A Framework for Substructural Type Systems *

James Wood[1] (✉) and Robert Atkey[1] (✉)

University of Strathclyde, Glasgow, UK
{james.wood.100,robert.atkey}@strath.ac.uk

Abstract. Mechanisation of programming language research is of growing interest, and the act of mechanising type systems and their metatheory is generally becoming easier as new techniques are invented. However, state-of-the-art techniques mostly rely on *structurality* of the type system — that weakening, contraction, and exchange are admissible and variables can be used unrestrictedly once assumed. Linear logic, and many related subsequent systems, provide motivations for breaking some of these assumptions.

We present a framework for mechanising the metatheory of certain substructural type systems, in a style resembling mechanised metatheory of structural type systems. The framework covers a wide range of simply typed syntaxes with semiring usage annotations, via a metasyntax of typing rules. The metasyntax for the premises of a typing rule is related to bunched logic, featuring both sharing and separating conjunction, roughly corresponding to the additive and multiplicative features of linear logic. We use the uniformity of syntaxes to derive type system-generic renaming, substitution, and a form of linearity checking.

Keywords: Formalised syntax · substructural types · mechanised metatheory · quantitative typing

1 Introduction

In this paper, we treat the metatheory of a class of substructural type systems related to linear logic [11]. This class is variously known as *coeffectful* [17, 18], *quantitative* [4, 7], or *resource-aware* [10], or is given no particular name [1, 19], and generalises bounded linear logic to track variable usage with semiring annotations. In all of these systems, we have some ambient semiring $\mathscr{R}$, and in the judgements of the type system, variables are annotated by elements of $\mathscr{R}$ describing *how* that variable can be used. The additive structure of $\mathscr{R}$ gives the ability to count, or otherwise accumulate, usages of variables in multiple subterms. The multiplicative structure gives rise to a form of modality, for example allowing multiple or unlimited reuse, or movement between security levels, in the type system.

* James Wood is supported by an EPSRC Studentship. Robert Atkey is supported by EPSRC grant EP/T026960/1.

The aspect of such systems we tackle here is their basic metatheory and mechanisation thereof.

We build upon both the general structural framework of Allais et al. [3] and the substructural techniques of Wood and Atkey [21]. The way Allais et al. consolidate and codify mechanisation techniques for propositional natural deduction systems based on intrinsically typed syntax and de Bruijn indices, we aim to replicate for linear-like systems based on semiring usage annotations. By picking a trivial semiring, our work can subsume that of Allais et al., except for the many pieces of machinery we have not yet ported to this new framework.

Our work complements that of Orchard et al. [17] on the Granule programming language. Where Granule focuses on writing programs *in* the language and running them, we focus on metatheoretic reasoning *about* type systems.

Our work is similar in scope to that of Licata et al. [13], though we work in a natural deduction style rather than a sequent calculus style. Where Licata et al. are much more agnostic in terms of substructurality — allowing for non-commutative and bunched logics — we are much more agnostic in terms of syntax. The system of Licata et al. is essentially a single calculus, supporting "product" (F) types and "function" (U) types, parametrised on a *mode theory* describing its structural rules. For this system, they derive the strong result of cut elimination. Meanwhile, we leave syntax design to the user, and consequently can only guarantee substitution (which we can only get because of our commitment to natural deduction).

This paper proceeds as follows. In section 2, we review and fix conventions pertaining to partially ordered semirings and vectors over them. In section 3, we introduce an informal meta-syntax allowing us to state substructural typing rules succinctly and without explicit reference to contexts. In section 4, we mechanise that meta-syntax, giving a type of *descriptions* of type systems, and interpreting those descriptions as types of intrinsically typed terms. In section 5, we discuss usage-aware environments: a generalisation of the structures used in simultaneous renaming and substitution proofs. We use environments in section 6 to state an alternative elimination principle for terms, and give examples of such eliminations in section 7. The examples are syntax-generic renaming and substitution, a specific denotational semantics, and a syntax-generic usage elaborator. Finally, we conclude and discuss future work in section 8

The work presented in this paper has been mechanised in Agda, with the code available for building upon [22].

2 Vectors over semirings

The basic algebraic structure we deal with is *partially ordered semirings*, or *posemirings* for short. A posemiring is a (not necessarily commutative) semiring on a partially ordered set, where both operations are monotonic. As in many similar formalisms, posemiring elements represent usage restrictions, with addition collecting restrictions from multiple uses, multiplication handling usage

under a modality, and the order giving subsumption of restrictions, comparable to subtyping.

Definition 1. *A* posemiring *is a tuple* $(\mathscr{R}, \leq, 0, +, 1, *)$ *such that* $(\mathscr{R}, \leq)$ *is a partially ordered set,* $(\mathscr{R}, 0, +)$ *is a commutative monoid,* $(\mathscr{R}, 1, *)$ *is a monoid,* $+$ *and* $*$ *are monotonic, and* $*$ *distributes over* 0 *and* $+$ *on both sides.*

Example 1 (Zero-one-many). The poset $\{0 > \omega < 1\}$ forms a posemiring under normal numeric addition (with $1 + 1 = 1 + \omega = \omega + \omega = \omega$) and multiplication (with $\omega * \omega = \omega$). This gives us a way to mark whether variables are unused (0), used linearly (1), or used unrestrictedly (ω) in the current (sub)term. The ordering says that unrestricted-use variables can also remain unused or be used linearly.

Example 2 (Variance). The set $\{\sim\sim, \uparrow\uparrow, \downarrow\downarrow, ??\}$, with $\sim\sim$ at the bottom and $??$ at the top of the order, forms a posemiring with addition being *meet*, 0 being *top* ($??$), 1 being $\uparrow\uparrow$, and multiplication being commutative and determined by $\downarrow\downarrow * \downarrow\downarrow = \uparrow\uparrow$ and $\sim\sim * \downarrow\downarrow = \sim\sim * \sim\sim = \sim\sim$. This gives us a way to track the variance with which variables are used, in the aim of all terms being monotonic in their free variables. $\uparrow\uparrow$ stands for covariance, $\downarrow\downarrow$ for contravariance, $\sim\sim$ for invariance, and $??$ for a variable with no guarantees, in which we must be constant.

An element of a chosen posemiring $\mathscr{R}$ describes the usage restrictions on a variable. Therefore, a *vector* of elements from $\mathscr{R}$ describes the usage restrictions of a whole context's worth of variables. From the posemiring operations of $\mathscr{R}$, we derive the standard vector operations of zero, addition, and multiplication by a scalar. We can also form the standard basis vectors at any given dimension. From the order on $\mathscr{R}$, we get a pointwise order on vectors.

Vectors of a given length form a *module* over the posemiring $\mathscr{R}$, analogously to how vectors over a field form a vector space. The partial order on such vectors is pointwise.

Definition 2. *A* (left) module over a posemiring*, given a posemiring* $\mathscr{R}$*, is a partially ordered commutative monoid* $(M, 0_M, +_M)$ *with, for each* $r \in \mathscr{R}$*, a pomonoid morphism* $r \cdot (-) : M \to M$*, such that the collection of these respects the posemiring structure on* r*. Specifically, for all instantiations of the variables:*

- *If* $r \leq r'$ *and* $u \leq u'$*, then* $r \cdot u \leq r' \cdot u'$*.*
- $r \cdot 0_M = 0_M$ *and* $r \cdot (u +_M v) = r \cdot u +_M r \cdot v$*.*
- $0 \cdot u = 0_M$ *and* $(r + s) \cdot u = r \cdot u +_M s \cdot u$*.*
- $1 \cdot u = u$ *and* $(r * s) \cdot u = r \cdot (s \cdot u)$*.*

We care to define modules so as to define *module morphisms*, also known as *linear maps*, which we use extensively when relating two contexts (as we do, for example, in simultaneous substitution). For the sake of mechanisation, we choose to define module morphisms *relationally* rather than *functionally*, giving a somewhat unfamiliar-looking definition that is equivalent to the usual

functional definition. The main advantage of this relational approach is that proofs of relatedness for typical linear maps compose and decompose via data constructors and pattern matching.

Definition 3. *A* (relational) linear map Ψ *between modules* M *and* N *over a posemiring* $\mathscr{R}$ *is a relation* $\sim$ *on the underlying sets of* M *and* N *satisfying the following laws (with* $\to$ *standing for implication and quantifiers binding most loosely).*

- $\forall u, u', v, v'.\ u \leq u' \to v' \leq v \to u \sim v \to u' \sim v'$
- $\forall v.\ (\exists u.\ u \leq 0\ \wedge\ u \sim v) \to v \leq 0$
- $\forall u_0, u_1, v.\ (\exists u.\ u \leq u_0 + u_1\ \wedge\ u \sim v) \to$
  $(\exists v_0, v_1.\ u_0 \sim v_0\ \wedge\ u_1 \sim v_1\ \wedge\ v \leq v_0 + v_1)$
- $\forall r, u', v.\ (\exists u.\ u \leq ru'\ \wedge\ u \sim v) \to (\exists v'.\ u' \sim v'\ \wedge\ v \leq rv')$
- $\forall u.\ \exists v.\ u \sim v\ \wedge\ \forall v'.\ u \sim v' \to v' \leq v$

Intuitively, $Q \sim P$, where P and Q are row vectors, is equivalent to $P \leq Q\Psi$, where Ψ is the matrix representing the linear map and on the right is a vector-matrix multiplication. It is important that we think of *row* vectors and *right*-multiplication by a matrix because, without commutativity of the underlying posemiring, we can only expect $(rQ)\Psi = r(Q\Psi)$ and not $\Psi(rQ) = r(\Psi Q)$. In section 5, we use the matrix notation for convenience, while in the Agda code we see Ψ .rel P Q.

3 Bunched Typing Rules

We now let $\mathscr{R}$ be an arbitrary posemiring. Our framework represents well typed and $\mathscr{R}$-usaged terms *intrinsically*. Intrinsic typing means that we represent well typed and $\mathscr{R}$-usaged terms (and *only* those) as inhabitants of an inductive family $\mathcal{R}\gamma \vdash A$ indexed by usage context $\mathcal{R}$, type context γ, and type A. We represent the shape of a context as a nullary-binary tree, with typing and usage contexts being functions that assign types and elements of $\mathscr{R}$, respectively, to leaves of the tree. Using trees instead of lists for typing contexts has the advantage that extension of a context by multiple variables does not lead to complex counting arguments to access the pre-existing variables, because context extension is (judgementally) injective. However, these precise details will eventually become irrelevant, as we will be able to use simultaneous renaming to smooth over any structural differences between contexts.

Figure 1 presents a prototypical example of a system that our framework can represent, which is a subsystem of the $\lambda\mathcal{R}$ system of Wood and Atkey [21]. Each rule is given as a constructor: the premises are named p, s, t, etc., and the conclusion is a constructor applied to those metalanguage variables. Object language variables are represented intrinsically as members of the type $\mathcal{R}\gamma \ni A$, which is a proof that the type A appears in the typing context, $i : \gamma \ni A$, together with a proof that $\mathcal{R} \leq \langle i |$. Expanding the vector notation, the latter condition says that the selected variable i must have a usage annotation ≤ 1 in $\mathcal{R}$, while

$$\frac{x : \mathcal{R}\gamma \ni A}{\mathsf{var}\ x : \mathcal{R}\gamma \vdash A}$$

$$\frac{t : \mathcal{R}\gamma, 1A \vdash B}{\multimap\!\mathsf{I}\ t : \mathcal{R}\gamma \vdash A \multimap B} \qquad \frac{p : \mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \begin{array}{r} s : \mathcal{P}\gamma \vdash A \multimap B \\ t : \mathcal{Q}\gamma \vdash A \end{array}}{\multimap\!\mathsf{E}\ p\ s\ t : \mathcal{R}\gamma \vdash B}$$

$$\frac{t : \mathcal{R}\gamma \vdash A_i}{\oplus\mathsf{I}_i\ t : \mathcal{R}\gamma \vdash A_0 \oplus A_1} \qquad \frac{p : \mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \begin{array}{ll} t : \mathcal{Q}\gamma, 1A \vdash C \\ s : \mathcal{P}\gamma \vdash A \oplus B & u : \mathcal{Q}\gamma, 1B \vdash C \end{array}}{\oplus\mathsf{E}\ p\ s\ t\ u : \mathcal{R}\gamma \vdash C}$$

$$\frac{p : \mathcal{R} \leq r\mathcal{P} \qquad t : \mathcal{P}\gamma \vdash A}{!\mathsf{I}\ p\ t : \mathcal{R}\gamma \vdash !rA} \qquad \frac{p : \mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \begin{array}{r} s : \mathcal{P}\gamma \vdash !rA \\ t : \mathcal{Q}\gamma, rA \vdash C \end{array}}{!\mathsf{E}\ p\ s\ t : \mathcal{R}\gamma \vdash C}$$

Fig. 1. A prototypical posemiring-usaged system

all other variables must have a usage annotation ≤ 0. We use the constructors $\swarrow$ and $\searrow$ to describe a path down the nullary-binary tree, terminated by the word here. The var rule imports variables into terms.

The remaining rules are the introduction and elimination rules for three type constructors: $\multimap\!\mathsf{I}$ and $\multimap\!\mathsf{E}$ for function types $A \multimap B$ where the bound variable is annotated with 1 for "use once"; $\oplus\mathsf{I}$ and $\oplus\mathsf{E}$ for sum types $A \oplus B$; and $!\mathsf{I}$ and $!\mathsf{E}$ for a $\mathscr{R}$-annotated exponential modality $!rA$.

There are two key observations to make about this system, which will guide the way we build our generic framework for $\mathscr{R}$-annotated substructural systems:

1. Every rule repeats the typing context γ throughout its premises and conclusion. The only time the typing context is modified is to add additional variables in the rules that bind fresh variables ($\multimap\!\mathsf{I}$, $\oplus\mathsf{E}$, $!\mathsf{E}$).
2. Rules with multiple typing premises must describe how the usages of the conclusion (always denoted $\mathcal{R}$) are distributed across the premises. In the $\multimap\!\mathsf{E}$ rule, the usages are separated into two parts $\mathcal{P}$ and $\mathcal{Q}$ for the premises. This is an example of a *multiplicative* rule in the terminology of Linear Logic [11]. In the $\oplus\mathsf{E}$ we see an example of an *additive* rule, where the usage context $\mathcal{Q}$ is shared between the premises t and u[1]. The $!\mathsf{I}$ rule uses scaling by r of the usages of the premise.

These observations indicate a way to regularise and streamline the presentation of this system. Instead of treating each premise and the conclusion as having potentially unrelated typing and usage constraints, we make use of combinators for combining premises that will relate their usage and typing contexts to the conclusion by construction. This idea comes from the work of Rouvoet et al. [20], including the $\dot{\to}$ and $\twoheadrightarrow$ connectives we use later. To handle binders, which introduce variables, we make use of a combinator that adds a variable with a given $\mathscr{R}$-annotation to an ambient context, without having to explicitly mention

[1] There is an unfortunate clash of terminology here: multiplicative rules *add* their usage contexts, while additive rules *share* their usage contexts.

the parts of the context that have not changed. This technique is already present in some paper presentations of type systems, and is formalised by Allais et al. [3]. To manage how usage annotations are distributed between premises, we use the separating ($*$) and sharing ($\dot{\times}$) conjunction connectives from Bunched Implications [16]. To handle the !I rule, we will need a *scaling* modality, $r \cdot -$. The semantics of the bunched connectives we will use in this paper are:

$$\begin{aligned}
\dot{1}\,\mathcal{R} &:= 1 \\
(T \mathbin{\dot{\times}} U)\,\mathcal{R} &:= T\,\mathcal{R} \times U\,\mathcal{R} \\
(T \mathbin{\dot{\to}} U)\,\mathcal{R} &:= T\,\mathcal{R} \to U\,\mathcal{R} \\
I^*\,\mathcal{R} &:= \mathcal{R} \leq 0 \\
(T * U)\,\mathcal{R} &:= \Sigma \mathcal{P}, \mathcal{Q}.\ (\mathcal{R} \leq \mathcal{P} + \mathcal{Q}) \times T\,\mathcal{P} \times U\,\mathcal{Q} \\
(T \mathbin{-\!\!*} U)\,\mathcal{P} &:= \Pi \mathcal{Q}, \mathcal{R}.\ (\mathcal{R} \leq \mathcal{P} + \mathcal{Q}) \to T\,\mathcal{Q} \to U\,\mathcal{R} \\
(r \cdot T)\,\mathcal{R} &:= \Sigma \mathcal{P}.\ (\mathcal{R} \leq r\mathcal{P}) \times T\,\mathcal{P}.
\end{aligned}$$

The function connectives $\dot{\to}$ and $-\!\!*$ are not used in typing rules, but are used in the rest of the framework (though one can interpret the horizontal line in a typing rule as $\dot{\to}$ plus universal quantification). An important point to note is that bunched combinators induce *linear* combinations of substructures, in the sense of the linear algebra of posemirings described in the previous section.

$$\frac{x : \exists\, A}{\mathsf{var}\ x : \vdash A} \qquad \frac{t : 1A \vdash B}{\multimap\!\mathsf{I}\ t : \vdash A \multimap B} \qquad \frac{(t : \vdash A \multimap B) \ * \ (s : \vdash A)}{\multimap\!\mathsf{E}\ t\ s : \vdash B}$$

$$\frac{t : \vdash A_i}{\oplus\mathsf{I}_i\ t : \vdash A_0 \oplus A_1} \qquad \frac{(s : \vdash A \oplus B) \ * \ ((t : 1A \vdash C) \ \dot{\times} \ (u : 1B \vdash C))}{\oplus\mathsf{E}\ s\ t\ u : \vdash C}$$

$$\frac{t : r \cdot (\vdash A)}{!\mathsf{I}\ t : \vdash\, !_r A} \qquad \frac{(s : \vdash\, !_r A) \ * \ (t : rA \vdash C)}{!\mathsf{E}\ s\ t : \vdash C}$$

Fig. 2. The prototypical system of figure 1 restated in terms of bunched combinators.

Figure 2 shows our prototypical system restated with implicit contexts and the bunched combinators. The inductive family is now denoted $\vdash A$, only mentioning context extensions, as we do in the rules $\multimap$I, $\oplus$E and !E. Thus, in the var rule, the context is completely suppressed. The $\multimap$I rule just has to state that a new variable with usage annotation 1 and type A is added to the context. The $\multimap$E rule uses the separating conjunction ($*$) to combine the premises, indicating that the usages of the two premises are added together for the conclusion. The $\oplus$E rule demonstrates the sharing conjunction $\dot{\times}$: the scrutinee term s and the clause terms t, u are combined by separating conjunction, because their usages must be combined, but the clause terms are combined by the sharing conjunction, because they have the same usage context.

Bunched combinators, along with suppression of unchanged typing contexts, leads to a more streamlined presentation of the system without the clutter of

explicit usage context inequalities. However, the larger advantage for us is that systems are constructed using these combinators *automatically* admit renaming, substitution, and other scope-, type-, and usage-safe traversals. If we were to allow arbitrary modification of the context in premises, these results would not be possible, since there would be no guarantee that a substitution (for instance) could be "pushed" up from a conclusion to the premises. As we will see in section 5, our generic notion of environment (e.g., a simultaneous substitution) is based around linear transformations, and so automatically commutes with the linear combinations of premises induced by the bunched connectives. This is the key to our generic results for all of the systems describable in our framework.

4 Generic syntax

We take the insights of the previous section and use them to build a generic framework for posemiring-annotated substructural systems in Agda. We will first show *descriptions* of systems, which are comprised of rules that have premises combined using the bunched combinators. We then show how to construct the Agda data type of intrinsically well scoped, typed, and resourced terms for any given system of our framework. We use the prototypical system from figure 2 as a running example. Section 4.3 presents further examples that our framework can express.

We now start to use Agda notation for record and data type declarations, to emphasise that our framework has been implemented.

4.1 Descriptions of Systems

A type System is made up of multiple Rules. Each Rule comprises a Premises and a conclusion type. We assume that there is a Ty : Set of types for the system in scope.

The Premise data type describes premises of rules, using the bunched combinators from section 3. A single premise is introduced by the $\langle \_ \text{`}\vdash \_ \rangle$ constructor. This allows binding of additional variables Δ (with specified types and usage annotations) and the specification of a conclusion type A for this premise. The remaining constructors are descriptions for the bunched connectives.

```
data Premises : Set where
  ⟨_`⊢_⟩ : (Δ : Ctx) (A : Ty) → Premises
  `𝕚 : Premises;  _`×_ : (p q : Premises) → Premises
  `I* : Premises; _`*_ : (p q : Premises) → Premises
  _`·_ : (r : Ann) (p : Premises) → Premises
```

A Rule is a pair of some Premises and a conclusion. We use an infix arrow as a suggestive notation for rules.

```
record Rule : Set where
  constructor _⟹_
  field premises : Premises; conclusion : Ty
```

Finally, a System consists of a set of rule labels (i.e., constructor names), and for each label a description of the corresponding rule. We use $\triangleright$ as infix notation for systems to associate the label set with the rules.

```
record System : Set₁ where
  constructor _▷_
  field Label : Set; rules : (l : Label) → Rule
```

As an example, we transcribe the system defined in figure 2 into a description. We give the set of types of this system as a data type Ty (together with a base type ι). We assume that there is a posemiring Ann in scope for the annotations.There is one label for each instantiation of a logical rule, but the labels contain no further information about subterms or restrictions on the context. This will be provided when we associate labels with Rules in a System.

```
data Ty : Set where
  ι : Ty
  _⊸_ _⊕_ : (A B : Ty) → Ty
  ! : (r : Ann) (A : Ty) → Ty

data Side : Set where ll rr : Side
```

```
data 'λR : Set where
  '⊸I '⊸E : (A B : Ty) → 'λR
  '⊕I : (i : Side) (A B : Ty) → 'λR
  '⊕E : (A B C : Ty) → 'λR
  '!I : (r : Ann) (A : Ty) → 'λR
  '!E : (r : Ann) (A C : Ty) → 'λR
```

To build a system, we associate with each label a rule:

```
λR : System
λR = 'λR ▷ λ where
  ('⊸I A B)    → ⟨ [ 1# · A ]ᶜ '⊢ B ⟩                          ⟹ (A ⊸ B)
  ('⊸E A B)    → (⟨ []ᶜ '⊢ A ⊸ B ⟩ '* ⟨ []ᶜ '⊢ A ⟩)            ⟹ B
  ('!I r A)    → (r '· ⟨ []ᶜ '⊢ A ⟩)                           ⟹ (! r A)
  ('!E r A C)  → (⟨ []ᶜ '⊢ ! r A ⟩ '* ⟨ [ r · A ]ᶜ '⊢ C ⟩)     ⟹ C
  ('⊕I ll A B) → ⟨ []ᶜ '⊢ A ⟩                                  ⟹ (A ⊕ B)
  ('⊕I rr A B) → ⟨ []ᶜ '⊢ B ⟩                                  ⟹ (A ⊕ B)
  ('⊕E A B C) →
    ⟨ []ᶜ '⊢ A ⊕ B ⟩ '* (⟨ [ 1# · A ]ᶜ '⊢ C ⟩ '×̇ ⟨ [ 1# · B ]ᶜ '⊢ C ⟩) ⟹ C
```

Compared to figure 2, modulo the Agda notation, we can see that the fundamental structure has been preserved: the rules match one-to-one, and the bunched premises are the same. A major difference is that we do not include a counterpart to the var rule in a System. Variables are common to all the systems representable in our framework.

4.2 Terms of a System

The next thing we want to do is to build terms in the described type system. The following definitions are useful for talking about types indexed over contexts, judgement forms, and judgement forms admitting newly bound variables, respectively.

```
OpenType : ∀ ℓ → Set (suc ℓ)
OpenType ℓ = Ctx → Set ℓ

OpenFam : ∀ ℓ → Set (suc ℓ)
OpenFam ℓ = Ctx → Ty → Set ℓ

ExtOpenFam : ∀ ℓ → Set (suc ℓ)
ExtOpenFam ℓ = Ctx → OpenFam ℓ
```

To specify the meaning of descriptions, we assume some X : ExtOpenFam, over which we form one layer of syntax, using the function ⟦_⟧p that interprets Premises defined below. The first argument to X is the new variables bound by this layer of syntax, as exemplified in the first clause of ⟦_⟧p. The second argument is the context containing the variables being carried over from the previous layer. Notice that this is not, in general, the same as the context from the previous layer, because the usage annotations may have been changed by connectives like _‘*_ and _‘·_. The third argument is the type of subterm required.

The remainder of the clauses of ⟦_⟧p are the interpretation into bunched combinators. The superscript ^{c} on the bunched connectives denotes that they have been lifted from predicates on usage vectors to predicates on contexts, with the type component of the context shared throughout. Additive connectives $\dot{\mathsf{I}}$ and $\dot{\times}$ are already polymorphic (not relying on anything specific about usage vectors), so do not need a ^{c} variant.

```
⟦_⟧p : Premises → ExtOpenFam ℓ → OpenType ℓ
⟦ ⟨ Δ ‘⊢ A ⟩ ⟧p X Γ = X Δ Γ A
⟦ ‘i ⟧p X = i;      ⟦ p ‘×̇ q ⟧p X = ⟦ p ⟧p X ×̇ ⟦ q ⟧p X
⟦ ‘I* ⟧p X = I*c; ⟦ p ‘* q ⟧p X = ⟦ p ⟧p X *c ⟦ q ⟧p X
⟦ r ‘· p ⟧p X = r ·c ⟦ p ⟧p X
```

The interpretation of a Rule checks that the rule targets the desired type and then interprets the rule's premises ps. Notice that the interpretation of the premises is independent of the conclusion of the rule, which accounts for the use of OpenType in ⟦_⟧p versus OpenFam in ⟦_⟧r.

```
⟦_⟧r : Rule → ExtOpenFam ℓ → OpenFam ℓ
⟦ ps ⟹ A′ ⟧r X Γ A = A′ ≡ A × ⟦ ps ⟧p X Γ
```

The interpretation of a System is to choose a rule label l from L and interpret the corresponding rule $rs\ l$ in the same context and for the same conclusion.

```
⟦_⟧s : System → ExtOpenFam ℓ → OpenFam ℓ
⟦ L ▷ rs ⟧s X Γ A = Σ[ l ∈ L ] ⟦ rs l ⟧r X Γ A
```

The most obvious way to make such an X is to use some existing OpenFam on an extended context. We defined Scope to do this: take the new variables $Δ$, concatenate them onto the existing context $Γ$, and pass the extended context onto the judgement T.

```
Scope : ∀ {ℓ} → OpenFam ℓ → ExtOpenFam ℓ
Scope T Δ Γ A = T (Γ ++ᶜ Δ) A
```

We use Scope to deal with new variables in syntax. Terms resemble the free monad over a layer-of-syntax functor, though that picture is complicated by variable binding. A term is either a variable or a use of a logical rule together with terms for each of the required subterms. The Size argument is a use of Agda's sized types to record that subterms are smaller than the surrounding term for the termination checker.

```
data [_,_]⊢_ (d : System) : Size → OpenFam 0ℓ where
  'var : ∀[ _∋_                    ⇾ [ d , ↑ sz ]⊢_ ]
  'con : ∀[ ⟦ d ⟧s (Scope [ d , sz ]⊢_) ⇾ [ d , ↑ sz ]⊢_ ]
```

This definition uses $\dot{\to}$, which, analogously to $\dot{\times}$, is an index-preserving version of the function space. We take $\dot{\to}$ to handle n many indices — in this case two (the context and the type). The notation $\forall[\ T\ ]$ stands for $\forall\ \{x_1\ \ldots\ x_n\} \to T\ x_1\ \ldots\ x_n$, where T is a type family with n many indices.

Terms in this data type are difficult to write by hand, due to the need for proofs that the usage contexts are handled correctly. For example, the following term is needed to show that, in the $\{0, 1, \omega\}$ (linearity) posemiring of example 1, $!\omega$ forms a comonad. Pattern synonyms $\multimap$I, !E′, and !I′ stand for applications of 'con, with the latter two taking explicit usage contexts and proofs. On concrete posemirings (as in this example), unification is particularly poor at inferring the usage contexts from the proofs because addition and multiplication are no longer (judgementally) injective. The function var# is a way of turning a statically known de Bruijn level and a usage proof into an application of 'var.

```
cojoin-!ω : ∀ A → [ λR , ∞ ] []ᶜ ⊢ (! ω# A ⊸ ! ω# (! ω# A))
cojoin-!ω A =
  ⊸I (!E′ ([] ++ [ 1# ]) ([] ++ [ 0# ]) ([]ₙ ++ₙ [ ≤-refl ]ₙ)
    (var# 0 (([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ []ₙ))
    (!I′    (([] ++ [ 0# ]) ++ [ ω# ])
            (([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ [ ≤-refl ]ₙ)
      (!I′ ((([] ++ [ 0# ]) ++ [ ω# ]) ++ [])
           ((([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ [ ≤-refl ]ₙ) ++ₙ []ₙ)
        (var# 1
          (((([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ [ ω≤1 ]ₙ) ++ₙ []ₙ) ++ₙ []ₙ))))))
```

Writing terms like this is clearly unsustainable. We will see a way of automating the necessary proofs via a System-generic elaborator in section 7.2.

4.3 Other syntaxes and syntactic forms

The system $\mu\tilde{\mu}$. We can encode a usage-annotated version of System L/the $\mu\tilde{\mu}$-calculus [8] — a syntax for classical logic — in such a way that contexts capture the undistinguished parts of the sequent. As such, the generic substitution lemma

we get in section 7.1 is the form of substitution required in standard $\mu\tilde{\mu}$-calculus metatheory. Though the $\mu\tilde{\mu}$-calculus is originally described as a sequent calculus [8], we use the techniques of Herbelin [12, p. 12] and Lovas and Crary [14] to present it as a natural deduction system, thus giving a notion of *variable* to the system.

Unlike the single judgement form of $\lambda\mathcal{R}$ and standard simply typed λ-calculi, the $\mu\tilde{\mu}$-calculus has three judgement forms: terms, coterms, and commands. Read logically, terms and coterms are seen to, respectively, prove and refute propositions (types), while commands exhibit contradictions. This means that the abstract Ty in the generic framework is instantiated to Conc (for *conclusion*) as below, with Ty not being exposed directly to the generic framework. For now, we just consider multiplicative disjunction $⅋$ (*par*) and negation/duality, beside an uninterpreted base type. These are enough to exhibit classical behaviour.

```
data Ty : Set where
  base : Ty
  _⅋_ : (rA sB : Ann × Ty) → Ty
  _^⊥ : (A : Ty) → Ty
```

```
data Conc : Set where
  com : Conc
  trm cot : (A : Ty) → Conc
```

With Ty instantiated as Conc, all terms are assigned Conc type, as are all the variables. No variables are given com type, similar to how in the bidirectional typing syntax of Allais et al. [3, p. 25], no variables are given Check type. How to observe this invariant is covered in the latter paper, so we will not repeat it here (having not yet seen how to write traversals on terms).

The syntax comprises a *cut* between a term and a coterm of the same type, the eponymous μ and $\tilde{\mu}$ constructs for proof by contradiction, and then term and coterm (introduction and elimination) forms for negation and *par*.

```
data ‘MMT : Set where
  ‘cut ‘μ ‘μ~ : (A : Ty) → ‘MMT
  ‘λ ‘λ~ : (A : Ty) → ‘MMT
  ‘⟨-,-⟩ ‘μ⟨-,-⟩ : (rA sB : Ann × Ty) → ‘MMT
```

```
MMT : System
MMT = ‘MMT ▷ λ where
  (‘cut A) → ⟨ []ᶜ ‘⊢ trm A ⟩ ‘∗ ⟨ []ᶜ ‘⊢ cot A ⟩ ⟹ com
  (‘μ A)   → ⟨ [ 1# , cot A ]ᶜ ‘⊢ com ⟩           ⟹ trm A
  (‘μ~ A)  → ⟨ [ 1# , trm A ]ᶜ ‘⊢ com ⟩           ⟹ cot A
  (‘λ A)   → ⟨ []ᶜ ‘⊢ cot A ⟩                     ⟹ trm (A ^⊥)
  (‘λ~ A)  → ⟨ []ᶜ ‘⊢ trm A ⟩                     ⟹ cot (A ^⊥)
  (‘⟨-,-⟩ rA@(r , A) sB@(s , B)) →
    r ‘· ⟨ []ᶜ ‘⊢ cot A ⟩ ‘∗ s ‘· ⟨ []ᶜ ‘⊢ cot B ⟩  ⟹ cot (rA ⅋ sB)
  (‘μ⟨-,-⟩ rA@(r , A) sB@(s , B)) →
    ⟨ [ r , cot A ]ᶜ ++ᶜ [ s , cot B ]ᶜ ‘⊢ com ⟩   ⟹ trm (rA ⅋ sB)
```

Duplicability There is one more bunched combinator we have experimented with adding to the framework:

$$(\Box T)\,\mathcal{R} \coloneqq \Sigma \mathcal{R}'.\ (\mathcal{R}' \leq \mathcal{R}) \times (\mathcal{R}' \leq 0) \times (\mathcal{R}' \leq \mathcal{R}' + \mathcal{R}') \times T\,\mathcal{R}'$$

The idea of $(\Box T)\,\mathcal{R}$ is to assert that $\mathcal{R}$, or some refinement of it, can be both discarded and duplicated indefinitely, and in the refinement we have a T. We use this combinator to introduce subterms that are used an unknown number of times, for example the continuations of the eliminator of an inductive type, or other fixed points. We can also use it in linear/non-linear style systems [6] to make sure linear variables are not available in the intuitionistic fragment.

Adding the $\Box$ combinator is the only thing we have found that requires our linear maps be functional rather than merely relational.

5 Environments

We have now seen how to build data types of intrinsically well typed and well usaged terms for a given System. In the next section, we will define a generic traversal function that assigns a "semantics" to each term. Traversals operate on open terms, so they need a way to assign semantics to variables in a typed and usage respecting manner. This is the function fulfilled by *environments*.

Given a semantic notion of variable $\mathcal{V} : \mathsf{OpenFam}$, we use the notation $\Gamma \overset{\mathcal{V}}{\models} A$ meaning $\mathcal{V}\,\Gamma\,A$ for the type of inhabitants of $\mathcal{V}$ in the context Γ at type A. In the non-substructural systems of Allais et al. [3], a $\mathcal{V}$-environment $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$ is nothing more than a function $\forall A \to \Delta \ni A \to \Gamma \overset{\mathcal{V}}{\models} A$, mapping variables to $\mathcal{V}$-things. In our usage annotated setting though, we must correctly distribute resources tracked by the annotations; making sure that we have enough resources in Γ to cover all the demands in Δ. Following our previous work [21], this accounting is expressed via the presence of a linear transformation:

Definition 4 (Environment). *A* $\mathcal{V}$-environment *between annotated contexts* Γ *and* Δ *(decomposed as* $\mathcal{P}\gamma$ *and* $\mathcal{Q}\delta$*, respectively, when convenient) is a linear map* $\Psi : \mathscr{R}^{|\Delta|} \to \mathscr{R}^{|\Gamma|}$ *(written postfix) such that* $\mathcal{P} \leq \mathcal{Q}\Psi$ *and for each* A*,* $\mathcal{P}'$*, and* $\mathcal{Q}'$ *such that* $\mathcal{P}' \leq \mathcal{Q}'\Psi$*, a "lookup" function from* $\mathcal{Q}'\delta \ni A$ *to* $\mathcal{P}'\gamma \overset{\mathcal{V}}{\models} A$*.*

In Agda code, we use $[\ \mathcal{V}\ ]\ \Gamma \vDash A$ instead of $\Gamma \overset{\mathcal{V}}{\models} A$ and $[\ \mathcal{V}\ ]\ \Gamma \Rightarrow^{e} \Delta$ instead of $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$.

The specification of the lookup function has some redundancy. Notice that, for $\mathcal{Q}'\Delta \ni A$ to hold, we must have $\mathcal{Q}' \leq \langle i|$ for some i. Instead of $\mathcal{P}' \leq \mathcal{Q}'\Psi$, asking for $\mathcal{P}' \leq \langle i|\Psi$ would be just as general. Additionally, all of the $\mathcal{V}$s we consider satisfy the *subusaging* property (that $\mathcal{P}' \leq \mathcal{P}$ yields a coercion $\mathcal{P}\Gamma \overset{\mathcal{V}}{\models} A \to \mathcal{P}'\Gamma \overset{\mathcal{V}}{\models} A$), in which case we could just ask for an inhabitant of $(\langle i|\Psi)\gamma \overset{\mathcal{V}}{\models} A$. However, we find the stated definition technically expedient because, by this point, basis vectors and raw indices (instead of usage-checked

variables) are below our level of abstraction. We prefer to work with linear relatedness and $\exists$-variables.

By instantiating $\mathcal{V}$ in definition 4, we obtain resource-correct versions of familiar notions: letting $\mathcal{V}$ be $\exists$ yields resource-correct renamings; and letting $\mathcal{V}$ be $\vdash$ (i.e., terms) yields resource-correct substitutions.

We may informally assign variable names to the entries in the domain context.

Example 3. Assume $\mathscr{R}$ is the natural numbers with ordering given by $=$ and the usual addition and multiplication. There is a $\exists$-environment (a renaming)

$$(6a : A, 0b : B, 1c : C, 0d : D) \overset{\exists}{\Longrightarrow} (1C, 2A, 4A).$$

The mapping of variables to variables and matrix giving the linear map Ψ are:

$$\begin{array}{l} 0a : A, 0b : B, 1c : C, 0 : D \exists\, c : C \\ 1a : A, 0b : B, 0c : C, 0 : D \exists\, a : A \\ 1a : A, 0b : B, 0c : C, 0 : D \exists\, a : A \end{array} \qquad \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Note that $(6\,0\,1\,0) = (1\,2\,4)\Psi$. The first column of Ψ, corresponding to variable $6a : A$, contains two 1s because it has been duplicated (via contraction). The second and fourth columns are all 0 because variables b and d have been discarded (via weakening). The third column contains one 1 because c is used once. This 1 appears above the 1s to its left because c has been permuted (via exchange) past a. Each of the rows in the matrix is a basis vector because variables can only be formed in contexts with basis-compatible annotations.

Relocation An environment $\rho : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$ does not determine $\mathcal{P}$ and $\mathcal{Q}$, we can replace them with any $\mathcal{P}'$ and $\mathcal{Q}'$ that are related by the linear map $\rho.\Psi$ (that is, the linear map of environment ρ):

Lemma 1 (relocate). *Given an environment $\rho : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$ and a $\mathcal{P}'$ and a $\mathcal{Q}'$ such that $\mathcal{P}' \leq \mathcal{Q}'(\rho.\Psi)$, there is also an environment of type $\mathcal{P}'\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}'\delta$ with the same linear map and action on variables.*

Relocation will be used when pushing environments into subterms in section 6.3.

Inductive Construction When $\mathcal{V}$ supports subusaging, we can construct a $\mathcal{V}$-environment by cases on the shape of the target context by the following rules, which use the bunched connectives from section 3:

$$\frac{I^*}{\langle\rangle : \overset{\mathcal{V}}{\Longrightarrow} \cdot} \qquad \frac{\rho : \overset{\mathcal{V}}{\Longrightarrow} \Delta_l \quad * \quad \sigma : \overset{\mathcal{V}}{\Longrightarrow} \Delta_r}{\langle \rho, \sigma \rangle : \overset{\mathcal{V}}{\Longrightarrow} \Delta_l, \Delta_r} \qquad \frac{r \cdot \left(M : \overset{\mathcal{V}}{\models} A\right)}{\langle M \rangle : \overset{\mathcal{V}}{\Longrightarrow} rA}$$

Left to right, we can create an environment into the empty context when all usage annotations on the source context are 0; we can create an environment into a concatenated context when we can additively split up the annotations of

the source context and produce environments into both halves from the split sources; and we can create an environment into a singleton context rA when we can divide the source context by r and produce a $\mathcal{V}$-value in the divided context of the appropriate type.

Example 4. Assume $\mathscr{R}$ is the natural numbers with ordering given by $=$ and the usual addition and multiplication, and $\vdash$ is the type of terms for a System with function application. There is an environment (substitution)

$$\langle\langle z\rangle, \langle y\, z\rangle\rangle : (0x : A, 2y : B \multimap C, 3z : B) \overset{\vdash}{\Longrightarrow} (1B, 2C).$$

We rely on the observations that $\begin{pmatrix}0 & 2 & 3\end{pmatrix} = \begin{pmatrix}0 & 0 & 1\end{pmatrix} + \begin{pmatrix}0 & 2 & 2\end{pmatrix}$ and, on the right, that $\begin{pmatrix}0 & 2 & 2\end{pmatrix} = 2\begin{pmatrix}0 & 1 & 1\end{pmatrix}$. Then, we have $0x : A, 0y : B \multimap C, 1z : B \vdash z : B$ and $0x : A, 1y : B \multimap C, 1z : B \vdash y\, z : C$.

We could have used these rules to inductively define what environments are. However, we found that this was difficult to work with. It is often easier to do linear algebraic proofs separately from the rest of an environment. For example, for identity and composition of environments (below), definition 4 is easier to use because we can rely on the identity and composition of linear maps. Concretely, an inductive proof of identity would, for example, involve constructing an environment of type $\mathcal{P}\gamma, \mathcal{Q}\delta \overset{\mathcal{V}}{\Longrightarrow} \mathcal{P}\gamma, \mathcal{Q}\delta$ by constructing environments of types $\mathcal{P}\gamma, 0\delta \overset{\mathcal{V}}{\Longrightarrow} \mathcal{P}\gamma$ and $0\gamma, \mathcal{Q}\delta \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$. These are not identity environments, so we would have to strengthen the induction hypothesis.

Renameability Renamings, i.e. $\exists$-environments, are a particularly important case of environments. Renamings form a category, with identity and composition following from the identity and composition of linear maps. As in the work of Fiore et al. [9], presheaves over renamings are an important class of open families. In Agda code, we abbreviate $\overset{\exists}{\Longrightarrow}$ (which would usually be $[\ \_\exists\_\ ]\_\Rightarrow^{e}\_$) as $\_\Rightarrow^{r}\_$.

In a setting where new variables can be bound in the middle of a derivation, it is important that the values we carry around while traversing a term can handle the existence of variables that appear but they do not use. We call any such notion of value *renameable*. The cofree renameable open type on an open type T is $\Box^r\, T$ (unrelated to the $\Box$ combinator mentioned at the end of section 4.3), with T then being renameable if it forms a $\Box^r$-coalgebra.

Definition 5. *For T an open type, $(\Box^r\, T)\, \Gamma \coloneqq \forall\Big[\ \Big((-) \overset{\exists}{\Longrightarrow} \Gamma\Big) \dot{\to} T\ \Big]$. That is, $\Box^r\, T$ holds at Γ when T holds not only at Γ, but also at any other Γ^+ which renames to Γ.*

Definition 6. *We say that T is* renameable *whenever there is a function $ren^T : \forall[\ T \dot{\to} \Box^r\, T\ ]$. That is, whenever T holds at Γ, it also holds at any Γ^+ which renames to Γ.*

A renameable notion of value gives rise to a renameable notion of environment, essentially by renaming each contained value in an appropriate way. On the other side, environments admit renamings of their codomains in the opposite direction to that given by renameability.

Lemma 2 (ren^Env). *If* $(-) \overset{\mathcal{V}}{\models} A$ *is renameable for all* A*, then so is* $(-) \overset{\mathcal{V}}{\Longrightarrow} \Delta$ *for all* Δ*.*

Lemma 3. *From* $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$ *and* $\Delta \overset{\exists}{\Longrightarrow} \Theta$*, we get* $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Theta$*.*

Proof sketch. Notice that the lookup component of an environment maps variables in the codomain to values in the domain. We can apply the renaming to these variables.

6 Semantics

Given a $\mathcal{V}$-environment $\Gamma \Rightarrow \Delta$, the function semantics we define in this section assigns a $\mathcal{C}$-value in context Γ to every term in context Δ, where $\mathcal{C}$ is an OpenFam being the carrier of the semantic interpretation of terms ($\mathcal{V}$ being the semantic interpretation of variables). Before we can define semantics, we need to treat recursion through rules' premises (section 6.1) and extension of environments when going under variable binders (section 6.2).

6.1 A layer of syntax is functorial

A basic property of the universe of syntaxes we described in section 4 is that every syntax supports a functorial action on subterms, realised by the function map-s. Its type says that to map a function f over a layer of syntax, there must be a linear map F relating the domain and codomain usage contexts, and f should be usable wherever the domain and codomain usage contexts are similarly related by F.

```
map-s : (s : System) →
  (∀ {Θ P′ Q′} → F .rel P′ Q′ → ∀[ X Θ (ctx P′ γ) ·→ Y Θ (ctx Q′ δ) ]) →
  (∀ {P Q} → F .rel P Q → ∀[ ⟦ s ⟧s X (ctx P γ) ·→ ⟦ s ⟧s Y (ctx Q δ) ])
```

This generality is needed because usage contexts change between a term and its immediate subterms—they are decomposed according to the bunched connectives used in the rules. X and Y are ExtOpenFams, with Θ being the context extension for a subterm (i.e., the variables newly bound in that subterm). Unlike usage annotations, types in the contexts γ and δ, and the conclusion types implicit here, are preserved throughout. This is the essence of the usage annotation based approach—we use traditional techniques for variable binding, with the usage annotations layered on top.

The heart of map-s is map-p, which recursively works through the structure ps of premises of the rule applied, acting on each subterm it finds. Here, particularly

in the clauses for $`*$ and $`\cdot$, we see why it is not enough for the function on subterms to apply at usage contexts P and Q — rather, it also needs to apply at any similarly related P' and Q'. In the case of $`*$, we have that $\mathcal{P} \leq \mathcal{P}_M + \mathcal{P}_N$, with M and N being collections of subterms in usage contexts $\mathcal{P}_M$ and $\mathcal{P}_N$, respectively. Linearity of F yields $\mathcal{Q}_M$ and $\mathcal{Q}_N$ such that $\mathcal{Q} \leq \mathcal{Q}_M + \mathcal{Q}_N$ and we use map-p recursively at $(\mathcal{P}_M, \mathcal{Q}_M)$ and $(\mathcal{P}_N, \mathcal{Q}_N)$ on M and N. The cases for $`\cdot$ and $`I^*$ are similar, each using a different aspect of linearity. In contrast, the cases for $`\dot{\mathsf{i}}$ and $`\dot{\times}$, which are the only constructors used in fully structural systems, do not involve any changes in the usage contexts.

```
map-p : (ps : Premises) →
  (∀ {Θ P′ Q′} → F .rel P′ Q′ → ∀[ X Θ (ctx P′ γ) ⇸ Y Θ (ctx Q′ δ) ]) →
  (∀ {P Q} → F .rel P Q → ⟦ ps ⟧p X (ctx P γ) → ⟦ ps ⟧p Y (ctx Q δ))
map-p ⟨ Γ `⊢ A ⟩ f r M                = f r M
map-p `i f r _                        = _
map-p (ps `× qs) f r (M , N)          = map-p ps f r M , map-p qs f r N
map-p `I* f r I*⟨ sp0 ⟩                = I*⟨ F .rel-0ₘ (sp0 , r) ⟩
map-p (ps `* qs) f r (M *⟨ sp+ ⟩ N) =
  let rM ↘, sp+′ ,↙ rN = F .rel-+ₘ (sp+ , r) in
  map-p ps f rM M *⟨ sp+′ ⟩ map-p qs f rN N
map-p (p `· ps) f r (⟨ sp* ⟩· M)       =
  let r′ , sp*′ = F .rel-*ₘ (sp* , r) in
  ⟨ sp*′ ⟩· map-p ps f r′ M
```

6.2 The Kripke function space

At this point we introduce a minor generalisation to OpenFam and ExtOpenFam: I —OpenFam and I —ExtOpenFam. We obtain the definition of I —OpenFam by replacing the textual occurrence of Ty by the parameter I.

The definition Kripke $\mathcal{V}\,\mathcal{C}\,\Delta$ is a kind of function space that describes a $\mathcal{C}$ value parametrised by Δ-many additional $\mathcal{V}$s (all correctly typed and usage annotated). It is used to describe how to go under binders in a Higher-Order Abstract Syntax style—to go under a binder we must provide semantic interpretations for all the additional variables:

```
Kripke : (𝒱 : OpenFam v) (𝒞 : I —OpenFam c) → I —ExtOpenFam _
Kripke = Wrap λ 𝒱 𝒞 Δ Γ A → □ʳ ([ 𝒱 ]_⇒ᵉ Δ -∗ᶜ [ 𝒞 ]_⊨ A) Γ
```

Wrap is a device that turns any type family into an equivalent type family that is judgementally injective in its indices, which helps with Agda's type inference. It turns the type family into a parametrised record with a single field get whose type is the type in the body of the λ-abstraction. For understanding the meaning of Kripke, Wrap can be ignored.

If Δ is of the form $s_1 B_1, \ldots, s_n B_n$, then Kripke $\mathcal{V}\ \mathcal{C}\ \Delta\ \Gamma\ A$ is equivalent to $\Box^r\ (s_1 \cdot^c [\ \mathcal{V}\ ]_\vDash B_1 \mathbin{-\!\!*}^c \cdots \mathbin{-\!\!*}^c s_n \cdot^c [\ \mathcal{V}\ ]_\vDash B_n \mathbin{-\!\!*}^c [\ \mathcal{C}\ ]_\vDash A)\ \Gamma$ by Currying. That is to say, the Kripke function is expecting a value for each newly bound variable,

at the multiplicity of its annotation, together with the resources supporting each of those values. We use the "magic wand" function space here to enforce the invariant that the freshly bound variables have usage annotations that are added to the existing variables, not shared with them. The use of the $\Box^r$ modality ensures that we can still use it in the presence of additional variables introduced by weakening.

Kripke is functorial in the $\mathcal{C}$ argument, as witnessed by the mapKC function, which is essentially post-composition:

```
mapKC : ∀ {A B} → ∀[ [ C ]⊨ A ⇸ [ C′ ]⊨ B ] →
          ∀ {Δ Γ} → Kripke V C Δ Γ A → Kripke V C′ Δ Γ B
mapKC f b .get ren .app* sp ρ = f (b .get ren .app* sp ρ)
```

6.3 Semantic traversal

We can now state the data required to implement a traversal assigning semantics to terms. For open families $\mathcal{V}$ and $\mathcal{C}$, interpreting variables and terms respectively, we assume that $\mathcal{V}$ is renameable, that $\mathcal{V}$ is embeddable in $\mathcal{C}$, and that we have an algebra for a layer of syntax, where bound variables are handled using the Kripke function space:

```
record Semantics (d : System) (V : OpenFam v) (C : OpenFam c)
       : Set (suc 0ℓ ⊔ v ⊔ c) where
  field
    ren^V : ∀ {A} → Renameable ([ V ]⊨ A)
    ⟦var⟧ : ∀[ V ⇸ C ]
    ⟦con⟧ : ∀[ ⟦ d ⟧s (Kripke V C) ⇸ C ]
```

We mutually define the action semantics and its lemma body. The purpose of semantics is to turn a term into a $\mathcal{C}$-value using a $\mathcal{V}$-environment and the fields of Semantics. Meanwhile, body does a similar job, but also deals with newly bound variables. In particular, body takes a term in a context extended by Θ, and produces a Kripke function from $\mathcal{V}$-values for Θ to $\mathcal{C}$-values.

```
semantics : ∀ {Γ Δ} → [ V ] Γ ⇒ᵉ Δ → ∀ {sz} →
  ∀[ [ d , sz ] Δ ⊢_ ⇸ [ C ] Γ ⊨_ ]
body : ∀ {Γ Δ} → [ V ] Γ ⇒ᵉ Δ → ∀ {sz Θ} →
  ∀[ Scope [ d , sz ]⊢_ Θ Δ ⇸ Kripke V C Θ Γ ]
```

To implement the new recursor semantics, we use the standard recursor, which in one case gives us a variable v, and in the other gives us a structure of subterms M, each of which is in an extended context. To deal with a variable v, we look it up in the environment ρ, then use the ⟦var⟧ field to map the resulting $\mathcal{V}$-value to a $\mathcal{C}$-value. To deal with a structure of subterms M, we use the functoriality of the syntactic structure to consider each subterm separately. On a subterm, we apply body, which amounts to a recursive call to semantics with an extended

environment. Recall that relocate (lemma 1) adjusts the environment ρ to work in the usage contexts of the subterms.

```
semantics ρ ('var v) = ⟦var⟧ $ ρ .lookup (ρ .fit-here) v
semantics ρ ('con M) = ⟦con⟧ $
  map-s (ρ .Ψ) d (λ r → body (relocate ρ r)) (ρ .fit-here) M
```

For body, we are given a subterm M, to which we want to apply semantics. To do so, we need an extended version of the initial environment ρ. We express this as the generation of a Kripke function that produces the extended environment given interpretations of the fresh variables. We take ρ, which is an environment covering Δ, and σ, which is an environment covering Θ, and glue them together using the inductive rules for generating environments, after having renamed ρ via lemma 2 to make it fit the new context Γ^+ (intended to be $\Gamma +\!\!+^c \Theta$):

```
extend : ∀ {Γ Δ Θ} →
  [ 𝒱 ] Γ ⇛ᵉ Δ → Kripke 𝒱 ([ 𝒱 ]_⇛ᵉ_) Θ Γ (Δ ++ᶜ Θ)
extend ρ .get ren .app* sp σ = ++ᵉ (ren^Env ren^𝒱 ρ ren *⟨ sp ⟩ σ)
```

To define body, we use mapK$\mathcal{C}$ to post-compose the environment extension by the λ-function taking an extended environment and acting with it on M.

```
body ρ M = mapK𝒞 (λ σ → semantics σ M) (extend ρ)
```

7 Example traversals

In this section, we provide three example uses of semantic traversals: generic renaming and substitution, a usage elaborator, and a denotational semantics. The reader is also encouraged to see the far greater range of examples in the work of Allais et al. [3], which should adapt to our usage-annotated setting. Renaming and substitution are essential results, while the latter two examples focus on usage annotations.

A result we will use throughout this section is *reification*. When we have an index-preserving mapping from usage-checked variables to $\mathcal{V}$-environments, we can construct environments of the form $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Gamma$ (identity environments) for all Γ. This lets us write the reify function, which simplifies our obligations in giving a Semantics by coercing Kripke functions into just $\mathcal{C}$-values in an extended context.

Lemma 4 (reify). *If $\mathcal{V}$ is an open family such that there is a function $v : \forall[\ \exists \dot{\to} \mathcal{V}\ ]$, we get a function of type $\forall[\ \mathrm{Kripke}\,\mathcal{V}\,\mathcal{C} \dot{\to} \mathrm{Scope}\,\mathcal{C}\ ]$ for any $\mathcal{C}$.*

Proof. Let $b : \mathrm{Kripke}\,\mathcal{V}\,\mathcal{C}\,\Delta\,\Gamma\,A$. That is, b is a Kripke function yielding $\mathcal{C}$-computations We want to apply b so as to get a $\mathcal{C}\,(\Gamma, \Delta)\,A$. Let $\mathcal{P}\gamma = \Gamma$ and $\mathcal{Q}\delta = \Delta$. The $\Box^r$ in the type of b allows us to reverse-rename Γ to $\Gamma, 0\delta$. Then we give the $\multimap$-function an argument in context $0\gamma, \Delta$, noting that $(\Gamma, 0\delta)+(0\gamma, \Delta) =$

(Γ, Δ), as we wanted from the result. The argument needs type $0\gamma, \Delta \overset{\mathcal{V}}{\Longrightarrow} \Delta$. We produce this via lemma 3 from an environment $\rho : 0\gamma, \Delta \overset{\mathcal{V}}{\Longrightarrow} 0\gamma, \Delta$ created using v and a renaming which is the complement to that used on $\square^r$.

All of the $\mathcal{V}$s used in examples in this paper support identity environments. However, Allais et al. [3, p. 27] give some important examples that do not support identity environments, and thus cannot use reify (lemma 4). The feature that causes the lack of support for identity environments is that a semantics can make use of the fact that only variables of particular kinds are bound by the syntax. In the examples of Allais et al., a bidirectionally typed language only binds variables that synthesise their type, as opposed to those whose type is checked. The semantics of type-checking and elaboration rely on variables synthesising their type, so $\mathcal{V}$ is chosen to cover only those variables. Instead of using reify, we must observe that each syntactic form only binds such synthesising variables. Similar phenomena would appear in, say, a call-by-value language where variables are values (not computations), or a polarised language where variables always have a polarity matching their type.

7.1 Renaming and substitution

In an unpublished note, McBride [15] gives a parametrised traversal yielding homomorphisms of syntax, the canonical examples of which are simultaneous renaming and simultaneous substitution. The parameters are collected in the record Kit. We make a minor change to the original presentation, where instead of our ren^$\mathcal{V}$ field, McBride has the field wk allowing only context extensions. As for the other two fields, vr allows us to map variables to $\mathcal{V}$-values, so as to put newly bound variables in environments; and tm allows us to extract terms from $\mathcal{V}$-values, as required when we use the environment to handle a free variable.

```
record Kit (d : System) (𝒱 : OpenFam v) : Set (suc 0ℓ ⊔ v) where
  field
    ren^𝒱 : ∀ {A} → Renameable ([ 𝒱 ]⊨ A)
    vr    : ∀[ _∋_ ⇾ 𝒱          ]
    tm    : ∀[ 𝒱   ⇾ [ d , ∞ ]⊢_ ]
```

Where McBride gave the traversal explicitly, we go via our generic semantic traversal. The first two fields of Semantics derive directly from fields of Kit. Meanwhile, to handle term constructors, we first reify to get a collection of traversed subterms, and then use ‘con to assemble these subterms into a similarly shaped syntactic form as we started with. The vr field is used implicitly in reify, as it is used to show that $\mathcal{V}$-identity environments exist.

```
kit→sem : Kit d 𝒱 → Semantics d 𝒱 [ d , ∞ ]⊢_
kit→sem K .ren^𝒱 = K .ren^𝒱
kit→sem K .⟦var⟧ = K .tm
kit→sem {d = d} K .⟦con⟧ = ‘con ∘ map-s′ d reify
  where open Kit K using (identityEnv)
```

The action of a syntactic traversal on logical rules is basically fixed: we preserve the logical rule and extend the environment with any newly bound variables according to vr. Meanwhile, the action on variables is relatively unconstrained: we look up the variable in the environment to get a $\mathcal{V}$-value, then transform that $\mathcal{V}$-value into a term using tm.

The idea of simultaneous renaming is that variables replace variables, whereas with simultaneous substitution, terms replace variables. This translates to environments for renaming containing $\exists$-values (variables), and environments for substitution containing $\vdash$-values (terms).

```
Ren-Kit : Kit d _∃_
Ren-Kit = record { ren^𝒱 = ren^∃ ; vr = id ; tm = `var }
```

Notice that ren^$\vdash$, witnessing the fact that terms are renameable, is a corollary of Ren-Kit.

```
Sub-Kit : Kit d [ d , ∞ ]_⊢_
Sub-Kit = record { ren^𝒱 = ren^⊢ ; vr = `var ; tm = id }
```

7.2 A usage elaborator

Using the constructs we have seen so far, producing example terms soon becomes extremely tedious. We can achieve some abbreviation by using pattern synonyms to wrap around `con expressions, but we still have to produce essentially bespoke proofs whenever we use a usage-sensitive part of the syntax. The size of each of these proofs is roughly proportional to the number of free variables, so the amount of proof we have to write grows roughly quadratically with the size of terms. An additional factor, which we can't see on paper, is that type checking time for these proofs soon becomes prohibitive to interactive development.

Our aim in this subsection is to automate usage constraint proofs, making terms both easier to write and more performant to check. We invoke the automation by writing terms in a syntax where usage constraints have been trivialised, and then use a semantic traversal over the simplified syntax to try to produce a fully elaborated term in the original syntax. We write the automation in a way that is generic in the syntax description, thus avoiding repetition and facilitating the prototyping of new type systems.

The type of syntax descriptions depends on the type of usage annotations because of variable binding. For example, in the !r-E rule of figure 2, the right premise binds a new variable with annotation r, where r is drawn from the ambient posemiring. The scaling combinator also makes direct reference to the posemiring. To produce a simplified syntax description, where usage constraints are trivialised, we set the ambient posemiring to the 1-element **0** posemiring. In contrast to syntax descriptions, even though types can contain usage annotations, the type of types does not depend on the type of usage annotations. This means that, in our simplified syntax, terms have types from the original system

even though variables have trivial usage annotations. We define the **0** posemiring as follows, being careful to use the 0-field record type ⊤ so that everything algebraic gets solved by Agda's η-laws. Indeed, in this very definition, all of the semiring operations and laws are canonically inferred.

```
0-poSemiring : PoSemiring 0ℓ 0ℓ 0ℓ
0-poSemiring = record
  { Carrier = ⊤; _≈_ = λ _ _ → ⊤; _≤_ = λ _ _ → ⊤ }
```

The elaboration process is monadic. In particular, we use the List/nondeterminism monad to give *all* of the possible annotation choices on the free variables of a term. We believe the commitment to multiple solutions is inherent when the syntax contains `i. For example, in the intermediate stages of elaborating $(\vdash \lambda x.\ (*, *)) : A \multimap \top \otimes \top$ with a usage counting posemiring (assuming reasonable rules for $\top$ and $\otimes$), it is unclear whether to use the variable x in the left $*$ or the right $*$. This uncertainty should be reflected in the final result.

The non-deterministic choices we make during elaboration are enumerated by the fields of NonDetInverses. These choices are driven by the typing rules and a candidate usage vector for the conclusion. For example, $+^{-1}\ r$ is needed when we encounter a `* in the syntax and the candidate usage annotation we are considering is r. Then, $+^{-1}\ r$ is a list of pairs of annotations p and q that r can split into, together with a proof of the splitting. For $0\#^{-1}$ and $1\#^{-1}$, inverses to constants, we are given the candidate r and typically return an empty list if the constraint cannot be satisfied, or a singleton list containing a proof. $*^{-1}$ is used when we encounter scaling, in which case we know both the scaling factor r (from the syntax description) and the candidate q. These inverse operations combine monadically (in fact, applicatively) to give inverses to the vector operations of zero, addition, scaling, and basis.

```
record NonDetInverses : Set where
  field
    0#⁻¹ : (r : Ann) → List (r ≤ 0#)
    +⁻¹ : (r : Ann) → List (∃ \ ((p , q) : _ × _) → r ≤ p + q)
    1#⁻¹ : (r : Ann) → List (r ≤ 1#)
    *⁻¹ : (r q : Ann) → List (∃ \ p → q ≤ r * p)
```

We choose the $\mathcal{V}$ of our semantics to be (unannotated) variables. For the $\mathcal{C}$, we consider *functions* from candidate usage vectors R to the list of elaborated derivations with usage annotations given by R. The protocol this encodes is that the user will provide an unannotated term together with a candidate usage context R, and usage elaboration returns a list of possible ways the term could be annotated such that the conclusion has usage context R. The module name U refers to the fact that we are taking the ambient posemiring to be **0** in OpenFam. The effect on OpenFam is that the usage annotations of any contexts we consider are uninformative (hence the _ on the left).

```
𝒞 : System → U.OpenFam _
𝒞 sys (U.ctx _ γ) A = ∀ R → List ([ sys , ∞ ] ctx R γ ⊢ A)
```

To traverse the unannotated terms, we produce a Semantics over the unannotated system uSystem sys. To write it, we make use of idiom brackets $(\!| \ldots |\!)$, which have the effect of replacing top-level spines of applications by (List-)applicative applications. Field by field, we already know that variables are renameable. To interpret a variable, we consider all the possible proofs that such a variable could be well annotated, and package them up as a variable term via the applicative machinery. Finally, for compound terms, we first reify the unannotated subterms, and then combine the subterms via a lemma.

```
elab-sem : ∀ sys → U.Semantics (uSystem sys) U._∃_ (𝒞 sys)
elab-sem sys .ren^𝒱 = U.ren^∃
elab-sem sys .⟦var⟧ (U.lvar i q _) R =
  (| 'var (| (lvar i q) (⟨ i |⁻¹ R) |) |)
elab-sem sys .⟦con⟧ b R =
  let rb = U.map-s′ (uSystem sys) U.reify b in
  (| 'con (lemma sys rb) |)
```

The lemma essentially goes through the shape of the premises, combining the collections of subterms in the natural way. For example, at each $\_\dot{\times}\_$, we take the Cartesian product of the possibilities of each half, and at each $\_*\_$, we non-deterministically split the usage annotations coming in, and then take the Cartesian product. When it comes to newly bound variables, the *syntax description* tells us their annotations, so there is no further non-determinism introduced here.

```
lemma : ∀ (sys : System) {A Γ} →
  U.⟦ uSystem sys ⟧s (U.Scope (𝒞 sys)) (uCtx Γ) A →
  List (⟦ sys ⟧s (Scope [ sys , ∞ ]⊢_) Γ A)
```

To actually use elab-sem on terms, we take the associated semantics and pass it the identity environment (an identity *renaming* in this case, because $\mathcal{V}$ is a family of variables). We use elab-unique, which further checks statically that exactly one derivation is returned. The candidate usage vector R will be [] for closed terms, and otherwise we have to supply the intended usage annotations.

We can now use the elaborator to automatically infer the usage annotations for the example at the end of section 4.2. This allows us to write:

```
cojoin-!ω : ∀ {A} → [ λR , ∞ ] []ᶜ ⊢ (! ω# A ⊸ ! ω# (! ω# A))
cojoin-!ω = elab-unique _ (⊸I (!E (var# 0) (!I (!I (var# 1))))) []
```

We have instantiated the usage elaborator so that: $0\#^{-1}$ is a singleton on 0 and ω, and empty on 1; $1\#^{-1}$ is a singleton on 1 and ω, and empty on 0; $+^{-1}$ gives $0 \mapsto [(0,0)]$, $1 \mapsto [(0,1),(1,0)]$, and $\omega \mapsto [(\omega,\omega)]$; and $*^{-1}$ gives $(\omega,0) \mapsto [0]$, $(\omega,1) \mapsto []$, and $(\omega,\omega) \mapsto [\omega]$ (omitting $(0,\_)$ and $(1,\_)$ cases for brevity). Note that we do not consider splitting ω up as, say, $1+\omega$, because this splitting would introduce more non-determinism but not allow any more terms to be typed. As such, the only non-determinism comes when we have variables annotated 1 and

need to do an additive split, like when we apply the !E rule below. At this point, the variable can become either 0-annotated in the left subterm and 1-annotated on the right, or vice-versa. We will find that, because the left subterm wants to use that variable, the former choice will be rejected. The function var# is a convenience for converting statically known natural numbers, representing de Bruijn *levels*, into variable terms.

7.3 A denotational semantics

To justify the name *semantics*, we give an example traversal that is a denotational semantics in the usual sense. The semantics we take is a refinement of that of Abel and Bernardy [2], which gives a way to extract parametricity theorems from substructurally typed programs. Example theorems are that all linear terms act as permutations on some fixed set of resources, and all monotonically typed terms are really monotonic in the way the typing suggests they are.

To abbreviate this section, we use a simplified syntax compared to $\lambda\mathcal{R}$. We allow for an arbitrary family of base types $BaseTy$, and a single type former $(r\ ,\ A) \multimap B$, equivalent to $(!\ r\ A) \multimap B$ from the earlier system.

```
data Ty : Set where
  base : BaseTy → Ty
  _—∘_ : (rA : Ann × Ty) (B : Ty) → Ty
```

In the term syntax, λ-abstraction now binds a variable with annotation r, while application needs to scale its argument by r (both in accordance with the function type they are acting on).

```
data 'AnnArr : Set where
  'lam 'app : (rA : Ann × Ty) (B : Ty) → 'AnnArr

AnnArr : System
AnnArr = 'AnnArr ▷ λ where
  ('lam rA B) → ⟨ [ rA ]ᶜ '⊢ B ⟩ ⟹ rA —∘ B
  ('app rA@(r , A) B) → ⟨ []ᶜ '⊢ rA —∘ B ⟩ '* r '· ⟨ []ᶜ '⊢ A ⟩ ⟹ B
```

As a running example, we take the usage annotations to be the 4-element variance posemiring (example 2). We establish the property that all terms are monotonic in their free variables. This monotonicity can be covariant or contravariant (or neither or both) depending on the annotation of each free variable. This provides an additional example to those of Abel and Bernardy.

We will take semantics of this system into *world-indexed relations* [2, 5]. A world-indexed relation (WRel) over a poset of worlds W is a set over which we have a W-indexed binary relation satisfying a presheaf-like property with respect to the order on W. The Agda code for world-indexed relations and constructions on them can be found in Wood and Atkey [22].

Example 5. When W is the 1-element set, a world-indexed relation is just a set equipped with a binary relation.

Morphisms (WRelMor) between world-indexed relations R and S consist of a mapping between the underlying sets such that, at each fixed world w, the mapping preserves relatedness from R to S.

When the poset of worlds forms a (relational) commutative monoid, such world-indexed relations support a symmetric monoidal closed structure, with objects denoted I^R, $\_\otimes^R\_$, and $\_\multimap^R\_$. These reuse the bunched connectives I^*, $*$, and $-\!\!*$, now over worlds rather than contexts.

The final piece of semantics we need is a *bang* operator. We allow the semantic *bang* to be an arbitrary annotation-indexed functor on world-indexed relations. This functor must respect all of the structure on the indices, making it a graded comonad over multiplication, as well as being lax monoidal at any particular index r. These laws are listed in the `Generic.Linear.Example.WRel` module in [22].

Example 6. With W being the 1-element set and annotations coming from the variance semiring, we can define the following *bang*. It is always the identity on the set component, while the relation component consists of flipping the relation for contravariance and taking conjunctions to achieve both covariance and contravariance. When we want neither covariance nor contravariance, we use the always true predicate on worlds $\dot{\mathsf{1}}$.

$$
\begin{aligned}
&!^R : \mathsf{WayUp} \to \mathsf{WRel}\ \_\leq^w\_ \to \mathsf{WRel}\ \_\leq^w\_ \\
&!^R\ a\ R\ .\mathsf{set} = R\ .\mathsf{set} \\
&!^R\ \uparrow\uparrow\ R\ .\mathsf{rel} = R\ .\mathsf{rel} \\
&!^R\ \downarrow\downarrow\ R\ .\mathsf{rel}\ x\ y = R\ .\mathsf{rel}\ y\ x \\
&!^R\ ??\ R\ .\mathsf{rel}\ x\ y = \dot{\mathsf{1}} \\
&!^R\ \sim\sim\ R\ .\mathsf{rel}\ x\ y = R\ .\mathsf{rel}\ x\ y \mathbin{\dot{\times}} R\ .\mathsf{rel}\ y\ x \\
&!^R\ a\ R\ .\mathsf{subres}\ \_ = \mathsf{id}
\end{aligned}
$$

The semantics of a type is given by $[\![\_]\!]$, which maps into world-indexed relations. The function type is interpreted using $\multimap^R$ and $!^R$. Contexts are interpreted by $[\![\_]\!]^c$, using $\otimes^R$ and I^R. Terms are interpreted as morphisms by the open family $[\![\_\vdash\_]\!]$. Variables are interpreted by lookup^R (definition omitted).

$$\mathsf{lookup}^R : \forall\ \{\Gamma\ A\} \to \Gamma \ni A \to [\![\ \Gamma \vdash A\ ]\!]$$

Now we give a Semantics. The choice of $\mathcal{V}$ as $\_\exists\_$ is somewhat arbitrary, given that a standard denotational semantics would not use intermediate environments in the same sense as renaming and substitution, but it allows us to reuse the standard facts that variables support renaming and identity environments. With this choice of $\mathcal{V}$ and $\mathcal{C}$, we interpret environment entries by lookup^R. Meanwhile, for the logical rules, we ignore environments by using reify to just deal with morphisms in an extended context. As such, λ-abstractions are easy to interpret, while applications require some massaging to remove the extension by an empty context, followed by some plumbing to split the interpretation of the context

according to the usage constraints and feed the interpretation of the argument n' into the interpretation of the function m'.

```
Wrel : Semantics AnnArr _∃_ ⟦_⊢_⟧
Wrel .ren^𝒱 = ren^∃
Wrel .⟦var⟧ = lookup^R
Wrel .⟦con⟧ (`lam (r , A) B , ≡.refl , m) = curry^R (reify m)
Wrel .⟦con⟧ {ctx R γ}
  (`app (r , A) B , ≡.refl , _*⟨_⟩_ {P} {rQ} m sp+ (⟨_⟩·_ {Q} sp* n)) =
  let n' : WRelMor ⟦ ctx Q γ ⟧^c ⟦ A ⟧
      n' = reify n ∘^R ⊗^R-unit^r←
      m' : WRelMor (⟦ ctx P γ ⟧^c ⊗^R !^R r ⟦ A ⟧) ⟦ B ⟧
      m' = uncurry^R (reify m ∘^R ⊗^R-unit^r←) in
  m' ∘^R map-⊗^R id^R (!^R-map n' ∘^R ctx-* sp*) ∘^R ctx-+ sp+
```

Then, the semantics of terms is given by the function semantics Wrel 1^r, where 1^r is the identity renaming.

Example 7. We can make a subtraction function from primitive addition and negation on integers. Subtraction is covariant in its first argument and contravariant in its second argument. We give the definition in pseudocode, though it is also amenable to the usage elaborator of section 7.2, suitably instantiated.

$$\sim\sim p : \uparrow\uparrow\mathbb{Z} \multimap \uparrow\uparrow\mathbb{Z} \multimap \mathbb{Z}, \sim\sim n : \downarrow\downarrow\mathbb{Z} \multimap \mathbb{Z} \vdash minus : \uparrow\uparrow\mathbb{Z} \multimap \downarrow\downarrow\mathbb{Z} \multimap \mathbb{Z}$$
$$minus \coloneqq \lambda x.\ \lambda y.\ p\,x\,(n\,y)$$

After feeding in Agda's addition and negation functions as the interpretations of the free variables (noting that they are both monotonic in the required way), we get the following free theorem.

```
thm : x ℤ.≤ x' → y' ℤ.≤ y → x ℤ.+ (ℤ.- y) ℤ.≤ x' ℤ.+ (ℤ.- y')
```

8 Conclusions

We have presented a framework for doing metatheory for a class of substructural type systems in Agda. The framework gives us renaming, substitution, and a usage elaborator for new syntaxes for free, which we hope can facilitate prototyping and the mechanisation of more interesting semantic results. Beside the mechanised framework itself, we believe its methodology — the use of bunched premise combinators — can guide and simplify the development of (potentially unmechanised) substructural type systems.

Our account of substructurality is based on the linear algebraic principles described by Wood and Atkey [21]. However, these details only really affect the definition of environment, in which the use of linear maps is motivated by them being the standard notion of morphism between vectors. We could imagine that a similar notion of morphism is found for the kind of annotations found in Licata et al. [13], allowing a framework to consider finer substructural systems.

References

[1] Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A Core Calculus of Dependency. In: POPL '99, pp. 147–160 (1999)

[2] Abel, A., Bernardy, J.P.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020), https://doi.org/10.1145/3408972, URL https://doi.org/10.1145/3408972

[3] Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. J. Funct. Program. **31**, e22 (2021), https://doi.org/10.1017/S0956796820000076, URL https://doi.org/10.1017/S0956796820000076

[4] Atkey, R.: The syntax and semantics of quantitative type theory. In: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom (2018), https://doi.org/10.1145/3209108.3209189

[5] Atkey, R., Wood, J.: Context constrained computation. In: 3rd Workshop on Type-Driven Development (TyDe '18), Extended Abstract (2018)

[6] Benton, P.: A mixed linear and non-linear logic: Proofs, terms and models. pp. 121–135, Springer-Verlag (1994)

[7] Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A Core Quantitative Coeffect Calculus. In: ESOP 2014, pp. 351–370 (2014)

[8] Curien, P.L., Herbelin, H.: The duality of computation. SIGPLAN Not. **35**(9), 233–243 (Sep 2000), ISSN 0362-1340, https://doi.org/10.1145/357766.351262, URL https://doi.org/10.1145/357766.351262

[9] Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158), pp. 193–202 (1999), https://doi.org/10.1109/LICS.1999.782615

[10] Ghica, D.R., Smith, A.I.: Bounded linear types in a resource semiring. In: ESOP 2014, pp. 331–350 (2014)

[11] Girard, J.Y.: Linear logic. Theor. Comp. Sci. **50**, 1–101 (1987)

[12] Herbelin, H.: C'est maintenant qu'on calcule, au cœur de la dualité. Habilitation (2005)

[13] Licata, D.R., Shulman, M., Riley, M.: A fibrational framework for substructural and modal logics. In: FSCD 2017, pp. 25:1–25:22 (2017), https://doi.org/10.4230/LIPIcs.FSCD.2017.25

[14] Lovas, W., Crary, K.: Structural normalization for classical natural deduction (2006)

[15] McBride, C.: Type-preserving renaming and substitution (2005), URL http://www.strictlypositive.org/ren-sub.pdf

[16] O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. BULLETIN OF SYMBOLIC LOGIC **5**(2), 215–244 (1999)

[17] Orchard, D.A., Liepelt, V., Eades, H.: Quantitative program reasoning with graded modal types. Proceedings of the ACM on Programming Languages **3** (June 2019)

[18] Petricek, T., Orchard, D.A., Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: ICFP 2014, pp. 123–135 (2014)
[19] Reed, J., Pierce, B.C.: Distance makes the types grow stronger. In: Hudak, P., Weirich, S. (eds.) ICFP 2010, pp. 157–168 (2010)
[20] Rouvoet, A., Bach Poulsen, C., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: CPP 2020, pp. 284–298 (2020), ISBN 9781450370974, https://doi.org/10.1145/3372885.3373818
[21] Wood, J., Atkey, R.: A linear algebra approach to linear metatheory. arXiv preprint arXiv:2005.02247 (2020)
[22] Wood, J., Atkey, R.: lamudri/generic-lr (Jan 2022), https://doi.org/10.5281/zenodo.5920051, URL https://doi.org/10.5281/zenodo.5920051