Strong Update, Disposal, and Encapsulation in Bunched Typing

Josh Berdine

Microsoft Research[1]

Peter W. O'Hearn

Queen Mary, University of London

Abstract

We present a bunched intermediate language for strong (type-changing) update and disposal of first-order references. In contrast to other substructural type systems, the additive constructs of bunched types allow the encapsulation of state that is shared by a collection of procedures.

Key words: bunched typing, separation logic, strong update, disposal, encapsulation, continuation-passing style

1 Introduction

Substructural type systems give a way to control the usage of resources in programs. The first example of such resource control was given by syntactic control of interference (SCI) [17], which used an affine type system (one that prohibits Contraction but not Weakening) to control aliasing of first-order references. This fed into the development of bunched typing and logic [13,15,12], in which substructural features live on the same footing as intuitionistic typing. The sense of "same footing" is summed up succinctly in the category-theoretic models of bunched typing: a model is a doubly cartesian closed category, i.e., a single category that admits both cartesian closed and symmetric monoidal closed structures. The full intuitionistic connectives of bunched typing were used to define an extended version of SCI that dealt with problems in the original version to do with recursion and continuations [12].

In a parallel line of development, type systems inspired by linear logic [7] were given that allowed for destructive update and safe disposal of memory in

[1] Work performed while supported by the EPSRC at Queen Mary, University of London

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a purely functional setting [23,24,25]. These languages in turn influenced imperative typed intermediate languages, that allow for safe disposal and strong update, where the type of a reference is allowed to change over time [10,26,4].

This paper was partly inspired by recent work reported in [9], which presented some of the ideas from typed intermediate languages in a pleasantly pared-down form, and which took some steps towards bunched logic or type theory by defining a kind of spatial possible worlds model. (Those new to the topic, in particular, are encouraged to read [9].) This helped us to better understand some of the issues involved, to put our finger on some of the differences, and now to take some steps of our own in the reverse direction (from bunched type theory towards typed intermediate languages). We refer to [9] for further discussion of how linear typing supports strong update and disposal, and their use in intermediate languages, and to [25,1] for general background on substructural type systems and references to the further literature.

The essential point is that these parallel lines of development have been applied in different ways. In the line with SCI and bunched typing, aliasing is controlled, and full-strength additive connectives are allowed. But the languages studied do not allow strong update or disposal. In the linear typing line, strong update and disposal are allowed, but additive connectives, in particular the additive product & of linear logic, are not.

A natural question, then, is whether bunched typing can also incorporate safe disposal and strong update. The answer is not immediate, because of examples from [12] which show that a number-of-uses reading, which provides the intuitive rationale for why strong update and disposal are sound in linear typing, is incompatible with bunched typing. A second question is whether any advantages are obtained by allowing the use of additives as in bunched typing. The purpose of this paper is to investigate these questions.

We present a language based on bunched types that indeed does support disposal and strong update. The language is presented in CPS (continuation-passing style) form. The type of the disposal operation is:

$$\mathit{free}_{\psi} : \mathsf{a} \to (\psi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a})$$

Here, a is the type of answers, $\to$ is the intuitionistic function type, and the multiplicative type $\psi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}$ is given a spatial reading: if you give me a reference occupying separate storage from what I have, I can produce an answer. The reading of free_{ψ} is that if you give me a continuation that references certain storage, and then a reference in completely separate storage, I can dispose the reference and then safely run the continuation. Notice that this reading says nothing about number of uses; safety of disposal stems from intuitions about separation (mirroring separation logic [16] and alias types [26]).

The answer type a here denotes command continuations in the sense of Reynolds [18]: a command continuation accepts a state and then runs. A conventional command is an additive function of type $\mathsf{a} \to \mathsf{a}$. We also have

the possibility of using "multiplicative commands" of type $\mathsf{a} \multimap \mathsf{a}$.

Because our language is in CPS, it should be regarded as (the bones of) an intermediate language rather than a source language. Indeed, it is nontrivial to define a direct-style language that supports disposal. The reason can be seen in terms of two expected additive typing rules:

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash (M, N) : \sigma \times \tau} \qquad\qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N : \tau}$$

If we think in terms of call-by-value evaluation, if M and N both dispose a common reference before producing a value, then (M, N) will lead to an error where the same reference is disposed twice. Similar remarks apply to $M\, N$.

A possible way out of this problem would be to use a form of Hoare typing, that is, a type system that has both preconditions and postconditions. A lower-tech way out is to use CPS. For example, a CPS typing rule

$$\frac{\Delta \vdash K : \mathsf{a}}{\Gamma \vdash M; K : \mathsf{a}}$$

can be thought of as typing M relative to pre- and post-conditions Γ and Δ. The pre/post analogy can be seen clearly in some typed intermediate languages, such as [21].

An eventual aim is to obtain a direct-style source language with all of bunched types that supports update and disposal. However, a prior question is whether bunched types can express these features at all. It is more sensible, scientifically, as a first step to investigate this question in terms of an existing type system, that of [15,12], rather than an hypothesized, not-yet-existing, system of higher-order pre/post typing using bunches. The formulation of such a system is a topic for future research, as is connections with separation logic (as might be hoped for given the connection between separation logic and bunched logic [8]).

Given that we have the full additives of bunched typing, the next question is whether this gives any added flexibility. In sum, we are able to transfer to a language with disposal and strong update a feature emphasized by Reynolds in his work on SCI: encapsulation of state that is shared by a collection of procedures.

The basic idea can be explained via an example of a procedure for generating counter objects. The methods have type $\mu = ((\mathsf{int} \to \mathsf{a}) \to \mathsf{a}) \wedge (\mathsf{a} \to \mathsf{a}) \wedge (\mathsf{a} \multimap \mathsf{a})$, an additive product of the type for a method to get the value of a counter, to increment, and to destroy it when done. The first two methods use the additive function type. They preserve the type and shape of the background heap and can be used one after the other. The destroy method is a multiplicative command, where the input continuation operates on a smaller heap than the command. The multiplicative command cannot be postcomposed with either of the other two methods, because there would not be a match-up of heap shape (semantically) or context (type theoretically). The

type of a client for such an object is $(\mu \multimap * \mathsf{a}) \to \mathsf{a}$, where the $\multimap *$ captures that the client is unaware of the state shared by the methods. In Section 3 we will derive a term (which presently is unlikely to make sense)

$$\mathsf{true} \vdash \lambda k.\, new\, \lambda_* x.\, x{:=}0\, \$\, \lambda_* x.\, k_{\sqcup}(mb) : (\mu \multimap * \mathsf{a}) \to \mathsf{a}$$

for generating counters, where mb is code for the method bodies.

As far as we are aware, this encapsulation behavior cannot be represented using monomorphic fragments of linearly-typed languages as in [23,25], because there a value of additive type $A \to B$ cannot contain any identifier denoting a linear object. And the reference is the quintessential linear object. Consequently, it is not possible to construct a collection of additive functions that share access to hidden state.

Before continuing we should say that the remarks we have made in this section should not be construed as ultimate arguments in favor of bunched typing. For instance, shared state can be represented by adding type quantification to a linear language. This is why we have included the qualification "monomorphic". Our remarks are intended just to pinpoint what some of the differences are when compared to previous work, and we do this simply to add information to the picture of substructural typing rather than to argue for ultimate advantage. Also, our toy language is purposely pared-down in order to let us investigate the problems tackled in this paper with a minimum of distraction. For this reason we do not include higher-order references or polymorphism, even though these would be necessary to have an expressive intermediate language as in, e.g., [10]; this is a direction for future work [6], and [5] gives a semantics of the language presented here extended with polymorphic location and region types.

2 Type System

We specialize the monomorphic bunched type theory of [15] by specifying several base types and constants. (We omit the coproduct types as they will not be required, though they are present in the model to be presented later.) The base types are the type int of integers, a primitive type a to use as the answer type in continuation-passing style, and primitive types $\psi\,\mathsf{ref}$ of references containing values of storable types, which are either int or true. We consider these as the two storable types just to allow us to investigate changing types (strong update).

The types and contexts of the system are given by the grammar:

$\phi, \psi ::= \mathsf{int} \mid \mathsf{true}$	storable types
$\varphi, \sigma, \tau ::= \mathsf{a} \mid \psi \mid \tau \wedge \tau \mid \tau \to \tau \mid \mathsf{emp} \mid \psi\,\mathsf{ref} \mid \tau * \tau \mid \tau \multimap * \tau$	types
$\Sigma, \Delta, \Gamma ::= x{:}\tau \mid \mathsf{true} \mid \Gamma\,;\Gamma \mid \mathsf{emp} \mid \Gamma\,,\Gamma$	bunches

where x ranges over some set of identifiers.

The ";" bunch-combining operation is additive, and will satisfy Contraction and Weakening, while "," is the multiplicative form which will satisfy

Table 1 Typing Rules

Identifiers and Constants

$$\frac{}{x{:}\tau \vdash x : \tau} \qquad \frac{\mathcal{S}(c) = \tau}{\mathsf{true} \vdash c : \tau}$$

Structural Rules

$$\frac{\Gamma \vdash M : \tau}{\Delta \vdash M : \tau}\ \Gamma \equiv \Delta \qquad \frac{\Gamma(\Delta) \vdash M : \tau}{\Gamma(\Delta\,;\Delta') \vdash M : \tau} \qquad \frac{\Gamma(\Delta\,;\Delta') \vdash M : \tau}{\Gamma(\Delta) \vdash M[i(\Delta)/i(\Delta')] : \tau}\ \Delta \cong \Delta'$$

Additives

$$\frac{}{\mathsf{true} \vdash \mathsf{true} : \mathsf{true}} \qquad \frac{\Gamma(\mathsf{true}) \vdash M : \tau \qquad \Delta \vdash N : \mathsf{true}}{\Gamma(\Delta) \vdash \mathit{let}\ \mathsf{true} = N\ \mathit{in}\ M : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Delta \vdash N : \tau}{\Gamma\,;\Delta \vdash (M\,;N) : \sigma \wedge \tau} \qquad \frac{\Gamma(x{:}\sigma\,;y{:}\tau) \vdash M : \varphi \qquad \Delta \vdash N : \sigma \wedge \tau}{\Gamma(\Delta) \vdash \mathit{let}\ (x{:}\sigma\,;y{:}\tau) = N\ \mathit{in}\ M : \varphi}$$

$$\frac{\Gamma\,;x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.\,M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Delta \vdash N : \sigma}{\Gamma\,;\Delta \vdash M\,N : \tau}$$

Multiplicatives

$$\frac{}{\mathsf{emp} \vdash \mathsf{emp} : \mathsf{emp}} \qquad \frac{\Gamma(\mathsf{emp}) \vdash M : \tau \qquad \Delta \vdash N : \mathsf{emp}}{\Gamma(\Delta) \vdash \mathit{let}\ \mathsf{emp} = N\ \mathit{in}\ M : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Delta \vdash N : \tau}{\Gamma\,,\Delta \vdash (M\,,N) : \sigma * \tau} \qquad \frac{\Gamma(x{:}\sigma\,,y{:}\tau) \vdash M : \varphi \qquad \Delta \vdash N : \sigma * \tau}{\Gamma(\Delta) \vdash \mathit{let}\ (x{:}\sigma\,,y{:}\tau) = N\ \mathit{in}\ M : \varphi}$$

$$\frac{\Gamma\,,x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda_* x{:}\sigma.\,M : \sigma \mathrel{-\!\!*} \tau} \qquad \frac{\Gamma \vdash M : \sigma \mathrel{-\!\!*} \tau \qquad \Delta \vdash N : \sigma}{\Gamma\,,\Delta \vdash M \llcorner N : \tau}$$

neither. Each has its own unit. The notation $\Gamma(\Delta)$ indicates that bunch Δ appears as a sub-bunch of Γ. We use $\equiv$ to denote the smallest congruence relation on bunches satisfying commutative monoid laws for ";" and true and, separately, for "," and emp. $\Gamma \cong \Delta$ indicates that Γ and Δ are isomorphic as trees; i.e., one can be obtained from the other by a suitable renaming of identifiers. The set of identifiers appearing in a bunch is notated $i(\Gamma)$. The typing rules are shown in Table 1, where $\mathcal{S}$ is an as-yet-unspecified association of constants to types. The last two structural rules are the bunched versions of Weakening and Contraction, while the first encompasses Exchange by its reference to the coherent equivalence (which is commutative).

The terms of the language are shown in the conclusions of the typing rules, and the constants (we are now specifying $\mathcal{S}$) are the following:

$$z : \mathsf{int} \quad \forall z \in \mathbb{Z}$$

$$
\begin{aligned}
+ &: \mathsf{int} \wedge \mathsf{int} \to \mathsf{int} \\
\mathit{swap}_{\psi,\phi} &: (\psi \wedge \phi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}) \to (\psi\,\mathsf{ref} \wedge \phi \mathrel{-\!\!*} \mathsf{a}) \\
\mathit{new} &: (\mathsf{true}\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}) \to \mathsf{a} \\
\mathit{free}_{\psi} &: \mathsf{a} \to (\psi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}) \\
!_{\psi} &: (\psi \to \mathsf{a}) \to (\psi\,\mathsf{ref} * \mathsf{true} \to \mathsf{a}) \\
:=_{\psi} &: \mathsf{a} \to ((\psi\,\mathsf{ref} * \mathsf{true}) \wedge \psi \to \mathsf{a}) \\
\mathit{frame}_{\varphi} &: (\mathsf{a} \to \mathsf{a}) \to ((\varphi \mathrel{-\!\!*} \mathsf{a}) \to (\varphi \mathrel{-\!\!*} \mathsf{a})) \\
\mathit{hoist}_{\sigma,\psi,\tau} &: (\sigma * (\psi \wedge \tau)) \to ((\sigma \wedge \psi) * \tau)
\end{aligned}
$$

We use continuation-transformer types (e.g., of form $(\sigma \to \mathsf{a}) \to (\tau \to \mathsf{a})$), as opposed to continuation-second types (e.g., of form $\tau \to (\sigma \to \mathsf{a}) \to \mathsf{a}$), because, despite some syntactic complications (e.g., the \$ syntax later), we find them easier to understand, following a rough analogy between $M : (\sigma \mathrel{-\!\!*} \mathsf{a}) \to (\tau \mathrel{-\!\!*} \mathsf{a})$ and the Hoare triple $\{\tau\}\ \mathrm{M}\ \{\sigma\}$.

The constant *swap* exchanges the type held within the reference and the one without (following [9]). We then have constants for allocation and deallocation. Next, there are operations for lookup and weak (type-preserving) update, which are more additive in character. Then there is a constant for adding invariants following the Frame rule of separation logic.

The role of the last constant is to teach the type system that storable types ψ are pure, meaning that they are insensitive to the heap, and hence can be moved in and out of $\wedge$ and $*$. In separation logic such facts are gotten via logical implications which stem from the fact that the two conjunctions, $\wedge$ and $*$, coincide for pure types. For the type theory, we will exhibit a model where $[\![\sigma * (\psi \wedge \tau)]\!] = [\![(\sigma \wedge \psi) * \tau]\!]$, so *hoist* is the identity function.

We could include a constant for recursion, with the usual type, without difficulty; it would just require us to use domains in place of sets in the model defined later. The next sections include examples and intuitive explanation of how these constants work.

Notation

We have played a little game in the term-forming rules for the multiplicative and additive products, using "," and ";" in the notation for pairs. This allows us to introduce some syntactic sugar to alleviate the inconvenience of the distinction between bunches (contexts) and types, which does not appear in the semantics. First, we write $\overline{\Gamma}$ for the "type of" Γ, that is, Γ without any identifiers, and with "," replaced by $*$ and ";" by $\wedge$. Similarly, we write $\underline{\Gamma}$ for Γ without any types. Then, for any Γ, we obtain:

$$\Gamma \vdash \underline{\Gamma} : \overline{\Gamma}$$

Additionally, we will destruct arbitrary bunches using *let* $\Sigma = N$ *in* M. When this syntactic sugar is defined by induction on Σ, we obtain admissibility (the case when Σ has form $x{:}\sigma$ is Cut) of the first rule in Table 2. Also, we introduce the obvious syntactic sugar for abstracting and then destructing

Table 2 Admissible and Derivable Typing Rules

$$\frac{\Gamma(\Sigma) \vdash M : \tau \qquad \Delta \vdash N : \overline{\Sigma}}{\Gamma(\Delta) \vdash \mathit{let}\ \Sigma = N\ \mathit{in}\ M : \tau} \qquad \frac{\Gamma\,;\Delta \vdash M : \tau}{\Gamma \vdash \lambda\Delta.\, M : \overline{\Delta} \to \tau} \qquad \frac{\Gamma\,,\Delta \vdash M : \tau}{\Gamma \vdash \lambda_*\Delta.\, M : \overline{\Delta} \mathrel{-\!\!*} \tau}$$

$$\frac{\Gamma \vdash M : \psi\,\mathsf{ref} \wedge \phi \quad \Delta \vdash K : \psi \wedge \phi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}}{\Gamma\,,\Delta \vdash \mathit{swap}\_M\,\$\,K : \mathsf{a}} \qquad \frac{\Gamma \vdash M : \psi\,\mathsf{ref} \quad \Delta \vdash K : \psi \wedge \psi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}}{\Gamma\,,\Delta \vdash !!_\psi\_M\,\$\,K : \mathsf{a}}$$

$$\frac{\Gamma \vdash K : \mathsf{true}\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}}{\Gamma \vdash \mathit{new}\,K : \mathsf{a}} \qquad \frac{\Gamma \vdash M : \psi\,\mathsf{ref} \quad \Gamma \vdash N : \phi \quad \Delta \vdash K : \phi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}}{\Gamma\,,\Delta \vdash M{:}\equiv_{\psi,\phi} N\,\$\,K \stackrel{\text{def}}{=} {:}\equiv_{\psi,\phi} K\_(M\,;N) : \mathsf{a}}$$

$$\frac{\Gamma \vdash M : \psi\,\mathsf{ref} \quad \Delta \vdash K : \mathsf{a}}{\Gamma\,,\Delta \vdash \mathit{free}_\psi\_M\,\$\,K : \mathsf{a}} \qquad \frac{\Gamma \vdash M : \psi\,\mathsf{ref} \quad (\Gamma\,,\Delta)\,;\Sigma \vdash K : \psi \to \mathsf{a}}{(\Gamma\,,\Delta)\,;\Sigma \vdash !_\psi\,M\,\$\,K \stackrel{\text{def}}{=} !_\psi\,K\,(M\,,\mathsf{true}) : \mathsf{a}}$$

$$\frac{\Gamma \vdash M : \psi\,\mathsf{ref} \quad (\Gamma\,,\Delta)\,;\Sigma \vdash N : \psi \quad (\Gamma\,,\Delta)\,;\Sigma \vdash K : \mathsf{a}}{(\Gamma\,,\Delta)\,;\Sigma \vdash M{:}=_{\psi,\phi} N\,\$\,K \stackrel{\text{def}}{=} {:}=_{\psi,\phi} K\,((M\,,\mathsf{true})\,;N) : \mathsf{a}}$$

$$\frac{\Gamma \vdash C : \mathsf{a} \to \mathsf{a} \quad \Gamma \vdash K : \varphi \mathrel{-\!\!*} \mathsf{a} \quad \Delta \vdash F : \varphi}{\Gamma\,,\Delta \vdash \mathit{frame}_\varphi\,C\,K\_F : \mathsf{a}}$$

$$\frac{\Gamma((\Delta\,;x{:}\,\psi)\,,\Sigma) \vdash M : \tau}{\Gamma(\Delta\,,(x{:}\,\psi\,;\Sigma)) \vdash \mathit{hoist}\,M \stackrel{\text{def}}{=} \mathit{let}\ (\Delta\,;x{:}\,\psi)\,,\Sigma = \mathit{hoist}\,\underline{\Delta\,,(x{:}\,\psi\,;\Sigma)}\ \mathit{in}\ M : \tau}$$

arbitrary bunches, yielding admissibility of the second and third rules.

In examples we write $M\_N\,\$\,K$ for $(M\,K)\_N$ and $M\,N\,\$\,K$ for $(M\,K)\,N$ in order to write programs from beginning to end. In particular, note that $\$$ binds tighter than application. For convenience, Table 2 contains derived typing rules for the constants (where we introduce syntactic sugar in some conclusions).

Before proceeding, we illustrate the system with a few introductory examples:

Double-dispose

To illustrate that the explicit deallocation is safe, we begin with the most basic ill-typed term, an attempt to double-dispose:

$$\dfrac{\dfrac{\dfrac{}{x{:}\,\mathsf{true}\,\mathsf{ref} \vdash x{:}\,\mathsf{true}\,\mathsf{ref}} \qquad \dfrac{\dfrac{?}{\mathsf{emp} \vdash x : \mathsf{true}\,\mathsf{ref}} \qquad \dfrac{}{k{:}\,\mathsf{a} \vdash k : \mathsf{a}}}{k{:}\,\mathsf{a} \vdash \mathit{free}\_x\,\$\,k : \mathsf{a}}}{k{:}\,\mathsf{a}\,,x{:}\,\mathsf{true}\,\mathsf{ref} \vdash \mathit{free}\_x\,\$\,\mathit{free}\_x\,\$\,k : \mathsf{a}}}{k{:}\,\mathsf{a} \vdash \mathit{new}\,\lambda_* x.\,\mathit{free}\_x\,\$\,\mathit{free}\_x\,\$\,k : \mathsf{a}}$$

We indicate that this is underivable with a '?' premiss.

Strong Update and Lookup

Defining a multiplicative form of lookup, $!!_\psi : (\psi \wedge \psi \,\mathsf{ref} \multimap \mathsf{a}) \to (\psi \,\mathsf{ref} \multimap \mathsf{a})$, and strong update, $:\equiv_{\psi,\phi} : (\phi \,\mathsf{ref} \multimap \mathsf{a}) \to (\psi \,\mathsf{ref} \wedge \phi \multimap \mathsf{a})$, (note the type change) in terms of *swap* and *hoist* provides an example of strong update: when called on integer references, two type-changing assignments are made to the argument.

$$
\begin{array}{l}
!!_\psi \;:\; (\psi \wedge \psi \,\mathsf{ref} \multimap \mathsf{a}) \to (\psi \,\mathsf{ref} \multimap \mathsf{a}) \\
\quad \stackrel{\text{def}}{=} \lambda k.\, \lambda_* x. \\
\qquad \mathit{swap}_\llcorner(x\,;\mathsf{true})\,\$\,\lambda_*(v\,;x).\,\mathit{hoist}\,(\\
\qquad \mathit{swap}_\llcorner(x\,;v)\,\$\,\lambda_*(u\,;x).\,\mathit{hoist} \\
\qquad k_\llcorner(v\,;x))
\end{array}
\qquad\qquad
\begin{array}{l}
:\equiv_{\psi,\phi} \;:\; (\phi \,\mathsf{ref} \multimap \mathsf{a}) \to (\psi \,\mathsf{ref} \wedge \phi \multimap \mathsf{a}) \\
\quad \stackrel{\text{def}}{=} \lambda k.\, \lambda_* x v. \\
\qquad \mathit{swap}_\llcorner x v\,\$\,\lambda_*(i\,;x). \\
\qquad k_\llcorner x
\end{array}
$$

We will use these operations in examples, and Table 2 includes derived typing rules for them. The lookup operation accepts a continuation, k, and reference, x, separate from k and returns the contents of x and x itself to k. Strong update is essentially just the *swap* operation except that $:\equiv$ does not return the overwritten value.

The interesting part of the derivation of !!

$$
\dfrac{\dfrac{(k\,;v\colon\psi)\,,\,(v'\colon\psi\,;x\colon\mathsf{true}\,\mathsf{ref}) \vdash \mathit{swap}_\llcorner(x\,;v')\,\$\,\lambda_*(u\,;x).\,\mathit{hoist}\,k_\llcorner(v\,;x) : \mathsf{a}}{k\,,\,(v\colon\psi\,;v'\colon\psi\,;x\colon\mathsf{true}\,\mathsf{ref}) \vdash \mathit{hoist}\,(\mathit{swap}_\llcorner(x\,;v')\,\$\,\lambda_*(u\,;x).\,\mathit{hoist}\,k_\llcorner(v\,;x)) : \mathsf{a}}}{k\,,\,(v\colon\psi\,;x\colon\mathsf{true}\,\mathsf{ref}) \vdash \mathit{hoist}\,(\mathit{swap}_\llcorner(x\,;v')\,\$\,\lambda_*(u\,;x).\,\mathit{hoist}\,k_\llcorner(v\,;x))[v/v'] : \mathsf{a}}
$$

illustrates the combination of Contraction (copying v to v') and use of *hoist* to rearrange the bunch which allows v to be used both in the argument and continuation of the first call to *swap*.

3 Encapsulation

We now present a series of examples pertaining to update and encapsulation.

Keeping Internal Resource Usage Internal

As a warm-up to show how the internal resource usage of a command does not flood the types of the rest of a program, consider composing *new* and *free*:

$$
\dfrac{\dfrac{\dfrac{}{x\colon\mathsf{true}\,\mathsf{ref} \vdash x : \mathsf{true}\,\mathsf{ref}} \qquad \dfrac{}{k\colon\mathsf{a} \vdash k : \mathsf{a}}}{k\colon\mathsf{a}\,,\,x\colon\mathsf{true}\,\mathsf{ref} \vdash \mathit{free}_\llcorner x\,\$\,k : \mathsf{a}}}{\mathsf{true} \vdash \lambda k.\,\mathit{new}\lambda_* x.\,\mathit{free}_\llcorner x\,\$\,k : \mathsf{a} \to \mathsf{a}}
$$

Since we have a command of type $\mathsf{a} \to \mathsf{a}$ in the additive empty context, the internal resource usage has no manifestation in the type. The denotation of

this command is the CPS identity function, meaning that subsequent allocations are free to choose the same location which was allocated and deallocated here.

State-Passing without Encapsulation

We can model commands that do change the state by using reference-expecting continuations, where the use of $-\!\!*$ in the continuation type prohibits aliasing. Here is an example that simply increments an integer reference:

$$\dfrac{\dfrac{}{x\colon \mathsf{int\ ref} \vdash x : \mathsf{int\ ref}} \qquad \dfrac{\dfrac{\vdots}{v\,;x \vdash x : \mathsf{int\ ref} \quad v\,;x \vdash v{+}1 : \mathsf{int}} \qquad \dfrac{}{k\colon \mathsf{int\ ref} -\!\!* \mathsf{a} \vdash k : \mathsf{int\ ref} -\!\!* \mathsf{a}}}{k\colon \mathsf{int\ ref} -\!\!* \mathsf{a}\,,\,(v\colon \mathsf{int}\,;x\colon \mathsf{int\ ref}) \vdash x{:}\equiv v{+}1\,\$\,k : \mathsf{a}}}{k\colon \mathsf{int\ ref} -\!\!* \mathsf{a} \vdash \lambda_*(v\,;x).\, x{:}\equiv v{+}1\,\$\,k : \mathsf{int} \wedge \mathsf{int\ ref} -\!\!* \mathsf{a}}$$

$$\dfrac{k\colon \mathsf{int\ ref} -\!\!* \mathsf{a}\,,\,x\colon \mathsf{int\ ref} \vdash !!\_x\,\$\,\lambda_*(v\,;x).\, x{:}\equiv v{+}1\,\$\,k : \mathsf{a}}{x\colon \mathsf{int\ ref} \vdash \lambda_* k.\, !!\_x\,\$\,\lambda_*(v\,;x).\, x{:}\equiv v{+}1\,\$\,k : (\mathsf{int\ ref} -\!\!* \mathsf{a}) -\!\!* \mathsf{a}}$$

This example lives within the common fragment of linear logic and BI, and is of the sort which systems built on linear logic, such as [20,9], handle well.

This example shows update at work, but does not provide any encapsulation. Instead of the type used in this example, we would like to have an increment command which did not have to pass the reference to its continuation, so that it satisfies the judgment $x\colon \mathsf{int\ ref} \vdash \mathit{inc} : \mathsf{a} \to \mathsf{a}$. We use an additive arrow here, intuitively, since the computation represented by the continuation should also have access to the reference, for instance, to *inc* it again. We are relying on the sharing interpretation of BI's additives [12]. That is, the additive arrow in $\mathsf{a} \to \mathsf{a}$ must mean that the function (command) and its argument (continuation) share the same resources, since the intent is for both to access the $\mathsf{int\ ref}$. A model that underpins this intuition, for the current language, will be given in Section 5.

Object Generation

Using the additive weak update and lookup operations, the increment operation on an integer reference sought above can be given by:

$$\dfrac{\dfrac{}{x\colon \mathsf{int\ ref} \vdash x : \mathsf{int\ ref}} \qquad \dfrac{\dfrac{\vdots}{(\mathsf{emp}\,,\,x)\,;k\,;v \vdash v{+}1 : \mathsf{int}} \quad \dfrac{\vdots}{(\mathsf{emp}\,,\,x)\,;k\,;v \vdash k : \mathsf{a}}}{(\mathsf{emp}\,,\,x\colon \mathsf{int\ ref})\,;k\colon \mathsf{a}\,;v\colon \mathsf{int} \vdash x{:}= v{+}1\,\$\,k : \mathsf{a}}}{x\colon \mathsf{int\ ref}\,;k\colon \mathsf{a}\,;v\colon \mathsf{int} \vdash x{:}= v{+}1\,\$\,k : \mathsf{a}}$$

$$\dfrac{x\colon \mathsf{int\ ref}\,;k\colon \mathsf{a} \vdash !\,x\,\$\,\lambda v.\, x{:}= v{+}1\,\$\,k : \mathsf{a} \to \mathsf{a}}{x\colon \mathsf{int\ ref} \vdash \mathit{inc} \stackrel{\mathrm{def}}{=} \lambda k.\, !\,x\,\$\,\lambda v.\, x{:}= v{+}1\,\$\,k : \mathsf{a} \to \mathsf{a}}$$

(Note that we elide some types which already appeared lower in the derivation.) The essential point here is just that we are able to have a reference

free in a term of additive function type. That way, the reference can change, without it being mentioned in the overall type of the term. However, what we must not do is change the type of the behind-the-scenes reference, for, if we are not revealing the reference type in the interface (the $\mathsf{a} \to \mathsf{a}$) like we do in state-passing then the type change will not be tracked. For this example, note how $k\colon \mathsf{a}\ ;\ v\colon \mathsf{int}$ is preserved across the call to $:=$, even though it is not separate from $x\colon \mathsf{int\ ref}$, on which $:=$ operates. The rule for $:\equiv$ does not admit a similar preservation of additively combined parts of the context (as it must not).

An operation to return the value of a reference $x\colon \mathsf{int\ ref} \vdash \mathit{get}\colon (\mathsf{int} \to \mathsf{a}) \to \mathsf{a}$ is straightforward. Finally, we also define a destructor:

$$\dfrac{\dfrac{}{x\colon \mathsf{int\ ref} \vdash x : \mathsf{int\ ref}} \qquad \dfrac{}{k\colon \mathsf{a} \vdash k : \mathsf{a}}}{\dfrac{x\colon \mathsf{int\ ref}\ ,\ k\colon \mathsf{a} \vdash \mathit{free}\_x\,\$\,k : \mathsf{a}}{x\colon \mathsf{int\ ref} \vdash \mathit{free} \stackrel{\text{def}}{=} \lambda_* k.\, \mathit{free}\_x\,\$\,k : \mathsf{a} \mathrel{-\!\!*} \mathsf{a}}}$$

Just like the *inc* and *get* code, this has an integer reference inside it. But its type uses $\mathrel{-\!\!*}$ rather than $\to$; in bunched typing neither of these is more special than the other when it comes to the kinds of identifiers that can be free. However, because we have a multiplicative command here, the continuation operates on a smaller heap than the command itself. Here the difference is $x\colon \mathsf{int\ ref}$, and so the continuation will not have access to x. With a multiplicative command we have the opportunity to dispose of a free reference, while in an additive command we cannot.

The essential difference between additive and multiplicative commands is that the former are sequentially composable; e.g.

$$\mathit{twice} : (\mathsf{a} \to \mathsf{a}) \to (\mathsf{a} \to \mathsf{a}) = \lambda c.\, \lambda k.\, c\,(c\,k)$$

while the latter are not, since the following does not typecheck:

$$\mathit{twice}_* : (\mathsf{a} \mathrel{-\!\!*} \mathsf{a}) \to (\mathsf{a} \mathrel{-\!\!*} \mathsf{a}) = \lambda c.\, \lambda_* k.\, c\_(c\_k)$$

Multiplicative commands are useful for deallocating shared resources, but they cannot be postcomposed with other commands that use that same resource.

Using these three operations, we can define an object constructor, where $\mu = ((\mathsf{int} \to \mathsf{a}) \to \mathsf{a}) \wedge (\mathsf{a} \to \mathsf{a}) \wedge (\mathsf{a} \mathrel{-\!\!*} \mathsf{a})$ is the type of the methods, by:

$$\dfrac{\dfrac{\dfrac{}{x\colon \mathsf{true\ ref} \vdash x : \mathsf{true\ ref}} \qquad \dfrac{\vdots}{x \vdash 0 : \mathsf{int}} \qquad \dfrac{\dfrac{\dfrac{}{k\colon \mu \mathrel{-\!\!*} \mathsf{a} \vdash k : \mu \mathrel{-\!\!*} \mathsf{a}} \qquad \dfrac{\vdots}{x\colon \mathsf{int\ ref} \vdash (\mathit{get}\ ;\ \mathit{inc}\ ;\ \mathit{free}) : \mu}}{k\ ,\ x\colon \mathsf{int\ ref} \vdash k\_(\mathit{get}\ ;\ \mathit{inc}\ ;\ \mathit{free}) : \mathsf{a}}}{k \vdash \lambda_* x.\, k\_(\mathit{get}\ ;\ \mathit{inc}\ ;\ \mathit{free}) : \mathsf{int\ ref} \mathrel{-\!\!*} \mathsf{a}}}{k\ ,\ x\colon \mathsf{true\ ref} \vdash x :\equiv 0\,\$\,\lambda_* x.\, k\_(\mathit{get}\ ;\ \mathit{inc}\ ;\ \mathit{free}) : \mathsf{a}}}{\mathsf{true} \vdash \mathit{new\_counter} \stackrel{\text{def}}{=} \lambda k.\, \mathit{new}\ \lambda_* x.\, x :\equiv 0\,\$\,\lambda_* x.\, k\_(\mathit{get}\ ;\ \mathit{inc}\ ;\ \mathit{free}) : (\mu \mathrel{-\!\!*} \mathsf{a}) \to \mathsf{a}}$$

We remind the reader that the semicolon is used for tuple-formation for additive products (enabling the game mentioned at the end of Section 2), and

not for sequential composition. Note the similarity between the types of *new_counter* and *new*: they have the same form, but *new* passes a reference cell to its client continuation while *new_counter* passes a tuple of commands.

A key point here is the type of the methods, which is an additive conjunction. This means that all the methods can share the integer reference between them. However, despite this degree of sharing, notice that x and k are "," separated in the typing context, meaning that the continuation k cannot access the integer reference x. And this is essential, because if we were to dispose x using *free*, and then k accessed x, unsoundness would result.

For comparison, discussions with Ahmed indicate that the nonrecursive, monomorphic fragments of systems based on linear logic such as [2] admit typing tuples of functions sharing some resources like that above, but without the *free* method. Also, the pair of *inc* and *free* could be constructed, but its type would not allow several calls to *inc* followed by one to *free*.

4 Framing

In program logic, frame axioms describe invariants for portions of state not altered by a computation [11]. A similar issue arises in resource typing, and this section investigates degrees of framing supported by the language.

Framing Multiplicative Commands (for free)

For multiplicative commands, such as $\mathit{free}\,K$ of type $\mathsf{true\ ref} \multimap \mathsf{a}$, framing is simple. Consider:

$$\dfrac{\dfrac{}{x\colon \mathsf{true\ ref} \vdash x : \mathsf{true\ ref}} \qquad \dfrac{\dfrac{}{y\colon \mathsf{true\ ref} \vdash y : \mathsf{true\ ref}} \qquad \Gamma \vdash K : \mathsf{a}}{\Gamma\,,\, y\colon \mathsf{true\ ref} \vdash \mathit{free}\,K\_y : \mathsf{a}}}{\Gamma\,,\, x\colon \mathsf{true\ ref}\,,\, y\colon \mathsf{true\ ref} \vdash \mathit{free}\,(\mathit{free}\,K\_y)\_x : \mathsf{a}}$$
$$\Gamma \vdash \mathit{new}\,\lambda_* x.\,\mathit{new}\,\lambda_* y.\,\mathit{free}\,(\mathit{free}\,K\_y)\_x : \mathsf{a}$$

For the call $\mathit{free}\,(\mathit{free}\,K\_y)\_x$ the context consists of $\Gamma\,,\, x\colon \mathsf{true\ ref}\,,\, y\colon \mathsf{true\ ref}$, while the command operates only on $x\colon \mathsf{true\ ref}$, which makes the rest of the context, Γ,$y\colon \mathsf{true\ ref}$, a frame axiom. As demonstrated by the derivation above, the form of the multiplicative application rule allows such frame axioms to just be sent to the command's branch of the derivation, giving us framing for free.

Frame-Threading

For additive commands things are not so easy: consider calling the methods of a counter with $\lambda k.\,\mathit{new\_counter}\,\lambda_*(g\,;i\,;f).\,i\,(i\,(f\_k))$. To typecheck this, we need to derive:

$$\dfrac{\dfrac{?}{k\colon \mathsf{a}\,,\,\Gamma \vdash i : \mathsf{a} \to \mathsf{a}} \qquad \cdots}{k\colon \mathsf{a}\,,\,\Gamma \vdash i\,(i\,(f\_k)) : \mathsf{a}}$$

where $\Gamma = g\colon (\mathsf{int} \to \mathsf{a}) \to \mathsf{a}\ ;\ i\colon \mathsf{a} \to \mathsf{a}\ ;\ f\colon \mathsf{a} \mathrel{-\!\!*} \mathsf{a}$. But the given premiss is untypable since "," does not admit Weakening. In actuality, the calls to $i\,i\,f$ will not interfere with k, so what we need is a way to keep k off to one side while calling the methods, and then pass it to f at the end.

One way to accomplish this would be to give *inc* type $(\mathsf{a} \mathrel{-\!\!*} \mathsf{a}) \to (\mathsf{a} \mathrel{-\!\!*} \mathsf{a})$, which could thread a continuation of type a (a frame axiom) through the computation. This would be complex and roundabout. Furthermore, when explicitly threading frames, typing examples where a function is called with frames of different types at different call-sites would potentially require giving each function as many types as there are call-sites. This would be very unfortunate.

Alias types use frame-threading, but lighten the burden using store polymorphism [26]. Thielecke [22] takes this frame-threading approach, but he avoids the complication by using polymorphic answer types and eliding term annotations for typing rules involving $\mathrel{-\!\!*}$.

The *frame* constant addresses these issues by allowing invariants to be added to (additive) commands directly, remaining in a monomorphic language, following separation logic's Frame rule.

Framing Additive Commands (using *frame*)

Using *frame*, the continuation can be held aside while calling the methods:

$$\dfrac{\dfrac{}{k\colon \mathsf{a} \vdash k : \mathsf{a}} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash \mathit{frame}_{\mathsf{a}}\, i : (\mathsf{a} \mathrel{-\!\!*} \mathsf{a}) \to (\mathsf{a} \mathrel{-\!\!*} \mathsf{a})} \quad \dfrac{\vdots}{\Gamma \vdash f : \mathsf{a} \mathrel{-\!\!*} \mathsf{a}}}{\Gamma \vdash \mathit{frame}_{\mathsf{a}}\, i\, f : \mathsf{a} \mathrel{-\!\!*} \mathsf{a}}}{\dfrac{k\colon \mathsf{a} \vdash k : \mathsf{a} \qquad \Gamma \vdash \mathit{frame}_{\mathsf{a}}\, i\, (\mathit{frame}_{\mathsf{a}}\, i\, f) : \mathsf{a} \mathrel{-\!\!*} \mathsf{a}}{k\colon \mathsf{a}\,, \Gamma \vdash \mathit{frame}_{\mathsf{a}}\, i\, (\mathit{frame}_{\mathsf{a}}\, i\, f) \lrcorner k : \mathsf{a}}}{\mathsf{true} \vdash \lambda k.\, \mathit{new\_counter}\, \lambda_*(g\ ;\ i\ ;\ f).\, \mathit{frame}_{\mathsf{a}}\, i\, (\mathit{frame}_{\mathsf{a}}\, i\, f) \lrcorner k : \mathsf{a} \to \mathsf{a}}$$

This example illustrates how the role of *frame* is to enable a version of sequential composition of (additive) commands in which the separate part of the context which a command does not need to run (in this case $k\colon \mathsf{a}$) is passed unchanged through to the subsequent command.

This derivation also demonstrates the utility of a tree-structured context, that is, bunches. The bunched structure of the context $k\colon \mathsf{a}\ ,\ (g\colon (\mathsf{int} \to \mathsf{a}) \to \mathsf{a}\ ;\ i\colon \mathsf{a} \to \mathsf{a}\ ;\ f\colon \mathsf{a} \mathrel{-\!\!*} \mathsf{a})$ expresses that the methods can all share access to a common reference, while the continuation is prohibited from sharing it.

The semantic difference between these approaches to framing is that with frame-threading, what is basically happening is that by giving commands store polymorphic or parameterized types, the fact that the command will not interfere with the parameter is being built-in manually. The *frame* constant, on the other hand, reveals that the meanings of commands are already guaranteed not to interfere with any possible separate parameter.

Recap

To summarize what the preceding examples have shown: utilizing only the multiplicative types and operations together with state-passing does not allow encapsulation of state, but the additive types and operations do allow expression of encapsulation. However, while framing was easy with only multiplicatives, the additives are more difficult. Frame-threading is one approach, but it suffers from some unnecessary complexities. Another approach is to draw inspiration from separation logic's Frame rule and capitalize on the concrete model to introduce the *frame* constant. (Here we have included only first-order framing, higher-order frame rules [14,3] could also be investigated.)

5 Model

Following [12], we describe a spatial possible worlds model. A world is a partial function which assigns a type to each allocated location. When such a function is undefined we regard the location as not being allocated. The set of values and partial commutative monoid of possible worlds are defined by

$$\mathrm{Val} \stackrel{\text{def}}{=} \{\mathsf{true}\} \cup \mathbb{Z} \qquad \text{and} \qquad \mathrm{Wld} \stackrel{\text{def}}{=} (\mathrm{Loc} \stackrel{\text{fin}}{\rightharpoonup} \psi, \varnothing, -\!*\!-)$$

where $w*w'$ takes the union of partial functions with disjoint domain, being undefined when the domains of w and w' overlap. We write $w \perp w'$ to mean that $w*w'$ is defined. Here, $\mathrm{Loc} \stackrel{\text{fin}}{\rightharpoonup} \psi$ denotes the set of all finite functions from locations to syntactic types, $\varnothing \in \mathrm{Loc} \stackrel{\text{fin}}{\rightharpoonup} \psi$, and $-\!*\!-$ is a partial binary operation on $\mathrm{Loc} \stackrel{\text{fin}}{\rightharpoonup} \psi$.

Storable types ψ denote sets of values, $\mathcal{V}(\psi)$:

$$\mathcal{V}(\mathsf{true}) \stackrel{\text{def}}{=} \{\mathsf{true}\} \qquad\qquad \mathcal{V}(\mathsf{int}) \stackrel{\text{def}}{=} \mathbb{Z}$$

For each world $w \in \mathrm{Wld}$,

$$H(w) \stackrel{\text{def}}{=} \prod_{\ell \in dom(w)} \mathcal{V}(w(\ell))$$

is the set of heaps compatible with w.

Types denote functions that map individual worlds to sets of values, so that $[\![\tau]\!]\, w$ is a set for each world $w \in \mathrm{Loc} \stackrel{\text{fin}}{\rightharpoonup} \psi$: a type denotes a world-indexed tuple of sets.

$$[\![\mathsf{a}]\!]\, w \stackrel{\text{def}}{=} H(w) \Rightarrow 2$$
$$[\![\psi]\!]\, w \stackrel{\text{def}}{=} \mathcal{V}(\psi)$$
$$[\![\mathsf{emp}]\!]\, w \stackrel{\text{def}}{=} \begin{cases} \{\mathsf{emp}\} & \text{if } w = \varnothing \\ \emptyset & \text{otherwise} \end{cases}$$
$$[\![\psi\ \mathsf{ref}]\!]\, w \stackrel{\text{def}}{=} \begin{cases} \{\ell\} & \text{if } w = \{[\ell : \psi]\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\sigma \wedge \tau]\!]\, w \stackrel{\text{def}}{=} ([\![\sigma]\!]\, w) \times ([\![\tau]\!]\, w)$$
$$[\![\sigma * \tau]\!]\, w \stackrel{\text{def}}{=} \sum_{(w',w'')|w=w'*w''} ([\![\sigma]\!]\, w') \times ([\![\tau]\!]\, w'')$$
$$[\![\sigma \to \tau]\!]\, w \stackrel{\text{def}}{=} ([\![\sigma]\!]\, w) \Rightarrow ([\![\tau]\!]\, w)$$
$$[\![\sigma \mathrel{-\!\!*} \tau]\!]\, w \stackrel{\text{def}}{=} \prod_{w' \perp w} ([\![\sigma]\!]\, w') \Rightarrow ([\![\tau]\!]\, w*w')$$

The set theoretic function space is denoted $\Rightarrow$, and 2 is the two-point set.

The meaning of the answer type $[\![\mathsf{a}]\!]\,w$ in a particular world consists of functions which accept heaps for that world and return an answer. They are the command continuations. The meanings of storable types are world-independent, which is to say that they are constant tuples. The interpretation of the reference type is noteworthy: it is nonempty only when the world w is a singleton. This is related to the "exact" points-to relation in separation logic, which denotes a single cell and nothing else.

The other connectives from bunched type theory are given their standard interpretations. The additive product $\wedge$ allows pairs of values that live at the same world, while the multiplicative $*$ takes pairs from separate worlds. The effect of this can be seen in $[\![\mathsf{int\ ref} \wedge \mathsf{int\ ref}]\!]\,w$ and $[\![\mathsf{int\ ref} * \mathsf{int\ ref}]\!]\,w$. In the former the world w can only be a singleton if the denotation is to be nonempty, and so the two elements of the tuple must be one and the same. In the latter w must have size precisely two, and different references are sent to each side of $*$. The semantics of the additive function type $\rightarrow$ requires that it takes arguments at a given world to results at that same world, while the multiplicative type asks for arguments from a separate, or fresh, world.

A constant c of type τ is interpreted by specifying, for each w, an element

$$[\![c]\!]\,w \in [\![\tau]\!]\,w$$

That is, a constant is a type-correct, world-indexed family of elements. In the following we recall the type of each constant before giving its denotation.

$$\begin{aligned}
\mathit{swap}_{\psi,\phi} &: (\psi \wedge \phi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}) \rightarrow (\psi\,\mathsf{ref} \wedge \phi \mathrel{-\!\!*} \mathsf{a}) \\
[\![\mathit{swap}_{\psi,\phi}]\!]\,w &\stackrel{\text{def}}{=} \lambda k^{\prod_{w'_l \perp w} [\![\psi \wedge \phi\,\mathsf{ref}]\!] w'_l \Rightarrow [\![\mathsf{a}]\!] w * w'_l}.\, \Pi w_l{}^{\perp w}.\, \lambda(l^{[\![\psi\,\mathsf{ref}]\!] w_l}, v^{[\![\phi]\!]}).\, \lambda h^{w * w_l}. \\
&\qquad k\,[l\colon \phi]\,(h(l), l)\,[h \mid l\colon v]
\end{aligned}$$

$$\begin{aligned}
\mathit{new} &: (\mathsf{true\ ref} \mathrel{-\!\!*} \mathsf{a}) \rightarrow \mathsf{a} \\
[\![\mathit{new}]\!]\,w &\stackrel{\text{def}}{=} \lambda k^{\prod_{w_l \perp w} [\![\mathsf{true\ ref}]\!] w_l \Rightarrow [\![\mathsf{a}]\!] w * w_l}.\, \lambda h^w.\, \textstyle\bigwedge_{l \notin dom(w)} k\,[l\colon \mathsf{true}]\,l\,[h \mid l\colon \mathsf{true}]
\end{aligned}$$

$$\begin{aligned}
\mathit{free}_{\psi} &: \mathsf{a} \rightarrow (\psi\,\mathsf{ref} \mathrel{-\!\!*} \mathsf{a}) \\
[\![\mathit{free}_{\psi}]\!]\,w &\stackrel{\text{def}}{=} \lambda k^{[\![\mathsf{a}]\!] w}.\, \Pi w_l{}^{\perp w}.\, \lambda l^{[\![\psi\,\mathsf{ref}]\!] w_l}.\, \lambda h^{w * w_l}.\, k\,h|_w
\end{aligned}$$

$$\begin{aligned}
\mathit{hoist}_{\sigma,\psi,\tau} &: (\sigma * (\psi \wedge \tau)) \rightarrow ((\sigma \wedge \psi) * \tau) \\
[\![\mathit{hoist}_{\sigma,\psi,\tau}]\!]\,w &\stackrel{\text{def}}{=} \lambda x.\, x
\end{aligned}$$

$$\begin{aligned}
!_{\psi} &: (\psi \rightarrow \mathsf{a}) \rightarrow (\psi\,\mathsf{ref} * \mathsf{true} \rightarrow \mathsf{a}) \\
[\![!_{\psi}]\!]\,w &\stackrel{\text{def}}{=} \lambda k^{[\![\psi \rightarrow \mathsf{a}]\!] w}.\, \lambda x^{[\![\psi\,\mathsf{ref} * \mathsf{true}]\!] w}.\, \lambda h^w. \\
&\qquad \mathit{case}\ x\ \mathit{of}\ \iota_{(w_l, w_t)}\,(l^{[\![\psi\,\mathsf{ref}]\!] w_l}, t^{[\![\mathsf{true}]\!] w_t}) \Mapsto k\,(h\,l)\,h
\end{aligned}$$

$$\begin{aligned}
:=_{\psi} &: \mathsf{a} \rightarrow ((\psi\,\mathsf{ref} * \mathsf{true}) \wedge \psi \rightarrow \mathsf{a}) \\
[\![:=_{\psi}]\!]\,w &\stackrel{\text{def}}{=} \lambda k^{[\![\mathsf{a}]\!] w}.\, \lambda(x^{[\![\psi\,\mathsf{ref} * \mathsf{true}]\!] w}, v^{[\![\psi]\!]}).\, \lambda h^w. \\
&\qquad \mathit{case}\ x\ \mathit{of}\ \iota_{(w_l, w_t)}\,(l^{[\![\psi\,\mathsf{ref}]\!] w_l}, t^{[\![\mathsf{true}]\!]}) \Mapsto k\,[h \mid l\colon v]
\end{aligned}$$

$$\mathit{frame}_{\varphi} : (\mathsf{a} \rightarrow \mathsf{a}) \rightarrow ((\varphi \mathrel{-\!\!*} \mathsf{a}) \rightarrow (\varphi \mathrel{-\!\!*} \mathsf{a}))$$

$$[\![\mathit{frame}_\varphi]\!]\, w \stackrel{\text{def}}{=} \lambda c^{[\![\mathsf{a}\to\mathsf{a}]\!]w}.\, \lambda k^{\Pi_{w_f \perp w}[\![\varphi]\!]w_f \Rightarrow [\![\mathsf{a}]\!]w * w_f}.\, \Pi {w_f}^{\perp w}.\, \lambda f^{[\![\varphi]\!]w_f}.\, \lambda {h'}^{w * w_f}.\\ c\,(\lambda h^w.\, k\, w_f\, f\, (h'|_{w_f} * h))\, h'|_w$$

Here, $h|_w$ denotes the subheap of h gotten from restricting the domain of h to the domain of world w.

It is worth talking through the semantics of a few examples. Let's start with *free*. It expects a continuation k in world w and a reference from a completely separate world from k's. Because of the singleton semantics of references, this separate world can only be a singleton, consisting of a single location l. We then take in a heap h that gives values to locations in w as well as l, as we are required to by the overall command continuation, and we have to produce an element of 2. We do this by restricting h to w, thus effectively deallocating l, and supplying the trimmed heap to our input continuation k.

The trickiest example is *new*. We first take in a reference-expecting continuation k, where that continuation expects a reference argument that is separate from the current world w. So we are forced, by the semantics of types, to find a fresh location if we are to use k. However, there is a problem: there are many such fresh locations. So we choose any of them, but require that the overall result be happy with whatever choice we make by taking the conjunction of the results, where in the definition $[\![\mathsf{a}]\!]\, w = H(w) \Rightarrow 2$ we are viewing the set 2 as the booleans consisting of true and false. This is the one place where we use knowledge of what 2 is.

To understand the use of conjunction here, it helps to recall one way of doing a continuation semantics of the nondeterministic choice (c_1 *choose* c_2) of two commands, i.e., continuation transformers of type $\mathsf{a} \to \mathsf{a}$. We could do this, if we know that 2 is used for final answers, with $\lambda k.\, \lambda s.\, ([\![c_1]\!]\, w\, k\, s \wedge [\![c_2]\!]\, w\, k\, s)$. Operationally, we would not expect both choices to be taken in a particular run, but $\wedge$ allows us to record, denotationally, that for the command to be happy it has to be content with whichever nondeterministic choice is made. (This use of maps from states to truth values in the answer type makes continuation transformers similar to predicate transformers.) The semantics of *new* uses this very idea, adjusted to account for a reference-expecting continuation rather than a plain command continuation as an argument, and considering a larger conjunction corresponding to a larger nondeterministic choice. Again, this does not require all choices to be taken operationally, but captures that k must be content with whatever choice is made.

We will not attempt to give a formal connection here to predicate transformers or operational semantics, but offer these remarks just as an intuitive aid that we have found helpful ourselves.

The information we have given is enough to specify the model for the whole language. What we have described is the semantics of types and constants for a semantics in the product category $\mathrm{Set}^{\mathrm{Wld}}$, where the doubly closed structure $(\wedge, \to, *, \mathrel{-\!\!*})$ is as in the basic separation model of [12]. To fill out the semantics we just have to follow the categorical scheme as laid down in [15,12]. We will

not repeat the details here, but will simply state

Theorem 5.1 (Soundness) *Every derivation of a typing judgment* $\Gamma \vdash M : \tau$ *determines a family of functions of semantic type*

$$[\![\overline{\Gamma}]\!]\, w \longrightarrow [\![\tau]\!]\, w$$

indexed by worlds w*.*

(Note: In the general functor-category models of [12] this family would come with naturality constraints. For this particular model, though, those constraints are trivial because Wld is a discrete category.)

This is the kind of result one expects of an explicitly-typed semantics (a "Church model"). Rather than introduce a kind of error, and then prove that it is avoided by typing, we build our meanings without mentioning error at all, and observe that our language stays within its language of discourse.

This possible world form of semantics can take some getting used to, particularly the valuations for terms. The semantics of types, though, is comparatively direct and plays the most important role, in determining what can and cannot be done in the language.

In addition to being a useful design aid, such models provide help in understanding essential differences between languages. For example, the remarks we made about encapsulation flowed from the observation that several linear languages can be modeled in possible world models like that here, where the nonlinear function type is interpreted essentially as $(A \mathrel{-\!\!*} B) \wedge \mathsf{emp}$. It is then immediate that such functions cannot be used to access shared, hidden state, and it is equally obvious that such sharing can be done using the cartesian closed structure (additive) that exists in the model. Generally speaking, the method of setting down a model first – to "put your domains on the table" [19] – is a powerful complement to the nowadays more prevalent operational methods, and it appears that much more can be got out of it in applications of substructural type systems.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. Technical Report TR-24-04, Harvard University, 2004.

[2] A. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *ICFP*, 2005.

[3] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *20th LICS*, pages 260–269, 2005.

[4] J. Cheney and G. Morrisett. A linearly typed assembly language. Technical Report 2003-1900, Department of Computer Science, Cornell University, 2003.

[5] M. Collinson and D. Pym. A bunched approach to the semantics of regions and locations. In *SPACE*, 2006.

[6] M. Collinson, D. J. Pym, and E. P. Robinson. On bunched polymorphism. In *CSL*, pages 36–50, 2005.

[7] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[8] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 14–26, 2001.

[9] G. Morrisett, A. J. Ahmed, and M. Fluet. L^3: A linear language with locations. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *LNCS*, pages 293–307, 2005.

[10] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM TOPLAS*, 21(3):527–568, 1999.

[11] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.

[12] P. W. O'Hearn. On bunched typing. *J. Functional Programming*, 13(4), 2003.

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

[14] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, 2004.

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

[16] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pages 55–74, 2002.

[17] J. C. Reynolds. Syntactic control of interference. In *5th POPL*, 1978.

[18] 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.

[19] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, 1971.

[20] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In G. Smolka, editor, *ESOP*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000.

[21] K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Manuscript, 2001.

[22] H. Thielecke. Frame rules from answer types for code pointers. In *POPL*, 2006.

[23] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, North Holland, 1990.

[24] P. Wadler. Is there a use for linear logic? In *PEPM*, pages 255–273, 1991.

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

[26] D. Walker and J. Morrisett. Alias types for recursive data structures. In *3rd Types in Compilation*, pages 177–206, 2001. LNCS 2071.