CARVe in Rocq: A Library for Substructural Meta-Theory

Daniel Zackon
McGill University, Montreal, Canada

Ryan Kavanagh
Université du Québec à Montréal, Montreal, Canada

Alberto Momigliano
Università degli Studi di Milano, Milan, Italy

Brigitte Pientka
McGill University, Montreal, Canada

Abstract

Substructural logics and languages are central to the semantics of resource-sensitive computation, yet their mechanization in a proof assistant remains technically demanding. Existing formalizations of substructural calculi often depend on ad hoc, system-specific infrastructure, making them difficult to extend and limiting reuse across developments. As a consequence, mechanizing even routine meta-theoretic proofs in substructural settings require substantial engineering effort.

This paper introduces a new Rocq library aimed at closing this methodological gap. Building on the CARVe ('Contexts as Resource Vectors') framework first implemented in Beluga, our design represents resource usage algebraically rather than structurally. This approach integrates smoothly with well-scoped de Bruijn syntax, eliminating much of the overhead associated with explicit context manipulation. By abstracting over resource algebras, the library captures within a single, uniform development the common structure underlying a wide range of substructural disciplines, including more abstract formulations such as adjoint systems. The library provides reusable machinery for core operations like context splitting and resource consumption, together with generic lemmas that apply uniformly across applications. We demonstrate the expressiveness and extensibility of the framework through several case studies covering diverse substructural calculi, showing that the approach enables more modular mechanizations with reduced proof overhead.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Logic and verification; Theory of computation → Proof theory

Keywords and phrases proof assistants, meta-theory, substructural type systems, linear logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.

Supplementary Material https://github.com/dzackon/carve-in-rocq

Acknowledgements The authors thank Kathrin Stark for useful conversations on the Rocq implementation.

1 Introduction

The structural rules of weakening, contraction, and exchange govern how assumptions in a logical system may be duplicated, discarded, or reordered. Omitting or relaxing these rules gives rise to logics such as linear, affine, or relevant logic, and to corresponding resource-sensitive programming languages. These logics and languages are collectively called *substructural*, and play an increasingly central role in the study of resource-sensitive computation across a wide range of applications, including the foundations of concurrency [13, 46], quantum computing [42], and memory management [28, 10].

Unlike the mechanization of purely structural type systems, where trade-offs between representations and proof techniques are comparatively well-understood, the landscape for substructural systems remains far less settled. Indeed, the Concurrent Calculi Formalisation Benchmark [15] identifies the mechanization of linearity as a central challenge problem for modern proof assistants. Although many mechanizations of substructural calculi exist (see the review in [48]) they usually rely on ad hoc system-specific infrastructure. While effective in isolation, these developments can be difficult to reuse or generalize. In recent years, several efforts have sought to provide a more principled foundation for formalizing substructural type systems [16, 47, 48].

Because substructural languages treat assumptions as resources rather than freely reusable hypotheses, mechanizing their meta-theory usually demands careful management of contexts of assumptions. Notably, typing rules often require splitting a context into disjoint parts—each allocated to a different subterm—so that no assumption is used more than permitted. When working with de Bruijn representations of syntax, one must carefully re-index variables after each such manipulation, which can substantially complicate formalization. This difficulty is compounded when reasoning about simultaneous substitutions, as required by many logical relations proofs, since the substitutions must be updated alongside the contexts.

In the linear setting, Rouvoet et al. [40] avoid such re-indexing of de Bruijn indices using *co-de Bruijn* syntax [2, 29]. This representation makes it possible to capture linearity constraints directly in the term structure. However, it is not clear whether co-de Bruijn syntax scales cleanly to richer substructural systems, and the resulting syntax is generally even less readable than standard de Bruijn representations. By contrast, using standard well-scoped de Bruijn representations [4] builds on established best practices and take advantage of robust libraries like Autosubst [44] for mechanized developments.

All these considerations have motivated the design of CARVe ("Contexts as Resource Vectors") [48], a general framework for representing and manipulating substructural contexts in mechanizations. The framework was originally implemented in Beluga [35, 36], a dependently-typed proof environment built on the logical framework LF. Building on an idea by (among others) Schack-Nielsen and Schürmann [41], CARVe annotates each assumption with an element of some *resource algebra*. These tags encode information about an assumption's availability, such as whether it may be used exactly once, multiple times, or not at all. Instead of physically restructuring a context when assumptions are used or distributed across subterms, CARVe updates their tags to reflect the new usage status. As a consequence, contexts remain structurally intact across proofs. This design integrates naturally with well-scoped de Bruijn representations of syntax, since the indices and scoping discipline are preserved automatically.

This paper presents `rocq-carve`, a new implementation of CARVe as a library in Rocq, providing a uniform and accessible framework for mechanizing substructural systems. Implementing CARVe in Rocq opens the door to broader practical use by the proof assistant's large and active user and developer community, and to integration with Rocq's broad ecosystem of libraries for more complex mechanizations. Indeed, Rocq's rich type system enables the formalization of meta-theoretic properties using CARVe that were previously difficult or impossible to capture in Beluga. At the same time, Rocq supports a more modular, reusable CARVe library, which can be readily instantiated with different resource algebras; `rocq-carve` supports parameterization by both concrete and generic resource algebras, the properties of which lift directly to contexts represented as either lists or maps. We illustrate the library's use through a weak-normalization proof for a generic substructural λ-calculus, and show how the framework can be reused for a range of other substructural calculi.

2 CARVe

2.1 Structural substructural contexts

Let us begin by considering a fragment of the linear λ-calculus:

$$\frac{}{x : T \vdash x : T}\,(\text{HYP}) \qquad \frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \lambda x : S.e : S \multimap T}\,(\multimap\text{I})$$

$$\frac{\Gamma_1 \vdash e_1 : S \multimap T \qquad \Gamma_2 \vdash e_2 : S}{\Gamma_1 \,;\, \Gamma_2 \vdash e_1\, e_2 : T}\,(\multimap\text{E})$$

The substructural requirement that each assumption be used exactly once makes it necessary to define an operation $\Gamma_1 \,;\, \Gamma_2$ for merging two disjoint contexts. Defining "$;$" as mere concatenation is insufficient in the absence of an explicit exchange rule. Accordingly, the operation may interleave assumptions arbitrarily and is inherently non-deterministic. From a mechanization perspective, a major consideration therefore is compatibility with nameless representations of syntax, such as de Bruijn indices. When contexts are split, all variables indices must be adjusted accordingly, and when assumptions are consumed, the remaining elements must be shifted to preserve correct scoping. This bookkeeping introduces significant complexity into the formalization of substructural type systems.

CARVe [48] adopts an alternative "no-splitting" approach: the central principle guiding its design is to keep contexts "structurally intact" when they are split or their elements are consumed. Changes in resource availability are tracked through *multiplicity* tags α on context elements:

$$\Delta ::= \cdot \mid \Delta, x : T^{\alpha}$$

The resource annotations are drawn from a user-defined *resource algebra*. This consists of a set A of multiplicities together with

1. a join relation $\mathsf{Join} \subseteq (A \times A) \times A$, suggestively written $a \circ b = c$; and
2. a relation $\mathsf{hasProp} \subseteq \{W, C\} \times A$ associating multiplicities to structural properties.

The join relation governs how resources may be combined or distributed; it is lifted to contexts to define a (deterministic, provided $\circ$ is functional) context merge relation, or viewed differently, a (potentially non-deterministic) context split relation.

$$\frac{}{\cdot \bowtie \cdot = \cdot} \qquad \frac{\Delta_1 \bowtie \Delta_2 = \Delta \qquad \alpha_1 \circ \alpha_2 = \alpha}{(\Delta_1, x : T^{\alpha_1}) \bowtie (\Delta_2, x : T^{\alpha_2}) = \Delta, x : T^{\alpha}}$$

Variable names are included here for readability but may be omitted. While T denotes here an object-level type, it may range over any kind of resource.

The second component $\mathsf{hasProp}$ provides a characterization of the structural properties P supported by the elements of A. These structural properties can be enforced at the level of judgments about resources—for example, in typing rules.

The algebraic structure ensures that operations such as context splitting and tag updating are well-behaved by construction: properties of the chosen algebra lift automatically to the context level. It is often beneficial to assume that the structure satisfies additional algebraic properties, such as commutativity and associativity, and that a partial unit $|a|$ exists for each $a \in A$. This modularity allows CARVe to serve as a general-purpose framework for mechanizing various substructural type systems without requiring modifications to the core context machinery.

The following table summarizes the basic context-level operations and properties that CARVe treats as primitives:

Property	Notation	Description
Merge	$\Delta_1 \bowtie \Delta_2 = \Delta$	Δ is the result of merging contexts Δ_1 and Δ_2
Look-up	$x : T^\alpha \in \Delta$	x appears with resource T and multiplicity α in Δ
Update	$\Delta[x : S^\beta]$	Replace x's resource with S and multiplicity β in Δ
Map	$\mathsf{map}_{f,g}\Delta$	Apply f to each resource and g to each multiplicity in Δ
Renaming	$\mathsf{ren}_\rho\Delta$	Apply the renaming ρ to the variables of Δ
Property check	$P \in \Delta$	$\mathsf{hasProp}\, P\, \alpha$ for each $x : T^\alpha \in \Delta$

Remark that viewed as an operation, the join relation of a given algebra may be partial, depending on the substructural system under consideration. Its induced context-level merge operation may therefore also be partial. As reasoning about partial functions in proof assistants generally introduces unnecessary complexity, we opt to represent the join operation as a ternary relation (see also the discussion in [19]). Still, our approach remains agnostic to the particular implementation of the above context operations and primitives—they could be relational or functional—and indeed to the representation of contexts.

2.2 Linear λ-calculus, revisited

We may now revisit the linear λ-calculus, encoded using CARVe. As the resource algebra, we take the partial commutative monoid $\mathcal{L} := (\{0, 1\}, \bullet, 0)$, whose join operation $\bullet$ is defined by:

$\bullet$	0	1
0	0	1
1	1	—

The element 0 denotes an assumption that has been consumed or is otherwise irrelevant. When parametrizing a context with $\mathcal{L}$, linear assumptions (with tag 1) must be allocated to exactly one branch of a context split by construction, since $1 \bullet 1$ is undefined. We set $\mathsf{hasProp}\, P\, 0$ for both $P = W$ and $P = C$, while 1 satisfies neither.

In CARVe, the typing rules become:

$$\frac{x : T^1 \in \Delta \qquad \mathsf{W} \in \Delta[x : T^0]}{\Delta \Vdash x : T}\,(\textsc{hyp}) \qquad \frac{\Delta, x : S^1 \Vdash e : T}{\Delta \Vdash \lambda x : S.e : S \multimap T}\,(\multimap\textsc{I})$$

$$\frac{\Delta_1 \Vdash e_1 : S \multimap T \qquad \Delta_2 \Vdash e_2 : S \qquad \Delta_1 \bowtie \Delta_2 = \Delta}{\Delta \Vdash e_1\, e_2 : T}\,(\multimap\textsc{E})$$

The encoding of application and abstraction is straightforward. Because contexts remain structurally intact throughout derivations, the variable rule (HYP) requires checking (1) that x appears with type T and linear multiplicity 1 in Δ, and (2) that the context resulting from updating x's tag to 0 is "exhausted," in the sense that all the other assumptions are tagged as 0. This ensures that the context is effectively singleton at the point of use.

$$\Delta \Vdash x : T^{\alpha} \quad \begin{matrix}(3)\\ \neg\mathsf{zero}(\alpha)\end{matrix}$$

$$\frac{\Delta, x : S^{\alpha} \Vdash e : T^{\alpha}}{\Delta \Vdash \lambda x : S.e : (S \multimap T)^{\alpha}}\ (\multimap\text{I})$$

$$\frac{\Delta_1 \Vdash e_1 : (S \multimap T)^{\alpha} \qquad \Delta_2 \Vdash e_2 : S^{\alpha} \qquad \Delta_1 \bowtie \Delta_2 = \Delta}{\Delta \Vdash e_1\, e_2 : T^{\alpha}}\ (\multimap\text{E})$$

Figure 1 Generic substructural λ-calculus

2.3 Beyond linearity

CARVe contexts can be instantiated with different resource algebras to enforce different substructural constraints on resource usage. For illustration, consider the resource algebras corresponding to the four canonical substructural disciplines satisfying exchange:

Algebra	Elements	Join operation	Unit	Properties held
Linear: $\mathcal{L}$	$0, 1_L$	$0 \bullet 0 = 0,\ 1_L \bullet 0 = 0 \bullet 1_L = 1_L$	0	$W, C \in 0$
Affine: $\mathcal{A}$	$0, 1_A$	$0 \bullet 0 = 0,\ 1_A \bullet 0 = 0 \bullet 1_A = 1_A$	0	$W, C \in 0,\ W \in 1_A$
Relevant: $\mathcal{R}$	$0, \omega_R$	$0 \bullet 0 = 0,$ $\omega_R \bullet 0 = 0 \bullet \omega_R = \omega_R \bullet \omega_R = \omega_R$	0	$W, C \in 0,\ C \in \omega_R$
Structural: $\mathcal{S}$	ω_S	$\omega_S \bullet \omega_S = \omega_S$	ω_S	$W, C \in \omega_S$

In contrast to linear assumptions, fully structural assumptions (tagged ω_S) always persist across splits, while relevant assumptions (tagged ω_R) *may* do so. These algebras can be combined by standard constructions; for instance, the resource semantics of dual intuitionistic-linear logic [8] arises from the disjoint union of the structural and linear algebras.

In addition to encoding concrete substructural languages, CARVe supports the formulation of typing systems parametrically over a *class* of resource algebras. Consider the core rules of a generic substructural λ-calculus extending the linear example (Figure 1). We assume that each multiplicity α is equipped with a designated unit $|\alpha|$, and include in the (HYP) rule an assumption that α is non-zero to prevent irrelevant assumptions from being derivable. Specifically, $\mathsf{zero}(\alpha)$ indicates that α is the unrestricted unit of a multiplicity that fails to satisfy weakening or contraction.

If we parametrize these rules by the affine algebra $\mathcal{A}$, condition (1) becomes vacuous. Similarly, condition (2) is vacuous when using the linear algebra $\mathcal{L}$, and all of conditions (1)–(3) disappear under the structural algebra $\mathcal{S}$. For the relevant algebra $\mathcal{R}$, all three checks are required. In the application rule, the context-merge condition also becomes vacuous under $\mathcal{S}$. In this way we recover the CARVe-style typing rules for the respective single-substructural natural deduction calculi. As anticipated above, using the combination of $\mathcal{L}$ and $\mathcal{S}$ we recover the implicational fragment of dual intuitionistic-linear logic.

We could instead, however, draw the multiplicities α from an arbitrary algebraic structure A equipped with a join relation, assumed to be functional, commutative, associative, and to admit a partial unit for each element. To ensure that this parametrized calculus is well-behaved, we must define and impose additional constraints governing weakening and contraction. These rule out pathological situations in which, for instance, two non-zero elements could join to form a zero element. (We will return to this in more detail in

```
Inductive mult := Zero | One.

Inductive join : mult → mult → mult → Prop :=
| m_00 : join Zero Zero Zero
| m_10 : join One Zero Zero
| m_01 : join Zero One One.

Definition hasW m := m = Zero.
Definition hasC m := m = Zero.
```

Figure 2 Rocq encoding of the linearity resource algebra

Section 3.1). Establishing meta-theoretic properties about this generic calculus therefore yields the corresponding results for every concrete instance "for free."

The reader will notice a strong resemblance to the structure of adjoint logic. Indeed, the system presented in Figure 1 may be viewed as a fragment of adjoint natural deduction [24], without interaction between distinct substructural modes and therefore without the need to equip algebra elements with a specific preorder. Side-condition (2) of the variable rule (HYP) enforces a global independence property analogous to the independence constraints characteristic of adjoint systems.

3 Implementation of CARVe in Rocq

Following the Beluga implementation of CARVe, we model the (potentially partial) binary operators of resource algebras in Rocq as ternary relations

```
join : A → A → A → Prop.
```

This design choice aligns with the constraints of most (constructive) type-theoretic proof assistant, which support only total computable functions. A judgment `join a b c` expresses that resources `a` and `b` can be combined to form `c`. We define predicates `A → Prop` stating whether a given multiplicity is a zero element or satisfies weakening or contraction. As a concrete example, Figure 2 presents an encoding of the linear resource algebra.

We define two kinds of contexts for use in case studies: list contexts and function contexts. List contexts are lists of resources tagged with an algebra element, i.e., terms of type

```
Definition lctx (R A : Type) : Type := list (R * A).
```

where `R` is a type of resources and `A` the algebra carrier. List contexts are suitable for modeling contexts of named or unnamed resources, like sequents. Function contexts are total functions

```
Definition fctx (D R A : Type) : Type := D → R * A.
```

where `D` is some index set. In particular, when working with well-scoped de Bruijn indices, elements in `D` are finite sets of natural numbers. In the list representation, `D` is implicitly represented by the index at which where a pair appears.

In implementing this infrastructure, we adopt the basic algebraic definitions from Appel et al.'s Mechanized Semantic Library (MSL) [5], which provides a relational join typeclass:

```
Class Join (A : Type) : Type :=
  join : A → A → A → Prop.
```

Although CARVe could be constructed in Rocq from the ground up, we integrate MSL for two main reasons. First, while `rocq-carve` contexts may be instantiated with arbitrary resource algebras about which users can prove properties, it is practically convenient to provide access to MSL's well-structured hierarchy of canonical constructions and derived lemmas for separation algebras and their variants, like multi-unit separation algebras. In particular, for any concrete resource algebra, one may prove that its join operation satisfies algebraic properties like associativity or commutativity and instantiate the MSL classes for separation algebras and the like; these instances let one make use of MSL's extensive libraries of pre-proven lemmas.

Second, the library provides operators to construct algebras compositionally from simpler ones while preserving the underlying axioms; this eliminates the need to re-implement these structures from scratch. Thus we lift `A`'s algebraic structure to list and function contexts using MSL's separation algebra generators. For example, a product generator defines the obvious component-wise join operator on products of algebras (we endow `R` with a trivial algebra),

```
Instance Join_fctx {JA : Join A} : Join fctx :=
  Join_fun (Join_prod Join_equiv JA).
```

and the list generator similarly defines a join operator on lists element-wise. The induced join operator on list contexts corresponds exactly to its inductive characterization in [48], and it instantiates the same MSL classes as `A`. This generator-based approach lets us lift algebraic properties from resource algebras to context merging for free, and avoids the need to prove such properties on a case-by-case basis in proof developments.

As another example, we can combine the aforementioned linearity resource algebra with the trivial fully structural algebra

```
Definition mult : Type := unit.
Definition join := fun _ _ _ ⇒ ⊤.
```

using MSL's operator on disjoint sums to obtain a dual intuitionistic-linear resource algebra:

```
Definition mult : Type := linear.mult + structural.mult.
Instance Join_mult : Join mult :=
  Join_sum _ linear.Join_mult _ structural.Join_mult.
```

Here `unit` is Rocq's singleton datatype with sole inhabitant `tt`, and `linear.Join_mult` and `structural.Join_mult` are the respectives instances of the `Join_mult` class for the linear and structural components.

3.1 Encoding typing rules of the generic substructural λ-calculus

To illustrate the use of `rocq-carve` for encoding languages and mechanizing their meta-theory, we return to the generic substructural λ-calculus of Section 1. We follow the standard well-scoped de Bruijn approach for representing terms, and use the Autosubst library [44] essentially out-of-the-box to obtain auto-generated substitution operations and their associated properties for terms. We model contexts as functions from finite sets of natural numbers

```
Definition tenv n := fctx (fin n) ty A.

Inductive has_type {A} {JA : Join A} {SA : Sep_alg A} {RA : Res_alg A} {n}
                   (Δ : tenv n) : tm n → ty * A → Prop :=
| t_Var :
    ∀ (T : ty) (fn : fin n) (a : A),
      lookup Δ fn = (T, a) →
      ctx_forall hasW (upd Δ fn (T, core a)) →
      (hasW a → ctx_forall hasW Δ) →
      (hasC a → ctx_forall hasC Δ) →
      ¬ zero a →
      has_type Δ (var fn) (T, a)
| t_Abs :
    ∀ (S T : ty) (e1 : tm (S n)) (a : A),
      has_type (scons (S, a) Δ) e1 (T, a) →
      has_type Δ (lam S e1) (Fun S T, a)
| t_App :
    ∀ (Δ1 Δ2 : tenv n) (S T : ty) (e1 e2 : tm n) (a : A),
      has_type Δ1 e1 (Fun S T, a) →
      has_type Δ2 e2 (S, a) →
      join Δ1 Δ2 Δ →
      has_type Δ (app e1 e2) (T, a).
```

Figure 3 Rocq encoding of the generic substructural λ-calculus using `rocq-carve`

(a) We highlight the origin of each definition by colour: those provided by `rocq-carve` are shown in red, those from the MSL library in brown, and those from Autosubst in blue.

to type-multiplicity pairs: `fctx (fin n) ty A`. Importantly, context splits require no index shifting or renaming to preserve well-scopedness, since CARVe contexts remain structurally intact across splits.

Figure 3 presents a encoding using `rocq-carve` of the generic calculus. Alongside the type `A` and an instance of the typeclass `Join A`, which fix the resource algebra and its join relation, the typing judgment is indexed by instances of the classes `Sep_alg` and `Res_alg`. The former stems from the MSL library. It imposes standard algebraic constraints on the join relation, in particular that it is functional, associative, commutative, and positive (also known as "conic"), and that a unit element `core a` exists for each `a : A`. In practice, some meta-theory additionally requires the join relation to satisfy cancellativity, i.e., for `A` to be an instance of MSL's `Canc_alg` class, which enforces units' uniqueness. These requirements are mild and are satisfied by each of the canonical resource algebras described in Section 2.3.

The newly-defined typeclass `Res_alg` (Figure 4) supplements these algebraic assumptions with the additional structure required to characterize substructurality in the generic calculus. An instance of `Res_alg` consists of predicates `hasW` and `hasC` identifying which multiplicities admit weakening and contraction, together with several properties abstracting their structural behaviour. These axioms were, in effect, reverse engineered: they are sufficient for the generic meta-theory, including type safety and weak normalization, to hold uniformly for any resource algebra satisfying them, without claiming optimality.

The predicate `ctx_forall P` Δ expresses that a property `P : A → Prop` holds for every multiplicity tag in a context Δ. It is defined in the obvious way, as are look-up `lookup` and update `upd`.

```
Class Res_alg (A : Type) {JA : Join A} := {

  (* Primitive definitions *)
  hasW : A → Prop;
  hasC : A → Prop;
  unr a := hasW a ∧ hasC a;
  zero a := unr a ∧ ∃ a', unit_for a a' ∧ ¬ unr a';

  (* Properties of elements satisfying weakening *)
  hasW_identity : ∀ {a}, identity a → hasW a;
  hasW_join_closed : ∀ {a1 a2 a}, join a1 a2 a → hasW a1 → hasW a2 → hasW a;

  (* Properties of elements satisfying contraction *)
  hasC_idem : ∀ {a}, join a a a → hasC a;
  hasC_join_distrib : ∀ {c a1 a2 a12 a},
    hasC c → join a1 a2 a12 → join a12 c a →
    ∃ a1' a2', join a1 c a1' ∧ join a2 c a2' ∧ join a1' a2' a;

  (* Properties of 'zero-like' elements *)
  zero_join_collapse : ∀ {a1 a2 a}, join a1 a2 a → zero a1 → zero a2 → a1 = a;
  nonzero_split : ∀ {a1 a2 a}, join a1 a2 a → ¬ zero a1 → a1 = a;
}.
```

Figure 4 Class of generic substructural resource algebras

3.2 Proving meta-theory generically

As an illustrative example of using `rocq-carve` for meta-theory, we will outline a proof of weak normalization via logical relations. We follow the approach of Dreyer et al. [20], in which the logical relation is split into value and expression components. We adapt the definitions and proofs of Stark [43], originally designed for a fully structural calculus, to allow for a clean comparison of the additional complexity introduced by substructurality. We define the dynamics of the calculus in the standard way, allowing reductions under binders.

First, a predicate `L` is defined by structural recursion on object-language types, expressing that a term is reducible at type `T`.

```
Fixpoint L {n} (T : ty) (e : tm n) : Prop :=
  match T with
  | Unit ⇒ ∃ v, mstep e v ∧ value v
  | Fun T1 T2 ⇒
      match e with
      | lam _ e' ⇒
          ∀ k (ξ : fin m → fin k) v,
            L T1 v → E (L T2) e'[v .: (ξ ≫ var)]
      | _ ⇒ ⊥
      end
  end.
```

At unit types, a term is reducible if it evaluates to a value. At function types `Fun T1 T2`, a term is reducible if it a λ-abstraction whose body, when instantiated with any argument `v` reducible at `T1` (and under extensions of any arbitary renaming ξ, written `v .: (ξ ≫ var)` in Autosubst notation), evaluates to a value reducible at `T2`. Separately, an expression-level

predicate `E` states that a term evaluates to a value satisfying such a property:

```
Definition E {m} (L : tm m → Prop) (e : tm m) : Prop :=
  ∃ v, mstep e v ∧ L v.
```

From these definitions, it is immediate that all reducible terms halt:

```
Lemma EL_halts {n} :
  ∀ (T : ty) (e : tm n), E (L T) e → ∃ v, mstep e v ∧ value v.
```

Next, we define the notion of reducible simultaneous substitutions and a semantic typing judgment:

```
Definition G {n k} (Δ : tenv n) (σ : fin n → tm k) : Prop :=
  ∀ (x : fin m), L (fst (Δ x)) (σ x).

Definition has_type_sem {n} (Δ : tenv n) (e : tm n) (T : ty) : Prop :=
  ∀ k (σ : fin n → tm k), G Δ σ → E (L T) e[σ].
```

Intuitively, `G Δ σ` asserts that the substitution σ maps each variable to a term reducible at the type assigned to it by the context Δ. The identity substitution `fun x => var x` trivially satisfies this property for the empty context `emptyC`, defined as the vacuous map from `fin 0` to some constant default value. The judgment `has_type_sem Δ e T` expresses that under any such substitution, `e` reduces to a value satisfying `L T`.

In traditional mechanizations of substructural systems with de Bruijn indices, the main complexity of logical relations proofs arises when reasoning about such simultaneous substitutions. Consider the fundamental lemma:

```
Lemma fundamental {n} :
  ∀ (Δ : tenv n) (M : tm n) (T : ty) (a : A),
    has_type Δ M (T, a) →
    has_type_sem Δ M T.
```

In the application case, the typing context is split. With standard approaches, one must then *also* explicitly split the substitution into two. This requires a delicate treatment of indices. Mechanizing this operation is non-trivial and typically introduces significant overhead. By contrast, CARVe keeps the full typing context intact across context splits. Because unavailable assumptions never appear in the syntax of terms by design, the substitution simply does not act on assumptions that were "removed" for resource-tracking purposes. As a result, it is safe to also keep simultaneous substitutions intact throughout.

Whereas the traditional approach inflates the proof effort, the CARVe approach to substitution-related reasoning maintains essentially the same the complexity as in the purely structural setting. Only one additional lemma is needed to show that reducible substitutions are preserved across context splits:

```
Lemma G_split {n k} :
  ∀ (Δ1 Δ2 Δ : tenv n) (σ : fin n → tm k),
    G Δ σ →
    join Δ1 Δ2 Δ →
    G Δ1 σ ∧ G Δ2 σ.
```

Putting these pieces together, we obtain weak normalization for our generic calculus: any closed, well-typed term halts.

System	Property	LoC
Generic substructural λ-calculus	Weak normalization (1)	194
Generic substructural λ-calculus	Weak normalization (2)	195
Generic substructural λ-calculus	Progress	40
Generic substructural λ-calculus	Preservation	328
Generic substructural λ-calculus	Let-elimination	71
Generic substructural λ-calculus	Equivalence with linear λ-calculus	43
CP	Progress	261
Adjoint natural deduction	Progress	66

Figure 5 Summary of case studies mechanized using `rocq-carve`

(a) Remark: The lines-of-code count above excludes definitions of syntax, typing rules, and so on. The `rocq-carve` library in its entirety counts some 3,400 lines of code including the case studies, but excluding code generated by or imported from Autosubst and MSL.

```
Lemma weak_norm :
  ∀ (e : tm 0) (T : ty) (a : A),
    has_type emptyC e (T, a) →
    ∃ v, mstep e v ∧ value v.
```

The goal of formalizing such a generic system is to obtain a uniform meta-theory for a family of substructural λ-calculi. Once proven generically, the results can be instantiated with a suitable resource algebra. The reader may be surprised that most of the preceding development is lifted directly from the proof by Stark [43]. While our proofs rely on lemmas provided by the `rocq-carve` library, their overall structure does not change dramatically, and the definitions and statements remain unchanged. This close correspondence highlights an important point usually obscured by the low-level bookkeeping involved in substructural context management: the logical-relations argument, and more generally the reduction semantics, do not fundamentally depend on resource usage. Using `rocq-carve`, adapting this normalization proof to cover linear, affine, relevant, and unrestricted disciplines, all at once, requires only some thirty additional lines of Rocq code.

4 Case studies

Alongside our implementation of CARVe, we have used it to mechanize several case studies in Rocq, summarized in Figure 5. These case studies serve both as validation of the design and as reference material for users of the library.

4.1 Generic substructural λ-calculus

Our central case studies concern the implicational fragment of the generic substructural λ-calculus introduced in Sections 2.3 and 3.1, parametrized by arbitrary instances of the `Sep_alg` and `Res_alg` typeclasses. In effect, this amounts to proving the desired results uniformly for the linear, affine, relevant, and structural λ-calculi at once.

In addition to the weak normalization proof outlined in Section 3.2, we also mechanize a second proof following the classical approach, in which a single logical predicate is defined recursively on the structure of types. In both developments, we proceed in the usual

way by proving that every well-typed term satisfies its logical predicate, from which weak normalization follows as a corollary.

Next, we implement proofs of progress and preservation for the calculus as a traditional benchmark. The proof of progress is a straightforward adaptation of the existing Rocq script from *Software Foundations* [37]. The proof of preservation is more involved due to the crucial substitution lemma. This in turn requires an auxiliary renaming lemma together with proofs of the structural properties governing zero-like assumptions.

To showcase another use of CARVe, we implement a simple function that translates expressions of the generic substructural λ-calculus with let-expressions into one without them, following the standard encoding of let-expressions as function applications. We show that this code-simplification program is type-preserving. While reasoning about such transformations cannot be done directly in Beluga [21], the proof in Rocq is routine.

Finally, as an additional property, we establish the equivalence between the generic substructural λ-calculus instantiated with the linearity resource algebra and the standard linear λ-calculus. The proof is by simple induction on the syntax of terms.

4.2 Adjoint natural deduction

As a proof of concept, we encode in `rocq-carve` the typing rules of adjoint logic [39] and of the adjoint natural deduction calculus [24]. The systems are parametrized by a resource-algebraic class extending `Res_alg` with a preorder. We extend the proof of progress from the generic substructural λ-calculus to the adjoint natural deduction calculus; we expect that other meta-theooretic proofs will also carry over with minimal additional effort.

4.3 CP

To illustrate the use of `rocq-carve` for mechanizing sequent calculi, we prove progress for a fragment of the session-typed process calculus CP [46]. CP is a proofs-as-processes interpretation of classical linear logic. Its processes are typed by list contexts that specify the types of communication channels. Proving progress for this calculus amounts to proving cut elimination for the underlying linear logic.

5 Related work

The intuition that logical consequence and provability depend on the way we use information precedes linear logic and is tightly intertwined with the history of the λ-calculus, sequents, and combinatory logic. In 1972, Urquhart [45] introduced a model of relevance logic where the truth of a formula is relative to a "piece of information" belonging to a join-semilattice. In strictness analysis, Mycroft [30] annotated function arguments to indicate if they are definitely or possibly used. In a sense, even approaches such as DILL [8] or LNL [9] based on multi-zonal contexts may be seen as tagging by location.

In a seminal paper, Girard et al. introduced *bounded* linear logic [22], where a family of modalities $!_x\, A$ replaces the binary state of the exponential with an index, allowing the logic to track the degree of reuse. This is the direct ancestor of modern quantitative and graded modal type theories [7, 17, 1], where those indexes have been refined in terms of (ordered) commutative monoids, quantales, and semirings. The same remark applies to adjoint logics [24]. While these studies share the high-level idea of using algebraic annotation to unify different logics, their aims differ substantially from ours: they provide foundations

to the design of resource-aware functional programming languages (see Granule, Snax [33] and Idris 2 [12]).

Similarly, we will not survey the full landscape of approaches to substructural context management in the literature; we instead refer the reader to [48] for a partial overview.

A much closer relative to our approach is the body of work on separation algebras [14, 38, 25, 26], developed for reasoning about program correctness—and, in general, meta-theoretic properties of (concurrent) programming languages—wherever some form of separation logic is warranted. It is no coincidence that we have adopted Dockin et al.'s [19, 6] formulation and implementation in the Mechanized Semantic Library. Note that the latter has also been used for meta-reasoning, as in [23].

5.1 Meta-frameworks

Given our focus, we review more carefully what may be called *meta-frameworks*: libraries and tools that support meta-theoretic reasoning over substructural logics.

Yalla [27] is a parameterized propositional linear logic system formalized in Rocq, designed for user-specific instantiation. By adjusting parameters—specifically those governing exchange and mix rules—users can generate a customized Rocq library that includes verified proofs for fundamental properties like cut admissibility and focusing for the primary system and its variants. Another meta-framework is EMTST, a Rocq library by Castro et al. [16] aimed at representing and validating theorems about session types: binders are encoded in a nameless style and contexts as finite maps à la MathComp, where splitting makes a context undefined when it would result in duplicated entries. Both libraries export several dozen functions and lemmas, but seem less general than `rocq-carve`, since they are geared towards a specific application domain and concentrate on (variations of) linearity as a resource.

Wood and Atkey's proposal [47], as formalized in Agda, is closer in generality to our approach. They represent variable usage via skew-semiring annotations, allowing contexts to remain unchanged while updating only the usage information of specific variables. By extending McBride's kits and traversals to a quantitative linear setting, they characterize the abstract conditions needed for binding-respecting traversals of simply typed λ-terms. In this view, usages are vectors, usage-preserving context maps are matrices, and the linearity properties of these maps provide precisely the lemmas required to show that traversals—and thus renaming, sub-usaging, and substitution—reserve both typing and usage. Extending the technique to settings such as CP or to general meta-theory remains, to our understating, unresolved.

Further afield are tools such as TATU [31], which automatically ensures properties such as cut elimination for object logics provided that they are encoded in linear logic with subexponentials.

6 Conclusions and future work

We have presented the Rocq library `rocq-carve` to facilitate mechanizations of substructural systems in which assumption usage is managed via resource tags drawn from a user-specified resource algebra. The library supports both list-based and functional representations of contexts. It integrates smoothly with (well-scoped) de Bruijn syntax, eliminating much of the overhead associated with explicit context manipulation in substructural settings. In particular, `rocq-carve` interoperates seamlessly with the Autosubst library for handling substitutions.

By design, our library is parametric in an underlying resource algebra, and we leverage this generality to study and prove meta-theoretic properties such as weak normalization and type-preservation generically. To ensure the library is robust, we have carried out a range of case studies, from natural-deduction-style type systems to sequent calculi. In addition, we have considered a broad range of resource algebras: structural, linear, affine, strict, and adjoint.

There are two main directions we plan to explore in future work. First, we intend to extend, refine, and improve the `rocq-carve` library itself. There are several natural paths to pursue:

1. Providing a general API to better expose the full range the operations and properties the library supports;
2. Supporting additional algebraic structures, such as step-indexed [26] and ordered [34] resource algebras, and hence the mechanization of a wider range of substructural type systems; and
3. Generalizing the infrastructure to handle other context representations, such as length-indexed lists, as well as intrinsically-typed representations of terms.

Second, we plan to use the `rocq-carve` library to study further meta-theoretic properties that can be proven generically. Our approach demonstrated that the proof of weak normalization extends to the substructural setting with few modifications. We wish to confirm that other logical-relations proofs similarly generalize under CARVe. Promising candidate benchmarks include substructural parametricity [3] and bisimilarity / program equivalence [11, 32]. This would shed light on Crole's question, "How linear is Howe?" [18]. Our guess: not much.

References

1 Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. A graded modal dependent type theory with a universe and erasure, formalized. *Proc. ACM Program. Lang.*, 7(ICFP):920–954, 2023. doi:10.1145/3607862.

2 Andreas Abel and Nicolai Kraus. A lambda term representation inspired by linear ordered logic. In Herman Geuvers and Gopalan Nadathur, editors, *Proc. LFMTP '11*, volume 71 of *Electron. Proc. Theor. Comput. Sci.*, pages 1–13, 2011. doi:10.4204/EPTCS.71.1.

3 C. B. Aberlé, Karl Crary, Chris Martens, and Frank Pfenning. Substructural parametricity. In Maribel Fernández, editor, *Proc. FSCD '25*, volume 337 of *LIPIcs*, pages 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. URL: https://doi.org/10.4230/LIPIcs.FSCD.2025.4, doi:10.4230/LIPICS.FSCD.2025.4.

4 Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Proc. TYPES '04*, volume 3839 of *Lect. Notes Comput. Sci.*, pages 1–16. Springer, 2006. doi:10.1007/11617990_1.

5 Andrew W. Appel, Robert Dockins, and Aquinas Hobor. Mechanized Semantic Library, 2009. URL: https://vst.cs.princeton.edu/msl/.

6 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

7 Robert Atkey. Syntax and semantics of quantitative type theory. In *Proc. LICS '18*, pages 56–65, 2018. doi:10.1145/3209108.3209189.

8 Andrew Barber. Dual intuitionistic linear logic. Technical report ECS-LFCS-96-347, University of Edinburgh, 1996. URL: https://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347.

9 P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *Proc. CSL '94*, pages 121–135. Springer, 1994. doi:10.5555/647844.736565.

10 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, December 2017. URL: https://dl.acm.org/doi/10.1145/3158093, doi:10.1145/3158093.

11 G. M. Bierman. Program equivalence in a linear functional language. *J. Funct. Program.*, 10(2):167190, 2000. doi:10.1017/S0956796899003639.

12 Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *Proc. ECOOP '21*, volume 194 of *LIPIcs*, pages 9:1–9:26, 2021. doi:10.4230/LIPICS.ECOOP.2021.9.

13 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *Proc. CONCUR '10*, volume 6269 of *Lect. Notes Comput. Sci.*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

14 Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proc. LICS '07*, pages 366–378, 2007. doi:10.1109/LICS.2007.30.

15 Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Nobuko Yoshida, and Daniel Zackon. The concurrent calculi formalisation benchmark. In Ilaria Castellani and Francesco Tiezzi, editors, *Proc. COORDINATION '24*, volume 14676 of *Lect. Notes Comput. Sci.*, pages 149–158, 2024. URL: https://doi.org/10.1007/978-3-031-62697-5_9, doi:10.1007/978-3-031-62697-5_9.

16 David Castro, Francisco Ferreira, and Nobuko Yoshida. EMTST: Engineering the meta-theory of session types. In Armin Biere and David Parker, editors, *Proc. TACAS '20*, volume 12079 of *Lect. Notes Comput. Sci.*, pages 278–285, 2020. doi:10.1007/978-3-030-45237-7_17.

17 Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):50:1–50:32, 2021. doi:10.1145/3434331.

18 R. L. Crole. How linear is Howe? In G. McCusker, A. Edalat, and S. Jourdan, editors, *Advances in Theory and Formal Methods*, pages 60–72. Imperial College Press, 1996.

19 Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In Zhenjiang Hu, editor, *Proc. APLAS '09*, volume 5904 of *Lect. Notes Comput. Sci.*, pages 161–177. Springer, 2009. doi:10.1007/978-3-642-10672-9_13.

20 Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. Semantics of type systems (lecture notes), February 2018. URL: https://plv.mpi-sws.org/semantics/2017/lecturenotes.pdf.

21 Aïna Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. LINCX: A linear logical framework with first-class contexts. In Hongseok Yang, editor, *Proc. ESOP '17*, volume 10201 of *Lect. Notes Comput. Sci.*, pages 530–555. Springer, 2017. doi:10.1007/978-3-662-54434-1_20.

22 Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992. doi:10.1016/0304-3975(92)90386-T.

23 Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proc. POPL '10*, pages 171–184. ACM, 2010. doi:10.1145/1706299.1706322.

24 Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. Adjoint natural deduction. In *Proc. FSCD '24*, 2024. doi:10.4230/LIPIcs.FSCD.2024.15.

25 Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. In Helmut Seidl, editor, *Proc. ESOP '12*, volume 7211 of *Lect. Notes Comput. Sci.*, pages 377–396. Springer, 2012. doi:10.1007/978-3-642-28869-2_19.

26 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

27 Olivier Laurent. Yalla. https://github.com/olaure01/yalla/, 2017.

28 Nicholas D. Matsakis and Felix S. Klock. The Rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014. doi:10.1145/2692956.2663188.

29 Conor McBride. Everybody's Got to Be Somewhere. In Robert Atkey and Sam Lindley, editors, *Proc. MSFP '18*, volume 275 of *Electron. Proc. Theor. Comput. Sci.*, pages 53–69, 2018. doi:10.4204/EPTCS.275.6.

30 Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. 4e Colloque International sur la Programmation*, pages 269–281. Springer, 1980. doi:10.5555/647324.721526.

31 Vivek Nigam, Elaine Pimentel, and Giselle Reis. An extended framework for specifying and reasoning about proof systems. *J. Log. Comput.*, 26(2):539–576, 2016. doi:10.1093/logcom/exu029.

32 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.*, 239:254–302, 2014. URL: https://www.sciencedirect.com/science/article/pii/S089054011400100X, doi:10.1016/j.ic.2014.08.001.

33 Frank Pfenning. Snax. https://bitbucket.org/fpfenning/snax/src/main/, 2024. (Version 1.8).

34 Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proc. LICS '09*, pages 101–110, 2009. doi:10.1109/LICS.2009.8.

35 Brigitte Pientka and Jana Dunfield. Programming with proofs and explicit contexts. In *Proc. PPDP '08*, pages 163–173, 2008. doi:10.1145/1389449.1389469.

36 Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *Proc. IJCAR '10*, volume 6173 of *Lect. Notes Comput. Sci.*, pages 15–21. Springer, 2010. doi:10.1007/978-3-642-14203-1_2.

37 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Ctlin Hricu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. 2025. (Version 6.7). URL: http://softwarefoundations.cis.upenn.edu.

38 François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *J. Funct. Program.*, 23(1):38–144, 2013. doi:10.1017/S0956796812000366.

39 Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL: https://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf.

40 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proc. CPP '20*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.

41 Anders Schack-Nielsen and Carsten Schürmann. Curry-style explicit substitutions for the linear and affine lambda calculus. In Jürgen Giesl and Reiner Hähnle, editors, *Proc. IJCAR '10*, volume 6173 of *Lect. Notes Comput. Sci.*, pages 1–14. Springer, 2010. doi:10.1007/978-3-642-14203-1_1.

42 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Math. Struct. Comput. Sci.*, 16(3):527–552, 2006. doi:10.1017/S0960129506005238.

43 Kathrin Stark. *Mechanising syntax with binders in Coq*. PhD thesis, Saarland University, Saarbrücken, 2019. URL: https://www.ps.uni-saarland.de/~kstark/thesis/.

44 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de Bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proc. CPP '19*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

45 Alasdair Urquhart. Semantics for relevant logics. *J. Symb. Log.*, 37(1):159–169, 1972. doi:10.2307/2272559.

46 Philip Wadler. Propositions as sessions. In *Proc. ICFP '12*, pages 273–286, 2012. doi:10.1145/2364527.2364568.

47 James Wood and Robert Atkey. A framework for substructural type systems. In Ilya Sergey, editor, *Proc. ESOP '22*, volume 13240 of *Lect. Notes Comput. Sci.*, pages 376–402, 2022. doi:10.1007/978-3-030-99336-8_14.

48 Daniel Zackon, Chuta Sano, Alberto Momigliano, and Brigitte Pientka. Split decisions: Explicit contexts for substructural languages. In *Proc. CPP '25*, pages 257–271, 2025. doi:10.1145/3703595.3705888.

A Generative AI statement

The `rocq-carve` library was not developed with the aid of generative artificial intelligence.

B Data availability statement

The `rocq-carve` library is provided as supplementary material to this submission.