form $[](x : A, [](\ldots))$, with the names and number of copies of x dictated by $\rho^{-1}(x)$. Combine all these transitions with the Congruence rule to produce a valid structural transition from Γ to a context with the same variables as Δ, and a renaming action equal to ρ.

2. Repeatedly use Flatten and possibly Single to produce the fully flattened context;

3. Apply S-Weak and Perm in the empty context to give the context the separation relation S_Δ;

4. Repeatedly apply Unflatten and possibly Single to produce the structure of Δ.

Note that the final three stages produce valid structural transitions that do not do any renaming, hence by Composition the structural transition $\Gamma \overset{\rho}{\Rightarrow} \Delta$ is valid. $\square$

Using these contexts and structural rules we can simulate the bunches of the affine $\alpha\lambda$-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 Flatten and its inverse:

$$\mathrm{S}(\mathrm{S}(\Delta_1, \Delta_2), \Delta_3) \Leftrightarrow \mathrm{S}\{\mathrm{S}/0\}(\Delta_1, \Delta_2, \Delta_3) = \mathrm{S}\{\mathrm{S}/1\}(\Delta_1, \Delta_2, \Delta_3) \Leftrightarrow \mathrm{S}(\Delta_1, \mathrm{S}(\Delta_2, \Delta_3))$$

where $\mathrm{S} = []$ or $\mathrm{S} = [0\#1]$. Since we have S-Weak and Weak we are simulating the affine $\alpha\lambda$-calculus.

In the affine $\alpha\lambda$-calculus, the (appropriately restricted form of) S-Weakening is derivable from Contraction and Weakening because the two unit contexts are identified. In terms of the translation from $\alpha\lambda$ contexts to λ_{sep} contexts

described above, the derivation is:

$$\begin{array}{rcl}
[1\#2](x:A, y:B) & \Rightarrow & [](([1\#2](x:A, y:B), [1\#2](x':A, y':B)) \\
& \Rightarrow & [](([1\#2](x:A, []()), [1\#2]([](), y':B)) \\
& \Rightarrow & [](x:A, y':B) \\
& \Rightarrow & [](x:A, y:B)
\end{array}$$

However, when the targeted context has a separation structure not expressible in the form $[]_2(\Gamma_1, \Gamma_2)$ this scheme fails. An example is:

$$\begin{array}{rl}
& [1\#2, 2\#3, 3\#4, 1\#4](w:A, x:B, y:C, z:D) \\
\Rightarrow & [1\#2, 2\#3, 3\#4](w:A, x:B, y:C, z:D)
\end{array}$$

2.2.2 Terms and Typing Judgements

A λ_{sep} *system* is a pair $(\mathcal{T}, \Phi)$ of primitive types and primitive operations $f : A \to B$ over the types generated from $\mathcal{T}$. A system generates a valid structural transition judgement by the rules in Figure 2.2 and a typing judgement by the rules in Figure 2.3 over the terms generated by this grammar:

$$\begin{array}{rcl}
e & ::= & x \\
& | & \lambda_{\mathrm{S}}(x_1 : A_1, \ldots, x_n : A_n).e \quad | \quad e@_{\mathrm{S}}(e_1, \ldots, e_n) \\
& | & \mathrm{S}(e_1, \ldots, e_n) \quad | \quad \text{let } \mathrm{S}(x_1, \ldots, x_n) = e_1 \text{ in } e_2 \\
& | & f\ e
\end{array}$$

where $f \in \Phi$.

We will write $\Phi(A, B)$ for the subset of Φ of the form $f : A \to B$.

By reading the contexts as representing the resources used by the term we obtain an informal justification of the typing rules. The rule SI 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, SE, exploits the structure of the contexts. The position of the hole in $\Delta(-)$ indicates the relationships

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

$$\frac{\overrightarrow{\Gamma \vdash e : A}}{\mathrm{S}(\overrightarrow{\Gamma}) \vdash \mathrm{S}(\overrightarrow{e}) : \mathrm{S}(\overrightarrow{A})}\ (\mathrm{SI}) \qquad \frac{\Gamma \vdash e : \mathrm{S}(\overrightarrow{A}) \qquad \Delta(\mathrm{S}(\overrightarrow{x : A})) \vdash e' : B}{\Delta(\Gamma) \vdash \text{let } \mathrm{S}(\overrightarrow{x}) = e \text{ in } e' : B}\ (\mathrm{SE})$$

$$\frac{\mathrm{S}(\Gamma, \overrightarrow{x : A}) \vdash e : B}{\Gamma \vdash \lambda_{\mathrm{S}}(\overrightarrow{x}).e : \overrightarrow{A} \xrightarrow{\mathrm{S}} B}\ (\xrightarrow{\mathrm{S}}\mathrm{I}) \qquad \frac{\Gamma \vdash e_1 : \overrightarrow{A} \xrightarrow{\mathrm{S}} B \qquad \overrightarrow{\Delta \vdash e_2 : A}}{\mathrm{S}(\Gamma, \overrightarrow{\Delta}) \vdash e_1 @_{\mathrm{S}} (\overrightarrow{e_2}) : B}\ (\xrightarrow{\mathrm{S}}\mathrm{E})$$

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

Figure 2.3: Typing Rules of a system $(\mathcal{T}, \Phi)$

that the resources used by the variables x_i must have with the rest of Δ; by substituting Γ directly into this hole we are maintaining the same relationships.

The rules $\xrightarrow{\mathrm{S}}$I and $\xrightarrow{\mathrm{S}}$E can be understood similarly; in the introduction rule we have the nested sub-context Γ 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.

The Prim rule incorporates the set of primitive operations Φ. We assume that primitive operations consume no resources themselves.

2.2.3 Basic Metatheory

We now present some basic properties of λ_{sep}. We first concentrate on properties of structural transitions, and then on derived and admissible typing rules.

A direct consequence of Lemma 2.2.1 is that this generalised form of Contraction

is valid:

$$\frac{\neg(x \mathrm{S}_\Gamma y)}{\Gamma(x:A)() \overset{[y \mapsto x]}{\Rightarrow} \Gamma(x:A)(y:A)}$$

where we have confused the variables x and y in $\Gamma(x:A)(y:A)$ and the holes they are substituted for. Variables in λ_{sep} contexts may be contracted exactly when they are not required to be separate.

For reasoning about the use of valid structural transitions in type derivations, we will need the following "factorisation" lemma that takes a valid structural transition whose codomain is the final context of a typing rule application and information about how the variables of the domain are distributed in the final term and splits the structural transition into parts that must go after the application of the typing rule, and parts that may go before. Using this lemma we will rewrite derivation trees containing structural rules to a normal form and so be able to prove coherence of the categorical interpretation (Theorem 3.3.4).

Lemma 2.2.2 The following two "factorisation" properties hold for valid structural transitions:

1. Given a valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \mathrm{S}(\Gamma_1, \ldots, \Gamma_n)$ and a mapping δ from $v(\Gamma)$ to subsets of $\{1, \ldots, n\}$ such that for all x, $i \in \delta(x)$ implies $\rho(y) = x$ for some y in Γ_i, there exist contexts $\Delta_1, \ldots, \Delta_n$ and valid structural transitions:

$$\Gamma \overset{\alpha}{\Rightarrow} \mathrm{S}(\Delta_1, \ldots, \Delta_n) \qquad\qquad \Delta_i \overset{\beta_i}{\Rightarrow} \Gamma_i$$

   Such that for all x, $\rho(e) = \alpha((\beta_1 \cup \ldots \cup \beta_n)(x))$ and for all $x \in v(\Gamma)$ and $y \in \alpha^{-1}(x)$, $y \in v(\Delta_i)$ iff $i \in \delta(x)$. Also, α and the Δ_is are determined solely by Γ, δ and S.

2. Given a valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \Gamma_1(\Gamma_2)$ and a mapping δ from $v(\Gamma)$ to subsets of $\{1, 2\}$ such that for all x, $i \in \delta(x)$ implies $\rho(y) = x$ for some y in Γ_i, there exist contexts Δ_1 and Δ_2 and valid structural transitions:

$$\Gamma \overset{\alpha}{\Rightarrow} \Delta_1(\Delta_2) \qquad\qquad \Delta_1(-) \overset{\beta_1}{\Rightarrow} \Gamma_1(-) \qquad\qquad \Delta_2 \overset{\beta_2}{\Rightarrow} \Gamma_2$$

Such that for all x, $\rho(e) = \alpha((\beta_1 \cup \beta_2)(e))$ and for all $x \in v(\Gamma)$ and $y \in \alpha^{-1}(x)$, $y \in v(\Delta_i)$ iff $i \in \delta(x)$. Also, α and the Δ_is are determined solely by Γ and δ.

Proof Construct the contexts Δ'_i as having variables z_i^x for each $x \in v(\Gamma)$ such that $i \in \delta(x)$ and separation relations $z_i^x \mathrm{S}_{\Delta'_i} z_i^y$ iff $x \mathrm{S}_\Gamma y$. Set $\alpha(z_i^x) = x$ and $\beta_i(x) = z_i^{\rho(x)}$. They are valid since ρ is. Since α and β are constructed from ρ they have the same renaming. Property 2 is proven similarly. □

We can derive more usual rules for products and functions in the case of total non-separation:

$$\frac{\overrightarrow{\Gamma \vdash e : A}}{\overrightarrow{\Gamma} \vdash []_{\mathrm{n}}(\overrightarrow{e}) : \overrightarrow{A}} \qquad \frac{\Gamma \vdash e : []_n(\overrightarrow{A}) \qquad 1 \leq i \leq n}{\Gamma \vdash \pi_i(e) : A_i}$$

$$\frac{\Gamma \vdash f : \overrightarrow{A} \xrightarrow{[]_{\mathrm{n}}} B \qquad \Gamma \vdash a_1 : A_1 \qquad \ldots \qquad \Gamma \vdash a_n : A_n}{\Gamma \vdash f @_{[]_{\mathrm{n}}}(\overrightarrow{a}) : B}$$

where $\pi_i(e)$ is defined as let $[]_{\mathrm{n}}(x_1, \ldots, x_n) = e$ in x_i.

Substitution is an admissible rule of λ_{sep}. Before we prove this, we need the following lemma that details how substitution interacts with the rule Struct:

Lemma 2.2.3 Assume:

- Typed variables $x_1 : A_1, ..., x_n : A_n$ and contexts $\Delta_1, ..., \Delta_n$;
- For each x_i, zero or more variables $y_1^i, ..., y_{k_i}^i$;
- A valid structural transition $\Gamma_1(x_1 : A_1)...(x_n : A_n) \overset{\alpha}{\Rightarrow} \Gamma_2\overrightarrow{(y^1 : A_1)}...\overrightarrow{(y^n : A_n)}$ such that for all i, j, $\alpha(y_j^i) = x_i$ and for all i, j_1, j_2, $\neg(y_{j_1}^i \mathrm{S}_{\Gamma_2} y_{j_2}^i)$.

For each y_j^i generate a new context from Δ_i called Δ_j^i by renaming the variables so there is a valid structural transition $\Delta_i \overset{\rho_j^i}{\Rightarrow} \Delta_j^i$ such that ρ_j^i is a bijection. Then there is a valid structural transition $\Gamma_1(\Delta_1)...(\Delta_n) \overset{\beta}{\Rightarrow} \Gamma_2\overrightarrow{(\Delta^1)}...\overrightarrow{(\Delta^n)}$ such that for all i, j and $z \in v(\Delta_j^i)$, $\beta(x) = \rho_i(z)$ and for all $z \in v(\Gamma_2(-))$, $\beta(z) = \alpha(z)$.

Proof Simply define β to have the same action as α on variables in $\Gamma_2(-)$ and the action of ρ_j^i for variables in Δ_j^i. The fact that for all i, j_1, j_2, $\neg(y_{j_1}^i \mathrm{S}_{\Gamma_2} y_{j_2}^i)$ implies that this gives a valid structural transition. □

Lemma 2.2.4 (Substitution) The following rule is admissible:

$$\frac{\Gamma\overrightarrow{(x : A)} \vdash e : B \qquad \overrightarrow{\Delta \vdash e : A}}{\Gamma\overrightarrow{(\Delta)} \vdash e \left[\overrightarrow{e/x}\right] : B} \text{ (Subst)}$$

Proof By induction on the derivation of $\Gamma\overrightarrow{(x : A)} \vdash e : B$:

Id Trivial.

Struct The derivation ends in the form:

$$\frac{\begin{array}{c}\Gamma_2\overrightarrow{(y^i : A_i)}...\overrightarrow{(y^n : A_n)} \vdash e' : B \\ \Gamma_1(x_1 : A_1)...(x_n : A_n) \overset{\alpha}{\Rightarrow} \Gamma_2\overrightarrow{(y^i : A_i)}...\overrightarrow{(y^n : A_n)}\end{array}}{\Gamma_1(x_1 : A_1)...(x_n : A_n) \vdash \alpha(e') : B}$$

where $\alpha(y^i_j) = x_i$ for all y^i_j. Applying Lemma 2.2.3 with α and $\overrightarrow{\Delta}$, we get contexts Δ^i_j, renamings $\Delta_i \overset{\rho_i}{\Rightarrow} \Delta^i_j$ and a valid structural transition β. Rename each of the judgements $\Delta_i \vdash e_i : A$ to get judgements $\Delta^i_j \vdash e^i_j : A_i$ such that $e_i = \rho^i_j(e^i_j)$. Apply the induction hypothesis to get a judgement:

$$\Gamma_2\overrightarrow{(\Delta^1)}...\overrightarrow{(\Delta^n)} \vdash e'[\overrightarrow{e^1/y^1}, ..., \overrightarrow{e^n/y^n}] : B$$

Applying Struct with the valid structural transition β gives:

$$\Gamma_1(\Delta_1)...(\Delta_n) \vdash \beta(e'[\overrightarrow{e^1/y^1}, ..., \overrightarrow{e^n/y^n}]) : B$$

By the properties of β guaranteed by Lemma 2.2.3, $\beta(e'[\overrightarrow{e^1/y^1}, ..., \overrightarrow{e^n/y^n}]) = \alpha(e')[e_1/x_1, ..., e_n/x_n]$, as required.

SI The derivation ends in a rule application of the form:

$$\frac{\Gamma_1\overrightarrow{(x_1 : A_1)} \vdash f_1 : A_1 \qquad \ldots \qquad \Gamma_n\overrightarrow{(x_n : A_n)} \vdash f_n : A_n}{\mathrm{S}(\Gamma_1\overrightarrow{(x_1 : A_1)}, \ldots, \Gamma_n\overrightarrow{(x_n : A_n)}) \vdash \mathrm{S}(f_1, \ldots, f_n) : \mathrm{S}(A_1, \ldots, A_n)}$$

Where each of the $\overrightarrow{(x_i : A_i)}$ are distinct by the definition of contexts. Apply the induction hypothesis to each of the premises; obtaining judgements $\Gamma_i\overrightarrow{(\Delta_i)} \vdash f_i \left[\overrightarrow{e_i/x_i}\right] : A_i$. The result then follows using the SI rule.

SE, $\overset{\mathrm{S}}{\longrightarrow}$I, $\overset{\mathrm{S}}{\longrightarrow}$E, Prim All these cases are similar to the previous case. □

Lemma 2.2.5 (Strengthening) If $\Gamma\overrightarrow{(x : A)} \vdash e : B$ and the $\overrightarrow{x}$ do not appear free in e then $\Gamma\overrightarrow{()} \vdash e : B$.

Proof By induction on the derivation of $\Gamma(x : A) \vdash e : B$. The only way to gain superfluous variables is via Struct(Weak). □

Using Lemma 2.2.1 makes it possible to easily prove convenient inversion principles for the calculus:

Lemma 2.2.6 (Inversion) The following inversion properties hold:

1. Given a derivation of $\Gamma \vdash \mathrm{S}(e_1, \ldots, e_n) : \mathrm{S}(A_1, \ldots, A_n)$ there exist derivations of the judgements $\Gamma_i \vdash e'_i : A_i$, $1 \leq i \leq n$, and a valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \mathrm{S}(\Gamma_1, \ldots, \Gamma_n)$ such that for all i, $\rho(e'_i) = e_i$.

2. Given a derivation of $\Gamma \vdash \lambda^{\mathrm{S}}(\overrightarrow{x}).e : \overrightarrow{A} \xrightarrow{\mathrm{S}} B$ there exists a derivation of the judgement $\mathrm{S}(\Gamma', \overrightarrow{x : A}) \vdash e' : B$ and a valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \Gamma'$ such that $\rho(e') = e$.

Proof Property (1) — the second is similar — is shown by induction on the height of the derivation. By the form of the term the only possible rules are SI and Struct: in the case of SI the property is satisfied with the trivial valid structural transition; in the case of Struct we apply the induction hypothesis and extend the resulting valid structural transition by the current structural rule. □

2.3 Example: Independence of Data

As an example let us consider a primitive for analysing statistical data. This analysis takes two items of data and returns a result. The analysis requires that the data come from independent sources to be statistically valid; we capture this constraint in the type of the analysis operation:

$$\texttt{analyse} : [1\#2](D, D) \to R$$

This operation is expressible in the $\alpha\lambda$-calculus by typing it so:

$$\texttt{analyse} : D * D \to R$$

Now consider three such analyses to be run over four items of data. It does not matter if we use the same data twice in two analyses, only that each analysis must operate on independent data. This can be represented as:

$$(\texttt{analyse}\ [1\#2](a,b), \texttt{analyse}\ [1\#2](b,c), \texttt{analyse}\ [1\#2](c,d)) : [](R,R,R)$$

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 λ_{sep} the context can be written as:

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

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:

$$((a : D; d : D), b : D, c : D)$$

where ';' represents possible sharing, and ',' no sharing. However, this requires that a and c be separate, whereas λ_{sep} does not require this. By Fact 1.2.1 this separation is not representable at all in the $\alpha\lambda$-calculus.

2.4 Equational Rules: λ_{sep} Theories

A λ_{sep} *theory* is a triple $(\mathcal{T}, \Phi, \Sigma)$ where $(\mathcal{T}, \Phi)$ is a λ_{sep} system and Σ is a collection of axioms of the form $\Gamma \vdash e = e' : A$ where $\Gamma \vdash e : A$ and $\Gamma \vdash e' : A$. A theory generates an equational judgement $\Gamma \vdash e_1 = e_2 : A$ by the rules in Figure 2.4.

The first two rules state that the equality judgements include the equality axioms and that equality is preserved by the structural rules. The remaining four rules are the β and η rules for products and functions. The premises for the β rules are more restrictive than the usual presentation, but Lemma 2.4.2 below shows that the usual rules are derivable. The η rule for functions is standard.

$$\frac{(\Gamma \vdash e = e' : A) \in \Sigma}{\Gamma \vdash e = e' : A}\ (\textsc{Eq-Ax}) \qquad \frac{\Gamma' \vdash e = e' : A \qquad \Gamma \overset{\rho}{\Rightarrow} \Gamma'}{\Gamma \vdash \rho(e) = \rho(e') : A}\ (\textsc{Eq-Struct})$$

$$\frac{\overrightarrow{\Delta \vdash e : A} \qquad \Gamma(\mathrm{S}(\overrightarrow{x : A})) \vdash f : B}{\Gamma(\mathrm{S}(\overrightarrow{\Delta})) \vdash (\text{let } \mathrm{S}(\overrightarrow{x}) = \mathrm{S}(\overrightarrow{e}) \text{ in } f) = f[\overrightarrow{e/x}] : B}\ (\textsc{Eq}\text{-}\beta\text{-S})$$

$$\frac{\Delta \vdash f : \mathrm{S}(\overrightarrow{A}) \qquad \Gamma(z : \mathrm{S}(\overrightarrow{A})) \vdash e : C}{\Gamma(\Delta) \vdash (\text{let } \mathrm{S}(\overrightarrow{x}) = e \text{ in } f[\mathrm{S}(\overrightarrow{x})/z]) = f[e/z] : C}\ (\textsc{Eq}\text{-}\eta\text{-S})$$

$$\frac{\mathrm{S}(\Gamma, \overrightarrow{x : A}) \vdash f : B \qquad \overrightarrow{\Delta \vdash a : A}}{\mathrm{S}(\Gamma, \overrightarrow{\Delta}) \vdash (\lambda^{\mathrm{S}}(\overrightarrow{x}).f)@_{\mathrm{S}}(\overrightarrow{a}) = f[\overrightarrow{a/x}] : B}\ (\textsc{Eq}\text{-}\beta\text{-}\lambda)$$

$$\frac{\Gamma \vdash e : \overrightarrow{A} \xrightarrow{\mathrm{S}} B \qquad \overrightarrow{x} \notin \Gamma}{\Gamma \vdash (\lambda^{\mathrm{S}}(\overrightarrow{x}).e@_{\mathrm{S}}(\overrightarrow{x})) = e : \overrightarrow{A} \xrightarrow{\mathrm{S}} B}\ (\textsc{Eq}\text{-}\eta\text{-}\lambda)$$

Plus reflexivity, symmetry, transitivity and congruence rules.

Figure 2.4: Equational Rules

The η rule for products is taken from Ghani's rule for linear tensor products [Gha95]. This rule subsumes the traditional η conversion rule, plus the commuting conversion rules needed for typing rules with "parasitic formulae" [GLT89]. Due to the inclusion of Contraction and Weakening for $[]_2(A, B)$ products and the unit $[]_0()$, the system includes the expected extensionality rules for them as derived rules:

Lemma 2.4.1 The following rules are derivable:

$$\frac{}{c : []_2(A,B) \vdash (\text{let } []_2(a,b) = c \text{ in } a, \text{let } []_2(a,b) = c \text{ in } b) = c : []_2(A,B)} \ (\eta\text{-}\times)$$

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

Proof For the first rule, from right to left:

$$\begin{aligned} & c \\ = \ & \text{let } []_2(x,y) = c \text{ in } []_2(x,y) \\ = \ & \text{let } []_2(x,y) = c \text{ in } []_2(\text{let } []_2(a,b) = []_2(x,y) \text{ in } a, \text{let } []_2(a,b) = []_2(x,y) \text{ in } b) \\ = \ & []_2(\text{let } []_2(a,b) = c \text{ in } a, \text{let } []_2(a,b) = c \text{ in } b) \end{aligned}$$

The steps are by applications of Eq-η-S, Eq-β-S and Eq-η-S respectively. The second rule can be shown to be derivable by similar steps. □

Using the inversion rules above, the restrictive form of the β rules can be relaxed to give the more usual presentation:

Lemma 2.4.2 The following are admissible equational rules

$$\frac{\Delta \vdash \mathrm{S}(\overrightarrow{e}) : \mathrm{S}(\overrightarrow{A}) \qquad \Gamma(\mathrm{S}(\overrightarrow{x : A})) \vdash f : B}{\Gamma(\Delta) \vdash (\text{let } \mathrm{S}(\overrightarrow{x}) = \mathrm{S}(\overrightarrow{e}) \text{ in } f) = f[\overrightarrow{e/x}] : B}$$

$$\frac{\Gamma \vdash \lambda^{\mathrm{S}}(\overrightarrow{x}).e : \overrightarrow{A} \xrightarrow{\mathrm{S}} B \qquad \overrightarrow{\Delta \vdash a : A}}{\mathrm{S}(\Gamma, \overrightarrow{\Delta}) \vdash (\lambda^{\mathrm{S}}(\overrightarrow{x}).e)@_{\mathrm{S}}(\overrightarrow{a}) = e[\overrightarrow{a/x}] : B}$$

Proof In each case invert the premises by Lemma 2.2.6, apply the appropriate equational rule and then apply Struct using the valid structural transition given by the inversion. □

The equational theory always produces well-formed judgements:

Proposition 2.4.3 If $\Gamma \vdash e = e' : A$ is derivable then $\Gamma \vdash e : A$ and $\Gamma \vdash e' : A$.

Proof By induction on the derivation of $\Gamma \vdash e = e' : A$. All cases are straightforward applications of the typing rules and Lemma 2.2.4. □

2.4.1 Translations

The simply-typed λ-calculus may be translated into a subset of λ_{sep} via a map $(\cdot)^*$. For types and contexts:

$$\begin{aligned} A^* &= A, \text{ when } A \text{ atomic} \\ (A \times B)^* &= []_1(A^*, B^*) \\ (A \to B)^* &= A^* \xrightarrow{[]_2} B^* \\ I^* &= []_0() \\ (x : A)^* &= x : A^* \\ (\Gamma_1; \Gamma_2) &= []_2(\Gamma_1^*, \Gamma_2^*) \end{aligned}$$

Terms:

$$\begin{aligned} x^* &= x \\ (e_1, e_2)^* &= []_2(e_1^*, e_2^*) \\ (\pi_i e)^* &= \text{let } []_2(x_1, x_2) = e^* \text{ in } x_i \\ (\lambda x : A.e)^* &= \lambda^{[]_2}(x : A).e^* \\ (e_1 e_2)^* &= e_1^* @_{[]_2} e_2^* \end{aligned}$$

This translation has the expected properties:

Proposition 2.4.4 If $\Gamma \vdash e : A$ in the simply typed λ-calculus, then $\Gamma^* \vdash e^* : A^*$ in λ_{sep}. If $\Gamma \vdash e_1 = e_2 : A$ in the simply typed λ-calculus, then $\Gamma^* \vdash e_1^* = e_2^* : A^*$ in λ_{sep}.

Proof By induction on the derivations. □

We extend the translation to the affine $\alpha\lambda$-calculus via a mapping $(\cdot)^\dagger$ (which has the same value on simple types as $(\cdot)^*$). On types and contexts:

$$\begin{aligned}(A \star B)^\dagger &= [0\#1]_2(A^\dagger, B^\dagger)\\ (A \multimap B)^\dagger &= A^\dagger \overset{[0\#1]_2}{\longrightarrow} B^\dagger\\ (\Gamma_1, \Gamma_2)^\dagger &= [0\#1]_2(\Gamma_1^\dagger, \Gamma_2^\dagger)\end{aligned}$$

Extension to terms:

$$\begin{aligned}(e_1 \star e_2)^\dagger &= [0\#1]_2(e_1^\dagger, e_2^\dagger)\\ (\text{let } x \star y = e_1 \text{ in } e_2)^\dagger &= \text{let } [0\#1]_2(x, y) = e_1^\dagger \text{ in } e_2^\dagger\\ (\lambda^\star x : A.e)^\dagger &= \lambda^{[0\#1]_2}(x : A).e^\dagger\\ (e_1 @ e_2)^\dagger &= e_1^\dagger @_{[0\#1]_2} e_2^\dagger\end{aligned}$$

We have written the term constructor of substructural function type of the $\alpha\lambda$-calculus as $\lambda^\star$ and for the simply typed function as λ rather than the λ and α used in [O'H03]. This translation also has the expected properties:

Proposition 2.4.5 If $\Gamma \vdash e : A$ in the $\alpha\lambda$-calculus, then $\Gamma^\dagger \vdash e^\dagger : A^\dagger$ in λ_{sep}. If $\Gamma \vdash e_1 = e_2 : A$ in the $\alpha\lambda$-calculus, then $\Gamma^\dagger \vdash e_1^\dagger = e_2^\dagger : A^\dagger$ in λ_{sep}.

Proof By induction on the derivations. □

2.5 Type Checking Algorithm

We now describe an algorithm for determining whether, for a given term e and set of type assignments V, there is a type derivation $\Gamma \vdash e : A$ such that $a(\Gamma) \subseteq V$. This is again complicated by the existence of syntax-free structural rules; the typing rules as given above do not determine exactly where structural rules must be inserted to give a derivation. We prove that there is an algorithm that soundly and completely determines the typability of λ_{sep} terms. Moreover, the typing context generated by the algorithm is minimal in a certain sense.

The algorithm operates by applying typing rules according to the structure of the term. When a typing rule does not immediately apply we try to bridge the gap by inserting instances of structural rules. This occurs in three ways:

1. Rules with multiple premises (SI, SE and $\overset{\mathrm{S}}{\longrightarrow}$E) may not always be applicable because their prospective antecedents may have shared variable names. In this case we define an operation merge that takes several typing contexts that may share variables and attempts to integrate them into a single context. This corresponds to a sequence of Contr applications in a typing derivation. The operation may fail if a variable has two uses which are constrained to be separate by the term.

2. In SE we need to determine whether the variables being pattern matched can be made to obey the separation relation defined by the term syntax. Also, we may need to change the separation relation to treat all the pattern matched variables uniformly with respect to the rest of the context. This corresponds to using S-Weak and Weak and is implemented by the operation group_1.

3. Similarly, for $\overset{\mathrm{S}}{\longrightarrow}$I we need to group the non-abstracted variables into a single sub-context. This is implemented by the operation group_2.

We define the algorithm by a set of syntax directed inference rules, as shown in Figure 2.5. The rules define a judgement $V, e \Mapsto A, \Theta$, where V is a set of type assignments $x : A$ (with no repeated variable names), e is a term, A is a type and Θ is an abstract context.

Definition 2.5.1 An *abstract context* is a pair $\Theta = \langle V, \mathrm{S} \rangle$ of a set of type assignments $x : A$ or named hole $-_a$, with no repeated variable or hole names and a separation relation on the the variable and hole names.

As for concrete contexts above we write $\Theta(-)_a$ for an abstract context with a named hole. We will omit the name of the hole if it is not important. Where they have disjoint sets of variables, we define the substitution of one abstract context into a hole in another as:

$$\Theta_1(\Theta_2)_a = \langle v(\Theta_1) \cup v(\Theta_2), \mathrm{S}_{\Theta_1}\{\mathrm{S}_{\Theta_2}/-_a\} \rangle$$

$$\frac{x : A \in V}{V, x \Rightarrow A, \langle x : A, []_1 \rangle}\ (\textsc{Alg-Id}) \qquad \frac{\overrightarrow{V, e \Rightarrow A, \Theta}}{V, \mathrm{S}(\overrightarrow{e}) \Rightarrow \mathrm{S}(\overrightarrow{A}), \mathsf{merge}(\mathrm{S}(\overrightarrow{-}), \overrightarrow{\Theta})}\ (\textsc{Alg-SI})$$

$$\frac{\begin{array}{c} V, e_1 \Rightarrow \mathrm{S}(\overrightarrow{A}), \Theta_1 \\ V \cup \{\overrightarrow{x : A}\}, e_2 \Rightarrow B, \Theta_2 \qquad \mathsf{group}_1(\Theta_2, \mathrm{S}, [\overrightarrow{x : A}]) = \Theta_2'(-) \end{array}}{V, \mathrm{let}\ \mathrm{S}(\overrightarrow{x}) = e_1\ \mathrm{in}\ e_2 \Rightarrow B, \mathsf{merge}(\Theta_2'(-), \Theta_1)}\ (\textsc{Alg-SE})$$

$$\frac{V \cup \{\overrightarrow{x : A}\}, e \Rightarrow B, \Theta \qquad \mathsf{group}_2(\Theta, \mathrm{S}, [\overrightarrow{x : A}]) = \Theta'}{V, \lambda^{\mathrm{S}}(\overrightarrow{x : A}).e \Rightarrow \overrightarrow{A} \xrightarrow{\mathrm{S}} B, \Theta'}\ (\textsc{Alg-}\xrightarrow{\mathrm{S}}\textsc{I})$$

$$\frac{V, f \Rightarrow \overrightarrow{A} \xrightarrow{\mathrm{S}} B, \Theta_f \qquad \overrightarrow{V, a \Rightarrow A, \Theta}}{V, f@_{\mathrm{S}}(\overrightarrow{a}) \Rightarrow B, \mathsf{merge}(\mathrm{S}(\overrightarrow{-}), \Theta_f, \overrightarrow{\Theta})}\ (\textsc{Alg-}\xrightarrow{\mathrm{S}}\textsc{E})$$

$$\frac{V, e \Rightarrow A, \Theta \qquad f : A \longrightarrow B \in \Phi}{V, fe \Rightarrow B, \Theta}\ (\textsc{Alg-Prim})$$

Figure 2.5: Syntax-directed Typechecking Rules

For a renaming α, $\alpha\Theta$ denotes the abstract context Θ with all the variables renamed via α. When renaming a variable to the same name as an existing variable the separation relationships of the two are merged.

By Lemma 2.2.1, we can use abstract contexts as the domains and codomains of valid structural transitions.

The operation merge as described above is used to merge two abstract contexts which share variables into a single context. Sometimes this is not possible since it may require variables that are meant to be separated to be merged, hence merge is partial.

Definition 2.5.2 Define the operation $\mathsf{merge}(\Theta_1(-)_a, \Theta_2)$ as:

$$\mathsf{merge}(\Theta_1(-), \Theta_2) = \begin{cases} \textit{undefined} & \text{if } \exists x \in V \text{.s.t. } x \mathrm{S}_{\Theta_1} a \text{ or } \exists x . \Theta_1[x] \neq \Theta_2[x] \\ \alpha^{-1}(\Theta_1(\alpha\Theta_2)) & \text{otherwise} \end{cases}$$

where $V = v(\Theta_1) \cap v(\Theta_2)$ and α is a renaming making all the variables in Θ_2 disjoint from those in Θ_1.

We define the iterated merging operator $\mathsf{merge}(\Theta\overrightarrow{(-)}, \overrightarrow{\Theta'})$ by the repeated application of the unary merge operation. This is only defined if all the nested applications of merge are defined.

We prove two essential properties of merge, required for the soundness and completeness properties of the typechecking algorithm. The first property states that each application of merge is witnessed by a valid structural transition. This is required for the soundness proof. The second property states that merge does not perform any more contractions than are required. This is required for the completeness proof.

Lemma 2.5.3 The following two properties hold of merge:

1. If $\mathsf{merge}(\Theta_1(-), \Theta_2)$ is defined then there exists a valid structural transition $\mathsf{merge}(\Theta_1(-), \Theta_2) \stackrel{\rho}{\Rightarrow} \Theta_1(\alpha(\Theta_2))$ such that $\rho = \alpha^{-1}$, where α is any renaming that makes the variables in Θ_1 disjoint from the variables in Θ_2.

2. Given valid structural transitions $\Gamma \stackrel{\alpha}{\Rightarrow} \Theta_1(\Delta)$ and $\Delta \stackrel{\beta}{\Rightarrow} \Theta_2$, such that α maps all variables in Θ_1 to themselves and β does no renaming, there exists a valid structural transition $\Gamma \stackrel{\rho}{\Rightarrow} \mathsf{merge}(\alpha(\Theta_1(-)), \alpha(\Theta_2))$ such that ρ does no renaming.

Proof For property 1 define $\rho(x) = \alpha^{-1}(x)$. This preserves typings since merge is defined. If $x \mathrm{S}_{\Theta_1(\alpha\Theta_2)} y$ then $\rho(x) \mathrm{S}_{\mathsf{merge}(\Theta_1(-),\Theta_2)} \rho(y)$ also because merge is define. For property 2 we have to show that if $x \mathrm{S}_{\mathsf{merge}(\alpha(\Theta_1(-)),\alpha(\Theta_2))} y$ then $x \mathrm{S}_\Gamma y$. This follows by the fact that merge introduces no extra separation over Θ_1 and Θ_2. □

The operation group_1 attempts to group variables into a sub-context with a given separation relation. It may fail in doing this if the variables are already constrained to be more separate.

Definition 2.5.4 Define the operation $\mathsf{group}_1(\Theta, \mathrm{S}, [\overrightarrow{x : A}])$ on an abstract context, separation relation and a list of type assignments as:

$$\mathsf{group}_1(\Theta, \mathrm{S}, [\overrightarrow{x : A}]) = \begin{cases} \textit{undefined} & \text{if } \exists x_i, x_j . x_i \mathrm{S}_\Theta x_j \text{ and } \neg(x_i \mathrm{S} x_j) \\ \langle \Theta \backslash \{\overrightarrow{x} : A\} \cup \{-_a\}, \mathrm{S}_{\Theta'} \rangle & \text{otherwise} \end{cases}$$

where the hole name $-_a$ is fresh and:

$$y \mathrm{S}_{\Theta'} z \Leftrightarrow \begin{cases} \textit{never} & \text{if } y = a \text{ and } z = a \\ \exists i . y \mathrm{S}_\Theta x_i & \text{if } z = a \\ \exists i . x_i \mathrm{S}_\Theta z & \text{if } y = a \\ y \mathrm{S}_\Theta z & \text{otherwise} \end{cases}$$

Similarly to merge we have two properties for group_1. The first states that there is always a valid structural transition to witness the operation of group_1. This is required for the soundness proof. The second states that group_1 is the "best" way of grouping the variables into this context. Again, this is required for the completeness proof.

Lemma 2.5.5 The following two properties hold of group_1:

1. If $\mathsf{group}_1(\Theta, \mathrm{S}, [\overrightarrow{x}]) = \Theta'(-)$ then there is a valid structural transition $\Theta'(\mathrm{S}(\overrightarrow{x : A})) \stackrel{\rho}{\Rightarrow} \Theta$.

2. Given a valid structural transition $\Gamma(\mathrm{S}(\overrightarrow{x : A})) \overset{\rho}{\Rightarrow} \Theta$ that does no renaming then, for some $\Gamma'(-)$, $\mathsf{group}_1(\Theta, \mathrm{S}, \overrightarrow{x}) = \Gamma'(-)$ and there is a valid structural transition $\Gamma(-) \overset{\rho'}{\Rightarrow} \Gamma'(-)$ that does no renaming.

Proof For property 1, define $\rho(x) = x$. This is a valid structural transition because group_1 only ever introduces separation. For property 2, group_1 is defined since, by the existence of ρ, the variables $\overrightarrow{x}$ are not too separated. Define $\rho'(x) = x$, this is a valid structural transition because group_1 only introduces separation between members and the hole where it already exists, which is already in $\Gamma(-)$. □

The last operation we need is group_2. This is similar to group_1.

Definition 2.5.6 Define the operation $\mathsf{group}_2(\Theta, \mathrm{S}, [\overrightarrow{x}])$ on an abstract context, separation relation and a set of type assignments as:

$$\mathsf{group}_2(\Theta, \mathrm{S}, [\overrightarrow{x}]) = \begin{cases} \textit{undefined} & \text{if } \exists x_i, x_j . x_i \mathrm{S}_\Theta x_j \text{ and } \neg(x_i \mathrm{S} x_j) \\ \textit{undefined} & \text{if } \exists x \in v(\Theta) \backslash \{\overrightarrow{x}\}, x_i . x \mathrm{S}_\Theta x_i \text{ and } \neg(0 \mathrm{S} x_j) \\ \langle \Theta \backslash \{\overrightarrow{x : A}\}, \mathrm{S}_\Theta \backslash \{\overrightarrow{x}\} \rangle & \text{otherwise} \end{cases}$$

Lemma 2.5.7 The following two properties hold of group_2:

1. If $\mathsf{group}_2(\Theta, \mathrm{S}, [\overrightarrow{x}]) = \Theta'$ then there is a valid structural transition $\mathrm{S}(\Theta', \overrightarrow{x : A}) \overset{\rho}{\Rightarrow} \Theta$.

2. Given a valid structural transition $\mathrm{S}(\Gamma, \overrightarrow{x : A}) \overset{\rho}{\Rightarrow} \Theta$ that does no renaming then, for some Γ', $\mathsf{group}_2(\Theta, \mathrm{S}, \overrightarrow{x}) = \Gamma'$ and there is a valid structural transition $\Gamma \overset{\rho'}{\Rightarrow} \Gamma'$ that does no renaming.

Proof Similar to Lemma 2.5.5. □

We can now prove the soundness of the typechecking algorithm, using the properties of the operations.

Theorem 2.5.8 (Soundness) If $V, e \Mapsto A, \Theta$ then $\Theta \vdash e : A$.

Proof By induction on the derivation of $V, e \Mapsto A, \Theta$.

Alg-Id In this case $\Theta = []_1(x : A)$ and $e = x$ so Id provides a derivation of $x : A \vdash x : A$.

ALG-SI By the induction hypothesis there are derivations of $\Theta_i \vdash e_i : A_i$. Rename them all to be disjoint to get derivations of $\Theta_i' \vdash e_i' : A_i$. Apply SI to get a derivation of $\mathrm{S}(\overrightarrow{\Theta'}) \vdash \mathrm{S}(\overrightarrow{e'}) : \mathrm{S}(\overrightarrow{A})$. By Lemma 2.5.3(1) there is a derivation of $\Theta \vdash e : S$, as required.

ALG-SE By the induction hypothesis there are derivations of $\Theta_1 \vdash e_1 : \mathrm{S}(\overrightarrow{A})$ and $\Theta_2 \vdash e_2 : B$. By the third premise of this rule and Lemma 2.5.5(1) there is a context Θ_2' and a derivation of $\Theta_2'(\mathrm{S}(\overrightarrow{x : A})) \vdash e_2 : B$. Renaming variables in Θ_1 and Θ_2' to make sure they are disjoint makes it possible to apply SE to get a derivation of $\Theta_2''(\Theta_1') \vdash \text{let } \mathrm{S}(\overrightarrow{x}) = e_1' \text{ in } e_2' : B$. Use Lemma 2.5.3(1) to obtain the required derivation.

ALG-$\xrightarrow{\mathrm{S}}$I By the induction hypothesis there is a derivation of $\Theta \vdash e : A$. By Lemma 2.5.7(1) there is a derivation of $\mathrm{S}(\Theta', \overrightarrow{x : A}) \vdash e : B$. Applying $\xrightarrow{\mathrm{S}}$I gives a derivation of $\Theta' \vdash \lambda^S(\overrightarrow{x : A}).e : B$ as required.

ALG-$\xrightarrow{\mathrm{S}}$E Similar to case for ALG-SI.

ALG-PRIM Apply induction hypothesis and apply PRIM. □

Completeness is more complicated. We use the factorisation lemma, Lemma 2.2.2, proven above to locally rewrite the derivation tree into a canonical form that moves contractions as far up the derivation tree as possible. This then ensures that they can be simulated by use of merge in the typechecking algorithm.

We will need the following simple weakening lemma:

Lemma 2.5.9 If $V, e \Rightarrowtriangle A, \Theta$ and $V \subseteq V'$ then $V', e \Rightarrowtriangle A, \Theta$.

Proof Induction on the derivation of $V, e \Rightarrowtriangle A, \Theta$. □

As for the proof of categorical coherence (Lemma 3.3.3) we prove a stronger property than is necessary. The extra valid structural transition allows us to do the rewriting necessary.

Lemma 2.5.10 Given a derivation of $\Gamma \vdash e : A$ and a valid structural transition $\Gamma' \overset{\rho}{\Rightarrow} \Gamma$ then there is a derivation of $a(\Gamma'), \rho(e) \Rightarrowtriangle A, \Theta$ and a valid structural transition $\Gamma' \overset{\rho'}{\Rightarrow} \Theta$, such that ρ' does no renaming.

Proof By induction on the height of the derivation of $\Gamma' \vdash e : A$. Analyse by cases on the last rule applied:

ID Derivation looks like:

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

An instance of the rule ALG-ID gives a derivation of $\{x : A\}, x \Mapsto A, \langle x : A, []_1 \rangle$. Renaming via ρ and applying Lemma 2.5.9 gives a derivation of $a(\Gamma'), \rho(x) \Mapsto A, \langle \rho(x) : A, []_1 \rangle$, and there is a valid structural transition $\Gamma' \stackrel{\rho'}{\Rightarrow} \rho(x) : A$, derived from ρ.

STRUCT The derivation ends in the form:

$$\frac{\Gamma' \vdash e : A \qquad \Gamma \stackrel{\alpha}{\Rightarrow} \Gamma' \ \mathit{valid}}{\Gamma \vdash \alpha(e) : A}$$

Extending ρ by α and applying the induction hypothesis gives a derivation of $a(\Gamma'), \rho(\alpha(e)) \Mapsto A, \Theta$ and a valid structural transition $\Gamma' \Rightarrow \Theta$, as required.

SI The derivation ends in the form:

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

Apply Lemma 2.2.2(1) to ρ to get $\Gamma' \stackrel{\alpha}{\Rightarrow} \mathrm{S}(\Delta_1, \ldots, \Delta_n)$ and $\Delta_i \stackrel{\beta_i}{\Rightarrow} \Gamma_i$. Apply the induction hypothesis to the premises and the β_is to get derivations of $a(\Delta_i), \beta_i(e_i) \Mapsto A_i, \Theta_i$ and valid structural transitions $\Delta_i \stackrel{\beta_i'}{\Rightarrow} \Theta_i$ that do no renaming. Rename via α and apply Lemma 2.5.9 to get derivations of:

$$a(\Gamma'), \alpha(\beta_i(e_i) \Mapsto A_i, \alpha(\Theta_i)$$

Apply ALG-SI to these to get a derivation of

$$a(\Gamma'), \rho(\mathrm{S}(\overrightarrow{e})) \Mapsto \mathrm{S}(\overrightarrow{A}), \mathsf{merge}(\mathrm{S}(\overrightarrow{-}), \overrightarrow{\alpha\Theta})$$

This is as required. The required valid structural transition is given by Lemma 2.5.3(2).

SE The derivation ends in the form:

$$\frac{\Gamma_1 \vdash e_1 : \mathrm{S}(A_1, \ldots, A_n) \qquad \Gamma_2(\mathrm{S}(x_1 : A_1, \ldots, x_n : A_n)) \vdash e_2 : B}{\Gamma_2(\Gamma_1) \vdash \text{let } \mathrm{S}(x_1, \ldots, x_n) = e_1 \text{ in } e_2 : B}$$

Apply Lemma 2.2.2(2) to ρ to get $\Gamma' \overset{\alpha}{\Rightarrow} \Delta_2(\Delta_1)$, $\Delta_1 \overset{\beta_1}{\Rightarrow} \Gamma_1$ and $\Delta_2(-) \overset{\beta_2}{\Rightarrow} \Gamma_2(-)$. Apply the induction hypothesis to the premises and the β_is to get derivations of:

$$a(\Delta_1), \beta_1(e_1) \Vdash \mathrm{S}(\overrightarrow{A}), \Theta_1 \qquad a(\Delta_2(\mathrm{S}(\overrightarrow{x : A}))), \beta_2(e_2) \Vdash B, \Theta_2$$

and valid structural transitions $\Delta_1 \overset{\beta_1'}{\Rightarrow} \Theta_1$ and $\Delta_2(\mathrm{S}(\overrightarrow{x : A})) \overset{\beta_2'}{\Rightarrow} \Theta_2$, both of which do no renaming. Rename via α and apply Lemma 2.5.9 to get:

$$a(\Gamma'), \alpha(\beta_1(e_1)) \Vdash \mathrm{S}(\overrightarrow{A}), \alpha(\Theta_1) \qquad a(\Gamma') \cup \{\overrightarrow{x : A}\}, \alpha(\beta_2(e_2)) \Vdash B, \alpha(\Theta_2)$$

Apply Lemma 2.5.5(2) to get a context Θ_2', a valid structural transition $\Delta_2(-) \Rightarrow \Theta_2'(-)$ and the fact that group_2 is defined. Apply Alg-SE to get:

$$a(\Gamma'), \rho(e) \Vdash A, \mathsf{merge}(\alpha(\Theta_2'(-)), \alpha(\Theta_1))$$

as required. By Lemma 2.5.3(2) the required valid structural transition exists.

$\overset{\mathrm{S}}{\longrightarrow}$I The derivation ends in the form:

$$\frac{\mathrm{S}(\Gamma, x_1 : A_1, \ldots, x_n : A_n) \vdash e : B}{\Gamma \vdash \lambda^{\mathrm{S}}(x_1 : A_1, \ldots, x_n : A_n).e : A_1, \ldots, A_n \overset{\mathrm{S}}{\longrightarrow} B}$$

We can extend ρ to a valid structural transition $\mathrm{S}(\Gamma', \overrightarrow{x : A}) \overset{\rho'}{\Rightarrow} \mathrm{S}(\Gamma, \overrightarrow{x : A})$. Apply the induction hypothesis with this to get a derivation of:

$$a(\mathrm{S}(\Gamma', \overrightarrow{x : A})), \rho(e) \Vdash B, \Theta$$

and a valid structural transition $\mathrm{S}(\Gamma', \overrightarrow{x : A}) \Rightarrow \Theta$ that does no renaming. Apply this to Lemma 2.5.7(2) to get a context Θ' and $\Gamma' \overset{\rho'}{\Rightarrow} \Theta'$. This is the required valid structural transition and an application of Alg-$\overset{\mathrm{S}}{\longrightarrow}$I gives the required derivation.

$\overset{\mathrm{S}}{\longrightarrow}$E Similar to case of SI.

Prim Follows directly by the induction hypothesis. □

Theorem 2.5.11 (Completeness) If $\Gamma \vdash e : A$ then there exists a derivation of $a(\Gamma), e \Mapsto A, \Theta$ and a valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \Theta$ that does no renaming.

Proof Instance of Lemma 2.5.10. □

Corollary 2.5.12 (Minimal Contexts) If $\Gamma \vdash e : A$ then there exists a context Γ' such that $\Gamma' \vdash e : A$ and for all Γ'' such that $\Gamma'' \vdash e : A$, there is a valid structural transition $\Gamma'' \Rightarrow \Gamma'$ that does no renaming.

Chapter 3

Categorical Semantics of λ_{sep}

In this chapter we describe the structure required of a category to soundly and coherently model λ_{sep}. Models in categories with this structure will form a complete class of models. Following the general categorical interpretation of substructural type theories sketched in Section 1.1.3 in the Introduction, we will interpret the types and contexts of λ_{sep} as objects and well-typed terms as arrows. Of particular importance is the interpretation of valid structural transitions as natural transformations.

Before defining the categorical structure we require, we recall the definition of a symmetric monoidal closed category and sketch how cartesian closed categories with additional closed symmetric monoidal structure, called Doubly Closed Categories (DCCs), are used to model the $\alpha\lambda$-calculus.

In Section 3.2 we define the structure required to model λ_{sep}. First, we define *separation products* which are used to model the contexts and separation product types, and the valid structural transitions Flatten and UnFlatten. We prove that this structure is coherent in the same sense as Mac Lane's property for symmetric monoidal structure. The structure required to interpret the rest of the rules for valid structural transitions is built up in pieces: first the S-Weakening and Permutation rules, and then the Weakening and Contraction rules. At each stage we prove that the structure is coherent, culminating in the fact that the interpretation of a valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \Delta$ is independent of its derivation, despite being defined over the structure of the derivation.

While defining the structure required we also define a category with the structure built from the judgements of λ_{sep} itself. We will use this to construct a term model and prove completeness. We also give two small examples of categories with separation products. To finish Section 3.2 we define *closure* for categories with separation products, used to interpret function types, and *separation functors*, functors that preserve separation product structure.

In Section 3.3 we define the interpretation of the typing judgements of a λ_{sep} system in a category with the structure defined in Section 3.2. We prove that this interpretation is coherent, sound and complete.

3.1 Symmetric Monoidal Structure

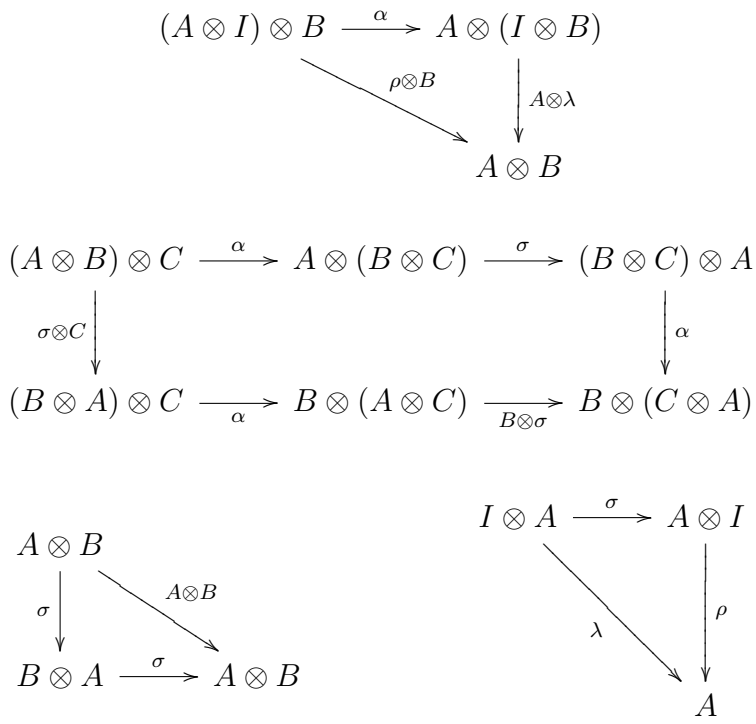
To fix notation we give the definition of symmetric monoidal structure [Mac98] on a category here:

Definition 3.1.1 Given a category $\mathcal{C}$, symmetric monoidal structure on $\mathcal{C}$ is a 6-tuple $(\otimes, I, \alpha, \lambda, \rho, \sigma)$ consisting of a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, an object I of $\mathcal{C}$ and natural isomorphisms:

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C) \qquad \lambda_A : I \otimes A \to A \qquad \rho_A : A \otimes I \to A$$
$$\sigma_{A,B} : A \otimes B \to B \otimes A$$

Subject to the following coherence diagrams:

$$\begin{array}{ccc}
((A \otimes B) \otimes C) \otimes D & \xrightarrow{\alpha \otimes D} & (A \otimes (B \otimes C)) \otimes D \\
\downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\
(A \otimes B) \otimes (C \otimes D) & & \\
\downarrow{\scriptstyle \alpha} & & \\
A \otimes (B \otimes (C \otimes D)) & \xleftarrow{A \otimes \alpha} & A \otimes ((B \otimes C) \otimes D)
\end{array}$$



If the components of α, λ, ρ and σ are all identities then this is *strict* symmetric monoidal structure.

A category with a specified symmetric monoidal structure is a *symmetric monoidal category* $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$. We will usually just refer to $\mathcal{C}$ being symmetric monoidal and leave the rest of the structure implicit. Usually they will be named as in this definition. When the symmetry natural transformation $\sigma : A \otimes B \to B \otimes A$ is missing then the structure is known as plain *monoidal structure*.

The importance of the coherence diagrams lies in the coherence theorem which is derived from them. All instances of arrows built from the natural isomorphisms of the symmetric monoidal structure and the $\otimes$ functor and composition with the same domain and codomain are equal. See Mac Lane [Mac98] and the extension to the structure required for λ_{sep} in Section 3.2.

A category with chosen finite products has symmetric monoidal structure. The bifunctor is the one derived from the $\times$ operation on objects and the unit

object is the chosen terminal object. The natural transformations are given by the appropriate combinations of pairing and projection.

Definition 3.1.2 A *closed* symmetric monoidal category $\mathcal{C}$ has, for each object A, a specified right adjoint to the functor $- \otimes A$, written $A \multimap -$.

The $\alpha\lambda$-calculus is interpreted in a category with chosen cartesian closed structure and symmetric monoidal closed structure, called a Doubly Closed Category. Contexts are interpreted using the two different products and types are interpreted in the same way, using the two different products for the two different product types and the closed structure for the two different function types.

$$[\![I]\!] = I \qquad [\![1]\!] = 1 \qquad [\![x : A]\!] = [\![A]\!] \qquad [\![\Gamma_1, \Gamma_2]\!] = [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!]$$
$$[\![\Gamma_1; \Gamma_2]\!] = [\![\Gamma_2]\!] \times [\![\Gamma_2]\!]$$

The interpretation of the rest of the $\alpha\lambda$-calculus effectively follows from this definition. See [O'H03] and [Pym02] for more details on the interpretation of the $\alpha\lambda$-calculus in Doubly Closed Categories. The structure we will present in the next section is an extension of Doubly Closed Structure.

We also recall the definition of a *symmetric monoidal functor*, a functor that preserves symmetric monoidal structure.

Definition 3.1.3 Given two symmetric monoidal categories $\mathcal{C}, \mathcal{D}$ a *symmetric monoidal functor* from $\mathcal{C}$ to $\mathcal{D}$ is a triple (F, F_1, F_2) where F is a functor $\mathcal{C} \to \mathcal{D}$, F_1 is a natural transformation $FA \otimes FB \to F(A \otimes B)$ and F_2 is an arrow $I \to FI$, such that the following diagrams commute:

$$\begin{array}{ccc}
(FA \otimes FB) \otimes FC & \xrightarrow{\alpha} & FA \otimes (FB \otimes FC) \\
\downarrow{\scriptstyle F_1 \otimes FC} & & \downarrow{\scriptstyle FA \otimes F_1} \\
F(A \otimes B) \otimes FC & & FA \otimes F(B \otimes C) \\
\downarrow{\scriptstyle F_1} & & \downarrow{\scriptstyle F_1} \\
F((A \otimes B) \otimes C) & \xrightarrow{F(\alpha)} & F(A \otimes (B \otimes C))
\end{array}
\qquad
\begin{array}{ccc}
I \otimes FA & \xrightarrow{\lambda} & FA \\
\downarrow{\scriptstyle F_2 \otimes FA} & & \uparrow{\scriptstyle F(\lambda)} \\
FI \otimes FA & \xrightarrow{F_1} & F(I \otimes A)
\end{array}$$

$$\begin{array}{ccc} FA \otimes FB & \xrightarrow{\sigma} & FB \otimes FA \\ \Big\downarrow{\scriptstyle F_1} & & \Big\downarrow{\scriptstyle F_1} \\ F(A \otimes B) & \xrightarrow{F(\sigma)} & F(B \otimes A) \end{array}$$

When F_1 and F_2 are isomorphisms then the functor is *strong*. If they are identities then the functor is *strict*.

Note that the "missing" diagram for ρ analogous to the diagram for λ follows from the diagrams for λ and σ and the axioms of a symmetric monoidal category.

3.2 Categorical Structure for λ_{sep}

We begin by introducing separation products, a generalisation of monoidal products. This gives us the basic structure required to model contexts and tuple types and the Flatten and UnFlatten structural rules. We will then consider the extra structure required to model the other structural rules. For each piece of structure we must also ensure that the appropriate interpretation is coherent.

To provide a completeness result, as well as an example of a category with the required structure, we will demonstrate each piece of structure in the term category constructed from the syntax of a theory.

Definition 3.2.1 For a λ_{sep} theory $\mathbb{T} = (\mathcal{T}, \Phi, \Sigma)$, the category $\mathbf{Tm}_{\mathbb{T}}$ has as objects the types of the theory and as arrows $A \to B$ equivalence classes of judgements $[x : A \vdash e : B]$ under the equational theory.

Identity arrows for each object A are given by $[x : A \vdash x : A]$. Composition $f; g$ of $f : A \to B$ and $g : B \to C$ is given by $[x : A \vdash g[f/x] : C]$.

Proposition 3.2.2 Definition 3.2.1 is a well-defined category.

Proof Identity is a left unit (right unit is similar):

$$[x : A \vdash x : A]; [x : A \vdash f : B] = [x : A \vdash f[x/x] : B] = [x : A \vdash f : B]$$

Composition is associative:

$$\begin{aligned}
& ([x : A \vdash f : B]; [x : B \vdash g : C]); [x : C \vdash h : D] \\
= \;& [x : A \vdash g[f/x] : C]; [x : C \vdash h : D] \\
= \;& [x : A \vdash h[g[f/x]/x] : D] \\
= \;& [x : A \vdash (h[g/x])[f/x] : D] \\
= \;& [x : A \vdash f : B]; [x : B \vdash h[g/x] : D] \\
= \;& [x : A \vdash f : B]; ([x : B \vdash g : C]; [x : C \vdash h : D]
\end{aligned}$$

where the middle line follows by usual properties of substitution. □

3.2.1 Separation Products

Definition 3.2.3 (Separation Products) For a category $\mathcal{C}$, *separation product* structure is a triple $(\underline{\mathrm{S}}, \alpha, \lambda)$ composed of a family of functors $\underline{\mathrm{S}} : \mathcal{C}^{|\mathrm{S}|} \to \mathcal{C}$, for each separation relation S and a family of natural transformations:

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

and a natural transformation:

$$\lambda_A : []_1(A) \to A$$

These must satisfy the following commutative diagrams:

1. It does not matter in which order we flatten sibling applications:

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

2. It does not matter in which order we flatten nested applications:

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

3. Flattening singleton separation products is the same as using λ:

$$\underline{\mathrm{S}}(\overrightarrow{A}, \underline{[]}(B), \overrightarrow{C}) \underset{\underline{\mathrm{S}}(\overrightarrow{A},\lambda,\overrightarrow{B})}{\overset{\alpha}{\rightrightarrows}} \underline{\mathrm{S}}(\overrightarrow{A}, B, \overrightarrow{C})$$

and

$$\underline{[]}(\underline{\mathrm{S}}(\overrightarrow{A})) \underset{\lambda}{\overset{\alpha}{\rightrightarrows}} \underline{\mathrm{S}}(\overrightarrow{A})$$

Note that in properties 1 and 2, the two different paths around the diagram, top-right and left-bottom, give the same substituted separation relations by Lemma 2.1.3.

We now prove coherence for separation products. In short, we wish to prove that any two arrows constructed from instances of α, α^{-1}, λ and λ^{-1} and composites of them with the same start and end points are equal. This is equivalent to showing that there is a unique arrow between two instances of separation product functor applications that is entirely constructed from αs, α^{-1}s, λs and λ^{-1}s. We will follow and adapt the technique of Mac Lane [Mac98] for monoidal categories.

The plan is to construct a category which forms a "model" of coherence in that each arrow of this category should give a canonical natural transformation in our target category that is equal to any other natural transformation with the same domain and codomain constructed from the λs and αs. We call this category $\mathbf{W}$:

Definition 3.2.4 (Category W) For each separation relation S define the set of *words* $\mathbf{W}_{\mathrm{S}}$ to be:

$$\mathbf{W}_{[]_0} = \{[]_0()\} \qquad \mathbf{W}_{[]_1} = \{[]_1(-), -\}$$

$$\mathbf{W}_{\mathrm{S}} = \{\mathrm{S}'(w_1, \ldots, w_n) | \exists\, \overrightarrow{\mathrm{S}}.w_i \in \mathbf{W}_{\mathrm{S_i}} \wedge \mathrm{S} = \mathrm{S}'\{\,\overrightarrow{\mathrm{S}}\,\}\},\ |\mathrm{S}| \geq 2$$

The category $\mathbf{W}$ has as objects the union of all such sets and for every $w_1, w_2 \in \mathbf{W}_{\mathrm{S}}$, for some S a unique arrow $w_1 \to w_2$.

The objects of $\mathbf{W}$ are all the possible ways of applying separation product functors, grouped by the separation relation they describe. The category $\mathbf{W}$ has

separation products. The separation product functors are defined on objects by the construction of words. The action on arrows and the αs and λs are all trivially defined by the uniqueness of arrows.

The objects of $\mathbf{W}$ abstractly represent instances of applications of separation product functors. For a category $\mathcal{C}$ with separation products, we map a word w of size n to a functor $\mathcal{C}^n \to \mathcal{C}$:

$$F([]_0()) = \star \mapsto \underline{[]_0}() \qquad F(-) = Id \qquad F(\mathrm{S}(w_1, ..., w_n)) = \underline{\mathrm{S}}(Fw_1, ..., Fw_n)$$

We can now state and prove the coherence theorem:

Theorem 3.2.5 (Coherence) For a category $\mathcal{C}$ with separation products and any arrow $w_1 \to w_2$ of $\mathbf{W}$ there is a unique natural transformation $Fw_1 \Rightarrow Fw_2$ called the canonical natural transformation, such that the identity arrow $\underline{[]_0}() \to \underline{[]_0}()$ is canonical, the identity transformation $\mathrm{id}_{\mathcal{C}} : \underline{[]_1} \Rightarrow \underline{[]_1}$ is canonical, all α, α^{-1}, λ and λ^{-1} are canonical and the composite and separation product of canonical arrows is canonical.

Proof We follow the proof of the similar proposition for monoidal categories in [Mac98]. There appear to be several choices for a natural transformation constructed from the separation product structure given an arrow of $\mathbf{W}$. We will show that they are all equal.

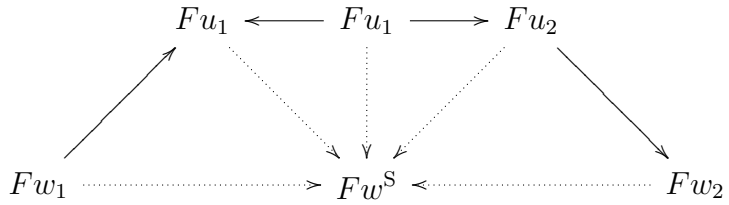
We describe the possible arrows by sequences of composable natural transformations β, constructed from αs, α^{-1}s, λs, λ^{-1}s and $\underline{\mathrm{S}}(1^a, \beta, 1^b)$-composites, such that the domain of the first arrow is Fw_1 and the codomain of the last is Fw_2. Sequences that consist solely of αs and composites of them are *flattening* paths, the those with α^{-1}s are *unflattening*.

Note that any $\underline{\mathrm{S}}(\ldots)$-nested instances of λs may be replaced by the appropriate αs, so we need only consider λs operating at the "top level" of the word structure. After this replacement, if a λ occurs in the middle of a sequence then it must be of the form $[]_1(-) \to -$ or its inverse and must be immediately followed by its inverse (to allow other components to be applied), therefore the pair may be removed, preserving the identity of the sequence. The only other possible locations for λ instances are at the start and end of the sequence, and these depend entirely on

whether the start or end words are of the form $-$. Thus, λ arrows are ignored in the rest of this proof.

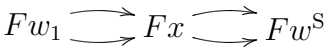
For each separation relation S there is a word w^{S} of the form $\text{S}(\overrightarrow{-})$ with a canonical flattening path from any other word with the same separation relation then applying flattenings in a left-to-right, depth-first order. With the reverse, unflattening, path this gives a canonical arrow between any two words.

The following diagram shows the components of an arbitrary sequence between the images of two words, displayed as flattening arrows. The dotted lines show the canonical paths to Fw^{S}.



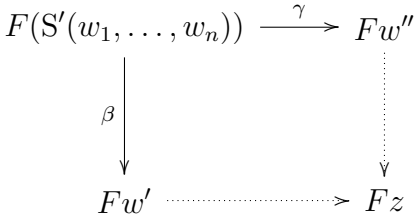
The diagram illustrates that to show that the path across the top is equal to the canonical path along the bottom it suffices to show that each of the triangles commutes.

Using the two commutative diagrams for separation products the paths can be rewritten to equal paths with all components that remove $[]_0$s at the start:



The two arrows on the left contain only $[]_0$ removals and x is u with all instances of $[]_0$ removed. All such sequences of $[]_0$-removing arrows are equal by the first commuting diagram for separation products.

The two right-hand arrows are also equal by induction on the count of nested non-$[]_1$ words, not including the outermost one. Note that every flattening arrow decreases the count. When the count is 0 or 1 there is only one possible path. For count > 1, consider branches of the form:



There are always arrows for the dotted edges, making the square commute:

$\beta = \underline{\mathrm{S}'}(1^a, \beta', 1^b)$ **:**

$\gamma = \underline{\mathrm{S}'}(1^c, \gamma', 1^d)$ **:** If $a = c$ then complete the square by induction on $|\mathrm{S}|$ (this is possible since u contains no instances of $[]_0()$). Otherwise, complete the square by functorality of $\underline{\mathrm{S}'}$.

$\gamma = \alpha$ **:** If γ flattens in position $a + 1$ and β' is an instance of α then complete the square as an instance of the second commuting diagram for separation products. Otherwise, complete the square by naturality.

$\beta = \alpha$ **:**

$\gamma = \underline{\mathrm{S}'}(1^c, \gamma', 1^d)$ **:** Similar to the second sub-case above.

$\gamma = \alpha$ **:** If γ and β operate on the same position then they are equal and the square may be completed by identity arrows. Otherwise, complete the square by the first commuting diagram for separation products.

By the induction hypothesis we know that any two paths from z are equal, hence any two paths from w_1 to w^{S} are equal. So we can define the unique canonical natural transformation to be the canonical path constructed. All of the natural transformations listed in the theorem statement are obviously canonical. □

From separation product structure we may define two distinct monoidal products.

Proposition 3.2.6 A category with separation products also has two monoidal products $\otimes$ and $\odot$, defined as:

$$A \otimes B = \underline{[]_2}(A, B) \qquad\qquad A \odot B = \underline{[1\#2]_2}(A, B)$$

sharing a unit $I = \underline{[]_0}()$. The associativity natural isomorphisms are defined as (where $\square = \otimes, \odot$):

$$A\square(B\square C) \xrightarrow{\alpha} \underline{\mathrm{S}}(A, B, C) \xrightarrow{\alpha^{-1}} (A\square B)\square C$$

where $\mathrm{S} = []_3$ or $[1\#2, 1\#3, 2\#3]_3$ as appropriate. The unit natural isomorphisms are defined as:

$$I \Box A \xrightarrow{\alpha;\lambda} A \qquad\qquad A \Box I \xrightarrow{\alpha;\lambda} A$$

Proof The required coherence diagrams hold by Theorem 3.2.5. □

Separation product structure is present in the term category:

Lemma 3.2.7 The category $\mathbf{Tm}_{\mathbb{T}}$ has separation products.

Proof Define the separation product functors as:

$$\underline{\mathrm{S}}(A_1, \ldots, A_n) = \mathrm{S}(A_1, \ldots, A_n)$$

$$\underline{\mathrm{S}}(\overrightarrow{f}) = [x : \mathrm{S}(\overrightarrow{A}) \vdash \text{let } \mathrm{S}(\overrightarrow{y}) = x \text{ in } \mathrm{S}(f_1[y_1/x], \ldots, f_n[y_n/x]) : \mathrm{S}(\overrightarrow{B})]$$

Define the instances of α and α^{-1} as (omitting the contexts and result types):

$$[\text{let } \mathrm{S}(\overrightarrow{a}, b, \overrightarrow{c}) = x \text{ in let } \mathrm{S}'(\overrightarrow{b}) = b \text{ in } \mathrm{S}\{\mathrm{S}'\}(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c})]$$

$$[\text{let } \mathrm{S}\{\mathrm{S}'\}(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}) = x \text{ in } \mathrm{S}(\overrightarrow{a}, \mathrm{S}'(\overrightarrow{b}), \overrightarrow{c})]$$

Define the instances of λ and λ^{-1} as:

$$[x : []_1(A) \vdash \text{let } []_1(z) = x \text{ in } z : A] \qquad [x : A \vdash []_1(x) : []_1(A)]$$

The fact that these obey the correct equations and are natural transformations follows by routine calculation with the equational theory. □

3.2.2 Permutation and S-Weakening

The two structural transitions Perm and S-Weak are similar in that they both operate on a single instance of a separation product – unlike (Un)Flatten – and do not duplicate or discard members – unlike Contr and Weak. Both may be treated as straightforward natural transformations and have similar diagrams for their coherence proofs.

We first define *Pre*-Permutation and *Pre*-S-Weakening structure on a category with separation products and what it means for the two structures to commute

with each other. We will then define a notion of substitution for these transitions into separation relations and vice versa, and use this to state the coherence axioms required for full Permutation and S-Weakening structure.

Definition 3.2.8 (Pre-Permutation Structure) *Pre-Permutation* for a category with separation products is a family of natural isomorphisms $\gamma[\sigma_{\mathrm{S}}]$, indexed by separation relation S and permutations σ on the set $\{0, \ldots, |\mathrm{S}| - 1\}$:

$$\gamma[\sigma_{\mathrm{S}}] : \underline{\mathrm{S}} \Rightarrow \underline{\sigma \mathrm{S}}$$

such that the following laws are obeyed:

$$\gamma[\sigma_{\mathrm{S}}]; \gamma[\sigma'_{\sigma \mathrm{S}}] = \gamma[(\sigma; \sigma')_{\mathrm{S}}] \qquad \gamma[id_{\mathrm{S}}]_A = id_A \qquad \gamma[\sigma_{\mathrm{S}}]^{-1} = \gamma[\sigma^{-1}_{\sigma \mathrm{S}}]$$

Definition 3.2.9 (Pre-S-Weakening Structure) *Pre-S-Weakening* for a category with separation products is a family of natural isomorphisms $\zeta[\mathrm{S}', \mathrm{S}]$ indexed by pairs of separation relations $\mathrm{S} \subseteq \mathrm{S}'$:

$$\zeta[\mathrm{S}', \mathrm{S}] : \underline{\mathrm{S}'} \Rightarrow \underline{\mathrm{S}}$$

such that the following laws are obeyed:

$$\zeta[\mathrm{S}, \mathrm{S}']; \zeta[\mathrm{S}', \mathrm{S}''] = \zeta[\mathrm{S}, \mathrm{S}''] \qquad \zeta[\mathrm{S}, \mathrm{S}]_A = id_A$$

Definition 3.2.10 A category with separation products, pre-permutation and pre-weakening *commutes* pre-permutation and pre-weakening if all instances of the following diagram commute:

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

The first two definitions define the basic structure required for interpreting Permutation and S-Weakening. The laws that each must obey will allow us to rewrite formal homogeneous sequences of composed permutation or S-weakening

transformations into single steps, thus providing a canonical form for such sequences. Definition 3.2.10 will allow us to move permutations and S-weakenings past each other, providing a canonical form for sequences containing both types of transformation.

To provide a canonical form for paths also involving αs and their inverses is trickier. The problem stems from the fact that the flattenings and unflattenings affect the separation relation that the transformation is operating upon and it is not immediately clear that it is always possible to preserve the meaning of a permutation or S-weakening when its separation relation is modified.

Our solution is to consider the substitution of transformations into separation relations and separation relations into transformations. Following the operation of substitution on separation relations, as represented in the category by the α natural isomorphisms, we commute transformations with flattenings by performing the appropriate substitution on the transformation. We first define an abstract notion of transformation that is suitable for formalising substitution.

Definition 3.2.11 An *abstract transformation* is a pair of separation relations S_1, S_2 such that $|\mathrm{S}_1| = |\mathrm{S}_2|$ and a bijection:

$$\square : \{0, \ldots, |\mathrm{S}_1| - 1\} \to \{0, \ldots, |\mathrm{S}_2| - 1\}$$

This is written as $\langle \mathrm{S}_1, \square, \mathrm{S}_2 \rangle$.

We identify two classes of abstract transformation that are captured by the pre-permutation and pre-S-weakening structure above. Abstract transformations that strictly preserve separation correspond to permutation natural transformations. Abstract transformations that are the identity on positions but may discard some separation correspond to S-weakening natural transformations. Given an abstract transformation of one of these classes, there is a natural transformation that models its action. We write the associated natural transformation as $\widehat{\square}$ since the start and finish separation relations will be clear from the context. Substitution of abstract transformations into separation relations and vice versa preserves these classes:

Definition 3.2.12 Given an abstract transformation $\langle \mathrm{S}_1, \Box, \mathrm{S}_2 \rangle$, a separation relation S and position i, define abstract transformations:

- $\langle \mathrm{S}\{\mathrm{S}_1/i\}, \mathrm{S}\{\Box/i\}, \mathrm{S}\{\mathrm{S}_2/i\} \rangle$, substituting $\Box$ into position i of S. Set:

$$\mathrm{S}\{\Box/i\}(x) = \begin{cases} \Box(x-i)+i & \text{if } \mathrm{norm}_n^i(x) = i \\ x & \text{otherwise} \end{cases}$$

- $\langle \mathrm{S}_1\{\mathrm{S}/i\}, \Box\{\mathrm{S}/i\}, \mathrm{S}_2\{\mathrm{S}/i\} \rangle$, substituting S into position i of $\Box$. Set:

$$\Box\{\mathrm{S}/i\}(x) = x - \mathrm{base}_n^i(\mathrm{norm}_n^i(x)) + \mathrm{base}_n^{\Box(i)}(\Box(\mathrm{norm}_n^i(x)))$$

  where:

$$\mathrm{base}_n^i(x) = \begin{cases} x & \text{if } x \leq i \\ x+n-1 & \text{if } x > i \end{cases}$$

Clearly, a permutation transformation that undergoes substitution remains a permutation transformation, as does an S-weakening transformation. We may now state the required property that a category must have to be able to commute transformations and flattening:

Definition 3.2.13 A category with separation products has *permutation structure* if it has pre-permutation structure and for each permutation transformation $\langle \mathrm{S}_1, \Box, \mathrm{S}_2 \rangle$ the following diagram commutes:

$$\begin{array}{ccc} \underline{\mathrm{S}}(\overrightarrow{A}, \underline{\mathrm{S}'}(\overrightarrow{B}), \overrightarrow{C}) & \xrightarrow{\underline{\mathrm{S}}(\ldots,\widehat{\Box},\ldots)} & \underline{\mathrm{S}}(\overrightarrow{A}, \underline{\Box\mathrm{S}'}(\Box\overrightarrow{B}), \overrightarrow{C}) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\ \underline{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}) & \xrightarrow{\widehat{S\{\Box\}}} & \underline{\mathrm{S}\{\Box\mathrm{S}'\}}(\overrightarrow{A}, \Box\overrightarrow{B}, \overrightarrow{C}) \end{array}$$

$$\begin{array}{ccc} \underline{\mathrm{S}}(\overrightarrow{A}, \underline{\mathrm{S}'}(\overrightarrow{B}), \overrightarrow{C}) & \xrightarrow{\widehat{\Box}} & \underline{\Box\mathrm{S}}(\Box(\overrightarrow{A}, \underline{\mathrm{S}'}(\overrightarrow{B}), \overrightarrow{C})) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\ \underline{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}) & \xrightarrow{\widehat{\Box\{S'/i\}}} & \underline{\Box\mathrm{S}\{\mathrm{S}'\}}(\Box\{S'/i\}(\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C})) \end{array}$$

Likewise, a category with separation products has S-*weakening structure* if it has pre-S-weakening structure and for each S-weakening transformation $\langle \mathrm{S}_1, \Box, \mathrm{S}_2 \rangle$ the above diagram commutes.

These four diagrams, along with the diagram in Definition 3.2.10, allow us to construct a canonical path between two words with separation relations S_1 and S_2 where $\mathrm{S}_2 \subseteq \mathrm{S}_1$, performing arbitrary permutations, such that any other path is equal to it. The canonical path extends the canonical path of the proof of Theorem 3.2.5 by flattening to the fully flattened word, performing the permutation, then the S-weakening and then unflattening to the final form. To formalise this we need to extend the definition of arrows in the word category $\mathbf{W}$:

Definition 3.2.14 (Category $\mathbf{W}'$) The category $\mathbf{W}'$ has as objects pairs $\langle w, \sigma \rangle$ of objects of $\mathbf{W}$ and permutations on the set $\{0, ..., |\mathrm{S}| - 1\}$ where S is the separation relation of w. It has a unique arrow $\langle w_1, \sigma_1 \rangle \to \langle w_2, \sigma_2 \rangle$ if $\mathrm{S}_{w_2} \subseteq \mathrm{S}_{w_1}$.

The objects of $\mathbf{W}'$ are also abstract functors constructed from separation products, as was the case for $\mathbf{W}$, but they may also permute their variables. Define a map from objects of size n of $\mathbf{W}'$ to functors $\mathcal{C}^n \to \mathcal{C}$ as $G'(\langle w, \sigma \rangle) = \sigma^*; Gw$ where G is as defined above and σ^* is the functor satisfying $\sigma^*(A_1, ..., A_n)_i = A_{\sigma(i)}$. By the discussion above we have the following coherence theorem for categories with separation products and permutation and S-weakening that commute:

Theorem 3.2.15 (Coherence 2) If a category $\mathcal{C}$ has separation products and permutation and S-weakening that commute, then for any arrow $f : \langle w_1, \sigma_1 \rangle \to \langle w_2, \sigma_2 \rangle$ of $\mathbf{W}'$ there is a unique natural transformation $G'(\langle w_1, \sigma_1 \rangle) \Rightarrow G'(\langle w_2, \sigma_2 \rangle)$ called the canonical natural transformation such that the identity arrow $\underline{[]_0}() \to \underline{[]_0}()$ is canonical, the identity transformation $\mathrm{id}_{\mathcal{C}} : []_1 \Rightarrow []_1$ is canonical, all α, α^{-1}, γ and ζ are canonical and the composite and S separation combination of canonical arrows is canonical.

The definitions of monoidal products on a category with separation products from Proposition 3.2.6 have extra properties when the category also has permutation and/or S-weakening:

Proposition 3.2.16 Given a category $\mathcal{C}$ with separation products take the definitions of monoidal products from Proposition 3.2.6.

1. If $\mathcal{C}$ has permutation structure then both monoidal products are symmetric, with the symmetry given by the appropriate instance of γ.

2. If $\mathcal{C}$ has S-weakening, then there is a natural transformation $A \odot B \Rightarrow A \otimes B$, given by S-weakening.

3. If $\mathcal{C}$ has S-weakening, then $\mathcal{C}$ is linearly distributive in the sense of [CS97, CS99]. The distribution natural transformations are given by the composites:

$$A \odot (B \otimes C) \cong \underline{[0\#1, 0\#2]}(A, B, C) \to \underline{[0\#1]}(A, B, C) \cong (A \odot B) \otimes C$$

$$(B \otimes C) \odot A \cong \underline{[0\#2, 1\#2]}(B, C, A) \to \underline{[1\#2]}(B, C, A) \cong B \otimes (C \odot A)$$

4. If $\mathcal{C}$ has S-weakening and permutation that commute, then the linear distribution of part 3 is symmetric in the sense of [CS97].

Proof The coherence diagrams all follow from Theorem 3.2.15. □

The term category has the all required structure, as expected:

Lemma 3.2.17 The category $\mathbf{Tm}_{\mathbb{T}}$ has permutation and S-weakening that commute.

Proof Define the permutation natural transformations as:

$$[x : \mathrm{S}(\overrightarrow{A}) \vdash \text{let } \mathrm{S}(x_1, \ldots, x_n) = x \text{ in } \sigma\mathrm{S}(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) : \sigma\mathrm{S}(\sigma\overrightarrow{A})]$$

Define the S-weakening natural transformations as:

$$[x : \mathrm{S}(\overrightarrow{A}) \vdash \text{let } \mathrm{S}(x_1, \ldots, x_n) = x \text{ in } \mathrm{S}'(x_1, \ldots, x_n) : \mathrm{S}'(\overrightarrow{A})]$$

That these definitions obey the conditions may be verified by calculation with the equational theory. □

3.2.3 Weakening and Contraction

The final piece of structure required for interpreting valid structural transitions is that for the rules Weakening and Contraction. As above we will give coherence requirements for the two natural transformations that we require and prove

that this gives coherence. The conditions we state will actually be weaker than we require to model the calculus itself, but they will be sufficient to model the contexts and valid structural transitions. We will give the complete structure in Section 3.3.

The rules Weakening and Contraction will be interpreted by two natural transformations:

$$\mathit{discard}_A : A \to \underline{[]_0}() \qquad\qquad \mathit{dup}_A : A \to \underline{[]_2}(A, A)$$

We motivate the coherence requirements we impose by considering the construction of a canonical derivation of a valid structural transition in the proof of Lemma 2.2.1. We will use this canonical derivation as the canonical path in the coherence proof below.

Recall that the canonical path is constructed from a structural transition $\Gamma \overset{\rho}{\Rightarrow} \Delta$ that preserves separation and types by first using Weakening and Contraction to match the action of the map ρ on variables. It then uses the rest of the rules to transform the resulting context to Δ. Therefore, the canonical path corresponds to the arrow:

$$\underline{\Gamma}(\overrightarrow{mk_x}); i$$

The notation $\underline{\Gamma}$ denotes a functor derived from a context Γ by the separation products with an argument for every variable. The natural transformations $mk_x : A \to \underline{[]_2}(A, ..., \underline{[]_0}())$ are composed from *dup* and *discard* to give $|\rho^{-1}(x)|$ copies of the variables position. The natural transformation i is the canonical natural transformation as defined from Theorem 3.2.15.

We need to ensure that any two constructions of mk_x are equal. We do this by requiring that, for every object A, the triple $(A, \mathit{dup}_A, \mathit{discard}_A)$ is a comonoid.

We will prove coherence by induction of an arbitrary path and show that it is equal to the canonical path. In the case of components built from any of $\alpha, \lambda, \gamma, \zeta$ the equality is covered by Theorem 3.2.15. In the case of components built from $\mathit{discard}_A$ and dup_A, we need a way to commute them with the natural transformation i. To do this we require that a single instance of w or *dup* operating on an application of a separation product functor may be decomposed into multiple

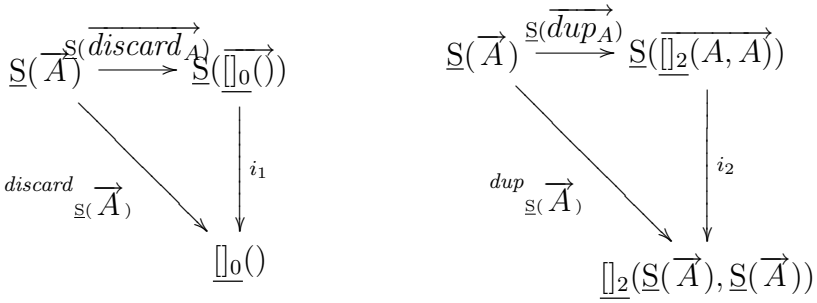
*discard*s or *dup*s operating on the constituent parts, composed with a canonical natural transformation. This ensures that we can commute post-composed instances of *discard* and *dup* to the mk_x instances that operate on individual variable positions.

We gather the requirements into a definition:

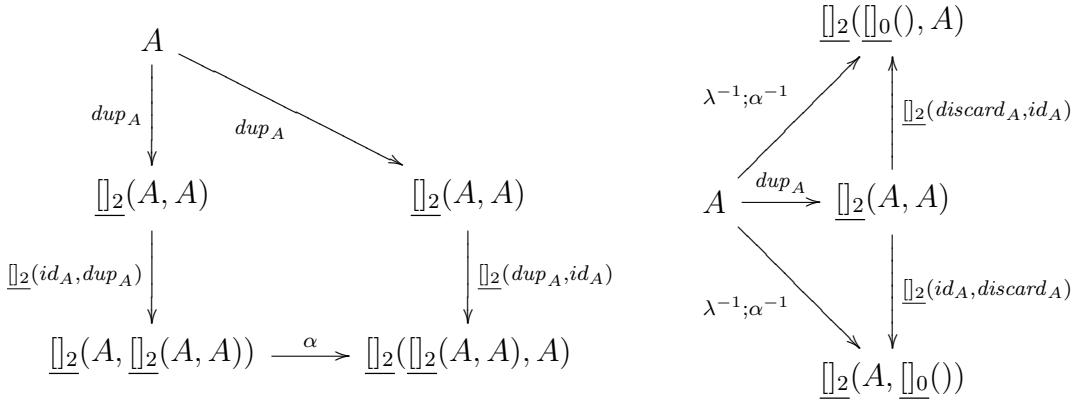
Definition 3.2.18 (Discarding and Duplication) A category with separation products and permutation and S-weakening that commute has *discarding and duplication* if it has a pair of natural transformations:

$$discard_A : A \to \underline{[]_0}() \qquad\qquad dup_A : A \to \underline{[]_2}(A, A)$$

such that the following diagrams commute. Both must preserve separation structure:



where i_1 and i_2 are the canonical natural transformations between these instances of separation product functors. Further, for each object A, the triple $(A, dup_A, discard_A)$ is a comonoid:



To show coherence we again extend the definition of word category to cope with words that may use repeated variables. The arrows of this category must obey the same separation preservation condition that we identified for valid structural transitions in Lemma 2.2.1.

Definition 3.2.19 (Category W″) The category $\mathbf{W}''$ has as objects triples $\langle w, n, f\rangle$ of objects of $\mathcal{W}$, natural numbers $n \leq |w|$ and maps $f : \{0, ..., |w|-1\} \to \{0, ..., n-1\}$. An arrow $\langle w_1, n_1, f_1\rangle \to \langle w_2, n_2, f_2\rangle$ where $n_1 = n_2$ is a map $\rho : \{0, ..., |w_1|-1\} \to \{0, ..., |w_2|-1\}$ such that $\rho(f_2(i)) = f_1(i)$ and if $i \mathrm{S}_{w_2} j$ then $\rho(i) \mathrm{S}_{w_1} \rho(j)$.

Given a category $\mathcal{C}$ with separation products We again define a map from objects of $\langle w, n, f\rangle \in \mathbf{W}''$ to functors $\mathcal{C}^n \to \mathcal{C}$ as $G''(\langle w, n, f\rangle) = f^*; Gw$ where f^* is the functor satisfying $f^*(A_1, ..., A_n)_i = A_{f(i)}$ and G is the functor defined above for plain separation products.

Theorem 3.2.20 (Coherence 3) If a category $\mathcal{C}$ has separation products, permutation and S-weakening that commute and discarding and duplication, then for any arrow $\rho : \langle w_1, n_1, f_1\rangle \to \langle w_2, n_2, f_2\rangle$ of $\mathbf{W}''$ there is a unique natural transformation $G''(\langle w_1, n_1, f_1\rangle) \Rightarrow G''(\langle w_2, n_2, f_2\rangle)$ called the canonical natural transformation such that the identity arrow $\underline{[]_0}() \to \underline{[]_0}()$ is canonical, the identity transformation $\mathrm{id}_{\mathcal{C}} : []_1 \Rightarrow []_1$ is canonical, all α, α^{-1}, γ, ζ, *discard* and *dup* are canonical and the composite and S separation combination of canonical arrows is canonical.

Proof As for Theorems 3.2.5 and 3.2.15 we define a canonical such natural transformation and prove that all other candidates are equal to it. The canonical natural transformation is defined using the construction of a derivation of a valid structural transition in Lemma 2.2.1. By the argument outlined above, it is equal to any other path constructed from the structure. □

The following proposition states what happens when some or all of the requirements for discarding and duplication are fulfilled by finite product structure.

Proposition 3.2.21 Assume a category $\mathcal{C}$ with separation products, permutation and S-weakening that commute and discarding and duplication.

1. If the object $\underline{[]_0}()$ is a choice for terminal object (so that $w_A$is the unique arrow) then the separation product preserving axioms is automatically fulfilled.

2. If $\mathcal{C}$ has products then $\underline{[]_2}(A, B)$ being a suitable choice for them is equivalent to the following diagram holding:

$$\begin{array}{ccc} \underline{[]_2}(A,B) & \xrightarrow{dup_{\underline{[]_2}(A,B)}} & \underline{[]_2}(\underline{[]_2}(A,B), \underline{[]_2}(A,B)) \\ \Big\downarrow{\scriptstyle id} & & \Big\downarrow{\scriptstyle \underline{[]_2}(\underline{[]_2}(A,discard_B),\underline{[]_2}(discard_A,B))} \\ \underline{[]_2}(A,B) & \xleftarrow{\underline{[]_2}(\alpha;\lambda,\alpha;\lambda)} & \underline{[]_2}(\underline{[]_2}(A,\underline{[]_0}()), \underline{[]_2}(\underline{[]_0}(),B)) \end{array}$$

   where the projections are defined as:

$$\pi_1 = \underline{[]_2}(A, discard_B); \alpha; \lambda \qquad \pi_2 = \underline{[]_2}(discard_A, B); \alpha; \lambda$$

   And for $f : X \to A$ and $g : X \to B$, define $\langle f, g\rangle = dup_X; \underline{[]_2}(f, g)$.

Proof Part 1 is immediate by the uniqueness of the arrows to the terminal object. For part 2, if the $\underline{[]_2}(A, B)$ is a product with the stated definitions then it the diagram certainly holds. For the converse we require this diagram and the comonoid properties of *discard* and *dup* for this to be a choice for the product. □

Again, the term category has the required structure:

Lemma 3.2.22 The category $\mathbf{Tm}_{\mathbb{T}}$ has discarding and duplication such that $\underline{[]_0}[]$ is terminal object and $\underline{[]_2}(A, B)$ is a product.

Proof The unique arrow from any object A to $\underline{[]_0}$ is defined as:

$$[x : A \vdash []_0() : []_0()]$$

This arrow is unique by the η-0 equational rule. The arrow dup_A is defined as:

$$[x : A \vdash []_2(x, x) : []_2(A, A)]$$

By calculation with the equational theory this can be seen to satisfy the equations. In particular, the derived η-$\times$ equational rule gives the surjective pairing property.

□

3.2.4 Separation Functors

We now state what it means for a functor to preserve the structure we have defined so far.

Definition 3.2.23 (Separation Functor) For two categories $\mathcal{C}$, $\mathcal{D}$ with separation products with permutation, S-weakening, discarding and duplication, a separation functor is a functor $F : \mathcal{C} \to \mathcal{D}$ and a family of natural transformations:

$$F_{\mathrm{S},A_1,\ldots,A_n} : \underline{\mathrm{S}}(\overrightarrow{F(A)}) \to F(\underline{\mathrm{S}}(\overrightarrow{A}))$$

Such that they preserve the natural transformations of the structure (where $\square \in \{\gamma, \zeta\}$ for the third diagram):

$$\begin{array}{ccc}
\underline{\mathrm{S}}(\overrightarrow{F(A)}, \underline{\mathrm{S}}'(\overrightarrow{F(B)}), \overrightarrow{F(C)}) & \xrightarrow{\alpha} & \underline{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{F(A)}, \overrightarrow{F(B)}, \overrightarrow{F(C)}) \\
\downarrow{\scriptstyle \underline{\mathrm{S}}(id,F_{\mathrm{S}'},id)} & & \\
\underline{\mathrm{S}}(\overrightarrow{F(A)}, F(\underline{\mathrm{S}}'(\overrightarrow{B})), \overrightarrow{F(C)}) & & \downarrow{\scriptstyle F_{\mathrm{S}\{\mathrm{S}'\}}} \\
\downarrow{\scriptstyle F_{\mathrm{S}}} & & \\
F(\underline{\mathrm{S}}(\overrightarrow{A}, \underline{\mathrm{S}}'(\overrightarrow{B}), \overrightarrow{C})) & \xrightarrow{F(\alpha)} & F(\underline{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}))
\end{array}$$

$$\begin{array}{ccc}
\underline{[]}(F(A)) & \xrightarrow{\lambda} & F(A) \\
\downarrow{\scriptstyle F_{[]}} & \nearrow{\scriptstyle F(\lambda)} & \\
F(\underline{[]}(A)) & &
\end{array}
\qquad
\begin{array}{ccc}
\underline{\mathrm{S}}(\overrightarrow{F(A)}) & \xrightarrow{\square} & \underline{\mathrm{S}}'(\square(\overrightarrow{F(A)})) \\
\downarrow{\scriptstyle F_{\mathrm{S}}} & & \downarrow{\scriptstyle F_{\mathrm{S}'}} \\
F(\underline{\mathrm{S}}(\overrightarrow{A})) & \xrightarrow{F(\square)} & F(\underline{\mathrm{S}}'(\square(\overrightarrow{A})))
\end{array}$$

$$\begin{array}{ccc}
FA & \xrightarrow{F(discard)} & F(\underline{[]}_0()) \\
& \searrow^{discard} & \downarrow{\scriptstyle F_{[]0}} \\
& & \underline{[]}_0()
\end{array}
\qquad
\begin{array}{ccc}
F(A) & \xrightarrow{dup} & \underline{[]}(F(A), F(A)) \\
& \searrow^{F(dup)} & \downarrow{\scriptstyle F_{[]}} \\
& & F(\underline{[]}(A, A))
\end{array}$$

A separation functor $(F, \{F_{\mathrm{S}}\})$ is a *strong separation functor* if the natural transformations F_{S} are actually isomorphisms.

The following properties of separation functors are a direct consequence of the definition:

Proposition 3.2.24 1. A separation functor $(F, \{F_{\text{S}}\}) : \mathcal{C} \to \mathcal{D}$ is also a symmetric monoidal functor (Definition 3.1.3) in the two monoidal products defined in Proposition 3.2.6 with the comparisons given by $F_{[]_0}$ in both cases for the unit and $F_{[]_2}$ and $F_{[0\#1]}$ for the two products.

2. If the discarding and duplication structure on two categories $\mathcal{C}$ and $\mathcal{D}$ is given by finite products and F preserves finite products then the final two diagrams in the definition hold automatically.

3.2.5 Two Example Separation Categories

Example 3.2.25 (Structural Morphisms) The structural morphisms of the previous chapter provide a simple example of a category with separation structure with permutation, S-weakening, weakening and duplication. This construction is less complicated than the full-blown term category construction. The objects of the category are contexts over a type language with only one type, where we identify two contexts if they are α-equivalence. The morphisms of the category are structural morphisms. That is, a morphism $\Gamma \to \Delta$ maps positions in Δ to positions in Γ, preserving separation. Clearly, we can define all the structural morphisms and they obey the coherence requirements trivially. We will call this category **SepCtxt**.

Example 3.2.26 ($\mathbf{Rel}^{\#}$) This example is a simplified version of Reddy's model of Interference Controlled Algol [Red94], we have changed from using the category of coherence spaces and linear maps as a base to just using sets and relations since we do not require fixpoints. Define the category $\mathbf{Rel}^{\#}$ as:

Objects Pairs $(X, \#_X)$ of a set X and a binary symmetric relation $\# \subseteq X \times X$;

Arrows $f : (X, \#_X) \to (Y, \#_Y)$ are relations $f \subseteq X \times Y$ such that if $(x_1, y_1) \in f$ and $(x_2, y_2) \in f$ then $y_1 \#_Y y_2$ implies $x_1 \#_X x_2$.

The intuition behind this definition is that the objects are sets of "events" with a relation that states when two events are independent. An arrow is a collection of input/output pairs (x, y); to see output event y one must observe input event x. The independence condition on arrows states that if two outputs are independent then the corresponding inputs must be independent: arrows may not introduce dependencies.

Define separation products as:

$$\underline{\mathrm{S}}((X_1, \#_{X_1}), ..., (X_n, \#_{X_n})) = (X_1 + ... + X_n, i.x \# j.y \Leftrightarrow i\mathrm{S}j \vee (i = j \wedge x \#_{X_i} y))$$

Thus, a separation product consists of tagged events from its constituent components, and two events are independent if they come from independent components (as specified by S), or if they are from the same component and are independent there.

The λ natural transformation is just the obvious isomorphism. The flattening and unflattening natural isomorphisms do the obvious renumbering: assuming objects $A_1, ..., A_{i-1}$, $B_1, ..., B_j$ and $C_1, ..., C_k$, define α to consist of components like so:

$$\begin{array}{rcl} \alpha & : & \underline{\mathrm{S}}(\overrightarrow{A}, \underline{\mathrm{S}'}(\overrightarrow{B}), \overrightarrow{C}) \to \underline{\mathrm{S}\{\mathrm{S}'/\mathrm{i}\}}(\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}) \\ & & \{(l.a, l.a) \quad | \quad 1 \leq l < i \wedge a \in A_l\} \\ \alpha & = & \cup\{(i.l.b, (i+l).b) \quad | \quad 1 \leq l \leq j \wedge b \in B_l\} \\ & & \cup\{(l.c, (l+j).c) \quad | \quad 1 \leq l \leq k \wedge c \in C_l\} \end{array}$$

This is easily seen to obey the independence condition, naturality and the two commuting diagrams. Likewise, permutation and S-weakening are the obvious renumbering and forgetful relations. The separation functor $\underline{[]_0}()$ is interpreted as the empty set, with the trivial independence relation. This is a suitable choice of terminal object in $\mathbf{Rel}^{\#}$ and so have discarding. Duplication is interpreted by the arrow:

$$\begin{array}{rcl} \mathit{dup} & : & A \to \underline{[]_2}(A, A) \\ \mathit{dup} & = & \{(a, 1.a) \quad | \quad a \in A\} \cup \{(a, 2.a) \quad | \quad a \in A\} \end{array}$$

Note that there is no similar arrow $A \to \underline{[1\#2]}_2(A, A)$. For all elements $a \in A$ we would have to have $a\#_A a$, which is not in general true. Reddy terms objects A such that $\forall a \in A.a\#_A a$ *passive* and uses them to interpret passive types in Interference Controlled Algol.

3.2.6 Separation Closure

The final property we require of a category to model λ_{sep} is that for function types. As usual this is specified as requiring the existence of specified right adjoints to existing functors.

Definition 3.2.27 A category with separation products is *separation closed* if each functor $\underline{\mathrm{S}}(-, \overrightarrow{A}) : \mathcal{C} \to \mathcal{C}$ has a specified right adjoint $[\overrightarrow{A} \xrightarrow{\mathrm{S}} -] : \mathcal{C} \to \mathcal{C}$. Call the isomorphism of homsets Λ:

$$\Lambda : \mathcal{C}(\underline{\mathrm{S}}(A, \overrightarrow{B}), C) \cong \mathcal{C}(A, [\overrightarrow{B} \xrightarrow{\mathrm{S}} C])$$

and the counit $ev_{\mathrm{S},A,B} : \underline{\mathrm{S}}([\overrightarrow{A} \xrightarrow{\mathrm{S}} B], \overrightarrow{A}) \to B$.

Lemma 3.2.28 The category $\mathrm{Tm}(\mathcal{T}, \mathcal{F}, \Phi)$ is separation closed.

Proof For each separation relation S the functor $[\overrightarrow{A} \xrightarrow{\mathrm{S}} B]$ is defined as $\overrightarrow{A} \xrightarrow{\mathrm{S}} B$ on objects and on arrows $[f_i] : B_i \to A_i$ and $[g] : C \to D$ as:

$$[x : \overrightarrow{A} \xrightarrow{\mathrm{S}} C \vdash \lambda^{\mathrm{S}}(\overrightarrow{x}).g[x@_S(f_1[x_1/x], \ldots, f_n[x_n/x])/x] : \overrightarrow{B} \xrightarrow{\mathrm{S}} D]$$

The unit and counit of the adjunction are:

$$[x : A \vdash \lambda^{\mathrm{S}}(\overrightarrow{x}).\mathrm{S}(x, \overrightarrow{x}) : \overrightarrow{A} \xrightarrow{\mathrm{S}} \mathrm{S}(A, \overrightarrow{A})]$$

$$[x : \mathrm{S}(\overrightarrow{A} \xrightarrow{\mathrm{S}} B, \overrightarrow{A}) \vdash \text{let } \mathrm{S}(f, \overrightarrow{a}) = x \text{ in } f@_{\mathrm{S}}(\overrightarrow{a}) : B]$$

These can easily be seen to obey the adjunction laws. □

3.3 Interpretation of λ_{sep}

We collect all the requirements on a category to soundly model λ_{sep} in a single definition:

Definition 3.3.1 A λ_{sep}-category is a category with separation products, permutation and S-weakening that commute, discarding and duplication given by finite products and separation closure.

Given a λ_{sep} system $(\mathcal{T}, \Phi)$ and a λ_{sep}-category $\mathcal{C}$, an *interpretation* of $(\mathcal{T}, \Phi)$ in $\mathcal{C}$ is a pair of maps $\mathcal{I} : \mathcal{T} \to \mathsf{Ob}\mathcal{C}$ and $\mathcal{I}_{A,B} : \Phi(A, B) \to \mathcal{C}(\llbracket A \rrbracket, \llbracket B \rrbracket)$ where $\llbracket \cdot \rrbracket$ is defined on types as:

$$\llbracket A \rrbracket = \mathcal{I}(A) \qquad \llbracket \mathrm{S}(A_1, \ldots, A_n) \rrbracket = \underline{\mathrm{S}}(\llbracket A_1 \rrbracket, \ldots, \llbracket A_n \rrbracket)$$

$$\llbracket A_1, \ldots, A_n \xrightarrow{\mathrm{S}} B \rrbracket = [\llbracket A_1 \rrbracket, \ldots, \llbracket A_n \rrbracket \xrightarrow{\mathrm{S}} \llbracket B \rrbracket]$$

Contexts are interpreted as functors $\mathcal{C}^n \to \mathcal{C}$, where n is the number of holes:

$$\llbracket x : A \rrbracket = \star \mapsto \llbracket A \rrbracket : 1 \to \mathcal{C}$$

$$\llbracket \mathrm{S}(\Gamma_1, ..., \Gamma_k) \rrbracket = (\llbracket \Gamma_1 \rrbracket \times ... \times \llbracket \Gamma_k \rrbracket); \underline{\mathrm{S}} : \mathcal{C}^{n_1 + ... + n_k} \to \mathcal{C} \qquad \llbracket -_a \rrbracket = Id : \mathcal{C} \to \mathcal{C}$$

We will just treat the interpretation of contexts with no holes as objects of $\mathcal{C}$. Valid structural transitions $\Gamma \overset{\rho}{\Rightarrow} \Delta$ are interpreted as natural transformations $\llbracket \Gamma \rrbracket \Rightarrow \rho^*; \llbracket \Delta \rrbracket$, where ρ^* is the functor that makes $\llbracket \Delta \rrbracket$ have the same arity as $\llbracket \Gamma \rrbracket$ by mapping holes according to ρ. The interpretation is defined by induction over the derivation in Figure 3.1. Typing judgements $\Gamma \vdash e : A$ are interpreted as arrows $\llbracket \Gamma \rrbracket \to \llbracket A \rrbracket$ by induction over their derivation as shown in Figure 3.2.

3.3.1 Coherence

Since we have defined the interpretation of typing judgements by induction on the structure of their typing derivations we are faced with the possibility that a single typing judgement may have two different interpretations. Fortunately, it follows from Theorem 3.2.20 and the factorisation lemma, Lemma 2.2.2, that any two derivations of the same judgement have the same interpretation.

$$\frac{\rho(\Gamma) = \Gamma'}{[\![\Gamma \overset{\rho}{\Rightarrow} \Gamma' \ \mathit{valid}]\!] = \mathit{id}} \qquad \frac{}{[\![\mathrm{S}(\overrightarrow{\Gamma}, \mathrm{S}'(\overrightarrow{\Delta}), \overrightarrow{\Theta}) \Leftrightarrow \mathrm{S}\{\mathrm{S}'/i\}(\overrightarrow{\Gamma}, \overrightarrow{\Delta}, \overrightarrow{\Theta}) \ \mathit{valid}]\!] = \alpha}$$

$$\frac{}{[\![[](\Gamma) \Leftrightarrow \Gamma \ \mathit{valid}]\!] = \lambda} \qquad \frac{\mathrm{S}' \subseteq \mathrm{S}}{[\![\mathrm{S}(\overrightarrow{\Gamma}) \Rightarrow \mathrm{S}'(\overrightarrow{\Gamma}) \ \mathit{valid}]\!] = \zeta[\mathrm{S}, \mathrm{S}']}$$

$$\frac{\sigma \text{ a permutation on } \{0, ..., |\mathrm{S}| - 1\}}{[\![\mathrm{S}(\overrightarrow{\Gamma}) \Leftrightarrow \sigma\mathrm{S}(\sigma\overrightarrow{\Gamma}) \ \mathit{valid}]\!] = \gamma[\sigma_{\mathrm{S}}]} \ (\textsc{Permutation})$$

$$\frac{\Gamma \equiv_\alpha \Gamma'}{[\![\Gamma \overset{[v(\Gamma') \mapsto v(\Gamma)]}{\Rightarrow} [](\Gamma, \Gamma') \ \mathit{valid}]\!] = \mathit{dup}} \ (\textsc{Contraction})$$

$$\frac{}{[\![\Gamma \Rightarrow []() \ \mathit{valid}]\!] = \mathit{discard}} \ (\textsc{Weakening})$$

Renaming, Composition and Congruence are defined by the identity, natural transformation composition and application of separation product functors respectively.

Figure 3.1: Interpretation of valid structural transitions

$$\frac{}{[\![x : A \vdash x : A]\!] = \mathrm{id}_{[\![A]\!]}} \qquad \frac{[\![\Gamma' \vdash e : A]\!] = e \qquad [\![\Gamma \overset{\rho}{\Rightarrow} \Gamma' \ \mathit{valid}]\!] = s}{[\![\Gamma \vdash \rho(e) : A]\!] = s; e}$$

$$\frac{\overrightarrow{[\![\Gamma \vdash e : A]\!] = e}}{[\![\mathrm{S}(\overrightarrow{\Gamma}) \vdash \mathrm{S}(\overrightarrow{e}) : \mathrm{S}(\overrightarrow{A})]\!] = \underline{\mathrm{S}}(\overrightarrow{e})} \ (\mathrm{SI})$$

$$\frac{[\![\Gamma \vdash e : \mathrm{S}(\overrightarrow{A})]\!] = e \qquad [\![\Delta(\mathrm{S}(\overrightarrow{x : A})) \vdash e' : B]\!] = e'}{[\![\Delta(\Gamma) \vdash \text{let } \mathrm{S}(\overrightarrow{x}) = e \text{ in } e' : B]\!] = [\![\Delta]\!](e); e'} \ (\mathrm{SE})$$

$$\frac{[\![\mathrm{S}(\Gamma, \overrightarrow{x : A}) \vdash e : B]\!] = e}{[\![\Gamma \vdash \lambda_{\mathrm{S}}(\overrightarrow{x}).e : \overrightarrow{A} \xrightarrow{\mathrm{S}} B]\!] = \Lambda(e)} \ (\xrightarrow{\mathrm{S}}\mathrm{I})$$

$$\frac{[\![\Gamma \vdash f : \overrightarrow{A} \xrightarrow{\mathrm{S}} B]\!] = f \qquad \overrightarrow{[\![\Delta \vdash a : A]\!] = a}}{[\![\mathrm{S}(\Gamma, \overrightarrow{\Delta}) \vdash f @_{\mathrm{S}}(\overrightarrow{a}) : B]\!] = \underline{\mathrm{S}}(f, \overrightarrow{a}); \mathrm{ev}} \ (\xrightarrow{\mathrm{S}}\mathrm{E})$$

$$\frac{[\![\Gamma \vdash e : A]\!] = e}{[\![\Gamma \vdash fe : B]\!] = e; \mathcal{I}_{A,B}(f)} \ (\textsc{Prim})$$

Figure 3.2: Interpretation of typing judgements

The method we use is adapted from the proof of the coherence of the interpretation of Syntactic Control of Interference Revisited by O'Hearn *et al* [OPTT99]. This method operates by locally rewriting the derivation tree by pushing contraction rules upwards until they reach the point where the two variables they are contracting are introduced into the same context. This effectively gives us a canonical form for the derivation and hence a way to show that any other derivation is equal. The factorisation Lemma 2.2.2 allows to do this rewriting.

Another possible approach is to give an explicit syntax for every typing derivation so that typing derivations are uniquely specified by judgements. A set of rewriting rules would then allow rewriting into a normal form. This could then be used to show that every judgement in the original system has a unique interpretation. This is the approach taken by Curien and Ghelli [CG92] for System $\mathrm{F}_{\leq}$. We choose the approach taken here to avoid the formulation of a second calculus where the structural rules are made explicit in the term syntax.

First we observe that the interpretation of valid structural transitions is coherent:

Lemma 3.3.2 If π_1 and π_2 are two different derivations of a judgement $\Gamma \stackrel{\rho}{\Rightarrow} \Delta$ *valid* then $[\![\pi_1]\!] = [\![\pi_2]\!]$.

Proof Follows directly from Theorem 3.2.20. □

The following lemma is an adaptation of the key coherence lemma of [OPTT99]. It is a stronger property than we require; the extra structural morphisms are the pieces of derivation tree that we will need to rewrite using the previous two lemmas and commute upwards in the tree.

Lemma 3.3.3 Given derivations π_1 and π_2 of judgements $\Gamma_1 \vdash e_1 : A$ and $\Gamma_2 \vdash e_2 : A$ respectively and valid structural transitions $\Gamma \stackrel{\rho_i}{\Rightarrow} \Gamma_i$ such that $\rho_1(e_1) = \rho_2(e_2)$ then $[\![\rho_1]\!]; [\![\pi_1]\!] = [\![\rho_2]\!]; [\![\pi_2]\!]$.

Proof Induction on the sum of the heights of the derivation trees π_1 and π_2. Name the term $\rho_1(e_1) = \rho_2(e_2)$ as e. Proceed by case analysis on the final rules applied in each derivation:

Both end with Id**:** By Lemma 3.3.2.

One ends with STRUCT**:** Append the valid structural transition to the appropriate structural morphism and apply the induction hypothesis.

Both end with SI**:** The derivations both end in the form:

$$\frac{\Gamma_{i1} \vdash e_{i1} : A_1 \qquad \ldots \qquad \Gamma_{in} \vdash e_{in} : A_n}{\mathrm{S}(\Gamma_{i1}, \ldots, \Gamma_{in}) \vdash \mathrm{S}(e_{i1}, \ldots, e_{in}) : \mathrm{S}(A_1, \ldots, A_n)}$$

where the derivations of the premises are labelled $\pi_{i1}, \ldots, \pi_{in}$. Apply Lemma 2.2.2(1) to the ρ_i, using the distribution of variables in e for δ, to get structural morphisms α_1, α_2 and $\overrightarrow{\beta_1}, \overrightarrow{\beta_2}$ such that $\alpha_1 = \alpha_2$. The result follows by the calculation:

$$\begin{aligned}
[\![\rho_1]\!]; [\![\pi_1]\!] &= [\![\alpha]\!]; \underline{\mathrm{S}}([\![\beta_{11}]\!], \ldots, [\![\beta_{1n}]\!]); \underline{\mathrm{S}}([\![\pi_{11}]\!], \ldots, [\![\pi_{1n}]\!]) \\
&= [\![\alpha]\!]; \underline{\mathrm{S}}([\![\beta_{11}]\!]; [\![\pi_{11}]\!], \ldots, [\![\beta_{1n}]\!]; [\![\pi_{1n}]\!]) \\
&= [\![\alpha]\!]; \underline{\mathrm{S}}([\![\beta_{21}]\!]; [\![\pi_{21}]\!], \ldots, [\![\beta_{2n}]\!]; [\![\pi_{2n}]\!]) \\
&= [\![\rho_2]\!]; [\![\pi_2]\!]
\end{aligned}$$

where the middle step follows from the induction hypothesis and the equalities between interpretations of valid structural transitions follows from Lemma 3.3.2.

Both end with SE**:** The derivations both end in the form:

$$\frac{\Gamma_{i1} \vdash e_{i1} : \mathrm{S}(A_1, \ldots, A_n) \qquad \Gamma_{i2}(\mathrm{S}(x_1 : A_1, \ldots, x_n : A_n)) \vdash e_{i2} : B}{\Gamma_{i2}(\Gamma_{i1}) \vdash \text{let } \mathrm{S}(x_1, \ldots, x_n) = e_1 \text{ in } e_2 : B}$$

where the derivations of the premises are labelled π_{i1} and π_{i2}. Apply Lemma 2.2.2(2) to the ρ_i, using the distribution of variables in e for δ, to get structural morphisms $\alpha = \alpha_1 = \alpha_2$, β_{11}, β_{12} and β_{21}, β_{22}. The result follows by the calculation:

$$\begin{aligned}
[\![\rho_1]\!]; [\![\pi_1]\!] &= [\![\rho_1]\!]; [\![\Gamma_{12}]\!]([\![\pi_{11}]\!]); [\![\pi_{12}]\!] \\
&= [\![\alpha]\!]; [\![\beta_{12}]\!]; [\![\Gamma_{12}]\!]([\![\beta_{11}]\!]); [\![\Gamma_{12}]\!]([\![\pi_{11}]\!]); [\![\pi_{12}]\!] \\
&= [\![\alpha]\!]; [\![\Delta_1]\!]([\![\beta_{11}]\!]; [\![\pi_{11}]\!]); [\![\beta_{12}]\!]; [\![\pi_{12}]\!]
\end{aligned}$$

$$\begin{aligned} &= [\![\alpha]\!]; [\![\Delta_1]\!]([\![\beta_{21}]\!]; [\![\pi_{21}]\!]); [\![\beta_{22}]\!]; [\![\pi_{22}]\!] \\ &= [\![\alpha]\!]; [\![\beta_{22}]\!]; [\![\Gamma_{22}]\!]([\![\beta_{21}]\!]); [\![\Gamma_{22}]\!]([\![\pi_{21}]\!]); [\![\pi_{22}]\!] \\ &= [\![\rho_2]\!]; [\![\pi_2]\!] \end{aligned}$$

Both end with $\xrightarrow{\text{S}}$I: The derivations both end in the form:

$$\frac{\text{S}(\Gamma_i, x_1 : A_1, \ldots, x_n : A_n) \vdash e_i : B}{\Gamma_i \vdash \lambda^{\text{S}}(x_1, \ldots, x_n).e_i : A_1, \ldots, A_n \xrightarrow{\text{S}} B}$$

where the derivation of the premise is labelled π_i'. The result follows by calculation:

$$\begin{aligned} [\![\rho_1]\!]; [\![\pi_1]\!] &= [\![\rho_1]\!]; \Lambda([\![\pi_1']\!]) \\ &= \Lambda(\underline{\text{S}}([\![\rho_1]\!], \overrightarrow{id}); [\![\pi_1']\!]) \\ &= \Lambda(\underline{\text{S}}([\![\rho_2]\!], \overrightarrow{id}); [\![\pi_2']\!]) \\ &= [\![\rho_2]\!]; \Lambda([\![\pi_2']\!]) \\ &= [\![\rho_2]\!]; [\![\pi_2]\!] \end{aligned}$$

Both end with $\xrightarrow{\text{S}}$E: Similar to the case for SI.

Both end with Prim**:** Trivial, since this rule is interpreted by post-composition. □

Theorem 3.3.4 (Coherence) If π_1 and π_2 are two derivations of the same judgement $\Gamma \vdash e : A$ then $[\![\pi_1]\!] = [\![\pi_2]\!]$.

Proof Instance of Lemma 3.3.3. □

3.3.2 Soundness and Completeness

Now that we are sure that we can interpret typing derivations in a well-defined way, we must make sure that the category soundly models the equational theory of the calculus. We do this with reference to a definition of a model of a λ_{sep} theory.

A *model* of a λ_{sep} theory $(\mathcal{T}, \Phi, \Sigma)$ is an interpretation $(\mathcal{I}, \mathcal{I}_{A,B})$ of the system $(\mathcal{T}, \Phi)$ with the condition that for all the axioms $\Gamma \vdash e_1 = e_2 : A$ in Σ, it is the case that $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

We begin by the proving the standard result about the interpretation of substitution. This follows the structure of the proof of the well-typing of substitution.

Lemma 3.3.5 (Soundness of Substitution) If $[\![\Gamma\overrightarrow{(x : A)} \vdash e : B]\!] = e$ and for all i, $[\![\Delta_i \vdash a_i : A_i]\!] = a_i$ then $[\![\Gamma\overrightarrow{(\Delta)} \vdash e[\overrightarrow{a/x}] : B]\!] = [\![\Gamma]\!]\overrightarrow{(a)}; e$

Proof By induction on the derivation of $\Gamma\overrightarrow{(x : A)} \vdash e : B$. This follows the structure of the proof of Lemma 2.2.4, the result follows by simple categorical reasoning about the interpretations. □

Soundness is a direct consequence of this:

Theorem 3.3.6 (Soundness) For a λ_{sep} theory $\mathbb{T}$ and a model $\mathbb{M}$ of this theory, if $\Gamma \vdash e = e' : A$ in $\mathbb{T}$ then $[\![\Gamma \vdash e : A]\!] = [\![\Gamma \vdash e' : A]\!]$ in $\mathbb{M}$.

Proof By induction on the derivation of $\Gamma \vdash e = e'$. All cases are straightforward — either by simple calculation or substitution (Lemma 3.3.5). □

To prove completeness, we construct a model of a λ_{sep} theory $\mathbb{T}$ in the term category $\mathbf{Tm}_{\mathbb{T}}$ we have developed in Lemmas 3.2.7, 3.2.17, 3.2.22 and 3.2.28. There is an obvious interpretation of the system component of $\mathbb{T}$ in $\mathbf{T}_{\mathbb{T}}$ that maps primitive types to their corresponding objects and primitive operations f to arrows $[x : A \vdash fx : B]$. To show that this is a model, we must prove a connection between the interpretation of a judgement $\Gamma \vdash e : A$ and an arrow of $\mathbf{Tm}_{\mathbb{T}}$. To this end, for a context Γ, define the corresponding type $\overline{\Gamma}$ by induction.

$$\overline{x : A} = A \qquad\qquad \overline{\mathrm{S}(\Gamma_1, ..., \Gamma_n)} = \mathrm{S}(\overline{\Gamma_1}, ..., \overline{\Gamma_2})$$

And given a judgement $\Gamma \vdash e : A$, then $x : \overline{\Gamma} \vdash \overline{\Gamma, x, e} : A$ is derivable, where:

$$\overline{y : A, x, e} = e[y/x]$$

$$\overline{\mathrm{S}(\Gamma_1, ..., \Gamma_n), x, e} = \text{let } \mathrm{S}(x_1, ..., x_n) = x \text{ in } \overline{\Gamma_1, x_1, ..., \overline{\Gamma_n, x_n, e}}$$

Proposition 3.3.7 (Term Model) The interpretation of a theory $\mathbb{T}$ in $\mathbf{T}_{\mathbb{T}}$ as defined above is a model with the property that $[\![\Gamma \vdash e : A]\!] = [x : \overline{\Gamma} \vdash \overline{\Gamma, x, e} : A]$.

Proof The property of the interpretation holds by induction on the derivation of $\Gamma \vdash e : A$ and reasoning about the interpretation using the equational rules. This, along with the construction of arrows of $\mathbf{Tm}_{\mathbb{T}}$ as equivalence classes, implies that the interpretation is a model. □

Theorem 3.3.8 (Completeness) If $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$ in all models then $\Gamma \vdash e_1 = e_2 : A$.

Proof We prove the contrapositive. Assume $\Gamma \vdash e_1 \neq e_2 : A$. This implies that $x : \overline{\Gamma} \vdash \overline{\Gamma, x, e_1} \neq \overline{\Gamma, x, e_2} : A$. Hence $[\![\Gamma \vdash e_1 : A]\!] \neq [\![\Gamma \vdash e_2 : A]\!]$ in the term model, i.e. the interpretations are not equal in all models. □

Chapter 4

Day's Construction and Presheaf Models

In [Day70], Day shows how closed (symmetric) monoidal structure on functor categories may be generated from *promonoidal structure* on the domain category, of which (symmetric) monoidal structure is an instance. In this chapter we extend Day's construction to separation products, in the special case of functors into **Set**, by defining *proseparation structure* and showing how this generates closed separation products on the functor category. A category with separation products (not necessarily closed) will form an instance of proseparation structure. To show how λ_{sep} models resources and their separation we give two models based on separation. The first, in Section 4.3.2, models global separation where there is a fixed notion of separation between resources. The second, in Section 4.3.3 models localised separation, where the notion of separation is local to the resources used in particular separation product.

Day's construction is a generalisation of the frame, or possible world, semantics of substructural logics [Res00]. Instead of boolean valued predicates on a partial order of possible worlds, we model types as **Set**-valued functors over a category. Functor category semantics were used by Reynolds and Oles to model the block structure of Idealised Algol [Rey81, Ole82]. Pym, O'Hearn and Yang used Day's construction to provide a models of BI and the $\alpha\lambda$-calculus [POY04, O'H03]

by constructing closed symmetric monoidal structure on the functor category from (possibly partial) symmetric monoidal structure on the domain category.

In Day's original paper he used the term *premonoidal* for the structure he requires on the domain category. We use the terms promonoidal and proseparation by analogy with profunctors, a categorical generalisation of relations [Win05, DS95, Bor94].

Since Day's construction is defined in terms of ends and coends, we shall first recall the definitions of dinaturality, end and coend and state some useful facts about them. We then give the definition of proseparation structure, our adaptation of Day's promonoidal structure, and then give three examples of classes of categories with the correct structure. The second and third of these give us an explicit resource interpretation of λ_{sep}.

4.1 Dinaturality and (Co)Ends

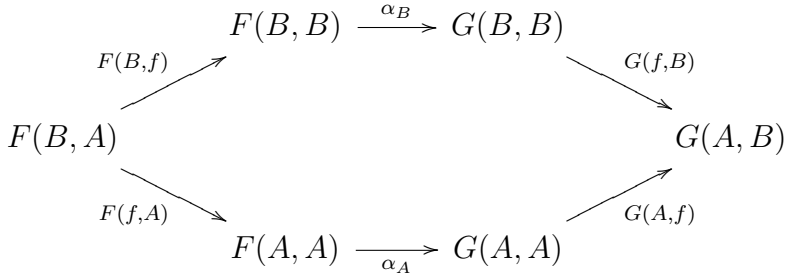
We now recall the definition of dinaturality, ends and coends and some useful facts about them. The material in the first subsection is a re-presentation of the section on dinaturality and (co)ends in [Mac98], §IX.4-8, and also some parts from [CHW02]. The second subsection contains a definition and a lemma from [Day70] that will be useful in the next section.

4.1.1 Definition and Properties

Dinatural transformations generalise the definition of natural transformations by describing transformations between functors that have both covariant and contravariant arguments.

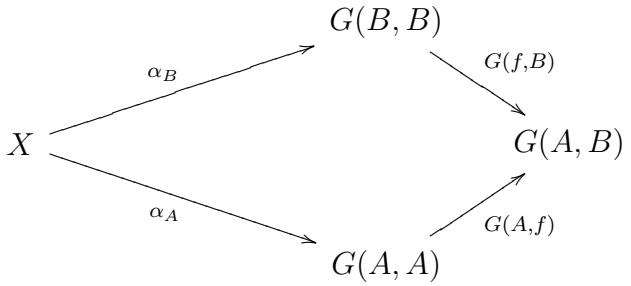
Definition 4.1.1 (Dinatural Transformation) For functors $F, G : \mathcal{C} \times \mathcal{C}^{\mathsf{op}} \to \mathcal{D}$, a dinatural transformation from $\alpha : F \stackrel{\cdot\cdot}{\to} G$ is a family of arrows $\alpha_C : F(C, C) \to G(C, C)$, indexed by objects C of $\mathcal{C}$, such that for every $f : A \to B$

in $\mathcal{C}$ the following diagram commutes:



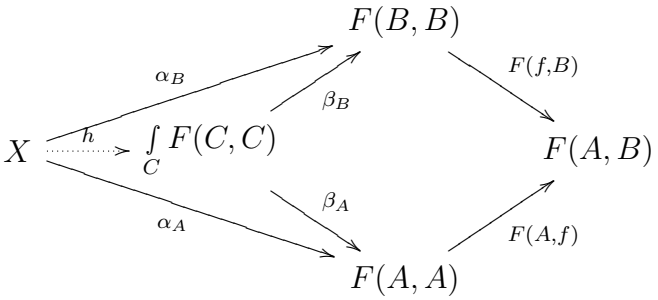
It is easy to see that this definition specialises to the definition of natural transformation when F and G are both constant in their first or second arguments.

A special case occurs when either F or G is constant. The case when F is constant is called a *wedge* $\alpha : X \stackrel{\cdot\cdot}{\to} G$:



The case when G is constant is also called a wedge $\alpha : F \stackrel{\cdot\cdot}{\to} X$. Ends and coends are universal such wedges:

Definition 4.1.2 (End) An *end* for a functor $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ is a universal wedge. The object part of the end is written $\int_A F(A, A)$. That is, there is a wedge $\beta : \int_C F(C, C) \stackrel{\cdot\cdot}{\to} F$ such that for any other wedge $\alpha : X \stackrel{\cdot\cdot}{\to} F$, there is a unique arrow $h : X \to \int_C F(C, C)$ making all instances of the following diagram commute:



Definition 4.1.3 (Coend) A *coend* of a functor $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ is a couniversal wedge. The object of the wedge is written is written $\int^A F(A, A)$. That is, there is a wedge $\beta : F \stackrel{\cdot\cdot}{\to} \int^C F(C, C)$ such that for any other wedge $\alpha : F \stackrel{\cdot\cdot}{\to} X$, there is a unique arrow $h : \int^C F(C, C) \to X$ making all instances of the dual of the above diagram for ends commute.

We will be concerned with (co)ends with the codomain category $\mathcal{D}$ equal to **Set** and where $\mathcal{C}$ is small. A choice for ends and coends in **Set** is given by the following proposition:

Proposition 4.1.4 ((Co)Ends in Set) A choice for the end of a functor $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ is given by:

$$\int_A F(A, A) = \{x \in \prod_A F(A, A) | \forall f : A \to B. F(f, B)x_B = F(A, f)x_A\}$$

A choice for the coend of a functor $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ is given by:

$$\int^A F(A, A) = (\sum_A F(A, A))/\approx$$

where $\approx$ is the least equivalence relation such that:

$$(A, x) \approx (B, y) \text{ if } \exists f : B \to A. \exists z \in F(A, B). x = F(A, f)z \wedge y = F(f, B)z$$

Proof See [CHW02]. □

By this proposition, an end in **Set** is isomorphic to a subset of the product of the sets $F(A, A)$ for all objects A in $\mathcal{C}$ which respect the action of the arrows of $\mathcal{C}$. We shall use ends as a generalisation of the $\forall$ quantifier of logic. Dually, coends in **Set** are isomorphic to a quotient of the disjoint sum of sets $F(A, A)$ for all A. We shall use coends as a generalisation of the $\exists$ quantifier. We shall not formalise these connections.

Given the informal connection to existential quantification, we read the elements of the universal wedge $\beta_A : F(A, A) \to \int^C F(C, C)$ can be read as "summing out" the variable A; instantiating the existential with the witness A.

To demonstrate the reading of ends as a categorical generalisation of universal quantification that respects naturality, the following proposition gives a

description of the natural transformations between two functors in terms of an end.

Proposition 4.1.5 (Ends and Natural Transformations) Given functors $F, G : \mathcal{C} \to \mathcal{D}$, ends describe the set of natural transformations $F \Rightarrow G$.

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_A \mathcal{D}(F(A), G(A))$$

Proof See §IX.5 in [Mac98]. □

Given this formulation of natural transformations as ends, we can restate the Yoneda Lemma in terms of ends:

Proposition 4.1.6 (Yoneda Lemma) Given functors $F : \mathcal{C} \to \mathbf{Set}$, $G : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$ and an object B of $\mathcal{C}$, the following are isomorphisms:

$$\int_A [\mathcal{C}(B, A), F(A)] \cong F(B) \qquad \int_A [\mathcal{C}(A, B), G(A)] \cong G(B)$$

These isomorphisms are natural in B and F.

We now state some propositions which give more useful properties of ends and coends. Firstly, if the functor F has additional parameters such that the end $\int_A F(A, A, P)$ exists for all P, then this induces a functor in the parameter. This proposition will allow us to use ends and coends to define objects of a functor category.

Proposition 4.1.7 (Parameterisation) Given a functor $F : \mathcal{P} \times \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ such that for every object $P \in \mathcal{P}$ it has an end $\alpha_P : \int_A F(P, A, A) \dddot{\to} F(P, -, -)$, there is a unique functor $H : \mathcal{P} \to \mathcal{D}$ such that $H(P) = \int_A (F(P, A, A)$ and the following diagram commutes for every $f : P \to P'$ in $\mathcal{P}$ and $C \in \mathcal{C}$:

$$\begin{array}{ccc} H(P) = \int\limits_A F(P, A, A) & \xrightarrow{(\alpha_P)_C} & F(P, C, C) \\ \Big\downarrow{\scriptstyle H(f)} & & \Big\downarrow{\scriptstyle F(f,C,C)} \\ H(P') = \int\limits_A F(P', A, A) & \xrightarrow{(\alpha'_P)_C} & F(P', C, C) \end{array}$$

The dual proposition also holds for coends.

Proof This is Theorem IX.7.2 in [Mac98]. □

It is also possible to express ends and coends as limits and colimits in a certain category. Therefore, standard facts about preservation of limits carry over to ends and coends:

Proposition 4.1.8 (Ends and Homsets) Given a functor $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ and an object B of $\mathcal{D}$, the homset functor $\mathcal{D}(-,-)$ preserves ends and reverses coends:

$$\int_A \mathcal{D}(B, F(A,A)) \cong \mathcal{D}(B, \int_A F(A,A))$$

$$\mathcal{D}(\int^A F(A,A), B) \cong \int_A \mathcal{D}(F(A,A), B)$$

Proof Follows directly from Proposition IX.5.3 in [Mac98], showing that ends may be expressed as limits in a certain category and the fact that homset functors preserve limits. □

It is also possible to change the order of iterated ends, in the same way that iterated limits may be interchanged:

Proposition 4.1.9 (Interchange of Iterated Ends) Let $F : \mathcal{A}^{\mathsf{op}} \times \mathcal{A} \times \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ be a functor such that for all $A, B \in \mathcal{A}$ the end $\int_C F(A,B,C,C)$ exists and for all $C, D \in \mathcal{C}$ the end $\int_A F(A,A,C,D)$ exists, then there is an isomorphism:

$$\int_A \int_C F(A,A,C,C) \cong \int_C \int_A F(A,A,C,C)$$

Proof This is the Corollary in §IX.8 in [Mac98]. □

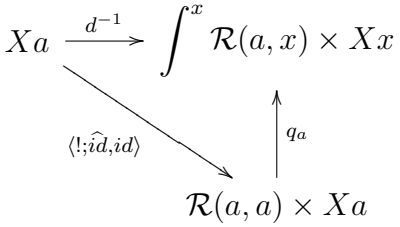
Finally, in this section, we state the Density formula:

Proposition 4.1.10 (Density Formula) Given functors $F : \mathcal{C} \to \mathbf{Set}$ and $G : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$, the following are isomorphisms:

$$d : \int^A F(A) \times \mathcal{C}(A,B) \cong F(B) \qquad d_{\mathsf{op}} : \int^A G(A) \times \mathcal{C}(B,A) \cong G(B)$$

All natural in B and any other variables in X. Moreover, the inverse d^{-1} obeys

the following diagram:



A similar diagram holds for d^{-1}_{op}.

Proof See Section 7.2 in [CHW02]. The property of the inverse can be seen to hold by working directly with the choice of coends in **Set** given above. □

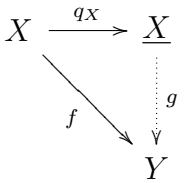
4.1.2 Day's Notation for Coends Involving $\times$

We will make heavy use of coends of the form $\int^A F(A,-) \times G(A,-)$ where F and G are functors of different variances in A into **Set**. Following Day, we abbreviate such expressions to $F(A,-) \underline{\times} G(A,-)$, the repeated variable A showing the bound variable of the coend. The rest of this subsection is a re-presentation of some of [Day70] on the properties of such expressions.

Observe that for every expression $\underline{X}$ involving uses of $\underline{\times}$ there is an expression X formed by replacing each $\underline{\times}$ by $\times$. There is also a canonical natural transformation $q_X : X \to \underline{X}$ defined by applying the universal wedges of the coends in a depth first manner.

We now state the following lemma from [Day70], which shows that this q_X is in fact a universal wedge for a multiple coend of all the variables bound in $\underline{X}$.

Lemma 4.1.11 Let Y be a functor into **Set** and $\underline{X}$ be an expression built using $\underline{\times}$. Let $f : X \to Y$ be a transformation, natural in all the variables bound in $\underline{X}$. Then there is a unique g such that the following diagram commutes:



Proof This is Day's Lemma 2.5. □

We now describe some of the applications of this Lemma noted by Day and used in his construction. They will be useful in the next section.

When $f = h; q_Y : X \to \underline{Y}$ we write the induced arrow as $\underline{h} : \underline{X} \to \underline{Y}$. Further, when $h = i \times j$, then we write $i \underline{\times} j$ instead of $\underline{i \times j}$. We can treat $\underline{\times}$ as being a two argument functor.

When h is one of the isomorphisms of symmetric monoidal structure of $\times$ in **Set**, the uniqueness given by Lemma 4.1.11 ensures that the coherence theorem for $\times$ also holds for $\underline{\times}$.

4.2 Proseparation

We are going to interpret the types of λ_{sep} as functors from a category of resources into the category **Set** of sets and functions. Thus, each type is interpreted semantically as a family of sets, indexed by the available resources.

To interpret separation products and functions we extend Day's construction for monoidal products and functions on functor categories. Day defines *promonoidal* structure on the domain category and shows how this can be used to define monoidal structure on the functor category. Day formulated his definitions and results for general functor categories $[\mathcal{C}, \mathcal{V}]$, where $\mathcal{V}$ is a complete and co-complete category and $\mathcal{C}$ is a small, $\mathcal{V}$-enriched category. For simplicity we shall restrict to the case where $\mathcal{V} = \mathbf{Set}$. In this case, $\mathcal{C}$ is a normal small category.

In this restricted case, the first half of Days' promonoidal structure consists of a pair of functors:

$$P : \mathcal{C}^{\mathsf{op}} \times \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set} \qquad\qquad J : \mathcal{C} \to \mathbf{Set}$$

These are used to define the monoidal products and the unit objects respectively. The monoidal products are defined by the coend formula:

$$(A \otimes B)x = \int^{a,b} Aa \times Bb \times P(a, b, x)$$

To finish the definition Day requires three natural isomorphisms:

$$\lambda_{ab} : Jx\underline{\times}P(x,a,b) \to \mathcal{C}(a,b) \qquad \rho_{ab} : Jx\underline{\times}P(a,x,b) \to \mathcal{C}(a,b)$$

$$\alpha_{abcd} : P(a,b,x)\underline{\times}P(x,c,d) \to P(b,c,x)\underline{\times}P(a,x,d)$$

These must obey axioms similar to the coherence axioms for monoidal structure. These isomorphisms are then used to generate the structural isomorphisms for the monoidal structure on the functor category. Their axioms ensure the satisfaction of the axioms for monoidal structure. Monoidal closure follows automatically from these definitions. Day then shows that the monoidal structure on the functor category can be made to be symmetric by postulating the existence of an extra natural isomorphism, obeying appropriate commuting diagrams.

We shall extend this definition to cover the structure required to model λ_{sep} by requiring the existence of a family of functors P_{S}, indexed by separation relations S. This definition, with the attendant isomorphisms and commutative diagrams in given in the next subsection, along with a proof that the do indeed satisfy the definitions of the previous chapter.

The next two sections, give the additional isomorphisms to give permutation and S-weakening structure and then duplication and weakening structure. Finally we show that, as for Day's construction, separation closure in the functor category follows from these definitions.

4.2.1 Base Definition

Definition 4.2.1 A category $\mathcal{C}$ has proseparation structure if it has, for each separation relation S, a functor:

$$P_{\text{S}} : (\mathcal{C}^{\mathsf{op}})^{|\text{S}|} \times \mathcal{C} \to \mathbf{Set}$$

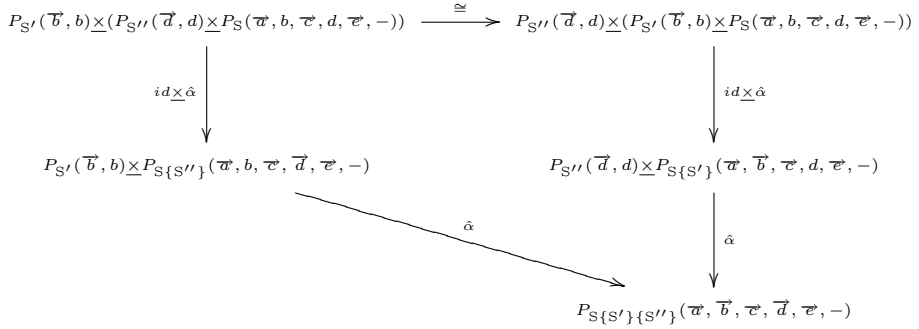
And natural isomorphisms:

$$\widehat{\alpha} : P_{\text{S}'}(\overrightarrow{s}, s)\underline{\times}P_{\text{S}}(\overrightarrow{r}, s, \overrightarrow{t}, u) \cong P_{\text{S}\{\text{S}'\}}(\overrightarrow{r}, \overrightarrow{s}, \overrightarrow{t}, u)$$

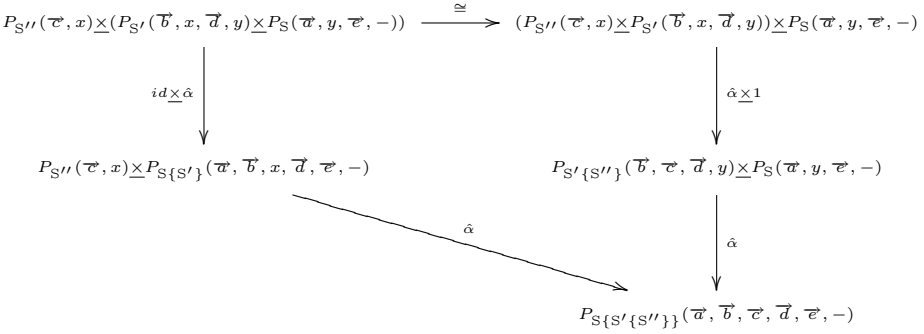
and

$$\widehat{\lambda} : P_{[]_1}(a,b) \cong \mathcal{C}(a,b)$$

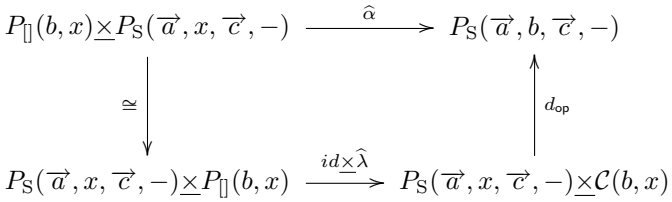
Such that the $\hat{\alpha}$s obey two commutative diagrams:



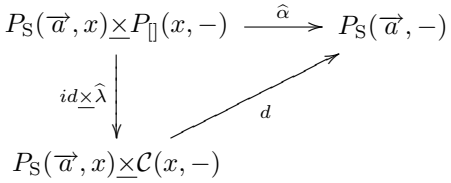
and,



The form of the substituted separation relations makes sense by Lemma 2.1.3. Further, the $\widehat{\lambda}$ and $\widehat{\alpha}$ must obey the following two diagrams:



and,



The commutative diagrams express the same conditions as the required diagrams in Definition 3.2.3. Indeed, each of the diagrams above will imply the corresponding diagram for the separation products in the functor category.

We now define how the proseparation structure is used to define separation structure in the functor category.

Definition 4.2.2 (Separation Products on $[\mathcal{C}, Set]$) Define separation products as the functor:

$$\underline{\mathrm{S}}(\overrightarrow{A}) = A_1 a_1 \underline{\times} (A_2 a_2 \underline{\times} (\ldots A_n a_n \underline{\times} P_{\mathrm{S}}(\overrightarrow{a}, -)) \ldots)$$

and the flattening natural isomorphisms α by:

$$\begin{array}{rl} & \underline{\mathrm{S}}(\overrightarrow{A}, \mathrm{S}'(\overrightarrow{B}), \overrightarrow{C}) \\ = & \overrightarrow{Aa} \underline{\times} (\overrightarrow{Bb} \underline{\times} P_{\mathrm{S}'}(\overrightarrow{b}, b)) \underline{\times} \overrightarrow{Cc} \underline{\times} P_{\mathrm{S}}(\overrightarrow{a}, b, \overrightarrow{c}, -) \\ \cong & \overrightarrow{Aa} \underline{\times} \overrightarrow{Bb} \underline{\times} \overrightarrow{Cc} \underline{\times} P_{\mathrm{S}'}(\overrightarrow{b}, b) \underline{\times} P_{\mathrm{S}}(\overrightarrow{a}, b, \overrightarrow{c}, -) \\ \stackrel{id \underline{\times} \hat{\alpha}}{\longrightarrow} & \overrightarrow{Aa} \underline{\times} \overrightarrow{Bb} \underline{\times} \overrightarrow{Cc} \underline{\times} P_{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}, -) \\ = & \underline{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{C}) \end{array}$$

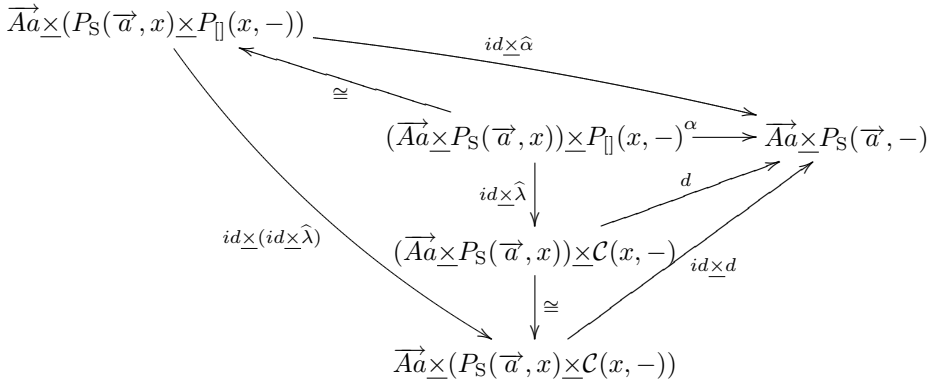
The natural isomorphisms λ are defined by:

$$\begin{array}{rl} & \underline{[]}_1(A) - \\ = & Aa \underline{\times} P_{[]}(a, -) \\ \stackrel{id \underline{\times} \hat{\lambda}}{\longrightarrow} & Aa \underline{\times} \mathcal{C}(a, -) \\ \stackrel{d}{\longrightarrow} & A- \end{array}$$

Proposition 4.2.3 Definition 4.2.2 defines separation products on $[\mathcal{C}, \mathbf{Set}]$.

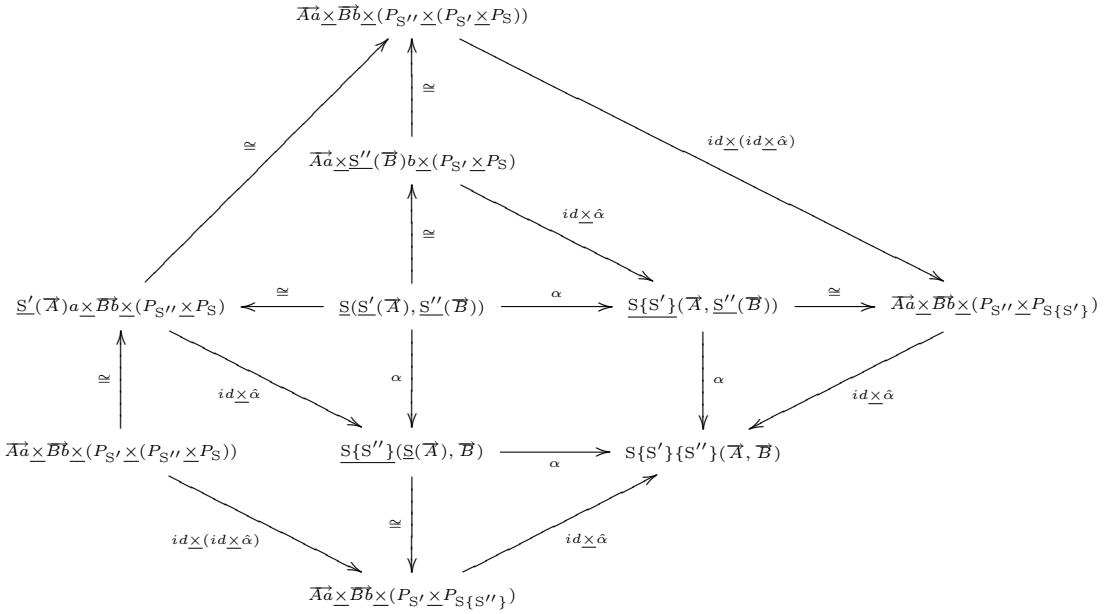
Proof Proposition 4.1.7 ensures that the definition of $\underline{\mathrm{S}}$ actually defines a functor. The definitions of α and λ give natural isomorphisms by Lemma 4.1.11. It remains to show that the definitions of α and λ satisfy the required properties.

The second condition for λ holds by examination of this diagram:



The topmost triangle commutes by the definition of α; the leftmost by naturality; and the bottom by Day's Lemma 2.9. Hence the inner diagram commutes. The first condition for λ holds by a similar diagram and an application of Day's Lemma 2.7.

The first diagram for $\hat{\alpha}$ can be seen to imply the first diagram for α by examination of the special case in the following diagram:



The outer edge of the diagram commutes by the first diagram for $\hat{\alpha}$ and the outer diagrams commute either by the coherence of the isomorphisms, the definition of α, or naturality. Hence, the inner diagram commutes, as required. The second diagram for α commutes by similar reasoning. □

4.2.2 Permutation and S-Weakening

Again, we must take care to get the correct coherence axioms. These are derived from the coherence axioms from the previous chapter.

Definition 4.2.4 (Proseparation Permutation) A category $\mathcal{C}$ that has proseparation structure has permutation if it has a family of natural isomorphisms, indexed by separation relations S and permutations σ on the set $\{0, \ldots, |\mathrm{S}| - 1\}$:

$$\widehat{\gamma[\sigma_{\mathrm{S}}]} : P_{\mathrm{S}}(\overrightarrow{r}, r) \cong P_{\sigma\mathrm{S}}(\sigma(\overrightarrow{r}), r)$$

Subject to the following equations:

$$\widehat{\gamma[\sigma_{\mathrm{S}}]};\widehat{\gamma[\sigma'_{\sigma\mathrm{S}}]} = \gamma[\widehat{(\sigma;\sigma')_{\mathrm{S}}}] \qquad \widehat{\gamma_{\mathrm{id}_S}}_A = id_A \qquad \widehat{\gamma[\sigma_{\mathrm{S}}]}^{-1} = \widehat{\gamma[\sigma_{\mathrm{S}}^{-1}]}$$

and the following two commuting diagrams:

$$\begin{array}{ccc} P_{\mathrm{S}'}(\overrightarrow{b},b)\underline{\times}P_{\mathrm{S}}(\overrightarrow{a},b,\overrightarrow{c},-) & \xrightarrow{\widehat{\gamma[\sigma_{\mathrm{S}'}]}\underline{\times}id} & P_{\sigma\mathrm{S}'}(\sigma(\overrightarrow{b}),b)\underline{\times}P_{\mathrm{S}}(\overrightarrow{a},b,\overrightarrow{c},-) \\ \downarrow\widehat{\alpha} & & \downarrow\widehat{\alpha} \\ P_{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a},\overrightarrow{b},\overrightarrow{c},-) & \xrightarrow{\widehat{\gamma[\mathrm{S}\{\sigma_{\mathrm{S}'}\}]}} & P_{\mathrm{S}\{\sigma\mathrm{S}'\}}(\overrightarrow{a},\sigma(\overrightarrow{b}),\overrightarrow{c},-) \end{array}$$

$$\begin{array}{ccc} P_{\mathrm{S}'}(\overrightarrow{b},b)\underline{\times}P_{\mathrm{S}}(\overrightarrow{a},b,\overrightarrow{c},-) & \xrightarrow{id\underline{\times}\widehat{\gamma[\sigma_{\mathrm{S}}]}} & P_{\mathrm{S}'}(\overrightarrow{b},b)\underline{\times}P_{\sigma\mathrm{S}}(\sigma(\overrightarrow{a},b,\overrightarrow{c}),-) \\ \downarrow\widehat{\alpha} & & \downarrow\widehat{\alpha} \\ P_{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a},\overrightarrow{b},\overrightarrow{c},-) & \xrightarrow{\widehat{\gamma[\sigma_{\mathrm{S}}\{\mathrm{S}'\}]}} & P_{\sigma\mathrm{S}\{\mathrm{S}'\}}(\sigma(\overrightarrow{a},[\overrightarrow{b}],\overrightarrow{c}),-) \end{array}$$

where the substitution of permutations into separation relations and vice versa is as defined in Definition 3.2.12.

Definition 4.2.5 (Proseparation S-Weakening) A category that has proseparation structure has S-*Weakening* if it has a family of natural transformations, indexed by pairs of separation relations $\mathrm{S} \subseteq \mathrm{S}'$:

$$\widehat{\zeta[\mathrm{S}',\mathrm{S}]} : P_{\mathrm{S}'}(\overrightarrow{a},a) \to P_{\mathrm{S}}(\overrightarrow{a},a)$$

Subject to the following equations:

$$\widehat{\zeta[\mathrm{S},\mathrm{S}']};\widehat{\zeta[\mathrm{S}',\mathrm{S}'']} = \widehat{\zeta[\mathrm{S},\mathrm{S}'']} \qquad \widehat{\zeta[\mathrm{S},\mathrm{S}]}_A = id_A$$

And also the analogues of the two diagrams from Definition 4.2.4 for the ζ natural transformations. Moreover, if the category has permutation and S-weakening then the following diagram should commute:

$$\begin{array}{ccc} P_{\mathrm{S}}(\overrightarrow{a},a) & \xrightarrow{\widehat{\gamma[\sigma_{\mathrm{S}}]}} & P_{\sigma\mathrm{S}}(\sigma(\overrightarrow{a}),a) \\ \downarrow\widehat{\zeta[S,S']} & & \downarrow\widehat{\zeta[\sigma S,\sigma S']} \\ P_{\mathrm{S}'}(\overrightarrow{a},a) & \xrightarrow{\widehat{\gamma[\sigma_{\mathrm{S}'}]}} & P_{\sigma\mathrm{S}'}(\sigma(\overrightarrow{a}),a) \end{array}$$

Definition 4.2.6 (Permutation and S-Weakening in $[\mathcal{C}, \mathbf{Set}]$) Define the family $\gamma[\sigma_{\mathrm{S}}] : \underline{\mathrm{S}}(\overrightarrow{A}) \to \underline{\sigma\mathrm{S}}(\sigma(\overrightarrow{A}))$ as:

$$\begin{array}{rl} & A_1 a_1 \underline{\times} (A_2 a_2 \underline{\times} (\ldots A_n a_n \underline{\times} P_{\mathrm{S}}(\overrightarrow{a}, -)) \ldots) \\ \cong; id \underline{\times} (\ldots \underline{\times} \widehat{\gamma[\sigma_{\mathrm{S}}]}) \longrightarrow & A_{\sigma(1)} a_{\sigma(1)} \underline{\times} (A_{\sigma(2)} a_{\sigma(2)} \underline{\times} (\ldots A_{\sigma(n)} a_{\sigma(n)} \underline{\times} P_{\sigma\mathrm{S}}(\sigma(\overrightarrow{a}), -)) \ldots) \end{array}$$

Define the family $\zeta[\mathrm{S}, \mathrm{S}'] : \underline{\mathrm{S}}(\overrightarrow{A}) \to \underline{\mathrm{S}'}(\overrightarrow{A})$ as:

$$\begin{array}{rl} & A_1 a_1 \underline{\times} (A_2 a_2 \underline{\times} (\ldots A_n a_n \underline{\times} P_{\mathrm{S}}(\overrightarrow{a}, -)) \ldots) \\ id \underline{\times} (\ldots \underline{\times} \widehat{\zeta[S, S']}) \longrightarrow & A_1 a_1 \underline{\times} (A_2 a_2 \underline{\times} (\ldots A_n a_n \underline{\times} P_{\mathrm{S}'}(\overrightarrow{a}, -)) \ldots) \end{array}$$

We can now show that these definitions do indeed define permutation and S-weakening in $[\mathcal{C}, \mathbf{Set}]$.

Proposition 4.2.7 Definition 4.2.6 defines the permutation and S-weakening of Section 3.2.2 in $[\mathcal{C}, \mathbf{Set}]$.

Proof We only verify the properties for the permutation natural transformations; the S-weakening properties are almost identical.

The definition of $\gamma[\sigma_{\mathrm{S}}]$ is a natural isomorphism by Lemma 4.1.11 with the fact that $\widehat{\gamma[\sigma_{\mathrm{S}}]}$ is a natural isomorphism. The algebraic laws hold by the algebraic laws for the proseparation permutation. Therefore this definition satisfies the requirements of Definition 3.2.8.

The commutativity of the two diagrams involving γ and α (Definition 3.2.13) follows from the commutativity of the two diagrams involving $\widehat{\gamma}$ and $\widehat{\alpha}$, by similar diagrams to the proof of Proposition 4.2.3. Similarly, the diagram involving γ and ζ (Definition 3.2.10) follows from the diagram requiring $\widehat{\gamma}$ and $\widehat{\zeta}$ commute. □

4.2.3 Discarding and Duplication

The final piece of structure required to model λ_{sep} is that for discarding and duplication. We are only able to construct categories where the object $\underline{[]_0}()$ is the terminal object because it is not possible to give an arrow from an arbitrary functor of $[\mathcal{C}, \mathbf{Set}]$ to $\underline{[]_0}()$ just by information on $\mathcal{C}$. As before, there is a one-one

correspondence between the axioms for the structure in a proseparation category and the axioms for the functor category.

Definition 4.2.8 (Proseparation Weakening and Duplication) A category with proseparation structure, permutation and S-weakening has weakening and duplication if $P_{[]_0}(a) = \{*\}$, so that $P_{[]_0}$ is the (unique up-to isomorphism) terminal object in $[\mathcal{C}, \mathbf{Set}]$, and there is a dinatural transformation:

$$\widehat{dup} : 1 \xrightarrow{\cdot\cdot} P_{[]_2}(-,-,+)$$

These are subject to the following commutative diagrams for left and right unit (where $\widehat{id_a}$ maps to the identity arrow for each a):

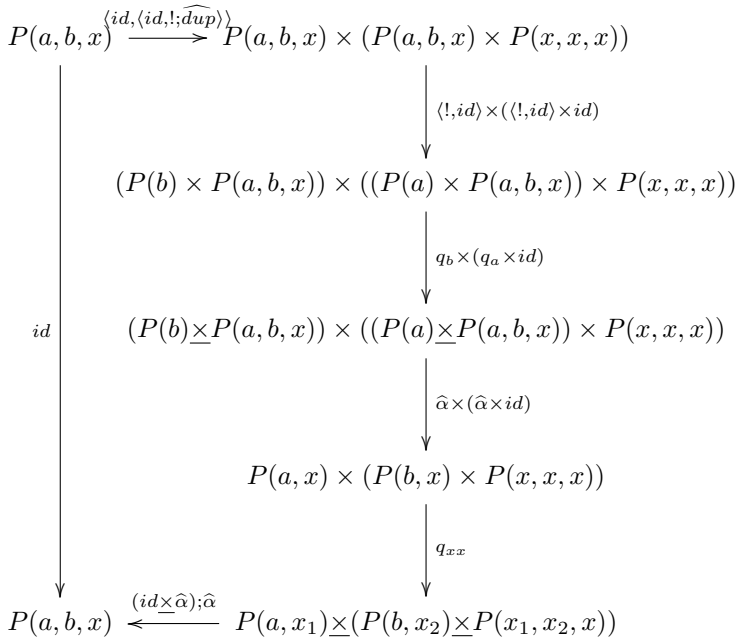
$$\begin{array}{ccc}
1 & \xrightarrow{\widehat{id_a}} & \mathcal{C}(a,a) \\
\downarrow{\scriptstyle \widehat{dup_a}} & & \downarrow{\scriptstyle \widehat{\lambda}} \\
P_{[]_2}(a,a,a) & & P_{[]_1}(a,a) \\
\downarrow{\scriptstyle \langle id,! \rangle} & & \downarrow{\scriptstyle \widehat{\alpha^{-1}}} \\
P_{[]_2}(a,a,a) \times P_{[]_0}(a) & \xrightarrow{q_a} & P_{[]_2}(a,x,a) \underline{\times} P_{[]_0}(x)
\end{array}$$

$$\begin{array}{ccc}
1 & \xrightarrow{\widehat{id_a}} & \mathcal{C}(a,a) \\
\downarrow{\scriptstyle \widehat{dup_a}} & & \downarrow{\scriptstyle \widehat{\lambda}} \\
P_{[]_2}(a,a,a) & & P_{[]_1}(a,a) \\
\downarrow{\scriptstyle \langle id,! \rangle} & & \downarrow{\scriptstyle \widehat{\alpha^{-1}}} \\
P_{[]_2}(a,a,a) \times P_{[]_0}(a) & \xrightarrow{q_a} & P_{[]_2}(x,a,a) \underline{\times} P_{[]_0}(x)
\end{array}$$

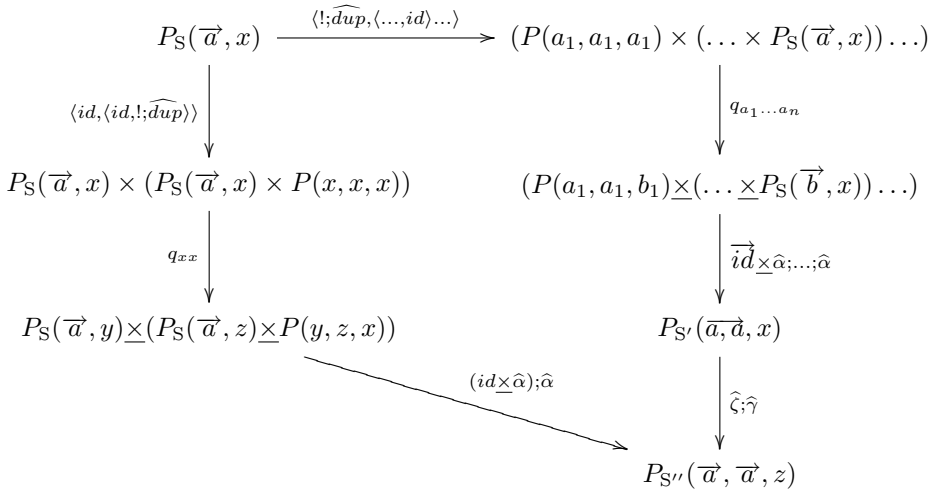
The following diagram for associativity must commute:

$$\begin{array}{ccccc}
1 & \xrightarrow{\langle \widehat{dup}, \widehat{dup} \rangle} & P(a,a,a) \times P(a,a,a) & \xrightarrow{q_a} & P(a,a,x) \underline{\times} P(a,x,a) \\
\downarrow{\scriptstyle \langle \widehat{dup}, \widehat{dup} \rangle} & & & & \downarrow{\scriptstyle \widehat{\alpha}} \\
P(a,a,a) \times P(a,a,a) & \xrightarrow{q_a} & P(a,a,x) \underline{\times} P(x,a,a) & \xrightarrow{\widehat{\alpha}} & P(a,a,a,a)
\end{array}$$

The following diagram for surjective pairing must commute:



Finally, the following diagram for the preservation of separation structure must commute:



Definition 4.2.9 (Weakening and Duplication in $[\mathcal{C}, \mathbf{Set}]$) Given a proseparation category $\mathcal{C}$, with weakening and duplication define define *dup* as the composite:

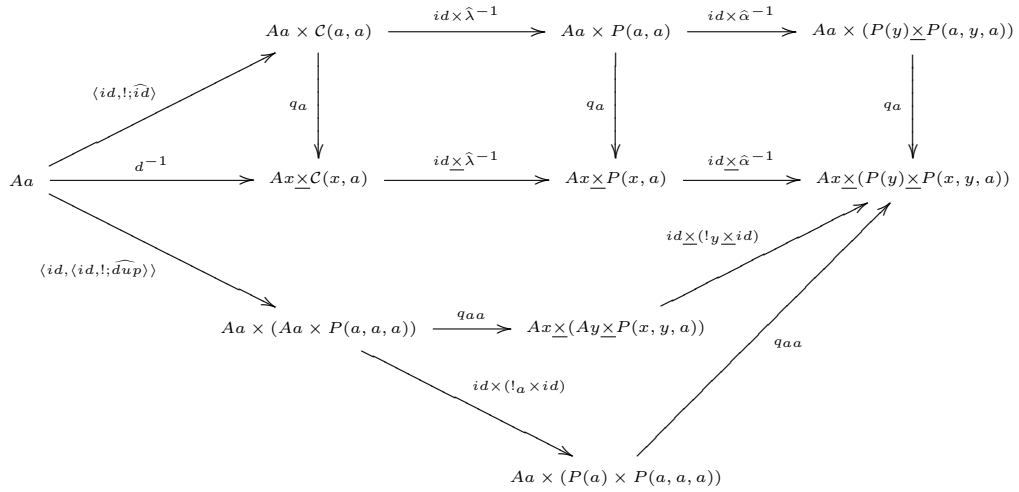
$$\begin{array}{c} Ax \\ \xrightarrow{\langle id,\langle id,!;dup\rangle\rangle} \\ Ax \times (Ax \times P(x,x,x)) \end{array}$$

$$\xrightarrow{q_{xx}} \qquad Aa \underline{\times} (Ab \underline{\times} P(a,b,x))$$

Proposition 4.2.10 Definition 4.2.9 defines the required discarding and duplication structure in $[\mathcal{C}, \mathbf{Set}]$, where $\underline{[]_2}(A,B)$ is the product.

Proof The defined family of arrows is clearly natural in A. Naturality in x follows from the dinaturality of $\widehat{dup}$. Since $P_{[]_0}(a)$ is the single element set, and $\underline{[]_0}()(a) = P_{[]_0}(a)$ by Definition 4.2.1, then $\underline{[]_0}()$ is the unique (up to isomorphism) terminal object in $[\mathcal{C}, \mathbf{Set}]$;

The diagram for right unit in Definition 4.2.8 implies the diagram for right unit in 3.2.18. The left unit case is similar. Consider the following diagram, which contains the definitional unfolding of the required diagram for $[\mathcal{C}, \mathbf{Set}]$ in the centre:



Along the top row, the first triangle commutes by the definition of d^{-1} and the two squares commute by the definition of the $\underline{\times}$ notation. The bottom "square" also commutes by the definition of the $\underline{\times}$ notation. The outer edges of the diagram are equal by the properties of finite products and the diagram for right unit in Definition 4.2.8. Hence the inner diagram commutes, as required.

The axiom for associativity may be seen to hold in similar fashion by writing out the required diagram in terms of the definitions of α and dup, then noting that the commutativity of the associativity diagram for $\widehat{dup}$ implies the commutativity of this diagram, using Lemma 4.1.11, naturality and the properties of finite products.

Similarly, the required diagram for separation preservation holds; the two sides of the diagram generated by unfolding the definitions can be shown to be equal to two arrows derived from the two sides of the diagram in Definition 4.2.8 for separation preservation. The proof relies on the uniqueness property of the arrows induced by Lemma 4.1.11.

Finally, the case for surjective pairing is similar to the previous case. The proof again relies on the uniqueness property of the arrows induced by Lemma 4.1.11. □

4.2.4 Separation Closure

We now show that, given the definition of a category with proseparation structure, the resulting functor category is separation closed.

Proposition 4.2.11 When $\mathcal{C}$ has proseparation structure then the category $[\mathcal{C}, \mathbf{Set}]$ is separation closed. The function objects are given by:

$$[\overrightarrow{A} \xrightarrow{\mathrm{S}} B]a_0 = \int_c [A_1 a_1 \underline{\times} (\ldots (A_n a_n \underline{\times} P_{\mathrm{S}}(a_0, \overrightarrow{a}, c))), Bc]$$

Proof We show that the functor $[\overrightarrow{A} \xrightarrow{\mathrm{S}} -]$, defined on objects as above, is right adjoint to the functor $\mathrm{S}(-, \overrightarrow{A})$ in $[\mathcal{C}, \mathbf{Set}]$.

$$\begin{aligned}
& [\mathcal{C}, \mathbf{Set}](\mathrm{S}(X, \overrightarrow{A}), B) \\
= & \int_c \left[\int^{a_0} Xa_0 \times (A_1 a_1 \underline{\times} \ldots P_{\mathrm{S}}(a_0, \overrightarrow{a}, c)), Bc \right] \\
\cong & \int_c \int_{a_0} [Xa_0 \times (A_1 a_1 \underline{\times} \ldots P_{\mathrm{S}}(a_0, \overrightarrow{a}, c)), Bc] \\
\cong & \int_c \int_{a_0} [Xa_0, [(A_1 a_1 \underline{\times} \ldots P_{\mathrm{S}}(a_0, \overrightarrow{a}, c)), Bc]] \\
\cong & \int_{a_0} \int_c [Xa_0, [(A_1 a_1 \underline{\times} \ldots P_{\mathrm{S}}(a_0, \overrightarrow{a}, c)), Bc]] \\
\cong & \int_{a_0} \left[Xa_0, \int_c [(A_1 a_1 \underline{\times} \ldots P_{\mathrm{S}}(a_0, \overrightarrow{a}, c)), Bc] \right] \\
= & \int_{a_0} \left[Xa_0, \left[\overrightarrow{A} \xrightarrow{\mathrm{S}} B \right] a_0 \right] \\
= & [\mathcal{C}, \mathbf{Set}](X, [\overrightarrow{A} \xrightarrow{\mathrm{S}} B])
\end{aligned}$$

In order, these lines are justified by: Definition 4.2.1 and Proposition 4.1.5; Proposition 4.1.8; Currying; Proposition 4.1.9; Proposition 4.1.8; the definition above; and Proposition 4.1.5. Each of the constituent parts is a natural isomorphism, so the resulting isomorphism of homsets is natural in B and X. □

Summing up the results of this section we have the following theorem:

Theorem 4.2.12 If $\mathcal{C}$ is a small category with proseparation structure as defined in Definition 4.2.1, Permutation as defined in Definition 4.2.4, S-Weakening as defined in Definition 4.2.5 and weakening and duplication as defined in Definition 4.2.8 then the functor category $[\mathcal{C}, \mathbf{Set}]$ is a λ_{sep}-category.

Proof This is Propositions 4.2.3, 4.2.7, 4.2.10 and 4.2.11. □

4.3 Instances of Proseparation Categories

The previous section has shown that, given the correct structure on a category $\mathcal{C}$, the functor category $[\mathcal{C}, \mathbf{Set}]$ can interpret λ_{sep}. In this section we shall give three examples of proseparation structure.

The first example assumes that $\mathcal{C}$ has separation structure and shows how it can be used to define proseparation structure. The second and third examples show how λ_{sep} models separation of resources.

4.3.1 Separation Categories

It is possible to dualise the constructions of Section 4.2 so that we require functors $P_{\text{S}} : \mathcal{C}^{|\text{S}|} \times \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$. The rest of the definitions can be dualised appropriately and the construction of separation structure takes place in the category $[\mathcal{C}^{\mathsf{op}}, \mathbf{Set}]$.

If $\mathcal{C}$ has separation structure with $\underline{[]}_0()$ the terminal object and $\underline{[]}_2(A, B)$ the product then it also has proseparation structure. Start by defining the functors P_{S}:

$$P_{\text{S}}(a_1, \ldots, a_n, a) = \mathcal{C}(a, \underline{\text{S}}(a_1, \ldots, a_n))$$

These clearly have the correct variances by the variances of the homset functors. Set $\widehat{\lambda}$ to be the Yoneda embedding of the λ arrows:

$$\begin{array}{rl} & \mathcal{C}(b,a) \\ \xrightarrow{\mathcal{C}(id,\lambda_a)} & \mathcal{C}(b,\underline{[]_0}(a)) \\ = & P_{[]_0}(a,b) \end{array}$$

And set $\widehat{\alpha}$ to be the composite:

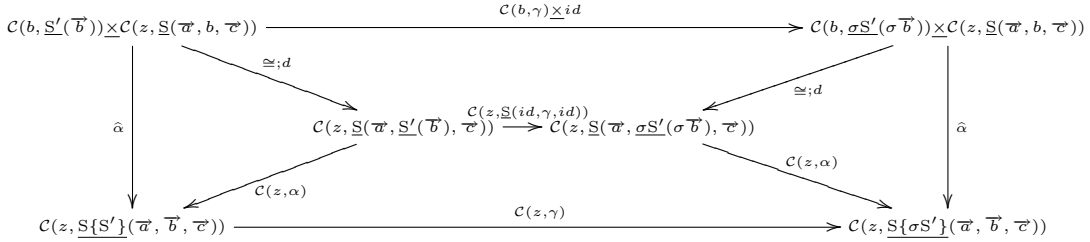
$$\begin{array}{rl} & P_{\mathrm{S}'}(\overrightarrow{b},b)\underline{\times}P_{\mathrm{S}}(\overrightarrow{a},b,\overrightarrow{c},x) \\ = & \mathcal{C}(b,\underline{\mathrm{S}'}(\overrightarrow{b}))\underline{\times}\mathcal{C}(x,\underline{\mathrm{S}}(\overrightarrow{a},b,\overrightarrow{c})) \\ \xrightarrow{\cong;d} & \mathcal{C}(x,\underline{\mathrm{S}}(\overrightarrow{a},\underline{\mathrm{S}'}(\overrightarrow{b}),\overrightarrow{c})) \\ \xrightarrow{\mathcal{C}(x,\alpha)} & \mathcal{C}(x,\underline{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a},\overrightarrow{b},\overrightarrow{c})) \\ = & P_{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a},\overrightarrow{b},\overrightarrow{c},x) \end{array}$$

For permutation and S-weakening, set $\widehat{\gamma[\sigma_{\mathrm{S}}]}$ to be $\mathcal{C}(id,\gamma[\sigma_{\mathrm{S}}])$ and $\widehat{\zeta[\mathrm{S},\mathrm{S}']}$ to be $\mathcal{C}(id,\zeta[\mathrm{S},\mathrm{S}'])$. Lastly, define $\widehat{dup_a}$ as the function $1 \to \mathcal{C}(a,\underline{[]_2}(a,a))$ whose value is always the dup_a arrow in $\mathcal{C}$.

With these definitions we have the following result:

Theorem 4.3.1 The definitions above define proseparation structure on $\mathcal{C}$ with permutation, S-weakening, duplication and weakening.

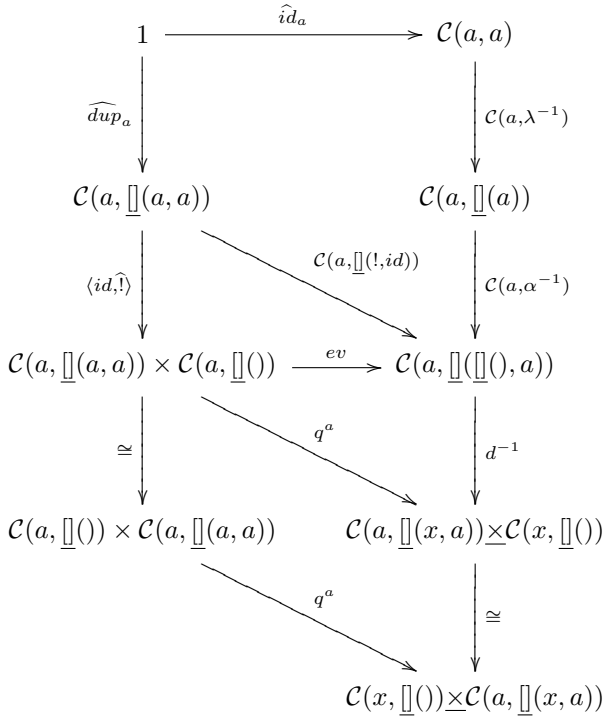
Proof The definitions should obey the axioms required for a proseparation category. To save space, we only show two illustrative instances. Firstly, for the first axiom for $\widehat{\gamma}$, consider the following diagram:



The left and right inner regions commute by the definition of $\widehat{\alpha}$; the upper region commutes by the naturality of d; and the lower region commutes by the appropriate axiom for γ. Hence, the outer diagram commutes, as required.

The second axiom for $\widehat{\gamma}$ and the rest for $\widehat{\zeta}$, $\widehat{\lambda}$ and $\widehat{\alpha}$ can be shown to hold in a similar way. The axiom describing the interaction between $\widehat{\gamma}$ and $\widehat{\zeta}$ is a trivial consequence of the appropriate axiom for γ and ζ.

Secondly, consider the following diagram for the left unit diagram for $\widehat{dup}$, where the outer edge is the one required to commute:



The top inner region of the diagram commutes by the comonoid axioms for *dup*; the next region down commutes by the properties of homset and functors and their evaluation; the next one commutes by the properties of d; and the final region commutes by Lemma 4.1.11. Hence, the whole diagram commutes, as required.

The other diagrams for associativity, surjective pairing and separation preservation all commute for similar reasons. Finally, $\widehat{dup}$ is a dinatural transformation since *dup* is a natural transformation. □

We can simplify the expression of the objects of the closed structure of Proposition 4.2.11 in this case:

$$
\begin{aligned}
& [\overrightarrow{A} \xrightarrow{\mathrm{S}} B]a_0 \\
= & \int_c [A_1 a_1 \underline{\times} (\ldots (A_n a_n \underline{\times} \mathcal{C}(c, \underline{\mathrm{S}}(a_0, \ldots, a_n)))), Bc] \\
\cong & \int_c \int_{a_1} \ldots \int_{a_n} [A_1 a_1 \times \ldots \times A_n a_n \times \mathcal{C}(c, \underline{\mathrm{S}}(a_0, \ldots, a_n)), Bc] \\
\cong & \int_c \int_{a_1} \ldots \int_{a_n} [A_1 a_1 \times \ldots \times A_n a_n, [\mathcal{C}(c, \underline{\mathrm{S}}(a_0, \ldots, a_n)), Bc]] \\
\cong & \int_{a_1} \ldots \int_{a_n} [A_1 a_1 \times \ldots \times A_n a_n, \int_c [\mathcal{C}(c, \underline{\mathrm{S}}(a_0, \ldots, a_n)), Bc]] \\
\cong & \int_{a_1} \ldots \int_{a_n} [A_1 a_1 \times \ldots \times A_n a_n, B(\underline{\mathrm{S}}(a_0, \ldots, a_n))]
\end{aligned}
$$

where all the isomorphisms follow from the propositions in Section 4.1 and are natural in $A_1, ..., A_n, B$ and a_0.

Thus this construction may be used to add separation closure to any non-closed Separation Category, such as **SepCtxt**, defined in Section 3.2.5. This construction also preserves the existing separation structure, in the sense of Definition 3.2.23:

Theorem 4.3.2 When $\mathcal{C}$ has proseparation structure, the Yoneda embedding $Y : \mathcal{C} \to [\mathcal{C}^{\mathsf{op}}, \mathbf{Set}]$ is a strong separation functor.

Proof Define m_{S} to be the repeated application of the d isomorphisms:

$$
\begin{aligned}
& \underline{\mathrm{S}}(Y a_1, \ldots, Y a_n) \\
= \quad & \mathcal{C}(a'_1, a_1) \underline{\times} (\ldots \underline{\times} \mathcal{C}(-, \mathrm{S}(a'_1, \ldots, a'_n))) \\
\xrightarrow{\overrightarrow{id} \underline{\times} d} \quad & \mathcal{C}(a'_1, a_1) \underline{\times} (\ldots \underline{\times} \mathcal{C}(-, \mathrm{S}(a'_1, \ldots, a_n))) \\
& \ldots \\
\xrightarrow{d} \quad & \mathcal{C}(-, \underline{\mathrm{S}}(a_1, \ldots, a_n)) \\
= \quad & Y(\underline{\mathrm{S}}(a_1, \ldots, a_n))
\end{aligned}
$$

Each of the required diagrams from Definition 3.2.23 can easily be seen to hold by writing them out in terms of the definitions. □

4.3.2 Resources with Separation and Combination

The previous section gave a general family of proseparation structures, but we still do not have a connection between the calculus and something resembling resources with separation. We rectify this in this section and the next.

Definition 4.3.3 A *Resource Category* is a category $\mathcal{R}$ with finite coproducts and a finite product preserving functor $\# : \mathcal{R}^{\mathsf{op}} \times \mathcal{R}^{\mathsf{op}} \to \mathbf{Set}$ that is symmetric in the sense that $r_1 \# r_2 = r_2 \# r_1$ for all objects r_1, r_2, and similarly for arrows.

We interpret the objects of $\mathcal{R}$ as representing the actual resources we are concerned about. An arrow $r_1 \to r_2$ implies that anything that is possible with resources r_1 is also possible with resources r_2. This is captured by the functorial action of the objects of $[\mathcal{R}, \mathbf{Set}]$. The existence of finite sums allows us to consider composite resources.

The functor $-\#-$ is a separation predicate on the objects of $\mathcal{R}$. If $r_1 \# r_2 \neq \emptyset$ then we regard r_1 and r_2 as being separate resources. This will be used to give meaning to the separation relations. The functor is contravariant so that if two resources are separate, and we take two "lesser" resources, then they will also be separate. The fact that this functor is finite product preserving implies the following isomorphism exists:

$$r\#(r_1 + ... + r_n) \cong r\#r_1 \times ... \times r\#r_n$$

That is, a resource separate from a composite resource is also separate from them individually. The symmetry requirement matches the definition of separation relations as a symmetric relation.

Example 4.3.4 Let X be some set of individual resources. Take $\mathcal{R}$ to be the category with objects the power set of X and an arrow $r_1 \to r_2$ iff $r_1 \subseteq r_2$. This $\mathcal{R}$ has finite sums given by set union and a separation predicate given by:

$$r_1 \# r_2 = \begin{cases} \{*\} & r_1 \cap r_2 = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

It is easy to see that $\mathcal{R}$ so defined is a resource category.

Given a resource category $\mathcal{R}$, define proseparation functors as:

$$P_{\mathrm{S}}(r_1, \ldots, r_n, s) = \overrightarrow{\mathcal{R}(r, s)} \times \Pi_{(i,j)\in \mathrm{S}} r_i \# r_j$$

Define $\widehat{\alpha}$ as the composite:

$$\begin{array}{lll}
 & \int^y P_{\mathrm{S}'}(\overrightarrow{b}, y) \times P_{\mathrm{S}}(\overrightarrow{a}, y, \overrightarrow{c}, x) & \\
= & \int^y (\overrightarrow{\mathcal{R}(b, y)} \times S') \times (\overrightarrow{\mathcal{R}(a, x)} \times \mathcal{R}(y, x) \times \overrightarrow{\mathcal{R}(c, x)} \times S) & \\
\cong & (\overrightarrow{\mathcal{R}(a, x)} \times \overrightarrow{\mathcal{R}(c, x)} \times S_{\not y} \times S') \times \int^y (\mathcal{R}(y, x) \times S_y) \times \mathcal{R}(b_1 + \ldots + b_n, y) & \\
\overset{id\times d_{\mathsf{op}}}{\cong} & (\overrightarrow{\mathcal{R}(a, x)} \times \overrightarrow{\mathcal{R}(c, x)} \times S_{\not y} \times S') \times (\mathcal{R}(b_1 + \ldots + b_n, x) \times S_{b_1+\ldots+b_n}) & \\
\cong & \overrightarrow{\mathcal{R}(a, x)} \times \overrightarrow{\mathcal{R}(b, x)} \times \overrightarrow{\mathcal{R}(c, x)} \times S\{S'\} & \\
= & P_{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}, x) &
\end{array}$$

where the terms S, S' and $S\{S'\}$ stand for the products of all the separation predicates required for the separation relations S, S′ and S{S′} respectively. The terms $S_{\not y}$ and S_y stand for the two parts of S not involving and involving the object y respectively, and $S_{b_1+\ldots+b_n}$ is S_y with all instances of y replaced by $b_1 + \ldots + b_n$.

The transformation $\widehat{\lambda} : P_{[]_1}(a, b) \cong \mathcal{R}(a, b)$ is the obvious isomorphism $\mathcal{R}(a, b) \times 1 \cong \mathcal{R}(a, b)$. The transformations $\widehat{\gamma}$ and $\widehat{\zeta}$ are defined by the obvious isomorphism and use of projection on the product of separation predicates respectively. The dinatural transformation $\widehat{dup}$ is defined as:

$$\widehat{dup_a} = \langle \widehat{id_a}, \widehat{id_a} \rangle : 1 \to \mathcal{R}(a, a) \times \mathcal{R}(a, a) = P_{[]_2}(a, a, a)$$

where $\widehat{id_a}$ maps the single element of 1 to the identity arrow in $\mathcal{R}(a, a)$.

Theorem 4.3.5 The above definitions define $\mathcal{R}$ as a category with proseparation structure with S-weakening, permutation, duplication and discarding.

Proof We check each of the conditions for the structure in turn. The P_{S} so defined are clearly functors by their construction. The families of arrows $\widehat{\alpha}$ are natural isomorphisms by their construction from natural isomorphisms. That they obey the two required commuting diagrams can be seen by noting that the

conditions boil down to the commuting of two instances of d_{op}, which are Day's Lemmas 2.10 and 2.11 [Day70].

The family $\widehat{\lambda}$ so defined is clearly a natural isomorphism. It obeys the two required interaction diagrams with $\widehat{\alpha}$ by a direct consequence of the definitions. The families defined for Permutation and S-Weakening are natural transformations by definition, and they clearly obey the required algebraic equations. The commuting diagrams for permutation can be seen to hold by writing them out in terms of the definitions and observing that since the functors $\mathcal{R}(-, x)$ and $-\#-$ preserve products on $\mathcal{R}^{\mathsf{op}}$ we can commute the permutation and flattening. The commuting diagrams for S-weakening also hold by writing them out in terms of their definitions. Similarly, the interaction diagram between permutation and S-Weakening is easily seen to hold.

The value of $P_{[]_1}(a)$ is always 1, the terminal object in **Set**. The family of arrows $\widehat{dup}$ is dinatural since $\widehat{id}$ is. The four diagrams for duplication all hold because of the way $\widehat{dup}$ is defined in terms of identity arrows and the equation for the inverse of the density formula. □

Corollary 4.3.6 Given a Resource Category $\mathcal{R}$, the functor category $[\mathcal{R}, \mathbf{Set}]$ is a λ_{sep}-category.

Given $[\mathcal{R}, \mathbf{Set}]$ as a model of λ_{sep}, we can use the Yoneda embedding to add extra types to the calculus to represent particular resources. For any object r of a resource category $\mathcal{R}$ define a new type Y_r interpreted by Yoneda:

$$[\![Y_r]\!] = \mathcal{R}(r, -)$$

Thus, in the case of Example 4.3.4 the meaning of Y_r is the set of all resources containing r. We can use this to specify fixed resources that other values must be separate from. 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:

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

The representation of named resources in the calculus has a precedent in the *nominals* of hybrid logic. See, e.g. [AB01].

4.3.3 Finite Sets and Injective Functions

Our next example of a category with proseparation structure in the category of finite sets and injective functions, **I**. We will define notions of separation and combination in **I** that are distinct from the Resource Categories of the previous subsection. This will give a different view, based on more localised notion of separation rather than the global separation of the previous section.

We note that the category **I** has symmetric monoidal structure, given on objects by disjoint union. This can be used, via Day's construction to model the $\alpha\lambda$-calculus. In fact, **I** is the free affine symmetric monoidal category over the one object, one arrow category.

Our notion of separation in **I** is based on comparisons between the maps from the bound variables of the coend defining separation products to the containing resource. Thus separation is determined by how the sub-resources fit together locally, rather than by a global predicate as was the case with Resource Categories.

Definition 4.3.7 Two arrows with common codomain, $f_1 : a_1 \to x$ and $f_2 : a_2 \to x$, in **I** are separate, $f_1 \# f_2$ if the ranges of f_1 and f_2 are disjoint.

Lemma 4.3.8 Some properties of separation:

1. If $f_1 \# f_2$ then $f_2 \# f_1$;

2. If $f_1 \# f_2$, then for all $f : x \to y$, $(f_1; f) \# (f_2; f)$;

3. If $f_1 \# f_2$, then for all $f : a_2' \to a_2$, $f_1 \# (f; f_2)$.

Proof Property 1 is trivial. Property 2 follows from the injectivity of f. Property 3 follows because pre-composition does not affect the range of f_2. □

Definition 4.3.9 Given a finite collection of arrows $\langle f_i : a_i \to x \rangle_{1 \leq i \leq n}$, form the *combination* of them: $\bigsqcup_i f_i$, equal to the union of the ranges of all the f_i.

Each of the arrows f_i factors through $\bigsqcup_i f_i$ as $f_i^{\#}; inc_{\bigsqcup_i f_i}$, where $f_i^{\#}$ is f_i with codomain restricted to its range and $inc_{\bigsqcup_i f_i}$ is the inclusion of $\bigsqcup_i f_i$ in x.

We now prove a lemma detailing how the separation and combination operations interact. This is what allows us to successfully model the flattening and unflattening structural rules.

Lemma 4.3.10 Given an arrow $f : a \to x$, a finite collection of arrows $\langle f_i : b_i \to x \rangle_{1 \leq i \leq n}$ and a separation relation S, $|S| = n$, such that for all i, $f \# f_i$ and for all $(i, j) \in$ S, $f_i \# f_j$, then $f \# inc_{\bigsqcup_i f_i}$ and for all $(i, j) \in$ S, $f_i' \# f_j'$.

Proof The range of inc is the union of the ranges of the f_i, so if the range of f is disjoint from all of them, it is disjoint from the union. Hence, $f \# inc$. For each of the f_i', the restriction of its codomain does not change its range, so $f_i \# f_j$ implies $f_i' \# f_j'$. □

Lemma 4.3.11 Two properties concerning the interaction of composition and combination. In both cases assume a finite collection of arrows $\langle f_i : a_i \to x \rangle_{1 \leq i \leq n}$.

1. Given an arrow $h : x \to x'$, there is an arrow $h^\dagger : \bigsqcup_i f_i \to \bigsqcup_i (f_i; h)$ such that $inc_{\bigsqcup_i f_i}; h = h^\dagger; inc_{\bigsqcup_{f_i;h}}$ and for all i, $f_i^{\#}; h^\dagger = (f_i; h)^{\#}$.

2. Given arrows $\langle g_i : a_i' \to a_i \rangle_{1 \leq i \leq n}$, there is an arrow $\langle g_i \rangle^\dagger : \bigsqcup_i (g_i; f_i) \to \bigsqcup_i f_i$ such that $\langle g_i \rangle^\dagger; inc_{\bigsqcup_i f_i} = inc_{\bigsqcup_i g_i; f_i}$.

3. Given an arrow $h : x \to x'$, there is an arrow $g : \bigsqcup_i (f_i; h) \to x$ such that $h; g = inc_{\bigsqcup_i f_i; h}$ and for all i, $(f_i; h)^{\#}; g = f_i$.

Proof For property 1, define $h^\dagger$ to be the restriction of h to $\bigsqcup_i f_i$. For property 2, define $\langle g_i \rangle^\dagger$ as the inclusion. For property 3, define $g(e) = h^{-1}(e)$, which is functional because h is injective, and is total because $\bigsqcup_i (f_i; h)$ only contains elements in the range of h. The equations in all three cases follow immediately. □

Using our separation predicate we define the proseparation functors as:

$$P_{\mathrm{S}}(a_1, ..., a_n, b) = \{(f_1 : a_1 \to b, ..., f_n : a_n \to b) \quad | \quad \forall (i,j) \in S. f_i \# f_j\}$$

The action of the functors on arrows is the obvious pre- and post-composition operations. By Lemma 4.3.8 these preserve the separation predicates.

Unfortunately, it is not easily possible to define the structural natural transformations from abstract components as we have done in the previous examples. We define then by working directly with our choice of coends. With this definition, the domain of the $\widehat{\alpha}$ arrows is:

$$\begin{aligned} & \int^y P_{\mathrm{S}'}(\overrightarrow{b}, y) \times P_{\mathrm{S}}(\overrightarrow{a}, y, \overrightarrow{c}, x) \\ = \quad & \{(y, \overrightarrow{f_b : b \to y}, \overrightarrow{f_a : a \to x}, f_y : y \to x, \overrightarrow{f_c : c \to x}) \quad | \quad \mathrm{S} \wedge \mathrm{S}'\} / \approx \end{aligned}$$

where S and S′ represent the separation predicates required by the separation relations and where $\approx$ is the least equivalence relation such that:

$$\begin{aligned} & (y, \overrightarrow{f_b}, \overrightarrow{f_a}, f_y, \overrightarrow{f_c}) \approx (y', \overrightarrow{f'_b}, \overrightarrow{f'_a}, f'_{y'}, \overrightarrow{f'_c}) \\ \text{if} \quad & \exists g : y \to y'. \overrightarrow{f_a = f'_a} \wedge \overrightarrow{f_c = f'_c} \wedge \overrightarrow{f_b ; g = f'_b} \wedge g ; f'_{y'} = f_y \end{aligned}$$

Define $\widehat{\alpha} : \int^y P_{\mathrm{S}'}(\overrightarrow{b}, y) \times P_{\mathrm{S}}(\overrightarrow{a}, y, \overrightarrow{c}, x) \to P_{\mathrm{S}\{\mathrm{S}'\}}(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}, x)$ and its inverse as:

$$\widehat{\alpha}([(y, \overrightarrow{f_b}, \overrightarrow{f_a}, f_y, \overrightarrow{f_c})]) = (\overrightarrow{f_a}, \overrightarrow{f_b ; f_y}, \overrightarrow{f_c})$$

$$\widehat{\alpha}^{-1}(\overrightarrow{f_a}, \overrightarrow{f_b}, \overrightarrow{f_c}) = [(\bigsqcup_i f_{b_i}, \overrightarrow{f_b^{\#}}, \overrightarrow{f_a}, inc_{\bigsqcup_i f_{b_i}}, \overrightarrow{f_c})]$$

Before proceeding we must check that this definition is well-defined.

Proposition 4.3.12 The families $\widehat{\alpha}$ and $\widehat{\alpha}^{-1}$ so defined are functional, natural in $\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$ and x and are mutually inverse.

Proof Firstly, $\widehat{\alpha}$ is functional: if $(y, \overrightarrow{f_b}, \overrightarrow{f_a}, f_y, \overrightarrow{f_c}) \approx (y', \overrightarrow{g_b}, \overrightarrow{g_a}, g_{y'}, \overrightarrow{g_c})$ then their values under $\widehat{\alpha}$ should be equal. Note that we need only check that $\widehat{\alpha}$ respects classes with respect to the condition on $\approx$ given above, the rest follows

from the definition of $\approx$ as the least equivalence relation. If there exists $h : y \to y'$ such that $\overrightarrow{f_b; h = g_b}$ and $f_y = h; g_{y'}$ then, for all f_{b_i}:

$$f_{b_i}; f_y = f_{b_i}; h; g_{y'} = g_{b_i}; g_{y'}$$

Hence, with the fact that $\overrightarrow{f_a = g_a}$ and $\overrightarrow{f_c = g_c}$, we have:

$$\widehat{\alpha}([(y, \overrightarrow{f_b}, \overrightarrow{f_a}, f_y, \overrightarrow{f_c})]) = \widehat{\alpha}([(y', \overrightarrow{g_b}, \overrightarrow{g_a}, g_{y'}, \overrightarrow{g_c})])$$

Also, $\widehat{\alpha}$ preserves the separation structure by Lemma 4.3.8. Therefore, $\widehat{\alpha}$ is a well-defined function. The function $\widehat{\alpha}^{-1}$ also preserves the separation by Lemma 4.3.10.

The naturality of $\widehat{\alpha}$ is clear from the definition of the functorial action of P_{S}; the pre- and post-composition operations are not interfered with by $\widehat{\alpha}$. Naturality for $\widehat{\alpha}^{-1}$ follows from the first two properties of Lemma 4.3.11 and the fact that $\approx$ is an equivalence relation.

Finally, $\widehat{\alpha}$ and $\widehat{\alpha}^{-1}$ are mutually inverse:

$$\begin{aligned}
\widehat{\alpha}(\widehat{\alpha}^{-1}(\overrightarrow{f_a}, \overrightarrow{f_b}, \overrightarrow{f_c})) &= \widehat{\alpha}([(\bigsqcup_i f_{b_i}, \overrightarrow{f_b^{\#}}, \overrightarrow{f_a}, inc_{\bigsqcup_i f_{b_i}}, \overrightarrow{f_c})]) \\
&= (\overrightarrow{f_a}, \overrightarrow{f_b^{\#}; inc_{\bigsqcup_i f_{b_i}}}, \overrightarrow{f_c}) \\
&= (\overrightarrow{f_a}, \overrightarrow{f_b}, \overrightarrow{f_C})
\end{aligned}$$

where the last line follows from the factorisation property of combination noted above. Conversely:

$$\begin{aligned}
\widehat{\alpha}^{-1}(\widehat{\alpha}([(y, \overrightarrow{f_b}, \overrightarrow{f_a}, f_y, \overrightarrow{f_c})])) &= \widehat{\alpha}^{-1}(\overrightarrow{f_a}, \overrightarrow{f_b; f_y}, \overrightarrow{f_c}) \\
&= [(\bigsqcup_i (f_{b_i}; f_y), \overrightarrow{(f_b; f_y)^{\#}}, \overrightarrow{f_a}, inc_{\bigsqcup_i (f_{b_i}; f_y)}, \overrightarrow{f_c})]
\end{aligned}$$

The argument and result are equal by Lemma 4.3.11, part 3, and the definition of $\approx$. □

The natural isomorphisms $\widehat{\lambda}$ are just the identity. Permutation natural isomorphisms $\widehat{\gamma}$ just permute the tuple of arrows. By the first property in Lemma 4.3.8 the separation property is maintained. S-Weakening, $\widehat{\zeta}$, is just the inclusion

of sets of tuples of arrows. Weakening holds since $P_{\mathrm{S}}(a)$ is a single element set. Duplication $\widehat{dup}$ is defined as:

$$\widehat{dup}_a(*) = (id_a, id_a)$$

Since there is no separation required in $P_{[]_2}(a, a, a)$, this is well defined.

Theorem 4.3.13 The definitions above define proseparation structure on **I**.

Proof It remains to verify that each of the required diagrams commutes. The two diagrams for $\widehat{\alpha}$ commute since it does not matter in which order we compose the arrows. The two diagrams for $\widehat{\lambda}$ commute because $\widehat{\lambda}$ is the identity. The two diagrams for $\widehat{\gamma}$ commute because permutation of the arrows does not affect their composition with other arrows. The diagrams for $\widehat{\zeta}$ commute trivially. The interaction diagram between $\widehat{\gamma}$ and $\widehat{\zeta}$ also commutes directly. The five diagrams for $\widehat{dup}$ all commute because $\widehat{dup}$ is defined in terms of identity arrows. □

Corollary 4.3.14 The functor category $[\mathbf{I}, \mathbf{Set}]$ is separation closed.

We can relate the constructions in this section to Day's original construction by noting that $P_{[1\#2]_2}(a, b, x) \cong \mathbf{I}(a \otimes b, x)$. Therefore we have that our definition of monoidal structure in $[\mathcal{I}, \mathbf{Set}]$ by Proposition 3.2.6 is isomorphic to Day's:

$$\int^{a,b} Aa \times Bb \times P_{[1\#2]_2}(a, b, x) \cong \int^{a,b} Aa \times Bb \times \mathbf{I}(a \otimes b, x)$$

Moreover, the Yoneda embedding preserves this structure.

4.3.3.1 Pullback Preservation

We could further refine this model of separation by restricting our attention to pullback preserving functors. This has the effect that each value $e \in Ax$ determines a least object of **I** that it requires (see Section 5 in [O'H93]). That is, values determine the resources they need. Hence we can have a function *supp* that gives the unique resource required by a value. In this case the definition of separation products is very simple:

$$\underline{\mathrm{S}}(A_1, ..., A_n)x = \{(a_1, ..., a_n) \in A_1x \times ... \times A_nx \quad | \quad \forall (i, j) \in \mathrm{S}.supp(a_i) \cap supp(a_j) = \emptyset\}$$

It is also the case now that the S-Weakening arrows are monomorphisms.

We do not follow up these ideas here, except to say that the category of pullback preserving functors $\mathbf{I} \to \mathbf{Set}$ is equivalent to the category of sheaves over $\mathbf{I}^{\mathsf{op}}$ with the atomic topology [MM92, Joh89] and also the category of nominal sets [GP01]. Nominal sets are the objects of FM-set theory, set theory over some collection of atoms $\mathbb{A}$. This set of atoms provides the resources on which our separation constraints can operate.

Chapter 5

Typed Computational Effects

In this chapter we develop the constructions for typed computational effects sketched in the introduction. We will use this in the next chapter to provide a categorical semantics for a calculus with explicit state types, and in Chapter 8 for an in-place update calculus.

We first review two existing definitions for the categorical semantics of computational effects: Freyd categories and strong monads. The definition we give for Freyd categories is different to the one in the literature [PR97, PT99], but we prove the two equivalent and our definition will be easier to extend to the typed computational effects case. We also give some examples of Freyd categories and strong monads, taken from the literature, showing how they are used to model various computational effects.

In Section 5.2, we give our definitions for typed computational effects. The first is parameterised Freyd categories in Section 5.2.1, an extension of the definition of Freyd categories. We also define a notion of closure for parameterised Freyd categories and give some examples of parameterised Freyd categories, modelling global typed state, category actions and composable continuations. Following this in Section 5.2.2 we give our definition of parameterised strong monads, an extension of strong monads. In Section 5.2.3 we show that, assuming closure, the two definitions are equivalent up to isomorphism.

We build on these definitions in Section 5.3 by extending both definitions to deal with the lifting of typed computational effects to larger effect types. In

the case of state, this will allow the embedding of local computations in larger states, in the sense of Separation Logic [Rey02]. We prove that the two extended definitions are equivalent, up to isomorphism, in Section 5.3.3.

In Section 5.4 we define two extra conditions that apply to parameterised Freyd categories and parameterised monads and show that they extend the equivalences of the previous sections. We consider the mono requirement for parameterised monads, copying the mono requirement for normal monads [Mog91], and extend this to a condition equivalent to the Kleisli functor being full and faithful for certain objects of the parameterising category. We also consider an appropriate notion of commutativity for double parameterised Freyd categories and monoidal parameterised strong monads. Finally in Section 5.4 we define a second notion of closure for double parameterised Freyd categories. We call a closed double parameterised Freyd category with this additional form of closure a Typed Command Category.

The original idea for parameterised monads came from a post on the haskell-cafe mailing list by Chung-chieh Shan[1]. The technical definitions of parameterised monads and the rest of the work in this chapter are the work of the author.

5.1 Computational Effects

5.1.1 Freyd Categories

As mentioned in the chapter introduction, the definition of Freyd category given by Power and Robinson [PR97] and Power and Thielecke [PT99] is different to ours. We give their definition in Section 5.1.1.1 and prove that it is equivalent. The definition that follows will be easier to extend to the parameterised case in Section 5.2.

In Power and Thielecke's original definition they require the base category $\mathcal{C}$ to have finite products. We generalise this slightly to requiring symmetric monoidal structure. Let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be a symmetric monoidal category.

[1] `http://haskell.org/pipermail/haskell-cafe/2004-July/006448.html`

Definition 5.1.1 Freyd category structure on $\mathcal{C}$ consists of a category $\mathcal{K}$ and three functors:

$$J : \mathcal{C} \to \mathcal{K} \qquad \circledcirc : \mathcal{C} \times \mathcal{K} \to \mathcal{K} \qquad \circledcirc : \mathcal{K} \times \mathcal{C} \to \mathcal{K}$$

Such that:

1. J is identity on objects;
2. The monoidal structure of $\mathcal{C}$ is respected: $A \circledcirc JB = JA \circledcirc B = J(A \otimes B)$ and $f \circledcirc Jg = Jf \circledcirc g = J(f \otimes g)$;
3. The family of arrows $J\alpha_{A,B,C}$ is natural in the following pairs of functors:
   $(-_1 \circledcirc -_2) \circledcirc -_3 \to -_1 \circledcirc (-_2 \otimes -_3) \qquad (-_1 \circledcirc -_2) \circledcirc -_3 \to -_1 \circledcirc (-_2 \circledcirc -_3)$
   $$(-_1 \otimes -_2) \circledcirc -_3 \to -_1 \circledcirc (-_2 \circledcirc -_3)$$
   The family of arrows $J\sigma_{A,B}$ is natural in the following pairs of functors:
   $$-_1 \circledcirc -_2 \to -_2 \circledcirc -_1 \qquad\qquad -_1 \circledcirc -_2 \to -_2 \circledcirc -_1$$
   The family of arrows $J\lambda_A$ is natural in the pair of functors $- \circledcirc I \to -$, and similarly for the right identity family, ρ.

The functors $\circledcirc$ and $\circledcirc$ are $\mathcal{C}$-actions on the category $\mathcal{K}$ in the sense of [BCS97].

As with symmetric monoidal structure we refer to a tuple $(\mathcal{C}, \mathcal{K}, J, \circledcirc, \circledcirc)$ (where $\mathcal{C}$ itself is an abbreviation for a symmetric monoidal category) as a Freyd category. As a shorthand we will just refer to some identity-on-objects functor $J : \mathcal{C} \to \mathcal{K}$ as being a Freyd category and leave the rest of the structure as implicit. We will refer to the two functors $\circledcirc$ and $\circledcirc$, along with their required properties, as premonoidal structure. This is following Power and Robinson's definition (see Section 5.1.1.1 below), even though in our definition they cannot be fully defined without the functor J.

Also following Power and Robinson, we define the notion of *centrality*.

Definition 5.1.2 Given a Freyd category $J : \mathcal{C} \to \mathcal{K}$, an arrow $c : A \to A'$ of $\mathcal{K}$ is *central* if, for all arrows $c' : B \to B'$ of $\mathcal{K}$:

$$c \circledcirc B; A' \circledcirc c' = A \circledcirc c'; c \circledcirc B'$$

The next lemma will be useful in establishing the connection between our definition of Freyd category and that of Power, Robinson and Thielecke (Section 5.1.1.1).

Lemma 5.1.3 Given a Freyd category $J : \mathcal{C} \to \mathcal{K}$, all arrows of the form Jf are central.

Proof For any arrows $f : A \to A'$ of $\mathcal{C}$ and $c : B \to B'$ of $\mathcal{K}$:

$$Jf \circledcirc B; A' \circledcirc c = J(f \otimes B); A' \circledcirc c = f \circledcirc c = A \circledcirc c; J(f \otimes B') = A \circledcirc c; Jf \circledcirc B$$

□

Definition 5.1.4 A Freyd category $J : \mathcal{C} \to \mathcal{K}$ is *closed* if the functor $- \circledcirc B$ has a specified right adjoint for all objects B.

Closure is used to interpret function types and plays a crucial role in the equivalence between Freyd categories and strong monads. We will use the notation $B \to -$ for the chosen right adjoint to $- \circledcirc B$. By Mac Lane's Theorem §IV.7.3 [Mac98], the adjunctions give a functor $- \to - : \mathcal{K}^{\mathsf{op}} \times \mathcal{K} \to \mathcal{C}$. We will use Λ for the isomorphism of homsets:

$$\Lambda : \mathcal{K}(A \otimes B, C) \cong \mathcal{C}(A, B \to C)$$

and ev for the counit:

$$\mathrm{ev}_{A,B} : (A \to B) \otimes A \to B$$

We now give some examples of Freyd categories that model some kind of computational effect: global state, tracing and continuations. It is possible to model many other kinds of computational effect with Freyd categories: see [BHM02] for other examples in terms of strong monads such as non-determinism and interactive input/output, also Moggi [Mog89b] and Stark [Sta94] give a strong monad for dynamic allocation. By Theorem 5.2.17, these strong monad examples can be translated into Freyd category examples.

Example 5.1.5 (Global State) Pick an object S of a symmetric monoidal category $\mathcal{C}$. Define $\mathcal{K}(A, B) = \mathcal{C}(A \otimes S, B \otimes S)$ and define J as identity on objects and $Jf = f \otimes S$ on arrows. Composition in $\mathcal{K}$ is just composition in $\mathcal{C}$. Define (for $f : A \to A'$ and $c : B \to B'$ in $\mathcal{K}$):

$$f \circledcirc c = \alpha; f \otimes c; \alpha^{-1} \qquad c \circledcirc f = \sigma \otimes S; \alpha; f \otimes c; \alpha^{-1}; \sigma^{-1} \otimes S$$

It is easy to check that this defines a Freyd category.

This Freyd category is closed when $\mathcal{C}$ is closed. A choice for the closure functor is given by:

$$A \to B = (A \otimes S) \multimap (B \otimes S)$$

The isomorphism of homsets is derived directly from the chosen closure on $\mathcal{C}$.

By construing the object S as representing possible states, we can see how this example models global state. Pure value-only computations are modelled in $\mathcal{C}$, while arrows in $\mathcal{K}$ have a "hidden" state component. The functor J embeds the value-only computations of $\mathcal{C}$ into $\mathcal{K}$ by pairing them with the identity state arrow. The two functors $\circledcirc$ and $\circledcirc$ provide a way of lifting a computation up to a larger context.

When $\mathcal{C}$ is **Set**, we can define store and lookup operations suitable for modelling a simple imperative language. Choose some set L of locations and a set V of values. Set $S = V^L$ and define:

$$\mathit{store} = ((l, v), s) \mapsto (*, s[l \mapsto v]) \in \mathcal{K}(L \times V, 1)$$

$$\mathit{lookup} = (l, s) \mapsto (s(l), s) \in \mathcal{K}(L, V)$$

where $s[l \mapsto v]$ is the function that maps l to v and every other l' to $s(l')$.

Example 5.1.6 (Tracing) Given a monoid $(M, e, \cdot)$ we can define a Freyd category $J : \mathbf{Set} \to \mathcal{K}$ where $\mathcal{K}(A, B) = \mathbf{Set}(A, B \times M)$ and $J(f) = a \mapsto (f(a), e)$. Composition in $\mathcal{K}$ is defined using the monoid operation $\cdot$ and premonoidal structure is defined using the finite product structure of **Set**. For every element m of the monoid $(M, e, \cdot)$ we have an arrow $\mathit{write}_m : 1 \to 1$ in $\mathcal{K}$ defined as $* \mapsto (*, m)$. This Freyd category can be used to model programs that emit tracing information, where the information is defined as the elements of the monoid and concatenation is the monoid operation.

Example 5.1.7 (Continuations) Given a cartesian closed category $\mathcal{C}$ with a chosen object R, the structure of a Freyd category can be used to interpret the order-sensitive aspect of continuations. Continuations abstract the concept of "the next step" of a computation and first class continuations allow the programmer to manipulate the flow control of the program. They have been used to provide denotational semantics of control flow features such as jumps and also as a programming language feature in their own right. See [Rey93] for a history of continuations and [SF89] for examples of the use of first-class continuations in programming. The languages Scheme [CKR98] and SML/NJ (see [HDM93]) have first-class continuations.

Define $\mathcal{K}(A, B) = \mathcal{C}(R^B, R^A)$ for a chosen object R of $\mathcal{C}$, with the identity-on-objects functor J defined as $Jf = R^f$. The functor $\circledcirc$ is defined as (for $f : A \to A'$ in $\mathcal{C}$ and $c : B \to B'$ in $\mathcal{K}$):

$$f \circledcirc g = \lambda kx.g(\lambda b.k(f(\pi_1 x), b))(\pi_2 x)$$

in the internal language of $\mathcal{C}$. The functor $\circledgt$ is defined similarly. It is easy to check that this defines a Freyd category.

Since $\mathcal{C}$ is closed, this Freyd category is closed. A choice for the right adjoint is:

$$A \to B = R^B \Rightarrow R^A$$

using $\Rightarrow$ and $-^-$ both to stand for the exponential functor of $\mathcal{C}$. The isomorphism of homsets is defined using the adjunction for the cartesian closure of $\mathcal{C}$.

The interpretation of a programming language in this Freyd category essentially does a Continuation Passing Style transform [Plo75]. This transformation makes the current continuation available and we can use it to implement first-class continuations.

Write $cont(A)$ for R^A, the object used to interpret the type of A continuations. Define two operators, using the internal language of $\mathcal{C}$:

$$\begin{aligned} call/cc &: \mathcal{K}(\Gamma \times cont(A), A) \to \mathcal{K}(\Gamma, A) \\ call/cc &= f \mapsto \lambda k.\lambda e.fk(e, k) \end{aligned}$$

and

$$\begin{aligned} throw_B &: \mathcal{K}(\Gamma, cont(A)) \times \mathcal{K}(\Gamma, A) \to \mathcal{K}(\Gamma, B) \\ throw_B &= (f, g) \mapsto \lambda k.\lambda e.g(\lambda a.f(\lambda k'.k'a)e)e \end{aligned}$$

The *call/cc* operator captures the current continuation (the argument k) and passes it to the program. A continuation may be used via the $throw_B$ operation, which discards the current continuation (hence the arbitrary return type interpreted by B) and passes control to the provided continuation.

Thielecke [Thi97] directly defines the structure required for interpreting languages with first-class continuations by starting with the definition of a Freyd category and adding a self-adjoint functor $\neg$ for interpreting continuation types. His method has the advantage of not explicitly stating the semantics in terms of continuation passing style. See also Selinger's control categories [Sel01].

5.1.1.1 Power and Robinson's Definition

In this subsection we describe Power, Robinson and Thielecke's definition of Freyd category [PR97, PT99]. They build up the definition in parts, concentrating on the non-bifunctorality of the $A \otimes B$ operation on objects.

Definition 5.1.8 *Binoidal structure* on a category $\mathcal{K}$ consists of, for every object $A \in \mathsf{Ob}\mathcal{K}$, a pair of functors:

$$A \oslash - : \mathcal{K} \to \mathcal{K} \qquad\qquad - \oslash A : \mathcal{K} \to \mathcal{K}$$

such that $A \oslash B = A \oslash B$.

For binoidal structure $(\oslash, \oslash)$, we will write the object given by $A \oslash B = A \oslash B$ as $A \otimes B$.

Definition 5.1.9 An arrow $f : A \to A'$ of a category $\mathcal{K}$ with binoidal structure is *central* if, for all arrows $g : B \to B'$, $f \oslash B; A' \oslash g = A \oslash g; f \oslash B'$. A central natural transformation is a natural transformation with all components central.

Definition 5.1.10 *Symmetric Premonoidal Structure* consists of a category $\mathcal{K}$, binoidal structure $(\circledcirc, \circledast)$, an object I and four central natural isomorphisms:

$$\alpha_{ABC} : (A \otimes B) \otimes C \to A \otimes (B \otimes C) \qquad \lambda_A : I \otimes A \to A \qquad \rho_A : A \otimes I \to A$$

$$\sigma_{AB} : A \otimes B \to B \otimes A$$

Naturality in this case means natural in all the possible combinations of functors making up the objects. For example, σ is natural for the pairs of functors:

$$A \circledcirc - \to - \circledast A \qquad\qquad - \circledast B \to B \circledcirc -$$

As with symmetric monoidal structure and our definition of Freyd categories above, we refer to the tuple $(\mathcal{K}, \circledcirc, \circledast, I, \alpha, \lambda, \rho, \sigma)$ that satisfies Definition 5.1.10 as a symmetric premonoidal category. We will often omit everything except the $\mathcal{K}$ to save space and leave the other parts of the structure as implicit.

Definition 5.1.11 A *strict symmetric premonoidal functor* $F : \mathcal{C} \to \mathcal{K}$ between two symmetric premonoidal categories is a functor F that strictly preserves all premonoidal structure and sends central arrows to central arrows.

Definition 5.1.12 A *Freyd category* is a pair of symmetric premonoidal categories $\mathcal{C}$, $\mathcal{K}$, where the symmetric premonoidal structure on $\mathcal{C}$ is given by symmetric monoidal structure, and an identity-on-objects strict symmetric premonoidal functor $J : \mathcal{C} \to \mathcal{K}$.

Definition 5.1.13 A Freyd category $J : \mathcal{C} \to \mathcal{K}$ is *closed* if, for all objects $A \in \mathsf{Ob}\mathcal{C}$, the functor $J(- \times A)$ has a specified right adjoint.

Theorem 5.1.14 There is a bijective mapping between our Freyd category definition and that of Power, Robinson and Thielecke. This bijection extends to closed Freyd categories.

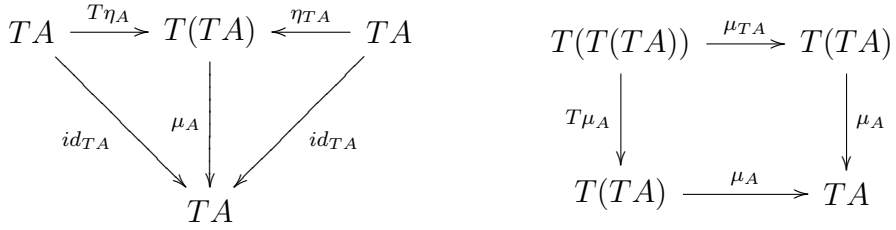
Proof The first part is a special case of the upcoming Theorem 5.3.7. For the second part, observe that, given the definitions in the proof of that theorem, the two definitions of closure are identical. □

5.1.2 Strong Monads

Moggi used strong monads to provide a categorical semantics for his Computational λ-calculus, λ_C, [Mog89a]. The computational λ-calculus is intended to be used as a meta-language for programming language semantics; the syntax of the language under study is translated into λ_C, with some extra constants, and the interpretation of λ_C in terms of strong monads provides a categorical interpretation of the original language. The distinguishing features of the source language, such as state, input-output, continuations or dynamic allocation, are modelled by choosing an appropriate monad. The point of using strong monads is that all these examples have a common requirement of the separation of values and commands and sequencing, and this is provided by strong monads.

As in the previous subsection, let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be a symmetric monoidal category.

Definition 5.1.15 A *monad* on $\mathcal{C}$ consists of an endofunctor $T : \mathcal{C} \to \mathcal{C}$, and two natural transformations: the *unit* $\eta : Id \Rightarrow T$ and the *multiplication* $\mu : T^2 \Rightarrow T$. These must obey the diagrams:



Given a monad (T, η, μ), Moggi interprets programs in a call-by-value programming language as arrows $A \to TB$. The definition of a Kleisli category forms a category of these arrows.

Definition 5.1.16 Given a monad (T, η, μ) on $\mathcal{C}$, the *Kleisli category* $\mathcal{C}_T$ has:

Objects Objects of $\mathcal{C}$;

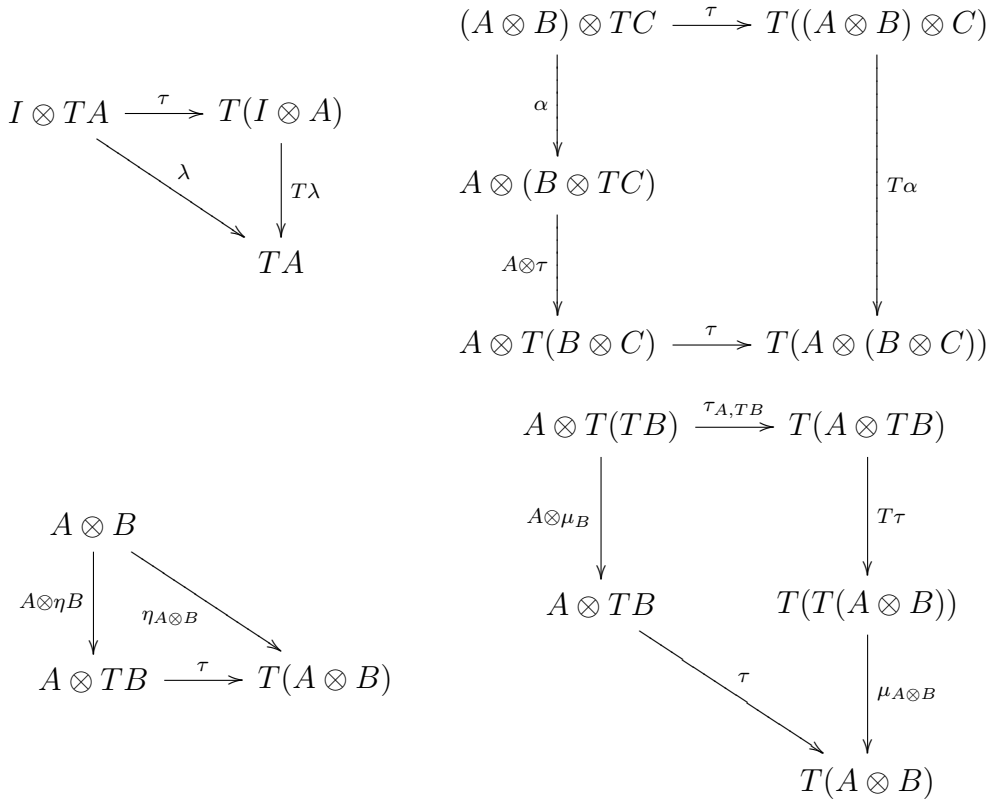
Arrows $A \to B$ Arrows $A \to TB$ of $\mathcal{C}$

Identities are defined as $\eta_A \in \mathcal{C}(A, TA) = \mathcal{C}_T(A, A)$ and composition of $f : A \to B$ and $g : B \to C$ is defined as $f; Tg; \mu_C \in \mathcal{C}(A, TC) = \mathcal{C}_T(A, C)$.

There is an identity-on-objects functor $J_T : \mathcal{C} \to \mathcal{C}_T$ defined as $J_T f = f; \eta_B$ on arrows. When constructing the equivalence with (parameterised) Freyd categories in Section 5.2.3, the Kleisli category will play the role of the category $\mathcal{K}$ in the definition of a Freyd category.

In order to correctly model typing contexts, the monad structure must interact with the symmetric monoidal structure of $\mathcal{C}$. This interaction is provided by a *strength* natural transformation. Moggi [Mog91] provides several other formulations of strength for a monad in terms of functor categories and enriched categories which serve to motivate the definition, but here we just use the formulation in terms of a natural transformation. The definition of strength for monads is originally due to Kock [Koc72].

Definition 5.1.17 Given a monad (T, η, μ), a *strength* is a natural transformation $\tau_{A,B} : A \otimes TB \to T(A \otimes B)$ that obeys the following diagrams:



Using the strength part of a strong monad we can define premonoidal structure on the Kleisli category. Given a strong monad (T, η, μ, τ), define functors (for

$f : A \to A'$ in $\mathcal{C}$ and $c : B \to B'$ in $\mathcal{C}_T$):

$$f \mathbin{\circledcirc}_T c = f \otimes c; \tau_{A',B'} \qquad\qquad c \mathbin{\circledcirc}_T f = c \otimes f; \sigma; \tau; T\sigma$$

On objects they are both equal to $\otimes$. This definition, generalised to parameterised strong monads, will lead to the equivalence between parameterised Freyd categories and parameterised strong monads in Section 5.2.3.

In order to interpret function types we use the definition of Kleisli exponential. We use the premonoidal structure just defined.

Definition 5.1.18 A symmetric monoidal category $\mathcal{C}$ with a strong monad (T, η, μ, τ) has *Kleisli Exponentials* if, for each object B, the functor $- \mathbin{\circledcirc}_T B : \mathcal{C} \to \mathcal{C}_T$ has a specified right adjoint.

As for closed Freyd categories, we use $B \to -$ for the chosen right adjoint and the following symbols for the natural isomorphism of homsets and the counit:

$$\Lambda : \mathcal{C}_T(A \otimes B, C) \cong \mathcal{C}(A, B \to C) \qquad\qquad \mathrm{ev}_{A,B} : (A \to B) \otimes A \to B$$

We now give some examples of strong monads, matching the examples of Freyd categories in Section 5.1.1. By the upcoming Theorem 5.2.17 we know that this can be done for all closed Freyd categories, but we spell out the details here to demonstrate the interpretation of computational effects in terms of strong monads. As noted above, there are many other examples of strong monads which model computational effects [BHM02, Mog89b, Mog91].

Example 5.1.19 (Global State) The global state Freyd category in Example 5.1.5 can be expressed as a strong monad, as long as the symmetric monoidal category $\mathcal{C}$ is closed. Choose an object S of $\mathcal{C}$ and define $TA = S \multimap (A \otimes S)$ with unit, multiplication and strength:

$$\eta_A = \Lambda(id_{A \otimes S}) \qquad \mu_A = S \multimap \mathrm{ev}_{S,(A \otimes S)} \qquad \tau_{A,B} = \Lambda(A \otimes \mathrm{ev}_{S,(B \otimes S)}; \alpha^{-1})$$

where Λ is the isomorphism of homsets for the closed structure and ev is the evaluation natural transformation. Since $\mathcal{C}$ is already assumed to be closed this monad has Kleisli exponentials. The operations *store* and *lookup* can be defined as for the Freyd category example.

Example 5.1.20 (Tracing) Example 5.1.6 can also be expressed as a strong monad. Given a monoid $(M, e, \cdot)$, define $T(X) = X \times M$ and $\eta_A = a \mapsto (a, e)$ and $\mu_A = ((a, m_1), m_2) \mapsto (a, m_1 \cdot m_2)$ and $\tau_{A,B} = (a, (b, m)) \mapsto ((a, b), m)$.

Example 5.1.21 (Continuations) Finally, the continuations Freyd category (Example 5.1.7) can also be expressed as a strong monad. Given a cartesian closed category $\mathcal{C}$, choose an object R and define $TA = R^{(R^A)}$ with unit, multiplication and strength (using the internal language of $\mathcal{C}$):

$$\eta_A = \lambda a.\lambda k.ka \qquad \mu_A = \lambda f.\lambda k.f(\lambda k'.k'k) \qquad \tau_{A,B} = \lambda(a, f).\lambda k.f(\lambda b.k(a, b))$$

The call-with-current-continuation and throw operators may be defined as operators on the Kleisli category in a similar way to the ones for the Freyd category.

5.2 Typed Computational Effects

5.2.1 Parameterised Freyd Categories

As described in Section 1.3.1, we extend Freyd categories $J : \mathcal{C} \to \mathcal{K}$ by adding a start and end object to each arrow of the category $\mathcal{K}$, this is used to interpret computations from some start state type to some final state type. State types are interpreted in a category $\mathcal{S}$. At this point we do not require any extra conditions on this category. Typing contexts and result types are still modelled in $\mathcal{C}$, so symmetric monoidal structure is required to model them, and premonoidal structure is required to model the lifting of a computation (interpreted as an arrow in $\mathcal{K}$) to a larger context. The functor J is now an identity-on-objects functor $\mathcal{C} \times \mathcal{S} \to \mathcal{K}$.

Our alternative definition of Freyd category is easier to extend to this new situation than Power, Robinson and Thielecke's definition because it is no longer possible to define premonoidal structure directly on $\mathcal{K}$ and have J preserve it. We must take into account the way that $\mathcal{C}$ and $\mathcal{S}$ are incorporated into $\mathcal{K}$ by J.

Let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be a symmetric monoidal category and let $\mathcal{S}$ be a category.

Definition 5.2.1 *Parameterised Freyd structure* on $\mathcal{C}, \mathcal{S}$ *with respect to* $\mathcal{C}$ consists of a category $\mathcal{K}$ and three functors:

$$J : \mathcal{C} \times \mathcal{S} \to \mathcal{K} \qquad \circledcirc : \mathcal{C} \times \mathcal{K} \to \mathcal{K} \qquad \circledcirc : \mathcal{K} \times \mathcal{C} \to \mathcal{K}$$

Such that:

1. J is identity-on-objects;
2. The monoidal structure of $\mathcal{C}$ is respected: $A \circledcirc J(B, X) = J(A, X) \circledcirc B = J(A \otimes B, X)$ and $f \circledcirc J(g, w) = J(f, w) \circledcirc g = J(f \otimes g, w)$;
3. The family of arrows $J(\alpha_{ABC}, id_S)$ must be natural in the following pairs of functors:
   $(-_1 \circledcirc -_2) \circledcirc -_3 \to -_1 \circledcirc (-_2 \otimes -_3) \qquad (-_1 \otimes -_2) \circledcirc -_3 \to -_1 \otimes (-_2 \circledcirc -_3)$
   $$(-_1 \otimes -_2) \otimes -_3 \to -_1 \otimes (-_2 \otimes -_3)$$
   The family of arrows $J(\sigma_{AB}, id_S)$ must be natural for the following pairs of functors:
   $$-_1 \circledcirc -_2 \to -_2 \circledcirc -_1 \qquad -_1 \otimes -_2 \to -_2 \otimes -_1$$
   The family of arrows $J(\lambda_A, id_X)$ must be natural in the pair of functors $- \circledcirc I \to -$, and similarly for the right identity natural transformation, ρ.

As before, we term a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{K}, J, \otimes, \circledcirc)$ a parameterised Freyd category when it satisfies this definition. We will often just refer to a functor $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ as a parameterised Freyd category and leave the rest of the structure implicit. The definition of Freyd category (Definition 5.1.1) is a special case of this definition when $\mathcal{S} = 1$.

When verifying naturality of the structure transformations it is only necessary to check that it holds for the $\mathcal{K}$ position. The naturality in the $\mathcal{C}$ arrows follows automatically from the naturality of α in $\mathcal{C}$:

$$\begin{aligned} & (\alpha, id); c \circledcirc (f \otimes g) \\ = \; & (\alpha, id); id \circledcirc (f \otimes g); c \circledcirc id \end{aligned}$$

$$\begin{aligned} &= (\alpha, id); (id, id) \circledcirc (f \otimes g); c \circledcirc id \\ &= (\alpha, id); (id \otimes (f \otimes g), id); c \circledcirc id \\ &= ((id \otimes f) \otimes g, id); (\alpha, id); c \circledcirc id \end{aligned}$$

Hence, if (α, id) is natural in the $\mathcal{K}$ argument it is natural in all arguments. A similar argument applies to the other naturality conditions for associativity and the other structure transformations.

Definition 5.2.2 A parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is *closed* if, for all $(B, S_1) \in \mathsf{Ob}\mathcal{K}$, the functor $- \circledcirc (B, S_1) : \mathcal{C} \to \mathcal{K}$ has a right adjoint.

We will use $(B, S_1) \to -$ for the chosen right adjoint. As for closed Freyd categories and Kleisli exponentials, we use the following symbols for the natural isomorphism of homsets and the counit:

$$\Lambda : \mathcal{K}((A \otimes B, S_1), (C, S_2)) \cong \mathcal{C}(A, (B, S_1) \to (C, S_2))$$

$$\mathrm{ev}_{A,B,S_1,S_2} : ((A, S_1) \to (B, S_2) \otimes A, S_1) \to (B, S_2)$$

Example 5.2.3 (Pair Categories) For any two categories $\mathcal{C}$ and $\mathcal{S}$, set $\mathcal{K} = \mathcal{C} \times \mathcal{S}$ and J to be the identity functor. If $\mathcal{C}$ has symmetric monoidal structure then J is a parameterised Freyd category with $f \circledcirc (g, s) = (f \otimes g, s)$.

The rest of our examples all generalise the examples given for Freyd categories to take advantage of the parameterisation.

Example 5.2.4 (Typed Global State) We extend Example 5.1.5. For any category $\mathcal{C}$ with symmetric monoidal structure $(\otimes, I, \alpha, \lambda, \rho, \sigma)$ and a category $\mathcal{S}$ with a functor $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$, define the category $\mathcal{K}$ as having objects: pairs of objects of $\mathcal{C}$ and $\mathcal{S}$; and homsets $\mathcal{K}((A, S_1), (B, S_2)) = \mathcal{C}(A \otimes \widehat{S_1}, B \otimes \widehat{S_2})$. Composition is carried over from $\mathcal{C}$ and identities are given by the obvious pair of identity arrows. Define $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ as the identity on objects and as $(f, s) \mapsto f \otimes \widehat{s}$ on arrows. Define (for $f : A_1 \to A_2$ and $c : (B_1, S_1) \to (B_2, S_2)$):

$$f \circledcirc c = \alpha; f \otimes c; \alpha^{-1} \qquad\qquad c \circledcirc f = \sigma \otimes \widehat{S_1}; \alpha; f \otimes c; \alpha^{-1}; \sigma \otimes \widehat{S_2}$$

It is easy to check that these definitions satisfy the properties required in Definition 5.2.1.

When the symmetric monoidal structure of $\mathcal{C}$ is closed then this Freyd category is closed. A choice for the functor is given by:

$$(A, S_1) \to (B, S_2) = (A \otimes \widehat{S_1}) \multimap (B \otimes \widehat{S_2})$$

the isomorphism of homsets is defined using the closed structure of $\mathcal{C}$.

Given these definitions we can define typed storage and retrieval operations. Assume that the symmetric monoidal structure on $\mathcal{C}$ is actually given by finite products and that $\mathcal{S}$ has a terminal object I preserved by $\widehat{\cdot}$ and define:

$$\begin{array}{rcl} store_S & : & (\widehat{S}, I) \to (1, S) \\ store_S & = & \widehat{S} \times 1 \stackrel{\langle \pi_2, \pi_1 \rangle}{\longrightarrow} 1 \times \widehat{S} \\ retrieve_S & : & (1, S) \to (\widehat{S}, S) \\ retrieve_S & = & 1 \times \widehat{S} \stackrel{\langle \pi_2, \pi_2 \rangle}{\longrightarrow} \widehat{S} \times \widehat{S} \end{array}$$

We will show in Appendix B that the parameterised monad corresponding to this parameterised Freyd category (Example 5.2.11) arises as the parameterised monad for the category of algebras with these operations obeying some axioms.

Example 5.2.5 (Category Actions) This example generalises Example 5.1.6 from monoids to categories. Take the category $\mathcal{C}$ to be **Set**. For any category $\mathcal{S}$ with small homsets, define $\mathcal{K}$ as having objects: pairs of sets and objects of $\mathcal{S}$; and homsets: $\mathcal{K}((A, S_1), (B, S_2)) = \mathbf{Set}(A, \mathcal{S}(S_1, S_2) \times B)$. Composition is defined as:

$$A \stackrel{c_1}{\longrightarrow} \mathcal{S}(S_1, S_2) \times B \stackrel{id \times c_2}{\longrightarrow} \mathcal{S}(S_1, S_2) \times \mathcal{S}(S_2, S_3) \times C \stackrel{comp \times id}{\longrightarrow} \mathcal{S}(S_1, S_3) \times C$$

where *comp* is composition in $\mathcal{S}$. Set $J(f, s) = \langle !; \widehat{s}, f \rangle$, where $\widehat{s} : 1 \to \mathcal{S}(S_1, S_2)$ is the function picking out the element s and ! is the unique map to the single element set. For $f : (A_1, S_1) \to (A_2, S_2)$ and $a : B_1 \to B_2$, define $a \circledcirc f = a \times f; \langle \pi_2, \langle \pi_1, \pi_3 \rangle \rangle$ and similarly for $f \circledcirc a$.

As an application of this consider the category **StkPrg** of very simple stack machine programs: objects are natural numbers denoting stack depths and the arrows are generated from the following rules:

$$\frac{}{0 \xrightarrow{[]} 0} \qquad \frac{i \text{ integer}}{0 \xrightarrow{[\mathsf{push}.i]} 1} \qquad \frac{}{2 \xrightarrow{[add]} 1} \qquad \frac{}{1 \xrightarrow{[dup]} 2}$$

$$\frac{a \xrightarrow{[\overrightarrow{c_1}]} b \qquad b \xrightarrow{[\overrightarrow{c_2}]} c}{a \xrightarrow{[\overrightarrow{c_1},\overrightarrow{c_2}]} c} \qquad \frac{a \xrightarrow{[\overrightarrow{c}]} b}{a+n \xrightarrow{[\overrightarrow{c}]} a+n}$$

Composition of $[\overrightarrow{c_1}] : a \to b$ and $[\overrightarrow{c_2}] : b \to c$ is defined as $[\overrightarrow{c_1}, \overrightarrow{c_2}]$, which is an arrow by these rules.

Taking $J : \mathbf{Set} \times |\mathbf{StkPrg}| \to \mathcal{K}$ as defined above, where $|\mathbf{StkPrg}|$ is the discrete category with natural numbers as objects, we can define the following arrows in $\mathcal{K}$:

$$\begin{aligned} push_n : (\mathbb{Z}, n) \to (1, n+1) &= i \mapsto (*, [\mathsf{push}.i]) \\ add_n : (1, n+2) \to (1, n) &= * \mapsto (*, [\mathsf{add}]) \\ dup_n : (1, n+1) \to (1, n+2) &= * \mapsto (*, [\mathsf{dup}]) \end{aligned}$$

where $\mathbb{Z}$ is the usual set of integers.

Thus, arrows in $\mathcal{K}$ model programs that construct stack machine programs that do not have the possibility of stack under- or over-flow at runtime and they do this parameterised by a "value" context. One can also envisage more complex examples involving typed stacks and the generation of programs with (backwards) jumps.

Since **Set** is cartesian closed this parameterised Freyd category is closed. A choice for the closure functor is:

$$(A, S_1) \to (B, S_2) = A \Rightarrow (B \times \mathcal{S}(S_1, S_2))$$

where $\Rightarrow$ is the set-theoretic function space. The isomorphism of homsets is directly derived from the cartesian closure of **Set**.

Example 5.2.6 (Composable Continuations) Parameterised Freyd categories provide a way to interpret Danvy and Filinski's composable continuations [DF89]. Composable continuations provide access to evaluation contexts smaller than the whole program, delimited at runtime by the "reset" operator. The current context is made available to the program by the "shift" operator. In contrast, the "call with current continuation" operator described in Example 5.1.7 only allows the entire program to be treated as the current context. The following is inspired by Wadler's attempt to express composable continuations in terms of monads [Wad94].

Given a symmetric monoidal closed category $\mathcal{C}$, define $\mathcal{K}$ to have objects pairs of objects of $\mathcal{C}$ and $\mathcal{K}((A_1, A_2), (B_1, B_2)) = \mathcal{C}(B_1 \multimap B_2, A_1 \multimap A_2)$. Composition in $\mathcal{K}$ is reverse composition in $\mathcal{C}$. Define the functor $J : \mathcal{C} \times |\mathcal{C}| \to \mathcal{K}$ to be $J(f, A) = f \multimap A$, where $|\mathcal{C}|$ is the discrete subcategory of $\mathcal{C}$.

For $f : C \to C'$ and $g : (A_1, A_2) \to (B_1, B_2)$ define

$$f \otimes g = \lambda kx.\text{let } (x_1, x_2) = x \text{ in } g(\lambda b.k(fx_1, b))x_2$$

in the internal language of $\mathcal{C}$.

In terms of the type system given by Danvy and Filinski in [DF89], a judgement $\rho, \alpha \vdash E : \tau, \beta$ is interpreted as an arrow $([\![\rho]\!], [\![\beta]\!]) \to ([\![\tau]\!], [\![\alpha]\!])$. The *reset* operator is interpreted as a function of homsets of $\mathcal{K}$:

$$\begin{aligned} \mathit{reset} &: \mathcal{K}((A, B), (X, X)) \to \mathcal{K}((A, Y), (B, Y)) \\ \mathit{reset} &= f \mapsto \lambda ka.k(f(\lambda x.x)a) \end{aligned}$$

Thus *reset* calls f with the empty context, represented by the identity function, and its input a; feeding the output to the current continuation.

The functor J is value-closed since $\mathcal{C}$ is closed. A choice for the right adjoint is:

$$(A_1, A_2) \to (B_1, B_2) = (B_1 \multimap B_2) \multimap (A_1 \multimap A_2)$$

The isomorphism of homsets is directly definable from the closed structure of $\mathcal{C}$.

With closure we can define the *shift* operator to complement the *reset* operator.

$$\mathit{shift} : \mathcal{K}((E \times (T, D) \to (A, D), B), (X, X)) \to \mathcal{K}((E, B), (T, A))$$

$$shift \quad = \quad f \mapsto \lambda ke.f(\lambda x.x)(e, \lambda k't.k'(kt))$$

See [DF89] and [Wad94] for examples of the use of *shift* and *reset*. This example needs much more work to establish the precise categorical properties of *shift* and *reset*, and to potentially axiomatise it without reference to an underlying continuation passing interpretation, following the lead set by Thielecke [Thi97].

5.2.2 Strong Parameterised Monads

This subsection extends the definition of a strong monad to that of a strong *parameterised* monad, where the parameterisation is over an arbitrary category $\mathcal{S}$. Strong parameterised monads, with the appropriate notion of Kleisli exponential, will be shown to be equivalent to closed parameterised Freyd categories in the next section.

As in the previous subsection, let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be a symmetric monoidal category and let $\mathcal{S}$ be a category.

Definition 5.2.7 An *$\mathcal{S}$-parameterised monad* on $\mathcal{C}$ is a triple (T, η, μ), consisting of a functor $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$; a family of arrows $\eta_{S,A} : A \to T(S, S, A)$, for each $A \in \mathsf{Ob}\mathcal{C}$ and $S \in \mathsf{Ob}\mathcal{S}$, natural in A and dinatural in S; and a family of arrows $\mu_{S_1,S_2,S_3,A} : T(S_1, S_2, T(S_2, S_3, A)) \to T(S_1, S_3, A)$, for each $A \in \mathsf{Ob}\mathcal{C}$ and $S_1, S_2, S_3 \in \mathsf{Ob}\mathcal{S}$, natural in S_1, S_3 and A and dinatural in S_2. These transformations must obey the following commuting diagrams:

$$\begin{array}{ccc}
T(S_1, S_2, T(S_2, S_3, T(S_3, S_4, -))) & \xrightarrow{T(S_1,S_2,\mu_{S_2,S_3,S_4})} & T(S_1, S_2, T(S_2, S_4, -)) \\
\Big\downarrow{\scriptstyle \mu_{S_1,S_2,S_3,T(S_3,S_4,-)}} & & \Big\downarrow{\scriptstyle \mu_{S_1,S_2,S_4}} \\
T(S_1, S_3, T(S_3, S_4, -)) & \xrightarrow{\mu_{S_1,S_3,S_4}} & T(S_1, S_4)
\end{array}$$

$$\begin{array}{ccc}
T(S_1, S_2, -) & \xrightarrow{\eta_{S_1,T(S_1,S_2,-)}} & T(S_1, S_1, T(S_1, S_2, -)) \\
\Big\downarrow{\scriptstyle T(S_1,S_2,\eta_{S_2})} & \searrow^{id} & \Big\downarrow{\scriptstyle \mu_{S_1,S_1,S_2}} \\
T(S_1, S_2, T(S_2, S_2, -)) & \xrightarrow{\mu_{S_1,S_2,S_2}} & T(S_1, S_2, -)
\end{array}$$

In Appendix B we will justify the name "parameterised monad" by relating this definition to adjunctions with parameters.

Analogously to the case for parameterised Freyd structure, when $\mathcal{S}$ is the one object, one arrow category this definition is obviously equivalent to the standard definition of a monad. We now extend the definition of Kleisli category (Definition 5.1.16) to parameterised monads.

Definition 5.2.8 Given an $\mathcal{S}$-parameterised monad (T, η, μ) on $\mathcal{C}$, define the Kleisli category $\mathcal{C}_T$ as:

Objects Pairs of objects of $\mathcal{C}$ and $\mathcal{S}$;

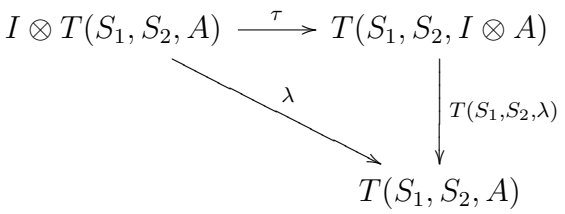
Arrows $\mathcal{C}_T((A, S_1), (B, S_2)) = \mathcal{C}(A, T(S_1, S_2, B))$.

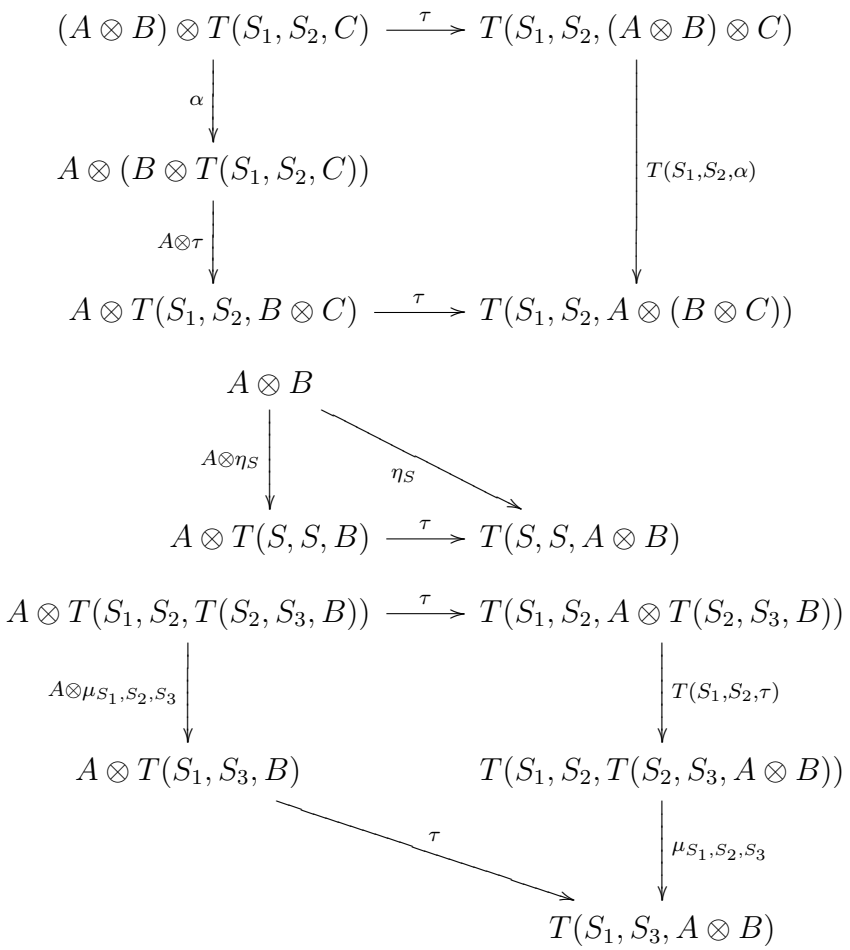
where identities are given by $\eta_{A,S} : A \to T(S, S, A)$ and composition of $f : (A, S_1) \to (B, S_2)$ and $g : (B, S_2) \to (C, S_3)$ is given by $f; T(S_1, S_2, g); \mu_{S_1,S_2,S_3,B}$.

The proof that this definition defines a category is almost identical to the standard one [Mac98] §VI.5; the extra parameterisation plays almost no role. There is also a functor $J_T : \mathcal{C} \times \mathcal{S} \to \mathcal{C}_T$ which is identity on objects and sends arrows $(f : A \to B, s : S_1 \to S_2)$ to $\eta_{S_1,A}; T(S_1, s, f)$. The proof that this defines a functor is by simple calculation.

The notion of strength generalises easily to parameterised monads:

Definition 5.2.9 Given an $\mathcal{S}$-parameterised monad (T, η, μ), a *strength* is a natural transformation $\tau_{A,S_1,S_2,B} : A \otimes T(S_1, S_2, B) \to T(S_1, S_2, A \otimes B)$ that obeys the following commuting diagrams:





As with non-parameterised strong monads we can define premonoidal structure on the Kleisli category using the strength. Given a parameterised strong monad (T, η, μ, τ), define functors (for $f : A \to A'$ in $\mathcal{C}$ and $c : (B, S_1) \to (B', S_2)$ in $\mathcal{C}_T$):

$$f \circledcirc_T c = f \otimes c; \tau_{A',S_1,S_2,B'}$$

$$c \circledcirc_T f = c \otimes f; \sigma_{T(S_1,S_2,B'),A'}; \tau_{A',S_1,S_2,B'}; T(S_1, S_2, \sigma_{A',B'})$$

On objects they are both equal to $\otimes$. The definition is part of the equivalence between parameterised Freyd categories and parameterised strong monads described in Section 5.2.3.

In order to interpret function types we extend the definition of Kleisli exponentials to parameterised strong monads and their Kleisli categories. We use the premonoidal functors just defined.

Definition 5.2.10 A symmetric monoidal category $\mathcal{C}$ with a strong monad (T, η, μ, τ) has *Kleisli exponentials* when, for all objects $(B, S_1) \in \mathsf{Ob}\mathcal{C}_T$, the functor $- \otimes_T (B, S_1) : \mathcal{C} \to \mathcal{C}_T$ has a specified right adjoint.

We will use the notation $(B, S_1) \to -$ for the given right adjoint. As above, we will use the following symbols for the isomorphism of homsets and the counit:

$$\Lambda : \mathcal{C}_T(A \otimes B, S_1), (C, S_2)) \cong \mathcal{C}(A, (B, S_1) \to (C, S_2))$$

$$\mathrm{ev}_{A,S_1,B,S_2} : ((A, S_1) \to (B, S_2) \otimes A, S_1) \to (B, S_2)$$

In the remainder of this section we show how some of the examples of parameterised Freyd categories can also be expressed as strong parameterised monads. Example 5.2.3, pair categories, is not expressible because it is not closed. All of the other examples are instances of the general Theorem 5.2.17, proven in the next subsection, relating parameterised Freyd categories and strong parameterised monads.

Example 5.2.11 (Typed Global State) Example 5.2.4 can be expressed as a strong parameterised monad when the category $\mathcal{C}$ is closed. Let $\mathcal{C}$ be a symmetric monoidal closed category and $\mathcal{S}$ an arbitrary category with a functor $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$. Define $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$ as $T(S_1, S_2, A) = S_1 \multimap (S_2 \otimes A)$, where $\multimap$ is the closure functor of $\mathcal{C}$. Monad unit, multiplication and strength are all defined in the obvious way from the structure of $\mathcal{C}$. A choice for Kleisli exponentials is given by $A \multimap S_1 \multimap (S_2 \otimes B)$. The typed store and retrieve operations given in Example 5.2.4 can also be expressed as arrows in the Kleisli category of this monad.

We also show in Appendix B that this monad also arises as the composite of an adjoint pair of functors between a cartesian closed category $\mathcal{C}$ and a category of typed global state algebras.

Example 5.2.12 (Category Actions) Example 5.2.5 can also be expressed as a parameterised monad. As before, let $\mathcal{C}$ be **Set** and $\mathcal{S}$ be any category with small homsets. Define the functor part of a monad as $T(S_1, S_2, A) = A \times \mathcal{S}(S_1, S_2)$. The unit and multiplication of the monad are defined using the identities and

composition of $\mathcal{S}$ respectively. Strength is defined using associativity. A choice for Kleisli exponentials is given by $(A, S_1) \to (B, S_2) = A \Rightarrow (B \times \mathcal{S}(S_1, S_2))$, using the cartesian closed structure of **Set**.

Example 5.2.13 (Composable Continuations) Example 5.2.6 is also expressible as a parameterised monad, via currying. Given a symmetric monoidal closed category $\mathcal{C}$, define the functor part of the monad $T : |\mathcal{C}|^{\mathsf{op}} \times |\mathcal{C}| \times \mathcal{C} \to \mathcal{C}$ as $T(R_1, R_2, A) = (A \multimap R_1) \multimap R_2$, where $|\mathcal{C}|$ is the discrete subcategory of $\mathcal{C}$ and $\multimap$ is the functor for the closed structure of $\mathcal{C}$. Monad unit, multiplication and strength are defined as (using the internal language of $\mathcal{C}$):

$$\eta_{S,A} = \lambda a.\lambda k.ka \qquad \mu_{S_1,S_2,S_3,A} = \lambda f.\lambda k.f(\lambda k'.k'k)$$

$$\tau_{A,S_1,S_2,B} = \lambda(a, f).\lambda k.f(\lambda b.k(a, b))$$

Note that these are all the same as the definitions for the normal continuations monad 5.1.21, but with more variation in the types. The *shift* and *reset* operations of Example 5.2.6 can also be given as operators on the Kleisli category of this monad. Kleisli exponentials can be given by using the closure $\mathcal{C}$.

5.2.3 Equivalence

The aim of this subsection is to prove that the definitions of strong parameterised monad with Kleisli exponentials and closed Freyd structure on a given $\mathcal{C}$ and $\mathcal{S}$ are equivalent up to isomorphism. We do this by constructing a category of each and proving that the categories are equivalent. As a special case when $\mathcal{S} = 1$ this will prove that closed Freyd structure and strong monads with Kleisli exponentials are also equivalent up to isomorphism.

As in the previous two subsections, let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be a symmetric monoidal category and let $\mathcal{S}$ be a category.

Definition 5.2.14 The category $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$ is defined as:

Objects Closed parameterised Freyd structure on $\mathcal{C}$, $\mathcal{S}$ with respect to $\mathcal{C}$;

Arrows An arrow $f : (\mathcal{K}_1, J_1, \circledcirc_1, \circledcirc_1, \rightarrow_1, \Lambda_1) \rightarrow (\mathcal{K}_2, J_2, \circledcirc_2, \circledcirc_2, \rightarrow_2, \Lambda_2)$ is a functor $f : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ that commutes with the structure:

$$J_1; f = J_2 \qquad Id \times f; \circledcirc_2 = \circledcirc_1; f \qquad f \times Id; \circledcirc_2 = \circledcirc_2; f$$

Identities are identity functors and composition is functor composition.

Note that this definition implies that all the functors underlying arrows in $\mathbf{CSF}(\mathcal{C}, \mathcal{S})$ are identity on objects.

Definition 5.2.15 The category $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$ is defined as:

Objects Strong $\mathcal{S}$-parameterised monads on $\mathcal{C}$ with Kleisli exponentials;

Arrows An arrow $f : (T_1, \eta_1, \mu_1, \tau_1, \rightarrow_1, \Lambda_1) \rightarrow (T_2, \eta_2, \mu_2, \tau_2, \rightarrow_2, \Lambda_2)$ is a natural transformation $f : T_1 \Rightarrow T_2 : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \rightarrow \mathcal{C}$ which must commute with all the structure of the monad:

$$\begin{array}{ccc} A & \xrightarrow{\eta_{1,S,A}} & T_1(S,S,A) \\ & \searrow^{\eta_{2,S,A}} & \downarrow{\scriptstyle f_{S,S,A}} \\ & & T_2(S,S,A) \end{array}$$

$$\begin{array}{ccc} A \otimes T_1(S_1,S_2,B) & \xrightarrow{\tau_{1,A,S_1,S_2,B}} & T(S_1,S_2,A\otimes B) \\ \downarrow{\scriptstyle A\otimes f_{S_1,S_2,B}} & & \downarrow{\scriptstyle f_{S_1,S_2,A\otimes B}} \\ A \otimes T_2(S_1,S_2,B) & \xrightarrow{\tau_{2,A,S_1,S_2,B}} & T_2(S_1,S_2,A\otimes B) \end{array}$$

$$\begin{array}{ccc} T_1(S_1,S_2,T_1(S_2,S_3,A)) & \xrightarrow{\mu_{1,S_1,S_2,S_3,A}} & T_1(S_1,S_3,A) \\ \downarrow{\scriptstyle f_{S_1,S_2,T_1(S_2,S_3,A)};T_2(S_1,S_2,f_{S_2,S_3,A})} & & \downarrow{\scriptstyle f_{S_1,S_3,A}} \\ T_2(S_1,S_2,T_2(S_2,S_3,A)) & \xrightarrow{\mu_{2,S_1,S_2,S_3,A}} & T_2(S_1,S_3,A) \end{array}$$

Identities are identity natural transformations and composition is by composition of natural transformations.

We must show that these two definitions actually define categories. In particular, that the given identities are really arrows of the category, and that the given composition is similarly well-defined.

Proposition 5.2.16 Definitions 5.2.14 and 5.2.15 define categories.

Proof Identity functors trivially satisfy the requirements of arrows of $\mathbf{CPF}(\mathcal{C},\mathcal{S})$; the three equations are trivially satisfied. Composed arrows are also well-defined:

$$J_1; f; g = J_2; g = J_3 \qquad Id \times (f;g); \circledcirc_3 = Id \times f; \circledcirc_2; g = \circledcirc_1; f; g$$
$$(f;g) \times Id; \circledcirc_3 = f \times Id; \circledcirc_2; g = \circledcirc_2; f; g$$

where each sequence of equations follows from the corresponding equations for the arrows f and g. For $\mathbf{CSPM}(\mathcal{C},\mathcal{S})$, it is immediate that identity natural transformations are the identity arrows. The composition of two arrows also obeys the three diagrams:

$$\eta_1; f; g = \eta_2; g = \eta_3 \qquad \tau_1; f; g = A \otimes f; \tau_2; g = A \otimes f; A \otimes g; \tau_3$$
$$\mu_1; f; g = T_1 f; f; \mu_2; g = T_1 f; f; T_2 g; g; \mu_3 = T_1(f;g); f; g; \mu_S$$

where each sequence of equations follows from the corresponding equations for the arrows f and g and, in the case of μ, by naturality of these arrows. In both cases, the fact that composition respects identities and is associative follows from standard facts about functors and natural transformations. □

We now show that our two categories are equivalent and, as a special case, that the non-parameterised definitions of Section 5.1 are equivalent. The bulk of the proof, which is comprised of tedious checking of the requirements of the definitions, is relegated to the appendix, in Section A.1.

Theorem 5.2.17 The categories $\mathbf{CPF}(\mathcal{C},\mathcal{S})$ and $\mathbf{CSPM}(\mathcal{C},\mathcal{S})$ are equivalent.

Proof We define a functor $F : \mathbf{CSPM}(\mathcal{C},\mathcal{S}) \to \mathbf{CPF}(\mathcal{C},\mathcal{S})$ and show that it is an equivalence. The functor F is defined as:

$$\begin{aligned}(T,\eta,\mu,\tau,- \to -,\Lambda)) &\mapsto (\mathcal{C}_T, J_T, \circledcirc_T, \circledcirc_T, - \to -, \Lambda)\\ f : T_1 \Rightarrow T_2 &\mapsto (g : (A,S_1) \to (B,S_2)) \mapsto g; f_{S_1,S_2,B}\end{aligned}$$

where the functor Ff is necessarily always identity on objects. See Section A.1.1 for the proof that this definition is well-defined.

This functor is full and faithful. For every arrow $f : F(T_1) \to F(T_2)$ of $\mathbf{CPF}(\mathcal{C},\mathcal{S})$, define $F^{-1}f : T_1 \Rightarrow T_2$ as $f(id_{T_1(S_1,S_2,A)})$ (treating f as a function

$\mathcal{C}(A, T_1(S_1, S_2, B)) \to \mathcal{C}(A, T_2(S_1, S_2, B)))$. See Section A.1.2 for a proof that this is indeed an inverse operation on arrows.

This functor is also essentially surjective. Given an object $X = (\mathcal{K}, J, \circledcirc, \circledcirc, - \to -, \Lambda)$ in $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$, define an object $Y = (T^X, \eta^X, \mu^X, \tau^X, - \to^X -, \Lambda^X)$ as:

$$\begin{aligned}
T^X(S_1, S_2, A) &= (I, S_1) \to (A, S_2) \\
\eta^X_{S,A} &= \Lambda(\rho_A, S) \\
\mu^X_{S_1,S_2,S_3,A} &= \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev}) \\
\tau^X_{A,S_1,S_2,B} &= \Lambda(J(\alpha, S_1); A \circledcirc \mathrm{ev}) \\
(A, S_1) \to^X (B, S_2) &= (A, S_1) \to (B, S_2) \\
(A, S_1) \to^X f &= (A, S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(f)) \\
\Lambda^X(f) &= \Lambda(J(\rho^{-1}_{A \otimes B}, S_1); \Lambda^{-1}(f))
\end{aligned}$$

where $\mathrm{ev} = \Lambda^{-1}(id)$. See Section A.1.3 for a proof that this is an object of $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$ and that $FY \cong X$.

Given that F is full and faithful and essentially surjective, it follows that F is an equivalence, as required. □

5.3 Monoidal Typed Computational Effects

In this section we deal with extending the definitions in the previous section with extra structure for embedding typed computations in larger state types. We prove that these two definitions are equivalent in Section 5.3.3.

5.3.1 Double Parameterised Freyd categories

We extend the definition of parameterised Freyd category (Definition 5.2.1) to allow computation in an state context by simply requiring premonoidal structure with respect to $\mathcal{S}$ as well as with respect to $\mathcal{C}$. The new definition is completely symmetric in terms of $\mathcal{C}$ and $\mathcal{S}$ and it is only the definition of closure that distinguishes between them. We call the new definition Double Parameterised Freyd structure.

With two premonoidal structures we can define a single premonoidal structure with respect to $\mathcal{C} \times \mathcal{S}$. Using this, we show that double parameterised Freyd categories are equivalent to certain Power, Robinson and Thielecke Freyd categories, as defined in Section 5.1.1.1.

Let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ and $(\mathcal{S}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be symmetric monoidal categories. The two monoidal structures are distinct but we use the same notation for both of them to maintain the connection with the definition of monoidal structure. We hope that this does not cause too much confusion.

Definition 5.3.1 *Double parameterised Freyd structure* on $\mathcal{C}$, $\mathcal{S}$ consists of a category $\mathcal{K}$ and five functors:

$$J : \mathcal{C} \times \mathcal{S} \to \mathcal{K} \qquad \circledcirc_{\mathcal{C}} : \mathcal{C} \times \mathcal{K} \to \mathcal{K} \qquad \circledcirc_{\mathcal{C}} : \mathcal{K} \times \mathcal{C} \to \mathcal{K} \qquad \circledcirc_{\mathcal{S}} : \mathcal{S} \times \mathcal{K} \to \mathcal{K}$$
$$\circledcirc_{\mathcal{S}} : \mathcal{K} \times \mathcal{S} \to \mathcal{K}$$

such that J is identity-on-objects and $(J, \circledcirc_{\mathcal{C}}, \circledcirc_{\mathcal{C}})$ and $(J, \circledcirc_{\mathcal{S}}, \circledcirc_{\mathcal{S}})$ obey the obvious adaptations of the conditions of Definition 5.2.1.

As before, we shall refer to a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{K}, J, \circledcirc_{\mathcal{C}}, \circledcirc_{\mathcal{C}}, \circledcirc_{\mathcal{S}}, \circledcirc_{\mathcal{S}})$, where $\mathcal{C}$ and $\mathcal{S}$ are symmetric monoidal categories, as a double parameterised Freyd category. We will often just abbreviate this to the functor part $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ and use the notation in the definition for the premonoidal structure.

Example 5.3.2 Extending Example 5.2.3, Pair Categories, if $\mathcal{S}$ is symmetric monoidal, then the identity functor $Id : \mathcal{C} \times \mathcal{S} \to \mathcal{C} \times \mathcal{S}$ can be given parameterised symmetric premonoidal structure with respect to $\mathcal{S}$ in the obvious way.

Example 5.3.3 Example 5.2.4, Typed Global State, can be extended to have parameterised symmetric premonoidal structure with respect to $\mathcal{S}$ when $\mathcal{S}$ has symmetric monoidal structure and the functor $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$ is strict symmetric monoidal. Define:

$$s \circledcirc_{\mathcal{S}} c = A \otimes (\widehat{S_1} \otimes \widehat{S_2}) \cong \widehat{S_1} \otimes (A \otimes \widehat{S_2}) \xrightarrow{s \otimes c} \widehat{S_1'} \otimes (B \otimes \widehat{S_2'}) \cong B \otimes (\widehat{S_1'} \otimes \widehat{S_2'})$$

$$c \circledcirc_{\mathcal{S}} s = A \otimes (\widehat{S_1} \otimes \widehat{S_2}) \cong (A \otimes \widehat{S_1}) \otimes \widehat{S_2} \xrightarrow{c \otimes s} (B \otimes \widehat{S_1'}) \otimes \widehat{S_2'} \cong B \otimes (\widehat{S_1'} \otimes \widehat{S_2'})$$

These operations allow us to lift stateful computations up to larger states; we can take a computation that operates locally, taking a state S_1 to a state S_2, and embed it into larger start and finish states $S \otimes S_1$ and $S \otimes S_2$. This is the basis of the frame rule and localised reasoning in Separation Logic [Rey02]. In Chapter 7 we give a more sophisticated example of a model of type-localised state using functor categories that captures the non-changing size of the heap as well.

Example 5.3.4 The Category Actions example, Example 5.2.5, can be given parameterised symmetric premonoidal structure with respect to $\mathcal{S}$ when $\mathcal{S}$ is symmetric monoidal. Define:

$$s \ltimes_{\mathcal{S}} c = A \xrightarrow{\langle !;\widehat{s},c\rangle} \mathcal{S}(S_1', S_2') \times \mathcal{S}(S_1, S_2) \times B \xrightarrow{\widehat{\otimes}\times B} \mathcal{S}(S_1' \otimes S_1, S_2' \otimes S_2) \times B$$

$$c \rtimes_{\mathcal{S}} s = A \xrightarrow{\langle c,!;\widehat{s}\rangle} \mathcal{S}(S_1, S_2) \times B \times \mathcal{S}(S_1', S_2') \xrightarrow{\cong;\widehat{\otimes}\times B} \mathcal{S}(S_1 \otimes S_1', S_2 \otimes S_2') \times B$$

Where $\widehat{s}$ is the arrow $1 \to \mathcal{S}(S_1', S_2')$ picking out the arrow s, and $\widehat{\otimes}$ is the action of the monoidal structure of $\mathcal{S}$ on arrows.

We now show that double parameterised Freyd categories are equivalent to certain Power, Robinson and Thielecke's Freyd categories (Section 5.1.1.1). We will do this by defining premonoidal structure on $\mathcal{K}$ using the two sets of premonoidal functors. Firstly, the two premonoidal structures on J commute:

Lemma 5.3.5 Given a double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, then the following equations hold for all $s : S_1 \to S_1$, $f : A_1 \to A_2$ and $c : (B_1, S_1') \to (B_2, S_2')$.

$$s \ltimes_{\mathcal{S}} (f \ltimes_{\mathcal{C}} c) = f \ltimes_{\mathcal{C}} (s \ltimes_{\mathcal{S}} c) \qquad (c \rtimes_{\mathcal{C}} f) \rtimes_{\mathcal{S}} s = (c \rtimes_{\mathcal{S}} s) \rtimes_{\mathcal{C}} f$$

$$s \ltimes_{\mathcal{S}} (c \rtimes_{\mathcal{C}} f) = (s \ltimes_{\mathcal{S}} c) \rtimes_{\mathcal{C}} f \qquad f \ltimes_{\mathcal{C}} (c \rtimes_{\mathcal{S}} s) = (f \ltimes_{\mathcal{C}} c) \rtimes_{\mathcal{S}} s$$

Proof The first equation: $s \ltimes_{\mathcal{S}} (f \ltimes_{\mathcal{C}} c) = (f \otimes id, s \otimes id); id \ltimes_{\mathcal{S}} (id \ltimes_{\mathcal{C}} c) = f \ltimes (s \ltimes c)$. The second equation is similar.

The third equation, by the second equation and naturality:

$$\begin{aligned} & s \ltimes_{\mathcal{S}} (c \rtimes_{\mathcal{C}} f) \\ = \; & s \ltimes_{\mathcal{S}} (c \rtimes_{\mathcal{C}} f); (B_2 \otimes A_2, \sigma; \sigma^{-1}) \end{aligned}$$

$$\begin{aligned} &= (B_1 \otimes A_1, \sigma); (c \mathbin{\ominus_{\mathcal{C}}} f) \mathbin{\ominus_{\mathcal{S}}} s; (B_2 \otimes A_2, \sigma^{-1}) \\ &= (B_1 \otimes A_1, \sigma); (c \mathbin{\ominus_{\mathcal{S}}} s) \mathbin{\ominus_{\mathcal{C}}} f; (B_2, \sigma^{-1}) \mathbin{\ominus} A_2 \\ &= (B_1 \otimes A_1, \sigma; \sigma^{-1}); (s \mathbin{\oslash_{\mathcal{S}}} c) \mathbin{\ominus_{\mathcal{C}}} f \\ &= (s \mathbin{\oslash_{\mathcal{S}}} c) \mathbin{\ominus_{\mathcal{C}}} f \end{aligned}$$

The fourth equation is similar. □

With this we can unambiguously define parameterised symmetric premonoidal structure with respect to $\mathcal{C} \times \mathcal{S}$:

$$c \mathbin{\ominus} (f, s) = (c \mathbin{\oslash_{\mathcal{C}}} f) \mathbin{\oslash_{\mathcal{S}}} s \qquad\qquad (f, s) \mathbin{\ominus} c = f \mathbin{\ominus_{\mathcal{C}}} (s \mathbin{\ominus_{\mathcal{S}}} c)$$

It is easy to see that the structure transformations generated by the pairs of structure natural transformations from $\mathcal{C}$ and $\mathcal{S}$ satisfy the required naturality conditions.

Next, all arrows given by J are central in the sense of Definition 5.1.2. This will allow us to establish that J is a strict symmetric premonoidal functor, and hence part of the structure of a Power, Robinson, Thielecke Freyd category.

Lemma 5.3.6 Given a double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, all arrows of the form $J(f, s)$ are central. That is, for all $c : (B_1, S_1') \to (B_2, S_2')$:

$$(A_1, S_1) \mathbin{\ominus} c; (f, s) \mathbin{\ominus} (B_2, S_2') = (f, s) \mathbin{\ominus} (B_1, S_1'); (A_2, S_2) \mathbin{\ominus} c$$

Proof $(A_1, S_1) \mathbin{\ominus} c; (f, s) \mathbin{\ominus} (B_2, S_2') = (f, s) \mathbin{\oslash} c = (f, s) \mathbin{\ominus} (B_1, S_1'); (A_2, S_2) \mathbin{\ominus} c$ □

The next proposition shows that double parameterised Freyd categories are equivalent to certain Power, Robinson, Thielecke Freyd categories. As a special case, when $\mathcal{S} = 1$, this proves Theorem 5.1.14 that our definition of Freyd categories and the Power, Robinson, Thielecke definition are also equivalent.

Theorem 5.3.7 Given a double Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ we can define symmetric premonoidal structure of $\mathcal{K}$, in the sense of Definition 5.1.10, such that J is a strict premonoidal functor.

Conversely, given a symmetric premonoidal category $\mathcal{K}$, in the sense of 5.1.10, and a functor $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ which is strict symmetric premonoidal, we can define

functors $\ominus_{\mathcal{C}}, \oslash_{\mathcal{C}}, \ominus_{\mathcal{S}}, \oslash_{\mathcal{S}}$ such that $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is a double parameterised Freyd category.

These two operations of definition are mutually inverse.

Proof Take the pointwise symmetric monoidal structure on $\mathcal{C} \times \mathcal{S}$ arising from the structure on the two categories.

Binoidal structure on $\mathcal{K}$ is given by $(A, S) \ominus' f = A \ominus (S \ominus f)$ and $f \oslash' (A, S) = (f \oslash A) \oslash S$. The natural isomorphisms for symmetric premonoidal structure on $\mathcal{K}$ are given by the pairing of the morphisms of the symmetric monoidal structure on $\mathcal{C}$ and $\mathcal{S}$, via J. Their naturality follows from the required naturality of the parameterised premonoidal structure and the commutativity properties of Lemma 5.3.5. By Lemma 5.3.6, every arrow of the form (f, s) in $\mathcal{K}$ is central. Hence J preserves centrality and, by construction of the premonoidal structure in $\mathcal{K}$, strictly preserves premonoidal structure.

For the converse, define $f \ominus_{\mathcal{C}} c$ as the composite:

$$\begin{array}{lcl} (A \times B, S) & \xrightarrow{J(id_{A\times B}, \lambda)} & (A \times B, I \otimes S) \\ & \xrightarrow{(A,I) \ominus c} & (A \times B', I \otimes S') \\ & \xrightarrow{J(f, id_I) \oslash (B', S')} & (A' \times B', I \otimes S') \\ & \xrightarrow{(id_{A'\times B'}, \lambda^{-1})} & (A' \times B', S') \end{array}$$

The functor $\oslash_{\mathcal{C}}$ with respect to $\mathcal{C}$ is defined similarly. See the appendix, Section A.2, for the proof that these definitions obey the requirements of a double parameterised Freyd category. Defining and verifying the symmetric premonoidal structure $(\ominus_{\mathcal{S}}, \oslash_{\mathcal{S}})$ with respect to $\mathcal{S}$ is almost identical.

That the two definitions are inverse can be easily seen by writing out the definitions and comparing them. □

5.3.2 Monoidal Parameterised Monads

Extending parameterised monads to allow typed computations to be embedded in larger state contexts is achieved by requiring two extra natural transformations on the monad, adjoining extra state context to the left and right of the computation

respectively. These will correspond directly to the premonoidal structure with respect to $\mathcal{S}$ as defined for double parameterised Freyd categories in the previous section.

Definition 5.3.8 An $\mathcal{S}$-parameterised monad (T, η, μ) has *monoidal multiplication* if there are transformations:

$$\mu_{\otimes S, S_1, S_2, A} : T(S_1, S_2, A) \to T(S_1 \otimes S, S_2 \otimes S, A)$$

$$\mu_{S \otimes, S_1, S_2, A} : T(S_1, S_2, A) \to T(S \otimes S_1, S \otimes S_2, A)$$

such that they are both dinatural in S, natural in all other variables and they obey the following commutative diagrams:

$$\begin{array}{ccc}
T(S_1, S_2, T(S_2, S_3, A)) & \xrightarrow{\mu_{\otimes S}} & T(S_1 \otimes S, S_2 \otimes S, T(S_2, S_3, A)) \\
\downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle T(S_1 \otimes S, S_2 \otimes S, \mu_{\otimes S})} \\
 & & T(S_1 \otimes S, S_2 \otimes S, T(S_2 \otimes S, S_3 \otimes S, A)) \\
 & & \downarrow{\scriptstyle \mu} \\
T(S_1, S_3, A) & \xrightarrow{\mu_{\otimes S}} & T(S_1 \otimes S, S_3 \otimes S, A)
\end{array}$$

$$\begin{array}{ccc}
T(S_1, S_2, T(S_2, S_3, A)) & \xrightarrow{\mu_{S \otimes}} & T(S \otimes S_1, S \otimes S_2, T(S_2, S_3, A)) \\
\downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle T(S \otimes S_1, S \otimes S_2, \mu_{S \otimes})} \\
 & & T(S \otimes S_1, S \otimes S_2, T(S \otimes S_2, S \otimes S_3, A)) \\
 & & \downarrow{\scriptstyle \mu} \\
T(S_1, S_3, A) & \xrightarrow{\mu_{S \otimes}} & T(S \otimes S_1, S \otimes S_3, A)
\end{array}$$

$$\begin{array}{ccc}
A & \xrightarrow{\eta_{S,A}} & T(S, S, A) \\
 & {\scriptstyle \eta_{S \otimes S', A}} \searrow & \downarrow{\scriptstyle \mu_{\otimes S'}} \\
 & & T(S \otimes S', S \otimes S', A)
\end{array}
\qquad
\begin{array}{ccc}
A & \xrightarrow{\eta_{S',A}} & T(S', S', A) \\
 & {\scriptstyle \eta_{S \otimes S', A}} \searrow & \downarrow{\scriptstyle \mu_{S \otimes}} \\
 & & T(S \otimes S', S \otimes S', A)
\end{array}$$

Two diagrams for symmetry:

$$\begin{array}{ccc} T(S_1, S_2, A) & \xrightarrow{\mu_{\otimes S}} & T(S_1 \otimes S, S_2 \otimes S, A) \\ \downarrow{\scriptstyle \mu_{S\otimes}} & & \downarrow{\scriptstyle T(\sigma, S_2 \otimes S, A)} \\ T(S \otimes S_1, S \otimes S_2, A) & \xrightarrow{T(S\otimes S_1, \sigma, A)} & T(S \otimes S_1, S_2 \otimes S, A) \end{array}$$

$$\begin{array}{ccc} T(S_1, S_2, A) & \xrightarrow{\mu_{S\otimes}} & T(S \otimes S_1, S \otimes S_2, A) \\ \downarrow{\scriptstyle \mu_{\otimes S}} & & \downarrow{\scriptstyle T(\sigma, S \otimes S_2, A)} \\ T(S_1 \otimes S, S_2 \otimes S, A) & \xrightarrow{T(S_1\otimes S, \sigma, A)} & T(S_1 \otimes S, S \otimes S_2, A) \end{array}$$

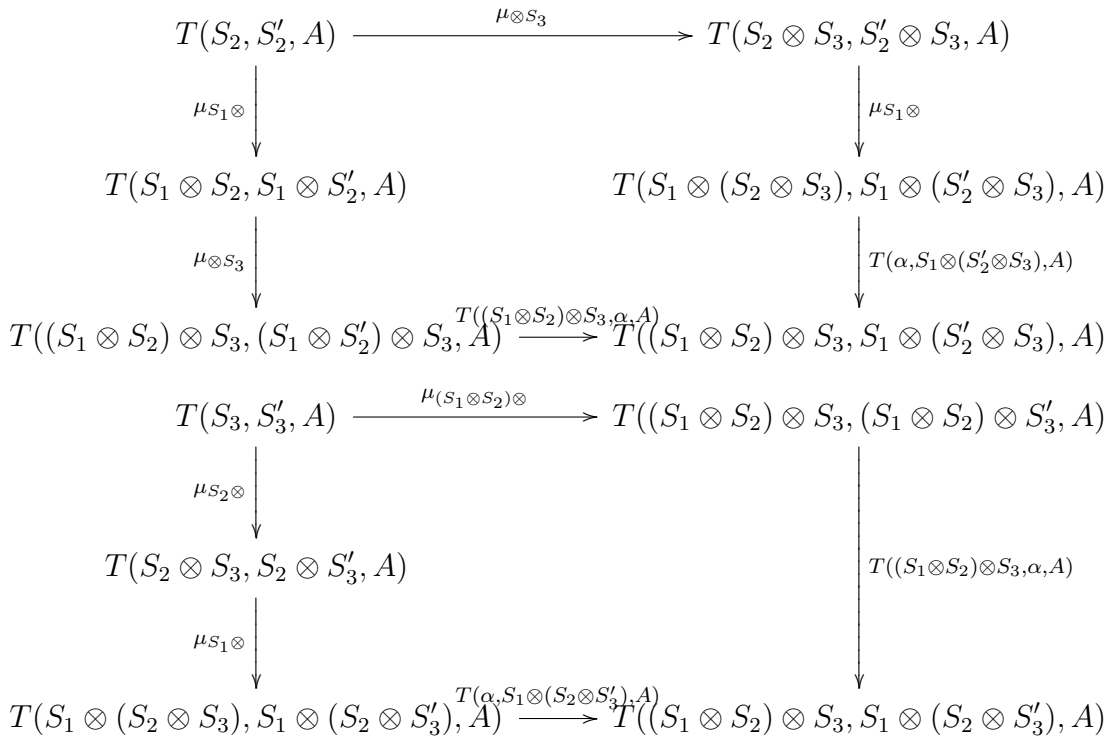
A diagram each for left and right units:

$$\begin{array}{ccc} T(S_1, S_2, A) & \xrightarrow{\mu_{I\otimes}} & T(I \otimes S_1, I \otimes S_2, A) \\ & {\scriptstyle T(\lambda, S_2, A)} \searrow & \downarrow{\scriptstyle T(I\otimes S_1, \lambda, A)} \\ & & T(I \otimes S_1, S_2, A) \end{array}$$

$$\begin{array}{ccc} T(S_1, S_2, A) & \xrightarrow{\mu_{\otimes I}} & T(S_1 \otimes I, S_2 \otimes I, A) \\ & {\scriptstyle T(\rho, S_2, A)} \searrow & \downarrow{\scriptstyle T(S_1\otimes I, \rho, A)} \\ & & T(S_1 \otimes I, S_2, A) \end{array}$$

Three diagrams for associativity:

$$\begin{array}{ccc} T(S_1, S_1', A) & \xrightarrow{\mu_{\otimes(S_2\otimes S_3)}} & T(S_1 \otimes (S_2 \otimes S_3), S_1' \otimes (S_2 \otimes S_3), A) \\ \downarrow{\scriptstyle \mu_{\otimes S_2}} & & \\ T(S_1 \otimes S_2, S_1' \otimes S_2, A) & & \downarrow{\scriptstyle T(\alpha, S_1' \otimes (S_2\otimes S_3), A)} \\ \downarrow{\scriptstyle \mu_{\otimes S_3}} & & \\ T((S_1 \otimes S_2) \otimes S_3, (S_1' \otimes S_2) \otimes S_3, A) & \xrightarrow{T((S_1\otimes S_2)\otimes S_3, \alpha, A)} & T((S_1 \otimes S_2) \otimes S_3, S_1' \otimes (S_2 \otimes S_3), A) \end{array}$$



Note that we do not have to necessarily require the two natural transformations. By the diagrams for symmetry, each can be expressed in terms of the other.

Before we show the equivalence of this structure and double parameterised Freyd structure we show how some of the example double parameterised Freyd categories are expressible as monoidal parameterised monads. Note that, again, the pair categories example (Example 5.3.2) is not expressible as a parameterised monad because it is not closed.

Example 5.3.9 The typed local state example, Example 5.3.3, is expressible in terms of monoidal parameterised monads. Take the strong monad defined in Example 5.2.11 for typed global state and, assuming that $\mathcal{S}$ is symmetric monoidal and the functor $\widehat{\cdot} \colon \mathcal{S} \to \mathcal{S}$ is strict symmetric monoidal, define:

$$\mu_{S\otimes} = \Lambda(S_1 \multimap (A \otimes S_2) \otimes \sigma; \alpha^{-1}; \mathrm{ev} \otimes S; \alpha; A \otimes \sigma) \qquad \mu_{\otimes S} = \Lambda(\alpha^{-1}; \mathrm{ev} \otimes S; \alpha)$$

Several pages of tedious calculation show that these definitions obey the axioms above.

Example 5.3.10 Example 5.3.4, monoidal category actions, is also expressible as a monoidal parameterised monad. Take the definition of the strong monad from Example 5.2.12 and define:

$$\mu_{S\otimes} = A \times \mathcal{S}(S_1, S_2) \xrightarrow{A\times\widehat{S\otimes}} A \times \mathcal{S}(S \otimes S_1, S \otimes S_2)$$

$$\mu_{\otimes S} = A \times \mathcal{S}(S_1, S_2) \xrightarrow{A\times\widehat{\otimes S}} A \times \mathcal{S}(S_1 \otimes S, S_2 \otimes S)$$

where $\widehat{S\otimes}$ and $\widehat{\otimes S}$ are the operations derived from the monoidal structure of $\mathcal{S}$ using the identity arrow $S \to S$.

5.3.3 Equivalence

We now extend the equivalence result of Section 5.2.3 to an equivalence between closed double parameterised Freyd categories and strong monoidal parameterised monads with Kleisli exponentials. As before, we define two categories of such structures over a pair of symmetric monoidal categories $\mathcal{C}$ and $\mathcal{S}$ and show that the two categories are equivalent. Much of the proof will be the same as for Theorem 5.2.17; we only need to verify that the extra structure is preserved by the equivalence.

As in the previous two sections, let $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ and $(\mathcal{S}, \otimes, I, \alpha, \lambda, \rho, \sigma)$ be symmetric monoidal categories. The definitions of the two categories are based on Definitions 5.2.14 and 5.2.15, extended with premonoidal structure with respect to $\mathcal{S}$ and monoidal multiplication respectively. The arrows of categories must preserve this new structure, as well as the old, in both cases.

Definition 5.3.11 The category $\mathbf{CDPF}(\mathcal{C}, \mathcal{S})$ is defined as:

Objects Closed Double Parameterised Freyd structure on $(\mathcal{C}, \mathcal{S})$;

Arrows $f : (\mathcal{K}_1, J_1, \circledcirc_{\mathcal{C},1}, \oslash_{\mathcal{C},1}, \circledcirc_{\mathcal{S},1}, \oslash_{\mathcal{S},1}, \to_1, \Lambda_1) \to (\mathcal{K}_2, J_2, \circledcirc_{\mathcal{C},2}, \oslash_{\mathcal{C},2}, \circledcirc_{\mathcal{S},2}, \oslash_{\mathcal{S},2}, \to_1, \Lambda_2)$ is a functor $f : \mathcal{K}_1 \to \mathcal{K}_2$ that commutes with the structure:

$$J_1; f = J_2 \qquad Id \times f; \circledcirc_{\mathcal{C},2} = \circledcirc_{\mathcal{C},1}; f \qquad Id \times f; \oslash_{\mathcal{C},2} = \oslash_{\mathcal{C},1}; f$$

$$Id \times f; \circledcirc_{\mathcal{S},2} = \circledcirc_{\mathcal{S},1}; f \qquad Id \times f; \oslash_{\mathcal{S},2} = \oslash_{\mathcal{S},1}; f$$

Identities are identity functors and composition is by functor composition.

Definition 5.3.12 The category $\mathbf{CSMPM}(\mathcal{C}, \mathcal{S})$ is defined as:

Objects Strong $\mathcal{S}$-parameterised monoidal monads on $\mathcal{C}$ with Kleisli exponentials;

Arrows An arrow f:

$$\begin{array}{rl} & (T_1, \eta_1, \mu_1, \tau_1, \mu_{\otimes S,1}, \mu_{S\otimes,1}, - \to_1 -, \Lambda_1) \\ \to & (T_2, \eta_2, \mu_2, \tau_2, \mu_{\otimes S,2}, \mu_{\otimes S,2}, - \to_2 -, \Lambda_2) \end{array}$$

is a natural transformation $f : T_1 \Rightarrow T_2 : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$ which commutes with all the structure of the monad:

$$\begin{array}{ccc} A & \xrightarrow{\eta_{1,S,A}} & T_1(S,S,A) \\ & \searrow^{\eta_{2,S,A}} & \downarrow{\scriptstyle f_{S,S,A}} \\ & & T_2(S,S,A) \end{array} \qquad \begin{array}{ccc} A \otimes T_1(S_1,S_2,B) & \xrightarrow{\tau_{1,A,S_1,S_2,B}} & T(S_1,S_2,A\otimes B) \\ \downarrow{\scriptstyle A\otimes f_{S_1,S_2,B}} & & \downarrow{\scriptstyle f_{S_1,S_2,A\otimes B}} \\ A \otimes T_2(S_1,S_2,B) & \xrightarrow{\tau_{2,A,S_1,S_2,B}} & T_2(S_1,S_2,A\otimes B) \end{array}$$

$$\begin{array}{ccc} T_1(S_1,S_2,T_1(S_2,S_3,A)) & \xrightarrow{\mu_{1,S_1,S_2,S_3,A}} & T_1(S_1,S_3,A) \\ \downarrow{\scriptstyle f_{S_1,S_2,T_1(S_2,S_3,A)};T_2(S_1,S_2,f_{S_2,S_3,A})} & & \downarrow{\scriptstyle f_{S_1,S_3,A}} \\ T_2(S_1,S_2,T_2(S_2,S_3,A)) & \xrightarrow{\mu_{2,S_1,S_2,S_3,A}} & T_2(S_1,S_3,A) \end{array}$$

$$\begin{array}{ccc} T_1(S_1,S_2,A) & \xrightarrow{\mu_{\otimes S,1}} & T_1(S_1\otimes S, S_2\otimes S, A) \\ \downarrow{\scriptstyle f_{S_1,S_2,A}} & & \downarrow{\scriptstyle f_{S_1\otimes S,S_2\otimes S,A}} \\ T_2(S_1,S_2,A) & \xrightarrow{\mu_{\otimes S,2}} & T_2(S_1\otimes S, S_2\otimes S, A) \end{array}$$

$$\begin{array}{ccc} T_1(S_1,S_2,A) & \xrightarrow{\mu_{S\otimes,1}} & T_1(S\otimes S_1, S\otimes S_2, A) \\ \downarrow{\scriptstyle f_{S_1,S_2,A}} & & \downarrow{\scriptstyle f_{S\otimes S_1,S\otimes S_1,A}} \\ T_2(S_1,S_2,A) & \xrightarrow{\mu_{S\otimes,2}} & T_2(S\otimes S_1, S\otimes S_2, A) \end{array}$$

Identities are identity natural transformations and composition is by natural transformation composition.

Proposition 5.3.13 Definitions 5.3.11 and 5.3.12 define categories.

Proof Essentially the same as the proof of Proposition 5.2.16. □

Theorem 5.3.14 The categories $\mathbf{CDPF}(\mathcal{C},\mathcal{S})$ and $\mathbf{CSMPM}(\mathcal{C},\mathcal{S})$ are equivalent.

Proof We use the same structure as the proof of Theorem 5.2.17, defining a functor $F : \mathbf{CDPF}(\mathcal{C},\mathcal{S}) \to \mathbf{CSMPM}(\mathcal{C},\mathcal{S})$ and showing that it is full and faithful and essentially surjective. Define F as:

$$\begin{array}{rcl}(T,\eta,\mu,\tau,\mu_{\otimes S},\mu_{S\otimes},-\!\!\rightarrow\!-,\Lambda) & \mapsto & (\mathcal{C}_T, J_T, \mathbin{\circledcirc}^T_{\mathcal{C}}, \mathbin{\circledcirc}^T_{\mathcal{C}}, \mathbin{\circledcirc}^T_{\mathcal{S}}, \mathbin{\circledcirc}^T_{\mathcal{S}}, -\!\!\rightarrow\!-,\Lambda)\\ f : T_1 \Rightarrow T_2 & \mapsto & (g : (A,S_1) \to (B,S_2)) \mapsto g; f_{S_1,S_2,B}\end{array}$$

where:

$$f \mathbin{\circledcirc}^T_{\mathcal{C}} c = f \otimes c; \tau \qquad\qquad c \mathbin{\circledcirc}^T_{\mathcal{C}} f = c \otimes f; \sigma; \tau; T(S_1,S_2,\sigma)$$

$$s \mathbin{\circledcirc}^T_{\mathcal{S}} c = c; \mu_{S_1\otimes}; T(S_1 \otimes S_2, s \otimes S_2', B) \qquad c \mathbin{\circledcirc}^T_{\mathcal{S}} s = c; \mu_{\otimes S_2}; T(S_1 \otimes S_2, S_1' \otimes c, B)$$

See the appendix, Section A.3.1, for the proof that this definition actually gives a functor. The proof builds on the first part of the proof of Theorem 5.2.17 (Section A.1.1). Moreover, F is full and faithful: see Section A.3.2.

This functor is also essentially surjective. Given an object

$$X = (\mathcal{K}, J, \mathbin{\circledcirc}_{\mathcal{C}}, \mathbin{\circledcirc}_{\mathcal{C}}, \mathbin{\circledcirc}_{\mathcal{S}}, \mathbin{\circledcirc}_{\mathcal{S}}, -\!\!\rightarrow\!-, \Lambda)$$

of $\mathbf{CDPF}(\mathcal{C},\mathcal{S})$, define an object $Y = (T^X, \eta^X, \mu^X, \tau^X, \mu^X_{S\otimes}, \mu^X_{\otimes S}, -\!\!\rightarrow\!-, \Lambda)$ as:

$$\begin{array}{rcl}T^X(S_1,S_2,A) & = & (I,S_1) \to (A,S_2)\\ \eta^X_{S,A} & = & \Lambda(\rho_A, S)\\ \mu^X_{S_1,S_2,S_3,A} & = & \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev})\\ \tau^X_{A,S_1,S_2,B} & = & \Lambda(J(\alpha, S_1); A \mathbin{\circledcirc} \mathrm{ev})\\ \mu^X_{S\otimes} & = & \Lambda(S \mathbin{\circledcirc}_{\mathcal{S}} \mathrm{ev})\\ \mu^X_{\otimes S} & = & \Lambda(\mathrm{ev} \mathbin{\circledcirc}_{\mathcal{S}} S)\\ (A,S_1) \to^X (B,S_2) & = & (A,S_1) \to (B,S_2)\\ (A,S_1) \to^X f & = & (A,S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(f))\\ \Lambda^X(f) & = & \Lambda(J(\rho^{-1}_{A\otimes B}, S_1); \Lambda^{-1}(f))\end{array}$$

See Section A.3.3 for the proof that this is an object of $\mathbf{CSMPM}(\mathcal{C}, \mathcal{S})$ and that $FX \cong Y$.

Given that F is full and faithful and essentially surjective, it follows that F is an equivalence, as required. □

5.4 Refinements of the Definitions

In this section we present several extra refinements that one might apply to the definitions above for certain situations. Each one is motivated by the monoidal state example, Example 5.3.3. This example does not actually satisfy all the conditions stated here, in particular the fullness property, but the model we give in Chapter 7 based on functor categories does.

Firstly, we consider the *mono requirement* for parameterised monads, taken from the mono requirement for normal monads [Mog91], and show that it implies that, for the induced parameterised Freyd category $J_T : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, the functor $J_T(-, S)$ is faithful for all S, and vice versa. We then extend the requirement to the unit $\eta_{A,S}$ having a right inverse for some fixed S. This is equivalent to the functor $J(-, S)$ being full. In the case of $S = I$ this says that commands that operate on the empty state are equivalent to pure values. We will use this condition for the semantics of λ_{inplc} in Chapter 8.

The second extra condition we consider is *commutativity*. Commutativity for double parameterised Freyd categories and monoidal strong parameterised monads refers to the situation when, given two commands $(A, S_1) \to (A', S_1')$ and $(B, S_2) \to (B', S_2')$, it does not matter in which order we execute them. The motivation for this is that the two commands are operating on distinct pieces of state and thus do not interfere with one another. The definition given here is not a generalisation of commutativity for strong monads [Koc72] since it relies on the parameterisation category $\mathcal{S}$. Tracking the side-effects via objects of $\mathcal{S}$ means that we can model state in a commutative double parameterised Freyd category, unlike the case for commutative Freyd categories. We will make use of commutative double parameterised Freyd categories in Chapter 8 for modelling

λ_{inplc}.

The final extension we consider is closure on the category $\mathcal{K}$ for a double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ with respect to the premonoidal structure defined with respect to $\mathcal{C} \times \mathcal{S}$. We will use this in the next chapter to interpret function types that may close over pieces of state, rather than those which have no state component, modelled by the closure defined above.

5.4.1 The Mono Requirement

The mono requirement for a monad states that all components of the unit of the monad are monomorphic. In terms of computations, the requirement states that values are included in the set of computations. Thus, if two computations constructed from pure values are equal, the pure values are equal. The parameterised Freyd category counterpart is to require that the functor $J(-, S)$ is faithful for all S.

With double parameterised Freyd categories, we intend that the type of the empty state is modelled by the monoidal unit object I in $\mathcal{S}$. Computations $(A, I) \to (B, I)$ operating on the empty state should therefore be in bijection with arrows in $A \to B$ in $\mathcal{C}$. That is, the functor $J(-, I)$ is full. The monad counterpart is to require that the arrow $\eta_{I,A}$ have a right inverse for all A.

Proposition 5.4.1 Under the constructions of Definition 5.2.8 and Theorems 5.2.17 and 5.3.14, the mono requirement and $J(-, S)$ being faithful are equivalent. The additional requirements of $J(-, I)$ being full and $\eta_{I,A}$ having a right inverse are also equivalent.

Proof $J_T(f, S) = J_T(g, S)$ implies $f; \eta_{S,B} = g; \eta_{S,B}$. By η mono, this implies $f = g$, as required. For the converse, the unit of the induced monad is $\Lambda(J(\rho, S))$. By naturality $g; \Lambda(J(\rho, S)) = \Lambda(J(\rho; g, S))$. If $\Lambda(J(\rho; g, s)) = \Lambda(J(\rho; h, S))$ then $\rho; g = \rho; h$ by the fact that Λ is an isomorphism and the faithfulness of J. Then $g = h$ since ρ has an inverse. Hence $\Lambda(J(\rho, S))$ is mono. Suppose additionally that $\eta_{I,A}$ has a right inverse $\eta'_{I,A}$. Given an arrow $f : A \to T(I, I, B)$ then $f; \eta'_{I,A} : A \to B$ is such that $J_T(f; \eta'_{I,S}, I) = f$. For the converse, define the right inverse

to $\Lambda(J(\rho, I))$ as the inverse image of $J(\rho^{-1}, I); \mathrm{ev} : ((1, I) \to (A, I), I) \to (A, I)$. Using the faithfulness of $J(-, I)$ it is easy to see this is as required. □

5.4.2 Commutativity

Definition 5.4.2 A double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is commutative if all arrows of $\mathcal{K}$ are central (Definition 5.1.2).

Proposition 5.4.3 If a double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is commutative then $\mathcal{K}$ is symmetric monoidal.

Proof Centrality means we can unambiguously define $c_1 \otimes c_2 = (c_1 \ominus_{\mathcal{C}} B) \ominus_{\mathcal{S}} S_2; S_1' \ominus_{\mathcal{S}} (A' \ominus_{\mathcal{C}} c_2)$. The natural isomorphisms are all given by the images under J of the symmetric monoidal structure on $\mathcal{C} \times \mathcal{S}$. □

Definition 5.4.4 A monoidal strong parameterised monad $(T, \eta, \mu, \tau, \mu_{\otimes S}, \mu_{S \otimes})$ is commutative if, for all arrows $c_1 : A \to T(S_1, S_1', A')$ and $c_2 : B \to T(S_2, S_2', B')$, the following two composed arrows are equal:

$$
\begin{aligned}
& A \otimes B \\
\xrightarrow{c_1 \otimes c_2} \quad & T(S_1, S_1', A') \otimes T(S_2, S_2', B') \\
\xrightarrow{\tau} \quad & T(S_2, S_2', T(S_1, S_1', A') \otimes B') \\
\xrightarrow{T(S_2, S_2', \sigma; \tau)} \quad & T(S_2, S_2', T(S_1, S_1', A' \otimes B')) \\
\xrightarrow{\mu_{S_1 \otimes}} \quad & T(S_1 \otimes S_2, S_1 \otimes S_2', T(S_1, S_1', A' \otimes B')) \\
\xrightarrow{T(S_1 \otimes S_2, S_1 \otimes S_2', \mu_{\otimes S_2'})} \quad & T(S_1 \otimes S_2, S_1 \otimes S_2', T(S_1 \otimes S_2', S_1' \otimes S_2', A' \otimes B')) \\
\xrightarrow{\mu} \quad & T(S_1 \otimes S_2, S_1' \otimes S_2', A' \otimes B')
\end{aligned}
$$

and

$$
\begin{aligned}
& A \otimes B \\
\xrightarrow{c_1 \otimes c_2} \quad & T(S_1, S_1', A') \otimes T(S_2, S_2', B') \\
\xrightarrow{\sigma; \tau} \quad & T(S_1, S_1', T(S_2, S_2', B') \otimes A') \\
\xrightarrow{T(S_1, S_1', \sigma; \tau)} \quad & T(S_1, S_1', T(S_2, S_2', A' \otimes B'))
\end{aligned}
$$

$$\begin{array}{rl} \xrightarrow{\mu_{\otimes S_2}} & T(S_1 \otimes S_2, S_1' \otimes S_2, T(S_2, S_2', A' \otimes B')) \\ \xrightarrow{T(S_1 \otimes S_2, S_1' \otimes S_2, \mu_{S_1' \otimes})} & T(S_1 \otimes S_2, S_1' \otimes S_2, T(S_1' \otimes S_2, S_1' \otimes S_2', A' \otimes B')) \\ \xrightarrow{\mu} & T(S_1 \otimes S_2, S_1' \otimes S_2', A' \otimes B') \end{array}$$

Proposition 5.4.5 If a double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is commutative, then the induced monoidal strong parameterised monad is. Conversely, if a monoidal strong parameterised monad $(T, \eta, \mu, \tau, \mu_{\otimes S}, \mu_{S\otimes})$ is commutative then the induced double parameterised Freyd category $J_T : \mathcal{C} \times \mathcal{S} \to \mathcal{C}_T$ is commutative.

Proof By long tedious calculation. □

5.4.3 $\mathcal{K}$-Closure and Typed Command Categories

Definition 5.4.6 A double parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is $\mathcal{K}$-closed if, for all objects $A \in \mathsf{Ob}\mathcal{C}$ and $S \in \mathsf{Ob}\mathcal{S}$, the functor $(- \ltimes_{\mathcal{C}} A) \ltimes_{\mathcal{S}} S$ has a specified right adjoint.

We will write the right adjoint as $(A, S_1) \multimap -$. Note that this definition also works for monoidal strong parameterised monads by requiring the adjoint on the Kleisli Category.

Since each object in $\mathcal{K}$ is actually a pair of objects in $\mathcal{C}$ and $\mathcal{S}$, the object $(A, S_1) \multimap (B, S_2)$ must be represented by a pair of objects. However, we will write it as a single object in its own right. This is motivated by the examples below: in the pair categories example, both components are dependent on the exact function space we are describing; but in the typed monoidal state example, the $\mathcal{S}$ component is always I.

For discussing $\mathcal{K}$-closure it is useful to have a notation of monoidal structure on objects in $\mathcal{K}$. We define $(A, S_1) \otimes (B, S_2) = (A \times B, S_1 \otimes S_2)$. Note that, unless all arrows in $\mathcal{K}$ are central, there is no corresponding structure on arrows.

Using this notation, we write the counit of this adjunction as:

$$\mathrm{ev}^{\multimap}_{A,S_1,B,S_2} : (A, S_1) \multimap (B, S_2) \otimes (A, S_1) \to (B, S_2)$$

Example 5.4.7 (Pair categories) In the case of Example 5.3.2, the parameterised premonoidal category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{C} \times \mathcal{S}$ is $\mathcal{K}$-closed when $\mathcal{C}$ and $\mathcal{S}$ are symmetric monoidal closed. Take the functor to be $(A, S_1) \multimap (B, S_2) = (A \multimap B, S_1 \multimap S_2)$. The isomorphism of homsets is derived directly from the closed structure on $\mathcal{C}$ and $\mathcal{S}$.

Example 5.4.8 (Typed Monoidal State) The monoidal state example, Example 5.3.3, is $\mathcal{K}$-closed when $\mathcal{C}$ is closed. A suitable choice for the right adjoint is:

$$(A, S_1) \multimap (B, S_2) = (A \times \widehat{S_1} \to B \times \widehat{S_2}, I)$$

The isomorphism of homsets is given by the closed structure of $\mathcal{C}$.

Proposition 5.4.9 If $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is commutative and $\mathcal{K}$-closed then it is symmetric monoidal closed.

Proof Follows directly from the definitions. □

We gather the definition of closed double parameterised Freyd category with $\mathcal{K}$-closure into a single definition, Typed Command Category.

Definition 5.4.10 (Typed Command Category) A *Typed Command Category* is a double parameterised Freyd category with $\mathcal{K}$-closure such that symmetric monoidal structure on $\mathcal{C}$ is given by finite products.

Chapter 6

Typed Command Calculus

In this chapter we define a substructural typed λ-calculus that is sound and complete for the Typed Command Categories of the previous chapter.

In Section 6.1, we describe the design of the calculus, based on the fine-grain call-by-value calculus of Levy, Power and Thielecke [LPT03]. The basic structure is that there are three calculi, one for each of the categories in a Double Parameterised Freyd-category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$. We describe the state calculus, modelled by the category $\mathcal{S}$, in Section 6.2.1. The value and command calculi, modelled by $\mathcal{C}$ and $\mathcal{K}$ respectively, are described in Section 6.2.2. They must be described together since the definition of value closure requires that the two calculi be mutually defined. In Section 6.2.3 we define Typed Command Theories and the equational judgements they generate.

We describe the interpretation in Section 6.3. Since the state calculus is independent, we give its interpretation in isolation in a symmetric monoidal category in Section 6.3.1. We describe the modelling of the whole calculus in Typed Command Categories in Section 6.3.2. We define models of Typed Command Theories in Section 6.3.3 and prove that they are sound and complete.

6.1 Design of the Calculus

The design of the Typed Command calculus is inspired by the fine-grain call-by-value calculus of Levy, Power and Thielecke. [LPT03]. They define a calculus

designed to be modelled in closed Freyd categories that syntactically distinguishes a set of terms that are "values", modelled by arrows of the domain category, from "producers" that are modelled in the codomain category. They have two type judgements, $\Gamma \vdash^{\mathsf{v}} V : A$ and $\Gamma \vdash^{\mathsf{p}} M : A$, for values and producers respectively. Values are included into the set of producers by the rule:

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{p}} \mathtt{produce}\ V : A}$$

Two producers are sequenced by a binding construct:

$$\frac{\Gamma \vdash^{\mathsf{p}} M : A \qquad \Gamma, x : A \vdash^{\mathsf{p}} N : B}{\Gamma \vdash^{\mathsf{p}} M\ \mathtt{to}\ x.N : B}$$

Following this idea of separate type judgements for terms that are interpreted separately in the model we construct three calculi, one for each of the three categories involved in a closed parameterised Freyd-category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$. Because of the adjunction defining the closure relating the categories $\mathcal{C}$ and $\mathcal{K}$, the calculi corresponding to these categories will be mutually defined. The three judgements have the form:

$$\Gamma \vdash e : A \qquad\qquad \Delta \vdash s : S \qquad\qquad \Gamma; \Delta \vdash c : A; S$$

for the judgements corresponding to $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{K}$ respectively. We will call the three judgements "value", "state" and "command" judgements, and similarly for the calculi they type. We include the value and state terms into the command calculus by the rule C-V-S:

$$\frac{\Gamma \vdash e : A \qquad \Delta \vdash s : S}{\Gamma; \Delta \vdash (e; s) : A; S}$$

This rule will be interpreted by the functor J. We now describe each of the three calculi in turn.

Corresponding to the category $\mathcal{S}$ of state descriptions and manipulations will be a basic substructural calculus, the *state calculus*. We formally describe this

calculus in Section 6.2.1. The types S of the state calculus will be used to describe the states of the store at the beginning and end of each command. The terms of the state calculus are intended to describe manipulations of these descriptions that do not alter or update the store. For example, we can rearrange store descriptions by associativity and exchange and eliminate/introduce descriptions of the empty store. The calculus has one type operator: $S_1 \otimes S_2$, representing the store made of two disjoint parts described by the descriptions S_1 and S_2, and one type constant I, representing the empty store. As with the other substructural calculi in this thesis, we control the rules governing the $S_1 \otimes S_2$ by controlling the structural rules. Hence, the state calculus is a basic substructural type system with only exchange, associativity and unit elimination structural rules. The state calculus is the same as the basic substructural calculus described in the introduction, Section 1.1.

The finite product category $\mathcal{C}$ is used to model the *value calculus*. This calculus has the standard constructs for unit and product types, interpreted by the finite product structure of $\mathcal{C}$. Contexts Γ are used in an intuitionistic manner; weakening, contraction and exchange are all admissible rules. The only non-standard part of this calculus is the rule V-$\to$I for introducing λ-abstracted commands with no free state variables:

$$\frac{\Gamma, x : A; z : S_1 \vdash c : B; S_2}{\Gamma \vdash \lambda(x; z) : (A; S_1).c : (A; S_1) \to (B; S_2)}$$

where the premise of the rule is a judgement of the command calculus.

The typing rules of the command calculus serve three broad purposes: the integration of the value and state calculi, corresponding to the action of the functor J; the sequencing of commands, corresponding to the premonoidal structure of J; and the rules dealing with function types.

The integration of the value and state judgements into the command calculus is accomplished by the rule C-V-S above which incorporates pairs of the value and state terms into the command calculus.

Sequencing of commands in context, semantically modelled by the premonoidal structure, is via a "let" construct. We are following Moggi's syntax for the com-

putational λ-calculus rather than Levy *et al*'s "to" notation. We have a unary C-Let expression:

$$\frac{\Gamma; \Delta_1 \vdash c_1 : A; S_1 \qquad \Gamma, x : A; \Delta_2, z : S_1 \vdash c_2 : B; S_2}{\Gamma; \Delta_2 \bowtie \Delta_1 \vdash \text{let } (x; z) \Leftarrow c_1 \text{ in } c_2 : B; S_2}$$

(the $\bowtie$ operator merges two contexts while not allowing duplicates). Due to the substructural nature of the state calculus we must also provide a way to eliminate state pair types $S_1 \otimes S_2$ in the command calculus, while preserving sequencing:

$$\frac{\Gamma; \Delta_1 \vdash c_1 : A; S_1 \otimes S_2 \qquad \Gamma, x : A; \Delta_2, z_1 : S_1, z_2 : S_2 \vdash c_2 : B; S_3}{\Gamma; \Delta_2 \bowtie \Delta_1 \vdash \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2 : B; S_3} \text{ (C-Let-}\otimes\text{)}$$

To see why this expression is required, consider a program involving these three commands:

$$\begin{aligned} \text{allocatePair} &: (1, I) \to (1, \text{cell} \otimes \text{cell}) \\ \text{storeInt} &: (\text{Int}, \text{cell}) \to (1, \text{cell[Int]}) \\ \text{storeBool} &: (\text{Bool}, \text{cell}) \to (1, \text{cell[Bool]}) \end{aligned}$$

We can use allocatePair:

$$\text{let } (x; z) \Leftarrow \text{allocatePair}(*_1, *_I) \text{ in } \ldots$$

However, without C-Let-$\otimes$ there would be no way to decompose the variable z bound in the body of this expression so we could use the two sides in two different commands. Using C-Let-$\otimes$ allows us to do this:

$$\begin{aligned} &\text{let } (x; z_1, z_2) \Leftarrow \text{allocatePair}(*_1; *_I) \text{ in} \\ &\text{let } (x; z_1') \Leftarrow \text{storeInt}(42; z_1) \text{ in} \\ &\text{let } (x; z_2') \Leftarrow \text{storeBool}(\text{true}; z_2) \text{ in} \\ &\quad (*_1; (z_1', z_2')) \end{aligned}$$

There is also a complimentary rule C-Let-I for eliminating variables of state type I.

Finally, there is a rule for eliminating closed commands, C-→E:

$$\frac{\Gamma \vdash e : (A; S_1) \to (B; S_2) \qquad \Gamma; \Delta \vdash c : A; S_1}{\Gamma; \Delta \vdash e @_{\to} c : B; S_2}$$

There are also introduction and elimination rules for functions which may contain free state variables: C-—∘I and C-—∘E. The value and command calculi are defined in Section 6.2.2.

6.2 Typed Command Calculus

A *Typed Command System* $(\mathcal{T}_V, \mathcal{T}_S, \Phi_V, \Phi_S, \Phi_C)$ consists of a set of primitive value types $\mathcal{T}_V$; a set of primitive state types $\mathcal{T}_S$; a set of primitive value operations $(f : A \to B) \in \Phi_V$, where A and B are value types generated by the grammar below; a set of primitive state operations $(p : S_1 \to S_2) \in \Phi_S$, where S_1 and S_2 are state types generated by the grammar below; and a set of primitive commands $(p : (A; S_1) \to (B; S_2)) \in \Phi_C$, where A and B are value types and S_1 and S_2 are state types. For types A and B, we will write $\Phi_V(A, B)$ for the subset of Φ_V of the form $f : A \to B$. Similarly for Φ_S and Φ_C.

Value types are generated from the primitive value types and are ranged over by A, B, etc.

$$A, B \;::=\; X \in \mathcal{T}_V \;\mid\; 1 \;\mid\; A \times B \;\mid\; (A; S_1) \to (B; S_2) \;\mid\; (A; S_1) \multimap_V (B; S_2)$$

where S_1 and S_2 are state types generated by this grammar:

$$S \;::=\; X \in \mathcal{T}_S \;\mid\; I \;\mid\; S_1 \otimes S_2 \;\mid\; (A; S_1) \multimap_S (B; S_2)$$

The value types are constructed from the primitive types by the unit 1 and product $\times$ constructors, and two function types. The value function type $(A; S_1) \to (B; S_2)$ is for functions with no free state variables. The semantical counterpart of this type is the adjunction between the categories $\mathcal{K}$ and $\mathcal{C}$. The command procedure type $(A; S_1) \multimap_V (B; S_2)$ will be interpreted by the $\mathcal{C}$ component of

the $\mathcal{K}$-closure. The state types are constructed from the primitive types, a substructural unit I, substructural pairs $\otimes$ and the state component of command procedures $(A; S_1) \multimap_S (B; S_2)$.

Typed Command Systems also generate three sets of terms and three typing judgements. Since the state type judgements are independent from the others we describe them separately in the first subsection 6.2.1. The value and command judgements are described in Section 6.2.2. In the final subsection 6.2.3 we give the definition of a Typed Command Theory and the rules for generating equational judgements.

6.2.1 State Calculus

A system generates state contexts Δ by the following grammar:

$$\Delta \quad ::= \quad I \quad | \quad \Delta, z : S$$

As usual, no variable name may appear more than once in a context. The notation $\Delta[z]$ is used to denote the type of the variable z in the context Δ, assuming it is present.

Pairs of contexts are joined using the non-deterministic merging operator $\Delta_1 \bowtie \Delta_2$ when Δ_1 and Δ_2 have disjoint sets of variable names. This operator takes the place of the structural transitions of the other calculi in this thesis. It ensures that if $\Delta_1 \bowtie \Delta_2 = \Delta$ then Δ has the same variables and type assignments as Δ_1 and Δ_2, merged in some order. The operator is defined by the clauses:

$$\begin{aligned} I \bowtie I &= I \\ (\Delta_1, z : A) \bowtie \Delta_2 &= (\Delta_1 \bowtie \Delta_2), z : A \\ \Delta_1 \bowtie (\Delta_2, z : A) &= (\Delta_1 \bowtie \Delta_2), z : A \end{aligned}$$

A system generates a set of terms by the following grammar, where $f \in \Phi_S$:

$$s \quad ::= \quad z \quad | \quad (s_1, s_2) \quad | \quad \star_I \quad | \quad fs$$

$$\begin{array}{ll} | & \text{let } (z_1, z_2) = s_1 \text{ in } s_2 \\ | & \text{let } \star_I = s_1 \text{ in } s_2 \end{array}$$

The choice of term constructs in governed by our choice of type operators. There are terms for introducing and eliminating pair types and for introducing and eliminating the unit type. There are also the standard terms for primitives and variables.

A system generates a typing judgement $\Delta \vdash s : S$ by the rules in Figure 6.1. Due to the separate contexts in the rules S-$\otimes$I, S-$\otimes$E and S-II, contraction is clearly not admissible. Likewise, due to the single variable in the S-Id rule, weakening is not admissible. The only structural rule beyond the basic ones is therefore Exchange.

$$\frac{}{x : S \vdash x : S}\ (\text{S-Id}) \qquad \frac{\Delta_1 \vdash s_1 : S_1 \qquad \Delta_2 \vdash s_2 : S_2}{\Delta_1 \bowtie \Delta_2 \vdash (s_1, s_2) : S_1 \otimes S_2}\ (\text{S-}\otimes\text{I})$$

$$\frac{\Delta_1 \vdash s_1 : S_1 \otimes S_2 \qquad \Delta_2, z_1 : S_1, z_2 : S_2 \vdash s_2 : S_3}{\Delta_1 \bowtie \Delta_2 \vdash \text{let } (z_1, z_2) = s_1 \text{ in } s_2 : S_3}\ (\text{S-}\otimes\text{E}) \qquad \frac{}{I \vdash \star_I : I}\ (\text{S-}I\text{I})$$

$$\frac{\Delta_1 \vdash s_1 : I \qquad \Delta_2 \vdash s_2 : S}{\Delta_1 \bowtie \Delta_2 \vdash \text{let } \star_I = s_1 \text{ in } s_2 : S}\ (\text{S-}I\text{E})$$

$$\frac{\Delta \vdash s : S_1 \qquad (f : S_1 \longrightarrow S_2) \in \Phi_S}{\Delta \vdash fs : S_2}\ (\text{S-Prim})$$

Figure 6.1: Typing rules for the State Calculus

Lemma 6.2.1 (State Calculus Substitution) The following rule is admissible in any state system:

$$\frac{\Delta_1 \vdash s_1 : S_1 \qquad \Delta_2, z : S_1 \vdash s_2 : S_2}{\Delta_1 \bowtie \Delta_2 \vdash s_2[s_1/z] : S_2}$$

Proof By induction on the derivation of $\Delta_2, z : S_1 \vdash s_2 : S_2$. □

6.2.2 Value and Command Calculi

A system generates a set of value contexts, ranged over by Γ:

$$\Gamma \quad ::= \quad \epsilon \quad | \quad \Gamma, x : A$$

A system generates two further typing judgements:

$$\Gamma \vdash e : A \qquad\qquad \Gamma; \Delta \vdash c : A; S$$

where the terms e and c are generated by the following two grammars respectively:

$$e \quad ::= \quad x \quad | \quad \star_1 \quad | \quad (e_1, e_2) \quad | \quad \pi_1 e \quad | \quad \pi_2 e \quad | \quad f e \quad | \quad \lambda^{\rightarrow}(x; z) : (A; S).c$$

$$c \quad ::= \quad (e; s) \quad | \quad \text{let } (x; z) \Leftarrow c_1 \text{ in } c_2 \quad | \quad \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2$$

$$| \quad \text{let } (x; \star_I) \Leftarrow c_1 \text{ in } c_2 \quad | \quad pc \quad | \quad e@_{\rightarrow} c \quad | \quad \lambda^{\multimap}(x; z) : (A; S).c \quad | \quad c_1@_{\multimap} c_2$$

We assume that the variables used in value expressions are disjoint from the variables used in the state calculus. This will simplify the expression of the substitution rules.

The rules for deriving typing judgements are shown in Figures 6.2 and 6.3. The rules are as described in Section 6.1. The rules for the value system are standard for a typed calculus with only a unit type 1 and product type $A_1 \times A_2$. Note that we have used a context with more than one variable in the variable introduction rule V-Id, and shared contexts in the product introduction rule V-$\times$E; consequently, the calculus admits Weakening, Exchange and Contraction. The rule V-$\rightarrow$I links the two calculi by typing $\lambda^{\rightarrow}$-abstracted commands with no free state variables as values.

The command calculus rules C-Struct and C-V-S incorporate the structural rules of the state calculus and value and state terms into the command calculus respectively. There are three sequencing rules: C-Let, the unary sequencing rule; and C-Let-$\otimes$ and C-Let-I, which sequence commands and eliminate state products and units respectively. The C-$\rightarrow$E eliminates abstracted commands generated by the V-$\rightarrow$I rule. Note that the first premise of this rule is a value calculus typing judgement. Finally, the C-Prim rule incorporates the primitive commands into the command calculus.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\text{V-ID}) \qquad \frac{}{\Gamma \vdash \star_1 : 1}\ (\text{V-1I}) \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2}\ (\text{V-}\times\text{I})$$

$$\frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i e : A_i}\ (\text{V-}\times\text{E}) \qquad \frac{\Gamma, x : A; z : S_1 \vdash c : B; S_2}{\Gamma \vdash \lambda^{\rightarrow}(x; z) : (A; S_1).c : (A, S_1) \rightarrow (B, S_2)}\ (\text{V-}\rightarrow\text{I})$$

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

Figure 6.2: Typing rules of the Value Calculus

To prove substitution we need the following weakening lemma. The slightly odd form of the contexts here makes it easier to apply in the proof that substitution is admissible.

Lemma 6.2.2 (Weakening) The following rules are admissible, when $\Gamma, \Gamma'', \Gamma'$ is a valid value-context.

$$\frac{\Gamma, \Gamma' \vdash e : A}{\Gamma, \Gamma'', \Gamma' \vdash e : A} \qquad \frac{\Gamma, \Gamma'; \Delta \vdash c : A; S}{\Gamma, \Gamma'', \Gamma'; \Delta \vdash c : A; S}$$

Proof By mutual induction on the derivations of $\Gamma, \Gamma' \vdash e : A$ and $\Gamma, \Gamma'; \Delta \vdash c : A; S$. □

Lemma 6.2.3 (Substitution) For any Typed Command System, the following rules are admissible:

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A, \Gamma' \vdash e_2 : B}{\Gamma, \Gamma' \vdash e_2[e_1/x] : B}$$

and

$$\frac{\Gamma \vdash e : A \qquad \Delta_1 \vdash s : S_1 \qquad \Gamma, x : A, \Gamma'; \Delta_2, z : S_1 \vdash c : B; S_2}{\Gamma, \Gamma'; \Delta_1 \bowtie \Delta_2 \vdash c[e/x, s/z] : B; S_2}$$

$$\frac{\Gamma \vdash e : A \qquad \Delta \vdash s : S}{\Gamma; \Delta \vdash (e; s) : A; S} \text{ (C-V-S)}$$

$$\frac{\Gamma; \Delta_1 \vdash c_1 : A; S_1 \qquad \Gamma, x : A; \Delta_2, z : S_1 \vdash c_2 : B; S_2}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash \text{let } (x; z) \Leftarrow c_1 \text{ in } c_2 : B; S_2} \text{ (C-Let)}$$

$$\frac{\Gamma; \Delta_1 \vdash c_1 : A; S_1 \otimes S_2 \qquad \Gamma, x : A; \Delta_2, z_1 : S_1, z_2 : S_2 \vdash c_2 : B; S_3}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2 : B; S_3} \text{ (C-Let-}\otimes\text{)}$$

$$\frac{\Gamma; \Delta_1 \vdash c_1 : A; I \qquad \Gamma, x : A; \Delta_2 \vdash c_2 : B; S_3}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash \text{let } (x; \star_I) \Leftarrow c_1 \text{ in } c_2 : B; S_3} \text{ (C-Let-}I\text{)}$$

$$\frac{\Gamma \vdash e : (A; S_1) \to (B; S_2) \qquad \Gamma; \Delta \vdash c : A; S_1}{\Gamma; \Delta \vdash e@_{\to} c : B; S_2} \text{ (C-}\to\text{E)}$$

$$\frac{\Gamma, x : A; \Delta, z : S_1 \vdash c : B; S_2}{\Gamma; \Delta \vdash \lambda^{\multimap}(x; z) : (A; S_1).c : (A; S_1) \multimap_V (B; S_2); (A; S_1) \multimap_S (B; S_2)} \text{ (C-}\multimap\text{I)}$$

$$\frac{\Gamma; \Delta_1 \vdash c_1 : (A; S_1) \multimap_V (B; S_2); (A; S_1) \multimap_S (B; S_2) \qquad \Gamma; \Delta_2 \vdash c_2 : A; S_1}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash c_1 @_{\multimap} c_2 : B; S_2} \text{ (C-}\multimap\text{E)}$$

$$\frac{\Gamma; \Delta \vdash c : A; S_1 \qquad (p : (A; S_1) \longrightarrow (B; S_2)) \in \Phi_C}{\Gamma; \Delta \vdash pc : B; S_2} \text{ (C-Prim)}$$

Figure 6.3: Typing rules of the Command Calculus

Proof We must first prove the following pair of rules admissible by mutual induction over the derivations of $\Gamma, x : A, \Gamma' \vdash e_1 : B$ and $\Gamma, x : A, \Gamma'; \Delta \vdash c : B; S$.

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A, \Gamma' \vdash e_2 : B}{\Gamma, \Gamma' \vdash e_2[e_1/x] : B} \qquad \frac{\Gamma \vdash e : A \qquad \Gamma, x : A, \Gamma'; \Delta \vdash c : B; S}{\Gamma, \Gamma'; \Delta \vdash c[e/x] : B; S}$$

All the cases are straightforward, using Lemma 6.2.2 for the V-Id case. We prove the second rule in the statement of the Lemma is admissible by induction over the derivation of $\Gamma, x : A, \Gamma'; \Delta_2(z : S_1) \vdash c : B; S_2$. This requires the use of both of the value-only substitution rules and Lemma 6.2.1 for substitution into state judgements. □

6.2.3 Equational Theory

A *Typed Command Theory* is an 8-tuple $(\mathcal{T}_V, \mathcal{T}_S, \Phi_V, \Phi_S, \Phi_C, \Sigma_V, \Sigma_S, \Sigma_C)$, where $(\mathcal{T}_V, \mathcal{T}_C, \Phi_V, \Phi_S, \Phi_C)$ is a Typed Command System; Σ_V is a set of value axioms $\Gamma \vdash e_1 = e_2 : A$; Σ_S is a set of state axioms $\Delta \vdash s_1 = s_2 : S$; and Σ_C is a set of command axioms $\Gamma; \Delta \vdash c_1 = c_2 : A; S$, such that both sides of every axiom form derivable typing judgements in the relevant component of the theory.

A theory generates a three equational judgements:

$$\Gamma \vdash e_1 = e_2 : A \qquad \Delta \vdash s_1 = s_2 : S \qquad \Gamma; \Delta \vdash c_1 = c_2 : A; S$$

by the rules in Figures 6.4, 6.5, 6.6 and 6.7.

The rules for the value calculus in Figure 6.5 are the standard ones for a non-substructural calculus with unit and product types. Likewise, the rules for the state calculus in Figure 6.4 are standard for a substructural calculus with substructural unit and product types, using Ghani's generalised η rule [Gha95].

The rules for the command calculus (Figures 6.6 and 6.7) require more explanation. The rule Eq-C-Ax incorporates the axioms of the Typed Command Theory into the equational judgements. The rule Eq-C-V-S lifts equational judgements from the value and state calculi into the command calculus. For each of the three "let" constructs there are β and η rules, plus a commuting conversion rule for each. It is not possible to use Ghani's extended η rule in the command

$$\frac{(\Delta \vdash s_1 = s_2 : S) \in \Sigma_S}{\Delta \vdash s_1 = s_2 : S}\ (\text{Eq-S-Ax})$$

$$\frac{\Delta_1 \vdash s_1 : S_1 \qquad \Delta_2 \vdash s_2 : S_2 \qquad \Delta_3, z_1 : S_1, z_2 : S_2 \vdash s_3 : S_3}{\Delta_3 \bowtie \Delta_1 \bowtie \Delta_2 \vdash (\text{let } (z_1, z_2) = (s_1, s_2) \text{ in } s_3) = s_2[s_1/z_1, s_2/z_2] : S_3}\ (\text{Eq-S-}\otimes\beta)$$

$$\frac{\Delta_1 \vdash s_1 : S_1 \otimes S_2 \qquad \Delta_2, z : S_1 \otimes S_2 \vdash s_2 : S_3}{\Delta_1 \bowtie \Delta_2 \vdash (\text{let } (z_1, z_2) = s_1 \text{ in } s_2[z_1 \otimes z_2/z]) = s_2[s_1/z] : S_3}\ (\text{Eq-S-}\otimes\eta)$$

$$\frac{\Delta(I) \vdash s : S}{\Delta(I) \vdash (\text{let } \star_I = \star_I \text{ in } s) = s : S}\ (\text{Eq-S-}I\beta)$$

$$\frac{\Delta_1 \vdash s_1 : I \qquad \Delta_2, z : I \vdash s_2 : S}{\Delta_1 \bowtie \Delta_2 \vdash (\text{let } \star_I = s_1 \text{ in } s_2[\star_I/z]) = s_2[s_1/z] : S}\ (\text{Eq-S-}I\eta)$$

Plus congruence, identity, symmetry and transitivity rules.

Figure 6.4: Equational rules for the State calculus

$$\frac{(\Gamma \vdash e_1 = e_2 : A) \in \Sigma_V}{\Gamma \vdash e_1 = e_2 : A}\ (\text{Eq-V-Ax}) \qquad \frac{\Gamma \vdash (e_1, e_2) : A_1 \times A_2}{\Gamma \vdash \pi_i(e_1, e_2) = e_i : A_i}\ (\text{Eq-V-}\times\beta)$$

$$\frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash (\pi_1 e, \pi_2 e) = e : A_1 \times A_2}\ (\text{Eq-V-}\times\eta) \qquad \frac{\Gamma \vdash e : 1}{\Gamma \vdash e = \star : 1}\ (\text{Eq-V-1})$$

$$\frac{\Gamma \vdash f : (A, S_1) \to (B, S_2)}{\Gamma \vdash (\lambda^{\to}(x, z).f@_{\to}(x; z)) = f : (A, S_1) \to (B, S_2)}\ (\text{Eq-V-}\eta)$$

Figure 6.5: Value Equational Rules

$$\frac{(\Gamma; S_1 \vdash c_1 = c_2 : A; S_2) \in \Sigma_C}{\Gamma; S_1 \vdash c_1 = c_2 : A; S_2} \text{ (Eq-C-Ax)}$$

$$\frac{\Gamma \vdash e_1 = e_2 : A \qquad \Delta \vdash s_1 = s_2 : S}{\Gamma; \Delta \vdash (e_1; s_1) = (e_2; s_2) : A; S} \text{ (Eq-C-V-S)}$$

$$\frac{\Gamma \vdash e : A \qquad \Delta_1 \vdash s : S_1 \qquad \Gamma, x : A; \Delta_2, z : S_1 \vdash c : B; S_2}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash (\text{let } (x; z) \Leftarrow (e; s) \text{ in } c) = c[e/x, s/z] : B; S_2} \text{ (Eq-C-Let-}\beta\text{)}$$

$$\frac{\Gamma; \Delta \vdash c : A; S}{\Gamma; \Delta \vdash (\text{let } (x; z) \Leftarrow c \text{ in } (x; z)) = c : A; S} \text{ (Eq-C-Let-}\eta\text{)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : A \qquad \Delta_1 \vdash s_1 : S_1 \\ \Delta_2 \vdash s_2 : S_2 \qquad \Gamma, x : A; \Delta_3, z_1 : S_1, z_2 : S_2 \vdash c : B; S_3\end{array}}{\begin{array}{l}\Gamma; \Delta_1 \bowtie (\Delta_2 \bowtie \Delta_3) \vdash \ (\text{let } (x; z_1, z_2) \Leftarrow (e; (s_1, s_2)) \text{ in } c) = \ : B; S_3 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad c[e/x, s_1/z_1, s_2/z_2]\end{array}} \text{ (Eq-C-Let-}\otimes\beta\text{)}$$

$$\frac{\Gamma; \Delta \vdash c : A; S_1 \otimes S_2}{\Gamma; \Delta \vdash (\text{let } (x; z_1, z_2) \Leftarrow c \text{ in } (x; (z_1, z_2))) = c : A; S_1 \otimes S_2} \text{ (Eq-C-Let-}\otimes\eta\text{)}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A; \Delta \vdash c : B; S}{\Gamma; \Delta \vdash (\text{let } (x; \star_I) \Leftarrow (e; \star_I) \text{ in } c) = c[e/x] : B; S} \text{ (Eq-C-Let-}I\beta\text{)}$$

$$\frac{\Gamma; \Delta \vdash c : A; I}{\Gamma; \Delta \vdash (\text{let } (x; \star_I) \Leftarrow c \text{ in } (x; \star_I)) = c : A; I} \text{ (Eq-C-Let-}\otimes\eta\text{)}$$

$$\frac{\Gamma, x : A; z : S_1 \vdash c : B; S_2 \qquad \Gamma; \Delta \vdash a : A; S_1}{\Gamma; \Delta \vdash (\lambda^{\rightarrow}(x, z).c)@_{\rightarrow} a = (\text{let } (x; z) \Leftarrow a \text{ in } c) : B; S_2} \text{ (Eq-C-}\rightarrow \beta\text{)}$$

Figure 6.6: Equational Rules of the Command Calculus, Part 1

$$\frac{\Gamma;\Delta_1 \vdash c_1 : A; S_1 \qquad \Gamma, x : A; \Delta_2, z : S_1 \vdash C[c_2] : B; S_2 \qquad C[-] \text{ does not bind of contain } x \text{ or } z}{\Gamma;\Delta_1 \bowtie \Delta_2 \vdash C[\text{let } (x; z) \Leftarrow c_1 \text{ in } c_2] = \text{let } (x; z) \Leftarrow c_1 \text{ in } C[c_2] : B; S_2} \text{ (Eq-C-Let-CC)}$$

$$\frac{\begin{array}{c}\Gamma;\Delta_1 \vdash c_1 : A; S_1 \otimes S_2 \\ \Gamma, x : A; \Delta_2, z_1 : S_1, z_2 : S_2 \vdash C[c_2] : B; S_3 \\ C[-] \text{ does not bind of contain } x, z_1 \text{ or } z_2\end{array}}{\begin{array}{l}\Gamma;\Delta_1 \bowtie \Delta_2 \vdash \quad C[\text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2] = \quad : B; S_3 \\ \qquad\qquad\qquad\qquad \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } C[c_2]\end{array}} \text{ (Eq-C-Let-CC-}\otimes\text{)}$$

$$\frac{\Gamma;\Delta_1 \vdash c_1 : A; \star_I \qquad \Gamma, x : A; \Delta_2 \vdash C[c_2] : B; S_2}{\Gamma;\Delta_1 \bowtie \Delta_2 \vdash C[\text{let } (x; \star_I) \Leftarrow c_1 \text{ in } c_2] = \text{let } (x; \star_I) \Leftarrow c_1 \text{ in } C[c_2] : B; S} \text{ (Eq-C-Let-CC-}I\text{)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B \\ \Delta_1 \vdash s_1 : S_1 \otimes S_2 \qquad \Delta_2, z_1 : S_1, z_2 : S_2 \vdash s_2 : S_3\end{array}}{\begin{array}{l}\Gamma;\Delta_1 \bowtie \Delta_1 \vdash \quad \text{let } (x; z_1, z_2) \Leftarrow (e_1; s_1) \text{ in } (e_2; s_2) = \quad : B; S_3 \\ \qquad\qquad\qquad\qquad (e_2[e_1/x]; \text{let } (z_1, z_2) \Leftarrow s_1 \text{ in } s_2)\end{array}} \text{ (Eq-C-Let-}\otimes\beta\text{-2)}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B \qquad \Delta \vdash s : S}{\Gamma;\Delta \vdash \text{let } (x; \star_I) \Leftarrow (e_1; \star_I) \text{ in } (e_2; s) = (e_2[e_1/x]; s) : B; S} \text{ (Eq-C-Let-}I\beta\text{-2)}$$

$$\frac{\Gamma, x : A; \Delta_1, z : S_1 \vdash c : B; S_2 \qquad \Gamma;\Delta_2 \vdash a : A; S_1}{\Gamma;\Delta_1 \bowtie \Delta_2 \vdash (\lambda^{\multimap}(x; z).c)@_{\multimap}a = (\text{let } (x; z) \Leftarrow a \text{ in } c) : B; S_2} \text{ (Eq-C-}\multimap\beta\text{)}$$

$$\frac{\Gamma;\Delta \vdash c : (A; S_1) \multimap_V (B; S_2); (A; S_1) \multimap_S (B; S_2)}{\Gamma;\Delta \vdash (\lambda^{\multimap}(x; z).c@(x; z)) = c : (A; S_1) \multimap_V (B; S_2); (A; S_1) \multimap_S (B; S_2)} \text{ (Eq-C-}\multimap\eta\text{)}$$

Plus reflexivity, symmetry, transitivity and congruence rules.

Figure 6.7: Equational Rules of the Command Calculus, Part 2

calculus because it does not have its own variables, only pairs of variables from the value and state calculi, so there is no way to refer to "holes" in terms where the substitution may take place.

The contexts for the rules Eq-C-Let-CC, Eq-C-Let-CC-$\otimes$ and Eq-C-Let-CC-I are generated by the grammar:

$$\begin{array}{rcl} C[-] & = & - \quad | \quad p\ C[-] \\ & | & \text{let } (x;z) \Leftarrow C[-] \text{ in } c \quad | \quad \text{let } (x;z) \Leftarrow c \text{ in } C[-] \\ & | & \text{let } (x;z_1,z_2) \Leftarrow C[-] \text{ in } c \quad | \quad \text{let } (x;z_1,z_2) \Leftarrow c \text{ in } C[-] \\ & | & \text{let } (x;\star_I) \Leftarrow C[-] \text{ in } c \quad | \quad \text{let } (x;\star_I) \Leftarrow c \text{ in } C[-] \\ & | & e@_{\rightarrow}C[-] \quad | \quad \lambda^{\multimap}(x;z).C[-] \quad | \quad C[-]@_{\multimap}c \quad | \quad c@_{\multimap}C[-] \end{array}$$

There are also two extra β rules for the "let" commands constructed by the rules C-Let-$\otimes$ and C-Let-I; they take eliminations that have been incorporated into the command calculus and lift them into the value and state calculi. Finally, there are $\beta\eta$ rules for $\lambda^{\rightarrow}$-abstracted commands, Eq-C-$\rightarrow$ β and Eq-V-$\rightarrow$ η, and the $\beta\eta$ rules for $\lambda^{\multimap}$-abstracted commands, Eq-C-$\multimap$ β and Eq-C-$\multimap$ η.

All the equational judgements generated by a Typed Command Theory are well-formed:

Proposition 6.2.4 The following three implications hold:

$$\begin{array}{rcl} \Gamma \vdash e_1 = e_2 : A & \Rightarrow & \Gamma \vdash e_1 : A \text{ and } \Gamma \vdash e_2 : A \\ \Delta \vdash s_1 = s_2 : S & \Rightarrow & \Delta \vdash s_1 : S \text{ and } \Delta \vdash s_2 : S \\ \Gamma;\Delta \vdash c_1 = c_2 : A;S & \Rightarrow & \Gamma;\Delta \vdash c_1 : A;S \text{ and } \Gamma;\Delta \vdash c_2 : A;S \end{array}$$

Proof For the state calculus we prove this by induction on the derivation of the judgement $\Delta \vdash s_1 = s_2 : S$ using lemma 6.2.1. For the value and command calculi, we prove this by mutual induction over the derivations of $\Gamma \vdash e_1 = e_2 : A$ and $\Gamma;\Delta \vdash c_1 = c_2 : A;S$. The induction hypothesis and the first part of this proposition are used for Eq-C-V-S. Lemma 6.2.3 is used for the β-rules and Lemma 6.3.2 for the commuting conversion rules. $\square$

6.3 Categorical Models

We will interpret Typed Command Systems in Typed Command Categories. Assume a Typed Command Category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$. We fix some notation. The category $\mathcal{C}$ has chosen finite products $(\times, \langle,\rangle, \pi_i, 1, !)$; the category $\mathcal{S}$ has symmetric monoidal structure $(\otimes, I, \alpha, \lambda, \rho, \sigma)$; and the category $\mathcal{K}$ has premonoidal structures with respect to $\mathcal{C}$ and $\mathcal{S}$: $(\circledcirc_{\mathcal{C}}, \circledcirc_{\mathcal{C}})$ and $(\circledcirc_{\mathcal{S}}, \circledcirc_{\mathcal{S}})$. The closed structure is given by a functor $- \to - : \mathcal{K}^{\mathsf{op}} \times \mathcal{K} \to \mathcal{C}$, an isomorphism of homsets $\Lambda^{\to}$ and counit $\mathrm{ev}_{\to}$. The $\mathcal{K}$-closed structure is given by a functor $(A, S) \multimap - : \mathcal{K} \to \mathcal{K}$, an isomorphism of homsets $\Lambda^{\multimap}$ and counit $\mathrm{ev}_{\multimap}$.

An *interpretation* of a Typed Command System $(\mathcal{T}_V, \mathcal{T}_S, \Phi_V, \Phi_S, \Phi_C)$ consists of five functions:

$$\mathcal{I}_V : \mathcal{T}_V \to \mathsf{Ob}\mathcal{C} \qquad \mathcal{I}_S : \mathcal{T}_S \to \mathsf{Ob}\mathcal{S} \qquad \mathcal{I}_{A,B} : \Phi_V(A, B) \to \mathcal{C}([\![A]\!], [\![B]\!])$$

$$\mathcal{I}_{S_1,S_2} : \Phi_S(S_1, S_2) \to \mathcal{S}([\![S_1]\!], [\![S_1]\!])$$

$$\mathcal{I}_{(A;S_1),(B;S_2)} : \Phi_C((A; S_1), (B; S_2)) \to \mathcal{K}(([\![A]\!], [\![S_1]\!]), ([\![B]\!]S_2]\!]))$$

where the map $[\![\cdot]\!]$ of value types to objects of $\mathcal{C}$ is defined as:

$$[\![1]\!] = 1 \qquad [\![X]\!] = \mathcal{I}_V(X) \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![(A; S_1) \to (B; S_2)]\!] = ([\![A]\!], [\![S_1]\!]) \to ([\![B]\!], [\![S_2]\!])$$

$$[\![(A; S_1) \multimap_V (B; S_2)]\!] = ([\![A]\!], [\![S_1]\!]) \multimap_V ([\![A]\!], [\![S_2]\!])$$

where the object $([\![A]\!], [\![S_1]\!]) \multimap_V ([\![A]\!], [\![S_2]\!])$ is the $\mathcal{C}$-component of the $\mathcal{K}$ object $([\![A]\!], [\![S_1]\!]) \multimap ([\![A]\!], [\![S_2]\!])$. The map $[\![\cdot]\!]$ from types to objects of $\mathcal{S}$ is defined as:

$$[\![I]\!] = I \qquad [\![X]\!] = \mathcal{I}_S(X) \qquad [\![S_1 \otimes S_2]\!] = [\![S_1]\!] \otimes [\![S_2]\!]$$

$$[\![(A; S_1) \multimap_S (B; S_2)]\!] = ([\![A]\!], [\![S_1]\!]) \multimap_S ([\![A]\!], [\![S_2]\!])$$

where the object $([\![A]\!], [\![S_1]\!]) \multimap_S ([\![A]\!], [\![S_2]\!])$ is the state component of the $\mathcal{K}$ object $([\![A]\!], [\![S_1]\!]) \multimap ([\![A]\!], [\![S_2]\!])$.

In the next two subsections we describe the interpretation of the type judgements of the three calculi in the categorical structure. The first subsection 6.3.1 describes the modelling of the state calculus, and the following subsection 6.3.2

$$\frac{}{[\![x : S \vdash x : S]\!] = id_{[\![S]\!]}} \qquad \frac{[\![\Delta_1 \vdash s_1 : S_1]\!] = s_1 \qquad [\![\Delta_2 \vdash s_2 : S_2]\!] = s_2}{[\![\Delta_1 \bowtie \Delta_2 \vdash (s_1, s_2) : S_1 \otimes S_2]\!] = s_1 \otimes s_2}$$

$$\frac{[\![\Delta_1 \vdash s_1 : S_1 \otimes S_2]\!] = s_1 \qquad [\![\Delta_2, z_1 : S_1, z_2 : S_2 \vdash s_2 : S_3]\!] = s_2}{[\![\Delta_1 \bowtie \Delta_2 \vdash \text{let } (z_1, z_2) = s_1 \text{ in } s_2 : S_3]\!] = [\![\Delta_2]\!](s_1); s_2}$$

$$\frac{}{[\![I \vdash \star_I : I]\!] = id_{[\![I]\!]}} \qquad \frac{[\![\Delta_1 \vdash s_1 : I]\!] = s_1 \qquad [\![\Delta_2 \vdash s_2 : S]\!] = s_2}{[\![\Delta_1 \bowtie \Delta_2 \vdash \text{let } \star_I = s_1 \text{ in } s_2 : S]\!] = [\![\Delta_2]\!](s_1); s_2}$$

$$\frac{[\![\Delta \vdash s : S_1]\!] = s \qquad (f : S_1 \longrightarrow S_2) \in \Phi_S}{[\![\Delta \vdash fs : S_2]\!] = s; \mathcal{I}(f)}$$

Figure 6.8: Interpretation of State Judgements

describes the modelling of the inter-dependent value and command calculi. At the end of each subsection we prove that the interpretation is coherent with respect to the typing judgements.

6.3.1 State Calculus Interpretation

Contexts are interpreted as objects in $\mathcal{S}$:

$$[\![I]\!] = I \qquad\qquad [\![\Delta, z : S]\!] = [\![\Delta]\!] \otimes [\![S]\!]$$

We interpret the merge operator $\Delta_1 \bowtie \Delta_2$ using the canonical structural maps of the symmetric monoidal structure on $\mathcal{S}$. Since this structure is known to be coherent ([Mac98] Theorem §XI.1.1.), and the merge operator does not affect the terms themselves (i.e. it does not rename variables) we shall ignore the effect of the structure maps and assume that we can always rewrite an interpretation of a context to the correct form.

Judgements $\Gamma \vdash s : S$ are interpreted as arrows $[\![\Gamma]\!] \to [\![S]\!]$ in $\mathcal{S}$. The interpretation is defined by induction on the derivation tree in Figure 6.8. The

interpretation is given the coherence of the structure maps of the monoidal structure.

6.3.2 Value and Command Calculi Interpretation

Value contexts are interpreted as objects of $\mathcal{C}$:

$$[\![\epsilon]\!] = 1 \qquad\qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

To express the interpretation of the substitution rules (Lemma 6.2.3) below, in Lemma 6.3.1, we will need a way to interpret the tails of value contexts as functors. We do this by the following interpretation:

$$[\![-, x : A]\!] = - \times [\![A]\!] \qquad\qquad [\![-, \Gamma, x : A]\!] = [\![-, \Gamma]\!] \times [\![A]\!]$$

Note that $([\![\Gamma]\!])[\![\Gamma']\!] = [\![\Gamma, \Gamma']\!]$ on objects.

Judgements $\Gamma \vdash e : A$ are interpreted as arrows $[\![\Gamma]\!] \to [\![A]\!]$ in $\mathcal{C}$ by induction over their derivation, using the rules in Figure 6.9. Command calculus judgements $\Gamma; \Delta \vdash c : A; S$ are interpreted as arrows $([\![\Gamma]\!], [\![\Delta]\!]) \to ([\![A]\!], [\![S]\!])$ in $\mathcal{K}$ by the rules in Figure 6.10. Again we assume that the interpretation of state contexts can always be rewritten to be in the appropriate form. The interpretation is coherent by the coherence of the symmetric monoidal structure of $\mathcal{S}$.

6.3.3 Typed Command Models

Given a Typed Command Theory $(\mathcal{T}_V, \mathcal{T}_S, \Phi_V, \Phi_S, \Phi_C, \Sigma_V, \Sigma_S, \Sigma_C)$, a *model* of this theory is an interpretation of the included Typed Command System with the added condition that for each axiom in Σ_V, Σ_S and Σ_C, the interpretations of both sides of the axiom are equal. This subsection is devoted to proving that this class of models is sound and complete. Firstly, we prove that the substitution rules from Lemmas 6.2.1 and 6.2.3 have the required interpretation.

$$\frac{x : A \in \Gamma}{[\![\Gamma \vdash x : A]\!] = \pi_i} \qquad \frac{}{[\![\Gamma \vdash \star : 1]\!] = !_{[\![\Gamma]\!]}}$$

$$\frac{[\![\Gamma \vdash e_1 : A_1]\!] = e_1 \qquad [\![\Gamma \vdash e_2 : A_2]\!] = e_2}{[\![\Gamma \vdash (e_1, e_2) : A_1 \times A_2]\!] = \langle e_1, e_2 \rangle} \qquad \frac{[\![\Gamma \vdash e : A_1 \times A_2]\!] = e}{[\![\Gamma \vdash \pi_i(e) : A_i]\!] = e; \pi_i}$$

$$\frac{[\![\Gamma \vdash e : A]\!] = e}{[\![\Gamma \vdash fe : B]\!] = e; \mathcal{I}_{A,B}(f)}$$

$$\frac{[\![\Gamma, x : A; z : S_1 \vdash c : B; S_2]\!] = c}{[\![\Gamma \vdash \lambda^{\rightarrow}(x; z) : (A; S_1).c : (A; S_1) \rightarrow (B; S_2)]\!] = \Lambda^{\rightarrow}(c)}$$

Figure 6.9: Interpretation of the Value Calculus

Lemma 6.3.1 The admissible rules of substitution are interpreted as:

$$\frac{[\![\Delta_1 \vdash s_1 : S_1]\!] = s_1 \qquad [\![\Delta_2, z : S_1 \vdash s_2 : S_2]\!] = s_2}{[\![\Delta_2 \bowtie \Delta_1 \vdash s_2[s_1/z] : S_3]\!] = [\![\Delta_2]\!] \otimes s_1; s_2}$$

$$\frac{[\![\Gamma \vdash e_1 : A]\!] = e_1 \qquad [\![\Gamma, x : A, \Gamma' \vdash e_2 : B]\!] = e_2}{[\![\Gamma, \Gamma' \vdash e_2[e_1/x] : B]\!] = (\langle [\![\Gamma]\!], e_1 \rangle)[\![\Gamma']\!]; e_2}$$

$$\frac{[\![\Gamma \vdash e_{\cdot} A]\!] = e \qquad [\![\Delta_1 \vdash s : S_1]\!] = s \qquad [\![\Gamma, x : A, \Gamma'; \Delta_2(z : S_1) \vdash c : B; S_2]\!] = c}{[\![\Gamma; \Delta_2 \bowtie \Delta_1 \vdash c[e/x, s/z] : B; S_2]\!] = J((\langle [\![\Gamma]\!], e \rangle)[\![\Gamma']\!], [\![\Delta_2]\!] \otimes s); c}$$

Proof For the state calculus substitution rule, this is by induction on the derivation of $\Delta_1, z : S_1 \vdash s_2 : S_2$. For the value and command substitutions, we

$$\frac{[\![\Gamma \vdash e : A]\!] = e \qquad [\![\Delta \vdash s : S]\!] = s}{[\![\Gamma; \Delta \vdash (e; s) : A; S]\!] = J(e, s)}$$

$$\frac{[\![\Gamma; \Delta_1 \vdash c_1 : A; S_1]\!] = c_1 \qquad [\![\Gamma, x : A; \Delta_2, z : S_1 \vdash c_2 : B; S_2]\!] = c_2}{[\![\Gamma; \Delta_1 \bowtie \Delta_2 \vdash \text{let } (x; z) \Leftarrow c_1 \text{ in } c_2 : B; S_2]\!] = J(dup, [\![\Delta_2 \bowtie \Delta_1]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} c_1); c_2}$$

$$\frac{[\![\Gamma; \Delta_1 \vdash c_1 : A; S_1 \otimes S_2]\!] = c_1 \qquad [\![\Gamma, x : A; \Delta_2, z_1 : S_1, z_2 : S_2 \vdash c_2 : B; S_3]\!] = c_2}{\begin{array}{c}[\![\Gamma; \Delta_2 \bowtie \Delta_1 \vdash \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2 : B; S_2]\!] \\ = J(dup, [\![\Delta_2 \bowtie \Delta_1]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} c_1); c_2\end{array}}$$

$$\frac{[\![\Gamma; \Delta_1 \vdash c_1 : A; I]\!] = c_1 \qquad [\![\Gamma, x : A; \Delta_2 \vdash c_2 : B; S_3]\!] = c_2}{\begin{array}{c}[\![\Gamma; \Delta_2 \bowtie \Delta_1 \vdash \text{let } (x; \star_I) \Leftarrow c_1 \text{ in } c_2 : B; S_2]\!] \\ = J(dup, [\![\Delta_2 \bowtie \Delta_1]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} c_1); c_2\end{array}}$$

$$\frac{[\![\Gamma \vdash f : (A; S_1) \to (B; S_2)]\!] = f \qquad [\![\Gamma; \Delta \vdash a : A; S_1]\!] = a}{[\![\Gamma; \Delta \vdash f @_{\to} a : B; S_2]\!] = J(dup, [\![\Delta]\!]); f \circledcirc a; \text{ev}}$$

$$\frac{[\![\Gamma, x : A; \Delta, z : S_1 \vdash c : B; S_2]\!] = c}{[\![\Gamma; \Delta \vdash \lambda^{\multimap}(x; z) : (A; S_1).c : (A; S_1) \multimap_V (B; S_2); (A; S_1) \multimap_S (B; S_2)]\!] = \Lambda^{\multimap}(c)}$$

$$\frac{\begin{array}{c}[\![\Gamma; \Delta_1 \vdash c_1 : (A; S_1) \multimap_V (B; S_2); (A; S_1) \multimap_S (B; S_2)]\!] = c_1 \\ [\![\Gamma; \Delta_2 \vdash c_2 : A; S_1]\!] = c_2\end{array}}{\begin{array}{c}[\![\Gamma; \Delta_1 \bowtie \Delta_2 \vdash c_1 @_{\multimap} c_2 : B; S_2]\!] = \\ J(dup, [\![\Delta_1 \bowtie \Delta_2]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} c_2); \\ (c_1 \circledcirc_{\mathcal{S}} [\![S_1]\!]) \circledcirc_{\mathcal{C}} [\![A]\!]; \text{ev}_{\multimap}\end{array}}$$

$$\frac{[\![\Gamma; \Delta \vdash c : A; S_1]\!] = c}{[\![\Gamma; \Delta \vdash pc : B; S_2]\!] = c; \mathcal{I}_{(A;S_1),(B;S_2)}(p)}$$

Figure 6.10: Interpretation of the Command Calculus

first prove the following interpretations of the two value-only substitution rules:

$$\frac{[\![\Gamma \vdash e_1 : A]\!] = e_1 \qquad [\![\Gamma, x : A, \Gamma' \vdash e_2 : B]\!] = e_2}{[\![\Gamma, \Gamma' \vdash e_2[e_1/x] : B]\!] = (\langle [\![\Gamma]\!], e_1 \rangle)[\![\Gamma']\!]; e_2}$$

$$\frac{[\![\Gamma \vdash e : A]\!] = e \qquad [\![\Gamma, x : A, \Gamma'; \Delta \vdash c : B; S]\!] = c}{[\![\Gamma, \Gamma'; \Delta \vdash c[e/x] : B; S]\!] = J((\langle [\![\Gamma]\!], e_1 \rangle)[\![\Gamma']\!], [\![\Delta]\!]); c}$$

We prove these by mutual induction on the derivations of $\Gamma, x : A, \Gamma' \vdash e_2 : B$ and $\Gamma, x : A, \Gamma'; \Delta \vdash c : B; S$. We prove the second rule is admissible by induction over the derivation of $\Gamma, x : A, \Gamma'; \Delta, z : S_1 \vdash c : B; S_2$. This requires the use of both value-only substitution interpretations and the state substitution rule. □

The next lemma ensures that the commuting conversion rules of the command calculus are typable and sound. This Lemma is also used to prove Proposition 6.2.4. We omit its proof which is by induction on the structure of the term contexts $C[-]$.

Lemma 6.3.2 1. Given derivations of $\Gamma, x : A; \Delta_2, z : S_1 \vdash C[s_2] : B; S_2$ and $\Gamma; \Delta_1 \vdash c_1 : A; S_1$ such that $C[-]$ does not bind or contain z, then there is a derivation of $\Gamma; \Delta_2 \bowtie \Delta_1 \vdash C[\text{let } (x; z) = c_1 \text{ in } c_2] : B; S_2$ with interpretation equal to $J(dup, [\![\Delta_2 \bowtie \Delta_1]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} [\![c_1]\!]); [\![c_2]\!]$.

2. Given derivations of $\Gamma, x : A; \Delta_2, z_1 : S_1, z_2 : S_2 \vdash C[s_2] : B; S_3$ and $\Gamma; \Delta_1 \vdash c_1 : A; S_1 \otimes S_2$ such that $C[-]$ does not bind or contain z_1, z_2, then there is a derivation of $\Gamma; \Delta_2 \bowtie \Delta_1 \vdash C[\text{let } (x; z_1, z_2) = c_1 \text{ in } c_2] : B; S_2$ with interpretation equal to $J(dup, [\![\Delta_2 \bowtie \Delta_2]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} [\![c_1]\!]); [\![c_2]\!]$.

3. Given derivations of $\Gamma, x : A; \Delta_2 \vdash C[s_2] : B; S_2$ and $\Gamma; \Delta_1 \vdash c_1 : A; I$ such that $C[-]$ does not bind or contain z, then there is a derivation of $\Gamma; \Delta_2 \bowtie \Delta_1 \vdash C[\text{let } (x; \star_I) = c_1 \text{ in } c_2] : B; S_2$ with interpretation equal to $J(dup, [\![\Delta_2 \bowtie \Delta_1]\!]); [\![\Gamma]\!] \circledcirc_{\mathcal{C}} ([\![\Delta_2]\!] \circledcirc_{\mathcal{S}} [\![c_1]\!]); [\![c_2]\!]$.

With these two lemmas we can prove the soundness of Typed Command Theories with respect to Typed Command Models.

Theorem 6.3.3 Given a model $\mathbb{M}$ of a Typed Command Theory $\mathbb{T}$. The following three implications hold:

$$\begin{aligned}
\Gamma \vdash e_1 = e_2 : A &\Rightarrow [\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!] \\
\Delta \vdash s_1 = s_2 : S &\Rightarrow [\![\Delta \vdash s_1 : S]\!] = [\![\Delta \vdash s_2 : S]\!] \\
\Gamma; \Delta \vdash c_1 = c_2 : A; S &\Rightarrow [\![\Gamma; \Delta \vdash c_1 : A; S]\!] = [\![\Gamma; \Delta \vdash c_2 : A; S]\!]
\end{aligned}$$

Proof For the state calculus judgement we prove this by induction over the derivation of $\Delta \vdash s_1 = s_2 : S$. Use Lemma 6.3.1 for Eq-S-$\otimes\beta$. For the value and command calculi judgements we prove this by mutual induction over the derivations of $\Gamma \vdash e_1 = e_2 : A$ and $\Gamma; \Delta \vdash e_1 = e_2 : A; S$. The induction hypothesis and the state calculus property are used for the Eq-C-V-S case. Lemma 6.3.1 is used for the β rules. Lemma 6.3.2 is used for the commuting conversion rules. □

We now prove that the Typed Command Models are complete, by showing that a model can be constructed from the type judgements of a Typed Command Theory. First we construct a Typed Command Category from a given Typed Command Theory, then we show that there is a model in this category obeying a property connecting the interpretation of judgements and their counterparts as arrows of the category. This will prove completeness.

Proposition 6.3.4 (Term Category) Let $\mathbb{T}$ be a Typed Command Theory $(\mathcal{T}_V, \mathcal{T}_S, \Phi_V, \Phi_S, \Phi_C, \Sigma_V, \Sigma_S, \Sigma_C)$. This definition defines a category $\mathcal{C}_\mathbb{T}$:

Objects Types generated from $\mathcal{T}_V$;

Arrows $A \to B$ Equivalence classes of terms $[x : A \vdash e : B]$.

Such that $\mathcal{C}_\mathbb{T}$ has finite products. This definition defines a category $\mathcal{S}_\mathbb{T}$ with symmetric monoidal structure:

Objects Types generated by $\mathcal{T}_S$;

Arrows $S_1 \to S_2$ Equivalence classes of terms $[z : S_1 \vdash s : S_2]$.

Also, this definition defines a category $\mathcal{K}_\mathbb{T}$:

Objects Pairs of types generated by $\mathcal{T}_V$ and $\mathcal{T}_S$;

Arrows $(A, S_1) \to (B, S_2)$ Equivalence classes of terms $[x : A; z : S_1 \vdash c : b; S_2]$.

Such that there is an identity-on-objects functor $J_{\mathbb{T}} : \mathcal{C}_{\mathbb{T}} \times \mathcal{S}_{\mathbb{T}} \to \mathcal{K}_{\mathbb{T}}$ that is a Typed Command Category.

Proof The identities for $\mathcal{C}_{\mathbb{T}}$, $\mathcal{S}_{\mathbb{T}}$ and $\mathcal{K}_{\mathbb{T}}$ are given by $[x : A \vdash x : A]$, $[z : S \vdash z : S]$ and $[x : A; z : S \vdash (x; z) : A; S]$ respectively. Composition is defined for $\mathcal{C}_{\mathbb{T}}$ as:

$$[x : A \vdash e_1 : B]; [x : B \vdash e_2 : C] = [x : A \vdash e_2[e_1/x] : C]$$

This is associative and obeys the identity laws as a consequence of the properties of substitution. Composition is also defined for $\mathcal{S}_{\mathbb{T}}$ by substitution. Composition is defined for $\mathcal{K}_{\mathbb{T}}$ as:

$$\begin{aligned} & [x : A; z : S_1 \vdash c_1 : B; S_2]; [x : B; z : S_2 \vdash c_2 : C; S_3] \\ = \; & [x : A; z : S_1 \vdash \text{let } (x; z) \Leftarrow c_1 \text{ in } c_2 : C; S_3] \end{aligned}$$

This obeys the identity laws by the Eq-C-Let-β and Eq-C-Let-η rules and is associative by the Eq-C-Let-CC rule.

Finite products on $\mathcal{C}_{\mathbb{T}}$ can be defined by taking 1 as the terminal object and $A \times B$ as the product of A and B. Symmetric monoidal structure is defined on $\mathcal{S}_{\mathbb{T}}$ by taking I as the unit object and $S_1 \otimes S_2$ as the operation of the functor on objects. On arrows it is defined using the S-$\otimes$E and S-$\otimes$I rules. The proof that these have the correct properties is standard [Cro94, Bar96].

Define a functor $J_{\mathbb{T}} : \mathcal{C}_{\mathbb{T}} \times \mathcal{S}_{\mathbb{T}} \to \mathcal{K}_{\mathbb{T}}$ as identity on objects, and on arrows as $J([x : A \vdash e : B], [z : S_1 \vdash s : S_2]) = [x : A; z : S_1 \vdash (e; s) : B; S_2]$. This is clearly a functor by the definition of identities in $\mathcal{K}_{\mathbb{T}}$ and the rule Eq-C-Let-β.

Define $\ltimes_{\mathcal{C}}$ as:

$$\begin{aligned} & [x : A_1 \vdash e : A_2] \ltimes_{\mathcal{C}} [x : B_1; z : S_1 \vdash c : B_2; S_2] \\ = \; & [x : A_1 \times B_1; z : S_2 \vdash \text{let}(y; z) = c_1[\pi_1 x/x] \text{ in}((e[\pi_2 x/x], y); z') : A_2 \times B_2; S_2] \end{aligned}$$

The functor $\rtimes_{\mathcal{C}}$ is defined similarly. They can be seen to obey the required naturality constraints by writing out the induced expressions in full and using

the properties of substitution and the β-reduction and commuting conversion rules.

Define $\circledcirc_{\mathcal{S}}$ as (the types on the final line have been hidden to save space):

$$\begin{aligned} & [z : S_1 \vdash s : S_2] \circledcirc_{\mathcal{S}} [x : A; z : S_1' \vdash c : B; S_2'] \\ = \; & [\text{let } (x; z_1, z_2) = (x; z) \text{ in let } (y; z_2') = c[z_2/z] \text{ in } (y; (s[z_1/z], z_2'))] \end{aligned}$$

The functor $\circledcirc_{\mathcal{S}}$ is defined similarly. They can be seen to obey the required naturality constraints in a similar way to the $\mathcal{C}$ premonoidal structure. The two required adjunction properties are given by the appropriate function types and their rules. □

Define the *term interpretation* of $\mathbb{T}$ in $J_{\mathbb{T}} : \mathcal{C}_{\mathbb{T}} \times \mathcal{S}_{\mathbb{T}} \to \mathcal{K}_{\mathbb{T}}$ by interpreting each primitive type as itself, and each primitive operation as $[x : A \vdash fx : B]$, $[z : S_1 \vdash pz : S_2]$ and $[x : A; z : S_1 \vdash c(x; z) : B; S_2]$ for value, state and command primitives respectively.

To establish that this is a model and hence completeness we need a connection between terms and heir interpretation in the state model. To this end, define the following functions from value and state contexts to value and state types respectively:

$$\overline{\epsilon} = 1 \qquad\qquad \overline{\Gamma, x : A} = \overline{\Gamma} \times A$$

and

$$\overline{I} = I \qquad\qquad \overline{\Delta, z : S} = \overline{\Delta} \otimes S$$

Now, given a pair of value judgements $\Gamma' \vdash e_1 : \overline{\Gamma}$ and $\Gamma \vdash e_2 : A$, define a term by induction on the structure Γ:

$$\overline{\epsilon, e_1, e_2} = e_2 \qquad\qquad \overline{(\Gamma, x : A), e_1, e_2} = (\overline{\Gamma, \pi_1(e_1), e_2})[\pi_2(e_1)/x]$$

It is clear by induction on the context that $x : \overline{\Gamma} \vdash \overline{\Gamma, x, e} : A$ is always derivable. Given a state judgement $\Delta \vdash s : S$ define a new term by induction on the context:

$$\overline{I, z, s} = \text{let } \star_I = z \text{ in } s \qquad\qquad \overline{(\Delta, z_2 : S), z, s} = \text{let } (z_1, z_2) = z \text{ in } \overline{\Delta, z_1, s}$$

Again it is clear that $z : \overline{\Delta} \vdash \overline{\Delta, z, s} : S$ is derivable. Given a command judgement $\Gamma; \Delta \vdash c : A; S$, define:

$$\overline{I, z, c} = \text{let } (d; \star_I) \Leftarrow (\star_1; z) \text{ in } c$$

$$\overline{(\Delta, z_2 : S), z, c} = \text{let } (d; z_1, z_2) \Leftarrow (\star_1; z) \text{ in } \overline{\Delta, z_1, c}$$

by induction on the state context, and, for a value judgement $\Gamma' \vdash e : \overline{\Gamma}$:

$$\overline{\epsilon, e, c} = c \qquad\qquad \overline{(\Gamma, x : A), e, c} = \overline{\Gamma, \pi_1 e, c[\pi_2 e/x]}$$

by induction on the value context Γ. Again it is clear that $x : \overline{\Gamma}; z : \overline{\Delta} \vdash \overline{\Delta, z, \overline{\Gamma, x, c}} : A; S$ is derivable. We use these definitions to state the required property of the term model.

Lemma 6.3.5 The term interpretation above is a model of $\mathbb{T}$ such that:

$$\begin{aligned} [\![\Gamma \vdash e : A]\!] &= [x : \overline{\Gamma} \vdash \overline{\Gamma, x, e} : A] \\ [\![\Delta \vdash s : S]\!] &= [z : \overline{\Delta} \vdash \overline{\Delta, z, s} : S] \\ [\![\Gamma; \Delta \vdash c : A; S]\!] &= [x : \overline{\Gamma}; z : \overline{\Delta} \vdash \overline{\Delta, z, \overline{\Gamma, x, c}} : A; S] \end{aligned}$$

Proof We first prove the property of the interpretation by induction on the derivation of $\Delta \vdash s : S$ and mutual induction on the derivations of $\Gamma \vdash e : A$ and $\Gamma; \Delta \vdash c : A; S$. This then implies that the interpretation is a model. □

With this model, we can deduce completeness:

Theorem 6.3.6 Assume a theory $\mathbb{T}$. If $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$ for all models of $\mathbb{T}$ then $\Gamma \vdash e_1 = e_2 : A$ in $\mathbb{T}$. And if $[\![\Delta \vdash s_1 : S]\!] = [\![\Delta \vdash s_2 : S]\!]$ for all models of $\mathbb{T}$ then $\Delta \vdash s_1 = s_2 : S$ in $\mathbb{T}$. Also, if $[\![\Gamma; \Delta \vdash c_1 : A; S]\!] = [\![\Gamma; \Delta \vdash c_2 : A; S]\!]$ for all models of $\mathbb{T}$ then $\Gamma; \Delta \vdash c_1 = c_2 : A; S$ in $\mathbb{T}$.

Proof We prove the contrapositive. Assume $\Gamma \vdash e_1 \neq e_2 : A$, and so $x : \overline{\Gamma} \vdash \overline{\Gamma, x, e_1} \neq \overline{\Gamma, x, e_2} : A$. Hence $[\![\Gamma \vdash e_1 : A]\!] \neq [\![\Gamma \vdash e_2 : A]\!]$ in the term model, i.e. the interpretations are not equal in all models. The other completeness properties are proven in the same way. □

6.4 Comparison to Alias Types

We claim that the Typed Command Calculus is the simply typed essence of the Alias Types system developed by Smith, Walker and Morrisett [SWM00]. Alias Types can type assembly language programs that manipulate pointers. The basic method that Alias Types uses to maintain safety is by linearly controlling permissions on how the memory may be accessed.

By "simply typed essence" we mean that the Typed Command Calculus captures the features of Alias Types that contribute to its type soundness with respect to a semantics that performs in-place update that are expressible only in simple types, i.e. without the indexed types that Alias Types has.

We do not give a formal translation of the Typed Command Calculus into Alias Types for three reasons. Firstly, the Alias Types system types assembly language instructions rather than λ terms and so any translation would get caught up in matters to do with compilation rather than typing. Secondly, the Alias Types system has a complicated system of indexed types [XP99] which we only present informally here. Thirdly, Alias Types does not have a notion of function type that closes over pieces of state to match the Typed Command Calculus' $\multimap$ functions.

The Alias Types system has three judgements of the form:

$$\Delta \vdash C = C' \qquad \Delta; \Gamma \vdash v : \tau \qquad \Delta; C; \Gamma \vdash \iota$$

where Δ is an indexing context containing location and constraint variables; C and C' are store constraints describing the current state of the store; Γ is a value context; v is a value term; τ is a value type and ι is a command.

Store Constraints are built from constraint variables ϵ, location typings $\{l \mapsto \tau\}$ and joined constraints $C \oplus C'$. The first judgement is derivable when C and C' have the same variables and location typings, possibly in a different order. The second judgement is for deriving the well-typing of pure values that do not perform any side-effecting operations. The third judgement types a command ι, starting with a store satisfying constraint C and a value context Γ. It has no result type since Alias Types uses a continuation-passing-style approach.

We define an analogy between Alias Types and the Typed Command Calculus by matching store constraints with state contexts and types, value contexts and types with Typed Command Calculus value contexts and types, and commands with Typed Command Calculus commands.

Under this analogy, the first judgement corresponds to the judgements of the state calculus, the second to judgements of the value calculus and the third to judgements of the command calculus. We further elaborate this analogy by demonstrating simplified versions of two of the typing rules.

Alias Types has a primitive `free` operator, typed thus:

$$\frac{\Delta; \Gamma \vdash v : ptr(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \tau\} \qquad \Delta; C' \oplus \{\eta \mapsto junk\}; \Gamma \vdash \iota}{\Delta; C; \Gamma \vdash \texttt{free}\ \ v; \iota}$$

where $ptr(\eta)$ is a singleton type with the location η as its only variable. There are three important things to note about this definition. Firstly, but the use of an auxiliary store constraint C', the `free` operation acts locally, within the context defined by C'. Compare this to the "let" constructs of the Typed Command Calculus that allow a command to be executed in a larger context. The main point of the definitions of double parameterised Freyd category and monoidal parameterised monad in Section 5.3 was to allow the lifting of commands to larger state contexts. Alias Types builds this lifting directly into the typing rules.

Secondly, the use of type indexing and singleton types allows Alias Types to separation the notions of pointer and permission. The Typed Command Calculus cannot do this since there is no way to make sure that a value variable and state variable are related in any way. We discuss adding indexed types to the Typed Command Calculus in Section 9.2.

Thirdly, the continuation command ι in this rule is passed the altered constraint $C' \oplus \{\eta \mapsto junk\}$ and the same value context Γ. This sequence is matched in the Typed Command Calculus by the use of shared value contexts and separated state contexts in the C-Let rules.

Alias Types also has functions that exist only as values. The introduction rule for them is, in simplified form (we have omitted the possibility for recursion

and multiple arguments):

$$\frac{\Delta \vdash \forall[\Delta'; C]\tau \to 0 \qquad \Delta, \Delta'; C; \Gamma, x : \tau \vdash \iota}{\Delta; \Gamma \vdash \lambda[\Delta'; C; x : \tau].\iota : \forall[\Delta'; C]\tau \to 0}$$

The first premise ensures that the function type is well-formed in the indexing context. Ignoring the parameter Δ', functions in Alias Types are very similar to $\to$ functions in the Typed Command Calculus, both take a state component and value component to do their work, and both are treated as values since they do not close over pieces of the state.

Due to these similar concepts in both Alias Types and the Typed Command Calculus and their relevance to safe in-place update we are confident that we have identified a suitable simply typed theory that will form a basis for future investigation of typed languages with explicit memory management.

Chapter 7

Heap bounded state model

In this chapter we describe a Typed Command Category that demonstrates how the structure may be used to model separation and side-effects on a heap of fixed size. Since it is a Typed Command Category it will be suitable for interpreting the Typed Command Calculus of the previous chapter. This category will also be suitable for modelling the λ_{inplc} calculus of Chapter 8.

We will show how to interpret data structures that occupy heap space and give new typing rules for the Typed Command Calculus for introducing and eliminating them in a way which respects the fact that they reside on the heap.

7.1 The Category

The three categories in the model are as follows: $\mathcal{C}$ is the category **Set**; $\mathcal{S}$ is the functor category $[\mathbf{P}, \mathbf{Set}]$, where $\mathbf{P}$ is the category of natural numbers and permutations; and $\mathcal{K}((A, S_1), (B, S_2)) = [\mathbf{P}, \mathbf{Set}](A \times S_1 -, B \times S_2 -)$, where $A \times S_1 -$ denotes the functor $F(X) = A \times S_1 X$, likewise for $B \times S_2 -$. The identity-on-objects functor $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ maps $(f, \langle s_i \rangle_{i \in \mathsf{Ob}\mathbf{P}})$ to $\langle f \times s_i \rangle_{i \in \mathsf{Ob}\mathbf{P}}$.

Arrows of $\mathcal{K}$ map an input value and store and to an output value and store. This is a refinement of Example 5.3.3 since it now includes notions of separation, boundedness and non-duplicability by the functor category structure. These properties all stem from the choice of functors from $\mathbf{P}$ to represent state types.

The reason for choosing $\mathbf{P}$ is that it is the free symmetric monoidal category

generated from the one object, one arrow category. Thus it provides the minimum amount of structure needed to describe a separated heap. We think of an object of $[\mathbf{P}, \mathbf{Set}]$ as family of sets indexed by the amount of memory cells available. The symmetric monoidal structure on objects is given by addition of natural numbers. We will make use of Day's construction [Day70] on the functor category $[\mathbf{P}, \mathbf{Set}]$ for interpreting composite state types. This is given by the following coend formula (see Section 4.1 for the definition of coends):

$$(S_1 \otimes S_2)n = \int^{n_1,n_2} S_1 n_1 \times S_2 n_2 \times \mathbf{P}(n_1 + n_2, n)$$

The empty heap is modelled by the functor $I0 = \{*\}$ and $In = \emptyset$ when $n \neq 0$.

By the definition of tensor product we can see that it models separated states. Given two state types S_1 and S_2, this definition ensures that the combined heap is large enough to contain both of them without sharing any memory cells. Also, in general, there are no arrows of the form $S \to S \otimes S$ in $\mathcal{S}$ by this construction, so we may not duplicate state. Neither are there any arrows of the form $S \to I$, so we may not discard state.

Assuming the functors used to interpret state types are sensibly defined with real computer memory in mind, the space occupied by the output of the computation is exactly the same as the space occupied by the input. This is in keeping with the heap-bounded property of LFPL [Hof00].

We define symmetric premonoidal structure on J with respect to both $\mathcal{C}$ and $\mathcal{S}$. Define $f \circledcirc_{\mathcal{C}} c$, where $f : A_1 \to A_2$ and $c : (B_1, S_1) \to (B_2, S_2)$, as:

$$(A_1 \times B_1) \times S_1 - \xrightarrow{\alpha} A_1 \times (B_1 \times S_1 -) \xrightarrow{f \times c} A_2 \times (B_2 \times S_2 -) \xrightarrow{\alpha^{-1}} (A_2 \times B_2) \times S_2 -$$

The definition $\circledcirc_{\mathcal{C}}$ is similar.

Define $s \circledcirc_{\mathcal{S}} c$, where $s : S_1 \to S_2$ and $c : (A_1, S_1') \to (A_2, S_2')$, as the composite (using Day's notation for co-ends involving $\times$):

$$\begin{array}{rl} & A_1 \times (S_1 n_1 \underline{\times} (S_1' n_2 \underline{\times} \mathbf{P}(n_1 + n_2, n))) \\ \cong & S_1 n_1 \underline{\times} ((A_1 \times S_1' n_2) \underline{\times} \mathbf{P}(n_1 + n_2, n)) \\ \xrightarrow{s \underline{\times} (c \times id)} & S_2 n_1 \underline{\times} ((A_2 \times S_2' n_2) \underline{\times} \mathbf{P}(n_1 + n_2, n)) \\ \cong & A_2 \times (S_2 n_1 \underline{\times} (S_2' n_2 \underline{\times} \mathbf{P}(n_1 + n_2, n))) \end{array}$$

The definition of $\circledcirc_{\mathcal{S}}$ is similar.

This construction is also closed in both senses defined in Chapter 5. Define:

$$(A, S_1) \rightarrow (B, S_2) = [\mathbf{P}, \mathbf{Set}](A \times S_1 -, B \times S_2 -)$$

For $\mathcal{K}$-closure, define the functor to be:

$$(A, S_1) \multimap (B, S_2) = (1, \int_n A \times S_1 n \Rightarrow B \times S_2(- + n))$$

where $\Rightarrow$ is the exponential functor in **Set**.

Theorem 7.1.1 With the definitions above, $J : \mathbf{Set} \times [\mathbf{P}, \mathbf{Set}] \rightarrow \mathcal{K}$ is a Typed Command Category.

Proof The functor $\circledcirc_{\mathcal{C}}$ respects the monoidal structure of **Set** on arrows:

$$f \circledcirc_{\mathcal{C}} J(g, s) = \alpha; f \times (g \times s); \alpha^{-1} = (f \times g) \times s; \alpha; \alpha^{-1} = (f \times g) \times s = J(f \times g, s)$$

Similarly for $\circledcirc_{\mathcal{C}}$. Naturality for the structure transformations is also easy to check. For example, the naturality in the third argument for $J(\alpha, id)$:

$$\begin{aligned} J(\alpha, id); A \circledcirc (B \circledcirc c) &= \alpha \times id; \alpha; id \times (\alpha; id \times c; \alpha^{-1}); \alpha^{-1} \\ &= \alpha; \alpha; id \times (id \times c); id \times \alpha^{-1}; \alpha^{-1} \\ &= \alpha; id \times c; \alpha; id \times \alpha^{-1}; \alpha^{-1} \\ &= \alpha; id \times c; \alpha^{-1}; \alpha \times id \\ &= (A \otimes B) \circledcirc c; J(\alpha, id) \end{aligned}$$

where the inner steps follow by coherence, naturality and coherence respectively. The other naturality properties are similar.

That the functors $\circledcirc_{\mathcal{S}}$ and $\circledcirc_{\mathcal{S}}$ preserve $\mathcal{S}$'s monoidal structure on arrows follows directly from the definition, coherence of the monoidal structure of $\times$ on **Set** and naturality. The naturality of the structure transformations under J holds for similar reasons to the naturality of the $\mathcal{C}$ transformations: by coherence and naturality.

The fact that $(A, S_1) \rightarrow -$ is right adjoint to $- \circledcirc_{\mathcal{C}} (A, S_1)$ is immediate from the cartesian closure of $[\mathbf{P}, \mathbf{Set}]$.

For $\mathcal{K}$-closure, the isomorphism of homsets required for the adjunction is given by:

$$\begin{aligned}
& [\mathbf{P}, \mathbf{Set}](A \times B \times \int^{n_1,n_2} S_1 n_1 \times S_2 n_2 \times \mathbf{P}(n_1 + n_2, -), C \times S_3 -) \\
\cong & \int_n (A \times B \times \int^{n_1,n_2} S_1 n_1 \times S_2 n_2 \times \mathbf{P}(n_1 + n_2, n) \Rightarrow C \times S_3 n) \\
\cong & \int_{n,n_1,n_2} (A \times B \times S_1 n_1 \times S_2 n_2 \times \mathbf{P}(n_1 + n_2, n) \Rightarrow C \times S_3 n) \\
\cong & \int_{n,n_1,n_2} (A \times S_1 n_1 \Rightarrow (B \times S_2 n_2 \times \mathbf{P}(n_1 + n_2, n) \Rightarrow C \times S_3 n)) \\
\cong & \int_{n_1,n_2} (A \times S_1 n_1 \Rightarrow (B \times S_2 n_2 \Rightarrow (\int_n (\mathbf{P}(n_1 + n_2, n) \Rightarrow C \times S_3 n)))) \\
\cong & \int_{n_1,n_2} (A \times S_1 n_1 \Rightarrow (B \times S_2 n_2 \Rightarrow C \times S_3(n_1 + n_2))) \\
\cong & \int_{n_1} (A \times S_1 n_1 \Rightarrow \int_{n_2} (B \times S_2 n_2 \Rightarrow C \times S_3(n_1 + n_2))) \\
\cong & [\mathbf{P}, \mathbf{Set}](A \times S_1 -, \int_{n_2} (B \times S_2 n_2 \Rightarrow C \times S_3(- + n_2)))
\end{aligned}$$

where the isomorphisms are by: the presentation of sets of natural transformations by an end; preservation of colimits; currying; preservation of limits; Yoneda; preservation of limits; and the presentation of sets of natural transformations as ends. The sequence of isomorphisms here is an adapted case of the sequence needed to prove that Day's construction is closed. □

This Typed Command Category also enjoys the extra properties identified for models of state in Section 5.4:

Proposition 7.1.2 The Typed Command Category $J : \mathbf{Set} \times [\mathbf{P}, \mathbf{Set}] \to \mathcal{K}$ is commutative, and the functor $J(-, I)$ is full and faithful.

Proof For commutativity, it must be the case that all arrows of $\mathcal{K}$ are central. Writing out the definitions of $(c_1 \circledcirc_{\mathcal{C}} B) \circledcirc_{\mathcal{S}} S_2; S_1' \circledcirc_{\mathcal{S}} (A' \circledcirc_{\mathcal{C}} c_2)$ and $S_1 \circledcirc_{\mathcal{S}} (A \circledcirc_{\mathcal{C}} c_2); (c_1 \circledcirc_{\mathcal{C}} B') \circledcirc_{\mathcal{S}} S_2'$ shows that they are both equal to:

$$(A \times B) \times (S_1 n_1 \underline{\times} (S_2 n_2 \underline{\times} \mathbf{P}(n_1 + n_2, n)))$$

$$\begin{array}{rl} \cong & (A \times S_1 n_1) \underline{\times} ((B \times S_2 n_2) \underline{\times} \mathbf{P}(n_1 + n_2, n)) \\ \xrightarrow{c_1 \underline{\times} (c_2 \underline{\times} \mathbf{P}(n_1+n_2,n))} & (A' \times S'_1 n_1) \underline{\times} ((B' \times S'_2 n_2) \underline{\times} \mathbf{P}(n_1 + n_2, n)) \\ \cong & (A' \times B') \times (S'_1 n_1 \underline{\times} (S'_2 n_2 \underline{\times} \mathbf{P}(n_1 + n_2, n))) \end{array}$$

thus this category is commutative. The functor $J(-, I)$ is full and faithful because given a natural transformation $f_n : A \times In \to B \times In$, it is only non-trivial when $n = 0$, and then it is equivalent to an arrow $A \to B$ since $I0 = \{*\}$. □

7.2 Boxed Data

We now demonstrate the interpretation of some simple datatypes that reside on the heap. The first will be "boxed" versions of value datatypes. These will model memory cells on the heap that contain a single value of the specified type. Thus the state type $[A]$ will represent memory cells that contain values of value type A. We will also have a special type $\Diamond$ which is isomorphic to $[1]$, representing unused memory cells. The use of a diamond for unused memory cells is taken from Hofmann's LFPL [Hof00].

For each boxed type there is a pair of primitive commands in the Typed Command Calculus for storing a value and destructively retrieving a value:

$$\texttt{store}_A : (A, \Diamond) \to (1, [A]) \qquad \texttt{retrieve}_A : (1, [A]) \to (A, \Diamond)$$

The `store` command takes the value to be stored and a diamond representing the memory location in which it is to be stored and returns a pointer to the memory cell. In a real implementation this pointer would have exactly the same value, but we return it here because its type has changed. The `retrieve` command performs the opposite operation, taking a pointer to a memory cell and splitting it into its value and abstract location.

There are two axiom schemes for this pair, stating that the two operations are inverse:

$$\texttt{store}_A(\texttt{retrieve}_A c) = c \qquad \texttt{retrieve}_A(\texttt{store}_A c) = c$$

The triviality of these axioms points to the fact that, with a functional interpretation, `store` and `retrieve` are both no-ops. But the point is that we can implement them using mutating operations on the store. It is the substructural typing of the Typed Command Calculus that allows such simple axioms.

Given some value type A with interpretation as a set $[\![A]\!]$, we interpret the boxed type $[A]$ as:

$$[\![[A]]\!]n = \begin{cases} [\![A]\!] & n = 1 \\ \emptyset & otherwise \end{cases}$$

A boxed value takes up exactly one cell of memory space. The diamond type also takes up one cell of memory space, but does not contain any useful information:

$$[\![\Diamond]\!]n = \begin{cases} \{*\} & n = 1 \\ \emptyset & otherwise \end{cases}$$

The interpretations of $\texttt{store}_A$ and $\texttt{retrieve}_A$ are simple:

$$store_A[1](a, *) = (*, a) \qquad\qquad retrieve_A[1](*, a) = (a, *)$$

we do not need to specify them at heap sizes other than 1 by the definitions of $[\![[A]]\!]$ and $[\![\Diamond]\!]$. These two operations are obviously inverse, and hence satisfy the axioms.

7.3 Singly-Linked Lists

A more complex data type is provided by singly linked lists that reside in the store. We will not represent the links in the semantics, since we have no notion of locations and hence no notion of pointers to do the linking. For each value type A, we assume a state type $L(A)$, representing lists in the store. There are two ways to introduce states containing lists:

$$\texttt{nil}_A : (1, I) \to (1, L(A)) \qquad\qquad \texttt{cons}_A : (A, \Diamond \otimes L(A)) \to (1, L(A))$$

The `nil` command works on an empty store (represented by I) since, operationally, the empty list will be represented by a null pointer. The `cons` command

takes an unused piece of the store, represented by the $\Diamond$ argument, and uses this to create a new cons cell containing the value and "pointing" to the rest of the list.

Lists are eliminated by means of an fold operator[1]:

$$\frac{\Gamma; \Delta_1 \vdash c_1 : B; S \qquad \Gamma, x : B, y : A; z_1 : \Diamond, z_2 : S \vdash c_2 : B; S \qquad \Gamma; \Delta_2 \vdash c_3 : (1, L(A))}{\Gamma; \Delta_1, \Delta_2 \vdash \mathrm{listfold}_{A,B,S}(c_1, (x, y; z_1, z_2)c_2, c_3) : B; S}$$

where the value variables x and y and the state variables z_1 and z_2 are bound in the command c_2.

The intended meaning is that listfold executes the command c_3 to obtain a list, it then executes the command c_1 and then iterates up the list, using a second command c_2 for each of the nodes. Hence, this is an in-place version of the standard foldr operation. At each node, the memory used by the node is made available for use by c_2 via its $\Diamond$ argument. Thus, listfold destroys the list, making it available for reuse as it traverses it. Note the use of a "loop invariant" S that is initialised by c_1 and preserved by c_2.

There are two axioms schemes for lists, for when the two possible pairs of introduction and elimination terms meet:

$$\mathrm{listfold}_{A,B,S}(c_1, (x, y; z_1, z_2)c_2, \mathtt{nil}_A(\star_1; \star_I)) = c_1$$

$$\begin{aligned} &\mathrm{listfold}_{A,B,S}(c_1, (x_1, x_2; z_1, z_2)c_2, \mathtt{cons}_A(x; (d, xs))) \\ &= \\ &\text{let } (a, s) = \mathrm{listfold}_{A,B,S}(c_1, (x, x_2; z_1, z_2)c_2, (\star_1; xs)) \\ &\text{in } c_2[a/x_1, x/x_2, s/z_1, d/z_2] \end{aligned}$$

Apart from the typing and the $\Diamond$ arguments, these axioms are very similar to what we would expect in a pure functional programming language with a "fold" operator.

[1]Note that, due to presence of function types in the Typed Command Calculus, we could present this new syntax as a primitive command. However, this would make the presentation of the example programs more unwieldy, even though we have to re-prove soundness and substitution.

Using these primitives we can write simple programs that exploit the use of in-place update. Here is a variant of the standard map function that updates the list in-place:

$$\begin{array}{l}\mathrm{map}_{A,B} = \lambda^{\rightarrow}(f; z) : ((A, I) \rightarrow (B, I); L(A)).\\ \mathrm{listfold}(\mathtt{nil}(\star_1; \star_I),\\ \quad (a : A, y : 1; r : L(B), d : \Diamond)\mathrm{let}(b; \star_I) = f@_{\rightarrow}(a, \star_I) \text{ in } \mathtt{cons}(b; (r, d)),\\ \quad (\star_1; l))\end{array}$$

The loop invariant in this application of listfold is of the type $L(B)$, the piece of state represented being the new list constructed in the memory of the old one.

Here is a short function that appends a list to another in place; the loop invariant is again the list that is being constructed from the old ones:

$$\begin{array}{l}\mathrm{append}_A = \lambda^{\rightarrow}(q; z) : (1; L(A) \otimes L(A)).\\ \text{let } (q; z_1, z_2) = (q; z) \text{ in}\\ \quad \mathrm{listfold}((\star_1; z_2), (x : A, y : 1; r : L(A), d : \Diamond)\mathtt{cons}(x; (r, d)), (\star_1; z_1))\end{array}$$

We now turn to the interpretation of the syntax for lists. We adapt the initial algebra semantics for inductive types to our situation. For more information on using initial algebras to interpret inductive types, see [Pit00]. At the level of Typed Command Categories, we interpret the `nil` and `cons` primitives by arrows in $\mathcal{K}$:

$$\mathit{nil}_{[\![A]\!]} : (1, I) \rightarrow (1, [\![L(A)]\!]) \qquad \mathit{cons}_{[\![A]\!]} : ([\![A]\!], [\![\Diamond]\!] \otimes [\![L(A)]\!]) \rightarrow (1, [\![L(A)]\!])$$

The fold operator is interpreted via a function of homsets, natural in X:

$$\mathit{listfold} : \mathcal{K}((X \times [\![A]\!] \times B, [\![\Diamond]\!] \otimes S), (B, S)) \rightarrow \mathcal{K}((X \times B, S \otimes [\![L(A)]\!]), (B, S))$$

With just this basic structure, we can now interpret the typing rule for listfold:

$$\frac{[\![\Gamma, x : B, y : A; z_1 : \Diamond, z_2 : S \vdash c_2 : B; S]\!] = c_2 \qquad [\![\Gamma; \Delta_2 \vdash c_3 : (1, L(A))]\!] = c_3}{\begin{array}{c}[\![\Gamma; \Delta_1, \Delta_2 \vdash \mathrm{listfold}_{A,B,S}(c_1, (x, y; z_1, z_2)c_2, c_3) : B; S]\!]\\ = (\langle id, id\rangle, id); ([\![\Gamma]\!], [\![\Delta_1]\!]) \circledcirc c_3; c_1 \circledcirc (1, [\![L(A)]\!]); \mathit{listfold}(c_2)\end{array}}$$

The, extended calculus, with this interpretation still has the substitution property of the unextended calculus (Lemmas 6.2.3 and 6.3.1):

Lemma 7.3.1 (Substitution) In the extended calculus, the following rules are admissible and have the given interpretation:

$$\frac{[\![\Gamma \vdash e_1 : A]\!] = e_1 \qquad [\![\Gamma, x : A, \Gamma' \vdash e_2 : B]\!] = e_2}{[\![\Gamma, \Gamma' \vdash e_2[e_1/x] : B]\!] = (\langle [\![\Gamma]\!], e_1 \rangle)[\![\Gamma']\!]; e_2}$$

$$\frac{\begin{array}{c}[\![\Gamma \vdash e : A]\!] = e \\ [\![\Delta_1 \vdash s : S_1]\!] = s \qquad [\![\Gamma, x : A, \Gamma'; \Delta_2(z : S_1) \vdash c : B; S_2]\!] = c\end{array}}{[\![\Gamma, \Gamma'; \Delta_2(\Delta_1) \vdash c[e/x, s/z] : B; S_2]\!] = ((\langle id, e \rangle)[\![\Gamma']\!], [\![\Delta_2]\!](s)); c}$$

Proof As before, we prove two value-only substitution rules admissible and with the correct interpretation by mutual induction on the derivations. The naturality of *listfold* in X is crucial. This is then used to prove the second substitution rule. □

For this interpretation to be a model, we must impose two requirements on the *nil* and *cons* arrows and the *listfold* operator, corresponding to the two axioms. The following two diagrams must commute:

$$\begin{array}{ccc}
(X \times B, S) & \xrightarrow{J(\rho^{-1}, \rho^{-1})} & (X \times B \times 1, S \otimes I) \\
\Big\downarrow{\scriptstyle J(\pi_2, S)} & & \Big\downarrow{\scriptstyle (X \times B, S) \oslash nil} \\
(B, S) & \xleftarrow{listfold(c)} & (X \times B, S \otimes [\![L(A)]\!])
\end{array}$$

$$\begin{array}{ccc}
(X \times B \times [\![A]\!], S \otimes [\![\Diamond]\!] \otimes [\![L(A)]\!]) & \xrightarrow{J(X \times B, S) \oslash cons} & (X \times B, S \otimes [\![L(A)]\!]) \\
\Big\downarrow{\scriptstyle J(\cong, \cong)} & & \\
(X \times [\![A]\!] \times X \times B, [\![\Diamond]\!] \otimes S \otimes [\![L(A)]\!]) & & \Big\downarrow{\scriptstyle listfold(c)} \\
\Big\downarrow{\scriptstyle (X \times [\![A]\!], [\![\Diamond]\!]) \oslash listfold(c)} & & \\
(X \times [\![A]\!] \times B, [\![\Diamond]\!] \otimes S) & \xrightarrow{\qquad c \qquad} & (B, S)
\end{array}$$

Soundness holds for the extended calculus, extending Theorem 6.3.3. It is proven by induction on the equational judgements:

Theorem 7.3.2 (Soundness) In the extended calculus, if $\Gamma \vdash e_1 = e_2 : A$ then $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ are derivable, and $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$. Likewise

if $\Gamma; \Delta \vdash c_1 = c_2 : A; S$ then $\Gamma; \Delta \vdash c_1 : A; S$ and $\Gamma; \Delta \vdash c_2 : A; S$ are derivable and $[\![\Gamma; \Delta \vdash c_1 : A; S]\!] = [\![\Gamma; \Delta \vdash c_2 : A; S]\!]$.

Moving back to our concrete model, we interpret the type of lists as the set of finite lists of elements of $[\![A]\!]$, the indexing by $\mathbf{P}$ determines the exact size of the list.

$$[\![L(A)]\!]n = \begin{cases} \{*\} & n = 0 \\ [\![L(A)]\!](n-1) \times [\![A]\!] & n \geq 1 \end{cases}$$

The definition of $[\![L(A)]\!](f : n \to n)$ is just the identity.

We need only define the component of the *nil* operation when $n = 0$, since $I(n) = \emptyset$ if $n \neq 0$:

$$nil_A[0](*, * \in I0) = (* \in 1, * \in L(A)0)$$

The resulting family is clearly natural in n. We define the *cons* operation by making use of the universal property of co-ends. The arrow *cons* is induced from the arrows:

$$\begin{aligned} cons^n_{n_1,n_2,A} &: [\![A]\!] \times [\![\Diamond]\!]n_1 \times [\![L(A)]\!]n_2 \times \mathbf{P}(n_1 + n_2, n) \to [\![L(A)]\!]n \\ cons^n_{n_1,n_2,A} &= (a, *, \langle a_1, ..., a_{n_2}\rangle, f) \mapsto \langle a_1, ..., a_{n_2}, a\rangle \end{aligned}$$

It is easy to check that this is dinatural in n_1 and n_2 and natural in n. Therefore, by Day's Lemma 2.2 [Day70], there is a transformation, natural in n:

$$cons^n_A : [\![A]\!] \times \int^{n_1,n_2} [\![\Diamond]\!]n_1 \times [\![L(A)]\!]n_2 \times \mathbf{P}(n_1 + n_2, n) \to [\![L(A)]\!]n$$

as required.

The *listfold* operator is also defined using the universal property. For any natural transformation $c_n : X \times [\![A]\!] \times B \times \int^{n_1,n_2} \Diamond n_1 \times Sn_2 \times \mathbf{P}(n_1 + n_2, n) \to B \times Sn$, there is a family of wedges:

$$c^n_{n_1,n_2} : X \times [\![A]\!] \times B \times \Diamond n_1 \times Sn_2 \times \mathbf{P}(n_1 + n_2, n) \to B \times Sn$$

which is dinatural in n_1 and n_2 and natural in n. Now define $listfold(c)^n_{n_1,n_2}$: $X \times B \times Sn_1 \times [\![L(A)]\!]n_2 \times \mathbf{P}(n_1+n_2, n) \to B \times Sn$ as:

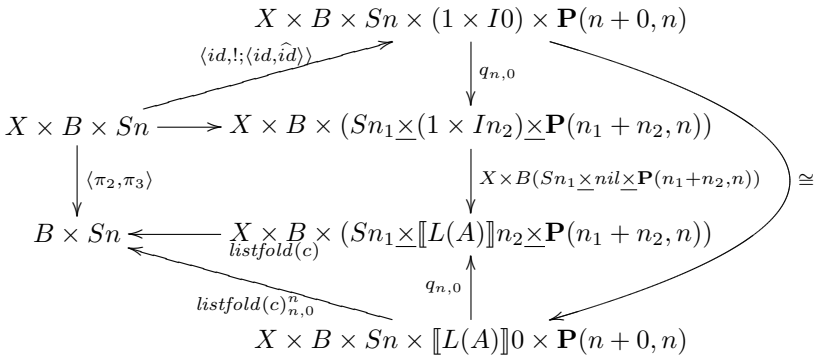
$$(x, b, s, \langle a_1, ..., a_{n_2}\rangle, f) \mapsto \left(\begin{array}{l} \text{let } (b_1, s_1) = c^{n_1+1}_{1,n_1}(x, a_1, b, *, s, id) \text{ in} \\ \quad ... \\ \text{let } (b_{n_2}, s_{n_2}) = c^{n_1+n_2}_{1,n_1+n_2-1}(x, a_{n_2}, b_{n_2-1}, *, s_{n_2-1}, id) \text{ in} \\ \quad (b_{n_2}, Sfs_{n_2}) \end{array} \right)$$

Note how, as each of the elements of the list is being processed, the memory space allocated to the list, n_2, is being transferred to the values of $S(n_1+i)$. This family of arrows is clearly natural in n and is dinatural in n_1 and n_2 by the naturality and dinaturality properties of the family $c^n_{n_1,n_2}$. Therefore, by Day's Lemma 2.2, we have a natural transformation:

$$listfold(c) : X \times B \times \int^{n_1,n_2} Sn_1 \times [\![L(A)]\!]n_2 \times \mathbf{P}(n_1+n_2, n) \to B \times Sn$$

as required.

It remains to verify the required diagrams for these operations. For the diagram involving *nil*, consider the following diagram, where the inner square is the diagram we require to commute:



The top and bottom triangles and the right-hand quadrilateral all commute by Lemma 4.1.11 and the definition of *nil*. The outer perimeter commutes by inspection of the definition of $listfold(c)^n_{n_1,n_2}$. Hence the inner diagram commutes, as required. The required diagram for *cons* may also be seen to hold by a similar, but larger diagram, relying on the universal property and the definition of $listfold(c)^n_{n_1,n_2}$.

One could also provide semantics for other (non-circular) linked data structures in this manner.

Chapter 8

An In-place Update Calculus

In the previous chapter we presented a calculus directly related to Typed Command Categories. In this chapter we present a calculus, λ_{inplc}, that is a reformulation in our framework of the linear type system with non-linear types proposed by Wadler [Wad90, Wad91], without the let! construct. A similar system has been published by Hofmann [Hof00] for heap-bounded in-place update. More advanced systems based on it include Walker and Watkin's system for combining linear typing and regions [WW01] and Morrisett, Ahmed and Fluet's Linear Language with Locations for pointer programs [MAF05].

We will show that commutative Typed Command Categories that obey the fullness property of Section 5.4 coherently and soundly model the calculus. We also give a direct categorical semantics for the calculus, of which commutative Typed Command Categories with the fullness property are an instance.

As in the Typed Command Calculus, λ_{inplc} regards values of certain types as "permissions" to access parts of the state. Some types have no state component, and hence confer no permission, these are termed *state-free* types. Whereas the Typed Command Calculus had a strict distinction between value types and state types, λ_{inplc} mixes the two. This in turn means that λ_{inplc} has a single context and result type, and looks more like a normal typed λ-calculus.

Most of the typing rules (Figure 8.2) make no distinction between state-free and non-state-free types. The distinction arises in two places. There are structural transitions only available to state-free context members and not available to

non-state-free members; context members of state-free type may be contracted and weakened, but context members of non-state-free type may not. This is justified by the reading of non-state-free types as carriers of permissions, which must not be duplicated or discarded. The distinction is also used in the two function types of the calculus. There is a state-free function type, whose values may not close over non-state-free variables, and a non-state-free function type, whose values may closed over non-state-free variables.

In Section 8.1 we introduce the syntax of types and contexts for λ_{inplc}, and describe the distinction between state-free and non-state-free types. We then describe the valid structural transitions of the calculus, paying careful attention to the distinction between state-free and non-state-free types, and the different structural rules that apply. Substitution in λ_{inplc} requires a more careful analysis to formulate a restricted rule, which we do in Section 8.2. To gain insight into the restricted form of the substitution rule that we must adopt, we sketch an interpretation of the calculus in a commutative TCC. We will see that the problem with substitution arises from an implicit notion of value in λ_{inplc} induced by the distinction between stateful and state-free types. Section 8.3 sets out the equational rules of λ_{inplc} theories.

The remainder of the chapter deals with direct categorical models of the calculus. Section 8.4 describes the categorical structure required to directly model λ_{inplc}: In-place Update Categories and proves that they are coherent, sound and complete class of models. Finally, Section 8.5 returns to Typed Command Categories and shows that they are an instance of In-place Update Categories. We translate the constructions of Chapter 7 for boxed data types and lists into λ_{inplc}.

8.1 In-place Update Systems

Assume a set of primitive types $\mathcal{T}$ and a disjoint set of state-free primitive types $\mathcal{T}_{\mathsf{sf}}$. The set of types is generated by the following grammar:

$$A \;::=\; X \in \mathcal{T} \cup \mathcal{T}_{\mathsf{sf}} \;\mid\; A_1 \otimes A_2 \;\mid\; I \;\mid\; A_1 \to A_2 \;\mid\; A_1 \multimap A_2$$

The predicate $\mathsf{sf}(A)$ determines a subset of all types which are *state-free*. Values of these types will not contain any state information at runtime and so can be treated as pure values.

$$\begin{aligned}
\mathsf{sf}(X) &\Leftrightarrow X \in \mathcal{T}_{\mathsf{sf}} \\
\mathsf{sf}(A_1 \otimes A_2) &\Leftrightarrow \mathsf{sf}(A_1) \wedge \mathsf{sf}(A_2) \\
\mathsf{sf}(I) &\Leftrightarrow \text{always} \\
\mathsf{sf}(A_1 \to A_2) &\Leftrightarrow \text{always} \\
\mathsf{sf}(A_1 \multimap A_2) &\Leftrightarrow \text{never}
\end{aligned}$$

The state-free predicate is derived from the set of state-free types. The intuitive semantics of state-free types is that they do not refer to the state, and do not confer any permissions to alter any state. A pair type $A_1 \otimes A_2$ is state free iff both sides are state-free; this is because values of a pair type are pairs of values of the constituent types, if neither part refers to the state, then the whole does not. The two function types are always and never state-free respectively; this is due to the fact that functions of type $A_1 \to A_2$ may only close over state-free variables, and functions of type $A_1 \multimap A_2$ may close over any variables.

Contexts are generated by the following grammar, where no variable x may appear more than once in any given context:

$$\Gamma, \Delta \quad ::= \quad I \quad | \quad x : A \quad | \quad \Gamma_1, \Gamma_2$$

For any variable x in a context Γ the notation $\Gamma[x]$ denotes the type assigned to that variable, and $v(\Gamma)$ denotes the list of variables in Γ given by a depth-first, left-to-right traversal. The state-free predicate is extended to contexts:

$$\begin{aligned}
\mathsf{sf}(I) &\Leftrightarrow \text{always} \\
\mathsf{sf}(x : A) &\Leftrightarrow \mathsf{sf}(A) \\
\mathsf{sf}(\Gamma_1, \Gamma_2) &\Leftrightarrow \mathsf{sf}(\Gamma_1) \wedge \mathsf{sf}(\Gamma_2)
\end{aligned}$$

We also consider contexts with holes by adding the following production to the grammar:

$$\Gamma, \Delta \quad ::= \quad \ldots \quad | \quad -_a$$

where a is a name for the hole, and no hole name may appear more than once in a context. We will write a context with a hole as $\Gamma(-)_a$, explicitly naming the hole, or as $\Gamma(-)$ when it does not matter what the hole is named. The notation $\Gamma(\Gamma')$ denotes a context with a hole with the hole filled in with the context Γ', i.e. the position of the hole in the tree is replaced by the context Γ'. This operation is only well-defined when the two contexts have disjoint variable and hole names.

Since we do not know whether holes will be filled in with state-free contexts or not, we take the conservative route with the extension of the state-free predicate to contexts with holes:

$$\mathsf{sf}(-_a) \quad \Leftrightarrow \quad \text{never}$$

A structural transition is a triple $\Gamma \overset{\rho}{\Rightarrow} \Delta$, where Γ and Δ are contexts and ρ is a mapping of variables in Δ to variables in Γ. The rules in Figure 8.1 define a judgement $\Gamma \overset{\rho}{\Rightarrow} \Delta$ *valid*, which identifies a subset of the structural transitions which are valid for λ_{inplc}.

$$\frac{\rho(\Gamma') = \Gamma}{\Gamma \overset{\rho}{\Rightarrow} \Gamma' \ \mathit{valid}} \qquad \frac{\Gamma_1 \overset{\alpha}{\Rightarrow} \Gamma_2 \ \mathit{valid} \qquad \Gamma_2 \overset{\beta}{\Rightarrow} \Gamma_3 \ \mathit{valid}}{\Gamma_1 \overset{\alpha;\beta}{\Rightarrow} \Gamma_3 \ \mathit{valid}}$$

$$\frac{\Gamma_1 \overset{\alpha}{\Rightarrow} \Gamma_1' \ \mathit{valid} \qquad \Gamma_2 \overset{\beta}{\Rightarrow} \Gamma_2' \ \mathit{valid}}{\Gamma_1, \Gamma_2 \overset{\alpha;\beta}{\Rightarrow} \Gamma_1', \Gamma_2' \ \mathit{valid}} \qquad \frac{}{(\Gamma_1, \Gamma_2), \Gamma_3 \Leftrightarrow \Gamma_1, (\Gamma_2, \Gamma_3) \ \mathit{valid}}$$

$$\frac{}{I, \Gamma \Leftrightarrow \Gamma \ \mathit{valid}} \qquad \frac{}{\Gamma, I \Leftrightarrow \Gamma \ \mathit{valid}} \qquad \frac{}{\Gamma_1, \Gamma_2 \Leftrightarrow \Gamma_2, \Gamma_1 \ \mathit{valid}} \qquad \frac{\mathsf{sf}(\Gamma)}{\Gamma \Rightarrow I \ \mathit{valid}}$$

$$\frac{\mathsf{sf}(\Gamma) \qquad \Gamma \equiv_\alpha \Gamma'}{\Gamma \overset{[v(\Gamma)/v(\Gamma')]}{\Rightarrow} \Gamma, \Gamma' \ \mathit{valid}}$$

Figure 8.1: Valid Structural Transitions

The valid structural transitions consist of the usual identity, composition, congruence, associativity, unit and exchange rules. There are also two conditional

rules of weakening and contraction. These are only for use with state-free contexts. The justification for the special transitions for state-free contexts relies on the reading of context members as "permissions" to perform side-effects. Non-state-free context members represent the fact that the term e in a judgement $\Gamma \vdash e : A$ requires access to a piece of the state. Permission to access a piece of the state may not be duplicated or discarded, since the first would lead to runtime type errors and the second to memory leaks. State-free context members, as the name suggests, only represent pure values and so they may be duplicated and discarded at will, by these two structural transitions.

Structural transitions between contexts with holes $\Gamma\overrightarrow{(-)_a} \stackrel{\rho}{\Rightarrow} \Delta\overrightarrow{(-)_b}$ are maps ρ that map variables in Δ to variables in Γ and hole names in Δ to hole names in Γ. Valid such structural transitions are included in the rules of Figure 8.1 by the identity/renaming rule.

To ease reasoning involving valid structural transitions, we have the following lemma, which characterises valid structural transitions. With this lemma we will be able to reason about valid structural extensions abstractly rather than worry about the exact derivation of validity. This will be useful for proving coherence and soundness.

Lemma 8.1.1 A structural transition $\Gamma \stackrel{\rho}{\Rightarrow} \Delta$ is valid iff:

1. $\forall y.\ |\{x : \rho(x) = y\}| \neq 1$ implies $\mathsf{sf}(\Gamma[y])$;
2. $\Delta[x] = \Gamma[\rho(x)]$.

The same characterisation extends to contexts with holes.

Proof The forward direction is by induction on the derivation of validity. It is easy to see that all the rules preserve the types of the variables. All of the basic transitions apart from weakening and contraction perform no renaming, and so they maintain the first property. Weakening and contraction also preserve property 1 since they only work on state-free contexts, whose components, by definition are all state-free.

For the reverse direction, given a structural transition $\Gamma \stackrel{\rho}{\Rightarrow} \Delta$ obeying the two properties, we construct a canonical derivation tree for it. For each variable

x in Γ with state-free type A there is a set $\rho^{-1}(x)$ of variables in Δ that are mapped to it. Say for a given x that the variables are $y_1, ..., y_n$. Construct a valid structural transition $x : A \overset{[x/y_1,...,x/y_n]}{\Rightarrow} (...(I, y_1 : A), ..., y_n : A_n) = \Delta_x$ by using weakening and contraction. This is possible since A is state free. For each non-state-free variable z, use the renaming rule to generate a structural transition $z : A \overset{[z' \mapsto z]}{\Rightarrow} z' : A = \Delta_z$. Likewise, for each hole a use the hole renaming rule generate the appropriate structural extension. Use the congruence rule to combine all these to get a single valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \Gamma'$, where Γ' is Γ with each state-free variable x replaced by Δ_x. The contexts Γ' and Δ have the same variables with the same types, so it is now possible to use the associativity, unit and exchange rules to generate a valid structural transition $\Gamma' \Rightarrow \Delta$ which does no renaming. Combining these using the composition rule shows that $\Gamma \overset{\rho}{\Rightarrow} \Delta$ is a valid structural transition. □

An *In-place Update System* $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi)$ consists of a set of primitive types, a disjoint set of primitive state-free types and a set of primitive operations Φ of the form $p : A_1 \to A_2$, where A_1 and A_2 are types generated by the grammar above. Along with types, contexts and valid structural transitions, a system generates a set of terms:

$$\begin{array}{rcl} e & ::= & x \mid \text{let } x = e_1 \text{ in } e_2 \mid pe \\ & \mid & \star_I \mid \text{let } \star_I = e_1 \text{ in } e_2 \\ & \mid & (e_1, e_2) \mid \text{let } (x, y) = e_1 \text{ in } e_2 \\ & \mid & \lambda_{\to} x : A.e \mid e_1 @_{\to} e_2 \\ & \mid & \lambda_{\multimap} x : A.e \mid e_1 @_{\multimap} e_2 \end{array}$$

and a typing judgement $\Gamma \vdash e : A$ by the rules in Figure 8.2.

For a given system $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi)$, and two types A_1 and A_2 generated by it, we will write $\Phi(A_1, A_2)$ for the subset of Φ of the form $p : A_1 \to A_2$.

We describe each of the typing rules in turn. The STRUCT rule incorporates the valid structural transitions into the calculus. The special transitions for state-free contexts mean that the behaviour of the pair introduction ($\otimes$I) and elimination ($\otimes$E) rules depends on the types of the components. When both components of

$$\frac{}{x : A \vdash x : A}\ (\textsc{Id}) \qquad \frac{\Gamma' \vdash e : A \qquad \Gamma \overset{\rho}{\Rightarrow} \Gamma'\ \mathit{valid}}{\Gamma \vdash \rho(e) : A}\ (\textsc{Struct}) \qquad \frac{}{I \vdash \star_I : I}\ (I\mathrm{I})$$

$$\frac{\Gamma_1 \vdash e_1 : I \qquad \Gamma_2(I) \vdash e_2 : A}{\Gamma_2(\Gamma_1) \vdash \mathrm{let}\ \star_I = e_1\ \mathrm{in}\ e_2 : A}\ (I\mathrm{E}) \qquad \frac{\Gamma_1 \vdash e_1 : A_1 \qquad \Gamma_2 \vdash e_2 : A_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : A_1 \otimes A_2}\ (\otimes\mathrm{I})$$

$$\frac{\Gamma_1 \vdash e_1 : A_1 \otimes A_2 \qquad \Gamma_2(x : A_1, y : A_2) \vdash e_2 : B}{\Gamma_2(\Gamma_1) \vdash \mathrm{let}\ (x, y) = e_1\ \mathrm{in}\ e_2 : B}\ (\otimes\mathrm{E})$$

$$\frac{\Gamma, x : A \vdash e : B \qquad \mathsf{sf}(\Gamma)}{\Gamma \vdash \lambda_{\rightarrow} x.e : A \rightarrow B}\ (\rightarrow\mathrm{I}) \qquad \frac{\Gamma_1 \vdash e_1 : A \rightarrow B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1 @_{\rightarrow} e_2 : B}\ (\rightarrow\mathrm{E})$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda_{\multimap} x.e : A \multimap B}\ (\multimap\mathrm{I}) \qquad \frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1 @_{\multimap} e_2 : B}\ (\multimap\mathrm{E})$$

$$\frac{\Gamma \vdash e : A \qquad (p : A \rightarrow B) \in \Phi}{\Gamma \vdash pe : B}\ (\textsc{Prim})$$

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2(x : A) \vdash e_2 : B}{\Gamma_2(\Gamma_1) \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : B}\ (\textsc{Let})$$

Figure 8.2: In-place-update Typing Rules

a pair are state-free the type acts as a normal product type. The rules $I\mathrm{I}$ and $I\mathrm{E}$ are the usual introduction and elimination rules for the unit type.

There are two function types in λ_{inplc}, similar to the two function types of λ_{local}. State-free functions, $A \to B$, do not close over any non-state-free context members (the $\mathsf{sf}(\Gamma)$ premise) and so are state-free themselves; permissions to any state that the body of the function operates on must be provided as an argument. Non-state-free functions, $A \multimap B$, may close over state and so are regarded as non-state-free and thus may not be discarded or duplicated, since this would imply that the permissions they contain be duplicated or discarded. Note that, due to the single context in λ_{inplc}, the differences between the two function types boil down to differences in their state-free status, and in the introduction rules. The elimination rules for both function types are identical.

The Prim rule incorporates the primitive operations from Φ into the set of typed terms.

8.2 Substitution

The λ_{inplc} calculus does not admit the full substitution rule. The conditional structural rules of contraction and weakening and the $A \to B$ introduction rule mean that an attempt to substitute a term typed in a non-state-free context for a state-free variable may be ill-typed. For example, consider the judgement:

$$x : \text{Int} \vdash (x, x) : \text{Int} \otimes \text{Int}$$

This judgement is valid if we assume that Int is state-free. Now consider another judgement:

$$d : \text{Cell} \vdash \text{location}(d) : \text{Int}$$

where the type Cell is not state-free. An attempt to substitute the term $\text{loc}(d)$ for x in the first judgement would result in an ill-typed term:

$$(\text{location}(d), \text{location}(d))$$

The variable d appears twice in the term, but since d is of a non-state-free type the term is not typable. The problem also arises when the variable to be substituted for occurs inside a $\lambda_{\rightarrow}$ term.

It is only safe to substitute a term with a non-state-free context for a variable when it has not been acted upon by a typing rule that assumes state-freeness. We term such occurrences of variables *non-linear* occurrences:

Definition 8.2.1 A variable x occurs *non-linearly* in a term e if its number of occurrences is not equal to one, or it appears free inside a $\lambda_{\rightarrow}$ term.

For a variable to occur any number of times other that one it must have been acted upon by Weakening or Contraction, and so must be state-free. Also, for it to have appeared free in a $\lambda_{\rightarrow}$ term it must have been state-free. Therefore, any variable that occurs non-linearly in state-free.

With this definition we can state a restricted substitution rule suitable for λ_{inplc}:

Lemma 8.2.2 (Restricted Substitution) The following rule of restricted substitution is admissible:

$$\frac{\overrightarrow{\Delta \vdash e : A} \qquad \Gamma\overrightarrow{(x : A)} \vdash e' : B \qquad x \text{ non-linear in } e' \text{ implies } \mathsf{sf}(\Delta)}{\Gamma\overrightarrow{(\Delta)} \vdash e'[\overrightarrow{e/x}] : B}$$

In order to prove this we will need the following lemma which states how substitution interacts with structural transitions:

Lemma 8.2.3 Assume:

- Typed variables $x_1 : A_1, ..., x_n : A_n$ and contexts $\Delta_1, ..., \Delta_n$;
- For each x_i, zero or more variables $y_1^i, ..., y_{k_i}^i$, such that if $k_i \neq 1$ then Δ_1 is state free;
- A valid structural transition $\Gamma_1(x_1 : A_1)...(x_n : A_n) \stackrel{\alpha}{\Rightarrow} \Gamma_2\overrightarrow{(y^1 : A_1)}...\overrightarrow{(y^n : A_n)}$, such that for all i, j, $\alpha(y_j^i) = x_i$.

For each y^i_j generate a new context from Δ_i, called Δ^i_j, by renaming the variables. I.e. there is a valid structural transition $\Delta_i \overset{\rho^i_j}{\Rightarrow} \Delta^i_j$ such that ρ^i_j is bijective. Then there is a valid structural transition $\Gamma_1(\Delta_1)...(\Delta_n) \overset{\beta}{\Rightarrow} \Gamma_2(\overrightarrow{\Delta^1})...(\overrightarrow{\Delta^n})$, such that for all i, j and $z \in v(\Delta^1_j)$, $\beta(z) = \rho_i(z)$ and for all $z \in v(\Gamma_2(-))$, $\beta(z) = \alpha(z)$.

Proof Simply define β to have the same action as α on variables in $\Gamma_2(-)$ and the action of ρ^i_j for variables in Δ^i_j. The fact that $k_i \neq 1$ implies $\mathsf{sf}(\Delta_i)$ means that this is a valid structural transition. □

Proof *(of Lemma 8.2.2)* By induction on the derivation of $\Gamma_2(x : A) \vdash e_2 : B$. Most cases are simple and follow just by applying the induction hypothesis and then applying the appropriate typing rule and relying on the fact that substitution of free variables commutes with all our term constructors. The only difficult case is Struct, which is handled by Lemma 8.2.3 in the same way as for λ_{sep} in the proof of Lemma 2.2.4. □

8.2.1 Interpretation in a Typed Command Category

This restricted substitution rule is all very well, but is there a deeper reason for the restriction of substitutable-for variables? In this subsection we attempt to answer this by sketching an interpretation of λ_{inplc} in a Typed Command Category.

Assume a commutative Typed Command Category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ such that the functor $J(-, I)$ is full and faithful.

Following the intuitive idea presented in the chapter introduction that the types of λ_{inplc} are composed of a stateful part and a state-free part, we interpret types as objects of $\mathcal{K}$. The stateful part is represented by the $\mathcal{S}$ component, and the state-free part is represented by the $\mathcal{C}$ component. State-free types of λ_{inplc} are interpreted as objects of $\mathcal{K}$ whose state component is I.

Contexts are interpreted as functors $\mathcal{K}^n \to \mathcal{K}$, where n is the number of holes in the context. We define the interpretation by induction on the structure of the context:

$$[\![-_a]\!] = Id : \mathcal{K} \to \mathcal{K} \qquad [\![x : A]\!] = \star \mapsto [\![A]\!] : 1 \to \mathcal{K} \qquad [\![I]\!] = \star \mapsto (I, I) : 1 \to \mathcal{K}$$

$$[\![\Gamma_1, \Gamma_2]\!] = ([\![\Gamma_1]\!] \times [\![\Gamma_2]\!]); \otimes : \mathcal{K}^{n_1+n_2} \to \mathcal{K}$$

where n_1 and n_2 are the number of holes in Γ_1 and Γ_2 respectively, and $\otimes$ is the symmetric monoidal product on $\mathcal{K}$ defined in Proposition 5.4.3.

Valid structural transitions are interpreted as natural transformations. In the case of stateful contexts, all the necessary natural transformations are given by the symmetric monoidal structure of $\mathcal{K}$. When a context is heap free then its state component is isomorphic to I by the interpretation of state-free types. Therefore, we can use the finite product structure in $\mathcal{C}$ to interpret the conditional Contraction and Weakening rules.

The interpretation of typing judgements is defined by induction over the typing derivation. Most of the cases are straightforward. However, we give the cases for the two function introductions:

$$\frac{[\![\Gamma, x : A \vdash e : B]\!] = e \qquad \mathsf{sf}(\Gamma)}{[\![\Gamma \vdash \lambda_{\rightarrow} x : A.e : A \rightarrow B]\!] = J(id, \cong_1); J(\Lambda^{\rightarrow}(J(id, \cong_2); e), I)}$$

$$\frac{[\![\Gamma, x : A \vdash e : B]\!] = e}{[\![\Gamma \vdash \lambda_{\multimap} x : A.e : A \rightarrow B]\!] = \Lambda^{\multimap}(e)}$$

where the isomorphisms $\cong_1$ and $\cong_2$ are the canonical ones taking multiple copies of I to a single copy and back again.

We now examine the restriction to non-linear occurrences in Lemma 8.2.2. As noted above, for a variable to occur non-linearly it must have state-free type. The restricted substitution rule then requires that the term to be substituted in must have a state-free context.

By the fullness property of $J(-, I)$ and the interpretation of value types, it is the case that terms with state-free context and result type are actually interpreted as *values* in commutative Typed Command Categories. The fact that we can always substitute in terms of this form confirms the explanation presented in the introduction that λ_{inplc} implicitly forces a call-by-value scheme by the construction of its type system.

It is still permissible to substitute in arbitrary well-typed terms for linearly occurring variables. This is explained by the restriction to commutative Typed Command Categories. Since the order of side-effects on the state is controlled by

the typing, not the actual ordering of commands, we can re-order the commands by substitution as long as the typing is preserved. Any more liberal substitution policy would however violate the non-sharing property of the monoidal product on $\mathcal{S}$ and so is not allowed.

8.3 Equational Theory

An *In-place Update Theory* $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi, \Sigma)$ is an In-place Update System $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi)$ plus a set Σ of axioms of the form $\Gamma \vdash e_1 = e_2 : A$ where $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ are derivable. A theory induces an equational judgement defined by the rules in Figures 8.3, 8.4 and 8.5.

$$\frac{(\Gamma \vdash e_1 = e_2 : A) \in \Sigma}{\Gamma \vdash e_1 = e_2 : A}\ (\textsc{Eq-Ax})$$

$$\frac{\Gamma \vdash e_1 = e_2 : A \qquad \Gamma \overset{\rho}{\Rightarrow} \Gamma' \ \mathit{valid}}{\Gamma' \vdash \rho(e_1) = \rho(e_2) : A}\ (\textsc{Eq-Struct})$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : A \\ \Gamma_2(z : A) \vdash e_2 : B \qquad z \text{ occurs non-linearly in } e_2 \text{ implies } \mathsf{sf}(\Gamma_1)\end{array}}{\Gamma_2(\Gamma_1) \vdash (\text{let } x = e_1 \text{ in } e_2[x/z]) = e_2[e_1/z] : B}\ (\textsc{Eq-Let})$$

Plus: reflexivity, symmetry, transitivity and congruence rules.

Figure 8.3: In-place Update Equational Rules: Basics

The rules in Figure 8.3 cover the basic rules for axioms, the equality preservation of the Struct rule and the rules for the Let construct. The rule for Let subsumes the usual β, η and commuting conversion rules. It is restricted by the side-condition to ensure that the substitutions are always valid. The system also includes the standard rules for reflexivity, symmetry, transitivity and congruence.

$$\frac{\Gamma(I) \vdash e : A}{\Gamma(I) \vdash (\text{let } \star_I = \star_I \text{ in } e) = e : A}\ (\text{Eq-}\beta\text{-}I)$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : I \\ \Gamma_2(z : I) \vdash e_2 : A \qquad z \text{ occurs non-linearly in } e_2 \text{ implies } \mathsf{sf}(\Gamma_1)\end{array}}{\Gamma_2(\Gamma_1) \vdash (\text{let } \star_I = e_1 \text{ in } e_2[\star_I/z]) = e_2[e_1/z] : A}\ (\text{Eq-}\eta\text{-}I)$$

$$\frac{\Gamma_1 \vdash e_1 : A_1 \qquad \Gamma_2 \vdash e_2 : A_2 \qquad \Gamma_3(x : A_1, y : A_2) \vdash e_3 : A_3}{\Gamma_3(\Gamma_1, \Gamma_2) \vdash (\text{let } (x, y) = (e_1, e_2) \text{ in } e_3) = (\text{let } x = e_1 \text{ in let } y = e_2 \text{ in } e_3) : A_3}\ (\text{Eq-}\beta\text{-}\otimes)$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : A \otimes B \\ \Gamma_2(z : A \otimes B) \vdash e_2 : C \qquad z \text{ occurs non-linearly in } e_2 \text{ implies } \mathsf{sf}(\Gamma_1)\end{array}}{\Gamma_2(\Gamma_1) \vdash (\text{let } (x, y) = e_1 \text{ in } e_2[(x, y)/z]) = e_2[e_1/z] : C}\ (\text{Eq-}\eta\text{-}\otimes)$$

Figure 8.4: In-place Update Equational Rules: Pairs

Figure 8.4 contains the equational rules for units and pairs. The β rule for pairs is standard albeit with the use of a unary let in the β rule for pairs to overcome the problem with substitution. We also have Ghani's extended η rule [Gha95] for units and tensor products which subsumes the commuting conversion rules. As with λ_{sep}, the extensionality rules for units and state-free products are derivable from these rules.

$$\frac{\Gamma, x : A \vdash e_1 : B \qquad \Gamma' \vdash e_2 : A \qquad \mathsf{sf}(\Gamma)}{\Gamma, \Gamma' \vdash (\lambda_{\rightarrow} x.e_1)@_{\rightarrow} e_2 = \text{let } x = e_2 \text{ in } e_1 : B} \ (\text{Eq-}\beta\text{-}\rightarrow)$$

$$\frac{\Gamma \vdash e : A \rightarrow B}{\Gamma \vdash (\lambda_{\rightarrow} x.e@_{\rightarrow} x) = e : A \rightarrow B} \ (\text{Eq-}\eta\text{-}\rightarrow)$$

$$\frac{\Gamma, x : A \vdash e_1 : B \qquad \Gamma' \vdash e_2 : A}{\Gamma, \Gamma' \vdash (\lambda_{\multimap} x.e_1)@_{\multimap} e_2 = \text{let } x = e_2 \text{ in } e_1 : B} \ (\text{Eq-}\beta\text{-}\multimap)$$

$$\frac{\Gamma \vdash e : A \multimap B}{\Gamma \vdash (\lambda_{\multimap} x.e@_{\multimap} x) = e : A \multimap B} \ (\text{Eq-}\eta\text{-}\multimap)$$

Figure 8.5: In-place Update Equational Rules: Functions

The final group of equational rules is shown in Figure 8.5. Both function types have the normal β and η rules, albeit using a unary let instead of substitution for the β rules. Note that both functions have identical equation rules; the function types only differ in the way they interact with the structural rules via the state-free-ness predicate.

Proposition 8.3.1 If $\Gamma \vdash e_1 = e_2 : A$ is derivable, then $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ are derivable.

Proof By induction over the derivation of $\Gamma \vdash e_1 = e_2 : A$. The case for Eq-Ax follows from the definition of a theory. All the other cases follow from the substitution lemma, Lemma 8.2.2. □

8.4 Categorical Semantics

In this section, we give the categorical structure for interpreting λ_{inplc} directly, and show that λ_{inplc} is sound and complete for the class of models with this structure. In the next section we will show that certain commutative closed double parameterised Freyd-categories form an instance of this structure.

Definition 8.4.1 An *In-place Update Category* consists of a symmetric monoidal closed category $\mathcal{K}$, with a finite product sub-category $\mathcal{C}$, such that the inclusion functor $J : \mathcal{C} \to \mathcal{K}$ is full as well as faithful and strong symmetric monoidal and that, for all objects A of $\mathcal{K}$ the functor $J(-) \otimes A$ has a specified right adjoint, written as $A \to -$.

As implied in the definition we will use $\otimes$ as the symbol for the functor part of the monoidal structure on $\mathcal{K}$, and $\times$ for the product structure on $\mathcal{C}$. Recall that a strong symmetric monoidal functor is one for which there is an isomorphism $m_{A,B} : JA \otimes JB \cong J(A \times B)$ and an isomorphism $m_I : I \cong J1$ which commute with the associativity, unit and symmetry natural isomorphisms. For the two closures we will use $\Lambda^{\to}$ and $\Lambda^{\multimap}$ for the two homset isomorphisms:

$$\Lambda^{\to} : \mathcal{K}(J(A) \otimes B, C) \cong \mathcal{C}(A, B \to C) \qquad \Lambda^{\multimap} : \mathcal{K}(A \otimes B, C) \cong \mathcal{C}(A, B \multimap C)$$

We will use $\text{ev}^{\to}$ and $\text{ev}^{\multimap}$ for the two counits, both of which have their components in $\mathcal{K}$:

$$\text{ev}^{\to}_{A,B} : J(A \to B) \otimes A \to B \qquad \text{ev}^{\multimap}_{A,B} : (A \multimap B) \otimes A \to B$$

An interpretation of an In-place Update System $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi)$ in an In-place Update Category $J : \mathcal{C} \to \mathcal{K}$ consists of three functions: $[\![\cdot]\!]_0 : \mathcal{T} \to \mathsf{Ob}\mathcal{K}$; $[\![\cdot]\!]_0 : \mathcal{T}_{\mathsf{sf}} \to \mathsf{Ob}\mathcal{C}$; and $[\![\cdot]\!]_0 : \Phi(A, B) \to \mathcal{K}([\![A]\!], [\![B]\!])$. An interpretation induces a

mapping from types to objects of $\mathcal{K}$:

$$\begin{array}{rcl} [\![X \in \mathcal{T}]\!] & = & [\![X]\!]_0 \\ [\![X \in \mathcal{T}_{\mathsf{sf}}]\!] & = & J([\![X]\!]_0) \\ [\![A_1 \otimes A_2]\!] & = & [\![A_1]\!] \otimes [\![A_2]\!] \\ [\![I]\!] & = & I \\ [\![A \to B]\!] & = & J([\![A]\!] \to [\![B]\!]) \\ [\![A \multimap B]\!] & = & [\![A]\!] \multimap [\![B]\!] \end{array}$$

And a mapping of contexts to functors $\mathcal{K}^n \to \mathcal{K}$ where n is the number of holes in the context:

$$[\![x : A]\!] = \star \mapsto [\![A]\!] : 1 \to \mathcal{K} \qquad [\![I]\!] = \star \mapsto I$$

$$[\![\Gamma_1, \Gamma_2]\!] = ([\![\Gamma_1]\!] \times [\![\Gamma_2]\!]); \otimes : \mathcal{K}^{n_1+n_2} \to \mathcal{K} \qquad [\![-_a]\!] = Id : \mathcal{C} \to \mathcal{C}$$

For types and contexts that are state-free, define by induction on the structure the following alternative interpretations in the category $\mathcal{C}$:

$$\begin{array}{rcl} [\![X \in \mathcal{T}_{\mathsf{sf}}]\!]_{\mathsf{sf}} & = & [\![X]\!]_0 \\ [\![A_1 \otimes A_2]\!]_{\mathsf{sf}} & = & [\![A_1]\!]_{\mathsf{sf}} \times [\![A_2]\!]_{\mathsf{sf}} \\ [\![I]\!]_{\mathsf{sf}} & = & 1 \\ [\![A \to B]\!]_{\mathsf{sf}} & = & [\![A]\!] \to [\![B]\!] \end{array} \qquad \begin{array}{rcl} [\![x : A]\!]_{\mathsf{sf}} & = & \star \mapsto [\![A]\!]_{\mathsf{sf}} \\ [\![I]\!]_{\mathsf{sf}} & = & \star \mapsto 1 \\ [\![\Gamma_1, \Gamma_2]\!]_{\mathsf{sf}} & = & ([\![\Gamma_1]\!] \times [\![\Gamma_2]\!]); \times \\ [\![-_a]\!] & = & Id \end{array}$$

Since the inclusion functor J is strong symmetric monoidal, there are induced isomorphisms $m_A : [\![A]\!] \cong J([\![A]\!]_{\mathsf{sf}})$ and $m_\Gamma : [\![\Gamma]\!] \cong J([\![\Gamma]\!]_{\mathsf{sf}})$ for state-free types A and state-free contexts Γ, defined by induction.

An interpretation also induces two mappings to arrows in $\mathcal{K}$, again both written as $[\![\cdot]\!]$, from derivations of valid structural transitions (Figure 8.6); and from derivations of typing judgements (Figure 8.7).

8.4.1 Coherence

As with all the substructural systems in this thesis we need to prove the coherence of this interpretation. As usual, we do this in two steps, proving coherence of the interpretation of valid structural transitions with respect to structural morphisms; and then proving the coherence of the interpretation of typing derivations.

$$\frac{\rho(\Gamma') = \Gamma}{[\![\Gamma \overset{\rho}{\Rightarrow} \Gamma \ \mathit{valid}]\!] = id_{[\![\Gamma]\!]}} \qquad \frac{[\![\Gamma_1 \overset{\alpha}{\Rightarrow} \Gamma_2 \ \mathit{valid}]\!] = a \qquad [\![\Gamma_2 \overset{\beta}{\Rightarrow} \Gamma_3 \ \mathit{valid}]\!] = b}{[\![\Gamma_1 \overset{\alpha;\beta}{\Rightarrow} \Gamma_3 \ \mathit{valid}]\!] = a; b}$$

$$\frac{[\![\Gamma_1 \overset{\alpha}{\Rightarrow} \Gamma_1' \ \mathit{valid}]\!] = a \qquad [\![\Gamma_2 \overset{\beta}{\Rightarrow} \Gamma_2' \ \mathit{valid}]\!] = b}{[\![\Gamma_1, \Gamma_2 \overset{\alpha;\beta}{\Rightarrow} \Gamma_1', \Gamma_2' \ \mathit{valid}]\!] = a \otimes b}$$

$$[\![(\Gamma_1, \Gamma_2), \Gamma_3 \Leftrightarrow \Gamma_1, (\Gamma_2, \Gamma_3) \ \mathit{valid}]\!] = \alpha \qquad [\![I, \Gamma \Leftrightarrow \Gamma \ \mathit{valid}]\!] = \lambda$$

$$[\![\Gamma, I \Leftrightarrow \Gamma \ \mathit{valid}]\!] = \rho \qquad [\![\Gamma_1, \Gamma_2 \Leftrightarrow \Gamma_2, \Gamma_1 \ \mathit{valid}]\!] = \sigma$$

$$\frac{\mathsf{sf}(\Gamma)}{[\![\Gamma \Rightarrow I \ \mathit{valid}]\!] = m_\Gamma; J(!_{[\![\Gamma]\!]})}$$

$$\frac{\mathsf{sf}(\Gamma) \qquad \Gamma \equiv_\alpha \Gamma'}{[\![\Gamma \overset{[v(\Gamma)/v(\Gamma')]}{\Rightarrow} \Gamma, \Gamma' \ \mathit{valid}]\!] = m_\Gamma; J(\langle [\![\Gamma]\!], [\![\Gamma]\!] \rangle); m^{-1}_{\Gamma,\Gamma'}}$$

Figure 8.6: Interpretation of Basic Structural Transitions

$$\frac{}{[\![x : A \vdash x : A]\!] = id_{[\![A]\!]}} \qquad \frac{[\![\Gamma' \vdash e : A]\!] = e \qquad [\![\Gamma \overset{\rho}{\Rightarrow} \Gamma' \ valid]\!] = s}{[\![\Gamma \vdash \rho(e) : A]\!] = s; e}$$

$$\frac{}{[\![I \vdash \star_I : I]\!] = id_I} \qquad \frac{[\![\Gamma_1 \vdash e_1 : I]\!] = e_1 \qquad [\![\Gamma_2(I) \vdash e_2 : A]\!] = e_2}{[\![\Gamma_2(\Gamma_1) \vdash \text{let } \star_I = e_1 \text{ in } e_2 : A]\!] = [\![\Gamma_2]\!](e_1); e_2}$$

$$\frac{[\![\Gamma_1 \vdash e_1 : A_1]\!] = e_1 \qquad [\![\Gamma_2 \vdash e_2 : A_2]\!] = e_2}{[\![\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : A_1 \otimes A_2]\!] = e_1 \otimes e_2}$$

$$\frac{[\![\Gamma_1 \vdash e_1 : A_1 \otimes A_2]\!] = e_1 \qquad [\![\Gamma_2(x : A_1, y : A_2) \vdash e_2 : B]\!] = e_2}{[\![\Gamma_2(\Gamma_1) \vdash \text{let } (x, y) = e_1 \text{ in } e_2 : B]\!] = [\![\Gamma_2]\!](e_1); e_2}$$

$$\frac{[\![\Gamma_1 \vdash e_1 : A]\!] = e_1 \qquad [\![\Gamma_2(x : A) \vdash e_2 : B]\!] = e_2}{[\![\Gamma_2(\Gamma_1) \vdash \text{let } x = e_1 \text{ in } e_2 : B]\!] = [\![\Gamma_2]\!](e_1); e_2}$$

$$\frac{[\![\Gamma, x : A \vdash e : B]\!] = e \qquad \mathsf{sf}(\Gamma)}{[\![\Gamma \vdash \lambda_{\rightarrow} x.e : A \rightarrow B]\!] = m_\Gamma; \Lambda^{\rightarrow}(m_\Gamma^{-1} \otimes [\![A]\!]; e)}$$

$$\frac{[\![\Gamma_1 \vdash e_1 : A \rightarrow B]\!] = e_1 \qquad [\![\Gamma_2 \vdash e_2 : A]\!] = e_2}{[\![\Gamma_1, \Gamma_2 \vdash e_1 @_{\rightarrow} e_2 : B]\!] = e_1 \otimes e_2; \mathrm{ev}^{\rightarrow}}$$

$$\frac{[\![\Gamma, x : A \vdash e : B]\!] = e}{[\![\Gamma \vdash \lambda_{\multimap} x.e : A \multimap B]\!] = \Lambda^{\multimap}(e)}$$

$$\frac{[\![\Gamma_1 \vdash e_1 : A \multimap B]\!] = e_1 \qquad [\![\Gamma_2 \vdash e_2 : A]\!] = e_2}{[\![\Gamma_1, \Gamma_2 \vdash e_1 @_{\multimap} e_2 : B]\!] = e_1 \otimes e_2; \mathrm{ev}^{\multimap}}$$

$$\frac{[\![\Gamma \vdash e : A]\!] = e \qquad (p : A \rightarrow B) \in \Phi}{[\![\Gamma \vdash pe : B]\!] = e; [\![p]\!]}$$

Figure 8.7: Interpretation of In-place Update Calculus

Lemma 8.4.2 If π_1 and π_2 are derivations of some valid structural transition $\Gamma \overset{\rho}{\Rightarrow} \Gamma'$ then $[\![\pi_1]\!] = [\![\pi_2]\!]$.

Proof Similar to Theorem 3.2.20, by showing that the interpretation of a transition is equal to the interpretation of the canonical transition derived from the corresponding morphism. We rewrite the derivation of the transition to a sequence of basic transitions-in-context. This rewriting preserves the categorical interpretation. We then show it has the same interpretation as the canonical transitions by induction over the length: the properties required in Definition 3.2.18 for commuting weakenings and duplications with context constructors hold automatically for finite product structure so we can commute the weakening and duplication arrows to the start of the expression, then using the fact that the interpretations of weakening and duplication form a comonoid gives coherence for the initial phase, and coherence of symmetric monoidal structure gives coherence of the second stage. The result then follows. □

The proof of coherence of the interpretation of typing judgements proceeds in a similar manner to the proof for λ_{sep} (Theorem 3.3.4). We omit the details since they are similar to the λ_{sep} proof, but we recap the procedure. First we must prove two "factorisation" lemmas for valid structural transitions that describe how valid structural transitions below a rule application can be factored into parts that appear above the rule application and the part that remains below, thus rewriting the derivation tree to a canonical form. Then we strengthen the result to include trailing structural transitions and proceed by induction on the height of the derivation. The factorisation property is used to rewrite the syntax-free structural rules and Lemma 8.4.2 means that we preserve the interpretation.

Theorem 8.4.3 (Coherence) If π_1 and π_2 are derivations of some typing judgement $\Gamma \vdash e : A$ then $[\![\pi_1]\!] = [\![\pi_2]\!]$.

8.4.2 Soundness and Completeness

A *model* of an In-place Update Theory $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi, \Sigma)$ is an interpretation $[\![\cdot]\!]_0$ of the system $(\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi)$ such that for all axioms $(\Gamma \vdash e_1 = e_2 : A) \in \Sigma$, $[\![\Gamma \vdash e_1 :$

$A]\!] = [\![\Gamma \vdash e_2 : A]\!]$. We now prove that this definition of a model of an In-place Update Theory is sound and complete. First we prove that the interpretation of substitution is as expected.

Lemma 8.4.4 The restricted substitution rule has the following interpretation:

$$\frac{[\![\Gamma_1 \vdash e_1 : A_1]\!] = e_1 \qquad [\![\Gamma_2(x : A_1) \vdash e_2 : A_2]\!] = e_2 \qquad x \text{ occurs non-linearly in } e_2 \text{ implies } \mathsf{sf}(\Gamma_2)}{[\![\Gamma_2(\Gamma_1) \vdash e_2[e_1/x] : A_2]\!] = [\![\Gamma_2]\!](e_1); e_2}$$

Proof First strengthen the statement to multiple simultaneous substitutions as in Lemma 8.2.2. Then proceed by induction on the derivation of $\Gamma_2(x : A_1) \vdash e_2 : A_2$. The only difficult case is for the rule STRUCT, which is handled by considering the categorical counterpart of Lemma 8.2.3 via Lemma 8.4.2. □

The soundness of the equational theory follows straightforwardly from this lemma.

Theorem 8.4.5 (Soundness) If $\Gamma \vdash e_1 = e_2 : A$ then $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

Proof By induction on the derivation of $\Gamma \vdash e_1 = e_2 : A$. □

We now show that the class of models defined is complete for a given In-place Update Theory. First we show that the syntax of a theory generates an In-place Update category:

Theorem 8.4.6 (Term Category) Given an In-place Update Theory $\mathbb{T} = (\mathcal{T}, \mathcal{T}_{\mathsf{sf}}, \Phi)$, there is an In-place Update Category, defined as follows: a category $\mathcal{K}_{\mathbb{T}}$:

Objects Types generated from $\mathbb{T}$;

Arrows Equivalence classes of typed expressions $[x : A_1 \vdash e : A_2]$.

and a full sub-category $\mathcal{C}_{\mathbb{T}}$:

Objects State-free types generated by $\mathbb{T}$;

Arrows Equivalence classes of typed expressions $[x : A_1 \vdash e : A_2]$.

Proof First we establish that $\mathcal{K}_\mathbb{T}$ and $\mathcal{C}_\mathbb{T}$ are categories. In both cases $\mathcal{K}_\mathbb{T}$ the identities are defined as $[x : A \vdash x : A]$ and composition of $[x : A \vdash e_1 : B]$ and $[x : B \vdash e_2 : C]$ is defined as $[x : A \vdash \text{let } x = e_1 \text{ in } e_2 : C]$. It is easy to check that these obey the required properties to make them categories.

The inclusion functor $J : \mathcal{C}_\mathbb{T} \to \mathcal{K}_\mathbb{T}$ is defined as $J(A) = A$ and $J([x : A \vdash e : B]) = [x : A \vdash e : B]$. This clearly defines a functor. It is obviously faithful since no more equalities are imposed on arrows in $\mathcal{K}_\mathbb{T}$ than in $\mathcal{C}_\mathbb{T}$. It is also full.

Symmetric monoidal structure is defined in both categories on objects A and B just as the type $A \otimes B$. The proof that this defines symmetric monoidal structure is standard [Bar96]. Given this definitions, it is clear that J is strict symmetric monoidal.

The monoidal structure on $\mathcal{C}_\mathbb{T}$ is also the categorical product. Define the isomorphism of homsets $\phi : \mathcal{C}_\mathbb{T}(A, B \otimes C) \cong \mathcal{C}_\mathbb{T}(A, B) \times \mathcal{C}_\mathbb{T}(A, C)$ as:

$$\phi([e]) = ([\text{let } (x_1, x_2) = e \text{ in } x_1], [\text{let } (x_1, x_2) = e \text{ in } x_2])$$

$$\phi^{-1}([e_1], [e_2]) = [(e_1, e_2)]$$

where we have elided the contexts and result types to save space. These are inverse:

$$\begin{aligned}
& \phi^{-1}(\phi([e])) \\
= \;& \phi^{-1}([\text{let } (x_1, x_2) = e \text{ in } x_1], [\text{let } (x_1, x_2) = e \text{ in } x_2]) \\
= \;& [(\text{let } (x_1, x_2) = e \text{ in } x_1, \text{let } (x_1, x_2) = e \text{ in } x_2)] \\
= \;& [\text{let } x = e \text{ in } (\text{let } (x_1, x_2) = x \text{ in } x_1, \text{let } (x_1, x_2) = x \text{ in } x_2)] \\
= \;& [\text{let } x = e \text{ in } x] \\
= \;& [e]
\end{aligned}$$

where the crucial steps follow from the Eq-β-Let and Eq-η-2-$\otimes$ rules which are only valid when the types are state-free, as they are in $\mathcal{C}_\mathbb{T}$. In the opposite direction:

$$\phi(\phi^{-1}([e_1], [e_2]))$$

$$\begin{aligned} &= \phi([(e_1, e_2)]) \\ &= ([\text{let } (x_1, x_2) = (e_1, e_2) \text{ in } x_1], [\text{let } (x_1, x_2) = (e_1, e_2) \text{ in } x_2]) \\ &= ([e_1], [e_2]) \end{aligned}$$

where we again rely on the ability to substitute when the types are state-free. It is easy to check that the isomorphism of homsets above is natural, so $\mathcal{T}_{\mathbb{T}}$ has categorical products. Moreover, the type I is a terminal object in $\mathcal{C}_{\mathbb{T}}$ by the Eq-η-2-I rule, and so $\mathcal{C}_{\mathbb{T}}$ has all finite products.

The functor $A \to -$ is defined on objects just as the type $A \to B$, and on arrows $[g]$ as $[\lambda_{\to} y.\text{let } x = x@_{\to} y \text{ in } g]$. Define the isomorphism of homsets as:

$$\Lambda^{\to}([e]) = [\lambda_{\to} y.\text{let } x = (x, y) \text{ in } e] \qquad \Lambda^{\to -1}([e]) = [\text{let } (x, y) = x \text{ in } e@_{\to} y]$$

It is easy to check that these two are inverse and natural. Symmetric monoidal closure in $\mathcal{K}_{\mathbb{T}}$ is defined and proven suitable similarly. □

We will give a model of a theory $\mathbb{T}$ in the In-place Update category $J_{\mathbb{T}} : \mathcal{C}_{\mathbb{T}} \to \mathcal{K}_{\mathbb{T}}$ by mapping types to their corresponding objects and primitive operations p to the terms $[x : A \vdash px : B]$. To use this model to prove completeness we must also have an equality between the interpretation of a judgement $\Gamma \vdash e : A$ and an arrow of $J_{\mathbb{T}} : \mathcal{C}_{\mathbb{T}} \to \mathcal{K}_{\mathbb{T}}$ derived from this judgement. To this end, for a context Γ, define the corresponding type $\overline{\Gamma}$ by induction:

$$\overline{I} = I \qquad \overline{x : A} = A \qquad \overline{\Gamma_1, \Gamma_2} = \overline{\Gamma_1} \otimes \overline{\Gamma_2}$$

And given a term $\Gamma \vdash e : A$, then $x : \overline{\Gamma} \vdash \overline{\Gamma, x, e_2} : A$ is derivable, where:

$$\overline{I, x, e} = \text{let } \star_I = x \text{ in } e \qquad \overline{y : A, x, e} = e[y/x]$$

$$\overline{(\Gamma_1, \Gamma_2), x, e} = \text{let } (x_1, x_2) = x \text{ in } \overline{\Gamma_1, x_1, \overline{\Gamma_2, x_2, e}}$$

This interpretation described above has the property that the interpretation of each judgement is equal to these translated judgements, and so is a model

Proposition 8.4.7 (Term Model) The interpretation of a theory $\mathbb{T}$ in the In-place Update Category defined in Theorem 8.4.6 is a model such that: $[\![\Gamma \vdash e : A]\!] = [x : \overline{\Gamma} \vdash \overline{\Gamma, x, e} : A]$.

Proof The property is proven by induction on the derivation of $\Gamma \vdash e : A$ and examining the interpretation. From this, and the construction of arrows of the term category as equivalence classes, it follows that we have defined a model. □

This term model is used to prove completeness:

Theorem 8.4.8 (Completeness) If $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$ for all models then $\Gamma \vdash e_1 = e_2 : A$.

Proof We prove the contrapositive. Assume $\Gamma \vdash e_1 \neq e_2 : A$. It is obviously the case that this implies that $x : \overline{\Gamma} \vdash \overline{\Gamma, x, e_1} \neq \overline{\Gamma, x, e_2} : A$, since the context Γ is the same in both cases. Hence $[\![\Gamma \vdash e_1 : A]\!] \neq [\![\Gamma \vdash e_2 : A]\!]$ in the term model, i.e. the interpretations are not equal in all models. □

8.5 Commutative Typed Command Categories

In this final section, we show that the Commutative Typed Command Categories with the fullness property are an instance of In-place Update categories. This will imply that the heap bounded state model of Chapter 7 is a model of λ_{inplc}. We finish the chapter by translating the constructions for boxed data types and singly linked lists into λ_{inplc}

Proposition 8.5.1 Given a commutative Typed Command Category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ such that the functor $J(-, I) : \mathcal{C} \to \mathcal{K}$ is full and faithful, then it is the case that $J(-, I) : \mathcal{C} \to \mathcal{K}$ is an In-place Update Category.

Proof We only need to check that we can make $J(-, I)$ a strong symmetric monoidal functor; the rest of the structure is taken directly from the structure of J. Define the isomorphisms as:

$$m_{A,B} : J(A, I) \otimes J(B, I) = J(A \times B, I \otimes I) \cong J(A \times B, I) \qquad o : J(1, I) = J(1, I)$$

These define $J(-, I)$ as a strong monoidal functor by the coherence of the symmetric monoidal structure on $\mathcal{S}$. □

By Theorem 5.3.14 and Propositions 5.4.1 and 5.4.5 we have the following corollary:

Corollary 8.5.2 Given a finite product category $\mathcal{C}$ and a symmetric monoidal category $\mathcal{S}$, commutative $\mathcal{S}$-parameterised monoidal strong monads on $\mathcal{C}$ that have the right inverse property and both types of closure also give In-place Update Categories.

This connection between linear types and (extended) monads improves on previous connections established by Benton and Wadler [BW96] and Chen and Hudak [CH97]. In comparison to Benton and Wadler's relationship between commutative monads and models of intuitionistic linear logic, we also require commutativity, but due to the generalised definition of monad we can model state. However, it does not appear that all In-place Update categories are expressible as instances of the above parameterised monad structure, since it is not clear what to choose for the category $\mathcal{S}$. We have also improved on the construction of Chen and Hudak since they only deal with a single linear type, whereas we may have many.

By Theorem 7.1.1 and Proposition 7.1.2, the heap bounded state model of Chapter 7 can interpret λ_{inplc}. We translate the extensions to the Typed Command Calculus presented in Chapter 7 to λ_{inplc}. Firstly, boxed data types. For every state-free type A there is a boxed counterpart $[A]$ that is stateful. There is also a type $\Diamond$, representing unused memory cells. There are two operations:

$$\texttt{store}_A : \Diamond \otimes A \to [A] \qquad \texttt{retrieve}_A : [A] \to \Diamond \otimes [A]$$

The types and operations are interpreted as for the corresponding types and operations in Chapter 7. The operations satisfy the axiom schemes:

$$\texttt{store}_A(\texttt{retrieve}_A e) = e \qquad \texttt{retrieve}_A(\texttt{store}_A e) = e$$

The singly linked list constructors are translated into λ_{inplc} as:

$$\texttt{nil}_A : I \to L(A) \qquad \texttt{cons}_A : A \otimes \Diamond \otimes L(A) \to L(A)$$

where A is a state-free type and $L(A)$ is a new stateful type and we have assumed syntactic sugar for n-ary products. The listfold syntax is translated into the typing rule:

$$\frac{\Gamma_1 \vdash c_1 : B \qquad \Gamma_2, x : B, y : A, d : \Diamond \vdash c_2 : B \qquad \Gamma_3 \vdash c_3 : L(A) \qquad \mathsf{sf}(\Gamma_2)}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathrm{listfold}_{A,B}(c_1, (x, y, d)c_2, c_3) : B}$$

where the type B is the loop invariant. The context Γ_2 must be state-free because it used many times in the loop. These satisfy the following axioms:

$$\mathrm{listfold}_{A,B}(c_1, (x, y, d)c_2, \mathtt{nil}_A \star_I) = c_1$$

$$\begin{array}{c} \mathrm{listfold}_{A,B}(c_1, (x, y', d')c_2, \mathtt{cons}_A(y, d, xs) \\ = \\ \mathrm{let}\ b = \mathrm{listfold}_{A,B}(c_1, (x, y', d')c_2, xs) \\ \mathrm{in}\ c_2[b/x, y/y', d/d'] \end{array}$$

The expression of these constructs is much simpler in λ_{inplc} due to the mixing of stateful and state-free types. As an example, consider the in-place map operation, a translation of one of the examples of Chapter 7:

$$\begin{array}{l} \mathrm{map}_{A,B} = \lambda^{\rightarrow} f : A \rightarrow B . \lambda^{\rightarrow} l : L(A). \\ \quad \mathrm{listfold}(\mathtt{nil}(\star_I), (l : L(B), a : A, d : \Diamond)\mathtt{cons}(f @_{\rightarrow} a, d, l), l) \end{array}$$

Chapter 9

Conclusions

We summarise the contributions of this thesis as follows:

- Formulation of λ_{sep}, an extension of the $\alpha\lambda$-calculus that allows the flexible expression of the separation relationships between members of the context; along with a careful proof of the well-formedness of the equational theory and a type-checking algorithm.

- A categorical semantics for λ_{sep}, for which the calculus is coherent, sound and complete. The semantics is an extension of the Doubly Closed Categories used for the interpretation of the $\alpha\lambda$-calculus.

- An extension of Day's construction of closed symmetric monoidal categories on pre-sheaf categories to the construction of our separation products. This construction allows the embedding of any non-closed separation category in one which is closed. We also presented two examples of this construction that demonstrated how λ_{sep} can be used to model resources and separation "globally" and "locally".

- The definition of Parameterised Freyd categories, a categorical model of typed localised side-effecting computation and their closed variant Typed Command Categories. We also defined the appropriate generalisation of the definition of a strong monad that is equivalent to parameterised Freyd categories.

- The definition of the Typed Command Calculus, a typed λ-calculus that is coherently, soundly and completely modelled by Typed Command Categories. This calculus is the simply typed essence of other type systems in the literature for typed memory management.

- We have given a concrete Typed Command Category based on Day's construction, showing how the calculus may be given an imperative semantics with in-place update and bounded heap.

- An equational theory for a variant of Wadler's linear type system with non-linear types. We have demonstrated that it can be modelled coherently and soundly by commutative Typed Command Categories, thereby explaining its semantics in terms of call-by-value computation and permission management. We have also given a class of models that is complete, as well as coherent and sound.

In addition, we have developed techniques and methods of presentation for the substructural type theories we have developed in this thesis, in particular the use of valid structural transitions, that have proved to be useful. They should also be useful for the metatheory of other substructural and modal type λ-calculi.

The original motivation of this thesis was to provide a foundational type system for in-place update, and to include a description of read-only usage based on [AH02] (see below for information on read-only usage and passivity). We originally thought that greater expression of separation in the type system and applying this to aliasing control would be useful. The result of this is the work presented on λ_{sep}. We now believe that the detailed expression of separation is not as vital as careful control of permissions to access state, as embodied in λ_{inplc} and the Typed Command Calculus. We had originally intended to describe values that were available for in-place update as "separate from everything else in the program". However, it proved to be easier to take the notion of permission as basic and work from there. We have not been able to give an account of read-only access, and we discuss this below with reference to other type systems in the literature that allow read-only access.

9.1 Related Work

We divide the themes of the thesis into three parts, as we did in the introduction chapter, and discuss related and future work. First we cover the concept of substructural type theories that applies to all the systems we have investigated in this thesis. We also refer to other work on substructural type theories that does not directly relate to this thesis. Then we discuss the expression of separation in a type theory and work related to our work on λ_{sep}, and then others' work related to λ_{inplc} and in-place update.

9.1.1 Substructural Typing

The name "Substructural Logics" was apparently first used in Došen's historical survey of such logics [Dos93] to describe logics that do not have a full complement of structural rules. Došen identifies intuitionistic logic as the earliest substructural logic since, from the point of view of Gentzen's sequent calculus [Gen35, Sza69], it is a restriction on the contexts of classical logic. Orlov [Orl28] was an early pioneer who axiomatised a logic without Weakening, now called relevant logic. Gentzen's sequent calculus makes the structural rules explicit and separate from the rules for the connectives, and it is this presentation that substructural type theories take their inspiration from. Substructural Logics and their algebraic and possible world semantics are described in detail in Restall's book [Res00].

Church's original formulation of the λ-calculus [Chu41] also included a variant called λI that did not allow the abstraction of a variable x over a term that did not contain it. By the Curry-Howard isomorphism [How80], we can see that a typed version of this is isomorphic to relevant logic; logic without Weakening.

Lambek [Lam58] defined an extremely basic substructural logic without the rules of Contraction, Exchange or Weakening. He applied this to the study of natural language grammars in linguistics. Type systems without Exchange have been investigated by Polakow and Pfenning [PP99] and applied to ordered memory layout by Petersen *et al* [PHCP03] and to ordered resource usage by Igarashi and Kobayashi [IK02].

Ideas from substructural logics have also been used in elsewhere in computer science to construct spatial logics and type systems that directly express spatial relationships such as the layout of mobile processes [CG03, VCHP04] or memory hierarchies [AJW03]. Hybrid logics lift aspects of the possible worlds structure used to interpret spatial logics into the logic itself [AB01].

Possible worlds semantics for intuitionistic and modal logics was originally developed by Kripke [Kri63c, Kri63b, Kri65, Kri63a], and has also been used to give semantics to substructural logics, see Restall's book for details [Res00]. Possible world semantics have been generalised categorically to (pre-)sheaves and used to give models of higher-order logic [LS88, MM92]. Reynolds and Oles used functor categories (a generalisation of presheaves) to give a semantics to the block structure of Idealised Algol [Rey81, Ole82, Ole85, Ole97]. Day's construction [Day70], that we extended to λ_{sep} in Chapter 4 and used in Chapter 7, is a categorical generalisation of the constructions given for substructural logics in [Res00]. It has also been used to give functor category semantics of Reynolds' Syntactic Control of Interference by O'Hearn and others [O'H93, OPTT99]. Pym, O'Hearn and Yang gave several possible worlds semantics of BI based on Day's construction [POY04].

9.1.2 Separation Typing

We split the related work in this area into several (overlapping) themes and discuss each separately.

Bunched Implications and the $\alpha\lambda$-calculus Pym and O'Hearn describe the Logic of Bunched Implications in [OP99] and Pym goes into more detail in his monograph [Pym02]. The primary innovations of BI are the bunched contexts and the combination of substructural $A \mathrel{-\!\!*} B$ and intuitionistic $A \to B$ implications. These enable BI to model some aspects of resource separation and have been used in Separation Logic [Rey02] as a language for assertions about programs that manipulate pointers.

The $\alpha\lambda$-calculus is the typed λ-calculus derived from BI via the Curry-

Howard isomorphism. It is described in Pym's book [Pym02] and by O'Hearn in [O'H03].

We have already stated that our calculus λ_{sep} is an strict extension of the $\alpha\lambda$-calculus. We have given a translation from the $\alpha\lambda$-calculus to λ_{sep} in Section 2.4.1 that preserves typing and equality. To go further we would like to be able to prove whether or not λ_{sep} is conservative over the $\alpha\lambda$-calculus. We conjecture that it is, but we do not currently have any way of proving it. Possible ways forward are semantic approaches based on embedding a model of the $\alpha\lambda$-calculus in a model of λ_{sep} and using a categorical gluing construction [Cro94]. The construction of a model of λ_{sep} in Section 4.3.3 from the free symmetric monoidal affine category may provide a way of constructing the embedding. Another approach may be to consider first-order $\alpha\lambda$-calculus terms as separation graph inclusions and use the result of [BdGR97] in some way.

Collinson, Pym and Robinson [CPR05] have defined a polymorphic version of the $\alpha\lambda$-calculus. They have extended the calculus with the type abstraction operators $\forall X$ and $\exists X$ of System F [Gir72, Rey74] (see also [GLT89]), as well as new quantifiers $\forall_* X$ and $\exists_* X$ that only quantify over types separated from the rest of the (type variable) context. They give a PER (Partial Equivalence Relation) semantics of their new system. We describe possible future work on a polymorphic extension of λ_{sep} below in Section 9.2.

POMset Logic The idea of augmenting contexts with a relation on the members has also been used in Retoré's POMset Logic [Ret97], an extension of linear logic. POMset (Partially Ordered Multiset) logic extends linear logic by adding a "before" connective $A < B$, such that $A \otimes B \vdash A < B \vdash A \parr B$. This is interpreted, via proof nets, as being possible uni-directional communication, with $\otimes$ as no communication and $\parr$ as possible bi-directional communication. 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. He proves that his extension of proof nets is complete for the coherence space semantics. 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 [Red93], da S. Corrêa, Haeusler and de Paiva [dSCHdP96] as a way of modelling temporal ordering in languages with state. The paper by da S. Corrêa *et al* also describe a semantics based on Dialectica Categories. Reddy's model in [Red94] is the basis for our model of λ_{sep} without function types in Section 3.2.5.

Syntactic Control of Interference and Passivity Reynolds introduced Syntactic Control of Interference in [Rey78, Rey89]. It is essentially the call-by-name imperative language Idealised Algol (IA) [Rey81] with a substructural type system to prohibit interference. Interference in a imperative language such as IA happens when two program phrases interact with a shared piece of store. Programs which work correctly when passed non-interfering arguments may fail when passed arguments that interfere. For example, consider the procedure:

$$\begin{array}{l}\texttt{reverse} = \lambda x, y : \textbf{integer array}. \\ \quad \text{for } i := 0 \text{ to } 49 \text{ do} \\ \quad\quad y[49 - i] := x[i]\end{array}$$

If called as `reverse` z z, then the procedure will not work as advertised; it will leave in z two copies of the original first half, the first one in order and the second reversed. Reynolds' solution to this problem was to restrict the rule of Contraction so that the variable z could not have been used twice, and hence the above bad invocation of `reverse` would be untypable.

In [O'H03], O'Hearn describes an extended version of SCI, SCI+, which uses the more powerful type system of the $\alpha\lambda$-calculus to express the separation required. This fixes some problems with expressing recursive procedures in SCI.

It seems straightforward to extend O'Hearn's SCI+ to use λ_{sep} and thus increase further the flexibility of the type system. However, it is unclear

to us if the extra flexibility is useful in anyway when one is programming in an idiomatic style. The extra flexibility of SCI+ fixes a problem with recursive function definitions, but to our knowledge no-one has demanded the extra flexibility in separation given by SCI+.

Another important part of SCI is the concept of *passivity*. The restriction of all uses of CONTRACTION is often unnecessary when dealing with program phrases that do not write to the store. For example, consider the invocation of `reverse` like so:

$$\begin{array}{l}\text{let } x' = \langle \lambda i.x[i], \lambda i, v.x[i] := \text{if } v < l \text{ then } v \text{ else } l \rangle \\ \text{let } y' = \langle \lambda i.y[i], \lambda i, v.y[i] := \text{if } v < l \text{ then } v \text{ else } l \rangle \\ \quad \texttt{reverse}\ x'\ y'\end{array}$$

The variables x' and y' are bound to arrays using Reynolds' decomposition of imperative variables as pairs of retrieval and update commands [Rey81].

The two identifiers do not write to shared storage – assuming that x and y do not interfere – but they both read from l. The shared l is harmless; it will not cause the operation of `reverse` to be incorrect. Reynolds terms the use of l as "passive" and states that passively used identifiers do not interfere.

The concept of passivity was first described in [Rey78], and he noted some difficulties in integrating it into the type system. In particular the apparently simplest way gives a system which does not enjoy the subject reduction property. This was fixed in a complicated way involving intersection types in [Rey89] and fixed in a simpler way using split contexts in [OPTT99]. O'Hearn discusses the connections between passivity and the "!" modal operator of linear logic in [O'H91]. We believe that passivity in SCI is linked to the investigation of read-only types in linear type systems discussed in the next section. The type system of [OPTT99] should provide inspiration for developing a foundational linear type system with read-only types.

9.1.3 In-place Update and Typed Command Categories

We again divide up related work in this area into several different (overlapping) themes and describe each separately.

Notions of Computation Moggi introduced Computational Monads [Mog91, Mog89a] in order to provide a uniform categorical semantics and metalanguage for programming languages with computational effects. Computational Monads abstract out the common factors of the difference between pure and effectful computations and the sequencing of computations in context common to all computational effects. Power and Robinson [PR97] introduced premonoidal categories as an alternative to monads, as well as κ-categories, an indexed category structure for modelling effectful languages [PT99, PT97, Pow00b].

Monads have also proven extremely useful for incorporating imperative side-effects into pure lazy functional languages such as Haskell [JW93].

Linear Logic Girard [Gir87] introduced Linear Logic, a substructural logic with only the rule of Exchange. This logic has a negation that is self inverse, so that $A^{\perp\perp} \equiv A$, but retains a straightforward constructive reading, unlike classical logic. Linear Logic also introduces a modality "!" that brings back the missing structural rules in a controlled manner. So the sequents $!A \vdash !A \otimes !A$ and $!A \vdash I$ are derivable. The inclusion of both linear and non-linear types in a single system was responsible for the large amount of interest in using systems based on Linear Logic for memory management. We cover this in the section on work related to λ_{inplc} below. See also [Wad93b, Tro93] for further introductions to Linear Logic. The categorical structure of classical linear logic, with the linear negation, had already been investigated by Barr several years before [Bar79].

Abramsky [Abr93] gave operational semantics for calculi based on intuitionistic and classical Linear Logic. Seely [See89] gave a categorical semantics for Linear Logic. The calculus and semantics did not quite match, and this precipitated a large amount of work on good calculi and semantics for linear

logic [Wad93a, Wad92, BBdPH93b, Ben95, BBdPH92, BBdPH93a, Bar96]. In particular the ! modality was a source of problems for constructing a term calculus closed under substitution. Ambler [Amb92] has investigated the categorical semantics of first-order linear logic.

Linear Logic has also proven useful for structuring the semantics of programming languages. The categories used in Domain theory [Plo93, BPR00] have some of the structure of intuitionistic linear logic as does the structure of Game Semantics [AMJ94, AM96], due to Linear Logic's number-of-uses interpretation.

Type systems based on linear logic have also been used for strictness analysis in compilers for lazy functional languages [WJ99].

Linear Logic, Memory Management and Side-effects Lafont [Laf88] noted that a language derived from linear logic has a useful "single-pointer" property that means that an implementation of such a language does not require a garbage collector since re-use of memory may be statically determined by the compiler. The static reuse of memory had already been considered by people such as Darlington and Burstall [DB76], but Linear Logic gave an apparent solid theoretical justification for such optimisations. Baker [Bak92, Bak95] followed this up with Linear Lisp, a language that allowed no aliasing and so had no need of a garbage collector.

Lafont and Baker considered a use of the contraction rule of !'d variables in Linear Logic to be operationally interpreted as copying. Chirimar, Gunter and Riecke [CGR96] presented a different interpretation, based on reference counting. Turner and Wadler [TW99] compare the two approaches. Mackie [Mac94] and Lincoln and Mitchell [LM92] also presented operational interpretations of linear logic with useful memory management properties.

The use of the ! modality for introducing non-linear types was investigated by Wadler [Wad91] and found to be a cause of problems for obtaining good memory management invariants. The use of the dereliction rule means that, at run-time, values of linear and non-linear types can become confused.

Wadler's system presented in [Wad90] split linear and non-linear types and is the main inspiration for the system we presented in Chapter 8, as well as most practical substructural type systems developed subsequently. See [Wal05] and [AFM05] for examples.

The system presented in [Wad90] allows explicit use of side-effects in a functional language via in-place update, as well as implicit static memory management. This was exploited by Hofmann [Hof00] to do programmer controlled explicit memory management. Guzmán and Hudak [GH90] also developed a system for in-place update, called the Single Threaded λ-calculus.

The pure functional programming language Concurrent Clean [NSvEP91] has included a linear-inspired system called Uniqueness types for incorporating side-effects [BS93]. This system appears to be slightly different to Wadler's system that we have investigated in this thesis, due to the possibility of a call-by-name interpretation. Harrington [Har01] has developed a logic and categorical semantics for Uniqueness types, which should allow comparisons between the two systems. The logic programming language Mercury also uses uniqueness assertions to control side-effects [Ove03].

The connection between linear typing and the monadic expression of side-effects has been investigated by Benton and Wadler [BHM02] and Chen and Hudak [CH97]. The presentation of a linear type system in terms of parameterised monads in this thesis gives a better connection since, even though it also requires commutativity as Benton and Wadler do, it widens the range of commutative monads to include state monads, as well as allowing more than the one linear type as allowed by Chen and Hudak's construction. Chen and Hudak's construction of monads from algebras appears to prefigure the construction of computational monads from algebras by Plotkin and Power [PP02]. See below for more information on this subject.

Read-only Types Despite the incorporation of non-linear types in linear type systems, the systems have proven to be much too restrictive in practise. The

primary reason identified by many authors has been that *read-only* uses of variables have been counted as destructive, meaning that the aliasing control is far too strong when one only wishes to have multiple readers on a variable. Wadler [Wad90] identified this in his first paper on linear typing by introducing a special let! expression that allowed a linear value to be aliased in a sub-expression as long as it was only read from.

The idea of read-only variables has been followed up by many others. Aspinall and Hofmann use it in their Aspect-typing system to make the type system of LFPL more flexible [AH02]. Odersky defines observable linear types that also provide a similar flexibility [Ode92] and Kobayashi defines Quasi-linear types to do a similar thing [Kob99]. Walker and Watkins [WW01] use a variant of Wadler's construct for non linear access to regions in their calculus. Fähndrich and DeLine [FD02] describe the solution in the Vault programming language for providing temporary unrestricted access to linear variables.

We consider a good explanation of read-only types essential for a good foundational in-place update type theory and we wish to extend λ_{inplc} to incorporate some form of read-only types based on the insights of the above authors. We believe that read-only types are related to passivity for SCI. This connection has already been investigated in part by O'Hearn [O'H91].

Other approaches to statically controlling side-effects There have been many other approaches to statically controlling side-effects. We mention some of the more influential ones here. Effect Systems [LG88] augment traditional type systems with information about the side-effects caused by a program's execution. Wadler [Wad99] has presented a connection between effect systems and monads indexed by effect types. The difference between the indexed monads presented by Wadler and our parameterised monads is that the indexed monads are indexed by one variable representing the effects encapsulated by that monad, whereas our parameterisation represents the start and finish states of the effectful computation.

Another approach to static memory management is that of Tofte and Talpin's region calculus [TT97]. This statically allocates dynamic program data into a stack of lexically scoped regions. Since the lifetime of a region is determined by the structure of the program code it does not require a garbage collector. Walker and Watkins [WW01] have used linear typing to remove the lexical scoping limitation by using the unique pointer property to allow safe deallocation of regions. Memory management by regions has been integrated into the Cyclone language [JMG+02, GMJ+02].

There have also been many other type systems more loosely based on Linear Logic than the ones described above, including Alias Types [WM00, SWM00], the Capability Calculus [WCM00] and Stateful Views [XZL05]. We describe possible connections to the work in this thesis below in Section 9.2

Moving from type systems to program logics, Separation Logic [Rey02, IO01] is a logic based on Hoare logic [Hoa69, Flo67] that uses BI to make assertions about heaps with pointers. The main feature of Separation Logic that makes it useful for reasoning about programs that manipulate the heap is the *Frame rule*:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}$$

where R does not mention any of the variables modified by C. This allows an assertion about a program C to be placed within a larger separate context. Therefore, one can reason about C locally and then insert it into a proof for a larger program and not have to worry about aliasing. Given our semantics in Chapter 7, we believe that there is a close connection between this rule and the definition of premonoidal structure with respect to $\mathcal{S}$ that we gave in Chapter 5.

Separation Logic has been successfully used to verify complex programs involving pointers [BTSR04] and has recently been extended to concurrent programs [O'H05]. Birkedal, Torp-Smith and Yang [BTSY05] have described a type system for Idealised Algol based on Separation Logic and the

structure of the semantics theyx describe seems very close to our monoidal parameterised monads in Chapter 5.

Algebras and Notions of Computation The inspiration for the results presented in Appendix B was the work of Plotkin and Power on determining monads from computationally natural systems of operations and equations [PP04, PP02]. With Hyland, they have applied this to the problem of combining monads [HPP02] and building up descriptions of programming languages in a modular fashion.

Plotkin and Power's method has been used by Stark [Sta05] to reconstruct previously known fully abstract semantics of the finite π-calculus. Also, the basic idea seems similar to that of Chen and Hudak's construction of monads from operations and axioms for linear datatypes [CH97].

9.2 Some Directions for Future Work

Finally, we discuss some directions for future work that we would like to follow up.

Generalisations of λ_{sep} We consider non-symmetric relations as a variation of λ_{sep}. Right from the start we have assumed that the separation relations S used in λ_{sep} are symmetric. In fact, the results we have proven also go through for non-symmetric relations as well. The existing structural transitions remain the same, and there is an option of adding a new structural transition Transitive, where TC(S) is the transitive closure of S:

$$\frac{\mathrm{S}' \subseteq \mathrm{TC}(\mathrm{S})}{\mathrm{S}(\overrightarrow{\Gamma}) \Rightarrow \mathrm{S}'(\overrightarrow{\Gamma})}\ (\textsc{Transitive})$$

The categorical semantics can be extended by adding an additional natural transformation with the appropriate coherence requirements.

The trickier part comes with trying to thinking of a use for non-symmetric relations. In [Atk04] we attempted to use it for describing allowable information flow, in the context of a security type system [SM03], but this

does not quite work, because there is no notion of control flow in λ_{sep}. Other possible uses may derive from the game semantics sketched below, or other substructural type systems that currently record orderings on context members by restricting the use of the exchange rule.

Categorical approaches to presenting substructural type theories In this thesis we have used a generalised presentation of substructural type theories, specialised to our needs. The presentation of contexts and valid structural transitions, and their interpretation as functors and natural transformations, is obviously related in some way to the presentation of structures on categories as 2-monads on 2-categories. We note the work of Lüth and Ghani on categorical rewriting systems [LG97] and the work of Power, Kelly and others on 2-monads, algebras and their application to substructural type systems and coherence [Pow95, PT05, Rob02, KP93, Pow89].

Polymorphic and Dependent λ_{sep} Extending λ_{sep} to the polymorphism described in [CPR05] would presumably include type quantifiers $\forall_{\text{S}} X_1, ..., X_n$ and $\exists_{\text{S}} X_1, ..., X_n$, where S is a separation relation of size $n+1$. This would match the n-ary function types that λ_{sep} has. Another possible direction of extension for λ_{sep} would be to consider polymorphism over separation relations. For instance, a program may be typable as:

$$\forall S.S(A_1, A_2) \xrightarrow{[]_2} []_2(A_1, A_2)$$

This kind of polymorphism may also require some kind of bounded quantification, such as that found in System $\text{F}_{<:}$ [CMMS94]. It may also be formally similar to the polymorphism over linear and non-linear types found in [WJ99].

To provide a semantics for such a polymorphic calculus, we consider a possible PER semantics. We could take the usual definition of a total combinatory algebra $(A, \cdot, k, s)$ [Lon95] and require a symmetric relation of separation on A, written $x\#y$, such that if $a\#c$ and $b\#c$ then $(a \cdot b)\#c$. The intuition is that the members of A represent computational objects that

carry resources, the separation of which is described by #. We can now use the usual construction to generate the category $PER(A)$ of partial equivalence relations over A and realisable functions, with the slight alteration that an element f of A used to realise an arrow must be resource-free, in the sense that $\forall a.f\#a$. This stops arrows from leaving more resources in the result than were in the input. We can define Separation Products on $PER(A)$:

$$\begin{aligned} & x(\underline{\mathrm{S}}(A_1, ..., A_n))y \\ \Leftrightarrow \quad & \forall i.(\pi_i \cdot x)A_i(\pi_i \cdot y) \wedge (\forall (i,j) \in \mathrm{S}.(\pi_i \cdot x)\#(\pi_j \cdot x) \wedge (\pi_i \cdot y)\#(\pi_j \cdot y)) \end{aligned}$$

where π_i denotes the appropriate projection combinator in A. Separation functions may be defined similarly. For a concrete example of this kind of structure take β-equivalence classes of untyped λ-terms over some set of constants $\mathbb{R}$. The resources contained within each equivalence class is defined as the intersection of all the sets of constants from $\mathbb{R}$ used by the members of the equivalence class. Two equivalence classes are separate if the intersection of their resources is empty.

Going further, we would also like to investigate dependently typed variants of λ_{sep}, following the Bunched Dependent Type Theory of Stark and Schöpp [SS04].

Distinguishing Reference and Permissions A primary limitation of the systems discussed in Chapters 6 and 8 is that they conflate the notions of reference and permission to access that reference. Successful formalisms for reasoning about and safely using complex imperative concepts such as pointers have often separated the notions of reference and permission. Examples include Separation Logic [Rey02] (see in particular [OYR04]), Alias Types [WM00, SWM00], the Capability Calculus [WCM00] and Stateful Views [XZL05].

Following Xi *et al*'s work on Stateful Views, we propose future work based on Xi and Pfenning's Dependent ML [XP99], extending the Typed Command Calculus of Chapter 6. Indexing types by *location* variables in the

style of Alias Types means that we can connect pure value types containing references to state types representing permissions. Judgements would therefore look like:

$$\Theta; \Gamma; \Delta \vdash e : A; S$$

where Γ, Δ, A and S all depend on the location context Θ. Pointers would be represented as singleton value types $\mathrm{ptr}(l)$, as in Alias Types, and permissions by state types $[l \mapsto \mathrm{int}]$, representing the type stored in location l. Storage and retrieval operations would be typed thus:

$$\texttt{store} : \Pi l.(\mathrm{ptr}(l) \times Int, [l \mapsto \Diamond]) \to (1, [l \mapsto \mathrm{Int}])$$

$$\texttt{retrieve} : \Pi l.(\mathrm{ptr}(l), [l \mapsto Int]) \to (Int, [l \mapsto Int])$$

where $\Pi l...$ denotes a dependent product over a location variable. Note that these are types in the value calculus.

We expect that the main advantage of such a system, apart from its flexibility, is that we can extract a traditional imperative program by forgetting all the state parts since pointers are no longer represented by values of state type, as they are in the Typed Command Calculus. Thus the typing essentially is equivalent to a proof in a logic similar to Separation Logic that verifies the safety of the extracted imperative program.

We can foresee one main potential disadvantage of the direct addition of indexed types to the Typed Command Calculus: the expression of existential types. Dependent product types, as used in the `store` and `retrieve` operations above, can be useful just over pure value functions, but it would be very useful for existential packages of value and state to be constructed. One useful instance is an allocation primitive that allocates some memory and returns a pointer to it and a permission. Without existential types we would have to incorporate such an operation as a construct in the language, or in a continuation passing style. Alias Types [SWM00] gets around this problem by building continuation passing style directly into the typing rules.

We expect that such a calculus should be soundly and completely modelled categorical by an application of Lawvere's hyperdoctrines [Law70], using a category of location contexts for indexing. Using the connection between the Typed Command Calculus and λ_{inplc} described in this thesis, we expect that we should also be able to derive a categorical semantics for the language L^3 [MAF05], a language with linear types that separates reference and permission, albeit possibly without L^3's existential types.

Better semantics The concrete model of in-place update presented in Chapter 7 is suitable for a simple model of LFPL [Hof00]. However, it misses several important aspects of real programming languages, such as allocation and deallocation and recursion. In addition, it does not seem suitable for models of the indexed type theory discussed above. Also, we have not related it in any way to the operational semantics of programming languages, which are usually taken as "the" semantics for proving type safety. Ahmed's step indexed models of mutable state [Ahm04], as used in [Ben05] and [MAF05] may provide a good semantics with the required properties.

Algebras and Parameterised Monads We believe it would be worthwhile to follow the work of Plotkin and Power [PP04, PP02] and develop a theory of algebras for parameterised monads suitable for modelling in-place update, extending the preliminary work presented in Appendix B. A crucial first step would be to find a suitable notion of "parameterised" Lawvere theory and prove a similar result to that for non-parameterised enriched Lawvere theories and finitary enriched monads by Power [Pow00a]. An interesting extra twist is provided by the monoidal structure possible on parameterised monads, and it remains to be seen how this would affect a definition of parameterised Lawvere theory.

Appendix A

Proofs for Chapter 5

A.1 Proofs for Theorem 5.2.17

Theorem 5.2.17 states that the categories $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$ and $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$ are equivalent. To show this equivalence we define a functor $F\mathbf{CSPM}(\mathcal{C}, \mathcal{S}) \to \mathbf{CPF}(\mathcal{C}, \mathcal{S})$:

$$\begin{aligned}(T, \eta, \mu, \tau, - \to -, \Lambda)) &\mapsto (\mathcal{C}_T, J_T, [f, c \mapsto f \otimes c; \tau], [c, f \mapsto c \otimes f; \tau'], - \to -, \Lambda) \\ f : T_1 \Rightarrow T_2 &\mapsto (g : (A, S_1) \to (B, S_2)) \mapsto g; f_{S_1, S_2, B}\end{aligned}$$

where τ' is the co-strength defined as $\sigma; \tau; T(S_1, S_2, \sigma)$ and the functor Ff is necessarily always identity on objects.

A.1.1 The functor F is well-defined

We verify that this definition is well-defined on objects by checking each of the required conditions of Definition 5.2.1. We know from the discussion after Definition 5.2.8 that $\mathcal{C}_T$ is a category and J_T is a functor. The mappings of arrows $\oslash$ and $\ominus$ are also functors. On identities:

$$id_A \oslash id_{B,S} = id \otimes \eta; \tau = \eta = id_{(A \times B, S)} \qquad id_{A,S} \ominus id_B =$$

$$\eta \otimes id; \sigma; \tau; T(S, S, \sigma) = \sigma; id \otimes \eta; \tau; T(S, S, \sigma) = \sigma; \eta; T(S, S, \sigma) = \eta = id_{(A \times B, S)}$$

On composition, for $\oslash$:

$$(f; g) \oslash (c_1; c_2)$$

$$\begin{aligned}
&= (f;g) \otimes (c_1; T(S_1, S_2, c_2); \mu); \tau \\
&= f \otimes c_1; g \otimes T(S_1, S_2, c_2); A_3 \otimes \mu; \tau \\
&= f \otimes c_1; g \otimes T(S_1, S_2, c_2); \tau; T(S_1, S_2, \tau); \mu \\
&= f \otimes c_1; \tau; T(S_1, S_2, g \otimes c_2; \tau); \mu \\
&= f \circledcirc c_1; g \circledcirc c_2
\end{aligned}$$

and for $\circledsmile$:

$$\begin{aligned}
& (c_1; c_2) \circledsmile (f; g) \\
= \; & (c_1; T(S_1, S_2, c_2); \mu) \otimes (f; g); \sigma; \tau; T(S_1, S_3, \sigma) \\
= \; & c_1 \otimes g; T(S_1, S_2, c_2) \otimes g; \sigma; A_3 \otimes \mu; \tau; T(S_1, S_3, \sigma) \\
= \; & c_1 \otimes g; T(S_1, S_2, c_2) \otimes g; \sigma; \tau; T(S_1, S_2, \tau); \mu; T(S_1, S_3, \sigma) \\
= \; & c_1 \otimes g; \sigma; \tau; T(S_1, S_2, g \otimes c_2; \tau); \mu; T(S_1, S_3, \sigma) \\
= \; & c_1 \otimes g; \sigma; \tau; T(S_1, S_2, g \otimes c_2; \tau; T(S_2, S_3, \sigma)); \mu \\
= \; & c_1 \otimes g; \sigma; \tau; T(S_1, S_2, \sigma; \sigma; g \otimes c_2; tau; T(S_2, S_3, \sigma)); \mu \\
= \; & c_1 \otimes g; \sigma; \tau; T(S_1, S_2, \sigma); T(S_1, S_2, c_2 \otimes g; \sigma; tau; T(S_2, S_3, \sigma)); \mu \\
= \; & c_1 \circledsmile g; c_2 \circledsmile g
\end{aligned}$$

The functors J_T, $\circledcirc$ and $\circledsmile$ obviously respect the monoidal structure of $\mathcal{C}$ on objects. On arrows, this follows from the naturality of strength and the interaction between the strength and the monad unit:

$$\begin{aligned}
f \circledcirc J_T(g, s) &= f \otimes (\eta; T(S_1, s, g)); \tau \\
&= A_1 \otimes \eta; \tau; T(S_1, s, f \otimes g) \\
&= \eta; T(S_1, s, f \otimes g) \\
&= J_T(f \otimes g, s)
\end{aligned}$$

The case for $\circledsmile$ can be verified in a similar way, using the naturality of the symmetry isomorphisms.

Naturality of $J_T(\sigma, -)$ in the first position:

$$A \circledcirc c; J_T(\sigma, S_2)$$

$$
\begin{aligned}
&= A \otimes c; \tau; T(S_1, S_2, \eta; T(S_2, S_2, \sigma)); \mu \\
&= A \otimes c; \tau; T(S_1, S_2, \sigma) \\
&= \sigma; \sigma; A \otimes c; \tau; T(S_1, S_2, \sigma) \\
&= \sigma; c \otimes A; \sigma; \tau; T(S_1, S_2, \sigma) \\
&= \eta; T(S_1, S_1, \sigma); T(S_1, S_1, c \otimes A; \sigma; \tau; T(S_1, S_2, \sigma)); \mu \\
&= J_T(\sigma, S_1); c \circledcirc A
\end{aligned}
$$

Naturality of $J_T(\sigma, -)$ in the second position:

$$
\begin{aligned}
& c \circledcirc A; J_T(\sigma, S_2) \\
= \; & c \otimes A; \sigma; \tau; T(S_1, S_2, \sigma); T(S_1, S_2, \eta; T(S_2, S_2, \sigma)); \mu \\
= \; & \sigma; A \otimes c; \tau; T(S_1, S_2, \sigma); T(S_1, S_2, \sigma) \\
= \; & \sigma; A \otimes c; \tau \\
= \; & \eta; T(S_1, S_1, \sigma); T(S_1, S_1, A \otimes c; \tau); \mu \\
= \; & J_T(\sigma, S_1); A \circledcirc c
\end{aligned}
$$

Naturality of $J_T(\lambda, -)$:

$$
\begin{aligned}
& I \circledcirc c; J_T(\lambda, S_2) \\
= \; & I \otimes c; \tau; T(S_1, S_2, \eta; T(S_2, S_2, \lambda)); \mu \\
= \; & I \otimes c; \tau; T(S_1, S_2, \lambda) \\
= \; & I \otimes c; \lambda \\
= \; & \lambda; c \\
= \; & \eta; T(S_1, S_1, \lambda); T(S_1, S_1, c); \mu \\
= \; & J_T(\lambda, S_1); c
\end{aligned}
$$

Naturality of $J_T(\rho, -)$:

$$
\begin{aligned}
& c \circledcirc I; J_T(\rho, S_1) \\
= \; & c \otimes I; \sigma; \tau; T(S_1, S_2, \sigma); T(S_1, S_2, \eta; T(S_2, S_2, \rho)); \mu \\
= \; & c \otimes I; \sigma; \tau; T(S_1, S_2, \sigma; \rho) \\
= \; & c \otimes I; \sigma; \tau; T(S_1, S_2, \lambda)
\end{aligned}
$$

$$
\begin{aligned}
&= c \otimes I; \sigma; \lambda \\
&= c \otimes I; \rho \\
&= \rho; c \\
&= \eta; T(S_1, S_1, \rho); T(S_1, S_1, c); \mu \\
&= J_T(\rho, S_1); c
\end{aligned}
$$

Naturality of $J_T(\alpha, -)$ in the first command position:

$$
\begin{aligned}
& (c \circledcirc B) \circledcirc C; J_T(\alpha, id) \\
= \; & (c \otimes B) \otimes C; (\sigma; \tau; T(S_1, S_2, \sigma)) \otimes C; \sigma; \tau; T(S_1, S_2, \sigma); T(S_1, S_2, \eta; T(S_2, S_2, \alpha)); \mu \\
= \; & (c \otimes B) \otimes C; (\sigma; \tau; T(S_1, S_2, \sigma)) \otimes C; \sigma; \tau; T(S_1, S_2, \sigma; \alpha) \\
= \; & (c \otimes B) \otimes C; \sigma \otimes C; \sigma; C \otimes \tau; \tau; T(S_1, S_2, C \otimes \sigma; \sigma; \alpha) \\
= \; & (c \otimes B) \otimes C; \sigma \otimes C; \sigma; C \otimes \tau; \tau; T(S_1, S_2, \alpha^{-1}; \sigma \otimes A; \sigma) \\
= \; & (c \otimes B) \otimes C; \sigma \otimes C; \sigma; \alpha^{-1}; \tau; T(S_1, S_2, \sigma \otimes A; \sigma) \\
= \; & (c \otimes B) \otimes C; \alpha; A \otimes \sigma; \sigma; \tau; T(S_1, S_2, \sigma \otimes A; \sigma) \\
= \; & \alpha; c \otimes (B \otimes C); A \otimes \sigma; \sigma; \tau; T(S_1, S_2, \sigma \otimes A; \sigma) \\
= \; & \alpha; c \otimes (B \otimes C); \sigma; \tau; T(S_1, S_2, \sigma) \\
= \; & \eta; T(S_1, S_1, \alpha); T(S_1, S_2, c \otimes (B \otimes C); \sigma; \tau; T(S_1, S_2, \sigma)); \mu \\
= \; & J_T(\alpha, S_1); c \circledcirc (B \otimes C)
\end{aligned}
$$

In the third command position:

$$
\begin{aligned}
& (A \otimes B) \circledcirc c; J_T(\alpha, S_2) \\
= \; & (A \otimes B) \otimes c; \tau; T(S_1, S_2, \eta; T(S_2, S_2, \alpha)); \mu \\
= \; & (A \otimes B) \otimes c; \tau; T(S_1, S_2, \alpha) \\
= \; & (A \otimes B) \otimes c; \alpha; A \otimes \tau; \tau \\
= \; & \alpha; A \otimes (B \otimes c); A \otimes \tau; \tau \\
= \; & \eta; T(S_1, S_1, \alpha); T(S_1, S_1, A \otimes (B \otimes c); A \otimes \tau; \tau); \mu \\
= \; & J_T(\alpha, S_1); A \circledcirc (B \circledcirc c)
\end{aligned}
$$

In the second command position, using the coherence of the symmetric monoidal

structure of $\mathcal{C}$, and the other naturality properties of $J_T(\alpha, -)$ and $J_T(\sigma, -)$:

$$\begin{aligned} & (A \circledcirc c) \circledcirc C; J_T(\alpha, S_2) \\ = \; & (A \circledcirc c) \circledcirc C; J_T(\sigma \otimes C; \alpha; \sigma^{-1}; \alpha; A \otimes \sigma, S_2) \\ = \; & J_T(\sigma \otimes C; \alpha; \sigma^{-1}; \alpha; A \otimes \sigma, S_1); A \circledcirc (c \circledcirc C) \\ = \; & J_T(\alpha, S_1); A \circledcirc (c \circledcirc C) \end{aligned}$$

Next, we verify that F is well-defined on arrows. Given $f : T_1 \Rightarrow T_2$, an arrow of $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$, then Ff is a functor, preserving identities ($\eta_{1,S,A}; f_{S,S,A} = \eta_{2,S,A}$) and composition:

$$\begin{aligned} & g; T_1 h; \mu_{S_1,S_2,S_3,C}; f_{S_1,S_3,C} \\ = \; & g; T_1 h; f_{S_1,S_2,T(S_2,S_3,C)}; T_2(S_1, S_2, f_{S_2,S_3,C}); \mu_{2,S_1,S_2,S_3,C} \\ = \; & g; f_{S_1,S_2,B}; T_2(S_1, S_2, h; f_{S_2,S_3,C}); \mu_{2,S_1,S_2,S_3,C} \end{aligned}$$

Moreover, it is an arrow of $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$. Firstly, it commutes with J_T:

$$Ff(J_{T_1} g) = Ff(g; \eta_{1,S,B}) = g; \eta_{1,S,B}; f_{S,S,B} = g; \eta_{2,S,B} = J_{T_2} g$$

It commutes with $\circledcirc$:

$$\begin{aligned} & \circledcirc_2(g, Ffc) \\ = \; & \circledcirc_2(g, c; f_{S_1,S_2,B'}) \\ = \; & g \otimes (c; f_{S_1,S_2,B'}); \tau_{2,A',S_1,S_2,B'} \\ = \; & g \otimes c; \tau_{1,A',S_1,S_2,B'}; f_{S_1,S_2,B'} \\ = \; & Ff(g \circledcirc_1 c) \end{aligned}$$

The steps to verify that it commutes with $\circledcirc$ are similar. Finally, we verify that F itself is a functor. Preservation of identities of $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$:

$$F(id)c = c; id = Id(c)$$

and the composition of $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$:

$$F(f; g)c = c; f; g = Fg(c; f) = Ff(Fg(c)) = (Ff; Fg)c$$

So F is a functor.

A.1.2 The functor F is full and faithful

We first prove a simple lemma about functors between Kleisli categories that will be useful for the calculations below.

Lemma A.1.1 (Fact about functors between Kleisli categories) If $f : \mathcal{C}_{T_1} \to \mathcal{C}_{T_2}$ is a functor such that $J_{T_1}; f = J_{T_2}$ then $g; f(id) = f(g)$

Proof By calculation:

$$\begin{aligned} & g; f(id) \\ = \; & g; \eta_2; T_2(S_1, S_1, f(id)); \mu_2 \\ = \; & f(g; \eta_1); T_2(S_1, S_1, f(id)); \mu_2 \\ = \; & f(g; \eta_1; \mu_1) \\ = \; & f(g) \end{aligned}$$

□

Recall that we have defined the supposed inverse to the arrow $f : F(T_1) \to F(T_2)$ as the family of arrows $(F^{-1}f)_{S_1,S_2,A} = f(id_{T_1(S_1,S_2,A)})$. We must show that this family is an arrow of $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$. Firstly, naturality in S_2 and A, for $s : S_2 \to S_2'$ and $g : A \to B$:

$$\begin{aligned} & f(id); T_2(S_1, s, g) \\ = \; & f(id); T_2(S_1, s, g); T_2(S_1, S_2', \eta_2); \mu_2 \\ = \; & f(id); T_2(S_1, S_2, \eta_2; T_2(S_2, s, g)); \mu_2 \\ = \; & f(id); T_2(S_1, S_1, f(\eta_1; T_1(S_2, s, g))); \mu_2 \\ = \; & f(T_1(S_1, S_2, \eta_1; T_1(S_2, s, g)); \mu_1) \\ = \; & f(T_1(S_1, s, g; \eta_1); \mu_1) \\ = \; & f(T_1(S_1, s, g)) \\ = \; & T_1(S_1, s, g); f(id) \end{aligned}$$

Naturality in S_1, for $s : S_1 \to S_1'$:

$$f(id); T_2(s, S_2, A)$$

$$
\begin{aligned}
&= f(id); T_2(s, S_2, A); \eta_2; \mu_2 \\
&= \eta_2; T_2(S_1, S_1, f(id)); T_2(S_1, S_1, T_2(s, S_2, A)); \mu_2 \\
&= \eta_2; T_2(S_1, S_1, f(id)); T_2(S_1, s, T_2(S_1', S_2, A)); \mu_2 \\
&= \eta_2; T_2(S_1, s, T_1(S_1', S_2, A)); T_2(S_1, S_1', f(id)); \mu_2 \\
&= f(\eta_1; T_1(S_1, s, T_1(S_1', S_2, A))); T_2(S_1, S_1', f(id)); \mu_2 \\
&= f(\eta_1; T_1(S_1, s, T_1(S_1', S_2, A)); \mu_1) \\
&= f(\eta_1; T_1(S_1, S_1, T_1(s, S_2, A)); \mu_1) \\
&= f(T_1(s, S_2, A); \eta_1; \mu_1) \\
&= f(T_1(s, S_2, A)) \\
&= T_1(s, S_2, A); f(id)
\end{aligned}
$$

Now we must verify that the defined natural transformation commutes with all the structure of the parameterised monad. Firstly, it commutes with η:

$$\eta_1; f(id) = f(\eta_1) = \eta_2$$

It commutes with μ:

$$
\begin{aligned}
& \mu_1; f(id) \\
= \;& f(\mu_1) \\
= \;& f(id); T_2(S_1, S_2, f(id)); \mu_2
\end{aligned}
$$

Finally, it commutes with τ:

$$
\begin{aligned}
& \tau_1; f(id) \\
= \;& f(\tau_1) \\
= \;& f(A \otimes T_1(S_1, S_2, B); \tau_1) \\
= \;& A \otimes f(id); \tau_2
\end{aligned}
$$

The last remaining task is to show that the two operations are mutually inverse. The following two calculations show this, where the second equality follows from Lemma A.1.1.

$$F^{-1}(Ff) = F^{-1}(g \mapsto g; f) = f$$

$$F(F^{-1}f) = F(f(id)) = g \mapsto g; f(id) = g \mapsto f(g) = f$$

Hence F is full and faithful, as required.

A.1.3 The functor F is essentially surjective

Recall that, given an object $X = (\mathcal{K}, J, \oslash, \oslash, - \to -, \Lambda)$ of $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$ we have defined the object $Y = (T^X, \eta^X, \mu^X, \tau^X, - \to^X -, \Lambda^X)$ as:

$$\begin{aligned}
T^X(S_1, S_2, A) &= (I, S_1) \to (A, S_2) \\
\eta^X_{S,A} &= \Lambda(\rho_A, S) \\
\mu^X_{S_1,S_2,S_3,A} &= \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev}) \\
\tau^X_{A,S_1,S_2,B} &= \Lambda(J(\alpha, S_1); A \oslash \mathrm{ev}) \\
(A, S_1) \to^X (B, S_2) &= (A, S_1) \to (B, S_2) \\
(A, S_1) \to^X f &= (A, S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(f)) \\
\Lambda^X(f) &= \Lambda(J(\rho^{-1}_{A \otimes B}, S_1); \Lambda^{-1}(f))
\end{aligned}$$

We must first show that this really is an object of $\mathbf{CSPM}(\mathcal{C}, \mathcal{S})$. The functorality of T^X follows directly from the functorality of $- \to -$. We check all the (di)naturality conditions of the defined families of arrows. Firstly, naturality of the family $\eta^X_{S,A}$ in A, for $f : A \to A'$:

$$\begin{aligned}
f; \eta^X_{S,A'} &= f; \Lambda(\rho_{A'}, S) \\
&= \Lambda((f \otimes I; \rho_{A'}, S)) \\
&= \Lambda((\rho_{A'}, S); (f, S)) \\
&= \Lambda(\rho_{A'}, S); (I, S) \to (f, S) \\
&= \eta^X_{S,A}; T^X(S, S, f)
\end{aligned}$$

Dinaturality of the family $\eta^X_{S,A}$ in S, for $s : S_1 \to S_2$:

$$\begin{aligned}
\eta^X_{A,S_1}; T^X(S_1, s, A) &= \Lambda(\rho_A, S_1); (I, S_1) \to (A, s) \\
&= \Lambda((\rho_A, S_1); (A, s)) \\
&= \Lambda((A \otimes I, s); (\rho_A, S_2)) \\
&= \Lambda(\rho_A, S_2); (I, s) \to (A, S_2) \\
&= \eta^X_{A,S_2}; T^X(s, S_2, A)
\end{aligned}$$

Naturality of the family $\mu^X_{S_1,S_2,S_3,A}$ in S_1, S_3 and A, for $s_1 : S_1 \to S_1'$, $s_3 : S_3 \to S_3'$ and $f : A \to B$: (we elide the subscripts on the ev and ρ natural transformations to conserve space)

$$\begin{aligned}
& T^X(s_1, S_2, T^X(S_2, s_3, f)); \mu^X_{S_1,S_2,S_3',B} \\
= \;& (I, s_1) \to ((I, S_2) \to (f, s_3), S_2); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda(((I, s_1) \to ((I, S_2) \to (f, s_3), S_2) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda(((I, S_1') \to ((I, S_2) \to (f, s_3), S_2) \otimes I, s_1); \mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda(((I, S_1') \to ((I, S_2) \to (f, s_3), S_2) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}_{,} S_2); \mathrm{ev}); (I, s_1) \to (B, S_3') \\
= \;& \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}; (f, s_3)); (I, s_1) \to (B, S_3') \\
= \;& \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}); (I, s_1) \to (f, s_3) \\
= \;& \mu^X_{S_1',S_2,S_3,A}; T^X(s_1, s_3, f)
\end{aligned}$$

Dinaturality of the family $\mu^X_{S_1,S_2,S_3,A}$ in S_2, for $s_2 : S_2 \to S_2'$: (again we elide the subscripts on ev and ρ)

$$\begin{aligned}
& T^X(S_1, s_2, T^X(S_2', S_3, A)); \mu^X_{S_1,S_2',S_3,A} \\
= \;& (I, S_1) \to ((I, S_2') \to (A, S_3), s_2); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2'); \mathrm{ev}) \\
= \;& \Lambda(((I, S_1) \to ((I, S_2') \to (A, S_3), s_2) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}, S_2'); \mathrm{ev}) \\
= \;& \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); ((I, S_2') \to (A, S_3) \otimes I, s_2); \mathrm{ev}) \\
= \;& \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); ((I, s_2) \to (A, S_3) \otimes I, S_2); \mathrm{ev}) \\
= \;& \Lambda(((I, S_1) \to ((I, s_2) \to (A, S_3), S_2) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& (I, S_1) \to ((I, s_2) \to (A, S_3), S_2); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& T^X(S_1, S_2, T^X(s_2, S_3, A)); \mu^X_{S_1,S_2,S_3,A}
\end{aligned}$$

Naturality of the family $\tau^X_{A,S_1,S_2,B}$ in all variables, for $f : A \to A'$, $s_1 : S_1 \to S_1'$, $s_2 : S_2 \to S_2'$ and $g : B \to B'$:

$$\begin{aligned}
& f \otimes T(s_1, s_2, g); \tau_{A',S_1,S_2',B'} \\
= \;& f \otimes (I, s_1) \to (g, s_2); \Lambda((\alpha, S_1); A' \otimes \mathrm{ev}) \\
= \;& \Lambda(((f \otimes (I, s_1) \to (g, s_2)) \otimes I, S_1); (\alpha, S_1); A' \otimes \mathrm{ev})
\end{aligned}$$

$$
\begin{aligned}
&= \Lambda((\alpha, S_1); (f \otimes ((I, s_1) \to (g, s_2) \otimes I), S_1); A' \circledcirc \mathrm{ev}) \\
&= \Lambda((\alpha, S_1); (A \otimes ((I, s_1) \to (B, S_2) \otimes I), S_1); A \circledcirc \mathrm{ev}; (f \otimes g, s_2)) \\
&= \Lambda((\alpha, S_1); (A \otimes ((I, S_1') \to (B, S_2) \otimes I), s_1); A \circledcirc \mathrm{ev}); (f \otimes g, s_2)) \\
&= \Lambda(((A \otimes (I, S_1') \to (B, S_2)) \otimes I, s_1); (\alpha, S_1); A \circledcirc \mathrm{ev}); (f \otimes g, s_2)) \\
&= \Lambda((\alpha, S_1); A \circledcirc \mathrm{ev}); (I, s_1) \to (f \otimes g, s_2) \\
&= \tau^X_{A,S_1',S_2,B}; T(s_1, s_2, f \otimes g)
\end{aligned}
$$

Now we verify the required axioms of monad structure hold. The first unit property for η^X and μ^X:

$$
\begin{aligned}
& \eta^X_{S_1,T(S_1,S_2,A)}; \mu_{S_1,S_2,S_3,A} \\
&= \Lambda(\rho, S_1); \Lambda(\mathrm{ev}; (\rho^{-1}, S_1); \mathrm{ev}) \\
&= \Lambda((\Lambda(\rho, S_1) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}, S_1); \mathrm{ev}) \\
&= \Lambda(\Lambda^{-1}(\Lambda(\rho, S_1)); (\rho^{-1}, S_1); \mathrm{ev}) \\
&= \Lambda((\rho, S_1); (\rho^{-1}, S_1); \mathrm{ev}) \\
&= id_{T^X(S_1,S_2,A)}
\end{aligned}
$$

The second unit property for η^X and μ^X:

$$
\begin{aligned}
& T^X(S_1, S_2, \eta_{S_2,A}); \mu^X_{S_1,S_2,S_2,A} \\
&= (I, S_1) \to (\Lambda(\rho, S_2), S_2); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
&= \Lambda(((I, S_1) \to (\Lambda(\rho, S_2), S_2) \otimes I); \mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
&= \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); (\Lambda(\rho, S_2) \otimes I, S_2); \mathrm{ev}) \\
&= \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \Lambda^{-1}(\Lambda(\rho, S_2))) \\
&= \Lambda(\mathrm{ev}) \\
&= id_{T^X(S_1,S_2,A)}
\end{aligned}
$$

The associativity property for μ^X:

$$
\begin{aligned}
& T^X(S_1, S_2, \mu_{S_2,S_3,S_4,A}); \mu^X_{S_1,S_2,S_4,A} \\
&= (I, S_1) \to (\Lambda(\mathrm{ev}; (\rho^{-1}, S_3); \mathrm{ev}), S_2); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
&= \Lambda(((I, S_1) \to (\Lambda(\mathrm{ev}; (\rho^{-1}, S_3); \mathrm{ev}), S_2) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev})
\end{aligned}
$$

$$
\begin{aligned}
&= \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); (\Lambda(\mathrm{ev}; (\rho^{-1}, S_3); \mathrm{ev}) \otimes I, S_2); \mathrm{ev}) \\
&= \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}; (\rho^{-1}, S_3); \mathrm{ev}) \\
&= \Lambda(\Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}); \mathrm{ev}; (\rho^{-1}, S_3); \mathrm{ev}) \\
&= \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}); \Lambda(\mathrm{ev}; (\rho^{-1}, S_3); \mathrm{ev}) \\
&= \mu^X_{S_1,S_2,S_3,T(S_3,S_4,A)}; \mu^X_{S_1,S_3,S_4,A}
\end{aligned}
$$

The interaction of τ^X and λ:

$$
\begin{aligned}
& \tau^X_{I,S_1,S_2,A}; T^X(S_1, S_2, \lambda_A) \\
&= \Lambda((\alpha, S_1); I \circledcirc \mathrm{ev}); (I, S_1) \to (\lambda_A, S_2) \\
&= \Lambda((\alpha, S_1); I \circledcirc \mathrm{ev}; (\lambda_A, S_2)) \\
&= \Lambda((\alpha, S_1); (\lambda, S_1); \mathrm{ev}) \\
&= \Lambda((\lambda \otimes I, S_1); \mathrm{ev}) \\
&= \lambda; \Lambda(\mathrm{ev}) \\
&= \lambda
\end{aligned}
$$

The interaction of τ^X and α:

$$
\begin{aligned}
& \alpha_{A,B,T^X(S_1,S_2,C)}; A \otimes \tau^X_{B,S_1,S_2,C}; \tau^X_{A,S_1,S_2,B\otimes C} \\
&= \alpha; A \otimes \Lambda((\alpha, S_1); B \circledcirc \mathrm{ev}); \Lambda((\alpha, S_1); A \circledcirc \mathrm{ev}) \\
&= \alpha; \Lambda(((A \otimes \Lambda((\alpha, S_1); B \circledcirc \mathrm{ev})) \otimes I, S_1); (\alpha, S_1); A \circledcirc \mathrm{ev}) \\
&= \alpha; \Lambda((\alpha, S_1); (A \otimes (\Lambda((\alpha, S_1); B \circledcirc \mathrm{ev}) \otimes I), S_1); A \circledcirc \mathrm{ev}) \\
&= \alpha; \Lambda((\alpha, S_1); (A \otimes \alpha, S_1); A \circledcirc (B \circledcirc \mathrm{ev})) \\
&= \Lambda((\alpha \otimes I, S_1); (\alpha, S_1); (A \otimes \alpha, S_1); A \circledcirc (B \circledcirc \mathrm{ev})) \\
&= \Lambda((\alpha, S_1); (\alpha, S_1); A \circledcirc (B \circledcirc \mathrm{ev})) \\
&= \Lambda((\alpha, S_1); (A \otimes B) \circledcirc \mathrm{ev}; (\alpha, S_2)) \\
&= \Lambda((\alpha, S_1); (A \otimes B) \circledcirc \mathrm{ev}); (I, S_1) \to (\alpha, S_2) \\
&= \tau^X_{A\otimes B,S_1,S_2,C}; T^X(S_1, S_2, \alpha_{A,B,C})
\end{aligned}
$$

The interaction of τ^X and η^X:

$$
A \otimes \eta^X_{S,B}; \tau^X_{A,S,S,B}
$$

$$
\begin{aligned}
&= A \otimes \Lambda((\rho, S)); \Lambda((\alpha, S); A \oslash \mathrm{ev}) \\
&= \Lambda(((A \otimes \Lambda(\rho, S)) \otimes I, S); (\alpha, S); A \oslash \mathrm{ev}) \\
&= \Lambda((\alpha, S); (A \otimes (\Lambda(\rho, S) \otimes I), S); A \oslash \mathrm{ev}) \\
&= \Lambda((\alpha, S); (A \otimes \rho, S)) \\
&= \Lambda((\rho, S)) \\
&= \eta^X_{S, A \otimes B}
\end{aligned}
$$

The interaction of τ^X and μ^X:

$$
\begin{aligned}
& \tau^X_{A,S_1,S_2,T^X(S_2,S_3,B)}; T^X(S_1, S_2, \tau^X_{A,S_2,S_3,B}); \mu^X_{S_1,S_2,S_3,A\otimes B} \\
= \;& \Lambda((\alpha, S_1); A \oslash \mathrm{ev}); (I, S_1) \to (\Lambda((\alpha, S_2); A \oslash \mathrm{ev}), S_2); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda((\alpha, S_1); A \oslash \mathrm{ev}; (\Lambda((\alpha, S_2); A \oslash \mathrm{ev}), S_2)); \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda((\Lambda((\alpha, S_1); A \oslash \mathrm{ev}; (\Lambda((\alpha, S_2); A \oslash \mathrm{ev}), S_2)) \otimes I, S_1); \mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda((\alpha, S_1); A \oslash \mathrm{ev}; (\Lambda((\alpha, S_2); A \oslash \mathrm{ev}), S_2); (\rho^{-1}, S_2); \mathrm{ev}) \\
= \;& \Lambda((\alpha, S_1); A \oslash \mathrm{ev}; (\rho^{-1}, S_2); (\Lambda((\alpha, S_2); A \oslash \mathrm{ev}) \otimes I, S_2); \mathrm{ev}) \\
= \;& \Lambda((\alpha, S_1); A \oslash \mathrm{ev}; (\rho^{-1}, S_2); (\alpha, S_2); A \oslash \mathrm{ev}) \\
= \;& \Lambda((\alpha, S_1); A \oslash \mathrm{ev}; (A \otimes \rho^{-1}, S_2); A \oslash \mathrm{ev}) \\
= \;& \Lambda((\alpha, S_1); (A \otimes (\Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}) \otimes I), S_1); A \oslash \mathrm{ev}) \\
= \;& \Lambda(((A \otimes \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev})) \otimes I, S_1); (\alpha, S_1); A \oslash \mathrm{ev}) \\
= \;& A \otimes \Lambda(\mathrm{ev}; (\rho^{-1}, S_2); \mathrm{ev}); \Lambda((\alpha, S_1); A \oslash \mathrm{ev}) \\
= \;& A \otimes \mu^X_{S_1,S_2,S_3,B}; \tau^X_{A,S_1,S_3,B}
\end{aligned}
$$

Before showing that the definitions for Kleisli exponentials meeting the axioms, we note the following identity for closed parameterised Freyd categories:

$$
\begin{aligned}
& \Lambda^{-1}(f; T^X(S_1, S_2, g); \mu^X) \\
= \;& \Lambda^{-1}(f; (I, S_1) \to (g, S_2); \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev})) \\
= \;& \Lambda^{-1}(f; \Lambda(J((I, S_1) \to (g, S_2) \otimes I, S_1); \mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev})) \\
= \;& \Lambda^{-1}(f; \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); J(g \otimes I, S_2); \mathrm{ev})) \\
= \;& \Lambda^{-1}(f; \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \Lambda^{-1}(g))
\end{aligned}
$$

$$\begin{aligned} &= \Lambda^{-1}(f; \Lambda(\mathrm{ev})); J(\rho^{-1}, S_2); \Lambda^{-1}(g)) \\ &= \Lambda^{-1}(f); J(\rho^{-1}, S_2); \Lambda^{-1}(g) \end{aligned}$$

First we show that the functor $(A, S_1) \to^X -$ is well-defined. On identities we have:

$$\begin{aligned} & (A, S_1) \to^X \eta^X \\ = \; & (A, S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(\Lambda(J(\rho, S_2)))) \\ = \; & (A, S_1) \to (J(\rho^{-1}, S_2); J(\rho, S_2)) \\ = \; & (A, S_1) \to id \\ = \; & id \end{aligned}$$

On composed arrows, by the identity shown above:

$$\begin{aligned} & (A, S_1) \to^X (f; T^X(S_2, S_2', g); \mu^X) \\ = \; & (A, S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(f; T^X(S_2, S_2', g); \mu^X)) \\ = \; & (A, S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(f); J(\rho^{-1}, S_2); \Lambda^{-1}(g)) \\ = \; & (A, S_1) \to^X f; (A, S_1) \to^X g \end{aligned}$$

The map of homsets $\Lambda^X : \mathcal{C}_T(A \otimes_T (B, S_1), (C, S_2)) \to \mathcal{C}(A, (B, S_1) \to (C, S_2))$ is natural in A. For $f : A \otimes B \to T^X(S_1, S_2, C)$ and $g : A' \to A$:

$$\begin{aligned} & g; \Lambda^X(f) \\ = \; & g; \Lambda(J(\rho^{-1}, S_1); \Lambda^{-1}(f)) \\ = \; & \Lambda(J(g \otimes B; \rho^{-1}, S_1); \Lambda^{-1}(f)) \\ = \; & \Lambda(J(\rho^{-1}; g \otimes B \otimes I, S_1); \Lambda^{-1}(f)) \\ = \; & \Lambda(J(\rho^{-1}, S_1); \Lambda^{-1}(g \otimes B; f)) \\ = \; & \Lambda^X(g \otimes B; f) \\ = \; & \Lambda^X(g \otimes B; f; \eta; \mu) \\ = \; & \Lambda^X(g \otimes B; \eta; T(S_1, S_1, f); \mu) \\ = \; & \Lambda^X((g; \eta) \otimes B; \tau; T(S_1, S_1, f); \mu) \\ = \; & \Lambda^X(g \otimes_T (B, S_1); T(S_1, S_1, f); \mu) \end{aligned}$$

The map Λ^X is also natural in (C, S_2). For $g : C \to T^X(S_2, S_2', C')$:

$$\begin{aligned}
& \Lambda^X(f); (B, S_1) \to^X g \\
= \;& \Lambda(J(\rho^{-1}, S_1); \Lambda^{-1}(f)); (B, S_1) \to (J(\rho^{-1}, S_2); \Lambda^{-1}(g)) \\
= \;& \Lambda(J(\rho^{-1}, S_1); \Lambda^{-1}(f); J(\rho^{-1}, S_2); \Lambda^{-1}(g)) \\
= \;& \Lambda(J(\rho^{-1}, S_1); \Lambda^{-1}(f; T^X(S_1, S_2, g); \mu^X)) \\
= \;& \Lambda^X(f; T^X(S_1, S_2, g); \mu^X)
\end{aligned}$$

The map Λ^X is also an isomorphism. The inverse is $\Lambda^{X-1}(f) = \Lambda(J(\rho, S); \Lambda^{-1}(f))$. That these are inverse follows directly from the fact that Λ and Λ^{-1} are inverse and ρ and ρ^{-1} are inverse.

Finally we show that FY is isomorphic to X in $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$. Define two identity on objects functors:

$$\begin{aligned}
K &: \mathcal{C}_{T^X} \to \mathcal{K} \\
Kf &= J(\rho^{-1}, S_1); \Lambda^{-1}(f) \\
L &: \mathcal{K} \to \mathcal{C}_{T^X} \\
Lf &= \Lambda(J(\rho, S_1); f)
\end{aligned}$$

It is easy to see that these are mutually inverse operations on arrows, so it only remains to show that they are actually arrows of $\mathbf{CPF}(\mathcal{C}, \mathcal{S})$. Firstly, they must be functors. They clearly preserve identities and for composition:

$$\begin{aligned}
& K(f; T^X(S_1, S_2, g); \mu) \\
= \;& J(\rho^{-1}, S_1); \Lambda^{-1}(f; (I, S_1) \to J(g, S_2); \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev})) \\
= \;& J(\rho^{-1}, S_1); f \circledS (I, S_1); (I, S_1) \to J(g, S_2) \circledS I; \mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev} \\
= \;& J(\rho^{-1}, S_1); f \circledS (I, S_1); \mathrm{ev}; J(g, S_1); J(\rho^{-1}, S_2); \mathrm{ev} \\
= \;& J(\rho^{-1}, S_1); f \circledS (I, S_1); \mathrm{ev}; J(\rho^{-1}, S_2); g \circledS (I, S_2); \mathrm{ev} \\
= \;& J(\rho^{-1}, S_1); \Lambda^{-1}(f); J(\rho^{-1}, S_2); \Lambda^{-1}(g) \\
= \;& K(f); K(g)
\end{aligned}$$

Since K and L are mutually inverse on arrows this proves that L preserves composition as well. Now we verify that the functor K preserves the parameterised

Freyd structure. This will also imply that L also preserves the structure since they are mutually inverse. For K, preserving the functor from $\mathcal{S} \times \mathcal{C}$:

$$\begin{aligned}
& K(J_{T^X}(s,f)) \\
= \;& K(\eta^X; T^X(S,s,f)) \\
= \;& K(\Lambda(J(\rho,S)); (I,S) \to J(s,f)) \\
= \;& J(\rho^{-1},S); \Lambda^{-1}(\Lambda(J(\rho,S)); (I,S) \to J(s,f)) \\
= \;& J(\rho^{-1},S); \Lambda^{-1}(\Lambda(J(\rho,S))); J(s,f) \\
= \;& J(\rho^{-1},S); J(\rho,S); J(s,f) \\
= \;& J(s,f)
\end{aligned}$$

The functor K preserves $\circledcirc$:

$$\begin{aligned}
& K(f \circledcirc_{T^X} c) \\
= \;& K(f \otimes c; \tau^X) \\
= \;& K(f \otimes c; \Lambda(J(\alpha, S_1); A \circledcirc \mathrm{ev})) \\
= \;& K(\Lambda((f \otimes c) \circledcirc (I,S_1); J(\alpha,S_1); A \circledcirc \mathrm{ev})) \\
= \;& K(\Lambda(J(\alpha,S_1); f \circledcirc J(c \otimes I, S_1); A \circledcirc \mathrm{ev})) \\
= \;& K(\Lambda(J(\alpha,S_1); f \circledcirc \Lambda^{-1}(c)) \\
= \;& J(\rho^{-1},S_1); \Lambda^{-1}(\Lambda(J(\alpha,S_1); f \circledcirc \Lambda^{-1}(c)) \\
= \;& J(\rho^{-1},S_1); J(\alpha,S_1); f \circledcirc \Lambda^{-1}(c) \\
= \;& A \circledcirc J(\rho^{-1},S_1); f \circledcirc \Lambda^{-1}(c) \\
= \;& f \circledcirc Kc
\end{aligned}$$

The proof that it preserves $\circledS$ is similar, and relies on the naturality of $J(\sigma, -)$. So the functor F is essentially surjective, as required.

A.2 Proof of Theorem 5.3.7

We need to verify that the double Freyd-category structure defined from a Power, Robinson, Thielecke Freyd-category is well-defined. We do this first for $\circledcirc_{\mathcal{C}}, \circledS_{\mathcal{C}}$.

Recall that we defined $f \ltimes_{\mathcal{C}} c$ as the composite:

$$\begin{array}{lll} (A \times B, S) & \stackrel{J(id_{A\times B},\lambda)}{\longrightarrow} & (A \times B, I \otimes S) \\ & \stackrel{(A,I)\ltimes c}{\longrightarrow} & (A \times B', I \otimes S') \\ & \stackrel{J(f,id_I)\ltimes(B',S')}{\longrightarrow} & (A' \times B', I \otimes S') \\ & \stackrel{(id_{A'\times B'},\lambda^{-1})}{\longrightarrow} & (A' \times B', S') \end{array}$$

Define $c \rtimes_{\mathcal{C}} f$ as the composite:

$$\begin{array}{lll} (B \times A, S) & \stackrel{J(id_{A\times B},\rho)}{\longrightarrow} & (B \times A, S \otimes I) \\ & \stackrel{(A,I)\ltimes c}{\longrightarrow} & (A \times B', S' \otimes I) \\ & \stackrel{J(f,id_I)\ltimes(B',S')}{\longrightarrow} & (A' \times B', S' \otimes I) \\ & \stackrel{J(id_{A'\times B'},\rho^{-1})}{\longrightarrow} & (A' \times B', S') \end{array}$$

The functors so defined preserve the monoidal structure on arrows:

$$\begin{array}{rcl} f \ltimes_{\mathcal{C}} J(g,s) & = & J(id,\lambda); J(A,I) \ltimes J(g,s); J(f,id) \rtimes J(B',S'); J(id,\lambda^{-1}) \\ & = & J(id,\lambda); J(f \otimes g, id \otimes s); J(id,\lambda^{-1}) \\ & = & (f \otimes g, s); (id, \lambda; \lambda^{-1}) \\ & = & (f \otimes g, s) \end{array}$$

where the second equality is by centrality and the strict premonoidal properties of J.

Naturality of $J(\sigma, S)$ in the first argument:

$$\begin{array}{rcl} & & J(\sigma,S); B \ltimes_{\mathcal{C}} c \\ & = & J(\sigma,S); J(id,\lambda); J(B,I) \ltimes c; J(id,\lambda^{-1}) \\ & = & J(id,\rho); J(\sigma,\sigma); J(B,I) \ltimes c; J(id,\lambda^{-1}) \\ & = & J(id,\rho); c \rtimes J(B,I); J(\sigma,\sigma); J(id,\lambda^{-1}) \\ & = & J(id,\rho); c \rtimes J(B,I); J(id,\rho^{-1}); J(\sigma,S') \\ & = & c \rtimes_{\mathcal{C}} B; (\sigma, S') \end{array}$$

Naturality in the second argument is similar.

Naturality of $J(\lambda, S)$:

$$\begin{aligned} & J(\lambda, S); 1 \circledcirc_{\mathcal{C}} c \\ = \; & J(\lambda, S); J(id, \lambda); J(1, I) \circledcirc c; J(id, \lambda^{-1}) \\ = \; & c; J(\lambda, \lambda); J(id, \lambda^{-1}) \\ = \; & c; (\lambda, S') \end{aligned}$$

Naturality of $J(\rho, S)$ is similar.

Naturality of (α, S) in the middle argument:

$$\begin{aligned} & J(\alpha, S); A \circledcirc_{\mathcal{C}} (c \circledcirc_{\mathcal{C}} B) \\ = \; & J(\alpha, S); J(id, \lambda); J(A, I) \circledcirc (J(id, \rho); c \circledcirc J(B, I); J(id, \rho^{-1})); J(id, \lambda^{-1}) \\ = \; & J(\alpha, S); J(id, \lambda; I \otimes \rho); J(A, I) \circledcirc (c \circledcirc J(B, I)); J(id, I \otimes \rho^{-1}; \lambda^{-1}) \\ = \; & J(id, \lambda; \rho); J(\alpha, \alpha); J(A, I) \circledcirc (c \circledcirc J(B, I)); J(id, I \otimes \rho^{-1}; \lambda^{-1}) \\ = \; & J(id, \lambda; \rho); (J(A, I) \circledcirc c) \circledcirc J(B, I); J(\alpha, \alpha); J(id, I \otimes \rho^{-1}; \lambda^{-1}) \\ = \; & J(id, \lambda; \rho); (J(A, I) \circledcirc c) \circledcirc J(B, I); J(id, \rho^{-1}; \lambda^{-1}); (\alpha, S') \\ = \; & (A \circledcirc_{\mathcal{C}} c) \circledcirc_{\mathcal{C}} B; J(\alpha, S) \end{aligned}$$

The other two naturality equations are proven similarly.

A.3 Proofs for Theorem 5.3.14

A.3.1 The functor F is well-defined

We must show that the definition of F is well-defined on objects and on arrows, and it preserves identities and composition. The bulk of the proof is identical to the proof of the non-monoidal case in Section A.1.1. For objects we need only verify that the defined $\circledcirc^T_{\mathcal{S}}$ and $\circledcirc^T_{\mathcal{S}}$ are well-defined and obey the required axioms for Double Parameterised Freyd categories. Recall that we have defined $s \circledcirc^T_{\mathcal{S}} c = c; \mu_{S_1 \otimes}; T(S_1 \otimes S_2; s \otimes S'_2, B)$. This is a functor. Using naturality and the second commuting diagram for $\mu_{\otimes}$ we obtain:

$$(s_1; s_2) \circledcirc^T_{\mathcal{S}} (c_1; c_2)$$

$$
\begin{aligned}
&= c_1; T(S_2, S_2', c_2); \mu; \mu_{S_1\otimes}; T(S_1 \otimes S_2, (s_1; s_2) \otimes S_2'', C) \\
&= c_1; T(S_2, S_2', c_2); \mu_{S_1\otimes}; T(S_1 \otimes S_2, S_1 \otimes S_2', \mu_{S_1\otimes}); \mu; T(S_1 \otimes S_2, (s_1; s_2) \otimes S_2'', C) \\
&= c_1; \mu_{S_1\otimes}; T(S_1 \otimes S_2, S_1 \otimes S_2', c_2; \mu_{S_1\otimes}); \mu; T(S_1 \otimes S_2, (s_1; s_2) \otimes S_2'', C) \\
&= c_1; \mu_{S_1\otimes}; T(S_1 \otimes S_2, S_1 \otimes S_2', c_2; \mu_{S_1\otimes}; T(S_1 \otimes S_2', (s_1; s_2) \otimes S_2'', C)); \mu \\
&= c_1; \mu_{S_1\otimes}; T(S_1 \otimes S_2, S_1 \otimes S_2', c_2; \mu_{S_1'\otimes}; T(s_1 \otimes S_2', s_2 \otimes S_2'', C)); \mu \\
&= c_1; \mu_{S_1\otimes}; T(S_1 \otimes S_2, s_1 \otimes S_2', B); T(S_1 \otimes S_2, S_1 \otimes S_2', c_2; \mu_{S_1'\otimes}; T(S_1' \otimes S_2', s_2 \otimes S_2'')); \mu \\
&= s_1 \mathbin{\circledcirc}^T_{\mathcal{S}} c_1; s_2 \mathbin{\circledcirc}^T_{\mathcal{S}} c_2
\end{aligned}
$$

as required. That $\circledcirc^T_{\mathcal{S}}$ respects identities is easier to show and follows directly from the fourth commuting diagram for $\mu_{S\otimes}$:

$$id_{S_1} \circledcirc id_{(S_2,A)} = \eta; \mu_{S\otimes} = \eta_{S_1\otimes S_2,A} = id_{(S_1\otimes S_2,A)}$$

The functor $\circledcirc^T_{\mathcal{S}}$ also strictly preserves the monoidal structure on $\mathcal{S}$. On objects this is obvious. On arrows, it follows from the identity law for $\mu_{S\otimes}$ and naturality:

$$
\begin{aligned}
& s_1 \mathbin{\circledcirc}^T_{\mathcal{S}} J_T(f, s_2) \\
= \;& \eta_{S_2,A}; T(S_2, s_2, f); \mu_{S_1\otimes}; T(S_1 \otimes S_2, s_1 \otimes S_2', B) \\
= \;& \eta_{S_2,A}; \mu_{S_1\otimes}; T(S_1 \otimes S_2, s_1 \otimes s_2, B) \\
= \;& \eta_{S_1\otimes S_2,A}; T(S_1 \otimes S_2, s_1 \otimes s_2, B) \\
= \;& J(s_1 \otimes s_2, f)
\end{aligned}
$$

The definition of $c \mathbin{\circledcirc}^T_{\mathcal{S}} s$ as $c; \mu_{\otimes S_2}; T(S_1 \otimes S_2, S_1' \otimes s, B)$ can also be seen to be a functor respecting the monoidal structure on arrows in a similar way.

Finally, the naturality conditions. These each follow from the appropriate diagram. We demonstrate the middle condition for α:

$$
\begin{aligned}
& (s_1 \mathbin{\circledcirc}^T_{\mathcal{S}} c) \mathbin{\circledcirc}^T_{\mathcal{S}} s_2; J(B, \alpha) \\
= \;& (s_1 \mathbin{\circledcirc}^T_{\mathcal{S}} c) \mathbin{\circledcirc}^T_{\mathcal{S}} s_2; T((S_1 \otimes S_2) \otimes S_3, (S_1' \otimes S_2') \otimes S_3', \eta; T((S_1' \otimes S_2') \otimes S_3', \alpha, B)); \mu \\
= \;& (s_1 \mathbin{\circledcirc}^T_{\mathcal{S}} c) \mathbin{\circledcirc}^T_{\mathcal{S}} s_2; T((S_1 \otimes S_2) \otimes S_3, \alpha, B) \\
= \;& \begin{aligned}[t] & c; \mu_{S_1\otimes}; T(S_1 \otimes S_2, s_1 \otimes S_2', B); \mu_{\otimes S_3}; \\ & \quad T((S_1 \otimes S_2) \otimes S_3, (S_1' \otimes S_2') \otimes s_2, B); T((S_1 \otimes S_2) \otimes S_3, \alpha, B) \end{aligned}
\end{aligned}
$$

$$
\begin{aligned}
&= c; \mu_{S_1 \otimes}; \mu_{\otimes S_3}; T((S_1 \otimes S_2) \otimes S_3, (s_1 \otimes S_2') \otimes s_2, B); T((S_1 \otimes S_2) \otimes S_3, \alpha, B) \\
&= c; \mu_{S_1 \otimes}; \mu_{\otimes S_3}; T((S_1 \otimes S_2) \otimes S_3, \alpha, B); T((S_1 \otimes S_2) \otimes S_3, s_1 \otimes (S_2' \otimes s_2), B) \\
&= c; \mu_{\otimes S_3}; \mu_{S_1 \otimes}; T(\alpha, S_1 \otimes (S_2' \otimes S_3), B); T((S_1 \otimes S_2) \otimes S_3, s_1 \otimes (S_2' \otimes s_2), B) \\
&= \begin{array}{l} c; \mu_{\otimes S_3}; T(S_2 \otimes S_3, S_2' \otimes s_2, B); \mu_{S_1 \otimes}; \\ \quad T(S_1 \otimes (S_2' \otimes S_3), s_1 \otimes (S_2' \otimes S_3), B); T(\alpha, S_1' \otimes (S_2' \otimes S_3'), B) \end{array} \\
&= s_1 \ominus_{\mathcal{S}}^{T} (c \oslash_{\mathcal{S}}^{T} s_2); T(\alpha, S_1 \otimes (S_2' \otimes S_3), B) \\
&= s_1 \ominus_{\mathcal{S}}^{T} (c \oslash_{\mathcal{S}}^{T} s_2); \eta; T((S_1 \otimes S_2) \otimes S_3, \alpha, T(S_1 \otimes (S_2 \otimes S_3), S_1' \otimes (S_2' \otimes S_3'), B)); \mu \\
&= \eta; T((S_1 \otimes S_2) \otimes S_3, \alpha, s_1 \ominus_{\mathcal{S}}^{T} (c \oslash_{\mathcal{S}}^{T} s_2)); \mu \\
&= J(A, \alpha); s_1 \ominus_{\mathcal{S}}^{T} (c \oslash_{\mathcal{S}}^{T} s_2)
\end{aligned}
$$

The other naturality conditions are proven similarly, by using the corresponding diagram for the monoidal multiplication. Hence F is well-defined on objects.

To show that F is well-defined on arrows we can re-use the proof in Section A.1.1; we need only verify that the functors Ff also preserve the functors $(\ominus_{\mathcal{S}}^{T}, \oslash_{\mathcal{S}}^{T})$. Commutativity with the $\ominus_{\mathcal{S}}^{T}$:

$$
\begin{aligned}
& \ominus_{\mathcal{S}}^{T_2}(s, Ffc) \\
= \; & \ominus_{\mathcal{S}}^{T_2}(s, c; f_{S_2, S_2', B}) \\
= \; & \eta_{2,S_1,A}; T_2(S_1, s, c; f_{S_2,S_2',B}); \mu_{\otimes,2,S_1,S_1',S_2,S_2',B} \\
= \; & \eta_{1,S_1,A}; f_{S_1,S_1,A}; T_2(S_1, s, c; f_{S_2,S_2',B}); \mu_{\otimes,2,S_1,S_1',S_2,S_2',B} \\
= \; & \eta_{1,S_1,A}; T(S_1, s, c); f_{S_1,S_1',A}; T_2(S_1, S_1', f_{S_2,S_2',B}); \mu_{\otimes,2,S_1,S_1',S_2,S_2',B} \\
= \; & \eta_{1,S_1,A}; T(S_1, s, c); \mu_{\otimes,1,S_1,S_1',S_2,S_2',B}; f_{S_1 \otimes S_2, S_1' \otimes S_2', B} \\
= \; & (s \ominus_{\mathcal{S}}^{T_1} c); f_{S_1 \otimes S_2, S_1' \otimes S_2', B} \\
= \; & Ff(s \ominus_{\mathcal{S}}^{T_1} c)
\end{aligned}
$$

The steps to verify that it commutes with $\oslash_{\mathcal{S}}^{T}$ are similar. Hence F is a functor.

A.3.2 The functor F is full and faithful

Recall that we have defined the supposed inverse to the arrow $f : F(T_1) \to F(T_2)$ as the family of arrows $(F^{-1}f)_{S_1,S_2,A} = f(id_{T_1(S_1,S_2,A)})$. We must show that this family is an arrow of $\mathbf{CSMPM}(\mathcal{C}, \mathcal{S})$ and that the two operations F and F^{-1}

are inverse. The bulk of the proof is identical to the proof for the non-monoidal case in Section A.1.2, we only need to verify that $F^{-1}f$ preserves the monoidal multiplication transformations, which it does by Lemma A.1.1 and the fact that f preserves the $\ominus^T_{\mathcal{S}}$ and $\oslash^T_{\mathcal{S}}$ functors on Kleisli categories:

$$\mu_{S\otimes,1}; f(id) = f(\mu_{S\otimes,1}) = f(id); \mu_{S\otimes,2}$$

The case for $\mu_{\otimes S}$ is similar. Hence F is full and faithful.

A.3.3 The functor F is essentially surjective

Recall that, given an object $X = (\mathcal{K}, J, \ominus_{\mathcal{C}}, \oslash_{\mathcal{C}}, \ominus_{\mathcal{S}}, \oslash_{\mathcal{S}}, - \to -, \Lambda)$ of $\mathbf{CDPF}(\mathcal{C}, \mathcal{S})$ we have defined the object $Y = (T^X, \eta^X, \mu^X, \tau^X, - \to^X -, \Lambda^X)$ as:

$$\begin{aligned}
T^X(S_1, S_2, A) &= (I, S_1) \to (A, S_2) \\
\eta^X_{S,A} &= \Lambda(J(\rho_A, S)) \\
\mu^X_{S_1,S_2,S_3,A} &= \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev}) \\
\tau^X_{A,S_1,S_2,B} &= \Lambda(J(\alpha, S_1); A \ominus_{\mathcal{C}} \mathrm{ev}) \\
\mu^X_{S\otimes} &= \Lambda(S \ominus_{\mathcal{S}} \mathrm{ev}) \\
\mu^X_{\otimes S} &= \Lambda(\mathrm{ev} \oslash_{\mathcal{S}} S) \\
(A, S_1) \to^X (B, S_2) &= (A, S_1) \to (B, S_2) \\
\Lambda^X(f) &= \Lambda(J(\rho^{-1}_{A\otimes B}, S_1); \Lambda^{-1}(f))
\end{aligned}$$

We must first show that this really is an object of $\mathbf{CSMPM}(\mathcal{C}, \mathcal{S})$. We have already done the bulk of the work in Section A.1.3 and it only remains to show that $\mu^X_{S\otimes}$ and $\mu^X_{\otimes S}$ are well-defined.

Firstly, the naturality of $\mu^X_{S\otimes}$:

$$\begin{aligned}
& T^X(s_1, s_2, f); \mu^X_{S\otimes,S_1,S_2',B} \\
= \;& J(I, s_1) \to J(f, s_2); \Lambda(S \ominus_{\mathcal{S}} \mathrm{ev}) \\
= \;& \Lambda(J(J(I, s_1) \to J(f, s_2) \otimes I, S \otimes S_1); S \ominus_{\mathcal{S}} \mathrm{ev}) \\
= \;& \Lambda(S \ominus_{\mathcal{S}} (J(J(I, s_1) \to J(f, s_2) \otimes I, S_1); \mathrm{ev})) \\
= \;& \Lambda(S \ominus_{\mathcal{S}} (J((I, S_1') \to (A, S_2) \otimes I, s_1); \mathrm{ev}; J(f, s_2)))
\end{aligned}$$

$$\begin{aligned}
&= \Lambda(J((I,S_1')\to(A,S_2)\otimes I, S\otimes s_1); S\circledcirc_{\mathcal{S}}\mathrm{ev}; S\circledcirc_{\mathcal{S}} J(f,s_2))\\
&= \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}); J(I,S\otimes s_1)\to J(f,S\otimes s_2)\\
&= \mu^X_{S\otimes,S_1',S_2,A}; T^X(s_1,s_2,f)
\end{aligned}$$

The proof of the naturality of $\mu^X_{\otimes S}$ is similar. Dinaturality of $\mu^X_{S\otimes}$:

$$\begin{aligned}
&\mu^X_{S\otimes,S_1,S_2,A}; T^X(s\otimes S_1, S\otimes S_2, A)\\
&= \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}); J(I,s\otimes S_1)\to(A,S\otimes S_2)\\
&= \Lambda(J((I,S_1)\to(A,S_2),s\otimes S_1); S\circledcirc_{\mathcal{S}}\mathrm{ev})\\
&= \Lambda(S'\circledcirc_{\mathcal{S}}\mathrm{ev}; J(A,s\otimes S_2))\\
&= \Lambda(S'\circledcirc_{\mathcal{S}}\mathrm{ev}); (I,S'\otimes S_1)\to(A,s\otimes S_2)\\
&= \mu^X_{S'\otimes,S_1,S_2,A}; T^X(S'\otimes S_1, s\otimes S_2, A)
\end{aligned}$$

The proof of the dinaturality of $\mu^X_{\otimes S}$ is similar. The defined monoidal multiplication commutes with the normal multiplication:

$$\begin{aligned}
&\mu^X_{S\otimes}; T^X(S\otimes S_1, S\otimes S_2, \mu^X_{S\otimes}); \mu^X\\
&= \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}); (I,S\otimes S_1)\to J(\Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}), S\otimes S_2); \Lambda(\mathrm{ev}; J(\rho^{-1}, S\otimes S_2); \mathrm{ev})\\
&= \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}; J(\Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}), S\otimes S_2)); \Lambda(\mathrm{ev}; J(\rho^{-1}, S\otimes S_2); \mathrm{ev})\\
&= \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}; J(\Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}), S\otimes S_2); J(\rho^{-1}, S\otimes S_2); \mathrm{ev})\\
&= \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev}; J(\rho^{-1}, S\otimes S_2); S\circledcirc_{\mathcal{S}}\mathrm{ev})\\
&= \Lambda(J(\Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev})\otimes I, S\otimes S_1); S\circledcirc_{\mathcal{S}}\mathrm{ev})\\
&= \Lambda(\mathrm{ev}; J(\rho^{-1}, S_2); \mathrm{ev}); \Lambda(S\circledcirc_{\mathcal{S}}\mathrm{ev})\\
&= \mu^X; \mu^X_{S\otimes}
\end{aligned}$$

Again, the proof for $\mu^X_{\otimes S}$ is similar. The defined monoidal multiplication also preserves identities:

$$\begin{aligned}
&\eta^X_{S,A}; \mu^X_{S'\otimes,S,S,A}\\
&= \Lambda(J(\rho,S)); \Lambda(S'\circledcirc_{\mathcal{S}}\mathrm{ev})\\
&= \Lambda(J(\Lambda(\rho,S)\otimes I, S'\otimes S); S'\circledcirc_{\mathcal{S}}\mathrm{ev})\\
&= \Lambda(J(\rho, S'\otimes S))\\
&= \eta^X_{S'\otimes S,A}
\end{aligned}$$

As before, the proof for $\mu^X_{\otimes S}$ is similar. For this to be an object of $\mathbf{CSMPM}(\mathcal{C},\mathcal{S})$ it remains to show that the symmetric monoidal structure diagrams hold. All of these follow from the corresponding naturality condition for the symmetric $\mathcal{S}$-premonoidal structure of X. As an example we show the middle case for associativity:

$$\begin{aligned}
& \mu^X_{\otimes S_3}; \mu^X_{S_1\otimes}; T^X(\alpha, S_1 \otimes (S_2' \otimes S_3), A) \\
= \;& \Lambda(\mathrm{ev} \ogreaterthan_{\mathcal{S}} S_3); \Lambda(S_1 \oless_{\mathcal{S}} \mathrm{ev}); J(I,\alpha) \to (A, S_1 \otimes (S_2' \otimes S_3)) \\
= \;& \Lambda(\Lambda(\mathrm{ev} \ogreaterthan_{\mathcal{S}} S_3) \oless_{\mathcal{C}} (I, S_1 \otimes (S_2 \otimes S_3)); S_1 \oless_{\mathcal{S}} \mathrm{ev}); J(I,\alpha) \to (A, S_1 \otimes (S_2' \otimes S_3)) \\
= \;& \Lambda(S_1 \oless_{\mathcal{S}} J(\Lambda(\mathrm{ev} \ogreaterthan_{\mathcal{S}} S_3) \otimes I, S_2 \otimes S_3); S_1 \oless_{\mathcal{S}} \mathrm{ev}); J(I,\alpha) \to (A, S_1 \otimes (S_2' \otimes S_3)) \\
= \;& \Lambda(S_1 \oless_{\mathcal{S}} (\mathrm{ev} \ogreaterthan_{\mathcal{S}} S_3)); J(I,\alpha) \to (A, S_1 \otimes (S_2' \otimes S_3)) \\
= \;& \Lambda(J(id,\alpha); S_1 \oless_{\mathcal{S}} (\mathrm{ev} \ogreaterthan_{\mathcal{S}} S_3)) \\
= \;& \Lambda((S_1 \oless_{\mathcal{S}} \mathrm{ev}) \ogreaterthan_{\mathcal{S}} S_3; J(A,\alpha)) \\
= \;& \Lambda((S_1 \oless_{\mathcal{S}} \mathrm{ev}) \ogreaterthan_{\mathcal{S}} S_3); (I, (S_1 \otimes S_2) \otimes S_3) \to J(A,\alpha) \\
= \;& \Lambda(S_1 \oless_{\mathcal{S}} \mathrm{ev}); \Lambda(\mathrm{ev} \ogreaterthan_{\mathcal{S}} S_3); (I, (S_1 \otimes S_2) \otimes S_3) \to J(A,\alpha) \\
= \;& \mu^X_{S_1\otimes}; \mu_{\otimes S_3}; T^X((S_1 \otimes S_2) \otimes S_3, \alpha, A)
\end{aligned}$$

Hence Y is an object of $\mathbf{CSMPM}(\mathcal{C},\mathcal{S})$. To finish the proof we must show that FY is isomorphic to X. We re-use the functors K and L defined in Section A.1.3, so we already know that they are mutually inverse and preserve J, $\oless_{\mathcal{C}}$ and $\ogreaterthan_{\mathcal{C}}$. We must show that they preserve $\oless_{\mathcal{S}}$ and $\ogreaterthan_{\mathcal{S}}$, noting that showing that this holds for K implies that it holds for L since they are mutually inverse.

$$\begin{aligned}
& K(s \oless_{\mathcal{S},T^X} c) \\
= \;& K(c; \mu^X_{S_1\otimes}; T^X(S_1 \otimes S_2, s \otimes S_2', B)) \\
= \;& K(c; \Lambda(S_1 \oless_{\mathcal{S}} \mathrm{ev}); (I, S_1 \otimes S_2) \to J(B, s \otimes S_2')) \\
= \;& K(c; \Lambda(s \oless_{\mathcal{S}} \mathrm{ev})) \\
= \;& K(\Lambda(c \oless (I, S_1 \otimes S_2); s \oless_{\mathcal{S}} \mathrm{ev})) \\
= \;& K(\Lambda(s \oless_{\mathcal{S}} \Lambda^{-1}(c))) \\
= \;& J(\rho^{-1}, S_1 \otimes S_2); \Lambda^{-1}(\Lambda(s \oless_{\mathcal{S}} \Lambda^{-1}(c))) \\
= \;& J(\rho^{-1}, S_1 \otimes S_2); s \oless_{\mathcal{S}} \Lambda^{-1}(c)
\end{aligned}$$

$$
\begin{aligned}
&= s \oslash_{\mathcal{S}} (J(\rho^{-1}, S_2); \Lambda^{-1}(c)) \\
&= s \oslash_{\mathcal{S}} Kc
\end{aligned}
$$

The case for $\oslash_{\mathcal{S}}$ is similar. Hence F is essentially surjective.

Appendix B

Adjunctions and Algebras with Parameters

In this appendix we explore the connection between parameterised monads and parameterised adjunctions. This appendix presents very preliminary work on the connection between parameterised monads and adjunctions, and especially between parameterised monads and algebras.

B.1 Parameterised Adjunctions

Parameterised adjunctions are families of normal adjunctions indexed by some category $\mathcal{S}$, subject to a naturality constraint. The relationship between parameterised monads and parameterised adjunctions is very similar to the relationship between monads and adjunctions. We start by defining parameterised adjunctions in their own right and showing how every parameterised adjunction gives a parameterised monad. Conversely, each parameterised monad gives two canonical parameterised adjunctions which determine the monad and are initial and terminal in the category of adjunctions determining the monad.

The terminal adjunction determined by a parameterised monad relates the category $\mathcal{C}$ to a category $\mathcal{C}^T$ of algebras for the monad. At the end of this section we will give a category of "typed state" algebras and show that it is equivalent to

the category of algebras for the typed global state monad in Example 5.2.11. This points to the possible development of a methodology for deriving parameterised monads for modelling computational effects from algebras in a similar manner to Plotkin and Power's [PP02]. Eventually we would like to have a close connection between algebras and parameterised monads, similar to the case described for (enriched) monads by Robinson [Rob02].

The development in this section closely follows the development of the relationship between non-parameterised monads and adjunctions in Mac Lane [Mac98] §VI.1-5. Sometimes we have only sketched some details in the proofs where they are similar to the case for non-parameterised monads. See also Barr and Wells [BW83] for more information on monads (where they are called triples).

Definition B.1.1 Given categories $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{D}$, an $\mathcal{S}$-parameterised adjunction from $\mathcal{C}$ to $\mathcal{D}$ is a triple $\langle F, G, \psi \rangle : \mathcal{C} \to \mathcal{D}$ where F and G are functors:

$$F : \mathcal{S} \times \mathcal{C} \to \mathcal{D} \qquad\qquad G : \mathcal{S}^{\mathsf{op}} \times \mathcal{D} \to \mathcal{C}$$

and ψ is an isomorphism of homsets, natural in A, B and S:

$$\psi : \mathcal{D}(F(S, A), B) \cong \mathcal{C}(A, G(S, B))$$

By Theorem §IV.7.3 in [Mac98], if we have a functor $F : \mathcal{S} \times \mathcal{C} \to \mathcal{D}$ such that for every object S, $F(S, -)$ has a right adjoint $G_S : \mathcal{D} \to \mathcal{C}$, then there is a unique way to make G into a bifunctor $\mathcal{S}^{\mathsf{op}} \times \mathcal{D} \to \mathcal{C}$ such that it is a parameterised adjunction in the sense of this definition.

As with non-parameterised adjunctions, we can express a parameterised adjunction in terms of a unit and counit:

$$\eta_{S,A} : A \to G(S, F(S, A)) \qquad\qquad \epsilon_{S,A} : F(S, G(S, A)) \to A$$

such that the are both natural in A, dinatural in S and obey the standard triangular identities. These are derived, as in non-parameterised adjunctions, as $\eta_{S,A} = \psi(id_{F(S,A)})$ and $\epsilon_{S,A} = \psi^{-1}(id_{G(S,A)})$. The fact that ψ is natural in S ensures that these are dinatural in S. Conversely, given a unit and counit we

can define $\psi f = \eta_{S,A}; G(S, f)$ and $\psi^{-1} f = F(S, g); \epsilon_{S,B}$. We will use the two presentations of parameterised adjunctions interchangeably.

Our interest in parameterised adjunctions lies in the fact that they are to parameterised monads as adjunctions are to monads. Firstly, every parameterised adjunction gives a parameterised monad in the same manner that an adjunction gives a monad:

Theorem B.1.2 Every $\mathcal{S}$-parameterised adjunction $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}$ gives an $\mathcal{S}$-parameterised monad on $\mathcal{C}$, defined as:

$$T(S_1, S_2, A) = G(S_1, F(S_2, A)) \qquad \eta^T_{S,A} = \eta_{S,A} \qquad \mu^T_{S_1,S_2,S_3,A} = G(S_1, \epsilon_{S_2,F(S_3,A)})$$

Proof The naturality and dinaturality of η^T and μ^T follow directly from η and ϵ's properties as unit and counit of an adjunction. The associativity law for the parameterised monad follows from the naturality of ϵ and the left and right unit laws follow from the triangular identities for the adjunction. □

In the opposite direction, from monads to adjunctions, we have the same situation as for non-parameterised monads: there are two canonical adjunctions arising from a parameterised monad, the initial and terminal objects in the category of adjunctions that define the monad.

First we define the category of adjunctions that we are interested in.

Definition B.1.3 Given an $\mathcal{S}$-parameterised monad (T, η, μ) on a category $\mathcal{C}$, the category $\mathbf{PAdj}(T)$ is defined as:

Objects $\mathcal{S}$-parameterised adjunctions $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}$ that define the monad (T, η, μ);

Arrows An arrow $K : (\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}) \to (\langle F', G', \eta, \epsilon' \rangle : \mathcal{C} \to \mathcal{D}')$ is a functor $K : \mathcal{D} \to \mathcal{D}'$ such that $G = Id \times K; G'$, $F' = F; K$ and $\epsilon'_{S,KA} = K\epsilon_{S,A}$.

Note that, by the condition that all the adjunctions form the same parameterised monad, all objects of this category have the same unit. The definition of arrow is derived from the standard definition of a transformation of adjoints,

extended to parameterised adjunctions and specialised to those that define the same monad.

Theorem B.1.4 Given an $\mathcal{S}$-parameterised monad (T, η, μ) on a category $\mathcal{C}$, the functor $J_T : \mathcal{S} \times \mathcal{C} \to \mathcal{C}_T$ defined in Definition 5.2.8 has a right adjoint $G_T : \mathcal{S}^{\mathsf{op}} \times \mathcal{C}_T \to \mathcal{C}$ such that (T, η, μ) is the parameterised monad determined by the adjunction. This adjunction is initial in $\mathbf{PAdj}(T)$.

Proof Define G_T as $G_T(S_1, (A, S_2)) = T(S_1, S_2, A)$ on objects and, for arrows $s : S_1 \to S_1'$ and $f : (A, S_2) \to (B, S_2')$, as $G_T(s, f) = T(s, S_2, f); \mu_{S_1,S_2,S_2',B}$ on arrows. Functorality of G_T can be checked by routine calculation. It is trivially the right adjoint to J_T since:

$$\mathcal{C}_T(J_T A, B) = \mathcal{C}_T(A, B) = \mathcal{C}(A, TB) = \mathcal{C}(A, G_T B)$$

The functor part of the monad derived from this adjunction is the same as the original monad: $G_T(S_1, J_T(S_2, A)) = TA$ on objects and $G_T(s_1, J_T(s_2, f)) = T(s_1, S_2, \eta_{S_2,A}; T(S_2, s_2, f)); \mu_{S_1,S_2,S_2',B} = T(s_1, s_2, f; \eta_{S_2',B}); \mu_{S_1,S_2',S_2',B} = T(s_1, s_2, f)$ by naturality, dinaturality and the unit monad laws on arrows. By the definition of identities in $\mathcal{C}_T$, the unit of the monad derived from the adjunction is η, the unit of the original monad. The multiplication of the derived monad is $\mu'_{S_1,S_2,S_3,A} = G(S_1, \epsilon_{T,S_2,F(S_3,A)})$, where $\epsilon_{T,S_1,(A,S_2)} = id_{T(S_1,S_2,A)}$, in $\mathcal{C}_T$, therefore $\mu' = \mu$ by the definition of G_T.

This parameterised adjunction is initial in the category $\mathbf{PAdj}(T)$. Given another parameterised adjunction $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}$ that defines the parameterised monad T, define a functor $K : \mathcal{C}_T \to \mathcal{D}$ as $K(A, S) = F(S, A)$ and $Kf = F(S_1, f); \epsilon_{S_1,F(S_2,B)}$. This is an arrow in $\mathbf{PAdj}(T)$:

$$G(S_1, K(A, S_2)) = G(S_1, F(S_2, A)) = G_T(S_1, (A, S_2))$$

$$G(s, K(f)) = G(s, F(S_2, f)); G(S_1, \epsilon_{S_2,F(S_2',B)}) = G_T(s, f)$$

$$K(J_T(S, A)) = K(A, S) = F(S, A)$$

$$K(J_T(s, f)) = K(\eta_{S_1,A}; T(S_1, s, f)) = F(s, f; \eta_{S_2,B}); \mu_{S_1,S_2,S_2,B} = F(s, f)$$

$$K(\epsilon_{T,S,A}) = F(S, id_{T(S,S,A)}); \epsilon_{S,F(S,A)} = \epsilon_{K(A,S)}$$

Any other $\mathbf{PAdj}(T)$ arrow $L : \mathcal{C}_T \to \mathcal{D}$ is equal to K. On objects $L(A,S) = LJ_T(A,S) = F(S,A) = K(A,S)$ and on arrows:

$$\begin{aligned} Kf &= F(S_1, f); \epsilon_{S_1, F(S_2,B)} \\ &= F(S_1, f); \epsilon_{S_1, L(B,S_2)} \\ &= L(J_T(S_1, f); \epsilon_{T,S_1,(B,S_2)}) \\ &= L(f; \eta_{S_1,S_1,B}; \mu_{S_1,S_1,S_2,B}) \\ &= Lf \end{aligned}$$

Hence K is the unique $\mathbf{PAdj}(T)$ arrow $\mathcal{C}_T \to \mathcal{D}$ and $\mathcal{C}_T$ is initial. □

The second canonical parameterised adjunction that arises from a parameterised monad is the parameterised version of the Eilenberg-Moore category of algebras for the monad. This definition is the generalisation of the standard definition of T-algebra to the parameterised setting.

Definition B.1.5 Given an $\mathcal{S}$-parameterised monad (T, η, μ) on a category $\mathcal{C}$, the Eilenberg-Moore category of algebras $\mathcal{C}^T$ is defined as:

Objects T-algebras: $\langle A : \mathcal{S}^{\mathsf{op}} \to \mathcal{C}, h_{S_1,S_2} : T(S_1, S_2, AS_2) \to AS_1 \rangle$, where the family h is natural in S_1 and dinatural in S_2 and satisfies these two diagrams:

$$\begin{array}{ccc} T(S_1, S_2, T(S_2, S_3, AS_3)) & \xrightarrow{T(S_1,S_2,h_{S_2,S_3})} & T(S_1, S_2, AS_2) \\ \downarrow{\scriptstyle \mu_{S_1,S_2,S_3}} & & \downarrow{\scriptstyle h_{S_1,S_2}} \\ T(S_1, S_3, AS_3) & \xrightarrow{h_{S_1,S_3}} & AS_1 \end{array}$$

$$\begin{array}{ccc} AS & \xrightarrow{\eta_S} & T(S, S, AS) \\ & \searrow^{AS} & \downarrow{\scriptstyle h_{S,S}} \\ & & AS \end{array}$$

Arrows An arrow $f : \langle A, h \rangle \to \langle A', h' \rangle$ is a natural transformation $f : A \Rightarrow A'$

such that this diagram commutes:

$$\begin{array}{ccc} T(S_1, S_2, AS_2) & \xrightarrow{T(S_1,S_2,f_{S_2})} & T(S_1, S_2, A'S_2) \\ \downarrow{\scriptstyle h_{S_1,S_2}} & & \downarrow{\scriptstyle h'_{S_1,S_2}} \\ AS_1 & \xrightarrow{f_{S_1}} & A'S_1 \end{array}$$

This definition clearly defines a category: the composition of two arrows is an arrow of the category by putting the two commuting squares side by side. The next theorem relates this category to the original category $\mathcal{C}$ by an adjunction which is terminal in $\mathbf{PAdj}(T)$.

Theorem B.1.6 Given an $\mathcal{S}$-parameterised monad (T, η, μ), the functors

$$\begin{array}{rcl} F^T & : & \mathcal{S} \times \mathcal{C} \to \mathcal{C}^T \\ F^T(S, A) & = & \langle T(-, S, A), \mu_{S_1,S_2,S} \rangle \\ F^T(s, f) & = & T(-, s, f) \\ G^T & : & \mathcal{S}^{\mathsf{op}} \times \mathcal{C}^T \to \mathcal{C} \\ G^T(S, \langle A, h \rangle) & = & AS \\ G^T(s, f) & = & As; f_{S_1} \end{array}$$

form a parameterised adjunction, with unit η and counit $\epsilon_{S,\langle A,h\rangle} = h_{-,S}$, whose monad is (T, η, μ). This adjunction is terminal in $\mathbf{PAdj}(T)$.

Proof The object part of F^T is well defined: the map $\mu_{S_1,S_2,S}$ satisfies the two diagrams for algebras by the associativity and left unit laws for monads. The arrow part is well defined by the naturality of μ. Also, F^T clearly preserves composition and identities. The definition of G^T also gives a functor: it is trivially well-defined on objects and arrows, and preserves composition by the naturality of arrows in $\mathcal{C}^T$.

The unit is natural and dinatural since it is the unit of a parameterised monad. The defined counit is natural and dinatural in the appropriate variables by the definition of algebras. The unit and counit satisfy the triangular identities by the

right unit law for a parameterised monad and the unit law for an algebra. Hence $\langle F^T, G^T, \eta, \epsilon^T \rangle$ is a parameterised adjunction.

This parameterised adjunction determines the monad T. The units are the same, as are the functors: $G^T(S_1, F^T(S_2, A)) = T(S_1, S_2, A)$ and the multiplication:

$$\mu^T_{S_1,S_2,S_3,A} = G^T(S_1, \epsilon_{S_2,F^T(S_3,A)}) = G^T(S_1, \epsilon_{S_2,\langle T(-,S_3,A),\mu_{-,-,S_3,A}\rangle}) =$$

$$G^T(S_1, \mu_{-,S_2,S_3,A}) = \mu_{S_1,S_2,S_3,A}$$

Given any other parameterised adjunction $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}$ in $\mathbf{PAdj}(T)$, define the functor $K : \mathcal{D} \to \mathcal{C}^T$ as $KA = \langle G(-, A), G(S_1, \epsilon_{S_2,A}) \rangle$ and $Kf = G(-, f)$. This is well-defined on objects since the first diagram for algebras commutes by naturality and the second is just one of the triangular identities for a parameterised adjunction. It is well-defined on arrows because ϵ is a natural transformation. The functor K is an arrow of $\mathbf{PAdj}(T)$:

$$G^T(S, KA) = G^T(S, \langle G(-, A), G(S_1, \epsilon_{S_2,A}) \rangle) = G(S, A)$$

$$G^T(s, Kf) = G^T(s, G(-, f)) = G(s, A); G(S_2, f) = G(s, f) \qquad K(F(S, A)) =$$

$$\langle G(-, F(S, A)), G(S_1, \epsilon_{S_2,F(S,A)}) \rangle = \langle T(-, S, A), \mu_{S_1,S_2,S,A} \rangle = F^T(S, A)$$

$$K(F(s, f)) = G(-, F(s, f)) = T(-, s, f) = F^T(s, f)$$

$$\epsilon^T_{S,KA} = \epsilon^T_{S,\langle G(-,A),G(S_1,\epsilon_{S_2,A})\rangle} = G(-, \epsilon_{S,A}) = K\epsilon_{S,A}$$

Given another arrow of $\mathbf{PAdj}(T)$, $L : \mathcal{D} \to \mathcal{C}^T$, it is equal to K: the equation $Id \times L; G^T = G$ implies that $LA = \langle G(-, A), h \rangle$, for some h and $Lf = G(-, f)$. For the arrow h:

$$h_{S_1,S_2} = \epsilon^T_{S_2,\langle G(-,A),h\rangle,S_1} = \epsilon^T_{S_2,LA} = (L\epsilon_{S_2,A})_{S_1} = G(S_1, \epsilon_{S_2,A})$$

Hence K is the unique arrow to $\mathcal{C}^T$ and $\mathcal{C}^T$ is terminal. □

B.2 Typed State Algebras

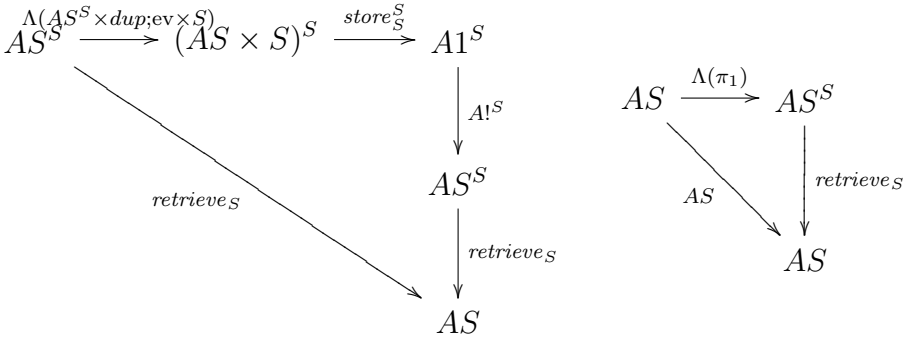
We will now show that a suitable category of "typed state algebras" is equivalent to the category of algebras for the typed global state monad on a cartesian closed

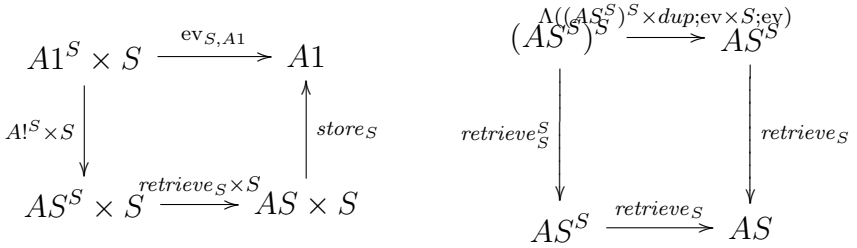
category in Example 5.2.11.

Let $\mathcal{C}$ be a cartesian closed category and let $\mathcal{S}$ be a category with a chosen terminal object and $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$ a functor that preserves the terminal object. We will write 1 for the chosen terminal objects in $\mathcal{S}$ and $\mathcal{C}$ and ! for the unique maps to them. Given such a pair of categories we define the category of typed state algebras over it.

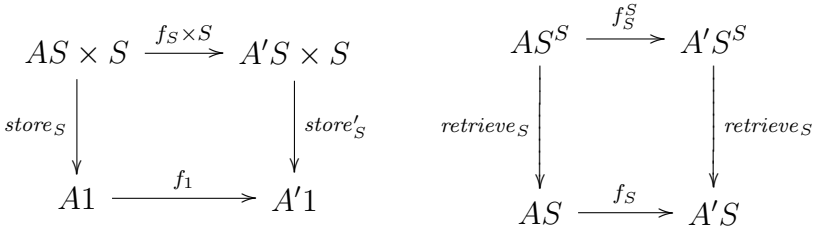
Definition B.2.1 The category $\mathbf{StAlg}(\mathcal{C}, \mathcal{S})$ is defined as:

Objects Triples $\langle A : \mathcal{S}^{\mathsf{op}} \to \mathcal{C}, store_S : AS \times \widehat{S} \to A1, retrieve_S : AS^{\widehat{S}} \to AS \rangle$, such that *store* is dinatural in S and *retrieve* is natural in S and the following diagrams commute:





Arrows An arrow $f : \langle A, store, retrieve \rangle \to \langle A', store', retrieve' \rangle$ is a natural transformation $f : A \Rightarrow A'$ that preserves the operations:



We informally justify the axioms of these algebras by thinking of the operations of the algebras as operations in a typed programming language. We do not have a formal link between parameterised algebraic operations and arrows in the Kleisli category as Plotkin and Power do [PP01], but can intuitively justify the axioms anyway. The *store* and *retrieve* operations correspond to primitives in the Typed Command Calculus of Chapter 6 typed as so:

$$\texttt{store}_S : (\widehat{S}, 1) \to (1, S) \qquad \texttt{retrieve}_S : (1, S) \to (\widehat{S}, S)$$

and the map to the terminal object in $\mathcal{S}$ corresponds to a term $\texttt{clear} : S \to I$ in the state calculus. The four axioms correspond to the following equations between terms:

$$\begin{array}{l} \text{let } (x; s_1) = \texttt{retrieve}(\star_1; s) \text{ in} \\ \text{let } (d; s_2) = \texttt{store}(x; \texttt{clear } s_1) \text{ in} \\ \quad (x; s_2) \end{array} \quad = \quad \texttt{retrieve}$$

$$\text{let } (x; s) = \texttt{retrieve}(\star_1; s) \text{ in } e \quad = \quad e \quad (x \text{ not free in e})$$

$$\begin{array}{l} \text{let } (d; s_1) = \texttt{store}(y; s) \text{ in} \\ \text{let } (x; s_2) = \texttt{retrieve}(\star_1; s_1) \text{ in} \\ \text{let } (d; s_3) = (\star_1; \texttt{clear } s_3) \text{ in} \\ \quad e \end{array} \quad = \quad e[y/x, s/s_3]$$

$$\begin{array}{l} \text{let } (x; s_1) = \texttt{retrieve}(\star_1; s) \text{ in} \\ \text{let } (y; s_2) = \texttt{retrieve}(\star_1; s_1) \text{ in} \\ \quad e \end{array} \quad = \quad \begin{array}{l} \text{let } (x; s_1) = \texttt{retrieve}(\star_1; s) \text{ in} \\ \quad e[x/y, s_1/s_2] \end{array}$$

In order, these equations state that retrieving and then storing is the same as just retrieving; retrieving and then discarding is the same as doing nothing; storing, retrieving and then clearing is the same as doing nothing; and retrieving twice is the same as retrieving once and using twice.

Ideally, we would now apply the parameterised analogue of Beck's Theorem ([Mac98] §VI.7 and [Bec67]) to show that this category is isomorphic to the Eilenberg-Moore category for the typed global state monad, i.e. that it is monadic

over $\mathcal{C}$. However, we do not have such a theorem at this time, so we prove the desired result by hand:

Theorem B.2.2 The functor $G : \mathcal{S}^{\mathsf{op}} \times \mathbf{StAlg}(\mathcal{C}, \mathcal{S}) \to \mathcal{C}$, defined as:

$$\begin{aligned} G(S, \langle A, store, retrieve\rangle) &= AS \\ G(s, f) &= As; f_{S_1} \end{aligned}$$

has a left parameterised adjoint which determines the typed global state monad $T(S_1, S_2, A) = (A \times \widehat{S_2})^{\widehat{S_1}}$. Moreover, the Eilenberg-Moore category of this monad is isomorphic to $\mathbf{StAlg}(\mathcal{C}, \mathcal{S})$.

Proof The tedious calculation in this proof has been placed at the end of this appendix, in Section B.3. Define the functor $F : \mathcal{S} \times \mathcal{C} \to \mathbf{StAlg}(\mathcal{C}, \mathcal{S})$ as:

$$\begin{aligned} F(S, A) &= \langle (A \times S)^-, st, rt\rangle \\ F(s, f) &= (f \times s)^- \end{aligned}$$

where:

$$st = \Lambda(\pi_1; \mathrm{ev}_{S_1,(A\times S)}) \qquad rt = \Lambda(((A\times S)^{S_1})^{S_1} \times dup; \mathrm{ev}_{S_1,(A\times S)^{S_1}} \times S_1; \mathrm{ev}_{S_1,(A\times S_1)})$$

This definition is well-defined on objects and arrows, see Section B.3.1. Before we show that it is the parameterised left adjoint, recall the unit and multiplication of the parameterised monad:

$$\eta_{S,A} = \Lambda(id) : A \to (A \times S)^S$$

$$\mu_{S_1,S_2,S_3,A} = \mathrm{ev}^{S_1}_{S_2,(A\times S_3)} : ((A \times S_3)^{S_2} \times S_2)^{S_1} \to (A \times S_3)^{S_1}$$

Take the unit of the parameterised adjunction to be the same as the unit of the parameterised monad; this is well-defined since $G(S_1, F(S_1, A)) = T(S_1, S_2, A)$ as defined and we already know that this unit is appropriately natural and dinatural. Define the counit to be:

$$\epsilon_{S,\langle A,store,retrieve\rangle,S'} = store^{S'}_S; A!^{S'}; retrieve_{S'}$$

See Section B.3.2 for the proof that this data forms a parameterised adjunction.

This adjunction determines the typed global state monad by the construction in Theorem B.1.2: we already know that it is the same as a functor and the units are defined to be the same. The derived multiplication is the same as the original multiplication:

$$\begin{aligned}
& \mu'_{S_1,S_2,S_3,A} \\
= \;& G(S_1, \epsilon_{S_2,F(S_3,A)}) \\
= \;& \epsilon_{S_2,F(S_3,A),S_1} \\
= \;& \Lambda(\pi_1; \mathrm{ev})^{S_1}; ((A \times S)^!)^{S_1}; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}) \\
= \;& \Lambda(\pi_1; \mathrm{ev})^{S_1}; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}) \\
= \;& \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \pi_1; \mathrm{ev}) \\
= \;& \Lambda(\mathrm{ev}; \mathrm{ev}) \\
= \;& \Lambda(\mathrm{ev}); \mathrm{ev}^{S_1} \\
= \;& \mathrm{ev}^{S_1} \\
= \;& \mu_{S_1,S_2,S_3,A}
\end{aligned}$$

Hence this adjunction gives the original typed global state monad.

We now show that the category $\mathbf{StAlg}(\mathcal{C}, \mathcal{S})$ is isomorphic to the Eilenberg-Moore category $\mathcal{C}^T$. Define:

$$\begin{aligned}
K &: \mathbf{StAlg}(\mathcal{C}, \mathcal{S}) \to \mathcal{C}^T \\
K(\langle A, \mathit{store}, \mathit{retrieve}\rangle) &= \langle A, \mathit{store}^{\widehat{S_1}}_{S_2}; A!^{\widehat{S_1}}; \mathit{retrieve}_{S_1}\rangle \\
Kf &= f \\
L &: \mathcal{C}^T \to \mathbf{StAlg}(\mathcal{C}, \mathcal{S}) \\
L(\langle A, h\rangle) &= \langle A, \Lambda(\pi_1); h_{1,S}, \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S); h_{S,S}\rangle \\
Lf &= f
\end{aligned}$$

The definition of K matches the definition of the terminal arrow to $\mathcal{C}^T$ in the category $\mathbf{PAdj}(T)$ given in the proof of Theorem B.1.6. Therefore we automatically know it is well-defined as a functor. The definition of L is also a functor, see Section B.3.3. These two functors form an isomorphism, see Section B.3.4.

Therefore the category $\mathbf{StAlg}(\mathcal{C}, \mathcal{S})$ is isomorphic to the category of algebras for the typed global state monad. □

We believe that Theorem B.2.2 strongly hints that there is a connection to be made between parameterised monads and some notion of typed algebras where operations have start and finish types taken from the objects of some category. We envisage that further research along this line would first prove an analogue of Beck's Theorem for parameterised monads and then find a suitable generalisation of Lawvere theories that corresponds to finitary parameterised monads. See [Rob02] for more information about the case for normal and enriched monads.

B.3 Proof Details

B.3.1 The functor F is well-defined

Recall the definition of $F : \mathcal{S} \times \mathcal{C} \to \mathbf{StAlg}(\mathcal{C}, \mathcal{S})$:

$$\begin{aligned} F(S, A) &= \langle (A \times S)^-, \Lambda(\pi_1; \mathrm{ev}_{S_1,(A\times S)}), \\ &\qquad \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev}_{S_1,(A\times S)^{S_1}} \times S_1; \mathrm{ev}_{S_1,(A\times S_1)})\rangle \\ F(s, f) &= (f \times s)^- \end{aligned}$$

We must check each of the axioms in turn, the dinaturality of the *store* family and the naturality of the *retrieve* family. The first axiom:

$$\begin{aligned} & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \mathit{store}_{S_1}^{S_1}; ((A \times S)^!)^{S_1}; \mathit{retrieve}_{S_1} \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \\ & \quad \Lambda(\pi_1; \mathrm{ev})^{S_1}; ((A \times S)^!)^S; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \Lambda(\pi_1; \mathrm{ev})^{S_1}; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \Lambda(\Lambda(\pi_1; \mathrm{ev})^{S_1} \times S_1; ((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \Lambda(((A \times S)^{S_1} \times S_1)^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \pi_1; \mathrm{ev}) \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \Lambda(\mathrm{ev}; \mathrm{ev}) \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \Lambda(\mathrm{ev}); \mathrm{ev}^{S_1} \\ = \; & \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1); \mathrm{ev}^{S_1} \end{aligned}$$

$$
\begin{aligned}
&= \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \mathit{retrieve}_{S_1}
\end{aligned}
$$

The second axiom:

$$
\begin{aligned}
& \Lambda(\pi_1); \mathit{retrieve}_{S_1} \\
&= \Lambda(\pi_1); \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda(\Lambda(\pi_1) \times S_1; ((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda((A \times S)^{S_1} \times \mathit{dup}; \pi_1 \times S_1; \mathrm{ev}) \\
&= \Lambda(\mathrm{ev}) \\
&= \mathit{id}
\end{aligned}
$$

The third axiom:

$$
\begin{aligned}
& ((A \times S)^{!})^{S_1} \times S_1; \mathit{retrieve}_{S_1} \times S_1; \mathit{store}_{S_1} \\
&= ((A \times S)^{!})^{S_1} \times S_1; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}) \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
&= \Lambda(((A \times S)^{!})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}) \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
&= \Lambda(((A \times S)^{1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; (A \times S)^{!} \times S_1; \mathrm{ev}) \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
&= \Lambda(((A \times S)^{1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; (A \times S)^{1} \times !; \mathrm{ev}) \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
&= \Lambda(((A \times S)^{1})^{S_1} \times \langle \mathit{id}, ! \rangle; \mathrm{ev} \times 1; ; \mathrm{ev}) \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
&= \Lambda(\pi_1; \Lambda(((A \times S)^{1})^{S_1} \times \langle \mathit{id}, ! \rangle; \mathrm{ev} \times 1; ; \mathrm{ev}) \times S_1; \mathrm{ev}) \\
&= \Lambda(\pi_1; ((A \times S)^{1})^{S_1} \times \langle \mathit{id}, ! \rangle; \mathrm{ev} \times 1; ; \mathrm{ev}) \\
&= \Lambda(\mathrm{ev} \times 1; ; \mathrm{ev}) \\
&= \mathrm{ev}; \Lambda(\mathrm{ev}) \\
&= \mathrm{ev}
\end{aligned}
$$

The fourth axiom:

$$
\begin{aligned}
& \mathit{retrieve}_{S_1}^{S_1}; \mathit{retrieve}_{S_1} \\
&= \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev})^{S_1}; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda(\Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev})^{S_1} \times S_1; ((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; ((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev})
\end{aligned}
$$

$$
\begin{aligned}
&= \Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; (((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup} \times S_1; \mathrm{ev} \times S_1 \times S_1; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; \Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \times S_1 \times S_1; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda(\Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \times S_1; ((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}); \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= \Lambda((((A \times S)^{S_1})^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}); \mathit{retrieve}_{S_1}
\end{aligned}
$$

Dinaturality of *store*:

$$
\begin{aligned}
& (A \times S)^f \times S_1; \mathit{store}_{S_1} \\
= \; & (A \times S)^f \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
= \; & \Lambda(\pi_1; (A \times S)^f \times S_1; \mathrm{ev}) \\
= \; & \Lambda(\pi_1; (A \times S)^{S_1} \times f; \mathrm{ev}) \\
= \; & (A \times S)^{S_2} \times f; \Lambda(\pi_1; \mathrm{ev}) \\
= \; & (A \times S)^{S_2} \times f; \mathit{store}_{S_1}
\end{aligned}
$$

Naturality of *retrieve*:

$$
\begin{aligned}
& ((A \times S)^f)^f; \mathit{retrieve}_{S_1} \\
= \; & ((A \times S)^f)^f; \Lambda(((A \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev}) \\
= \; & \Lambda(((A \times S)^{S_2})^{S_2} \times \mathit{dup}; ((A \times S)^{S_2})^{S_2} \times f \times f; \mathrm{ev} \times S_2; \mathrm{ev}) \\
= \; & \Lambda(((A \times S)^{S_2})^{S_2} \times f; ((A \times S)^{S_2})^{S_2} \times \mathit{dup}; \mathrm{ev} \times S_2; \mathrm{ev}) \\
= \; & \Lambda(((A \times S)^{S_2})^{S_2} \times \mathit{dup}; \mathrm{ev} \times S_2; \mathrm{ev}); (A \times S)^f \\
= \; & \mathit{retrieve}_{S_2}; (A \times S)^f
\end{aligned}
$$

Hence F is well defined on objects. The definition of F is also well-defined on arrows, firstly, commutativity with the defined *store* natural transformations:

$$
\begin{aligned}
& F(s, f)_{S_1} \times S_1; \mathit{store}_{S_1} \\
= \; & (f \times s)^{S_1} \times S_1; \Lambda(\pi_1; \mathrm{ev}) \\
= \; & \Lambda((f \times s)^{S_1} \times S_1 \times S_1; \pi_1; \mathrm{ev}) \\
= \; & \Lambda(\pi_1; \mathrm{ev}; f \times s) \\
= \; & \Lambda(\pi_1; \mathrm{ev}); (f \times s)^{S_1} \\
= \; & \mathit{store}_{S_1}; F(s, f)_{S_1}
\end{aligned}
$$

Secondly, commutativity with the defined *retrieve* natural transformations is similar:

$$\begin{aligned}
& (F(s,f)_{S_1})^{S_1}; retrieve_{S_1} \\
= \;& ((f \times s)^{S_1})^{S_1}; \Lambda(((B \times S')^{S_1})^{S_1} \times dup; \mathrm{ev} \times S_1; \mathrm{ev}) \\
= \;& \Lambda(((A \times S)^{S_1})^{S_1} \times dup; \mathrm{ev} \times S_1; \mathrm{ev}; f \times s) \\
= \;& \Lambda(((A \times S)^{S_1})^{S_1} \times dup; \mathrm{ev} \times S_1; \mathrm{ev}); (f \times s)^{S_1} \\
= \;& retrieve_{S_1}; F(s,f)_{S_1}
\end{aligned}$$

B.3.2 The functor F is a parameterised left adjoint

Recall that the unit and counit are defined as:

$$\eta_{A,S} = \Lambda(id_{A \times S}) \qquad \epsilon_{S,\langle A, store, retrieve\rangle, S'} = store_S^{S'}; A!^{S'}; retrieve_{S'}$$

It is easy to see that these are well-defined in terms of objects. The counit is an arrow of $\mathbf{StAlg}(\mathcal{C}, \mathcal{S})$; it commutes with the *store* transformations:

$$\begin{aligned}
& \epsilon_{S,\langle A, store, retrieve\rangle, S_1} \times S_1; store_{S_1} \\
= \;& (store_S^{S_1}; A!^{S_1}; retrieve_{S_1}) \times S_1; store_{S_1} \\
= \;& store_S^{S_1} \times S_1; \mathrm{ev} \\
= \;& store_S^{S_1} \times S_1; \mathrm{ev}; \Lambda(\pi_1); retrieve_1 \\
= \;& \mathrm{ev}; store_S; \Lambda(\pi_1); retrieve_1 \\
= \;& \mathrm{ev}; \Lambda(\pi_1); store_S^1; retrieve_1 \\
= \;& \Lambda(\pi_1; \mathrm{ev}); store_S^1; A!^1; retrieve_1 \\
= \;& \Lambda(\pi_1; \mathrm{ev}); \epsilon_{S,\langle A, store, retrieve\rangle, 1}
\end{aligned}$$

and it commutes with the *retrieve* natural transformations:

$$\begin{aligned}
& \epsilon^{S_1}_{S,\langle A, store, retrieve\rangle, S_1}; retrieve_{S_1} \\
= \;& (store_S^{S_1})^{S_1}; (A!^{S_1})^{S_1}; retrieve_{S_1}^{S_1}; retrieve_{S_1} \\
= \;& (store_S^{S_1})^{S_1}; (A!^{S_1})^{S_1}; \Lambda((AS_1^{S_1})^{S_1} \times dup; \mathrm{ev} \times S_1; \mathrm{ev}); retrieve_{S_1}
\end{aligned}$$

$$\begin{aligned}
&= \Lambda(((AS \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}; \mathit{store}_S; A!); \mathit{retrieve}_{S_1} \\
&= \Lambda(((AS \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}); \mathit{store}_S^{S_1}; A!^{S_1}; \mathit{retrieve}_{S_1} \\
&= \Lambda(((AS \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}); \epsilon_{S,\langle A,\mathit{store},\mathit{retrieve}\rangle,S_1}
\end{aligned}$$

This definition is also natural:

$$\begin{aligned}
& F(S, G(S, f))_{S_1}; \epsilon_{S,\langle B,\mathit{store},\mathit{retrieve}\rangle,S_1} \\
= \;& (f_S \times S)^{S_1}; \mathit{store}_S^{S_1}; B!^{S_1}; \mathit{retrieve}_{S_1} \\
= \;& \mathit{store}_S^{S_1}; A!^{S_1}; \mathit{retrieve}_{S_1}; f_{S_1} \\
= \;& \epsilon_{S,\langle A,\mathit{store},\mathit{retrieve}\rangle,S_1}; f_{S_1}
\end{aligned}$$

and dinatural:

$$\begin{aligned}
& F(s, G(S, \langle A, \mathit{store}, \mathit{retrieve}\rangle))_{S_1}; \epsilon_{S,\langle A,\mathit{store},\mathit{retrieve}\rangle,S_1} \\
= \;& (AS \times s)^{S_1}; \mathit{store}_S^{S_1}; A!^{S_1}; \mathit{retrieve}_{S_1} \\
= \;& (As \times S')^{S_1}; \mathit{store}_{S'}^{S_1}; A!^{S_1}; \mathit{retrieve}_{S_1} \\
= \;& F(S', G(s, \langle A, \mathit{store}, \mathit{retrieve}\rangle))_{S_1}; \epsilon_{S',\langle A,\mathit{store},\mathit{retrieve}\rangle,S_1}
\end{aligned}$$

This is a good definition of counit; it satisfies the triangle equalities for a parameterised monad:

$$\begin{aligned}
& \eta_{S,G(S,\langle A,\mathit{store},\mathit{retrieve}\rangle)}; G(S, \epsilon_{S,\langle A,\mathit{store},\mathit{retrieve}\rangle}) \\
= \;& \eta_{S,G(S,\langle A,\mathit{store},\mathit{retrieve}\rangle)}; \epsilon_{S,\langle A,\mathit{store},\mathit{retrieve}\rangle,S} \\
= \;& \Lambda(\mathit{id}); \mathit{store}^S; A!^S; \mathit{retrieve}_S \\
= \;& \Lambda(\Lambda(\pi_1) \times S; AS^S \times \mathit{dup}; \mathrm{ev} \times S); \mathit{store}^S; A!^S; \mathit{retrieve}_S \\
= \;& \Lambda(\pi_1); \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S); \mathit{store}^S; A!^S; \mathit{retrieve}_S \\
= \;& \Lambda(\pi_1); \mathit{retrieve}_S \\
= \;& \mathit{id}
\end{aligned}$$

and:

$$\begin{aligned}
& F(S, \eta_{S,A})_{S_1}; \epsilon_{S,F(S,A),S_1} \\
= \;& (\Lambda(\mathit{id}) \times S)^{S_1}; \Lambda(\pi_1; \mathrm{ev})^{S_1}; (((A \times S)^S \times S)^!)^{S_1}; \\
& \Lambda((((A \times S)^S \times S)^{S_1})^{S_1} \times \mathit{dup}; \mathrm{ev} \times S_1; \mathrm{ev})
\end{aligned}$$

$$
\begin{aligned}
&= (\Lambda(id) \times S)^{S_1}; \Lambda(\pi_1; \mathrm{ev})^{S_1}; \Lambda((((A \times S)^S \times S)^{S_1})^{S_1} \times dup; \mathrm{ev} \times S_1; \mathrm{ev}) \\
&= (\Lambda(id) \times S)^{S_1}; \Lambda((((A \times S)^S \times S)^{S_1})^{S_1} \times dup; \mathrm{ev} \times S_1; \pi_1; \mathrm{ev}) \\
&= (\Lambda(id) \times S)^{S_1}; \Lambda(\mathrm{ev}; \mathrm{ev}) \\
&= (\Lambda(id) \times S)^{S_1}; \Lambda(\mathrm{ev}); \mathrm{ev}^{S_1} \\
&= (\Lambda(id) \times S)^{S_1}; \mathrm{ev}^{S_1} \\
&= id
\end{aligned}
$$

Hence $\langle F, G, \eta, \epsilon \rangle$ is a parameterised adjunction.

B.3.3 The functor L is well-defined

Recall the definition of L:

$$
\begin{aligned}
L &: \mathcal{C}^T \to \mathbf{StAlg}(\mathcal{C}, \mathcal{S}) \\
L(\langle A, h \rangle) &= \langle A, \Lambda(\pi_1); h_{1,S}, \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \rangle \\
Lf &= f
\end{aligned}
$$

Firstly, it is well-defined on objects: the first axiom:

$$
\begin{aligned}
&\Lambda(AS^S \times dup; \mathrm{ev} \times S); store^S_S; A!^S; retrieve_S \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\pi_1)^S; h^S_{1,S}; A!^S; \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\pi_1)^S; ((AS \times S)^!)^S; h^S_{S,S}; \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\pi_1)^S; h^S_{S,S}; \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\pi_1)^S; h^S_{S,S}; \Lambda(AS^S \times dup; \mathrm{ev} \times S); (h_{S,S} \times S)^S; h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\pi_1)^S; \Lambda(AS^S \times dup; \mathrm{ev} \times S); \mathrm{ev}^S; h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\pi_1)^S; \Lambda(AS^S \times dup; \mathrm{ev} \times S; \mathrm{ev}); h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(AS^S \times dup; \mathrm{ev} \times S; \pi_1); h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); \Lambda(\mathrm{ev}); h_{S,S} \\
= \;& \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \\
= \;& retrieve_S
\end{aligned}
$$

The second axiom:

$$\begin{aligned}
& \Lambda(\pi_1); \mathit{retrieve}_S \\
= \;& \Lambda(\pi_1); \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda(AS^S \times \mathit{dup}; \pi_1 \times S); h_{S,S} \\
= \;& \Lambda(\mathit{id}); h_{S,S} \\
= \;& \mathit{id}
\end{aligned}$$

The third axiom:

$$\begin{aligned}
& A!^S \times S; \mathit{retrieve}_S \times S; \mathit{store}_S \\
= \;& A!^S \times S; \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S) \times S; h_{S,S} \times S; \Lambda(\pi_1); h_{1,S} \\
= \;& A!^S \times S; \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S) \times S; \Lambda(\pi_1); (h_{S,S} \times S)^1; h_{1,S} \\
= \;& A!^S \times S; \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S) \times S; \Lambda(\pi_1); \mathrm{ev}^1; h_{1,S} \\
= \;& \Lambda(A1^S \times \mathit{dup}; \mathrm{ev} \times S) \times S; (A! \times S)^S \times S; \Lambda(\pi_1); \mathrm{ev}^1; h_{1,S} \\
= \;& \Lambda(A1^S \times \mathit{dup}; \mathrm{ev} \times S); \Lambda(\pi_1); ((A! \times S)^S \times S)^1; \mathrm{ev}^1; h_{1,S} \\
= \;& \Lambda(A1^S \times \mathit{dup}; \mathrm{ev} \times S); \Lambda(\pi_1); \mathrm{ev}^1; (A! \times S)^1; h_{1,S} \\
= \;& \Lambda(A1^S \times \mathit{dup}; \mathrm{ev} \times S); \Lambda(\pi_1); \mathrm{ev}^1; (A1 \times !)^1; h_{1,1} \\
= \;& \Lambda(\pi_1; \Lambda(A1^S \times \mathit{dup}; \mathrm{ev} \times S); \mathrm{ev}); (A1 \times !)^1; h_{1,1} \\
= \;& \Lambda(\pi_1; A1^S \times \mathit{dup}; \mathrm{ev} \times S); (A1 \times !)^1; h_{1,1} \\
= \;& \Lambda(\pi_1; A1^S \times \mathit{dup}; \mathrm{ev} \times !); h_{1,1} \\
= \;& \Lambda(\mathrm{ev} \times S); h_{1,1} \\
= \;& \mathrm{ev}; \Lambda(\mathit{id}); h_{1,1} \\
= \;& \mathrm{ev}
\end{aligned}$$

The fourth axiom:

$$\begin{aligned}
& \Lambda((AS^S)^S \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}); \mathit{retrieve}_S \\
= \;& \Lambda((AS^S)^S \times \mathit{dup}; \mathrm{ev} \times S; \mathrm{ev}); \Lambda(AS^S \times \mathit{dup}; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda((AS^S)^S \times \mathit{dup}; (AS^S)^S \times \mathit{dup} \times S; \mathrm{ev} \times S; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda((AS^S)^S \times \mathit{dup}; \mathrm{ev} \times S; \Lambda((AS^S)^S \times \mathit{dup}; \mathrm{ev} \times S); \mathrm{ev}); h_{S,S}
\end{aligned}$$

$$
\begin{aligned}
&= \Lambda((AS^S)^S \times dup; \mathrm{ev} \times S)^S; \Lambda(AS^S \times dup; \mathrm{ev} \times S; \mathrm{ev}); h_{S,S} \\
&= \Lambda((AS^S)^S \times dup; \mathrm{ev} \times S)^S; \Lambda(AS^S \times dup; \mathrm{ev} \times S); \mathrm{ev}^S; h_{S,S} \\
&= \Lambda((AS^S)^S \times dup; \mathrm{ev} \times S)^S; \Lambda(AS^S \times dup; \mathrm{ev} \times S); (h_{S,S} \times S)^S; h_{S,S} \\
&= \Lambda((AS^S)^S \times dup; \mathrm{ev} \times S)^S; h^S_{S,S}; \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \\
&= retrieve^S_S; retrieve_S
\end{aligned}
$$

The defined *store* family is dinatural:

$$
\begin{aligned}
& As \times S; store_S \\
= \;& As \times S; \Lambda(\pi_1); h_{1,S} \\
= \;& \Lambda(\pi_1); (As \times S)^1; h_{1,S} \\
= \;& \Lambda(\pi_1); (AS' \times s)^1; h_{1,S'} \\
= \;& AS' \times s; \Lambda(\pi_1); h_{1,S'} \\
= \;& AS' \times s; store_{S'}
\end{aligned}
$$

The defined *retrieve* family is natural:

$$
\begin{aligned}
& Af^f; retrieve_S \\
= \;& Af^f; \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda(AS'^{S'} \times dup; Af^f \times S \times S; \mathrm{ev} \times S); h_{S,S} \\
= \;& \Lambda(AS'^{S'} \times dup; AS'^{S'} \times f \times S; \mathrm{ev} \times S; Af \times S); h_{S,S} \\
= \;& \Lambda(AS'^{S'} \times dup; AS'^{S'} \times f \times S; \mathrm{ev} \times S); (Af \times S)^S; h_{S,S} \\
= \;& \Lambda(AS'^{S'} \times dup; AS'^{S'} \times f \times S; \mathrm{ev} \times S); (AS' \times f)^S; h_{S,S'} \\
= \;& \Lambda(AS'^{S'} \times dup; AS'^{S'} \times f \times f; \mathrm{ev} \times S'); h_{S,S'} \\
= \;& \Lambda(AS'^{S'} \times f; AS'^{S'} \times dup; \mathrm{ev} \times S'); h_{S,S'} \\
= \;& \Lambda(AS'^{S'} \times dup; \mathrm{ev} \times S'); (AS' \times S')^f; h_{S,S'} \\
= \;& \Lambda(AS'^{S'} \times dup; \mathrm{ev} \times S'); h_{S',S'}; Af \\
= \;& retrieve_{S'}; Af
\end{aligned}
$$

Hence L is well-defined on objects. The definition is also well-defined on arrows; commutativity with *store*:

$$L(f)_S \times S; store_S$$

$$\begin{aligned} &= f_S \times S; \Lambda(\pi_1); h^B_{1,S} \\ &= \Lambda(\pi_1); (f_S \times S)^1; h^B_{1,S} \\ &= \Lambda(\pi_1); h^A_{1,S}; f_1 \\ &= store_S; L(f)_1 \end{aligned}$$

and commutativity with *retrieve*:

$$\begin{aligned} & L(f)^S_S; retrieve_S \\ = \; & f^S_S; \Lambda(AS^S \times dup; \mathrm{ev} \times S); h^B_{S,S} \\ = \; & \Lambda(AS^S \times dup; \mathrm{ev} \times S); (f_S \times S)^S; h^B_{S,S} \\ = \; & \Lambda(AS^S \times dup; \mathrm{ev} \times S); h^A_{S,S}; f_S \\ = \; & retrieve_S; f_S \end{aligned}$$

Hence L is a functor $\mathcal{C}^T \to \mathbf{StAlg}(\mathcal{C}, \mathcal{S})$.

B.3.4 The functors L and K form an isomorphism

It is immediate that they are bijective on arrows. On objects, for $K; L$:

$$\begin{aligned} & L(K(\langle A, store, retrieve\rangle)) \\ = \; & L(\langle A, store^{S_1}_{S_2}; A!^{S_1}; retrieve_{S_1}\rangle) \\ = \; & \langle A, \Lambda(\pi_1); store^1_S; A!^1; retrieve_1, \Lambda(AS^S \times dup; \mathrm{ev} \times S); store^S_S; A!^S; retrieve_S\rangle \end{aligned}$$

We must check that these new store and retrieve operations are equal to the originals:

$$\begin{aligned} & \Lambda(\pi_1); store^1_S; A!^1; retrieve_1 \\ = \; & store_S; \Lambda(\pi_1); retrieve_1 \\ = \; & store_S \end{aligned}$$

and:

$$\begin{aligned} & \Lambda(AS^S \times dup; \mathrm{ev} \times S); store^S_S; A!^S; retrieve_S \\ = \; & retrieve_S \end{aligned}$$

Hence $K; L = Id$. In the opposite direction:

$$\begin{aligned} & K(L(\langle A, h\rangle)) \\ = \;& K(\langle A, \Lambda(\pi_1); h_{1,S}, \Lambda(AS^S \times dup; \mathrm{ev} \times S); h_{S,S}\rangle) \\ = \;& \langle A, \Lambda(\pi_1)^{S_1}; h^{S_1}_{1,S_2}; A!^{S_1}; \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S_1); h_{S_1,S_1}\rangle \end{aligned}$$

We must check that the two structure maps are equal:

$$\begin{aligned} & \Lambda(\pi_1)^{S_1}; h^{S_1}_{1,S_2}; A!^{S_1}; \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S_1); h_{S_1,S_1} \\ = \;& \Lambda(\pi_1)^{S_1}; ((AS_2 \times S_2)^!)^{S_1}; h^{S_1}_{S_1,S_2}; \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S); h_{S_1,S_1} \\ = \;& \Lambda(\pi_1)^{S_1}; h^{S_1}_{S_1,S_2}; \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S_1); h_{S_1,S_1} \\ = \;& \Lambda(\pi_1)^{S_1}; \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S); (h_{S_1,S_2} \times S_1)^{S_1}; h_{S_1,S_1} \\ = \;& \Lambda(\pi_1)^{S_1}; \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S); \mathrm{ev}^{S_1}; h_{S_1,S_2} \\ = \;& \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S; \Lambda(\pi_1) \times S; \mathrm{ev}); h_{S_1,S_2} \\ = \;& \Lambda(AS_1^{S_1} \times dup; \mathrm{ev} \times S; \pi_1); h_{S_1,S_2} \\ = \;& \Lambda(\mathrm{ev}); h_{S_1,S_2} \\ = \;& h_{S_1,S_2} \end{aligned}$$

Hence $L; K = Id$ and the categories $\mathcal{C}^T$ and $\mathbf{StAlg}(\mathcal{C}, \mathcal{S})$ are isomorphic.

Bibliography

[AB01] Carlos Areces and Patrick Blackburn. Bringing them all together. *Logic and Computation*, 11(5), 2001. Editorial of special issue on Hybrid Logics.

[Abr93] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[AFM05] Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *International Conference on Functional Programming*, 2005. to appear.

[AH02] David Aspinall and Martin Hofmann. Another type system for in-place update. In D. Le Métayer, editor, *Programming Languages and Systems, Proceedings of 11th European Symposium on Programming*, volume 2305, pages 36–52. Springer-Verlag, 2002. Lecture Notes in Computer Science.

[Ahm04] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[AJW03] Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science*, pages 33–44, June 2003.

[AM96] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.

[Amb92] Simon John Ambler. *First Order Linear Logic in Symmetric Monoidal Closed Categories*. PhD thesis, LFCS, Edinburgh University, January 1992.

[AMJ94] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for pcf. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1994.

[Atk04] Robert Atkey. A λ-calculus for resource separation. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004*, volume 3142 of *Lecture Notes in Computer Science*, pages 158–170. Springer, July 2004.

[Bak92] Henry G. Baker. Lively linear lisp–'look ma, no garbage!'. *ACM Sigplan notices*, 27(8):89–98, 1992.

[Bak95] Henry G. Baker. 'use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM Sigplan Notices*, 30(1):45–52, January 1995.

[Bar79] M. Barr. **-autonomous categories*. Number 752 in Lecture Notes in Mathematics. Springer, 1979.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, LFCS, University of Edinburgh, 1996.

[BBdPH92] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. Term assignment for intuitionistic linear logic. Technical Report 262, Computer Laboratory, University of Cambridge, August 1992.

[BBdPH93a] 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*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BBdPH93b] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. Linear lambda calculus and categorical models revisited. In *Proceedings of Sixth Conference on Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, 1993.

[BCS97] R. F. Blute, J. R. B. Cockett, and R. A. G. Seely. Categories for computation in context and unified logic. *Journal of Pure and Applied Algebra*, 116:49–98, 1997.

[BdGR97] 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.

[Bec67] Jonathan Mock Beck. *Triples, algebras and cohomology*. PhD thesis, Columbia University, 1967. Available as Reprints in Theory and Applications of Categories, No. 2, 2003.

[Ben95] N. Benton. A mixed linear and non-linear logic: proofs, terms and models. In *Proceedings of Computer Science Logic '94*, volume 933 of *Lecture Notes in Computer Science*, 1995.

[Ben05] Nick Benton. A typed compositional logic for a stack-based abstract machine. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, November 2005. To appear.

[BHM02] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics: Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer-Verlag GmbH, 2002.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In J. Lind-

skov Knudsen, editor, *Proceedings of ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2001.

[Bor94] Francis Borceaux. *Handbook of Categorical Algebra I*, volume 50 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.

[BPR00] Gavin M. Bierman, Andrew M. Pitts, and Claudio V. Russo. Operational properties of lily, a polymorphic linear lambda calculus with recursion. *Electr. Notes Theor. Comput. Sci.*, 41(3), 2000.

[BS93] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In R. K. Shyamasundar, editor, *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 41–51. Springer-Verlag, 1993. Lecture Notes in Computer Science 761.

[BTSR04] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In Jones and Leroy [JL04], pages 220–231.

[BTSY05] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules. In *LICS*, pages 260–269. IEEE Computer Society, 2005.

[BW83] Michael Barr and Charles Wells. *Toposes, Triples and Theories*. Springer-Verlag, 1983. `http://www.cwru.edu/artsci/math/wells/pub/ttt.html`.

[BW96] Nick Benton and Philip Wadler. Linear logic, monads, and the lambda calculus. In *Proceedings of 11th IEEE Symposium on Logic in Computer Science*, 1996.

[CG92] P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type checking in fsub. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

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

[CGR96] J. Chirimar, C. A. Gunter, and J. G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[CH97] Chih-Ping Chen and Paul Hudak. Rolling your own mutable ADT—A connection between linear types and monads. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Paris, France, 15–17 1997.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[CHW02] Mario José Cáccamo, J. Martin E. Hyland, and Glynn Winskel. Lecture notes in category theory (draft). Unpublished manuscript, June 2002.

[CKR98] William D. Clinger, Richard Kelsey, and Jonathan Rees. Revised^5 report on the algorithmic language scheme. *Journal of Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1/2):4–56, February 1994.

[CPR05] Matthew Collinson, David Pym, and Edmund Robinson. On bunched polymorphism. In Luke Ong, editor, *Computer Science*

Logic: 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, volume 3634 of *Lecture Notes in Computer Science*. Springer, 2005.

[Cro94] Roy Crole. *Categories for Types*. Cambridge University Press, 1994.

[CS97] J. R. B. Cockett and R. A. G. Seely. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114(2):133–173, 1997.

[CS99] J. R. B. Cockett and R. A. G. Seely. Linearly distributive functors. *Journal of Pure and Applied Algebra*, 143:155–203, 1999.

[Day70] 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.

[DB76] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[DF89] Olivier Danvy and Andrzej Filinski. A functional abstract of typed contexts. Technical report, DIKU – Computer Science Department, University of Copenhagen, August 1989.

[Dos93] Kosta Dosen. A historical introduction to substructural logics. In Schroeder-Heister and Došen [SHD93], pages 1–30.

[DS95] Brian Day and Ross Street. Kan extensions along promonoidal functors. *Theory and Applications of Categories*, 1(4):72–77, 1995.

[dSCHdP96] 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.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, number 19 in Proc. Amer. Math. Soc. Symposia in Applied Mathematics, pages 19–31, 1967.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schießen. In *Mathematische Zeitschrift 39* [Sza69], pages 176–210.

[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

[Gha95] Neil Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, 1995.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[GMJ+02] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI*, pages 282–293, 2002.

[GP01] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[Har01] Dana G. Harrington. A type system for destructive updates in declarative programming languages. Master's thesis, University of Calgary, 2001.

[HDM93] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4), 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on combinatory logic, lambda calculus, and formalism.* Academic Press, 1980.

[HPP02] Martin Hyland, Gordon D. Plotkin, and John Power. Combining computational effects: commutativity & sum. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *IFIP TCS*, volume 223 of *IFIP Conference Proceedings*, pages 474–484. Kluwer, 2002.

[IK02] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 331–342, New York, NY, USA, 2002. ACM Press.

[IO01] Samin Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 14–26, January 2001.

[ISO99] ISO. *Programming languages – C.* 1999. ISO/IEC 9899:1999.

[JL04] Neil D. Jones and Xavier Leroy, editors. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

ming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. ACM, 2004.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of USENIX Annual Technical Conference*, pages 275–288, June 2002.

[Joh89] P. T. Johnstone. A topos-theorist looks at dilators. *J. Pure and Applied Algebra*, 58(3):235–249, 1989.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL*, pages 71–84, 1993.

[Kob99] Naoki Kobayashi. Quasi-linear types. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 29–42, New York, NY, 1999.

[Koc72] Anders Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23, 1972.

[KP93] G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads. *Journal of Pure and Applied Algebra*, (89):163–179, 1993.

[Kri63a] S. Kripke. Semantical analysis of intuitionistic logic I. In J. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, 1963.

[Kri63b] S. Kripke. Semantical analysis of modal logic I. *Zeitschrift fur math. Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kri63c] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Kri65] S. Kripke. Semantical analysis of modal logic II. In Addison, Henkin, and Tarski, editors, *The theory of models*. North-Holland, 1965.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, (65):154–169, 1958.

[Law70] F. W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. *Proc. Sympos. Pure Math.*, XVII:1–14, 1970.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[LG97] Christoph Lüth and Neil Ghani. Monads and modular term rewriting. In Eugenio Moggi and Giuseppe Rosolini, editors, *Category Theory and Computer Science*, volume 1290 of *Lecture Notes in Computer Science*, pages 69–86. Springer, 1997.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *Proceedings 7th Annual IEEE Symp. on Logic in Computer Science, LICS'92*, pages 235–246. IEEE Computer Society Press, June 1992.

[Lon95] John Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1995.

[LPT03] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185:182–210, 2003.

[LS88] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*. Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1988.

[Mac94] Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 2nd edition, 1998.

[MAF05] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. L^3: A linear language with locations. In Urzyczyn [Urz05], pages 293–307.

[MM92] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.

[Mog89a] E. Moggi. Computational lambda-calculus and monads. In Rohit Parikh, editor, *Proceedings of the Fourth Annual IEEE Symp. on Logic in Computer Science, LICS 1989*, pages 14–23. IEEE Computer Society Press, June 1989.

[Mog89b] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[NSvEP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Concurrent clean. In Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem, editors, *PARLE (2)*, volume 506 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 1991.

[Ode92] Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

[O'H91] P. W. O'Hearn. Linear logic and interference control (preliminary report). In D H. Pitt, P.-L. Curien, S. Abramsky, A. M. Pitts, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, number 530 in Lecture Notes in Computer Science. Springer, September 1991.

[O'H93] P. W. O'Hearn. A model for syntactic control of interference. *Math. Struct. in Comput. Sci.*, 3:435–465, 1993.

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

[O'H05] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 2005.

[Ole82] Frank J. Oles. *A Category Theoretic Approach To Semantics of Programming Languages*. PhD thesis, Syracuse University, Syracuse, New York, August 1982.

[Ole85] Frank J. Oles. Type algebras, functor categories, and block structure. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, 1985.

[Ole97] Frank J. Oles. Functor categories and store shapes. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like languages, Volume 2*, pages 3–12. Birkhäuser, 1997.

[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.

[Orl28] I. E. Orlov. The calculus of compatibility in propositions. *Matheˇmaticheskiĭ sbornik*, 35:263–286, 1928. (In Russian).

[Ove03] David Overton. *Precise and expressive mode systems for typed logic programming languages*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, December 2003.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In Jones and Leroy [JL04], pages 268–280.

[PHCP03] 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.

[Pit00] A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[Plo93] G. D. Plotkin. Type theory and recursion. In Moshe Vardi, editor, *Proceedings of the Eighth Annual IEEE Symp. on Logic in Computer Science, LICS 1993*, pages 374–374. IEEE Computer Society Press, June 1993. Invited Talk.

[Pow89] A. J. Power. A general coherence result. *J. Pure Appl. Algebra*, 57(2):165–173, March 1989.

[Pow95] A. John Power. Why tricategories? *Information and Computation*, 120(2):251–262, 1995.

[Pow00a] A. J. Power. Enriched lawvere theories. *Theory and Applications of Categories*, pages 83–93, 2000.

[Pow00b] John Power. Models for the computational lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 40, 2000.

[POY04] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.

[PP99] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, number 1581 in Lecture Notes in Computer Science, pages 295–309. Springer-Verlag, April 1999.

[PP01] Gordon D. Plotkin and John Power. Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.*, (45), 2001.

[PP02] Gordon D. Plotkin and John Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures : 5th International Conference, FOSSACS 2002*, number 2303 in Lecture Notes in Computer Science. Springer-Verlag, April 2002.

[PP04] Gordon D. Plotkin and A. John Power. Computational effects and operations: An overview. *Electr. Notes Theor. Comput. Sci.*, 73:149–163, 2004.

[PR97] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Math. Struct. in Comp. Science*, 7:453–468, 1997.

[PT97] John Power and Hayo Thielecke. Environments, continuation semantics and indexed categories. In *Proc. Theoretical Aspects of Computer Science*, volume 1281 of *Lecture Notes in Computer Science*, pages 391–414, 1997.

[PT99] John Power and Hayo Thielecke. Closed freyd- and kappa-categories. In *ICALP*, volume 1644 of *Lecture Notes in Computer Science*. Springer, 1999.

[PT05] John Power and Miki Tanaka. Binding signatures for generic contexts. In Urzyczyn [Urz05], pages 308–323.

[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.

[Red93] Uday Reddy. A linear logic model of state. Electronic manuscript: `http://www.cs.bham.ac.uk/~udr/`, October 1993.

[Red94] Uday Reddy. Passivity and independence. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994*, pages 342–352. IEEE Computer Society Press, July 1994.

[Res00] Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000.

[Ret97] 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.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Paris, France*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

[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.

[Rey81] John C. Reynolds. The essence of Algol. In Jaco W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[Rey89] 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.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3–4):233–247, November 1993.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[Rob02] Edmund Robinson. Variations on algebra: Monadicity and generalisations of equational theories. *Formal Asp. Comput.*, 13(3-5):308–326, 2002.

[See89] R. A. G. Seely. Linear logic, *-autonomous categories, and cofree algebras. In *Categories in Computer Science and Logic*, number 92 in AMS Contemporary Mathematics, June 1989.

[Sel01] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

[SF89] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. MIT Press, 1989.

[SHD93] Peter Schroeder-Heister and Kosta Došen, editors. *Substructural Logics*. Number 2 in Studies in Logic and Computation. Oxford University Press, 1993.

[SM03] 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.

[SS04] Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *CSL*, volume 3210 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2004.

[Sta94] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also available as Technical Report 363, University of Cambridge Computer Laboratory.

[Sta05] Ian Stark. Free-algebra models for the i-calculus. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2005.

[SWM00] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland, Amsterdam, 1969.

[Thi97] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.

[Tro93] A. S. Troelstra. Tutorial on Linear Logic. In Schroeder-Heister and Došen [SHD93], pages 327–355.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[TW99] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999.

[Urz05] Pawel Urzyczyn, editor. *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*. Springer, 2005.

[VCHP04] Tom Murphyø VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS*, pages 286–295. IEEE Computer Society, 2004.

[VTL82] 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.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, Amsterdam, 1990.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, 1991.

[Wad92] P. Wadler. There's no substitute for linear logic. Presented at *Workshop on Mathematical Foundations Of Programming Language Semantics*, Oxford, April 1992.

[Wad93a] P. Wadler. A syntax for linear logic. In *Ninth International Conference on the Mathematical Foundations of Programming Semantics*,

number 802 in Lecture Notes in Computer Science. Springer-Verlag, April 1993.

[Wad93b] P. L. Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk*, New York, NY, 1993. Springer-Verlag.

[Wad94] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, January 1994.

[Wad99] Philip Wadler. The marriage of effects and monads. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 63–74, 1999.

[Wal05] David Walker. Substructural Type Systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–43. MIT Press, 2005.

[WCM00] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[Win05] Glynn Winskel. Relations in concurrency. In Prakash Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 2–11. IEEE Computer Society Press, June 2005. Invited Talk.

[WJ99] Keith Wansbrough and Simon L. Peyton Jones. Once upon a polymorphic type. In *POPL*, pages 15–28, 1999.

[WM00] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In Robert Harper, editor, *Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer, 2000.

[WW01] David Walker and Kevin Watkins. On regions and linear types. In *ICFP*, pages 181–192, 2001.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[XZL05] Hongwei Xi, Dengping Zhu, and Yanka Li. Applied type system with stateful views. Technical Report BUCS-2005-03, Computer Science Department, Boston University, January 2005.