Bunching for Regions and Locations

Matthew Collinson [1,2]

Department of Computer Science
University of Bath
Bath, United Kingdom

David Pym [1,3]

Hewlett-Packard Laboratories
Bristol, United Kingdom

Abstract

There are a number of applied lambda-calculi in which terms and types are annotated with parameters denoting either locations or locations in machine memory. Such calculi have been designed with safe memory-management operations in mind.

It is difficult to construct directly denotational models for existing calculi of this kind. We approach the problem differently, by starting from a class of mathematical models that describe some of the essential semantic properties intended in these calculi. In particular, disjointness conditions between regions (or locations) are implicit in many of the memory-management operations.

Bunched polymorphism provides natural type-theoretic mechanisms for capturing the disjointness conditions in such models. We illustrate this by adding regions to the basic disjointness model of $\alpha\lambda$, the lambda-calculus associated to the logic of bunched implications. We show how both additive and multiplicative polymorphic quantifiers arise naturally in our models. A locations model is a special case. In order to relate this enterprise back to previous work on memory-management, we provide an example in which the model is refined and used to provide a denotational semantics for a language with explicit allocation and disposal of regions.

Key words: Denotational semantics, type, polymorphism, logic of bunched implications, region, location, reference.

[1] This research was partially supported by the UK EPSRC grant ‘Bunched ML’.
[2] Email: `cssmc@cs.bath.ac.uk`
[3] Email: `david.pym@hp.com`

1 Introduction

In recent years, there has been an upsurge of interest in the use of regions amongst the types, semantics and programming language communities. This interest can be traced back to the influential papers by Lucassen and Gifford [12] and Tofte and Talpin [25].

The system of [12] was designed to provide an analysis of side-effects in languages that combine functional and imperative programming. The side-effects were concerned with changes to machine state induced by evaluating expressions containing references. Regions were introduced to describe the area of store in which side-effects of expressions occur. This gave control over how the side-effects of various sub-expressions of some larger expression interact. Regions were taken to be infinite and references could be allocated, read and updated. In particular, references were always allocated within a specific, named region by the programmer.

Most of the current work on regions is on type systems for memory-management. Such type systems are intended to be a way of recycling unused memory earlier than would be done by a garbage-collector and in such a way that it is easy to see when memory has been allocated and deallocated. A recent survey of region-based memory-management can be found in [10]. In common with the languages described there, our type systems enjoy the so-called *region-safety* property; that is, there are no accesses to unallocated or deallocated regions occur at run time.

In [25], (almost) all values of an ML-like language are put in regions. This is done by a translation to a language in which expressions are annotated with region variables. The scopes of these region variables are then delimited by a certain kind of let expression. At runtime, these region variables are mapped to actual regions. For example, a value-creating program M is (roughly) translated into an annotated program $\mathsf{letreg}\ \rho\ \mathsf{in}\ M'\ \mathsf{at}\ \rho\ \mathsf{end}$, where M' is formed by suitably translating sub-expressions of M. When this is executed, first a region of memory (corresponding to ρ) is allocated, M' is evaluated (storing and retrieving values from ρ), then ρ is deallocated. Since letreg is the only construct for allocating and deallocating regions, a stack discipline for regions is enforced. An important part of this system is the existence of functions which are polymorphic in their region variables.

Later, the alias type family of languages was developed — see Walker and Morrisett [28] — also with safe operations for memory-management in mind. The types in these systems typically contain location parameters.

A key concept in this paper is that of *location*. For our present purposes, the principal properties that we need to capture are that a location is an indivisible part of machine memory and that values of arbitrary size may be stored at a location. Thus our conception of location is a (limited) abstraction of, for example, formulations in which large values are stored using multiple linked memory addresses. A *region* is simply a set of locations in machine memory. Regions may overlap. Following, for example, [12], a *reference* is to a location within a specified region. Thus our characterization of locations and regions obtains at a level of abstraction similar to that of resources in the semantics of **BI** [18,17], the mathematical structure of which captures the view that the essential properties of resources at that level of abstraction are their composability and comparability.

Region and location systems are usually presented together with a detailed operational semantics. These are often rather complicated, but various safety properties can nevertheless be verified. On the other hand, denotational models of region and location languages are almost never presented — an exception is Banerjee, Heintze and Riecke [1]. Recently Morrisett, Ahmed and Fluet [13] have also taken some steps in that direction. We believe that mathematical models are no less important for these languages than they are for any other. The usual arguments for denotational semantics in terms of conceptual clarity, abstract correctness criteria and methods for proving equivalence of programs (possibly in different implementations or even languages) all hold in this setting; see Scott and Strachey [21].

In this paper, we approach memory-management from a denotational viewpoint. We present techniques for building models of type-systems in which region and location parameters are present in types. The two key techniques are the use of bunched polymorphism and the construction of models that are indexed categories over a base of regions.

The point of bunched polymorphism is to provide additive and multiplicative variants of polymorphic quantification within a single lambda-calculus. Theoretical aspects were studied in Collinson, Pym, and Robinson [3]. In that paper, a definition of categorical model was given. These consist of variants of hyperdoctrines for polymorphism [22] together with extra structure to intepret the multiplicatives. Thus the fibres of the underlying indexed category consist of doubly-closed categories, while the base has an additional monoidal structure to interpret the multiplicative combination of type variables. The monoid also supports a kind of weak projection and this enables one to define the functor that models the multiplicative quantifier as the right-adjoint to the induced substitution. Soundness and completeness theorems are given. A model based on partial equivalence relations was given: the indexed category has pairs of pers in the fibres.

We are concerned herein with a simpler situation, in which we have region rather than type variables. This enables us to give a more concrete resource reading to the bunched polymorphic quantifiers. From a logical perspective this constitutes a step down from second-order propositional to second-order predicate quantification. In models such as those we present, the additive quantifier turns out to be ordinary second-order quantification: it says ‘for all predicates’ in the model. This gives a mechanism for treating region polymorphic functions. The multiplicative quantifier also has an appealing and useful character. It may be read as saying ‘for all new predicates’, where ‘new’ means disjoint from the predicates used to interpret the scope of the quantifier. We put this multiplicative quantifier to good use.

Disjointness properties are crucial to the allocation and deallocation of regions. When a region is allocated, it is assumed that it is disjoint from all previously allocated regions. After a region is deallocated, there should be no use of any of its locations, and so it must be disjoint from all the regions used by the remainder of the computation. Implicit disjointness conditions between region variables may be captured using the syntax of bunched polymorphism. Similar comments apply to calculi with location variables.

We present a method of constructing models (indexing over a certain category) which is quite general. The particular models we use as examples are constructed from the basic disjointness model (BDM) of O'Hearn [14] for $\alpha\lambda$, the lambda calculus associated with the bunched logic, **BI**, of O'Hearn and Pym [15,18]. The BDM is a relative both of Reynolds-Oles functor category semantics [16,19], and of the heap semantics of separation logic, see Reynolds [20]. In the BDM, types are denoted by variable sets (presheaves). These are sets parametrized by sets of machine locations. The way locations are used makes it an obvious setting to attempt regioning. The fact that it supports $\alpha\lambda$ helps to develop a language for references.

As this is a first attempt to construct simple denotational models of regions we do not expect to retain all the useful properties of pre-existing region and location languages.

We begin by presenting a modification of the bunched polymorphic lambda calculus introduced by Collinson, Pym, and Robinson [3]. We review the BDM and its notion of state. We define regions and the category of realms (bunches of region variables) which is fundamental to our appoach. As an example of a semantics we show how to construct an indexed category with fibres based on the BDM and indexing by realms. We describe the interpretation of additive and multiplicative quantifiers. We show how this specializes to the case of location variables. We give a suitable modification to the notion of BDM state. We develop the example in more depth by giving a programming language with allocation and deallocation of regions. In this language, based on Berdine and O'Hearn [2], only references are placed in regions. We refine the BDM with regions to give a semantics and show how it supports the allocation and deallocation of regions.

We thank Josh Berdine and Peter O'Hearn for discussions and help with this work. In particular, their language and model in [2] is the starting point for our language and model for references in regions. Our models are also descended from the PER models of bunched polymorphism of our earlier joint work with Edmund Robinson [3]. Finally, we thank the anonymous referees for providing helpful comments in response to preliminary versions of this work.

2 Bunched Region Polymorphism

The calculi we wish to treat in this paper have types containing region variables and contexts containing bunches of such variables. The set-up of these calculi follows our earlier treatment of bunched polymorphism [3] for type (rather than region) variables. The principal syntactic differences between these cases are in the polymorphic applications: region variables must be instantiated with regions, whereas type variables may be instantiated with more general types.

This section presents the bunched polymorphic lambda-calculus that forms the essential functional core of all the type systems we will use in this paper. The calculus is based on the $\alpha\lambda$-calculus corresponding to **BI**, see [15,18]. Recall that this features contexts that are certain trees, called bunches, with ordinary variables at the leaves. We extend $\alpha\lambda$ to a new calculus by adding a context zone for type variables that also contains bunches.

The extension is orthogonal in the sense that the bunching in the two levels of variable are entirely independent. There are perfectly sensible calculi that are bunched in each, but not both, of the context zones. We have chosen a calculus with bunching at first-order in order to help us develop the language in Section 4.

Assume a countable collection of *region variables*, written ρ, possibly with subscripts, superscripts, primes and a countable collection of (ordinary) *variables*, written x, y, z.

The types are generated by the grammar

$$\tau := \top \mid I \mid \rho \ \mathsf{ref} \mid \tau \wedge \tau \mid \tau * \tau \mid \tau \to \tau \mid \tau \mathrel{-\!\!*} \tau \mid \forall\rho.\tau \mid \forall_*\rho.\tau \ \ ,$$

where ρ is a region variable. The operators $\top$, $\wedge$, $\to$ and $\forall$ are the additive unit, product, function space and polymorphic abstraction (universal quantifier), respectively. There are multiplicative operators for unit I, product $*$, function space $\mathrel{-\!\!*}$ and polymorphic abstraction $\forall_*$. We allow the letters σ, τ to range over types. A region variable ρ is *free* in τ if it is not bound by (in the scope of) a quantifier $\forall\rho$ or $\forall_*\rho$.

The type ρ ref introduces region variables into the type system. The idea is that a value with this type will be a reference to some location l in the region determined by ρ (in a suitable environment). For this first language, we are not assuming that the location holds some value. Consequently, we have none of the standard operations for references (dereference, update, allocation, disposal) or regions (allocation, disposal). These will be added when we elaborate the model in Section 4. Our initial aim is to investigate additive and multiplicative quantification over region variables and to build a simple set-theoretic model.

A *realm* is a bunch of region variables. These are generated as follows

$$B := \emptyset \mid \rho \mid B, B \mid B;B \ \ ,$$

subject to the restriction that any region variable may occur at most once in a bunch. Let B (and variants with primes, subscripts and superscripts) range over realms.

We write $B \vdash \tau$ and say the type τ is *well-formed over (the realm)* B when every free region variable in τ is present in B. In particular this means that the first-order formations for types ($\top$, $\wedge$, $\rightarrow$, I, $*$, $-\!*$) take place over fixed realms.

A *context* is a bunch of typed ordinary variables. These are generated by

$$\Gamma := \emptyset \mid \emptyset_* \mid x : \tau \mid \Gamma, \Gamma \mid \Gamma; \Gamma \ ,$$

with x a variable, τ a type and so that any variable occurs at most once. We use the letters Γ and Δ for contexts. The units $\emptyset$ and $\emptyset_*$ are distinct from the unit $\emptyset$ for realms. Write $B \vdash \Gamma$, and say that the context Γ is *well-formed over* B, when $B \vdash \tau$ for each variable $x : \tau$ in Γ. Our contexts are just the contexts of $\alpha\lambda$ over realms.

Bunches can be regarded as trees with labelled nodes. Bunches are always subject to a pair of equivalence relations, see [18] and this applies to both our realms and our contexts. The first equivalence $\equiv$ on bunches is used to build structural rules that allow us to permute variables in realms or contexts. It is given by commutative monoid rules for ";", for "," and by a congruence to ensure that the monoid rules can be applied at arbitrary depth in any bunch. The second relation $\cong$ is used to control contraction rules. The equivalence $\cong$ on realms is simply renaming of type variables: $B \cong B'$ if B' can be obtained from B by renaming bijectively with region variables. The relation $\Gamma \cong \Delta$ between contexts holds just when Δ can be obtained by relabelling the variables of the leaves of Γ in a type preserving way: any leaf $x : \tau$ of Γ must correspond to a node $y : \tau$ of Δ.

A *sub-bunch* of a bunch B is a sub-tree B' such that all leaves of B' are leaves of B. Let $B(B_1 \mid \ldots \mid B_n)$ be the notation for a bunch (realm or context) B with distinct, distinguished sub-bunches $B_1, \ldots, B_n$. The bunch $B[B'_1/B_1, \ldots B'_n/B_n]$ is formed by replacing each distinguished bunch B_i in B with B'_i. We reiterate that variables may only occur once in a context and that type variables may only occur once in a realm.

The terms of the language are given by the following grammar

$$\begin{aligned} M := {} & x \mid \top \mid I \mid \text{ let } I \text{ be } M \text{ in } M \mid \langle M, M \rangle \mid \pi_1 M \\ & \mid \pi_2 M \mid M * M \mid \text{ let } (x, y) \text{ be } M \text{ in } M \mid \lambda x : \tau . M \\ & \mid \mathrm{app}(M, M) \mid \lambda_* x : \tau . M \mid \mathrm{app}_*(M, M) \mid \Lambda \rho . M \\ & \mid \mathrm{App}(M, \rho) \mid \Lambda_* \rho . M \mid \mathrm{App}_*(M, \rho) \ , \end{aligned}$$

where ρ is a region variable, τ is a type, B is a realm and x is a variable. We use the letters M, N for terms.

We now attempt to convey the intended meanings behind our language — these will be made precise when we give denotations. A fundamental idea in this paper is to use the bunched structure of realms to keep track of assumed separation between interpretations of region variables. If two region variables are separated multiplicatively (by a comma) then they denote regions which do not overlap. On the other hand region variables separated additively may overlap. In this particular set-up, the parametrization by region variables enters the language through types $\rho\ \mathsf{ref}$. This is intended to be the type of references to locations in ρ. The first-order types $\top, \wedge, \rightarrow, I, *, -\!\!*$ have their standard disjointness readings, see [14] for example. A term $M : \forall\rho.\tau$ may be instantiated with any region to give term $\mathrm{App}(M,\rho) : \tau$. Therefore we have an explicit form of region polymorphism. In contrast, a term $M : \forall_*\rho.\tau$ may be instantiated with any region that is disjoint from all others appearing free in M and τ. This will enable us to type constants for region allocation and disposal in Section 4.

Let $FV(-)$ be the set of variables which are in a context $(-)$ or free (not bound by a lambda abstraction) in a term $(-)$. We use the notation $FRV(-)$ for the set of region variables which occur free in a realm $(-)$, type $(-)$, the types of the variables in the context $(-)$ or the type of the term $(-)$, respectively. In a term $\mathrm{App}_*(M,\rho)$, the type variable ρ is free. For each term M, let $\mu(M)$ be the set of region variables that are free and that arise in this way in M, that is, that are used to instantiate a multiplicative quantifier in a subterm of M.

The calculus produces *(term formation) judgements* of the form

$$B \mid \Gamma \vdash M : \tau \ ,$$

that a term M is well-typed with τ, given the bunch of region variables B and the bunch of (ordinary) variables Γ. These judgements depend on the well-formedness of types and contexts over realms. The judgements are derived according to a system of rules, a representative fragment of which is shown in Figure 1. In addition to the rules shown, there are introduction and elimination rules for rules for additive $(\top)$ and multiplicative (I) units, additive $(\wedge)$ and multiplicative $(*)$ conjunction, additive functions $(\rightarrow)$, contraction (C) and equivalence (E) for contexts. These may be found in [3]. All of the rules, other than the quantifier rules and the realm structurals, use a fixed realm B. That is to say, they are essentially the familiar rules for $\alpha\lambda$, but parametrized by the realm. Let the side-condition (††) on $(\forall I)$ and $(\forall_* I)$ be $\rho \notin FRV(\Gamma)$. The elimination rules $(\wedge E)$, $(*E)$, $(\rightarrow E)$ and $(-\!\!*E)$ are subject to the side-condition

$$(\dagger) \qquad \mu(N) \cap FRV(M) = \emptyset$$

that requires the separation of certain of the free region variables present. This side-condition makes substitution of terms M for variables x in terms

$$(Ax)\ \frac{B \vdash \tau}{B \mid x : \tau \vdash x : \tau} \qquad \frac{B \mid \Gamma(\Delta) \vdash M : \tau \quad B \vdash \Delta'}{B \mid \Gamma(\Delta; \Delta') \vdash M : \tau}\ (W)$$

$$(-\!\!*I)\ \frac{B \mid \Gamma, x : \sigma \vdash M : \tau}{B \mid \Gamma \vdash \lambda_* x : \sigma . M : \sigma -\!\!* \tau} \qquad (\dagger)\ \frac{B \mid \Gamma \vdash N : \sigma -\!\!* \tau \quad B \mid \Delta \vdash M : \sigma}{B \mid \Gamma, \Delta \vdash \mathsf{app}_*(N, M) : \tau}\ (-\!\!*E)$$

$$(\forall I)\ \frac{B; \rho \mid \Gamma \vdash M : \tau}{B \mid \Gamma \vdash \Lambda \rho . M : \forall \rho . \tau}\ (\dagger\dagger) \qquad \frac{B \mid \Gamma \vdash M : \forall \rho . \tau}{B; \rho' \mid \Gamma \vdash \mathsf{App}(M, \rho') : \tau[\rho'/\rho]}\ (\forall E)$$

$$(\forall_* I)\ \frac{B, \rho \mid \Gamma \vdash M : \tau}{B \mid \Gamma \vdash \Lambda_* \rho . M : \forall_* \rho . \tau}\ (\dagger\dagger) \qquad \frac{B \mid \Gamma \vdash M : \forall_* \rho . \tau}{B, \rho' \mid \Gamma \vdash \mathsf{App}_*(M, \rho') : \tau[\rho'/\rho]}\ (\forall_* E)$$

$$(FW)\ \frac{B' \mid \Gamma \vdash M : \tau}{B(B') \mid \Gamma \vdash M : \tau} \qquad (B_1 \equiv B_1')\ \frac{B_1 \mid \Gamma \vdash M : \tau}{B_1' \mid \Gamma \vdash M : \tau}\ (FE)$$

$$(FC)\ \frac{B(B_1; B_1') \mid \Gamma \vdash M : \tau}{B(B_1) \mid \Gamma[B_1/B_1'] \vdash M[B_1/B_1'] : \tau[B_1/B_1']}\ (B_1 \cong B_1')$$

Fig. 1. Term formations

N an admissible rule over a fixed realm. However, this requires that none the regions used by M is used to instantiate a multiplicative quantifier in the formation of N. This makes sense since the region used to form a witness to such an instantiation may be required to be disjoint from those required for x, and therefore also M. The corresponding side-conditions on elimination rules are necessary since subject-reduction requires substitution.

The reductions for this system consist of the reductions for $\alpha\lambda$ together with the evident $\beta\eta\zeta$-rules over a fixed realm, see [3] for details. The crucial metatheoretic properties (*i.e.*, admissible substitution (cut), normalization, subject-reduction) of the system all hold: the proofs are simplifications of those from the type variable case.

3 A Region Disjointness Model

Models for $\alpha\lambda$ consist of cartesian doubly closed categories (CDCCs). An important model called the basic disjointness model (BDM) for $\alpha\lambda$, is given in [14]. This model is based on the category of sets together with a category of sets of machine locations called worlds. It is the simplest of a series of models for $\alpha\lambda$ that explain the sharing interpretation of $\alpha\lambda$ in terms of computations with machine memory as resource.

In this section, we develop the BDM so that it supports bunched region polymorphism. We call the new model the *basic region disjointness model* (BRDM). This is an indexed category, in which the objects of the base are realms. It is an instance of (a simplification of) the general categorical model in [3]. The BRDM may be understood, for the most part, without knowledge of indexed categories since most of the constructions are really just about families of sets and functions.

3.1 The Basic Disjointness Model

The construction in [14] begins with a given infinite set, *Loc*, of *locations.* A *world* is a finite set of locations. Let $\mathcal{W} = \mathcal{P}_f(Loc)$ be the set of all worlds. This is also regarded as a discrete category under the same name.

The BDM is based on the functor category $Set^{\mathcal{W}}$, where *Set* is the category of sets and functions. Objects are used to denote the types and morphisms the terms of $\alpha\lambda$.

The cartesian closed structure is given pointwise and this is used to interpret the additive product and function types. The other monoidal closed structure (used to interpret the multiplicative products and function types) is slightly more involved. A partial, binary operation $*$ on $\mathcal{W}$ is defined by

$$V * W = \begin{cases} V \cup W & \text{if } V \cap W = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

for $V, W \in \mathcal{W}$. For any world W, let $W\#$ be the family of worlds which are disjoint from W. Following [4,5], this leads to a tensor $*$ on $Set^{\mathcal{W}}$. For $A, B \in Set^{\mathcal{W}}$ this is given by

$$(A * B)W = \{\langle U, V, a, b\rangle \mid a \in AU, b \in BV, U * V = W\}$$

at $W \in \mathcal{W}$. The corresponding exponent has

$$(A \mathrel{-\!*} B)V \;=\; \prod_{W \in V\#} (A(W) \Longrightarrow B(V * W))$$

at any world $V \in \mathcal{W}$, where $\Longrightarrow$ is the set-theoretic function space. This gives rise to the sharing interpretation of the multiplicatives: multiplicative pairs come from disjoint worlds; multiplicative functions take arguments from worlds disjoint from the one that the function uses.

Interesting examples using the separation properties of multiplicatives are produced in [14] through the use of a type of stores. In order to do this we let $L : \mathcal{W} \longrightarrow Set$ be the inclusion functor, so that $LW = W$ is thought of as a set of locations. Let $N : \mathcal{W} \longrightarrow Set$ be the constant functor to the natural numbers. Define a functor $S = L \longrightarrow (1 \vee (N \wedge L))$ using the evident pointwise operations. For some singleton $\{a\}$ and any $W \in \mathcal{W}$, we have $SW = W \Longrightarrow (\{a\} + (N \times W))$. Then we may regard $s \in SW$ as a representation of the portion of store at world W: the element a represents the null pointer and every other location contains both an integer and a location.

We now introduce some additional notation. Suppose $\mathcal{W}'$ is any sub-family of $\mathcal{W}$ and t is a family of functors with $t(U) \in \mathit{Set}^{\mathcal{W}}$ for each $U \in \mathcal{W}'$. Let $\prod_{U \in \mathcal{W}'}$ be the pointwise lifting of the product. That is,

$$(\prod_{U \in \mathcal{W}'} t(U))V = \prod_{U \in \mathcal{W}'} t(U)V$$

for any $V \in \mathcal{W}'$. Write π^U for both the U-th projection of $\prod_{U \in \mathcal{W}'}$ and its pointwise lifting.

3.2 Locations, Regions, and Realms

Recall that a location is an indivisible part of memory — for this section, we do not store values — and that a region is a — for this section, finite — subset of the set of locations. In this particular model, a region is the same as a world. This will not always be the case, and it is useful to distinguish regions and worlds as they play different roles.

For any function $R : \mathit{FRV}(B) \longrightarrow \mathcal{W}$, let

$$R[B_i] = \bigcup\{R(\rho) \mid \rho \in \mathit{FRV}(B)\} \subseteq \mathit{Loc}$$

for any B_i that is a sub-bunch of B. A *(region) environment* for the realm B is a function $R : \mathit{FRV}(B) \longrightarrow \mathcal{W}$ such that, if (B_1, B_2) is a sub-bunch of B, then $R[B_1] \cap R[B_2] = \emptyset$. Notice that if, for example, $(B_1; B_2)$ is a sub-bunch of B, then $R[B_1] \cap R[B_2]$ need not necessarily be empty. Let $\mathit{Loc}_R = R[B]$ for any region environment R for B. Let $\mathit{REnv}(B)$ be the set of region environments of B.

We now construct a category, *Realms*, with objects consisting of realms. Let A and B be realms, and let $\mathit{FRV}(B) = \{\rho'_1, \ldots, \rho'_n\}$. A *substitution* $(-)[\rho_1/\rho'_1, \ldots, \rho_n/\rho'_n]$ of variables from A for variables from B consists of $\rho_1, \ldots, \rho_n$ (not necessarily distinct) drawn from A such that: if B_i, B_j is a sub-bunch of B, ρ'_i is in B_i and ρ'_j is in B_j then there is a sub-bunch A_i, A_j of A with ρ_i in A_i and ρ_j in A_j. Note the special case if $n = 0$. An arrow in *Realms* from A to B is precisely a substitution of variables from A for variables from B. The verification that this is a category is routine. Furthermore, this category has finite products and an additional monoidal structure. This monoid is in fact a pseudoproduct, as described in [3].

Let $s : A \longrightarrow B$ be a substitution. For any $R \in \mathit{REnv}(A)$, we may define the function $R^s : \mathit{FRV}(B) \longrightarrow \mathcal{W}$. If $s = (-)[\rho_1/\rho'_1, \ldots, \rho_n/\rho'_n]$ take $R^s(\rho'_i) = R(\rho_i)$, for $1 \leq i \leq n$. This gives a function

$$(-)^s : \mathit{REnv}(A) \longrightarrow \mathit{REnv}(B)$$

between region environments for different realms.

We may extend any region environment R for B so that for some fresh region variable $\rho \notin \mathit{FRV}(B)$ we have

$$R^U : \mathit{FRV}(B) \cup \{\rho\} \longrightarrow \mathcal{W}$$

given by

$$R^U(\rho_i) = \begin{cases} R(\rho_i) & \text{if } \rho_i \in \mathit{FRV}(B) \\ U & \text{if } \rho_i = \rho \end{cases}$$

for any chosen region U. Let R be a region environment for B. Then R^U is an environment for $B; \rho$. If $\mathit{Loc}_R \cap U = \emptyset$, then R^U is also an environment for B, ρ.

3.3 The Indexed Category

A *semantic type* over a realm B is just a function $\tau : \mathit{REnv}(B) \longrightarrow \mathit{Set}^{\mathcal{W}}$. That is, it is a family of functors, indexed by region environments. A morphism $f : \tau \longrightarrow \tau'$ between semantic types is a family of arrows (natural transformations)

$$(f_R : \tau R \longrightarrow \tau' R)_{R \in \mathit{REnv}(B)}$$

indexed by region environments for B. That is, it is a family of functions indexed by region environments and worlds. Lifting composition and identity pointwise from $\mathit{Set}^{\mathcal{W}}$ gives a category $P(B)$ of semantic types over B. For each realm B, the category $P(B)$ is a CDCC, with the operations given by pointwise lifting from $\mathit{Set}^{\mathcal{W}}$.

Given a substitution $s : A \longrightarrow B$, define a functor $P(s) = s^* : P(B) \longrightarrow P(A)$ as follows. Let τ be a semantic type over B. Then

$$s^*(\tau)(R) = \tau(R^s)$$

where $R \in \mathit{REnv}(A)$. Given $f : \tau \longrightarrow \tau'$ over B we have $s^*(f) : s^*(\tau) \longrightarrow s^*(\tau')$ given by a family of natural transformations

$$s^*(f)_R = f_{R^s} : \tau(R^s) \longrightarrow \tau'(R^s)$$

where $R \in \mathit{REnv}(A)$.

The assignments for s^* make it a functor that preserves CDCC structure on-the-nose. The proofs of these facts are simple verifications from the definitions. In particular, the second follows from the pointwise nature of the definition. Further calculation shows that P gives a functor $P : \mathit{Realms} \longrightarrow \mathbf{CDCC}$ to the category of CDCCs and strict CDCC functors.

There are substitutions for projections $\pi : B; \alpha \longrightarrow B$ and $\psi : B, \alpha \longrightarrow B$ given by putting α_i for any α_i in B. The functor $\pi^* : P(B) \longrightarrow P(B; \alpha)$ maps an object $\tau \in P(B)$ to the family $\pi^*(\tau)(R) = \tau(R\restriction_{\mathit{FRV}(B)})$ indexed by $R \in \mathit{REnv}(B; \alpha)$. An arrow $f : \tau \longrightarrow \tau'$ of $P(B)$ is mapped to $\pi^*(f) : \pi^*(\tau) \longrightarrow$

$\pi^*(\tau')$ in $P(B;\alpha)$ given by the family $\pi^*(f)_R = f{\restriction_{FRV(B)}} : \tau(R{\restriction_{FRV(B)}}) \longrightarrow \tau'(R{\restriction_{FRV(B)}})$ indexed by $R \in \mathit{REnv}(B;\alpha)$. The description of the functor $\psi^* : P(B) \longrightarrow P(B,\alpha)$ is identical to the description of π^*, except that the families are now indexed by $R \in \mathit{REnv}(B,\alpha)$.

3.4 Functors for Universal Quantifiers

Define a functor $\Pi : P(B;\rho) \longrightarrow P(B)$ for the additive quantifier as follows. Send an object τ to $\Pi(\tau)$ with

$$\Pi(\tau)(R) = \prod_{U \in \mathcal{W}} \tau(R^U) \tag{1}$$

where R is an environment for B. Given $f : \tau \longrightarrow \tau'$, assign $\Pi(f)_R : \Pi(\tau)(R) \longrightarrow \Pi(\tau')(R)$ indexed by $R \in \mathit{REnv}(B)$, with component projection

$$\Pi(f)_{R,U} = f_{R^U} \circ \pi^U : \prod_{U \in \mathcal{W}} \tau(R^U) \longrightarrow \tau'(R^U)$$

at each $U \in \mathcal{W}$.

For the multiplicative quantifier we use the functor $\Psi : P(B,\rho) \longrightarrow P(B)$. This sends an object τ to $\Psi(\tau)$ with

$$\Psi(\tau)(R) = \prod_{U \in \mathit{Loc}_R\#} \tau(R^U) \tag{2}$$

where R is an environment for B. The action of Ψ on arrows is by a similar restriction of the indexing to regions disjoint from Loc_R.

Theorem 3.1 *There are adjunctions $\pi^* \dashv \Pi$ and $\psi^* \dashv \Psi$.*

The proof of the theorem is by unwinding the definitions of arrows in $P(B)$, $P(B;\rho)$ and $P(B,\rho)$. For the additive, this shows that the family underlying an arrow $\pi^*(\tau) \longrightarrow \tau'$ boils down to the same thing as the family underlying an arrow $\tau \longrightarrow \Pi(\tau')$. A similar proof gives the multiplicative case.

The Beck-Chevalley condition is a standard property required of indexed categorical models of ordinary (additive) quantifiers. It must be checked explicitly since the existence of the functors needed to model quantifiers does not necessarily guaranteee that it holds. From a logical point of view, the Beck-Chevalley condition says that quantification commutes with substitution: if we substitute after quantifying over ρ, it is the same as quantifying over ρ and then substituting. More information about the Beck-Chevalley condition can be found in [11].

Proposition 3.2 *The Beck-Chevalley condition for Π holds. That is, for any $s : A \longrightarrow B$, we have that*

$$\Pi \circ P(s \times 1) = P(s) \circ \Pi$$

holds. A weak version of the Beck-Chevalley condition holds for Ψ*. For any* $s : A \longrightarrow B$*, there is a bunch* B' *and an arrow* $s' : A \longrightarrow B'$ *such that*

$$w \circ s' = s \qquad\qquad \Psi \circ P(s' \otimes 1) = P(s') \circ \Psi$$

both hold, where $w : B' \longrightarrow B$ *is weakening.*

The full Beck-Chevalley condition does not hold for multiplicative quantification because of the disjointness condition on the indexing defining the action of the functor Π. Intuitively, if we add a new type variable ρ' by weakening before quantifying over ρ then ρ' must be disjoint from ρ. This is not necessarily the case if we quantify over ρ and then weaken in ρ'. The weak Beck-Chevalley condition makes appropriate modifications to deal with these separation issues.

3.5 Summary of the Model Structure

The fact that we have a model of the calculus follows immediately from the categorical results above. Some comments on this are in order.

Polymorphic types are interpreted using

$$\begin{aligned} [\![B \vdash \forall\rho.\tau]\!](R) &= \textstyle\prod_{U \in \mathcal{W}} \overline{\tau}(R^U) \\ [\![B \vdash \forall_*\rho.\tau]\!](R) &= \textstyle\prod_{U \in Loc_R\#} \underline{\tau}(R^U) \end{aligned} \tag{3}$$

where $R \in REnv(B)$, $\overline{\tau} = [\![B;\rho \vdash \tau]\!]$ and $\underline{\tau} = [\![B,\rho \vdash \tau]\!]$. The interpretation of the additive quantifier is such that the quantified variable ranges over all regions. By contrast, for the multiplicative it ranges only over all fresh regions, that is, those that are disjoint from all others used in the interpretation of the type. This hints at a connection between polymorphism and (region) allocation and disposal — something that has been suggested by several authors in the past. Teasing out the precise nature of this connection requires making some refinements to the model and careful, but relatively minor, changes to the calculus. This in done in Section 4.

Terms are interpreted using the hyperdoctrine structure of P; see [3]. The CDCC structure of a fibre $P(B)$ is reflected as the fixing of the realm in the first-order rules of Section 2.

In this language the region variables enter the language through the types ρ ref. At this stage we do not develop examples directly using these types. However, in order to be definite interpret

$$[\![B \vdash \rho\ \mathsf{ref}]\!]RW = \begin{cases} \{l\} \text{ if } \{l\} = W \subseteq R(\rho) \\ \emptyset \quad \text{otherwise} \end{cases}$$

for any region environment R and world W. A similar choice is made and explained in Section 4.

We obtain, with the interpretation of terms M given in [3], the following soundness property:

Theorem 3.3 *If $B \mid \Gamma \vdash M : \tau$ is provable, then there is an arrow $[\![M]\!] : [\![B \vdash \Gamma]\!] \longrightarrow [\![B \vdash \tau]\!]$ in the fibre over B.*

An alternative model exists in which worlds depend on realms. Let R be a region environment. A world, W, is *environmentally friendly* if $W \subseteq Loc_R$. Semantic types are defined as before, except that now, after being given a region environment, they accept only environmentally friendly worlds. This model provides functors to model both of the universal quantifiers. The multiplicative turns out to be simpler than the additive. The additive no longer satisfies the Beck-Chevalley condition, so is not quite ordinary polymorphism. For this reason, we prefer not to insist that worlds are environmentally friendly. In the BRDM, and in Section 4, compatibility constraints between worlds and realms are in the interpretation of references rather than the structure.

3.6 States for the BDM with Regions

We could simply take the BDM states to give a notion of state that is constant at each realm. However, we can refine the notion of state in order to take full advantage of region structure.

If $B = B_1; B_2$ or $B = B_1, B_2$ then write $R_i = R{\restriction}_{B_i} = R{\restriction}_{FRV(B_i)}$ and $W_i = W \cap Loc_{R_i}$, for $W \in \mathcal{W}$ and $i = 1, 2$. From any function s with domain W, we define $s_i = s{\restriction}_{W_i}$ to be the restriction to W_i.

Let $R : FRV(B) \longrightarrow \mathcal{W}$ be a region environment for a realm B. Define the *states* functor at R, $S_R : \mathcal{W} \longrightarrow Set$, recursively on the structure of the underlying realm B:

- if $B = \emptyset$ then $S_R W = \{\perp : \emptyset \Longrightarrow \{a\}\}$
- if $B = \rho$ then $S_R W = (W \cap R(\rho)) \Longrightarrow \{a\} + (N \times (W \cap R(\rho))$
- if $B = B_1; B_2$ or B_1, B_2 then $s \in S_R W$ iff $s_i \in S_{R_i} W_i$ for both $i = 1, 2$,

where $\{a\}$ is any fixed one-element set. The set $S_R W$ is always contained in the function space $(W \cap Loc_R) \Longrightarrow \{a\} + (N \times (W \cap Loc_R))$.

The clause for ρ enforces the condition that any linked system of locations that intersects $R(\rho)$ must be entirely contained within $R(\rho)$. Notice that the condition $s_i : W_i \longrightarrow \{a\} + (N \times W_i)$ is guaranteed in the final of the three defining clauses, so that the store s_i is contained entirely within the region W_i. It is important to understand the difference between the final two parts of that third clause. In the case $B = B_1; B_2$, the worlds W_1 and W_2 may overlap; therefore a location in W_i may point to a location in W_j for $i \neq j$. In the case $B = B_1, B_2$, the worlds W_1 and W_2 do not overlap, so no pointer in W_i may see a location in W_j. A procedure with a state parameter s that is typed over a realm B_1, B_2 could be guaranteed to produce an output entirely contained in $R[B_1]$ and so does not intersect $R[B_2]$. If the argument over B_2 is not used again, then we may dispose of $R[B_2]$. An example of this kind is given in [24], involving two lists, stored in separate regions, that are concatenated into one of those regions.

3.7 The BDM with Singleton Regions

The region model specializes easily to a location model: a model for a calculus with location variables rather than region variables.

Location variables are intended to range over locations rather than regions. The language is just as before; only the interpretation changes. A reference $x : \rho\ \mathsf{ref}$ is intended to live at the location specified by ρ. This kind of type features extensively in some of the more highly-developed calculi for memory management — see for example [13,28].

We continue to use $FRV(-)$, even though we now have location rather than region variables. We modify the notion of region environment to a *location environment*, which is a map $R : FRV(B) \longrightarrow Loc$ so that each location variable is associated to a unique location in an environment. In a location evnironment R with $R(\rho) = l$, a reference $x : \rho\ \mathsf{ref}$ lies at location l.

Note that when two location variables are combined multiplicatively they must be mapped to distinct locations by location environments. Semantic types, the base category and the functors induced by substitutions are constructed using the methods above.

An environment R for B may be extended with a location l to an environment R^l for $B; \rho$ by taking $R^l(\rho_i) = R(\rho_i)$ if $\rho_i \in FRV(B)$ and $R^l(\rho_i) = l$ otherwise. This is also an environment for B, ρ when $l \notin Loc_R$.

The functors $\Pi : P(B\ ; \rho) \longrightarrow P(B)$ and $\Psi : P(B, \rho) \longrightarrow P(B)$ used to interpret the quantifiers act on suitable objects τ as

$$\Pi(\tau)(R) = \prod_{l \in Loc} \tau(R^l) \qquad \Psi(\tau)(R) = \prod_{l \notin Loc_R} \tau(R^l)$$

where R is any location environment for B. That is

$$\begin{aligned} [\![B \vdash \forall\rho.\tau]\!](R) &= \textstyle\prod_{l \in Loc} \overline{\tau}(R^l) \\ [\![B \vdash \forall_*\rho.\tau]\!](R) &= \textstyle\prod_{l \notin Loc_R} \underline{\tau}(R^l) \end{aligned} \tag{4}$$

where $R \in REnv(B)$, $\overline{\tau} = [\![B; \rho \vdash \tau]\!]$ and $\underline{\tau} = [\![B, \rho \vdash \tau]\!]$. The additive is just ordinary quantification over individuals, whereas the multiplicative is a kind of fresh quantification: the quantified variable ranges over unused locations.

The interpretation of reference types is modified to be:

$$[\![B \vdash \rho\ \mathsf{ref}]\!]RW = \begin{cases} W & \text{if } W = \{R(\rho)\} \\ \emptyset & \text{otherwise} \end{cases}$$

for any location environment R and world W.

In the next section, we develop a language for region allocation and disposal via multiplicative polymorphism in region variables. It should be possible to do a similar thing for allocation of individual references using multiplicative polymorphism in location variables, but we have not worked through all the consequences of this approach.

4 Region Allocation and Disposal

In this section, we refine both the BRDM and the bunched region calculus to show how it can be used to support region allocation and disposal.

We introduce a bunched polymorphic region language with references, based on a first-order language given by Berdine and O'Hearn [2]. This is a variant of the language for allocation, strong update and disposal of first-order references given in given in [2]. It is founded on the first-order bunched lambda-calculus $\alpha\lambda$ but, for technical reasons (related to the soundness of disposal), is formulated in continuation-passing-style (CPS). Their language has a denotational semantics on a version of the BDM, so it can be integrated neatly into our approach.

We make no claims for the practical significance of our region language. Our primary aim is to show that denotational models that capture some of the most fundamental properties of region languages can be constructed using our methods. The fundamental extension to Berdine and O'Hearn's language is the addition of bunched region polymorphism.

Importantly, our language requires no new proof rules or reduction relations, so changes to the metatheory from Section 2 are minimal.

4.1 The Region Language

The types, τ, are generated from *storable types*, $\sigma := \top \mid \mathsf{int}$, as follows:

$$\tau := \sigma \mid I \mid \tau \wedge \tau \mid \tau \to \tau \mid \tau * \tau \mid \tau \mathrel{-\!\!*} \tau \mid \forall \rho.\tau \mid \forall_* \rho.\tau \mid \mathsf{a} \mid \mathsf{H}(B,\tau) \mid (\rho,\sigma)\mathsf{ref}$$

where we use a new type constant a, and type constructors H and ref.

A reference containing a value of type σ and located in region ρ will be of type $(\rho,\sigma)\mathsf{ref}$. An inhabitant of $\mathsf{H}(B,\tau)$ is a continuation that takes heaps and the regions in B to produce values of type τ. Thus the realm B in $\mathsf{H}(B,\tau)$ has similarities to the syntactic entitites known as effects in the typing judgements of the Tofte-Talpin system. The type a is used in types of continuations

$$0 : \mathsf{int}$$

$$\mathsf{succ} : \mathsf{int} \to \mathsf{int}$$

$$!_{\sigma,B,\rho} : ((\rho,\sigma)\mathsf{ref} * \top) \to (\sigma \to \mathsf{H}(B(\rho),\mathsf{a})) \to \mathsf{H}(B(\rho),\mathsf{a})$$

$$:=_{\sigma,B,\rho} : ((\rho,\sigma)\mathsf{ref} * \top) \to \sigma \to \mathsf{H}(B(\rho),\mathsf{a}) \to \mathsf{H}(B(\rho),\mathsf{a})$$

$$:\equiv_{\sigma,\sigma',B,\rho} : (\rho,\sigma)\mathsf{ref} \to \sigma' \to ((\rho,\sigma')\mathsf{ref} \mathrel{-\!\!*} \mathsf{H}(B(\rho),\mathsf{a})) \mathrel{-\!\!*} \mathsf{H}(B(\rho),\mathsf{a})$$

$$\mathsf{new}_{B,\rho} : ((\rho,\top)\mathsf{ref} \mathrel{-\!\!*} \mathsf{H}(B(\rho),\mathsf{a})) \to \mathsf{H}(B(\rho),\mathsf{a})$$

$$\mathsf{free}_{\sigma,B,\rho} : (\rho,\sigma)\mathsf{ref} \to \mathsf{H}(B(\rho),\mathsf{a}) \mathrel{-\!\!*} \mathsf{H}(B(\rho),\mathsf{a})$$

$$\mathsf{newregion}_B : (\forall_*\rho.\mathsf{H}((B,\rho),\mathsf{a})) \to \mathsf{H}(B,\mathsf{a})$$

$$\mathsf{freeregion}_B : \forall_*\rho.\mathsf{H}(B,\mathsf{a}) \to \mathsf{H}((B,\rho),\mathsf{a})$$

Fig. 2. Region Allocation and Reference Constants

$H(B,\mathsf{a})$ to describe alterations to the heap through allocation, deallocation and strong update of references. A fundamental idea in [2] is that additive types $\mathsf{H}(B,\mathsf{a}) \to \mathsf{H}(B,\mathsf{a})$ describe commands that do not alter the heap. In contrast, commands $\mathsf{H}(B,\mathsf{a}) \mathrel{-\!\!*} \mathsf{H}(B,\mathsf{a})$ can make such changes. This tracks a similar idea for logical connectives in separation logic.

As before, the well-formed types $B \vdash \tau$ are just those with $FRV(\tau) \subseteq FRV(B)$. The contexts over a realm B are generated as before, subject to these new types.

The well-formed terms are generated using the rules from Section 2 together with families of constants as shown in Figure 2. The subscripts on the constants indicate the indexing families. These will usually be omitted. The $\mathsf{newregion}$ and $\mathsf{freeregion}$ constants are well-typed over any realm containing B. The new, free, $!$, $:=$ and $:\equiv$ constants are well-typed over the realm B and context $\emptyset_*$.

The zero and successor constants have their usual meanings. There are constants new and free for allocation and deallocation of individual references.

The constant ! is dereferencing, whilst := is type-preserving (weak) update of references. We also have :≡ for type-changing (strong) update of references. Lastly, we have constants $\mathsf{newregion}$ and $\mathsf{freeregion}$ that allocate and dispose of entire regions.

The way to get a handle on these constants is to look at associated derived rules for manipulation of references. For example, supposing we are given appropriate terms N and K with

$$B \mid \Gamma \vdash N : \sigma' \qquad B \mid \Delta \vdash K : (\rho, \sigma')\mathsf{ref} \multimap \mathsf{H}(B, \mathsf{a})$$

and the abbreviations

$$\begin{aligned} \tau_1 &:= (\rho, \sigma)\mathsf{ref} \to \sigma' \to ((\rho, \sigma')\mathsf{ref} \multimap \mathsf{H}(B(\rho), \mathsf{a})) \multimap \mathsf{H}(B(\rho), \mathsf{a}) \\ \tau_2 &:= \sigma' \to ((\rho, \sigma')\mathsf{ref} \multimap \mathsf{H}(B(\rho), \mathsf{a})) \multimap \mathsf{H}(B(\rho), \mathsf{a}) \\ \tau_3 &:= ((\rho, \sigma')\mathsf{ref} \multimap \mathsf{H}(B(\rho), \mathsf{a})) \multimap \mathsf{H}(B(\rho), \mathsf{a}) \end{aligned}$$

for types. Then, given a reference $B \mid \Gamma \vdash M : (\rho, \sigma)\mathsf{ref}$, we have a derivation as follows:

$$\dfrac{\dfrac{\dfrac{B \mid \Gamma \vdash {:\equiv} : \tau_1 \quad B \mid \Gamma \vdash M : (\rho, \sigma)\mathsf{ref}}{B \mid \Gamma \vdash \mathrm{app}({:\equiv}, M) : \tau_2} \quad \dfrac{\vdots}{N}}{B \mid \Gamma \vdash \mathrm{app}(\mathrm{app}({:\equiv}, M), N) : \tau_3} \quad \dfrac{\vdots}{K}}{B \mid \Gamma, \Delta \vdash \mathrm{app}_*(\mathrm{app}((\mathrm{app}({:\equiv}, M), N), K) : \mathsf{H}(B, \mathsf{a})}$$

and we normally abbreviate the root term as

$$M :\equiv N; K$$

in order to have the program written in sequence. Notice that the terms $M :\equiv N$ and K are combined multiplicatively, so (according to the sharing interpretation) use disjoint store. For this reason, the reference $M : (\rho, \sigma)\mathsf{ref}$ may not be used in K. It is replaced by the reference of type $(\rho, \sigma')\mathsf{ref}$ which K expects. The meanings of the other constants may be revealed in the same way. They have been omitted for reasons of space and should not be too hard to reconstruct. See also [2] for similar derivations.

Turning to the constants for region allocation and disposal there are derived rules

$$\dfrac{B \mid \Gamma \vdash M : \forall_* \rho. \mathsf{H}((B', \rho), \mathsf{a})}{B \mid \Gamma \vdash \mathsf{newregion}\ ; M : \mathsf{H}(B', \mathsf{a})} \qquad \dfrac{B \mid \Gamma \vdash M : \mathsf{H}(B', \mathsf{a})}{B, \rho \mid \Gamma \vdash \mathsf{freeregion}\ \rho\ ; M : \mathsf{H}((B', \rho), \mathsf{a})}$$

for allocation and disposal of regions. Furthermore, we have a derived rule

$$\dfrac{B, \rho \mid \Gamma \vdash M : \mathsf{H}((B', \rho), \mathsf{a})}{B \mid \Gamma \vdash \mathsf{newregion}\ ; \Lambda_* \rho. M : \mathsf{H}(B', \mathsf{a})} \quad (\rho \notin \mathit{FRV}(\Gamma))$$

provided the side-condition is met.

These rules exhibit the close connection between quantification and allocation and between instantiation and disposal. We could have used such proof rules directly to define region allocation and disposal. However, the proof rules for the multiplicative quantifier and their interpretations are sufficient to enable us to write these commands as constants. Since no special proof rules are needed, the new region calculus is just a straightforward, applied variant of the bunched polymorphic calculus. Changes to the metatheory are minimal. It remains to show that the new constants can all be interpreted in a suitable model.

For newregion, the intention is that the new region will be given the name ρ and that this will be associated with a set of locations that is disjoint from those used for any prior part of M. The operation freeregion can be used whenever the region to be disposed is unused by the rest of the computation. Recall that for this kind of CPS expression there is a Hoare-like relationship where the context of the conclusion is the pre-condition and the context of the premise is the post-condition for the term in the conclusion. The realms in these derived rules are then seen to verify that our intentions for newregion and freeregion are met.

As an example of a continuation we have

$$\begin{gathered}\mathsf{newregion}; \Lambda_* \rho'.\mathsf{new}; \lambda_* y : (\rho', \top)\mathsf{ref}.y :\equiv 42; \lambda_* z : (\rho', \mathsf{int})\mathsf{ref}.\\ \mathsf{newregion}; \Lambda_* \rho.\mathsf{new}; \lambda_* x : (\rho, \mathsf{int})\mathsf{ref}.!z; \lambda w : \mathsf{int}.x := w;\\ \mathsf{freeregion}\ \rho'; x := 36; \mathsf{freeregion}\ \rho; K : \mathsf{H}(B, \mathsf{a}) \ .\end{gathered}$$

The uses of ';' in the term are abbreviations for applications of constants that are better read as concatenations of continuations. This (rather trivial) program opens a region ρ', creates a new reference y in ρ', updates (strongly) the value in y to 42, opens another region ρ, creates a new integer reference x in ρ, updates x with the value held in y, frees ρ' (including y), updates x with the value 36, frees region ρ (including x) and continues with K. This example illustrates the fact that it is possible to open first one region then another, dispose the first and then the second. The safety of this is guaranteed by the typing. The reader can doubtless see that the derivation of such a term is rather cumbersome. In fact, in order to type this term we need to add further constants that encapsulate the equality $[\![B \vdash \tau_1 * (\sigma \wedge \tau_2)]\!] = [\![B \vdash (\tau_1 \wedge \sigma) * \tau_2]\!]$ that holds in the model (see also the constant hoist of [2]). We conjecture that it should be possible (with a fair bit of work) to write longer programs that make more interesting use of regions.

4.2 A Model for the Region Language

We combine the BRDM with the denotational model from [2] to give a model of the present language.

Since arbitrarily many references may be allocated using any one region variable we are forced to use a model in which all regions are assumed to be infinite. Let *InfReg* be the set of all countable subsets of *Loc* with countable complement. The complement must be countable in order to support allocation of regions.

Redefine a *region environment* R for a realm B to be a function $R : \mathit{FRV}(B) \longrightarrow \mathit{InfReg}$ such that Loc_R has a countable complement and if ρ and ρ' are separated by a ',' in B then $R(\rho) \cap R(\rho') = \emptyset$. Here, let $\overline{\mathit{Loc}_R}$ be the set of infinite regions U such that $\mathit{Loc}_R \cup U$ has countable complement.

A *(typed) world* is a finite partial function from locations to storable types. That is, it is a finite set of locations together with a store typing. For worlds v and w write $v \cap w = \emptyset$ when the domains of the two functions do not overlap. When this is the case write $v \cup w$ for the partial function whose graph is the union of the graphs of v and w. Worlds constitute a partial commutative monoid $\mathit{Wld} = (\mathit{Loc} \overset{\mathit{fin}}{\rightharpoonup} \sigma, \emptyset, *)$ where

$$(v * w) = \begin{cases} v \cup w & \text{if } v \cap w = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

and $\emptyset$ is the partial function with empty domain. In what follows, we also regard *Wld* as a discrete category. This induces a CDCC structure on $\mathit{Set}^{\mathit{Wld}}$, similar to that of the BDM, and following Day's method [4,5].

A *semantic type* over a realm B, is a family τ of functors $\tau R : \mathit{Wld} \longrightarrow \mathit{Set}$ indexed by $R \in \mathit{REnv}(B)$. An arrow between semantic types is a family of natural transformations $f_R : \tau_R \longrightarrow \tau'_R$ indexed by $R \in \mathit{REnv}(B)$. This gives a category $P(B)$ which is a CDCC, since $\mathit{Set}^{\mathit{Wld}}$ is a CDCC. Furthermore, it extends to an indexed category over the category of realms, following the BRDM construction almost exactly. In particular, we recover the functors Π and Ψ used to interpret the additive and multiplicative universal quantifiers. However, note that the index U in (1) and (2) is now drawn from $\overline{\mathit{Loc}_R}$ rather than $\mathcal{W}$.

A *storable value* is an integer (of type int) or the unit (of type $\top$). That is, they are values of some storable type. A *heap* is a finite partial function from locations to storeable values. Let $[\![\mathsf{int}]\!]$ be the set of integers and $[\![\top]\!]$ be the one-element set $\top$.

Define

$$H(R, w) = \prod_{l \in \mathit{dom}(w) \cap \mathit{Loc}_R} [\![w(l)]\!]$$

to be the set of heaps compatible with w and R for the realm B. Note the compatibility constraint relating worlds to regions.

Interpret a type $B \vdash \tau$ as a function from region environments for B to objects of $\mathit{Set}^{\mathit{Wld}}$. The interpretation is specified in Figure 3, where $\{e\}$ is a given singleton set, $\mathbf{2} = \{0, 1\}$ is a given two-element set, $\Longrightarrow$ is the function

$$[\![B \vdash \mathsf{int}]\!]Rw = \mathbb{Z} \qquad [\![B \vdash \top]\!]Rw = \{\top\} \qquad [\![B \vdash \mathsf{a}]\!]Rw = \mathbf{2}$$

$$[\![B \vdash I]\!]Rw = \begin{cases} \{e\} \text{ if } W = \emptyset \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$[\![B \vdash (\rho, \sigma)\mathsf{ref}]\!]Rw = \begin{cases} \{l\} \text{ if } w = [l : \sigma] \text{ and } l \in R(\rho) \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$[\![B \vdash - \odot -]\!] \text{ for } \odot = \wedge, \to, *, -\!\!* \;\; \text{using the CDCC P(B)}$$

$$[\![B \vdash \mathsf{H}(B', \tau)]\!]Rw = (H(R', w) \Longrightarrow [\![B' \vdash \tau]\!]R'w)$$

$$[\![B \vdash \forall\rho.\tau]\!] = \Pi([\![B; \rho \vdash \tau]\!]) \qquad [\![B \vdash \forall_*\rho.\tau]\!] = \Psi([\![B, \rho \vdash \tau]\!])$$

Fig. 3. Interpretation of CPS language

space, $R' = R\restriction_{B'}$ and the cases using Π and Ψ expand as in (3) but with U drawn from the set $\overline{Loc_R}$ rather than $\mathcal{W}$.

The motivations for our intepretations are very similar to those explained in [2]. In particular, interpretations of int, $\top$, I, $\wedge$, $\to$, $*$ and $-\!\!*$ are just the same. Notice how a continuation of type $\mathsf{H}(B, \tau)$ is interpreted as a function that takes heaps on the store (restricted to B) to values of type τ. In particular, continuations in $H(B, \mathsf{a})$ interrogate the heap (on B) to provide Boolean answers. A fundamental idea in [2] is to try to get a type system that lies in a propositions-as-types correspondence with a variant of separation logic [20]. This, together with the regions, drives the interpretation of $(\rho, \sigma)\mathsf{ref}$. Roughly speaking, we have $x : (\rho, \sigma)\mathsf{ref}$ corresponding to $x \mapsto v$, where $v : \sigma$ at location l in the heap on the singleton $\{l\}$ (and also l is in the region ρ).

The interpretation of terms other than the reference constants and region operators follows the pattern of interpretation from the BRDM. The interpretation of reference constants are modifications of the interpretations given in [2]. In particular, the interpretation of new requires that the interpretations of region variables are infinite. In practice, finite regions would be sufficient: their size would be bounded above by the number of individual references allocated within them.

Consider the interpretation of $\mathsf{newregion}_B$. Now, for any suitable R and w,

$$\begin{aligned}&[\![B \vdash \forall_*\rho.\mathsf{H}((B,\rho),\mathsf{a})]\!]Rw\\ =\ &\Psi[\![B,\rho \vdash \mathsf{H}((B,\rho),\mathsf{a})]\!]Rw\\ =\ &\textstyle\prod_{U\in Loc_R\#\cap\overline{Loc_R}}([\![B,\rho \vdash \mathsf{H}((B,\rho),\mathsf{a})]\!]R^U w)\\ =\ &\textstyle\prod_{U\in locR\#\cap\overline{Loc_R}}(H(R^U,w) \Longrightarrow \mathbf{2})\end{aligned}$$

holds. Selecting any region $U \in Loc_R\# \cap \overline{Loc_R}$ we have a projection function $\pi^U : [\![B \vdash \forall_*\rho.\mathsf{H}((B,\rho),\mathsf{a})]\!]Rw \longrightarrow (H(R^U,w) \Longrightarrow \mathbf{2})$. There are functions $F_{R,U,w} : (H(R^U,w) \Longrightarrow \mathbf{2}) \longrightarrow (H(R,w) \Longrightarrow \mathbf{2})$ such that for $k : H(R^U,w) \longrightarrow \mathbf{2}$ and $h \in H(R,w)$, we have $F_{R,U,w}(k)(h) = k(h')$, where $h'(l) = h(l)$ if $l \in Loc_R$, otherwise $h'(l) = 0$ if $w(l) = \mathsf{int}$ or $h'(l) = \top$ if $w(l) = \top$. This gives $F_{R,U,w} \circ \pi^U : [\![B \vdash \forall_*\rho.\mathsf{H}((B,\rho),\mathsf{a})]\!]Rw \longrightarrow [\![B \vdash \mathsf{H}(B,\mathsf{a})]\!]Rw$. To get the required arrow over $[\![B]\!]$, for each R and w select some such U and use the functions $F_{R,U,w} \circ \pi^U$. Note that we needed that Loc_R has a countable complement, but that it then follows that Loc_{R^U} has countable complement so we may use $\mathsf{newregion}$ again.

Now consider the interpretation of a term $\mathsf{freeregion}_B$. We need an arrow $[\![B \vdash \emptyset]\!] \longrightarrow [\![B \vdash \forall_*\rho.\mathsf{H}(B,\mathsf{a}) \to \mathsf{H}((B,\rho),\mathsf{a})]\!]$ over B. This is given using for all R and w and $U \in locR\#\cap\overline{Loc_R}$ the functions $G_{U,R,w} : (H(R,w) \Longrightarrow \mathbf{2}) \longrightarrow (H(R^U,w) \Longrightarrow \mathbf{2})$ with $G_{U,R,w}(k)(h) = k(h')$, where $k \in H(R,w) \Longrightarrow \mathbf{2}$, $h \in H(R^U,w)$ and $h' = h\restriction_{Loc_{R'}}$.

Notice how the semantics of $\mathsf{newregion}$ and $\mathsf{freeregion}$ extend and restrict, respctively, the part of the heap that may be used.

Finally, we remark that, with appropriate interpretations of the remaining constants, the soundness property (Theorem 3.3) carries over to this setting.

5 Conclusions and Comparisons

There are many possible avenues which have been left unexplored. Some region calculi allow for region (location) constants that may be substituted for region (location) variables: our model could be modified to support this by altering the definition of substitution between realms. In [14], the BDM is refined in various ways, making it possible to give semantics to languages with more complex features, including recursion. Similar steps could be taken with this model. At present our model only allows very simple references, following [2]. This should be extended to treat references to other values, such as functions and references.

An alternative way to handle allocation and deallocation of individual references would be to type references using location rather than region variables and to make appropriate changes to the constants and the model.

The relationship between the languages we have presented and previous region and location languages remains to be clarified. Our reference language is in CPS and only puts references in regions. In contrast, the language in [25] is in direct style but has effect labellings on types and puts almost all values in regions. It may be worthwhile to see whether other region languages, such as the linearly typed languages described in [6,13,27,28,29] and the type-and-effect systems described in [9,12,25], can be given a denotational semantics by translating them into ours. This would also give a measure of the expressiveness of our language relative to others.

In most region (and location) systems, the regions obey a stack discipline. In contrast, our system does not require this: we may interleave allocations and disposals at will and the disjointness conditions implicit in the types will give a static guarantee of soundness. In some situations this could allow for a more efficient recycling of memory. A number of other authors have been led to consider typed region calculi of this kind, for example [6,9].

Bunched existential quantifiers are given in [3] and could be exploited in region models. An extreme form of the multiplicative existential, that hides all of the regions used to form the representation type, has some similarities to the region function closures of [25] and could also be used to describe abstract data-types with encapsulated state.

It may not be unreasonable to expect some relationship between the multiplicative quantifier in (4) and the freshness quantifiers of [7].

We remind the reader that bunched polymorphism and first-order bunching are independent. Both happen to be supported by the BRDM, but the use of bunched polymorphism together with the category of realms should be regarded as a general technique for building models of languages with region and location parameters.

References

[1] A. Banerjee, N. Heintze, and J.G. Riecke. Region analysis and the polymorphic lambda calculus. In *LICS '99*, pages 88–97. IEEE Press, 1999.

[2] J. Berdine and P. O'Hearn. Strong update, disposal and encapsulation in bunched typing. In *Mathematical Foundations of Programming Semantics, MFPS22*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006. This volume.

[3] M. Collinson, D. Pym, and E. Robinson. On Bunched Polymorphism (Extended Abstract). In *Computer Science Logic*, volume 3634 of *LNCS*. Springer, 2005.

[4] B.J. Day. On closed categories of functors. In *Proceedings of the Midwest Category Seminar*, volume 137 of *LNM*. Springer, 1970.

[5] B.J. Day. An embedding theorem for closed categories. In *Proceedings of the Sydney Category Seminar 1972/73*, volume 420 of *LNM*. Springer, 1973.

[6] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In *Programming Languages and Systems, ESOP 2006*, volume 3924 of *LNCS*. Springer, 2006.

[7] M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[8] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupres dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[9] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory-management. In *Principles and Practice of Declarative Programming, PPDP01*, pages 175–186. ACM Press, 2001.

[10] F. Henglein, H. Makholm, and H. Niss. Effect types and region-based memory-management. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[11] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1999.

[12] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *POPL '88*, pages 45–57. ACM Press, 1988.

[13] G. Morrisett, A. Ahmed, and M. Fluet. L3: A linear language with locations. In *TLCA'05*, volume 3461 of *LNCS*, pages 293–307. Springer, 2005.

[14] P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13:747–796, 2003.

[15] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[16] F.J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, Syracuse, 1982.

[17] D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: The semantics of **BI**. *Theoretical Computer Science*, 315(1):257–305, 2004.

[18] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002. Errata at: http://www.cs.bath.ac.uk/ pym/BI-monograph-errata.pdf.

[19] J.C. Reynolds. The essence of algol. In J.W. de Bakker and J.C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[20] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS '02*, pages 55–74. IEEE Press, 2002.

[21] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Technical report, Oxford University Computing Laboratory Programming Research Group, 1971.

[22] R.A.G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52:969–989, 1987.

[23] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[24] M. Tofte. A brief introduction to regions. In *International Symposium on Memory Management '88*, pages 186–195. ACM Press, 1998.

[25] M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *POPL '94*, pages 188–201. ACM Press, 1994.

[26] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 2:109–176, 1997.

[27] D. Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[28] D. Walker and J.G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, volume 2071 of *LNCS*, pages 177–206. Springer, 2001.

[29] D. Walker and K. Watkins. On regions and linear types. In *ICFP '01*, pages 181–192. ACM Press, 2001.