Recovering Purity with Comonads and Capabilities

ANONYMOUS AUTHOR(S)

In this paper, we take a pervasively effectful (in the style of ML) typed lambda calculus, and show how to *extend* it to permit capturing pure expressions with types. Our key observation is that, just as the pure simply-typed lambda calculus can be extended to support effects with a monadic type discipline, an impure typed lambda calculus can be extended to support purity with a *comonadic* type discipline.

We establish the correctness of our type system via a simple denotational model, which we call the *capability space* model. Our model formalizes the intuition common to systems programmers that the ability to perform effects should be controlled via access to a permission or capability, and that a program is *capability-safe* if it performs no effects that it does not have a runtime capability for. We then identify the axiomatic categorical structure that the capability space model validates, and use these axioms to give a categorical semantics for our comonadic type system. We then give an equational theory (substitution and the call-by-value β and η laws) for the imperative lambda calculus, and show its soundness relative to this semantics.

Finally, we give a translation of the pure simply-typed lambda calculus into our comonadic imperative calculus, and show that any two terms which are $\beta\eta$-equal in the STLC are equal in the equational theory of the comonadic calculus, establishing that pure programs can be mapped in an equation-preserving way into our imperative calculus.

1 INTRODUCTION

Consider the two following definitions of the familiar `map` functional, which applies a function to each element of a list.

```
map1 : ∀ a b. (a → b) → List a → List b
map1 f []        = []
map1 f (x :: xs) = let zs = map1 f xs in
                   let  z = f x in
                   z :: zs

map2 : ∀ a b. (a → b) → List a → List b
map2 f []        = []
map2 f (x :: xs) = let  z = f x in
                   let zs = map2 f xs in
                   z :: zs
```

In a purely functional language like Haskell, these two definitions are equivalent. But in an *impure* functional language like ML. the difference between these two definitions is *observable*:

```
let xs = ["left "; "to "; "right "]

let f s = stdout.print(s); s

let ys = map1 f xs  -- Prints "right to left "
let zs = map2 f xs  -- Prints "left to right "
```

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

So something as innocuous-seeming as a `print` function can radically change the equational theory of the language: no program transformation that changes the order in which sub-expressions are evaluated is in general sound. This greatly complicates reasoning about programs, as well as hindering many desirable program optimisations such as list fusion and deforestation [Wadler 1990]. Transformations that are unconditionally valid in a pure language must, in an impure language, be gated by complex whole-program analyses tracking the purity of sub-expressions.

Contributions. It is received wisdom that much as a drop of ink cannot be removed from a glass of water, once a language supports ambient effects, there is no way to regain the full equational theory of a pure programming language. In this paper, we show that this folk belief is *false*: we extend an ambiently effectful language to support purity. Entertainingly, it turns out that just as monads are a good tool to extend pure languages with effects, **comonads** are a good tool to extend impure languages with purity!

- We take a pervasively effectful lambda calculus in the style of ML and show how to *extend* it with a *comonadic* type discipline modelling the intuitions underpinning the *object-capability model* [Lauer and Needham 1979; Levy 1984; Miller 2006] developed in the systems community. The object-capability model advises that the ability to perform effects should be controlled via access to a permission or capability, and that a program is *capability-safe* precisely when it can only perform effects that it possesses a runtime capability for.
- We show that the typing rules are faithful to the object-capability model by giving our language a denotational semantics, which we call the *capability space* model. Capability spaces are a simple, direct formalisation of the ideas underpinning the object-capability model, which extends the most naive model of the lambda calculus – sets and functions – with just enough structure to model capability-safety. In our model, a type is just a set X (denoting a set o f values), together with a relation w_X saying which capabilities each value x may own. Morphisms $f : X \to Y$ are *capability-safe* if the capabilities of $f(x)$ are bounded by the capabilities of x.
  It is already known in the systems community that even effectful, untyped lambda-calculi can be made capability-safe by removing features exposing ambient authority. Our model and type system demonstrates that this observation is incomplete – having a comonad witnessing the *denial* of a capability is also very beneficial. In particular, this greatly simplifies the process of *capability taming*, making it possible to make the standard library capability-safe in an incremental fashion.
- We then identify the axiomatic categorical structure the capability space model validates, and use these axioms to give a categorical semantics for our comonadic type system. We then give an equational theory (substitution and the call-by-value β and η laws) for the imperative lambda calculus, and show its soundness relative to this semantics.
- Finally, we give a translation of the pure simply-typed lambda calculus into our comonadic imperative calculus, and show that any two terms which are $\beta\eta$-equal in the STLC are equal in the equational theory of the comonadic calculus under the translation, establishing that pure programs can be mapped in an equation-preserving way into our imperative calculus.

Detailed proofs of the lemmas and theorems, as well as additional material are given in the supplementary appendices, and we refer to them in the text.

2 PURITY FROM CAPABILITIES

The *object-capability* model is a methodology originating in the operating systems community for building secure operating systems and hardware. The idea behind this model is that systems must be able to control permissions to perform potentially dangerous or insecure operations, and that a

good way to control access is to tie the right to perform actions to values in a programming language, dubbed *capabilities*. Then, the usual variable-binding and parameter-passing mechanisms of the language can be used to grant rights to perform actions — access to a capability can be prohibited to a client by simply not passing it the capability as an argument. To quote Miller [2006]:

> Our object-capability model is essentially the untyped call-by-value lambda calculus with applicative-order local side effects and a restricted form of **eval** — the model Actors and Scheme are based on. This correspondence of objects, lambda calculus, and capabilities was noticed several times by 1973.

We use this observation to design our language – we begin with the observation that it is possession of the capability to perform effects that distinguishes impure from pure code. In the example in section 1, the operation `f` that distinguished between `map1` and `map2` contained a reference to `stdout`, and so had the intrinsic authority to print to the standard output – that is, `f` was not a capability-safe function.

The c.`print`(s) operation takes the channel c and prints the string s to it. If we did not possess the capability c, then we could not invoke the print operation upon this channel. This property is actually fundamental to the object-capability model, which says that the *only* way to access capabilities must be through capability values. Therefore, if we view channels as capabilities, we know that evaluating a piece of code *lacking* any capabilities cannot print at all.

Naturally, there are many data types in a real programming language beyond channels, but each value can access some set of capabilities (eg, a list of files can access any of the channels in the list, or a closure can access any capability it receives as an argument or possesses in its environment). So for each value, we can bound the set of possible effects it enables by the capabilities it owns.

This lets us approximate the notion of a "capability-safe program" in a simple and brutal fashion: we can judge a term to be capability-safe if it can directly access **zero** capabilities. Lacking access to any channels, it has no intrinsic ability to do I/O, and hence must be capability-safe. Furthermore, we introduce **two kinds of variables:** *safe* variables and arbitrary (or *impure*) variables. By restricting the substitution to only permit substituting capability-safe terms for safe variables, the judgement of safety will be stable under substitution. Then, by internalising the safety judgement as a type, we can pass safe values – i.e., values without access to any capabilities – as first-class values.

To understand this, let us begin with a simple call-by-value higher-order functional language extended with types for string constants, channels (or output file handles), and a single effect: outputting a string onto a channel with the expression `chan.print(s)`. There is no monadic or effect typing discipline here; the type of `print` is just as one might see in OCaml or Java.

```
print : Channel → String → Unit
```

For example, here is a simple function to print each element of a pair of strings to a given channel:

```
print_pair : String × String → Channel → Unit
print_pair = fun p chan →
                chan.print(fst p);
                chan.print(snd p)
```

Here, for clarity we use a semicolon for sequencing, and write `print` in method-invocation style *à la* Java (to make it easy to distinguish the file handle from the string argument).

To support capability safety (and thereby obtain purity as a side-effect(!)) we extend the language with a new type constructor `Safe a`, denoting the set of expressions of type `a` which are *capability-safe* – i.e., they own no file handles and so their execution cannot do any printing, unless a capability is passed. We add the introduction form `box(e)` to introduce a value whose type is `Safe a`; the type

system accepts this if `e` has type `a` and is recognisably safe, but rejects it otherwise. Here, "recognisably safe" means that the term `e` does not refer to any capability literals, and all of its free variables are safe variables.

To eliminate a value of type `Safe a`, we will use *pattern matching*, writing the elimination form `let box(x) = e1 in e2` to bind the pure expression in `e1` to the variable `x`. The only difference from ordinary pattern matching is that `x` is marked as a safe variable, permitting it to occur inside of safe expressions. Intuitively, this makes sense – `e1` evaluates to a safe value, and so its result should be allowed to be used by other safe expressions.

It turns out that this discipline of tracking whether a variable is safe or not is precisely a *comonadic* type discipline, corresponding to the □ modality in S4 modal logic. Capability-safety is not exactly the same thing as purity, but we will show how to recover *purity from capability-safety* later in this section, and then prove that this encoding works later on. We illustrate the comonadic behaviour of the `Safe` type constructor with the following examples.

If we know that a value is safe, we can `extract` it, giving up that information. Also, since `Safe` is only expressing a property of the underlying value, applying it twice achieves nothing, making `duplicate` an isomorphism. This expresses an idempotent comonad, which encodes the property that a value of type `Safe a` is *safe*.

```
extract : ∀ a. Safe a → a
extract box(x) = x

duplicate : ∀ a. Safe a → Safe (Safe a)
duplicate box(x) = box(box(x))
```

Also, observe that we can apply safe functions to safe values to get safe results, thereby making it *almost* an `Applicative` functor, as shown below. Syntactically, `box(f x)` is accepted, since both the variables `f` and `x` are known to be safe, and so are permitted to occur inside of a safe expression.

```
(⊛) : ∀ a b. Safe (a → b) → Safe a → Safe b
(⊛) box(f) box(x) = box(f x) -- accepted
```

However, arbitrary values are not `Safe` – we cannot mark any value `x` safe because it could own capabilities. So this function is rejected.

```
pure : ∀ a. a → Safe a
pure x = box(x) -- REJECTED
```

Nor can we write an `fmap` for `Safe`, which applies an arbitrary function to a pure argument, and tries to return a pure result.

```
fmap : ∀ a b. (a → b) → Safe a → Safe b
fmap f box(x) = box(f x)  -- REJECTED
```

Semantically, the function `f` may own capabilities, and so it may have side-effects. Syntactically, since `f` is an impure variable, it is simply not allowed to occur in the pure expression `box(f x)`. Only if we mark both the function and the argument as `Safe` can we apply it, as we saw in `(⊛)`.

However, `Safe` *is* a functor in the semantic sense – the absence of an `fmap` action indicates that this functor lacks *tensorial strength*. (This also means that safety is not definable in Haskell, since all definable functors are strong.)

The capability discipline permits typing functions whose behaviour is intermediate between pure and effectful. First, suppose we see the following type signature for a print function:

```
safe_print : Safe (Channel → String → Unit)
-- definition not visible
```

Without looking at the definition of `safe_print`, we can make some inferences about its side-effects. Since it is marked safe, we can immediately infer that *if* this function performs a side-effect, it can print *only on the channel* that it binds. In other words, it *cannot* use an ambient capability to perform side-effects.

Similarly, consider the following type declaration:

```
multi_print : Safe (List Channel → String → Unit)
-- definition not visible
```

Again, we do not know anything about the body of the definition (perhaps it prints its string argument to all of the channels it receives, or perhaps not), but due to the typing discipline, we know that `multi_print` is safe, and hence, owns no capabilities of its own. As a result, we can make some inferences about the following two declarations:

```
x : Unit                                y : Unit
x = let box(f) = multi_print in         y = let box(f) = multi_print in
    f [stdout, stderr] "Hello world"        f [] "Hello world"
```

The definition of `x` passes two channels to `multi_print`, and so it may have an effect (it might use it to print on either of these channels). On the other hand, we *know* that the evaluation of `y` *will not* have an effect – we know that `multi_print` owned no channels, and we did not give it any channels, therefore it can perform *no effects*. The purity of this function will *depends on the inputs that were passed to it*. Moreover, we know this without having to see the definition of `multi_print`!

Even though capability-safety is a more primitive notion than purity, it is strong enough to encode purity. Revisiting our `map` example from section 1, we can now rewrite it using **`Safe`**.

```
map : ∀ a b. Safe (Safe a → b) → List (Safe a) → List b
map box(f) []        = []
map box(f) (x :: xs) = let  z = f x in
                       let zs = map (box(f)) xs in
                       z :: zs
```

Intuitively, a safe function can only have an effect if its argument gives it any capabilities, and we can prohibit a function argument from bearing capabilities by giving it a safe type. Hence, we can model the *pure* function space $A \Rightarrow B$ with the impure function space, with the type **`Safe`**`(Safe A → B)`.

An additional benefit of the comonadic type discipline is that it dramatically simplifies the process of *capability taming*. A language is capability-safe when programs have no way to access to *ambient authorities*. As a result, capability-safety has historically been understood not just as a property of the language, but also of the standard library. In particular, if the standard library exposes globally-visible channels like **`stdout`** and **`stderr`**, any program in the language can refer to them, and thereby have write effects. As a result, a project like Joe-E [Mettler et al. 2010] involves a massive effort to rewrite the whole standard library of Java. In constrast, a language with a safety comonad affords a gradual approach – the bindings in the standard library can all be marked impure by default, and as the functions are audited, they can gradually be marked safe, allowing more and more capability-safe programs can be written. This lets language implementors and programmers gradually opt-in to capability safety, making it easier to migrate language ecosystems, and also illustrates the importance of being able to track the safety of variable bindings.

Types	A, B	::=	$\mathsf{unit} \mid \mathsf{str} \mid \mathsf{cap}$
			$\mid A \times B \mid A \Rightarrow B \mid \Box A$
Terms	e	::=	$() \mid s \mid e_1 \,.\, \mathsf{print}(e_2)$
			$\mid (e_1 \,, e_2) \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e$
			$\mid x \mid \lambda x : A.\, e \mid e_1\, e_2$
			$\mid \mathsf{box}\, \boxed{e} \mid \mathsf{let\ box}\, \boxed{x} = e_1\, \mathsf{in}\, e_2$
Values	v	::=	$() \mid s \mid (v_1 \,, v_2)$
			$\mid x \mid \lambda x : A.\, e \mid \mathsf{box}\, \boxed{e}$
Qualifiers	q, r	::=	$\mathbf{s} \mid \mathbf{i}$
Contexts	Γ, Δ, Ψ	::=	$\cdot \mid \Gamma, x : A^q$
Substitutions	θ, ϕ	::=	$\langle\rangle \mid \langle \theta, e^q/x \rangle$

Fig. 1. Grammar

3 TYPING

We give the grammar of our language in figure 1.

We have the usual type constructors for unit, products, and functions from the simply-typed lambda calculus. In addition to this, we have the type str for strings, and the type cap representing output channels (used in the imperative $e_1 \,.\, \mathsf{print}(e_2)$ statement). Finally, we add the comonadic $\Box$ type constructor which corresponds to the **Safe** type constructor we introduced in section 2.

Despite the fact that there is a *type* cap of channels, and a print operation which uses them, there are no introduction forms for them. This is intentional! The absence of this facility corresponds to the principle of *capability safety* – the only capabilities a program should possess are those that are passed by its caller. So, a complete program will either be a function that receives a capability token as an argument, or have free variables that the system can bind capability tokens to. [1]

The expressions in our language include the usual ones from the simply-typed lambda calculus, constants s for strings, and print. We also have an introduction form box $\boxed{e}$, and a let box elimination form for the $\Box A$ type; we'll explain how these work later. Values are a subset of expressions, but box turns any expression into a value. [2]

We would like a modal type system where we can distinguish between expressions with and without side-effects. Following the style of [Pfenning and Davies 2001] for S4 modal logic, we could build a dual-context calculus. However, such a setup makes it difficult to define substitution; we can avoid dual contexts by tagging terms with qualifiers instead. We use two qualifiers that we can annotate terms with, in the appropriate places. We use **s** to tag *safe* terms, and **i** to tag *impure* terms. [3]

Next, we define contexts of variables. A well-formed context is either the empty context $\cdot$, or an extended context with a variable x of type A and qualifier q. Finally, we give a grammar for substitutions. A substitution is either the empty substitution $\langle\rangle$, or an extended substitution with an expression e substituted for variable x qualified by q.

[1] Of course, a full system should have the ability to create new private capabilities of its own. We omit this to keep the denotational semantics simple, but discuss how to add it in section 8.

[2] We write sequencing as $e_1 \,;\, e_2$, which is sugar for $(\lambda x : \mathsf{unit}.\, e_2)\, e_1$.

[3] We use different colours to distinguish *safe* and *impure* syntactic objects, and we'll follow this convention henceforth. When we have unknown qualifiers occurring on terms, we *highlight* them in a different colour, and the colour changes to the appropriate one when the qualifier is **s** or **i**.

$x : A^q \in \Gamma$ x is a variable of type A with qualifier q in context Γ
$\Gamma \vdash e : A$ e is an expression of type A in context Γ
$\Gamma \vdash^{\mathbf{s}} e : A$ e is a *safe* expression of type A in context Γ

(a) Typing Judgements

$\Gamma \supseteq \Delta$ Γ is a weakening of context Δ
$\Gamma \vdash \theta : \Delta$ θ is a well-formed substitution from context Γ to Δ

(b) Weakening and Substitution Judgements

$\Gamma \vdash e_1 \approx e_2 : A$ e_1 and e_2 are equal expressions of type A in context Γ

(c) Equality Judgements

Fig. 2. Judgement forms

$$\frac{}{\Gamma \vdash () : \mathsf{unit}}\ \text{unitI} \qquad \frac{}{\Gamma \vdash s : \mathsf{str}}\ \text{strI} \qquad \frac{\Gamma \vdash e_1 : \mathsf{cap} \qquad \Gamma \vdash e_2 : \mathsf{str}}{\Gamma \vdash e_1.\mathsf{print}(e_2) : \mathsf{unit}}\ \textsc{Print}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\ \times\text{I} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{fst}\, e : A}\ \times\text{E}_1 \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{snd}\, e : B}\ \times\text{E}_2$$

$$\frac{x : A^q \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma, x : A^{\mathbf{i}} \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \Rightarrow B}\ \Rightarrow\text{I} \qquad \frac{\Gamma \vdash e_1 : A \Rightarrow B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \Rightarrow\text{E}$$

$$\frac{\Gamma^{\mathbf{s}} \vdash e : A}{\Gamma \vdash^{\mathbf{s}} e : A}\ \textsc{ctx-safe} \qquad \frac{\Gamma \vdash^{\mathbf{s}} e : A}{\Gamma \vdash \mathsf{box}\, \boxed{e} : \square A}\ \square\text{I} \qquad \frac{\Gamma \vdash e_1 : \square A \qquad \Gamma, x : A^{\mathbf{s}} \vdash e_2 : B}{\Gamma \vdash \mathsf{let\ box}\, \boxed{x} = e_1\, \mathsf{in}\, e_2 : B}\ \square\text{E}$$

Fig. 3. Typing Rules

3.1 Typing Judgements

In figure 2a we introduce three kinds of judgement forms, and give typing rules in figure 3.

We have the usual introduction and elimination rules for constants and products. If a variable is present in the context, we can introduce it, using the Var rule. In the introduction rule for functions $\Rightarrow$ I, we mark the hypothesis as *impure* when forming a λ-expression, because we do not want to restrict function arguments in general. The elimination rule $\Rightarrow$ E, or function application works as usual. The print statement performs side-effects but has the type unit. We need to do more work to add the comonadic type constructor.

We can mark a term as *safe* if it was well-typed in a *safe* context, where every variable has the **s** annotation. So we define a syntactic *purify* operation, which acts on contexts; applying it drops the terms with the *impure* annotation, as shown in figure 4a. This is expressed by the ctx-safe rule, which introduces a *safe* expression using the *safe* judgement form. And then, we can put it in a box using the $\square$I rule, to get a $\square$-typed value.

$$(\cdot)^{\mathbf{s}} := \cdot \qquad (\Gamma, x : A^{\mathbf{s}})^{\mathbf{s}} := \Gamma^{\mathbf{s}}, x : A^{\mathbf{s}} \qquad (\Gamma, x : A^{\mathbf{i}})^{\mathbf{s}} := \Gamma^{\mathbf{s}}$$

(a)

$$\langle\rangle^{\mathbf{s}} := \langle\rangle \qquad \langle\theta, e^{\mathbf{s}}/x\rangle^{\mathbf{s}} := \langle\theta^{\mathbf{s}}, e^{\mathbf{s}}/x\rangle \qquad \langle\theta, e^{\mathbf{i}}/x\rangle^{\mathbf{s}} := \theta^{\mathbf{s}}$$

(b)

Fig. 4. Purifying Contexts and Substitutions

$$\frac{}{x : A^q \in (\Gamma, x : A^q)}\ \in\text{-ID} \qquad \frac{x : A^q \in \Gamma \qquad (x \neq y)}{x : A^q \in (\Gamma, y : B^r)}\ \in\text{-EX}$$

(a) Context Membership Rules

$$\frac{}{\cdot \supseteq \cdot}\ \supseteq\text{-ID} \qquad \frac{\Gamma \supseteq \Delta}{\Gamma, x : A^q \supseteq \Delta, x : A^q}\ \supseteq\text{-CONG} \qquad \frac{\Gamma \supseteq \Delta}{\Gamma, x : A^q \supseteq \Delta}\ \supseteq\text{-WK}$$

(b) Weakening Rules

$$\frac{}{\Gamma \vdash \langle\rangle : \cdot}\ \text{SUB-ID}$$

$$\frac{\Gamma \vdash \theta : \Delta \qquad \Gamma \vdash^{\mathbf{s}} e : A}{\Gamma \vdash \langle\theta, e^{\mathbf{s}}/x\rangle : \Delta, x : A^{\mathbf{s}}}\ \text{SUB-SAFE} \qquad \frac{\Gamma \vdash \theta : \Delta \qquad \Gamma \vdash v : A}{\Gamma \vdash \langle\theta, v^{\mathbf{i}}/x\rangle : \Delta, x : A^{\mathbf{i}}}\ \text{SUB-IMPURE}$$

(c) Substitution Rules

Fig. 5. Membership, Weakening and Substitution Rules

We give an elimination rule □E using the let box binding form. Given an expression in the □ type, we bind the underlying *safe* expression to the variable x. With an extended context that has a free variable x marked *safe*, if we can produce a well-typed expression in the motive, the elimination is complete.

3.2 Weakening and Substitution

Next, we can define syntactic weakening and substitution.

3.2.1 Membership. We give the standard rules for the context membership judgement in figure 5a, following Barendregt's variable convention. The only difference is that variables now have an extra *safety* annotation.

3.2.2 Weakening. The context weakening relation follows the usual rules, as shown in figure 5b, with the extra *safety* annotation on free variables in contexts. $\Gamma \supseteq \Delta$ indicates that Γ has more variables than Δ, and is defined as an inductive relation in figure 5b. We can prove a syntactic weakening lemma.

Lemma 3.1 Syntactic weakening. *If* $\Gamma \supseteq \Delta$ *and* $\Delta \vdash e : A$, *then* $\Gamma \vdash e : A$.

3.2.3 Substitution. Substitution requires a bit more care. First, we define the judgement $\Gamma \vdash \theta : \Delta$, which says that θ is a well-formed substitution from context Γ to Δ. Since our language is effectful,

we restrict the definition of substitutions, in figure 5c to substitute *values* for *impure* variables, while permitting *safe* expressions for *safe* variables.

Furthermore, we define the syntactic substitution function, which applies a substitution on raw terms. This is mostly standard, but when substituting under a binder, we do a renaming of the bound variable by extending the substitution with an appropriately annotated variable. To substitute inside a box-ed expression, we have to *purify* the substitution when using it. We extend the *purify* operation to substitutions as well; it simply drops the *impure* substitutions, as shown in figure 4b.

Definition 3.2 (Syntactic substitution on variables).

$$\theta[x] := \begin{cases} \lightning & \theta = \langle\rangle \\ e & \theta = \langle \phi, e^q/x \rangle \\ \phi[x] & \theta = \langle \phi, e^q/y \rangle, x \neq y \end{cases}$$

Definition 3.3 (Syntactic substitution on raw terms).

$$\begin{aligned}
\theta(x) &:= \theta[x] \\
\theta(()) &:= () \\
\theta(s) &:= s \\
\theta((e_1 \,, e_2)) &:= (\theta(e_1) \,, \theta(e_2)) \\
\theta(\mathsf{fst}\, e) &:= \mathsf{fst}\, \theta(e) \\
\theta(\mathsf{snd}\, e) &:= \mathsf{snd}\, \theta(e) \\
\theta(\lambda x.\, e) &:= \lambda y.\, \langle \theta, y^{\mathbf{i}}/x \rangle(e) \\
\theta(e_1\, e_2) &:= \theta(e_1)\, \theta(e_2) \\
\theta(\mathsf{box}\, \boxed{e}) &:= \mathsf{box}\, \boxed{\theta^{\mathbf{s}}(e)} \\
\theta(\mathsf{let\ box}\, \boxed{x} = e_1\, \mathsf{in}\, e_2) &:= \mathsf{let\ box}\, \boxed{y} = \theta(e_1)\, \mathsf{in}\, \langle \theta, y^{\mathbf{s}}/x \rangle(e_2) \\
\theta(e_1 \,.\, \mathsf{print}(e_2)) &:= \theta(e_1) \,.\, \mathsf{print}(\theta(e_2))
\end{aligned}$$

Then, we can prove the type-correctness of substitution:

Theorem 3.4 Syntactic substitution. *If* $\Gamma \vdash \theta : \Delta$ *and* $\Delta \vdash e : A$, *then* $\Gamma \vdash \theta(e) : A$.

4 SEMANTICS

In this section, we describe a concrete denotational model of capabilities and the abstract categorical structure it models.

4.1 Capability Spaces

Let $\mathcal{C}$ be a fixed set of capability names, possibly countably infinite. The powerset $\mathfrak{P}(\mathcal{C})$ denotes the set of all subsets of $\mathcal{C}$, and $(\mathfrak{P}(\mathcal{C}); \emptyset, \mathcal{C}, \subseteq)$ is the complete lattice ordered by set inclusion.

A capability space $X = (|X|\,, w_X)$ is a set $|X|$ with a weight relation $w_X : |X| \twoheadrightarrow \mathfrak{P}(\mathcal{C})$ that assigns sets of capabilities to each member in X. Intuitively, we think of the set $|X|$ as the set of values of the type X, and we think of the weight relation w_X as defining the possible sets of capabilities that each value may own.

We require maps between capability spaces to preserve weights, i.e., a map between the underlying sets $|X|$ and $|Y|$ is a morphism of capability spaces iff for each x in $|X|$, all the weights in Y for $f(x)$ are bounded by the weights in X for x. If we think of a function $f : X \to Y$ as a term of type Y with a free variable of type X, then this condition ensures that the capabilities of the term are limited to at most those of its free variables. In other words, weight-preserving functions are precisely those

which are capability-safe; they do not have unauthorised access to arbitrary capabilities, and they *do not have any ambient authority*.

We now formally define the category of capability spaces $\mathcal{C}$, with objects as capability spaces and morphisms as weight-preserving functions.

Definition 4.1 (Category $\mathcal{C}$ of capability spaces).

$$\mathcal{O}bj_{\mathcal{C}} \coloneqq X = (|X| : \mathrm{Set},\ w_X : |X| \twoheadrightarrow \mathfrak{P}(\mathcal{C}))$$

$$\mathcal{H}om_{\mathcal{C}}(X, Y) \coloneqq \left\{ f \in |X| \to |Y| \;\middle|\; \begin{array}{l} \forall x, C_x, w_X(x, C_x) \Rightarrow \\ \exists C_y \subseteq C_x, w_Y(f(x), C_y) \end{array} \right\}$$

We remark that the definition of this category is inspired by the category of length spaces defined by Hofmann [2003], which again associates intensional information (in his work, memory usage, and in ours, capabilities) to a set-theoretic semantics.

4.2 The Direct Semantics

We now actually have specified enough of the semantic model to interpret our language.

Because the capability space model is a "structured sets" model, in which each object is a set with some additional structure (i.e., the weights), and morphisms are ordinary set-theoretic functions (which are required to preserve this structure), we can interpret an expression e with typing derivation $\Gamma \vdash e : A$, as a function $\Gamma \to TA$. This is an ordinary set-theoretic function which takes an element of Γ (i.e., a substitution binding each variable to an element of its type) to a monadic computation (a writer monad) producing an element of A. To make this clear, we first give an interpretation written in the style of a monadic program in Haskell syntax in figure 6.

For example, function application $e_1\ e_2$ exhibits a right-to-left evaluation order: we first evaluate e_2 (with environment γ) to an argument a, then evaluate e_1 (with environment γ) to a function f, and then apply the argument to the function. The $e_1 \,.\, \mathsf{print}(e_2)$ method evaluates e_1 to a channel c, e_2 to a string s, and then represents its effect using the writer monad: it returns a map saying that s was printed to the channel c. The interpretation of box $\boxed{e}$ is perhaps the most interesting – it interprets e in a context where all capability-bearing bindings are discarded. As a result, even though e is a monadic term, we know that it could not have written to any channels, and so we can then discard (using `fst`) the writer monad's output component without losing any information.

However, while writing the semantics as a naive set-theoretic semantics makes it easy to read, we still have to check that this definition actually does define a genuine weight-preserving morphism between capability spaces. As the interpretation of box $\boxed{e}$ makes clear, this is not a trivial fact. Indeed, even though this semantics is in fact capability-safe, checking that is an incredibly tedious and error-prone affair – we have to go through every semantic clause and check not just that each and every operation we use is weight-preserving, but that all their compositions are weight-preserving as well.

To manage and organize this work more efficiently, we turn to a categorical semantics. In the categorical semantics, each type is an object, and each type constructor is interpreted as a functor with operators satisfying some universal properties. This way, we can check that the interpretation of each type connective works the way we want in isolation, without having to worry about any interactions with the rest of the calculus. Furthermore, the universal properties make it easy to check that our language satisfies the equational theory that we desire.

Another important benefit is that by formulating the semantics in a categorical style, the semantics and equational theory only depends upon the algebraic structure of the category of capability spaces. That is, we use the *cartesian closed* structure, the *monoidal idempotent comonad*, the *strong monad*, and the *cancellation isomorphism* Φ; the proofs of our theorems use the universal property for each categorical construction. Indeed, our semantics is nearly independent of the specific set

$$[\![\frac{}{\Gamma \vdash () : \mathsf{unit}}]\!]\gamma := \texttt{return ()} \qquad [\![\frac{}{\Gamma \vdash s : \mathsf{str}}]\!]\gamma := \texttt{return s}$$

$$[\![\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}]\!]\gamma := \begin{array}{l}\texttt{do g} \leftarrow [\![\Gamma \vdash e_2 : B]\!]\gamma \\ \quad \texttt{f} \leftarrow [\![\Gamma \vdash e_1 : A]\!]\gamma \\ \quad \texttt{return (f,g)}\end{array}$$

$$[\![\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{fst}\, e : A}]\!]\gamma := \begin{array}{l}\texttt{do f} \leftarrow [\![\Gamma \vdash e : A \times B]\!]\gamma \\ \quad \texttt{return (fst f)}\end{array}$$

$$[\![\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{snd}\, e : B}]\!]\gamma := \begin{array}{l}\texttt{do f} \leftarrow [\![\Gamma \vdash e : A \times B]\!]\gamma \\ \quad \texttt{return (snd f)}\end{array}$$

$$[\![\frac{x : A^q \in \Gamma}{\Gamma \vdash x : A}]\!]\gamma := \texttt{return } (\gamma\ \texttt{x})$$

$$[\![\frac{\Gamma, x : A^{\mathbf{i}} \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \Rightarrow B}]\!]\gamma := \texttt{return (fun a} \rightarrow [\![\Gamma, x : A^{\mathbf{i}} \vdash e : B]\!]\,(\gamma, \texttt{a}))$$

$$[\![\frac{\Gamma \vdash e_1 : A \Rightarrow B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}]\!]\gamma := \begin{array}{l}\texttt{do a} \leftarrow [\![\Gamma \vdash e_2 : A]\!]\gamma \\ \quad \texttt{f} \leftarrow [\![\Gamma \vdash e_1 : A \Rightarrow B]\!]\gamma \\ \quad \texttt{f a}\end{array}$$

$$[\![\frac{\Gamma \vdash e_1 : \mathsf{cap} \qquad \Gamma \vdash e_2 : \mathsf{str}}{\Gamma \vdash e_1.\mathsf{print}(e_2) : \mathsf{unit}}]\!]\gamma := \begin{array}{l}\texttt{do s} \leftarrow [\![\Gamma \vdash e_2 : \mathsf{str}]\!]\gamma \\ \quad \texttt{c} \leftarrow [\![\Gamma \vdash e_1 : \mathsf{cap}]\!]\gamma \\ \quad \texttt{((),fun c'} \rightarrow \texttt{if c = c'} \\ \qquad\qquad\qquad \texttt{then s else } \varepsilon\texttt{)}\end{array}$$

$$[\![\frac{\Gamma^{\mathbf{s}} \vdash e : A}{\Gamma \vdash \mathsf{box}\,\boxed{e} : \square A}]\!]\gamma := \texttt{return (fst } ([\![\Gamma^{\mathbf{s}} \vdash e : A]\!]\gamma^{\mathbf{s}}))$$

$$[\![\frac{\Gamma \vdash e_1 : \square A \qquad \Gamma, x : A^{\mathbf{s}} \vdash e_2 : B}{\Gamma \vdash \mathsf{let\ box}\,\boxed{x} = e_1\,\mathsf{in}\,e_2 : B}]\!]\gamma := \begin{array}{l}\texttt{do a} \leftarrow [\![\Gamma \vdash e_1 : \square A]\!]\gamma \\ \quad [\![\Gamma, x : A^{\mathbf{s}} \vdash e_2 : B]\!]\,(\gamma,\ \texttt{a})\end{array}$$

Fig. 6. Direct interpretation of expressions

of effects – we only use the specific definition of the monad in the interpretation of print. Since our theorems depend only upon the algebraic structure, our results will still hold if we switched to another category with this structure. We say more about that in section 8.

We give the categorical structure of the category of capability spaces in the remainder of this section, and then give the categorical intepretation (which is actually semantically identical to the direct interpretation) in the following section.

4.3 Cartesian Closed Structure

We now observe that $\mathcal{C}$ inherits the *cartesian closed* structure of Set. The definitions are the same as in the case of sets, but we additionally have to verify that the morphisms are weight-preserving.

Expression	Type	Weight
`unit`	`Unit`	$\emptyset$
`stdout`	`Channel`	$\{$ `stdout` $\}$
`fun c → unit`	`Channel → Unit`	$\emptyset$
`fun c → c`	`Channel → Channel`	$\emptyset$
`fun c → c.print("hello")`	`Channel → Unit`	$\emptyset$
`fun c → stdout.print("hello")`	`Channel → Unit`	$\{$ `stdout` $\}$
(c_1, c_2)	`Channel × Channel`	$\{ c_1, c_2 \}$
[`stdout`, c_1, c_2]	`List Channel`	$\{$ `stdout`, c_1, c_2 $\}$

Fig. 7. Expressions and their capability weights

Definition 4.2 (Terminal Object).

$$|1| := \{ * \}$$
$$w_1 := \{ (*, \emptyset) \}$$

The terminal object 1 is the usual singleton set, and it has no capabilities. For any object A, the unique terminal map $! : A \to 1$ is given by $!_A(a) = *$, which is evidently weight preserving.

Definition 4.3 (Product).

$$|A \times B| := |A| \times |B|$$
$$w_{A \times B} := \{ ((a,b), C_a \cup C_b) \mid w_A(a, C_a) \wedge w_B(b, C_b) \}$$

Products are formed by pairing as usual, and the set of capabilities of a pair of values is the union of their capabilities. The projection maps $\pi_i : A_1 \times A_2 \to A_i$ are just the projections on the underlying sets, which are weight preserving as well. We verify the universal property in **??** in the appendix.

Definition 4.4 (Exponential).

$$|A \to B| := |A| \to |B|$$
$$w_{A \to B} := \left\{ (f, C_f) \;\middle|\; \begin{array}{l} \forall a, C_a, w_A(a, C_a) \Rightarrow \\ \exists C_b \subseteq C_f \cup C_a, w_B(f(a), C_b) \end{array} \right\}$$

Exponentials are given by functions on the underlying sets, but we have to assign capabilities to the closure. We only record those capabilities which are induced by the function, for some value in the domain. That is, for a function closure $f : A \to B$, if a given value $a \in A$ has weight assignment C_a, and if there is a weight assignment C_b for $f(a)$, then the weight of the closure f is given by the all the capabilities it had access to its environment.

We verify that our definition satisfies the currying isomorphism in **??** in the appendix, where we name the currying/uncurrying and evaluation maps.

This cartesian closed structure on $\mathcal{C}$ suffices to interpret the simply-typed lambda calculus. To illustrate the semantics, we give some examples of closed terms with their unique capability weightings in figure 7.

4.4 Monad

Our language supports printing strings along a channel, and to model this print effect, we will structure our semantics monadically, in the style of Moggi [1991]. We define a strong monad T on $\mathcal{C}$ as follows.

Definition 4.5 ($\Sigma^* : \mathcal{C}$). Σ^* is the set of strings, with an empty string $\varepsilon : 1 \to \Sigma^*$, and a multiplication $\bullet : \Sigma^* \times \Sigma^* \to \Sigma^*$ given by concatenation, making it a monoid object. Strings are constants and hence do not have any weights.

Definition 4.6 ($\mathcal{C} : \mathcal{C}$). $\mathcal{C}$ is the object of capabilities in $\mathcal{C}$ such that $w_{\mathcal{C}} = \{ (c, \{ c \}) \}$ for every $c \in \mathcal{C}$. Note that there are no global sections for this object, because the map $1 \to \mathcal{C}$ *is not weight-preserving*. In other words, we do not have access to arbitrary capabilities, as evident by the lack of an introduction rule for the cap type. This indicates the lack of ambient authority.

Definition 4.7 ($T : \mathcal{C} \longrightarrow \mathcal{C}$).

$$
\begin{aligned}
|T(A)| &:= |A| \times (\mathcal{C} \to \Sigma^*) \\
w_{T(A)} &:= \left\{ ((a, o), C_a \cup \{ c \mid o(c) \neq \varepsilon \}) \;\middle|\; w_A(a, C_a) \right\}
\end{aligned}
$$

Using the monoid $(\Sigma^*; \varepsilon, \bullet)$, we can define T to be the writer monad which adds an output function that records the output produced in each channel. The weight of a monadic computation is taken to be the weight of the returned value, unioned with all the channels that *anything* was written to. This corresponds to the intuition that a computation which performs I/O on a channel must possess the capability to do so.

Definition 4.8 (T is a monad). The unit and multiplication of the monad are defined below. We check that they are morphisms, and state and verify the monad laws in **??** in the appendix.

$$
\begin{aligned}
\eta_A &: A \to TA & \qquad \mu_A &: TTA \to TA \\
a &\mapsto (a, \lambda c.\varepsilon) & ((a, o_1), o_2) &\mapsto (a, \lambda c. o_2(c) \bullet o_1(c))
\end{aligned}
$$

Definition 4.9 (T is a strong monad). T is strong with respect to products, with a natural family of left and right strengthening maps.

$$
\begin{aligned}
\tau_{A,B} : A \times TB &\to T(A \times B) & \qquad \sigma_{A,B} : TA \times B &\to T(A \times B) \\
(a, (b, o)) &\mapsto ((a, b), o) & ((a, o), b) &\mapsto ((a, b), o)
\end{aligned}
$$

We use this to define the natural map $\beta_{A,B}$, which evaluates a pair of effects, as follows. Notice that it evaluates the effect on the right before the one on the left; we expand more on that in **??** in the appendix, and verify the appropriate coherences.

$$
\begin{aligned}
\beta_{A,B} &: TA \times TB \to T(A \times B) \\
\beta_{A,B} &:= \tau_{TA,B} \fatsemi T\sigma_{A,B} \fatsemi \mu_{A \times B}
\end{aligned}
$$

4.5 Comonad

To model the □ type constructor, we define an endofunctor □ on $\mathcal{C}$ below; it filters out values that *do not* possess any capabilities, i.e., values that are *safe*.

Definition 4.10 ($\Box : \mathcal{C} \longrightarrow \mathcal{C}$).

$$
\begin{aligned}
|\Box A| &:= \left\{ a \in |A| \;\middle|\; \forall C_A, w_A(a, C_a) \Rightarrow C_a = \emptyset \right\} \\
w_{\Box A} &:= \{ (a, \emptyset) \}
\end{aligned}
$$

On objects, we simply restrict the set to the subset of values that *only* have the empty set $\emptyset$ of capabilities. □ acts on morphisms by restricting the domain of the function to $|\Box A|$. For any weight-preserving function f, $\Box(f)$ is trivially weight-preserving as a function between sets with empty capabilities.

This type constructor is especially useful at function type $\Box(A \to B)$, since in general the environment can hold capabilities, and the $\Box$ constructor lets us rule those out. We further claim that $\Box$ is an idempotent strong monoidal comonad.

Definition 4.11 ($\Box$ is an idempotent comonad). The counit ε and comultiplication δ of the comonad are the natural families of maps given by the inclusion and the identity maps on the underlying set. δ is a natural isomorphism making it idempotent. We state and verify the comonad laws in **??** in the appendix.

$$\begin{array}{cc} \varepsilon_A : \Box A \to A & \delta_A : \Box A \xrightarrow{\sim} \Box\Box A \\ a \mapsto a & a \mapsto a \end{array}$$

Definition 4.12 ($\Box$ is a strong monoidal functor). The functor is strong monoidal, in that it preserves the monoidal structure of products (and tensors, see the sequel in subsection 4.7). The identity element is preserved, and we have *natural isomorphisms* given by pairing on the underlying sets.

$$\begin{array}{cc} m^1 : 1 \xrightarrow{\sim} \Box 1 & m^{\times}_{A,B} : (\Box A \times \Box B) \xrightarrow{\sim} \Box(A \times B) \\ * \mapsto * & (a\,,b) \mapsto (a\,,b) \end{array}$$

$$\begin{array}{cc} m^I : I \xrightarrow{\sim} \Box I & m^{\otimes}_{A,B} : (\Box A \otimes \Box B) \xrightarrow{\sim} \Box(A \otimes B) \\ * \mapsto * & (a\,,b) \mapsto (a\,,b) \end{array}$$

We remark that $\Box$ is not a strong comonad, i.e., it does not possess a tensorial strength. This makes it impossible to evaluate an arbitrary function under the comonad, as we saw in section 2. [4]

4.6 The Comonad cancels the Monad

We make the following observation. There is an isomorphism Φ_A, natural in A, where the comonad $\Box$ cancels the monad T. In programming terms, this says that *an effectful computation with no capabilities can perform no effects* — i.e., it is *safe*. Note that this definition works because of the particular definition of the monad T we chose, in which the weight of a computation includes all the channels it printed on. Consequently a computation of weight zero cannot print on any channel, and so must be *safe*! We verify this fact in **??** in the appendix.

Definition 4.13 ($\Phi : \Box T \Rightarrow \Box$).

$$\begin{array}{c} \Phi_A : \Box TA \xrightarrow{\sim} \Box A \\ (a, o) \mapsto a \end{array}$$

This property is crucial and we will exploit it to manage our syntax: we use it to justify treating terms in *safe* contexts as *safe*, without needing a second grammar for *safe* expressions.

4.7 Other remarks

While the monad and comonad, together with the cartesian closed structure, suffice to interpret our language, it is worth noting that the category $\mathcal{C}$ also admits a *monoidal closed* structure.

[4] For Haskellers, the $\Box$ functor is not a `Functor`!

687 *4.7.1 Monoidal Closed Structure.*

688
689 *Definition 4.14 (Tensor product).*

690
$$\begin{aligned} |A \otimes B| &:= |A| \times |B| \\ w_{A\otimes B} &:= \left\{ ((a,b), C_a \cup C_b) \;\middle|\; C_a \# C_b \wedge w_A(a, C_a) \wedge w_B(b, C_b) \right\} \\ I &:= 1 \end{aligned}$$

693
694 The tensor product is given by pairing, with unit 1, but it only restricts to pairs whose sets of
695 capabilities are disjoint. However, this tensor product also enjoys a right adjoint.

696 *Definition 4.15 (Linear exponential).*

697
$$\begin{aligned} |A \multimap B| &:= |A| \to |B| \\ w_{A\multimap B} &:= \left\{ (f, C_f) \;\middle|\; \begin{array}{l} \forall a, C_a, w_A(a, C_a) \wedge C_f \# C_a \Rightarrow \\ \exists C_b \subseteq C_f \cup C_a, w_B(f(a), C_b) \end{array} \right\} \end{aligned}$$

701
702 The linear exponential works the same way as the exponential, except that we have to restrict it
703 to satisfy the disjointness condition for the tensor product. We verify that this definition satisfies
704 the tensor-hom adjunction in **??** in the appendix.

705 This supports an interpretation of a *linear* (actually, affine) type theory. The disjointness condi-
706 tions in the interpretation of tensor product and linear implication are essentially the same as the
707 disjointness conditions in the definition of the separating conjunction $A * B$ and magic wand $A \mathrel{-\!\!*} B$
708 in separation logic [Reynolds 2002]. In separation logic, capabilities correspond to ownership of par-
709 ticular memory locations, and in our setting, capabilities correspond to the right to access a channel.

710 Our model reassuringly suggests that operating systems researchers and program verification
711 researchers both identified the same notion of capability. However, it seems that the fact that these
712 are *exactly* the same idea was overlooked because operating systems researchers focused on the
713 cartesian closed structure, and semanticists focused on the monoidal closed structure!

714 *4.7.2 Adding other effects.* While we used the writer monad for print, we can also define other
715 interesting monads using the capability space model which can be used to interpret a language with
716 other effects. For example, we show how to define an exception monad which allows raising a single
717 exception, and a state monad with a global heap, in **????** in the appendix. For each of these monads,
718 we need to choose a suitable weight assignment, all of which can be cancelled by our *safety* comonad!

719
720 ## 5 INTERPRETATION

721
722 We now interpret the syntax of our language. We adopt some standard notation to work with our
723 categorical combinators. [5] The sequential composition of two arrows, in the diagrammatic order, is
724 $f ; g$. The product of morphisms f and g is $\langle f , g \rangle$ (also called a fork operation in the algebra of pro-
725 gramming community [Gibbons 2000]), and $[f \times g]$ is parallel composition with products. We define
726 these using the universal property of products and composition (as shown in **??** in the appendix).

727
728 ### 5.1 Types and Contexts

729 We interpret types as objects in $\mathcal{C}$, as shown in figure 8a. Note that we use the monad in the inter-
730 pretation of functions, following the call-by-value computational lambda-calculus interpretation
731 in [Moggi 1989]. We use the comonad to interpret the $\square$ modality. We use the particular objects Σ^*
732 and $\mathcal{C}$ to interpret strings and capabilities respectively.

733
734 [5] We sometimes drop the denotation symbol for brevity, i.e., we write $!_\Gamma$ instead of $!_{[\![\Gamma]\!]}$, or δ_{Γ^s} instead of $\delta_{[\![\Gamma^s]\!]}$.

735

$$\llbracket \mathsf{unit} \rrbracket := 1 \qquad \llbracket A \times B \rrbracket := \llbracket A \rrbracket \times \llbracket B \rrbracket$$
$$\llbracket \mathsf{str} \rrbracket := \Sigma^* \qquad \llbracket A \Rightarrow B \rrbracket := \llbracket A \rrbracket \to T\llbracket B \rrbracket$$
$$\llbracket \mathsf{cap} \rrbracket := \mathcal{C} \qquad \llbracket \Box A \rrbracket := \Box\llbracket A \rrbracket$$

(a) $\llbracket A \rrbracket : \mathcal{O}bj_{\mathcal{C}}$

$$\llbracket \cdot \rrbracket := 1$$
$$\llbracket \Gamma, x : A^{\mathsf{s}} \rrbracket := \llbracket \Gamma \rrbracket \times \Box\llbracket A \rrbracket$$
$$\llbracket \Gamma, x : A^{\mathsf{i}} \rrbracket := \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket$$

(b) $\llbracket \Gamma \rrbracket : \mathcal{O}bj_{\mathcal{C}}$

Fig. 8. Interpretation of types and contexts

We interpret contexts as finite products of objects, in figure 8b. The comonad is used to interpret the *safe* variables in the context, while the *impure* variables are just arbitrary objects in $\mathcal{C}$.

The judgement $x : A^q \in \Gamma$ is interpreted as a morphism in $\mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$, which we give later in figure 11a. It projects out the appropriately typed and annotated variable from the product in the context. For *safe* variables, we need to use the counit ε to get out of the comonad. [6]

5.2 Expressions

We now give an interpretation for expressions $\Gamma \vdash e : A$, and *safe* expressions $\Gamma \vdash^{\mathsf{s}} e : A$, in figure 9.

To interpret unitI, we use the terminal map ! to simply get to the terminal object 1, then lift it into the monad using η, without performing any effects.

For pair introduction ×I, we evaluate both components of the pair, and compose, then use the strength of the monad T with the β combinator to form the product. [7]

We eliminate products using the $\times\mathrm{E}_1$ and $\times\mathrm{E}_2$ rules. These are interpreted using the corresponding product projection maps, under the functorial action of T.

Variables are introduced using the Var rule, which is interpreted by looking up in the context, for which we use the interpretation of our context membership judgement. This is followed by a trivial lifting into the monad.

To interpret functions using the ⇒ I rule, we simply use the currying map, since our context extension is interpreted as a product. Then we lift it into the monad using η.

To eliminate functions using the ⇒ E rule, we evaluate the operator and operand in an application, followed by a use of the monad strength β to turn it into a pair. Then we use the evaluation map under the functor T to apply the argument. Since the function is effectful, we have to collapse the effects using a μ.

To interpret the □ I rule, we need to interpret the *safe* judgement (defined later), which gives a value of type $\Box A$, and then we lift it into the monad.

To eliminate a box-ed value using the □ E rule, we first evaluate f, which gives a value of type $\Box A$, but under the monad T. We can use it to introduce a *safe* variable in the context, but we use the strength of the monad to shift the product under the T and get an extended context. We evaluate g under this extended context, and then use a μ to collapse the effects.

Finally, to interpret the Print rule, we need to perform a non-trivial effect. We define the function p which builds an output function that records the output on channels. Given any channel c and string s, it returns a value of type $T1$ containing the trivial value $*$; the output function instantiates

[6] When interpreting judgements and inference rules, we write $\llbracket \dfrac{\mathcal{J}_1 \cdots \mathcal{J}_n}{\mathcal{J}} \rrbracket$ to mean the interpretation of $\mathcal{J}$, i.e., we recursively define $\llbracket \mathcal{J} \rrbracket$ under the assumption that we have an interpretation for $\mathcal{J}_i$, i.e., $\llbracket \mathcal{J}_1 \rrbracket, \ldots, \llbracket \mathcal{J}_n \rrbracket$.

[7] The vigilant reader will have noticed that β evaluates the pair from right to left, so the action on the right will be performed first, like OCaml! This is also useful when interpreting function application, because we evaluate the argument first.

$$[\![\frac{}{\Gamma \vdash () : \mathsf{unit}}]\!] := !_\Gamma ; \eta_1 \qquad [\![\frac{}{\Gamma \vdash s : \mathsf{str}}]\!] := !_\Gamma ; \ulcorner s \urcorner ; \eta_{\Sigma^*}$$

$$[\![\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1 , e_2) : A \times B}]\!] := \begin{array}{ll} let & \begin{cases} f & := \ [\![\Gamma \vdash e_1 : A]\!] \\ g & := \ [\![\Gamma \vdash e_2 : B]\!] \end{cases} \\ in & \langle f , g \rangle ; \beta_{A,B} \end{array}$$

$$[\![\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{fst}\, e : A}]\!] := [\![\Gamma \vdash e : A \times B]\!] ; T\pi_1 \qquad [\![\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{snd}\, e : B}]\!] := [\![\Gamma \vdash e : A \times B]\!] ; T\pi_2$$

$$[\![\frac{x : A^q \in \Gamma}{\Gamma \vdash x : A}]\!] := [\![x : A^q \in \Gamma]\!] ; \eta_A$$

$$[\![\frac{\Gamma, x : A^{\mathbf{i}} \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \Rightarrow B}]\!] := \mathsf{curry}\,([\![\Gamma, x : A^{\mathbf{i}} \vdash e : B]\!]) ; \eta_{A \to TB}$$

$$[\![\frac{\Gamma \vdash e_1 : A \Rightarrow B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}]\!] := \begin{array}{ll} let & \begin{cases} f & := \ [\![\Gamma \vdash e_1 : A \Rightarrow B]\!] \\ g & := \ [\![\Gamma \vdash e_2 : A]\!] \end{cases} \\ in & \langle f , g \rangle ; \beta_{A \to TB, A} ; T\,\mathsf{ev}_{A,TB} ; \mu_B \end{array}$$

$$[\![\frac{\Gamma \vdash e_1 : \mathsf{cap} \qquad \Gamma \vdash e_2 : \mathsf{str}}{\Gamma \vdash e_1.\mathsf{print}(e_2) : \mathsf{unit}}]\!] := \begin{array}{ll} let & \left\{ \begin{array}{lll} f & := & [\![\Gamma \vdash e_1 : \mathsf{cap}]\!] \\ g & := & [\![\Gamma \vdash e_2 : \mathsf{str}]\!] \\ p & : & \mathcal{C} \times \Sigma^* \to T1 \\ (c, s) & \mapsto & \left(*\ ,\ \lambda c'. \begin{cases} s & \text{if } c = c' \\ \varepsilon & \text{otherwise} \end{cases} \right) \end{array} \right. \\ in & \langle f , g \rangle ; \beta_{\mathcal{C}, \Sigma^*} ; Tp ; \mu_1 \end{array}$$

$$[\![\frac{\Gamma \vdash^{\mathbf{s}} e : A}{\Gamma \vdash \mathsf{box}\, \boxed{e} : \square A}]\!] := [\![\Gamma \vdash^{\mathbf{s}} e : A]\!]_p ; \eta_{\square A}$$

$$[\![\frac{\Gamma^{\mathbf{s}} \vdash e : A}{\Gamma \vdash^{\mathbf{s}} e : A}]\!]_p := \rho(\Gamma) ; \mathcal{M}(\Gamma) ; \square [\![\Gamma^{\mathbf{s}} \vdash e : A]\!] ; \Phi_A$$

$$[\![\frac{\Gamma \vdash e_1 : \square A \qquad \Gamma, x : A^{\mathbf{s}} \vdash e_2 : B}{\Gamma \vdash \mathsf{let\ box}\, \boxed{x} = e_1 \,\mathsf{in}\, e_2 : B}]\!] := \begin{array}{ll} let & \begin{cases} f & := \ [\![\Gamma \vdash e_1 : \square A]\!] \\ g & := \ [\![\Gamma, x : A^{\mathbf{s}} \vdash e_2 : B]\!] \end{cases} \\ in & \langle id_\Gamma , f \rangle ; \tau_{\Gamma, \square A} ; Tg ; \mu_B \end{array}$$

Fig. 9. Interpretation of expressions, $[\![\Gamma \vdash e : A]\!] : \mathcal{H}om_{\mathcal{C}}([\![\Gamma]\!], T[\![A]\!])$, $[\![\Gamma \vdash^{\mathbf{s}} e : A]\!]_p : \mathcal{H}om_{\mathcal{C}}([\![\Gamma]\!], \square[\![A]\!])$

a channel c' – if it equals c, we record the string s, otherwise we just choose the empty string ε. We interpret the arguments of print and apply them to p to evaluate it. [8] The rest of the interpretation is similar to the one for $\Rightarrow$ E, with output type 1.

We used a different interpretation function for *safe* expressions, which we define below.

We need to interpret the *purify* operation **s** on contexts, for which we define the map $\rho(\Gamma)$ in figure 10a. We also need another combinator $\mathcal{M}(\Gamma)$, defined in figure 10b, which uses the monoidal action and the idempotence of the comonad $\square$ to distribute the $\square$ over the products in Γ. Note that $\mathcal{M}(\Gamma)$ is an isomorphism because m and δ are.

[8] $\ulcorner s \urcorner : \mathcal{H}om_{\mathcal{C}}(1 , \Sigma^*)$ is the global element that picks the literal s in Σ^*.

$$\rho(\cdot) := id_1 \qquad \mathcal{M}(\cdot) := id_1$$
$$\rho(\Gamma, x : A^{\mathbf{s}}) := [\rho(\Gamma) \times id_{\Box A}] \qquad \mathcal{M}(\Gamma, x : A^{\mathbf{s}}) := [\mathcal{M}(\Gamma) \times \delta_A] \fatsemi m^{\times}_{\Gamma^{\mathbf{s}}, \Box A}$$
$$\rho(\Gamma, x : A^{\mathbf{i}}) := \pi_1 \fatsemi \rho(\Gamma) \qquad \mathcal{M}(\Gamma, x : A^{\mathbf{i}}) := \mathcal{M}(\Gamma)$$

(a) $\rho(\Gamma) : \mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \llbracket \Gamma^{\mathbf{s}} \rrbracket)$ (b) $\mathcal{M}(\Gamma) : \mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma^{\mathbf{s}} \rrbracket, \Box \llbracket \Gamma^{\mathbf{s}} \rrbracket)$

Fig. 10. $\rho(\Gamma)$ and $\mathcal{M}(\Gamma)$

$$\llbracket \frac{}{\cdot \supseteq \cdot} \rrbracket := id_1$$
$$\llbracket \frac{}{x : A^{\mathbf{i}} \in (\Gamma, x : A^{\mathbf{i}})} \rrbracket := \pi_2 \qquad \llbracket \frac{\Gamma \supseteq \Delta}{\Gamma, x : A^{q} \supseteq \Delta} \rrbracket := \pi_1 \fatsemi \llbracket \Gamma \supseteq \Delta \rrbracket$$
$$\llbracket \frac{}{x : A^{\mathbf{s}} \in (\Gamma, x : A^{\mathbf{s}})} \rrbracket := \pi_2 \fatsemi \varepsilon_A \qquad \llbracket \frac{\Gamma \supseteq \Delta}{\Gamma, x : A^{\mathbf{s}} \supseteq \Delta, x : A^{\mathbf{s}}} \rrbracket := [\llbracket \Gamma \supseteq \Delta \rrbracket \times id_{\Box A}]$$
$$\llbracket \frac{x : A^{q} \in \Gamma \qquad (x \neq y)}{x : A^{q} \in (\Gamma, y : B^{r})} \rrbracket := \pi_1 \fatsemi \llbracket x : A^{q} \in \Gamma \rrbracket \qquad \llbracket \frac{\Gamma \supseteq \Delta}{\Gamma, x : A^{\mathbf{i}} \supseteq \Delta, x : A^{\mathbf{i}}} \rrbracket := [\llbracket \Gamma \supseteq \Delta \rrbracket \times id_A]$$

(a) $\llbracket x : A^{q} \in \Gamma \rrbracket : \mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$ (b) $\mathsf{Wk}(\Gamma \supseteq \Delta) := \llbracket \Gamma \supseteq \Delta \rrbracket : \mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \llbracket \Delta \rrbracket)$

Fig. 11. Interpretation of Membership and Weakening

Now, the interpretation function for *safe* expressions $\Gamma \vdash^{\mathbf{s}} e : A$ uses the CTX-SAFE rule, and is defined as a morphism in $\mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \Box \llbracket A \rrbracket)$. We *purify* the context to a *safe* one, so that we can evaluate the expression. However, we need a value in $\Box A$, but the expression interpretation would produce something in TA. Now, we can only cancel the monad under the comonad, so we use the $\mathcal{M}(\Gamma)$ map which uses the idempotence of $\Box$ to do a readjustment. We can now evaluate the expression under the $\Box$ in the *safe* context, which gives a monadic value of type TA under the comonad $\Box$. We can finally use Φ to cancel the monad T under the $\Box$.

5.3 Weakening and Substitution

We now give semantics for the syntactic weakening and substitution operations.

5.3.1 Weakening. For contexts Γ and Δ, we interpret the weakening judgement $\Gamma \supseteq \Delta$ as a morphism in $\mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \llbracket \Delta \rrbracket)$, as shown in figure 11b. We also refer to it as the weakening map $\mathsf{Wk}(\Gamma \supseteq \Delta)$. We prove a semantic weakening lemma, analogous to the syntactic weakening lemma 3.1.

LEMMA 5.1 SEMANTIC WEAKENING. *If* $\Gamma \supseteq \Delta$ *and* $\Delta \vdash e : A$, *then*
$$\llbracket \Gamma \vdash e : A \rrbracket = \mathsf{Wk}(\Gamma \supseteq \Delta) \fatsemi \llbracket \Delta \vdash e : A \rrbracket.$$

5.3.2 Substitution. We now interpret a substitution $\Gamma \vdash \theta : \Delta$ as a morphism in $\mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, \llbracket \Delta \rrbracket)$, as shown in figure 12b. However, this is not a trivial iteration of the expression interpretation. The reason is that the interpretation of contexts in figure 8b interprets a variable $x : A^{\mathbf{i}}$ in the context as an element of the type $\llbracket A \rrbracket$, and a variable $x : A^{\mathbf{s}}$ as an element of the type $\Box \llbracket A \rrbracket$. However, an expression $\Gamma \vdash e : A$ will be interpreted as a morphism in $\mathcal{H}om_{\mathcal{C}}(\llbracket \Gamma \rrbracket, T \llbracket A \rrbracket)$. Operationally, we resolve this mismatch by only substituting *values* for variables in call-by-value languages, and

$$[\![\frac{}{\Gamma \vdash () : \mathsf{unit}}]\!]_v := !_\Gamma$$

$$[\![\frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash (v_1, v_2) : A \times B}]\!]_v := \langle [\![\Gamma \vdash v_1 : A]\!]_v , [\![\Gamma \vdash v_2 : B]\!]_v \rangle$$

$$[\![\frac{x : A^q \in \Gamma}{\Gamma \vdash x : A}]\!]_v := [\![x : A^q \in \Gamma]\!]$$

$$[\![\frac{\Gamma, x : A^{\mathbf{i}} \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \Rightarrow B}]\!]_v := \mathsf{curry}\,([\![\Gamma, x : A^{\mathbf{i}} \vdash e : B]\!])$$

$$[\![\frac{\Gamma \vdash^{\mathbf{s}} e : A}{\Gamma \vdash \mathsf{box}\,\boxed{e} : \Box A}]\!]_v := [\![\Gamma \vdash^{\mathbf{s}} e : A]\!]_p$$

(a) $[\![\Gamma \vdash v : A]\!]_v : \mathcal{H}om_{\mathcal{C}}\,([\![\Gamma]\!] , [\![A]\!])$

$$[\![\frac{}{\Gamma \vdash \langle\rangle : \cdot}]\!] := !_\Gamma$$

$$[\![\frac{\Gamma \vdash \theta : \Delta \qquad \Gamma \vdash^{\mathbf{s}} e : A}{\Gamma \vdash \langle \theta, e^{\mathbf{s}}/x \rangle : \Delta, x : A^{\mathbf{s}}}]\!] := \langle [\![\Gamma \vdash \theta : \Delta]\!] , [\![\Gamma \vdash^{\mathbf{s}} e : A]\!]_p \rangle$$

$$[\![\frac{\Gamma \vdash \theta : \Delta \qquad \Gamma \vdash v : A}{\Gamma \vdash \langle \theta, v^{\mathbf{i}}/x \rangle : \Delta, x : A^{\mathbf{i}}}]\!] := \langle [\![\Gamma \vdash \theta : \Delta]\!] , [\![\Gamma \vdash v : A]\!]_v \rangle$$

(b) $[\![\Gamma \vdash \theta : \Delta]\!] : \mathcal{H}om_{\mathcal{C}}\,([\![\Gamma]\!] , [\![\Delta]\!])$

Fig. 12. Interpretation of values and substitution

indeed, our definition of substitutions in figure 5c restricts the definition of substitution to range over values in the rule SUB-IMPURE.

Therefore, we mimic this syntactic restriction in the semantics, by giving a separate interpretation only for values, interpreting the judgement $\Gamma \vdash v : A$ as a morphism in $\mathcal{H}om_{\mathcal{C}}\,([\![\Gamma]\!] , [\![A]\!])$, in figure 12a. Note in particular that the value interpretation yields an element of $[\![A]\!]$, as the context interpretation requires, rather than an element of $T[\![A]\!]$. This value interpretation makes use of the expression interpretation in the interpretation of λ-expressions, but the expression relation does not directly refer to the value interpretation. There are alternative presentations such as fine-grain call-by-value [Levy et al. 2003], which have a separate syntactic class of values and value judgements, and hence make the value and expression interpretations mutually recursive. However, we choose not to do that in order to remain close to the usual presentation.

Note that box $\boxed{e}$ expressions are also values, and our *safe* interpretation does the right thing for box values, since the interpretation of $\Box A$ uses the comonad, $\Box[\![A]\!]$. With the interpretation of values in hand, we can define the substitution interpretation as follows.

We use the *safe* expression interpretation to interpret the SUB-SAFE rule, and the *impure* value interpretation for the SUB-IMPURE rule.

Finally, we prove the semantic analogue of the syntactic substitution theorem 3.4. We prove two auxiliary lemmas 5.2 and 5.3, characterising the expression interpretation of *safe expressions* and

$$
\begin{array}{rcl}
\mathcal{C} & ::= & [\cdot] \mid e\ \mathcal{C} \mid \mathcal{C}\ e \mid \lambda x : A.\ \mathcal{C} \\
& \mid & \mathsf{fst}\ \mathcal{C} \mid \mathsf{snd}\ \mathcal{C} \mid (e\ ,\ \mathcal{C}) \mid (\mathcal{C}\ ,\ e) \\
& \mid & \mathsf{box}\ \boxed{\mathcal{C}} \mid \mathsf{let\ box}\ \boxed{x} = \mathcal{C}\ \mathsf{in}\ e \mid \mathsf{let\ box}\ \boxed{x} = e\ \mathsf{in}\ \mathcal{C} \\
\mathcal{E} & ::= & [\cdot] \mid e\ \mathcal{E} \mid \mathcal{E}\ v \\
& \mid & \mathsf{fst}\ \mathcal{E} \mid \mathsf{snd}\ \mathcal{E} \mid (e\ ,\ \mathcal{E}) \mid (\mathcal{E}\ ,\ v) \\
& \mid & \mathsf{let\ box}\ \boxed{x} = \mathcal{E}\ \mathsf{in}\ e \mid \mathsf{let\ box}\ \boxed{x} = v\ \mathsf{in}\ \mathcal{E}
\end{array}
$$

Fig. 13. Grammar extended with Evaluation Contexts

impure values. The lemmas show that the interpretation for each ends in a trivial lifting into the monad T using η. This makes the proof of the semantic substitution theorem 5.4 possible.

Lemma 5.2 Safe interpretation. *If* $\Gamma \vdash^{\mathbf{s}} e : A$, *then*

$$[\![\Gamma \vdash e : A]\!] = [\![\Gamma \vdash^{\mathbf{s}} e : A]\!]_p \ ; \varepsilon_A \ ; \eta_A.$$

Lemma 5.3 Value interpretation. *If* $\Gamma \vdash v : A$, *then*

$$[\![\Gamma \vdash v : A]\!] = [\![\Gamma \vdash v : A]\!]_v \ ; \eta_A.$$

Theorem 5.4 Semantic substitution. *If* $\Gamma \vdash \theta : \Delta$ *and* $\Delta \vdash e : A$, *then*

$$[\![\Gamma \vdash \theta(e) : A]\!] = [\![\Gamma \vdash \theta : \Delta]\!] \ ; [\![\Delta \vdash e : A]\!].$$

6 EQUATIONAL THEORY

We have an extension of the call-by-value simply-typed lambda calculus, so we want the usual $\beta\eta$-equations to hold in our theory. However, we also added new expression forms for the $\square$ type. We want computation and extensionality rules for the box form and the let box binding form. To handle the commuting conversions [Girard et al. 1989], we use evaluation contexts.

We extend our grammar with two kinds of evaluation contexts — a *safe* evaluation context $\mathcal{C}$, and an *impure* evaluation context $\mathcal{E}$, as shown in figure 13. The intuition is that $\mathcal{E}$ allows safe reductions for *impure* expressions, i.e., it picks out the contexts consistent with the evaluation order of the call-by-value simply-typed lambda calculus. The *safe* evaluation context $\mathcal{C}$ allows redexes in every sub-expression; but it is restricted only to *safe* expressions. The hole $[\cdot]$ is the empty evaluation context. We use the notation $\mathcal{C}\langle\langle e \rangle\rangle$ or $\mathcal{E}\langle\langle e \rangle\rangle$ to indicate that we're replacing the hole in the respective evaluation context with e.

We define a judgement form for equality of terms, as shown in figure 2c, and state the rules for the equational theory in figures 14 and 15. We have the usual REFL, SYM, and TRANS rules which give the reflexive, symmetric, and transitive closure, so that the equality relation is an equivalence, and the CONG rules for each term former, which make the relation a congruence closure.

We have the computation rules $\times_1\beta$ and $\times_2\beta$ for pairs; we only allow values for these rules. The $\times\eta$ rule is the extensionality rule for pairs, but again, restricted to values.

The $\Rightarrow\ \beta$ rule is the usual call-by-value computation rule for an application of a λ-expression to an argument. [9] Since the calculus has effects, we only allow the operand to be a value. For example, consider the function $f := \lambda x : \mathsf{unit}.\ x\ ;\ x$. We can safely β-reduce $f\ ()$ to $()\ ;\ ()$, but allowing a β-reduction for $f\ (c.\mathsf{print}(s))$ would duplicate the effect!

We add η rules for functions, but we need to be careful because we have effects. For example, consider the expression $f := c.\mathsf{print}(s)\ ;\ \lambda x.\ x$. On η-expansion, we get $g := \lambda y.\ f\ y$, but now the

[9] The notation $[v/x]e$ is shorthand for $\langle\langle \Gamma \rangle, v^{\mathbf{i}}/x\rangle(e)$ where $\langle \Gamma \rangle$ is the identity substitution $\Gamma \vdash \langle \Gamma \rangle : \Gamma$.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e \approx e : A}\ \text{REFL} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A}{\Gamma \vdash e_2 \approx e_1 : A}\ \text{SYM} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A \qquad \Gamma \vdash e_2 \approx e_3 : A}{\Gamma \vdash e_1 \approx e_3 : A}\ \text{TRANS}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A \times B}{\Gamma \vdash \mathsf{fst}\, e_1 \approx \mathsf{fst}\, e_2 : A}\ \text{fst-CONG} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A \times B}{\Gamma \vdash \mathsf{snd}\, e_1 \approx \mathsf{snd}\, e_2 : B}\ \text{snd-CONG}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A \qquad \Gamma \vdash e_3 \approx e_4 : B}{\Gamma \vdash (e_1 , e_3) \approx (e_2 , e_4) : A \times B}\ \text{PAIR-CONG} \qquad \frac{\Gamma, x : A^{\mathbf{i}} \vdash e_1 \approx e_2 : B}{\Gamma \vdash \lambda x : A.\, e_1 \approx \lambda x : A.\, e_2 : A \Rightarrow B}\ \lambda\text{-CONG}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A \Rightarrow B \qquad \Gamma \vdash e_3 \approx e_4 : A}{\Gamma \vdash e_1\, e_3 \approx e_2\, e_4 : B}\ \text{APP-CONG} \qquad \frac{\Gamma^{\mathbf{s}} \vdash e_1 \approx e_2 : A}{\Gamma \vdash \mathsf{box}\, \boxed{e_1} \approx \mathsf{box}\, \boxed{e_2} : \square A}\ \text{box-CONG}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : \square A \qquad \Gamma, x : A^{\mathbf{s}} \vdash e_3 \approx e_4 : B}{\Gamma \vdash (\mathsf{let\ box}\, \boxed{x} = e_1\, \mathsf{in}\, e_3) \approx (\mathsf{let\ box}\, \boxed{x} = e_2\, \mathsf{in}\, e_4) : B}\ \text{let box-CONG}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : \mathsf{cap} \qquad \Gamma \vdash e_3 \approx e_4 : \mathsf{str}}{\Gamma \vdash e_1.\mathsf{print}(e_3) \approx e_2.\mathsf{print}(e_4) : \mathsf{unit}}\ \text{print-CONG}$$

Fig. 14. Equivalence and Congruence rules for the Equational Theory

print operation is suspended in the closure, and doesn't evaluate when we apply g. Hence, we add two forms of η rules for functions — the $\Rightarrow$ η-IMPURE rule only allows η-expansion for values, and the $\Rightarrow$ η-SAFE rule allows η-expansion also for expressions that are *safe*.

The computation rule $\square\beta$ for the $\square$ type allows computation under the let box binder. If we bind a box-ed expression under the let box binder, we can substitute the underlying expression in the motive. This is safe because e_1 is forced to be a *safe* expression.

Finally, we have the η expansion rules for the $\square$ type, which pushes an expression in an evaluation context under a let box binder. The $\square\eta$–*safe* rule uses the *safe* evaluation context $\mathcal{C}$, while the $\square\eta$–*impure* rule uses the *impure* evaluation context $\mathcal{E}$. The only difference in the rules is that the $\mathcal{C}$ evaluation context can be plugged with *safe* expressions only.

We prove that our equality rules are sound with respect to our categorical semantics. If two expressions are equal in the equational theory, they have equal interpretations in the semantics.

THEOREM 6.1 SOUNDNESS OF $\approx$. *If* $\Gamma \vdash e_1 \approx e_2 : A$, *then* $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

7 EMBEDDING

Our language is an extension of the call-by-value simply-typed lambda calculus. But how could we claim that it is really an *extension*? In this section, we show that we can *embed* the simply-typed lambda calculus into our calculus, in an equation preserving way. We state the full simply-typed lambda calculus including its $\beta\eta$-equational theory in figure 16.

We give the grammar and judgements in figures 16a and 16b, typing rules in figure 16c, and the $\beta\eta$-equational theory in figure 16d. Note that we choose to use the base type unit, and we leave out products because their embedding is trivial and uninteresting for our purpose.

$$\frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash \mathsf{fst}\,(v_1 , v_2) \approx v_1 : A}\ \times_1\beta \qquad \frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash \mathsf{snd}\,(v_1 , v_2) \approx v_2 : B}\ \times_2\beta$$

$$\frac{\Gamma \vdash v : A \times B}{\Gamma \vdash v \approx (\mathsf{fst}\, v\,, \mathsf{snd}\, v) : A \times B}\ \times\eta$$

$$\frac{\Gamma, x : A^{\mathbf{i}} \vdash e : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x : A.\ e)\, v \approx [v/x]e : B}\ \Rightarrow \beta$$

$$\frac{\Gamma \vdash v : A \Rightarrow B}{\Gamma \vdash v \approx \lambda x : A.\ v\, x : A \Rightarrow B}\ \Rightarrow \eta\text{-IMPURE} \qquad \frac{\Gamma \vdash^{\mathbf{s}} e : A \Rightarrow B}{\Gamma \vdash e \approx \lambda x : A.\ e\, x : A \Rightarrow B}\ \Rightarrow \eta\text{-SAFE}$$

$$\frac{\Gamma^{\mathbf{s}} \vdash e_1 : A \qquad \Gamma, x : A^{\mathbf{s}} \vdash e_2 : B}{\Gamma \vdash \mathsf{let\ box}\, \boxed{x} = \mathsf{box}\, \boxed{e_1}\ \mathsf{in}\ e_2 \approx [e_1/x]e_2 : B}\ \Box\beta$$

$$\frac{\Gamma \vdash^{\mathbf{s}} e : \Box A \qquad \Gamma \vdash \mathcal{C}\langle\!\langle e \rangle\!\rangle : B \qquad \Gamma \vdash \mathsf{let\ box}\, \boxed{x} = e\ \mathsf{in}\ \mathcal{C}\langle\!\langle \mathsf{box}\, \boxed{x} \rangle\!\rangle : B}{\Gamma \vdash \mathcal{C}\langle\!\langle e \rangle\!\rangle \approx \mathsf{let\ box}\, \boxed{x} = e\ \mathsf{in}\ \mathcal{C}\langle\!\langle \mathsf{box}\, \boxed{x} \rangle\!\rangle : B}\ \Box\eta\text{-SAFE}$$

$$\frac{\Gamma \vdash e : \Box A \qquad \Gamma \vdash \mathcal{E}\langle\!\langle e \rangle\!\rangle : B \qquad \Gamma \vdash \mathsf{let\ box}\, \boxed{x} = e\ \mathsf{in}\ \mathcal{E}\langle\!\langle \mathsf{box}\, \boxed{x} \rangle\!\rangle : B}{\Gamma \vdash \mathcal{E}\langle\!\langle e \rangle\!\rangle \approx \mathsf{let\ box}\, \boxed{x} = e\ \mathsf{in}\ \mathcal{E}\langle\!\langle \mathsf{box}\, \boxed{x} \rangle\!\rangle : B}\ \Box\eta\text{-IMPURE}$$

Fig. 15. Equational Theory

We define an embedding function from the simply-typed lambda calculus to our calculus. We use the notation $\underset{\frown}{X}$ to denote the embedding of a syntactic object X from STLC into our calculus. We give the syntactic translation of types, contexts, and raw terms in figure 17.

To embed the function type, we embed the domain and codomain, but we apply our comonadic type constructor $\Box$ to restrict the domain to a *safe* type. This embedding is quite like the Gödel-McKinsey-Tarski embedding of the intuitionistic propositional calculus into classical S4 modal logic, as outlined in [McKinsey and Tarski 1948], but we do not need to apply the $\Box$ type constructor on the codomain, because our functions are *capability-safe*. We remark that this is similar to the embedding of lax logic into S4 modal logic described in [Pfenning and Davies 2001], as well as the embedding of intuitionistic logic into linear logic [Girard 1987].

When embedding contexts, we mark the variables as *safe* using the $\mathbf{s}$ annotation. To embed functions and applications, we need to use the introduction and elimination forms for $\Box$. When embedding a λ-expression, the bound variable is embedded as a term of $\Box$ type, so we eliminate the underlying variable using the let box binding form before using it in the body. To embed an application, we simply put the argument in a box.

We show that this translation preserves typing, i.e., well-typed expressions embed to well-typed expressions. Then, we show that the $\beta\eta$-equational theory of the *pure* call-by-value simply-typed lambda calculus is preserved under the translation. If two expressions are equal in the simply-typed lambda calculus, they *remain equal* after embedding into our imperative calculus.

$$
\begin{array}{lrcl}
\textsc{Types} & A, B & ::= & \mathsf{unit} \mid A \Rightarrow B \\
\textsc{Terms} & e & ::= & () \mid x \mid \lambda x : A.\, e \mid e_1\, e_2 \\
\textsc{Values} & v & ::= & () \mid x \mid \lambda x : A.\, e \\
\textsc{Contexts} & \Gamma, \Delta, \Psi & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

(a) Grammar for STLC

$$
\begin{array}{rl}
x : A \in \Gamma & x \text{ is a variable of type } A \text{ in context } \Gamma \\
\Gamma \vdash_\lambda e : A & e \text{ is an expression of type } A \text{ in context } \Gamma \\
\Gamma \vdash_\lambda e_1 \approx e_2 : A & e_1 \text{ and } e_2 \text{ are equal expressions of type } A \text{ in context } \Gamma
\end{array}
$$

(b) Judgements for STLC

$$
\frac{}{\Gamma \vdash_\lambda () : \mathsf{unit}}\ \mathsf{unitI}
\qquad
\frac{x : A \in \Gamma}{\Gamma \vdash_\lambda x : A}\ \textsc{Var}
$$

$$
\frac{\Gamma, x : A \vdash_\lambda e : B}{\Gamma \vdash_\lambda \lambda x : A.\, e : A \Rightarrow B}\ \Rightarrow\!\mathrm{I}
\qquad
\frac{\Gamma \vdash_\lambda e_1 : A \Rightarrow B \qquad \Gamma \vdash_\lambda e_2 : A}{\Gamma \vdash_\lambda e_1\, e_2 : B}\ \Rightarrow\!\mathrm{E}
$$

(c) Typing rules for STLC

$$
\frac{\Gamma \vdash_\lambda e : A}{\Gamma \vdash_\lambda e \approx e : A}\ \textsc{refl}
\qquad
\frac{\Gamma \vdash_\lambda e_1 \approx e_2 : A}{\Gamma \vdash_\lambda e_2 \approx e_1 : A}\ \textsc{sym}
$$

$$
\frac{\Gamma \vdash_\lambda e_1 \approx e_2 : A \qquad \Gamma \vdash_\lambda e_2 \approx e_3 : A}{\Gamma \vdash_\lambda e_1 \approx e_3 : A}\ \textsc{trans}
\qquad
\frac{\Gamma, x : A \vdash_\lambda e_1 \approx e_2 : B}{\Gamma \vdash_\lambda \lambda x : A.\, e_1 \approx \lambda x : A.\, e_2 : A \Rightarrow B}\ \lambda\text{-}\textsc{cong}
$$

$$
\frac{\Gamma \vdash_\lambda e_1 \approx e_2 : A \Rightarrow B \qquad \Gamma \vdash_\lambda e_3 \approx e_4 : A}{\Gamma \vdash_\lambda e_1\, e_3 \approx e_2\, e_4 : B}\ \textsc{app-cong}
$$

$$
\frac{\Gamma, x : A \vdash_\lambda e_1 : B \qquad \Gamma \vdash_\lambda e_2 : A}{\Gamma \vdash_\lambda (\lambda x : A.\, e_1)\, e_2 \approx [e_2/x]e_1 : B}\ \Rightarrow\beta
\qquad
\frac{\Gamma \vdash_\lambda e : A \Rightarrow B}{\Gamma \vdash_\lambda e \approx \lambda x : A.\, e\, x : A \Rightarrow B}\ \Rightarrow\eta
$$

(d) Equational Theory for STLC

Fig. 16. The *pure* call-by-value simply-typed lambda calculus

$$
\begin{array}{lrcl}
\textsc{Types} & \mathsf{unit} & := & \mathsf{unit} \\
& A \Rightarrow B & := & \Box A \Rightarrow B \\
\textsc{Contexts} & \cdot & := & \cdot \\
& \Gamma, x : A & := & \Gamma, x : A^{\mathbf{s}}
\end{array}
\qquad
\begin{array}{lrcl}
\textsc{Terms} & () & := & () \\
& x & := & x \\
& \lambda x : A.\, e & := & \lambda z : \Box A.\, \mathsf{let\ box}\ x = z\ \mathsf{in}\ e \\
& e_1\, e_2 & := & e_1\ \mathsf{box}\ e_2
\end{array}
$$

Fig. 17. Embedding STLC

Theorem 7.1 Preservation of typing. *If* $\Gamma \vdash_\lambda e : A$, *then* $\Gamma \vdash e : A$.

Theorem 7.2 Preservation of equality. *If* $\Gamma \vdash_\lambda e_1 \approx e_2 : A$, *then* $\Gamma \vdash e_1 \approx e_2 : A$.

Finally, we show that our imperative calculus is a conservative extension of the simply-typed lambda calculus. To do so, we claim that if two embedded terms are equal in the extended theory, then they must have been equal in the smaller theory. This shows that the equational theory of the imperative calculus does not introduce any extra equations that would destroy the computational properties of the *pure* simply-typed lambda calculus.

Theorem 7.3 Conservative Extension. *If* $\Gamma \vdash_\lambda e_1 : A, \Gamma \vdash_\lambda e_2 : A$, *and* $\Gamma \vdash e_1 \approx e_2 : A$, *then* $\Gamma \vdash_\lambda e_1 \approx e_2 : A$.

8 DISCUSSION AND FUTURE WORK

There has been a vast amount of work on integrating effects into purely functional languages. Ironically though, even the very definition of what a purely functional language is has historically been a contested one. Sabry [1998] proposed that a functional language is pure when its behaviour under different evaluation strategies is "morally" the same, in the sense of Danielsson et al. [2006]. That is, if changing the evaluation strategy from call-by-value to (say) call-by-need could only change the divergence/error behaviour of programs in a language, then the language is pure. In contrast, the definition we use in this paper is less sophisticated: we take purity to be the preservation of the $\beta\eta$ equational theory of the simply-typed lambda calculus. However, it lets us prove the correctness of our embedding in an appealingly simple way, by translating derivations of equality.

The use of substructural type systems to control access to mutable data is also a long-running theme in the development of programming languages. It is so long-running, in fact, that it actually predates linear logic [Girard 1987] by nearly a decade! Reynolds' Syntactic Control of Interference [Reynolds 1978] proposed using a substructural type discipline to prevent aliased access to data structures. The intuition that substructural logic corresponds to ownership of capabilities is also a very old one – O'Hearn [1993] uses it to explain his model of SCI, and Crary et al. [1999] compare their static capabilities to the capabilities in the HYDRA system of Wulf et al. [1974].

However, these comparisons remained informal, due to the fact that semanticists tended to use capabilities in a substructural fashion (e.g., see [Crary et al. 1999; Terauchi and Aiken 2006]), but from the very outset ([Dennis and Horn 1966]) to modern day applications like capability-safe Javascript [Maffeis et al. 2010], systems designers have tended to use capabilities *non-linearly*. In particular, they thought it was desirable for a principal to hand a capability to two different deputies, which is a design principle obviously incompatible with linearity.

The idea that the linear implication and intuitionistic implication could coexist, without one reducing to the other, first arose in the logic of bunched implications [O'Hearn and Pym 1999]. This led to separation logic [Reynolds 2002], which has been very successful at verifying programs with aliasable state. However, even though the semantics of separation logic supports BI, the bulk of the tooling infrastructure for separation logic (such as Smallfoot [Berdine et al. 2006]) have focused on the substructural fragment, often even omitting anything not in the linear fragment.

However, one observation very important to our work did arise from work on separation logic. Dodds et al. [2009] made the critical observation that in addition to being able to assert ownership, it is extremely useful to be able to *deny* the ownership of a capability. Basically, knowing that a client program *lacks* any capabilities can make it safe to invoke it in a secure context.

The comonadic structure behind denial was also known informally: it arises in the work of Morrisett et al. [2005], where the exponential comonad in linear logic is modelled as the *lack* of any heap ownership; and in an intuitionistic context, the work on functional reactive programming [Krishnaswami 2013] used a capability to create temporal values, and a comonad denying ownership of it permitted writing space-leak-free reactive programs. However, both of these papers used operational unary logical relations models, and so did not prove anything about the equational theory.

Equational theories are easier to get with denotational models, and our model derives from the work of Hofmann [2003]. In his work, he developed a denotational model of space-bounded computation, by taking a naive set-theoretic semantics, and then augmenting it with intensional information. His sets were augmented with a *length function* saying how much memory each value used, and in ours, we use a weight function saying how many capabilities each value holds. (In fact, he even notes that his category also forms a model of bunched implications!) We think his approach has a high power-to-weight ratio, and hope we have shown that it has broad applicability as well.

However, this semantics is certainly not the last word: e.g., the semantics in this paper does not model the allocation of new capabilities as a program executes. In the categorical semantics of bunched logics, it is common to use functor categories, such as functors from the *category of finite sets and injections* $\mathcal{I}$, to Set, or presheaves over some other monoidal category. The functor category forms a model of BI, inheriting the cartesian closed structure where the limits are computed Kripke-style in Set, and also a monoidal closed structure using the tensor product from the monoidal category and *Day convolution*. In addition, the ability to move to a bigger set permits modelling allocation of new names and channels (e.g., as is done in models of the ν-calculus [Stark 1996]).

Another natural question is how we might handle recursion, as our explicit description of the category of capability spaces $\mathcal{C}$ in section 4 seems quite tied to Set. By replaying this in a category like CPO rather than Set, we may be able to derive a domain-theoretic analogue of capability spaces.

Another direction for future work lies in the observation that our $\Box$ comonad in subsection 4.5 takes away *all* capabilities, yielding a system with a syntax like that of Pfenning and Davies [2001] with an interpretation close to the axiomatic categorical semantics proposed by Alechina et al. [2001] and Kobayashi [1997]. However, we could consider a *graded* or *indexed* version of the same, i.e., $\Box_C$, which only takes away a set of capabilities $C \in \mathfrak{P}(\mathcal{C})$ from a value. Our hope would be that this could form a model of systems like bounded linear logic [Dal Lago and Hofmann 2009; Orchard et al. 2019], or other systems of coeffects [Petricek et al. 2014]. One issue we foresee is that while this indexed comonad would still be a strong monoidal functor, it loses the idempotence property, which we used in our interpretation and proofs.

There has also been a great deal of work on using monads and effect systems [Gifford and Lucassen 1986; Moggi 1989; Nielson and Nielson 1999; Wadler 1998] to control the usage of effects. However, the general idea of using a static tag which broadcasts that an effect *may* occur seems somewhat the reverse of the idea of object capabilities, where access to a dynamically-passed value determines whether an effect can occur. The key feature of our system is that the comonad does not say what effects are possible, but rather asserts that effects are *absent*. This manifests in the cancellation law (in subsection 4.6) of the comonad and the monad. Still, the very phrases *"may perform"* and *"does not possess"* hint that some sort of duality ought to exist.

REFERENCES

Natasha Alechina, Michael Mendler, Valeria de Paiva, and Eike Ritter. 2001. Categorical and Kripke Semantics for Constructive S4 Modal Logic. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 292–307. https://doi.org/10.1007/3-540-44802-0_21

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf,

and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–137.

Karl Crary, David Walker, and J. Gregory Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 262–275. https://doi.org/10.1145/292540.292564

Ugo Dal Lago and Martin Hofmann. 2009. Bounded Linear Logic, Revisited. In *Typed Lambda Calculi and Applications*, Pierre-Louis Curien (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94.

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and Loose Reasoning is Morally Correct. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, 206–217. https://doi.org/10.1145/1111037.1111056 Charleston, South Carolina, USA.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. https://doi.org/10.1145/365230.365252

Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 363–377.

Jeremy Gibbons. 2000. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures (Lecture Notes in Computer Science)*, Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons (Eds.), Vol. 2297. Springer, 149–202. https://doi.org/10.1007/3-540-47797-7_5

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (Jan 1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press, New York, NY, USA. 217–241 pages. https://doi.org/10.1007/978-1-4612-2822-6_8

Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Information and Computation* 183, 1 (may 2003), 57–85. https://doi.org/10.1016/s0890-5401(03)00009-9

Satoshi Kobayashi. 1997. Monad as modality. *Theoretical Computer Science* 175, 1 (1997), 29 – 74. https://doi.org/10.1016/S0304-3975(96)00169-7

Neelakantan R. Krishnaswami. 2013. Higher-Order Reactive Programming without Spacetime Leaks. In *International Conference on Functional Programming (ICFP)*.

Hugh C. Lauer and Roger M. Needham. 1979. On the Duality of Operating System Structures. *ACM SIGOPS Operating Systems Review* 13, 2 (apr 1979), 3–19. https://doi.org/10.1145/850657.850658

Henry M Levy. 1984. *Capability-based computer systems*. Digital Press.

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (Sep 2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

S. Maffeis, J. C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *2010 IEEE Symposium on Security and Privacy*. 125–140. https://doi.org/10.1109/SP.2010.16

J. C. C. McKinsey and Alfred Tarski. 1948. Some Theorems About the Sentential Calculi of Lewis and Heyting. *J. Symb. Log.* 13, 1 (1948), 1–15. https://doi.org/10.2307/2268135

Adrian Mettler, David A. Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society. https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. USA. Advisor(s) Shapiro, Jonathan S. AAI3245526.

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2005. L3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 293–307.

Flemming Nielson and Hanne Riis Nielson. 1999. *Type and Effect Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–136. https://doi.org/10.1007/3-540-48092-7_6

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bulleting Symbolic Logic* 5, 2 (06 1999), 215–244. https://projecteuclid.org:443/euclid.bsl/1182353620

Dominic A. Orchard, Vilem Liepelt, and Harley Eades. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* (June 2019). https://kar.kent.ac.uk/74450/

P. W. O'Hearn. 1993. A model for syntactic control of interference. *Mathematical Structures in Computer Science* 3, 4 (Dec 1993), 435–465. https://doi.org/10.1017/S0960129500000311

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 123–135. https://doi.org/10.1145/2628136.2628160

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322

John C. Reynolds. 1978. Syntactic Control of Interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78)*. ACM, 39–46. https://doi.org/10.1145/512760.512766 event-place: Tucson, Arizona.

J. C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Amr Sabry. 1998. What is a purely functional language? *Journal of Functional Programming* 8, 1 (Jan 1998), 1–22. https://doi.org/10.1017/S0956796897002943

Ian Stark. 1996. Categorical models for local names. *LISP and Symbolic Computation* 9, 1 (01 Feb 1996), 77–107. https://doi.org/10.1007/BF01806033

Tachio Terauchi and Alex Aiken. 2006. A Capability Calculus for Concurrency and Determinism. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings (Lecture Notes in Computer Science)*, Christel Baier and Holger Hermanns (Eds.), Vol. 4137. Springer, 218–232. https://doi.org/10.1007/11817949_15

Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (jun 1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-a

Philip Wadler. 1998. The Marriage of Effects and Monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/289423.289429

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM* 17, 6 (Jun 1974), 337–345. https://doi.org/10.1145/355616.364017