Split Decisions: Explicit Contexts for Substructural Languages

Daniel Zackon
McGill University
Montreal, Canada
daniel.zackon@mail.mcgill.ca

Chuta Sano
McGill University
Montreal, Canada
chuta.sano@mail.mcgill.ca

Alberto Momigliano
University of Milan
Milan, Italy
momigliano@di.unimi.it

Brigitte Pientka
McGill University
Montreal, Canada
brigitte.pientka@mcgill.ca

Abstract

A central challenge in mechanizing the meta-theory of substructural languages is modeling contexts. Although various ad hoc approaches to this problem exist, we lack a set of good practices and a simple infrastructure that can be leveraged for mechanizing a wide range of substructural systems.

In this work, we describe Contexts as Resource Vectors (CARVe), a general syntactic infrastructure for managing substructural contexts, where elements are annotated with tags from a resource algebra denoting their availability. Assumptions persist as contexts are manipulated since we model resource consumption by changing their tags. We may thus define relations between substructural contexts via simultaneous substitutions without the need to split them. Moreover, we establish a series of algebraic properties about context operations that are typically required to carry out proofs in practice. CARVe is implemented in the proof assistant Beluga.

To illustrate best practices for using our infrastructure, we give a detailed reformulation of the linear sequent calculus and bidirectional linear λ-calculus in terms of CARVe's context operations and prove their equivalence using the aforementioned algebraic properties. In addition, we apply CARVe to mechanize a diverse set of systems, from the affine λ-calculus to the session-typed process calculus CP, giving us confidence that CARVe is sufficiently general to mechanize a broad range of substructural systems.

CCS Concepts: • **Software and its engineering** → *Formal software verification*; • **Theory of computation** → *Logic and verification*; *Proof theory*.

CPP '25, January 20–21, 2025, Denver, CO, USA

ACM ISBN 979-8-4007-1347-7/25/01
https://doi.org/10.1145/3703595.3705888

Keywords: mechanized meta-theory, substructural type systems, substructural logics, linear logic, verification

ACM Reference Format:
Daniel Zackon, Chuta Sano, Alberto Momigliano, and Brigitte Pientka. 2025. Split Decisions: Explicit Contexts for Substructural Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3703595.3705888

1 Introduction

Over the past decades, substructural type systems and logics have been used to reason about resources in a wide range of programming language applications, including quantum computing [51], concurrent programming [5, 54], and memory management, as in Rust [36] and linear Haskell [4]. The need for mechanized meta-theory to rigorously ensure the trustworthiness of such systems is significant, as even seemingly minor extensions can compromise basic safety properties. Still, mechanization is in general a major undertaking, and substructurality demands further careful attention. Indeed, mechanizing linearity is one of the challenges in the recent Concurrent Calculi Formalisation Benchmark [8].

Substructural systems usually control and sometimes eliminate *contraction* (which permits a formula to be used more than once) and *weakening* (which permits a formula to be left unused).[1] Consequently, determining how to track and allocate formulas, seen as resources, is a key challenge when formalizing and proving meta-theoretical properties of these substructural languages within a proof assistant.

Consider, for illustration, the following implicational fragment of an intuitionistic linear sequent calculus.

$$\frac{}{A \vdash A}\ (\text{hyp}) \qquad \frac{\Delta_1 \vdash A \quad \Delta_2, A \vdash B}{\Delta_1, \Delta_2 \vdash B}\ (\text{cut})$$

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ (\multimap\text{R}) \qquad \frac{\Delta_1 \vdash A \quad \Delta_2, B \vdash C}{\Delta_1, \Delta_2, A \multimap B \vdash C}\ (\multimap\text{L})$$

[1] By and large, these systems admit exchange, and so we restrict our attention to substructurality in the sense of controlling contraction and weakening.

Reading the ($\multimap$L) rule bottom-up, we check that an assumption $A \multimap B$ appears in the conclusion's context and consume it; then, we split up all *other* assumptions into two pairwise disjoint subcontexts for the premises, and add to one a *new* assumption B. From a mechanization standpoint, it is usually sensible to make this splitting procedure explicit with a context join operation $\Delta_1 \bowtie \Delta_2 = \Delta$ that is both commutative and associative. Yet splitting a context is fundamentally a non-deterministic process. The other rules above present related considerations.

These context operations take on additional complexities in richer systems that involve not only linear but also affine or even ordinary intuitionistic assumptions. Note in particular the (hyp) rule, which enforces the linear usage of propositions; with the addition of intuitionistic assumptions, encoding this rule requires checking that all linear assumptions have been used in the context apart from A.

The prevailing approach to encoding substructural systems is to model contexts explicitly, typically as lists or dictionaries, with a naïve non-deterministic splitting operation resembling on-paper developments (see for example [33, 52, 58]). Often, separate contexts are used to track assumptions governed by different structural rules. While this solution facilitates encodings that more faithfully mimic their on-paper counterparts, they are inconvenient in realistic implementations and incompatible with de Bruijn encodings and explicit simultaneous substitutions [50]. Alternative techniques have been developed with the aim of eliminating or simplifying some of this overhead, including leftover typing [1] and tracking variable usage through proof terms [16, 47]. Though elegant, these solutions generally demand substantial modifications to the on-paper formulation of a calculus. Moreover, the first approach fits algorithmic rather than declarative proof systems, while the latter seems challenging to generalize to richer substructural logics such as adjoint logic [31, 43].

In this paper, we follow an alternative strategy first outlined by Schack-Nielsen and Schürmann [48, 50]: we model substructural typing contexts explicitly, parametric to some algebra for specifying resource usage. As a result, users may obtain contexts for particular substructural systems such as linear or affine type systems "for free" by specifying a suitable algebra. While variables are retained in contexts, their resource annotations may change. Thus the underlying context remains intact. Ignoring the usages, the rules of a given substructural system closely resemble their intuitionistic counterparts. This approach has two main advantages: firstly, it is in principle compatible with a range of encodings for binders, including de Bruijn, locally nameless, and higher-order abstract syntax (HOAS); secondly, it lends itself naturally to representing simultaneous substitutions without splitting them.

Using this framework, we develop **C**ontexts **a**s **R**esource **Ve**ctors (CARVe), a low-level syntactic infrastructure for implementing substructural systems and reasoning about their meta-theory. CARVe supports a small set of core context operations defined as relations: merging (or splitting) contexts, updating a given assumption in the context (either its tag or the actual item stored), and checking whether a context's available resources have been consumed. From these primitive operations we can define several other context operations, including look-up and the permutation of elements. We further implement simultaneous substitutions as context relations. We establish a series of algebraic and well-formedness properties of context operations that are required to carry out proofs in practice. CARVe is implemented in the proof assistant Beluga [40, 41], which enables us to explore multiple binding approaches—including de Bruijn encodings and HOAS—and so to better understand CARVe's generality. Still, CARVe is not restricted to Beluga can in principle be realized any proof assistant.

Our approach is related to that of Wood and Atkey [56, 57], whereby variables are annotated with values from a skew semiring indicating their usage by terms. While they take a category-theoretic perspective and use combinators to manage resource vectors and syntax traversals, we define context operations directly on the syntax of contexts—as low-level relations on lists—together with meta-theoretic properties. This makes our framework intuitive to use and to apply to a wide range of systems.

To illustrate best practices for using our infrastructure, we give a detailed reformulation of the linear sequent calculus and bidirectional linear λ-calculus in terms of CARVe's context operations and prove their equivalence using the aforementioned algebraic properties. In addition, we apply CARVe to mechanize a diverse set of systems, giving us confidence that CARVe is sufficiently general to mechanize a broad range of substructural systems together with meta-theoretic proofs. Beyond the cited equivalence proof, we have mechanized cut-elimination for the linear sequent calculus, type preservation for the affine and linear λ-calculus with both a substitution and environment-based operational semantics, type preservation for the propositional fragment of the session-typed process calculus CP [54], and a translation between the encoding of CP using CARVe and an encoding using explicit linearity predicates to track resource usage [47].

An artifact [59] containing the full formalization may be downloaded directly from

https://zenodo.org/records/13777001.

Further information about the artifact, including a paper-to-artifact correspondence guide and instructions for installation and execution, may be found in the `README.md` file.

2 Contexts as Resource Vectors: An Infrastructure

The representation of contexts is central when implementing and reasoning about systems that support variable binding mechanisms such as programming languages, type systems, and logics. The prevalent and natural choice for these systems is to model contexts as lists of assumptions. We follow in this tradition. In CARVe, context entries will be typed variables $x : A$ annotated with elements α of some algebra, to be read as *multiplicities*. (For presentation purposes, we detail the case where we have names, but the approach works seamlessly in a nameless setting.) Informally, multiplicities specify the availability of a resource, and the associated algebraic structure sets out how they may be subdivided and composed.

Our definition of contexts does not assume *well-formedness*, in the sense of variable names being pairwise distinct. While this is needed when proving properties like type uniqueness, for generality and concision we choose to keep the infrastructure agnostic to whether and how well-formedness is enforced. It is, however, an invariant that CARVe's primitive context operations will preserve.

In the remainder of this section, we will outline these operations and the properties they possess, organized into three key themes: *resource allocation*, *exhaustedness*, and *context search and manipulation*.

2.1 Resource Allocation

One difficulty in mechanizing multi-premise multiplicative rules like ($\multimap$L) in section 1 is determining how to split contexts to allocate resources. The central idea behind our approach is to keep contexts *intact* as they are joined and split, with only their multiplicities differing. In other words, instead of merging two disjoint contexts, we "weave together" compatible multiplicities at each position of a single context. We define an explicit non-deterministic split—or, viewed differently, deterministic merge—operation $\bowtie_\circ$ on contexts, which is parametrized by a binary operation $\circ$ over some algebraic structure.

The operation $\circ$ may be partial, and hence so may $\bowtie_\circ$. Following Dockins *et al.* [18], we will present these operations as ternary *relations* between multiplicities and contexts, respectively. Every proof assistant supports this presentation, which is simpler to accommodate compared to a functional one, where partiality tends to take its toll.

Context merge is inductively defined by the rules:

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

Different choices of resource algebras encode different substructural properties. We will, for the present, focus our attention on the monoid

$$\mathcal{L} = (\{0, 1\}, \bullet, 0)$$

that characterizes allocation for linear or affine contexts, where $\bullet$ is defined by the following multiplication table:

$\bullet$	0	1
0	0	1
1	1	–

The multiplicity 1 denotes a variable available exactly once. A variable of multiplicity 0 is irrelevant in the current branch of a derivation: it was either never available, has been previously consumed, or is available linearly elsewhere. Note that the operation $\bullet$ is partial, as $1 \bullet 1$ is left undefined (–).

As is well-known, $\mathcal{L}$ satisfies several desirable algebraic properties. Notably, it is

- Functional: $\alpha \bullet \beta = \gamma$ and $\alpha \bullet \beta = \gamma'$ imply $\gamma = \gamma'$;
- Cancellative: $\alpha \bullet \beta = \gamma$ and $\alpha' \bullet \beta = \gamma$ imply $\alpha = \alpha'$;
- Commutative: $\alpha \bullet \beta = \gamma$ implies $\beta \bullet \alpha = \gamma$; and
- Zero-sum-free: $\alpha \bullet \beta = 0$ implies $\alpha = \beta = 0$.

The latter property prevents used resources from arbitrarily transforming into linear ones.

In addition to its modularity, this construction has the advantage of allowing the underlying algebraic structure to naturally impose itself on $\bowtie_\circ$. Specifically:

Proposition 2.1. *Let $\Delta = x_1 :^{\alpha_1} A_1, \ldots, x_n :^{\alpha_n} A_n$ be a context and $\mathcal{M} = (M, \circ, i)$ a monoid. Then*

$$\Delta\mathcal{M} \coloneqq (\{x_1 :^{\beta_1} A_1, \ldots, x_n :^{\beta_n} A_n \mid \beta_1, \ldots, \beta_n \in M\}, \bowtie_\circ, i\Delta)$$

is a monoid. Moreover, if $\mathcal{M}$ is commutative, then $\Delta\mathcal{M}$ is commutative, and so forth.

Here $\alpha\Delta$ denotes the result of setting all multiplicities in a context Δ to α.

Context merge also preserves well-formedness:

Proposition 2.2. *Suppose $\Delta_1 \bowtie \Delta_2 = \Delta$. Then the following are equivalent: (1) Δ_1 is well-formed, (2) Δ_2 is well-formed, and (3) Δ is well-formed.*

2.2 Exhaustedness

In substructural logics, it is common for a rule to demand an empty context, such as the right rule for the multiplicative unit of the linear sequent calculus. In our tag-based approach, where context elements persist, this is enforced by requiring that the context contain only "harmless" assumptions, in the sense that they are subject to weakening. We call such a context *exhausted*, denoted by $\mathsf{exh}(\Delta)$. Exhaustedness can also be used to enforce that a context is singleton, as in the case of the axiom rule of the linear sequent calculus (see section 3).

Checking that a context is exhausted entails checking the harmlessness of its elements. This check will depend on the chosen algebra or the system being encoded. For systems using $\mathcal{L}$ as a resource algebra, only used assumptions of multiplicity 0 are regarded as harmless. Unrestricted assumptions will be considered harmless in systems that support them (see section 5).

When only the algebraic unit is deemed harmless, as in the linear case, an exhausted context corresponds to the unit element of the context monoid from Proposition 2.1.

2.3 Context Search and Manipulation

Though extending a context with a new assumption or determining its top-most element are simple tasks, a more careful approach is needed for arbitrary context search and manipulation. In line with our principle of preserving contexts as they are split, multiplicity tags enable us to effectively model changes in resource availability by simply updating a variable's multiplicity.

In the linear or affine case, this means modifying a variable's tag from 1 to 0 or *vice versa*. Some applications (*e.g.*, [54]) also require changing the *types* of variables as computation occurs. Accordingly, we choose to take a general version of context updating as a primitive in its own right, which can be used to implement updating the type and multiplicity of a variable as a special case. We write

$$\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B] = \Delta'$$

to mean that $x :^{\alpha} A$ appears at the n-th position from the head of Δ, and replacing its occurrence in Δ with $y :^{\beta} B$ results in the context Δ'. It is defined in the expected manner by recursively traversing the context:

$$\frac{}{(\Delta, x :^{\alpha} A)[x :^{\alpha} A \mapsto_{|\Delta|+1} y :^{\beta} B] = \Delta, y :^{\beta} B}$$

$$\frac{\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B] = \Delta'}{(\Delta, z :^{\gamma} C)[x :^{\alpha} A \mapsto_n y :^{\beta} B] = \Delta', z :^{\gamma} C}$$

As with merge, we define updating relationally. For simplicity of notation, and as updating is functional and look-up unique by Proposition 2.3, we will at times abuse notation and use $\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B]$ to represent the resulting context (and thus, implicitly, state the existence of $x :^{\alpha} A$ somewhere in Δ). We also omit the index n where it is inessential.

Conversely, when adopting a nameless approach using, *e.g.*, de Bruijn indices, all information about names may be dropped in favor of information about location. Still, we permit names to be updated in the general definition. This allows us to define variable swapping within a context, denoted $\Delta[x \rightleftarrows y]$, crucial in proving the admissibility of exchange without needing a separate predicate for permuting context elements:

$$\Delta[x \rightleftarrows y] = \Delta' :\equiv$$
$$\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B][y :^{\beta} B \mapsto_m x :^{\alpha} A] = \Delta'$$
$$\text{for some } n \neq m$$

(The condition $n \neq m$ ensures that the y in the second update is an element of Δ, rather than the one newly created at n.)

One may also define context membership properties like the following via updating:

$$x :^{\alpha} A \in_n \Delta :\equiv \Delta[x :^{\alpha} A \mapsto_n x :^{\alpha} A] = \Delta$$
$$x :^{\alpha} A \in \Delta :\equiv \Delta[x :^{\alpha} A \mapsto_n x :^{\alpha} A] = \Delta \text{ for some } n$$

Having a single operation in this way significantly reduces the number of lemmas needed for reasoning about the manipulation of contexts. Furthermore, context updating has a range of useful properties that may be exploited in proofs. If we regard updating as a relation between two contexts, and the information in the square brackets as labels, then the set of well-formed contexts (of some fixed length) forms a labeled transition system satisfying the following.

Proposition 2.3. *Context updating satisfies the following.*

Functionality: If $\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B] = \Delta'$ *and* $\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B] = \Delta''$, *then* $\Delta' = \Delta''$;

Reflexivity: If $x :^{\alpha} A$ *appears at the* n*-th position of* Δ, *then* $\Delta[x :^{\alpha} A \mapsto_n x :^{\alpha} A] = \Delta$;

Symmetry: $\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B] = \Delta'$ *implies* $\Delta'[y :^{\beta} B \mapsto_n x :^{\alpha} A] = \Delta$;

Transitivity: $\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B][y :^{\beta} B \mapsto_n z :^{\gamma} C] = \Delta[x :^{\alpha} A \mapsto_n z :^{\gamma} C]$;

Confluence: $\Delta[x_1 :^{\alpha_1} A_1 \mapsto_n x_2 :^{\alpha_2} A_2][y_1 :^{\beta_1} B_1 \mapsto_m y_2 :^{\beta_2} B_2] = \Delta[y_1 :^{\beta_1} B_1 \mapsto_m y_2 :^{\beta_2} B_2][x_1 :^{\alpha_1} A_1 \mapsto_n x_2 :^{\alpha_2} A_2]$; *and*

Distributivity over $\bowtie_\circ$: $\Delta_1[x :^{\alpha_1} A \mapsto_n y :^{\beta_1} B] \bowtie_\circ \Delta_2[x :^{\alpha_2} A \mapsto_n y :^{\beta_2} B] = (\Delta_1 \bowtie_\circ \Delta_2)[x :^{\alpha} A \mapsto_n y :^{\beta_1 \circ \beta_2} B]$.

We also have various look-up properties:

Proposition 2.4. *Context look-up satisfies the following.*

Uniqueness: If $x :^{\alpha} A \in_n \Delta$ *and* $y :^{\beta} B \in_n \Delta$, *then* $x = y$, $A = B$, *and* $\alpha = \beta$;

Preservation under update: If $x :^{\alpha} A \in_n \Delta$, $\Delta[y :^{\beta} B \mapsto_m z :^{\gamma} C] = \Delta'$, *and* $n \neq m$, *then* $x :^{\alpha} A \in_n \Delta'$;

Preservation under splits: If $x :^{\alpha} A \in_n \Delta$ *and* $\Delta_1 \bowtie_\circ \Delta_2 = \Delta$, *then* $x :^{\alpha_1} A \in_n \Delta_1$ *and* $x :^{\alpha_2} A \in_n \Delta_2$ *for some* α_1, α_2 *such that* $\alpha_1 \circ \alpha_2 = \alpha$; *and*

Preservation under merge: If $x :^{\alpha} A \in_n \Delta_1$ *and* $\Delta_1 \bowtie_\circ \Delta_2 = \Delta$, *then* $x :^{\alpha_2} A \in_n \Delta_2$ *and* $x :^{\alpha} A \in_n \Delta$ *for some* α_2, α *such that* $\alpha_1 \circ \alpha_2 = \alpha$.

For some of the properties mentioned above, allowing indices to be arbitrary necessitates additional assumptions about well-formedness. Updating interacts with well-formedness as expected:

Proposition 2.5. *Let* Δ *be well-formed and* $x :^{\alpha} A \in_n \Delta$. *Then the following properties hold:*

Variable uniqueness: If $y :^{\beta} B \in_m \Delta$, *then* $x = y$ *if and only if* $n = m$; *and*

Preservation under update: If $x = y$ *or* $y \notin \Delta$, *then* $\Delta[x :^{\alpha} A \mapsto_n y :^{\beta} B]$ *is well-formed for any* β, B.

3 CARVe in Action

In this section we will present a high-level look at how systems may be reformulated using CARVe. (We will discuss their actual mechanizations together with more low-level details in section 4.) We revisit the linear sequent calculus introduced in section 1 alongside a bidirectional linear natural deduction calculus with proof terms, and sketch out a proof of their equivalence.

3.1 Reformulating with CARVe

Let us use the turnstile $\Vdash$ to distinguish typing judgments in the CARVe setting. We will parameterize contexts by the monoid $\mathcal{L}$ of subsection 2.1; for simplicity of notation, we will write $\bowtie$ for $\bowtie_\bullet$ throughout this section.

Consider first the hypothesis rule of the linear sequent calculus. If $\Delta \Vdash A$ is derived therefrom, then A appears as the type of some linear variable somewhere in Δ, and "using it up" results in an exhausted context:

$$\frac{\Delta[x :^1 A \mapsto x :^0 A] = \Delta' \quad \mathsf{exh}(\Delta')}{\Delta \Vdash A}$$

If $\Delta \Vdash A \multimap B$ is derived from the $\multimap$ right rule, then adding a fresh linear variable to Δ of type A should give us B. We follow the principle that, reading derivations bottom-up, contexts grow with new variables pushed to the top of the context.

$$\frac{\Delta, x :^1 A \Vdash B}{\Delta \Vdash A \multimap B}$$

If $\Delta \Vdash C$ is derived from the $\multimap$ left rule, then (1) $x :^1 A \multimap B$ appears somewhere in Δ for some x; and (2) using x results in a context that can be split into subcontexts Δ_1 and Δ_2 such that (a) $\Delta_1 \Vdash A$ and (b) $\Delta_2, y :^1 B \Vdash C$:

$$\frac{\begin{array}{cc}\Delta_1 \bowtie \Delta_2 = \Delta' & \Delta_1 \Vdash A \\ \Delta[x :^1 A \multimap B \mapsto x :^0 A \multimap B] = \Delta' & \Delta_2, y :^1 B \Vdash C\end{array}}{\Delta \Vdash C}$$

Reformulating the cut rule is similar.

$$\frac{\Delta_1 \bowtie \Delta_2 = \Delta \quad \Delta_1 \vdash A \quad \Delta_2, x :^1 A \vdash B}{\Delta \vdash B}$$

Next, we formulate a bidirectional linear natural deduction calculus in CARVe following the same principles.

$$\frac{\Delta \Vdash e \Leftarrow A}{\Delta \Vdash (e : A) \Rightarrow A} \qquad \frac{\Delta \Vdash e \Rightarrow A}{\Delta \Vdash e \Leftarrow A}$$

$$\frac{\Delta[x :^1 A \mapsto x :^0 A] = \Delta' \quad \mathsf{exh}(\Delta')}{\Delta \Vdash x \Rightarrow A} \qquad \frac{\Delta, x :^1 A \Vdash e \Leftarrow B}{\Delta \Vdash \lambda x.\, e \Leftarrow A \multimap B}$$

$$\frac{\Delta_1 \bowtie \Delta_2 = \Delta \quad \Delta_1 \Vdash e_1 \Rightarrow A \multimap B \quad \Delta_2 \Vdash e_2 \Leftarrow A}{\Delta \Vdash e_1 e_2 \Rightarrow B}$$

3.2 Equivalence Theorem

To showcase how CARVe lends itself well to proofs using simultaneous substitution, we take this approach to prove the two linear systems' equivalence. A benefit of this proof is that it preserves the structure of the derivation between the two systems.

We define well-typed simultaneous substitutions (judgment $\Delta_1 \Vdash \sigma : \Gamma$), which map variables to terms, as follows.

$$\frac{\mathsf{exh}(\Delta)}{\Delta \Vdash \cdot : \cdot} \qquad \frac{\Delta_1 \Vdash \sigma : \Gamma \quad \Delta_2 \Vdash e \Leftarrow C \quad \Delta_1 \bowtie \Delta_2 = \Delta}{\Delta \Vdash (\sigma, e) : (\Gamma, x :^1 C)}$$

$$\frac{\Delta \Vdash \sigma : \Gamma \quad \Delta' \Vdash e \Leftarrow C \quad 0\Delta = 0\Delta'}{\Delta \Vdash (\sigma, e) : (\Gamma, x :^0 C)}$$

As per [50], since the structure of the typing contexts remain intact as their resources are subdivided, we avoid splitting the substitution. In the final rule, the premise $0\Delta = 0\Delta'$ permits e to use an arbitrary collection of assumptions in Δ, regardless of their availability.

We state three properties of this substitution:

Lemma 3.1 (Properties of substitution)**.**

1. *Exhaustedness: If* $\Delta \Vdash \sigma : \Gamma$ *and* $\mathsf{exh}(\Gamma)$*, then* $\mathsf{exh}(\Delta)$*;*
2. *Merge: If* $\Delta \Vdash \sigma : \Gamma$ *and* $\Gamma = \Gamma_1 \bowtie \Gamma_2$*, then* $\Delta_1 \Vdash \sigma : \Gamma_1$ *and* $\Delta_2 \Vdash \sigma : \Gamma_2$ *for some* Δ_1, Δ_2 *such that* $\Delta = \Delta_1 \bowtie \Delta_2$*.*
3. *Resource consumption: If* $\Delta \Vdash \sigma : \Gamma$ *and* $\Gamma[x :^1 A \mapsto x :^0 A] = \Gamma'$*, then* $\Delta_1 \Vdash \sigma : \Gamma'$ *and* $\Delta_2 \Vdash e \Leftarrow A$ *for some* Δ_1, Δ_2, e *such that* $\Delta = \Delta_1 \bowtie \Delta_2$*.*

The second lemma asserts that substitutions remain stable under context splits, and the third enables one to "carry over" a substitution term to an updated context.

We are now prepared to sketch a proof of the two systems' equivalence. Throughout, we will assign names to derived judgments for a later comparison with a mechanized proof of the same in section 4.2.

Theorem 3.2 (Equivalence)**.**

1. *If* $\Gamma \Vdash C$ *and* $\Delta \Vdash \sigma : \Gamma$*, then* $\Delta \Vdash e \Leftarrow C$ *for some* e*;*
2. *If* $\Delta \Vdash e \Leftarrow C$*, then* $\Delta \Vdash C$*; and*
3. *If* $\Delta_1 \Vdash e \Rightarrow A$ *and* $\Delta_2, x :^1 A \Vdash C$*, and* $\Delta_1 \bowtie \Delta_2 = \Delta$*, then* $\Delta \Vdash C$*.*

Proof. The proof of the first statement by structural induction on the first typing derivation. Let us denote by $\mathcal{D}, \mathcal{S}$ the initial assumptions. We consider two representative cases.

First, suppose that the last rule in the typing derivation was (hyp). Then

a. $\mathcal{U} : \Gamma[x :^1 C \mapsto x :^0 C] = \Gamma'$ and $\mathcal{E}_1 : \mathsf{exh}(\Gamma')$ by inversion on $\mathcal{D}$
b. $\mathcal{S}_1 : \Delta_1 \Vdash \sigma : \Gamma'$ and $\mathcal{C}_1 : \Delta_2 \Vdash e \Leftarrow C$, and $\mathcal{M}_1 : \Delta_1 \bowtie \Delta_2 = \Delta$ for some e, Δ_1, Δ_2 by Lemma 3.1 (3) using $\mathcal{S}, \mathcal{U}$
c. $\mathcal{E}_2 : \mathsf{exh}(\Delta_1)$ by Lemma 3.1 (1) using $\mathcal{S}_1, \mathcal{E}_1$
d. $\Delta_2 = \Delta$ by identity property of $\bowtie$ using $\mathcal{M}_1, \mathcal{E}_2$

e. $\Delta \Vdash e \Leftarrow C$ by C_1

The second case we consider is where the last rule in the typing derivation was ($\multimap$L). In this case:

a. $\mathcal{U} : \Gamma[x :^1 A \multimap B \mapsto x :^0 A \multimap B] = \Gamma'$, $\mathcal{M}_1 : \Gamma_1 \bowtie \Gamma_2 = \Gamma'$, $\mathcal{D}_1 : \Gamma_1 \Vdash A$, and $\mathcal{D}_2 : \Gamma_2, y :^1 B \Vdash C$ by inversion on $\mathcal{D}$
b. $\mathcal{S}_1 : \Delta_{1,2} \Vdash \sigma : \Gamma'$, $\mathcal{D}_1 : \Delta_3 \Vdash e_1 \Leftarrow A \multimap B$, and $\mathcal{M}_2 : \Delta_{1,2} \bowtie \Delta_3 = \Delta$ for some $e_1, \Delta_{1,2}, \Delta_3$ by Lemma 3.1 (3) using $\mathcal{U}, \mathcal{S}$
c. $\mathcal{S}_2 : \Delta_1 \Vdash \sigma : \Gamma_1$, $\mathcal{S}_3 : \Delta_2 \Vdash \sigma : \Gamma_2$, and $\mathcal{M}_3 : \Delta_1 \bowtie \Delta_2 = \Delta_{1,2}$ for some Δ_1, Δ_2 by Lemma 3.1 (2) with $\mathcal{S}_1$, $\mathcal{M}_1$
d. $\mathcal{M}_3 : \Delta_2 \bowtie \Delta_{3,1} = \Delta$, $\mathcal{M}_4 : \Delta_3 \bowtie \Delta_1 = \Delta_{3,1}$ for some $\Delta_{3,1}$ by assoc., comm. of $\bowtie$ using $\mathcal{M}_2$,$\mathcal{M}_3$
e. $\mathcal{D}_2 : \Delta_1 \Vdash e_2 \Leftarrow A$ for some e_2 by I.H. using $\mathcal{D}_1$, $\mathcal{S}_2$
f. $\mathcal{D}_3 : \Delta_{3,1} \Vdash e_1(e_2 : A \multimap B) \Rightarrow B$ by (coe), ($\multimap$E) with $\mathcal{D}_1$,$\mathcal{D}_2$,$\mathcal{M}_4$
g. $\mathcal{D}_3' : \Delta_{3,1} \Vdash e_1(e_2 : A \multimap B) \Leftarrow B$ by (conv) with $\mathcal{D}_3$
h. $\mathcal{S}_4 : \Delta \Vdash \sigma_2, e_1\ e_2 : \Gamma_2, y :^1 B$ by def. using $\mathcal{S}_3$, $\mathcal{D}_3'$, $\mathcal{M}_5$
i. $\Delta \Vdash e \Leftarrow C$ for some e by I.H. using $\mathcal{D}_2$, $\mathcal{S}_4$

The proofs of the second and third statements are by mutual induction on the first derivation, appealing to the algebraic properties of merge. □

4 Implementation and Case Studies

The infrastructure presented in section 2 has been implemented in Beluga [40], a dependently-typed proof environment based on the logical framework LF, which uses an underlying λ-calculus as a meta-language for representing and reasoning about deductive systems [29]. In this section we will present an overview of the implementation and the case studies encoded. A table summarizing lemma usage in the mechanizations may be found in subsection 4.3. We refer the reader to the artifact [59] containing the formalization for full details.

4.1 Implementing CARVe

Building on Crary's [15] study of explicit contexts in LF, we use higher-order representations for syntax and represent object-level substructural typing contexts as lists. Their type family—indexed by length—is defined in the standard way.

```
LF lctx : nat → type =
| nil : lctx zero
| cons : lctx N → obj → tp → mult → lctx (suc N);
```

The objects that appear as resources in explicit contexts (*e.g.*, variables, channel names) are assigned the type `obj : type`. When these are simply variables, the type will be defined with no constructors; instead, assumptions of type `obj` will be collected in an LF context Ψ classified by the schema `schema ctx = obj`. (In a nameless setting, one may leave the LF context empty so long as a constructor is defined for `obj`.)

The type `tp : type` encodes object-level types, and `mult : type` multiplicities. In all but one of the case studies in this section, contexts will be parameterized by the monoid $\mathcal{L}$.

```
LF mult : type =
| 𝟘 : mult
| 𝟙 : mult;

LF • : mult → mult → mult → type =
| •/00 : • 𝟘 𝟘 𝟘
| •/10 : • 𝟙 𝟘 𝟙
| •/01 : • 𝟘 𝟙 𝟙;
```

Since the general definition of exhausted contexts is independent of any specific algebraic structure, we define the allowed "harmless" multiplicities using the type `hal`:

```
LF hal : mult → type =
| hal/0 : hal 𝟘;
```

Context Operations. Merging, updating, looking up a variable, and exhaustedness checks are represented in LF by the following.

```
LF merge : lctx N → lctx N → lctx N → type =
| mg/n : merge nil nil nil
| mg/c : merge Δ1 Δ2 Δ → • α1 α2 α
  → merge (cons Δ1 X A α1) (cons Δ2 X A α2) (cons Δ X A α);

LF upd : lctx N → nat → obj → obj → tp → tp → mult → mult
  → lctx N → type =
| upd/t : {Δ:lctx N}
    upd (cons Δ X A α) (suc N) X Y A B α β (cons Δ Y B β)
| upd/n : upd Δ n X Y A B α β Δ'
  → upd (cons Δ Z C γ) n X Y A B α β (cons Δ' Z C γ);

LF lookup_n : obj → lctx _ → type =
| lookn : upd Δ _ X _ _ _ _ _ _ → lookup_n X Δ;

LF exh : lctx _ → type =
| exh/n : exh nil
| exh/c : exh Δ → hal α → exh (cons Δ _ _ α);
```

The underscores _ above indicate holes that may be inferred by Beluga's type reconstruction.

Well-formedness. Our implementation of CARVe cannot enforce well-formedness at the LF level where we specify typing rules, since variables are maintained at the meta-level. While HOAS directly enforces the freshness of *new* variables, this is not "known" at the meta-level. We eschew non-declarative solutions such as in [15] and encode well-formedness as a meta-level predicate on contexts. Specifically, we define a computation-level inductive type Wf indexed by a typing context $[\Psi \vdash \Delta]$ as a *contextual object* [38]. (Here Ψ is the ambient LF context for variable names and Δ is the explicit typing context.)

```
false : type.
inductive Wf : (Ψ:ctx) {Δ:[Ψ ⊢ lctx N]} ctype =
| Wf/n : Wf [Ψ ⊢ nil]
| Wf/c : Wf [Ψ ⊢ Δ] → ([Ψ ⊢ lookup_n #p Δ] → [ ⊢ false])
      → Wf [Ψ ⊢ cons Δ #p A α];
```

The parentheses around the parameter Ψ:**ctx** indicate that it is treated implicitly. The constructor `Wf/n` defines the well-formedness of the empty context under any LF context. The second constructor `Wf/c` specifies the inductive case where we can extend Δ with x only if x is a parameter variable (enforced by the tag #) and is not in the context's domain. We do so with a function that produces from an object of type $[\Psi \vdash x \in \Delta]$ an object of the uninhabited type $[\vdash \text{false}]$, representing a contradiction.

Properties. Let us consider the commutativity of merge as an example of how properties about CARVe context operations may be represented in Beluga. The proof is implemented as a total recursive function.

```
rec merge_comm :
  (Ψ:ctx) [Ψ ⊢ merge Δ1 Δ2 Δ] → [Ψ ⊢ merge Δ2 Δ1 Δ] =
```

The above type signature states that, given a proof of $\Delta_1 \bowtie \Delta_2 = \Delta$ under the LF context Ψ, we may obtain a proof of $\Delta_2 \bowtie \Delta_1 = \Delta$. The property is easily proved by pattern matching:

```
  fn mg ⇒ % introduce object of type [Ψ ⊢ merge Δ1 Δ2 Δ]
  case mg of % pattern match on mg
  | [Ψ ⊢ mg/n] ⇒ [Ψ ⊢ mg/n] % nil case
  | [Ψ ⊢ mg/c M1 T1] ⇒ % cons case
    let [Ψ ⊢ M2] = merge_comm [Ψ ⊢ M1] in % invoke IH
    let [Ψ ⊢ T2] = mult_comm [Ψ ⊢ T1] in % lemma
    [Ψ ⊢ mg/c M2 T2];
```

where the commutativity of •

```
rec mult_comm : [ ⊢ • α1 α2 α] → [ ⊢ • α2 α1 α] = ... ;
```

is used as a lemma in the induction step.

4.2 Case Studies

We have used CARVe to mechanize and prove meta-theoretical properties about a variety of formal systems: the linear sequent calculus, the bidirectional linear natural deduction calculus, the session-typed process calculus CP, the linear and the affine λ-calculus. For space considerations, we will narrow our focus on the first three systems, while briefly touching on some distinguishing features of the others.

Linear Natural Deduction and Sequent Calculi. Let us restrict our attention to the implicational fragments of these systems introduced in section 3. To represent terms from the linear natural deduction calculus, we reuse the type **obj**, making use of HOAS to encode λ-terms.

```
LF obj : type =
| coerce : obj → tp → obj
| lam : (obj → obj) → obj
| app : obj → obj → obj;
```

```
LF tp : type =
| base : tp
| -o : tp → tp → tp;
--infix -o 5 right.
```

We represent the typing judgments $\Delta \Vdash e \Rightarrow A$ and $\Delta \Vdash e \Leftarrow A$ using mutually-recursive data-types

```
LF syn : lctx _ → obj → tp → type =
| coe : chk Δ e A → syn Δ (coerce e A) A
| init : upd Δ _ X X A A 𝟙 𝟘 Δ' → exh Δ' → syn Δ X A
| E-o : syn Δ1 s (A -o B) → chk Δ2 e A
  → merge Δ1 Δ2 Δ → syn Δ (app s e) B

and LF chk : lctx _ → obj → tp → type =
| conv : syn Δ e A → chk Δ e A
| I-o : ({x:obj} chk (cons Δ x A 𝟙) (e x) B)
  → chk Δ (lam e) (A -o B);
```

and the sequent calculus typing judgment $\Delta \Vdash A$ by:

```
LF seq : lctx _ → tp → type =
| var : upd Δ _ X X A A 𝟙 𝟘 Δ' → exh Δ' → seq Δ A
| cut : merge Δ1 Δ2 Δ
  → seq Δ1 A → ({x:obj} seq (cons Δ2 x A 𝟙) C)
  → seq Δ C
| R-o : ({x:obj} seq (cons Δ x A 𝟙) B) → seq Δ (A -o B)
| L-o : upd Δ _ X X (A -o B) (A -o B) 𝟙 𝟘 Δ'
  → merge Δ1 Δ2 Δ'
  → seq Δ1 A → ({x:obj} seq (cons Δ2 x B 𝟙) C)
  → seq Δ C;
```

We encode simultaneous substitutions σ as lists of terms:

```
LF subst : nat → type =
| empty : subst zero
| scons : subst N → obj → subst (suc N);
```

Next, the well-typed substitution judgment $\Delta \Vdash \sigma : \Gamma$ provides a mapping from σ to each variable in Γ.

```
LF wf_subst : lctx _ → subst N → lctx N → type =
| wf_subst_empty : exh Δ → wf_subst Δ empty nil
| wf_subst_cons1 : wf_subst Δ1 σ Γ
  → chk Δ2 e T → merge Δ1 Δ2 Δ
  → wf_subst Δ (scons σ e) (cons Γ _ T 𝟙)
| wf_subst_cons0 : wf_subst Δ σ Γ
  → chk Δ' e T → same_elts Δ Δ'
  → wf_subst Δ (scons σ e) (cons Γ _ T 𝟘);
```

The assumption **same_elts** Δ Δ' corresponds to the premise $0\Delta = 0\Delta'$ in the on-paper definition.

As Beluga does not support existential quantification directly, we encode the existence of a term e such that $\Delta \Vdash e \Leftarrow C$ with the type **inhabit**.

```
LF inhabit : lctx _ → tp → type =
| inh : chk Δ _ C → inhabit Δ C;
```

We are now ready to present the forward direction of the equivalence proof. Figure 1 includes the two cases presented in subsection 3.1, where the code is annotated with the corresponding line numbers from the earlier on-paper proof. The proofs are identical to that presentation, with two exceptions, marked with an asterisk: the substitution judgment must at one point be unboxed, and we must invoke a "pruning" lemma to strengthen the LF context. Signatures of all lemmas used may be found in Appendix A.

The artifact contains the proof of the other direction as well as a proof of cut elimination for the linear sequent calculus.

```
rec seq2nd : (Ψ:ctx) [Ψ ⊢ seq Γ C] → [Ψ ⊢ wf_subst Δ σ Γ]
  → [Ψ ⊢ inhabit Δ C] =
  fn d, s ⇒ case d of
  % (hyp) case
  | [Ψ ⊢ var U E1] ⇒                                  % a
    let [Ψ ⊢ S] = s in                                % *
    let [Ψ ⊢ sub-up S1 C1 M1 _ _] =
       subst_upd [Ψ ⊢ S] [Ψ ⊢ U] in                   % b
    let E2 = subst_exh [Ψ ⊢ S1] [Ψ ⊢ E1] in           % c
    let [Ψ ⊢ cx/refl] = merge_id [Ψ ⊢ M1] E2 in        % d
    [Ψ ⊢ inh C1]                                      % e
  % (-o L) case
  | [Ψ ⊢ L-o U M1 D1 \x.D2] ⇒                         % a
    let [Ψ ⊢ S] = s in                                % *
    let [Ψ ⊢ sub-up S1 D1 M2 _ _] =
      subst_upd [Ψ ⊢ S] [Ψ ⊢ U] in                    % b
    let [Ψ ⊢ sub-mg S2 S3 M3 _ _] =
      subst_merge [Ψ ⊢ S1] [Ψ ⊢ M1] in                % c
    let [Ψ ⊢ M3'] = merge_comm [Ψ ⊢ M3] in            % d
    let [Ψ ⊢ mg-assoc M4' M5 _ _] =
      merge_assoc [Ψ ⊢ M2] [Ψ ⊢ M3'] in               % d
    let [Ψ ⊢ M4] = merge_comm [Ψ ⊢ M4'] in            % d
    let [Ψ ⊢ inh D2] = seq2nd [Ψ ⊢ D1] [Ψ ⊢ S2] in    % e
    let [Ψ ⊢ D3] = [Ψ ⊢ E-o (coe D1) D2 M4] in        % f
    let [Ψ ⊢ D3'] = [Ψ ⊢ conv D3] in                  % g
    let [Ψ ⊢ S4] = [Ψ ⊢ wf_subst_cons1 S3 D3' M5] in  % h
    let [_,x:obj ⊢ inh D4'] =
      seq2nd [Ψ,x:obj ⊢ D2] [Ψ,x:obj ⊢ S4[..]] in     % i
    let Prune-Chk [Ψ ⊢ D4] [Ψ,x:obj ⊢ _] =
      prune_chk [Ψ,x:obj ⊢ D4'] in                    % *
    [Ψ ⊢ inh D4]
```

Figure 1. Code fragment of one direction of the equivalence proof

CP. Another system encoded using CARVe is the multiplicative-additive fragment of the session-typed process calculus CP [54]. CP's process language is a variant of the π-calculus, and its type system is obtained directly from classical linear logic [26]. In session-typed systems, variables in typing contexts represent *names of channels*, communication over which is prescribed by their *session types*: we interpret the typing judgment $P \vdash x_1 : A_1, \ldots, x_n : A_n$ as stating "process P uses each channel x_i exactly once according to protocol A_i."

For instance, the two structural rules in classical linear logic—identity and cut—correspond to message forwarding and parallel composition, respectively. The process $x \leftrightarrow y$ links two dual channels, under the condition that all other channels have already been used. The process $\nu x : A.(P \parallel Q)$ spawns a fresh channel with dual endpoints along which processes P and Q can communicate.

$$\frac{\mathsf{exh}(\Delta[y :^1 A^\perp \mapsto y :^0 A^\perp][x :^1 A \mapsto x :^0 A])}{x \leftrightarrow y \vdash \Delta} \text{ (hyp)}$$

$$\frac{P \vdash \Delta_1, x : A \quad Q \vdash \Delta_2, x : A^\perp \quad \Delta_1 \bowtie \Delta_2 = \Delta}{\nu x : A.(P \parallel Q) \vdash \Delta} \text{ (cut)}$$

In the forwarding rule, the choice to consume y before x is arbitrary; by confluence (Proposition 2.3), we could well have done the reverse.

The internal choice processes $x[\mathsf{inl}]; P$ and $x[\mathsf{inr}]; P$ send a binary choice label along x and continue as P. Dually, the external choice process $x.\mathsf{case}(P, Q)$ offers a binary choice on x and continues as *either* P or Q, depending on the choice received.

$$\frac{P \vdash \Delta[x :^1 A \oplus B \mapsto x :^0 A \oplus B], x' : A}{x[\mathsf{inl}]; P \vdash \Delta} \ (\oplus_1)$$

$$\frac{P \vdash \Delta[x :^1 A \oplus B \mapsto x :^0 A \oplus B], x' : B}{x[\mathsf{inr}]; P \vdash \Delta} \ (\oplus_2)$$

$$\frac{Q \vdash \Delta[x :^1 A \,\&\, B \mapsto x :^0 A \,\&\, B], x' : B \quad P \vdash \Delta[x :^1 A \,\&\, B \mapsto x :^0 A \,\&\, B], x' : A}{x.\mathsf{case}(P, Q) \vdash \Delta} \ (\&)$$

In encoding the typing rules above using CARVe, we have adopted the continuation-passing principle [17]. That is, each channel is treated as a single-use entity carrying exactly one message: when communication occurs, the channel is closed and a fresh continuation channel is created and bound to the continuation process. This style has the benefit of making dependencies explicit, and is in keeping with our philosophy of varying multiplicities while keeping contexts intact. It also makes our encoding adequate with respect to Structural CP (SCP) [47], which requires continuations. We note, however, that we can encode CP in CARVe using the standard approach of re-using channels' names.

In LF, the binding of continuation channels are represented using HOAS:

```
LF obj : type =
| fwd : obj → obj → obj
| pcomp : tp → (obj → obj) → (obj → obj) → obj
| inl : obj → (obj → obj) → obj
| inr : obj → (obj → obj) → obj                ... ;
```

We then encode the CP typing judgment in LF as the type family `oft`:

```
LF oft : obj → lctx _ → type =

| oft/fwd : dual A A' → upd Δ _ Y Y A' A' 𝟙 𝟘 Δ'
  → upd Δ' _ X X A A 𝟙 𝟘 Δ'' → exh Δ''
  → oft (fwd X Y) Δ

| oft/pcomp : dual A A' → merge Δ1 Δ2 Δ
  → ({x:obj} oft (P x) (cons Δ1 x A 𝟙))
  → ({x:obj} oft (Q x) (cons Δ2 x A' 𝟙))
  → oft (pcomp A P Q) Δ

| oft/inl : upd Δ _ X X (A ⊕ B) (A ⊕ B) 𝟙 𝟘 Δ'
  → ({x:obj} oft (P x) (cons Δ' x A 𝟙))
```

```
  → oft (inl X P) Δ

| oft/inr : upd Δ _ X X (A ⊕ B) (A ⊕ B) 𝟙 𝟘 Δ'
  → ({x:obj} oft (P x) (cons Δ' x B 𝟙))
  → oft (inr X P) Δ

| oft/choice : upd Δ _ X X (A & B) (A & B) 𝟙 𝟘 Δ'
  → ({x:obj} oft (P x) (cons Δ' x A 𝟙))
  → ({x:obj} oft (Q x) (cons Δ' x B 𝟙))
  → oft (choice X P Q) Δ                    ... ;
```

We mechanize two main results about CP. The first is type preservation, whose proof relies on a renaming lemma and makes heavy use of the algebraic properties of update and merge. The second is the equivalence of CP with SCP. SCP is an alternative type structure for CP processes that locally enforces linearity with a logical predicate. In short, in SCP the typing rule for any process P that binds a linear variable x includes a condition that it is used linearly, encoded as a predicate `linear : (obj → proc) → type`. The typing judgment is therefore encoded as a judgment on a process `wtp : proc → type`, while the typing of channel names uses a hypothetical judgment of type `hyp : obj → tp → type` kept in an ambient LF context Φ following the schema

```
schema hctx = block ch:obj, h:hyp ch _;
```

We encode a bijective translation between the typing judgments of CP and SCP. To give this bijection, we need only consider typing and linearity (processes and types use the same syntax). To translate from CP to SCP, we define a relation `Enc [Ψ ⊢ Δ] $[Φ ⊢ $σ]` that translates a linear context Δ under Ψ into an intuitionistic context Φ and builds a weakening substitution σ mapping names in Ψ to ones in Φ. In the reverse direction, we must inspect the typing derivation of a process P to determine which channels are used linearly. We then use this information to assign multiplicities to variables in explicit contexts. The ternary relation `Dec [Ψ ⊢ P] [Ψ ⊢ Δ] $[Φ ⊢ $σ]` relates Δ and Φ, where the choice of multiplicities in Δ depends on P, and builds a weakening substitution mapping variables in Ψ to ones in Φ. We encode both context relations in Beluga as inductive datatypes, following [20]. For instance, the translation from linear to intuitionistic contexts is defined by

```
inductive Enc : (Ψ:ctx) (Φ:hctx)
  {Δ:[Ψ ⊢ lctx N]} {$σ:$[Φ ⊢ Ψ]} ctype =
| Enc/n : Enc [ ⊢ nil] $[ ⊢ ^]
| Enc/c : Enc [Ψ ⊢ Δ] $[Φ ⊢ $σ]
  → Enc [Ψ,ch:obj ⊢ cons Δ[..] ch A[] _]
  $[Φ,b:block ch:obj,h:hyp ch A[] ⊢ $σ[..],b.ch];
```

The `^` above denotes the empty substitution, and the weakening substitution `[]` specifies that the metavariable `A` is closed. The dollar signs `$` indicates that σ is a substitution variable of the specified substitution meta-type.

The fundamental lemma that makes this translation possible extracts a linearity judgment from a CP typing judgment.

```
rec oft_linear : (Ψ:ctx) [Ψ,x:obj ⊢ oft P (cons Δ x A 𝟙)]
  → [Ψ ⊢ linear \x.P] = ... ;
```

We then prove the equivalence of the two systems via the defined relations.

```
rec cp2scp : Enc [Ψ ⊢ Δ] $[Φ ⊢ $σ]
  → [Ψ ⊢ oft P Δ] → [Φ ⊢ wtp P[$σ]] = ... ;

rec scp2cp : Dec [Ψ ⊢ P] [Ψ ⊢ Δ] $[Φ ⊢ $σ]
  → [Φ ⊢ wtp P[$σ]] → [Ψ ⊢ oft P Δ] = ... ;
```

The first proof is by structural induction on the typing judgment, and the second on the encoding relation. This result serves as a formalized proof the "internal" adequacy of SCP with respect to a first-order encoding of CP. To our knowledge, this is the first mechanized adequacy proof in a session-typed setting.

Linear λ-Calculus. We mechanize the linear λ-calculus using CARVe in two styles: the first uses HOAS, and the second de Bruijn levels with an environment-based operational semantics. We prove type preservation in both settings. In the former encoding, we include exponentials by adjusting the underlying resource algebra per section 5, but omit them from the de Bruijn formulation for simplicity.

As the first approach is relatively standard, let us focus on the second formulation, which yields a compact type preservation proof without the need for a substitution lemma. In this setting, environments η are represented as lists of *closures* (pairs of environments and terms):

```
LF obj : type =
| var : nat → obj
| app : obj → obj → obj
| abs : obj → obj;

LF venv : nat → type =
| empty : venv zero
| vcons : venv N → val → venv (suc N)

and LF val : type =
| closure : venv _ → obj → val;
```

A relation of type `hasty_env : venv N → lctx N → type` associates an environment with a linear typing context of the same length, representing a simultaneous substitution and yielding value typing judgments `hasty : val → tp → type`.

Next, we encode environment evaluation ("under environment η, term M evaluates to value w"):

```
LF eval : venv _ → obj → val → type =
| eval/var : lookup_venv n W η → eval η (var n) W
| eval/abs : eval η (abs M) (closure η (abs M))
| eval/app : eval η M (closure η' (abs M'))
  → eval η N W' → eval (vcons η' W') M' W
  → eval η (app M N) W;
```

Note in particular that evaluating a variable n returns the value that appears at level n in the environment, and evaluating a function returns its closure.

Finally, we encode term typing judgments using CARVe constructs.

```
LF oft : lctx _ → obj → tp → type =
| oft/var : upd Δ n _ _ A A 𝟙 𝟘 Δ' → exh Δ'
  → oft Δ (var n) A
| oft/abs : oft (cons Δ _ A 𝟙) M B
  → oft Δ (abs M) (arrow A B)
| oft/app : oft Δ₁ M (arrow A B) → oft Δ₂ N A
  → merge Δ₁ Δ₂ Δ → oft Δ (app M N) B;
```

The proof of type preservation

```
rec tps : [ ⊢ eval η M W] → [ ⊢ hasty_env η Δ]
  → [ ⊢ oft Δ M T] → [ ⊢ hasty W T] = ... ;
```

is by structural induction on the evaluation judgment. Two main lemmas are used: one stating that the environment's domain is preserved under merge, and another about the correspondence between the context relation, context lookup, and value typing.

Affine λ-Calculus. Affine systems are characterized by forbidding contraction while allowing weakening. We capture them by the same monoid $\mathcal{L}$, provided that we modify the conditions at the leaves of a proof tree. Accordingly, the affine variable rule becomes

$$\frac{x :^1 A \in \Delta}{\Delta \Vdash A}$$

where we simply check that x occurs linearly.

The encoding of this system otherwise uses the same syntax and semantics as the linear case using HOAS, with exponentials omitted for simplicity. We again prove type preservation for this system. To prove the variable case of the substitution property, an additional lemma is needed stating that typing is preserved under merges.

4.3 Lemma Usage

In Table 1, we summarize some quantitative metrics concerning the usage of lemma in mechanizing the main theorem for each encoded system. The results are identified by their name in the code base (*e.g.*, "`tps`" instead of "type preservation"). We note the number of lines of the proof term for each result (under the heading "Theorem"), and analogously the number of lines devoted to proving lemmas ("Lemmas") or properties from the reusable common infrastructure ("Common") that—either directly or transitively—support the final proof. The reader will note a significant utilization of the common infrastructure, with the percentage of shared CARVe lemmas ranging between 42.6% and 77.6% of the overall code base for each development.

5 Varying the Algebra

So far, we have considered only one resource algebra. However, CARVe can be readily adapted to other scenarios by parameterizing contexts by other algebraic structures of interest. The structures described in this section have been implemented in Beluga with their algebraic properties. With few exceptions (isolated from the rest in the artifact), all proven lemmas about context operations hold regardless of the specific choice of structure.

Intuitionistic Assumptions. The trivial monoid

$$\mathcal{I} = (\{\omega\}, \{(\langle\omega, \omega\rangle, \omega)\}, \omega),$$

characterizes allocation for fully intuitionistic contexts and amounts to having no annotations at all. Here ω denotes a variable always available.

To model intuitionistic resources alongside linear or affine ones, we may simply enrich $\mathcal{L}$ with ω as a third element:

$\bullet$	0	1	ω
0	0	1	–
1	1	–	–
ω	–	–	ω

In regard to exhaustedness, we consider as harmless those elements of multiplicity either 0 or ω. While the structure is only a commutative semigroup, these harmless elements are partial units for 0 and 1 and for ω, respectively.

Parametrizing contexts by this structure allows one to encode exponential modalities in the style of dual intuitionistic linear logic [3] while making use of only one typing context. For example, in such a setting, the variable rule has two distinct forms:

$$\frac{\Delta[x :^1 A \mapsto x :^0 A] = \Delta' \quad \mathsf{exh}(\Delta')}{\Delta \Vdash A}\ (\mathsf{hyp}_1)$$

$$\frac{x :^\omega A \in \Delta \quad \mathsf{exh}(\Delta)}{\Delta \Vdash A}\ (\mathsf{hyp}_\omega)$$

Remark that the semigroup can be extended to a monoid by defining $\alpha \bullet \omega = \omega \bullet \alpha = 1 \bullet 1 = \omega$ for any α. This is the none-one-tons semiring of McBride [37] under addition.

Strict Assumptions. Strict (also called *relevant* or *relevance*) systems allow contraction and forbid weakening. In practice, strict assumptions must be used *at least* once in a derivation. We model such systems by reinterpreting $\mathcal{L}$ as the monoid $(\{1, \omega\}, \bullet, \omega)$, with $\bullet$ given by the following table:

$\bullet$	1	ω
1	–	1
ω	1	ω

In contrast with linear or affine systems, we also require both (hyp_1) and (hyp_ω) variable rules as in the joint linear-intuitionistic case.

The multiplicity 1 is used to denote a strict assumption. When used, its tag is changed not to 0 (unavailable) but to ω (available now unrestrictedly.) When a context is split, a strict assumption must still be used at least once in one of the branches but becomes available unrestrictedly in the other. A context is considered exhausted when only unrestricted assumptions remain.

Table 1. Summary of lemma usage in implementation

System	Result	Theorem	Lemmas	Common
Affine λ-calculus (`affine_lam`)	`tps`	22 (3.9 %)	105 (18.5 %)	441 (77.6 %)
Linear λ-calculus with HOAS (`lin_lam`)	`tps`	45 (5.8 %)	208 (27 %)	519 (67.2 %)
Linear λ-calculus with de Bruijn (`closures`)	`tps`	16 (15 %)	36 (35 %)	51 (50 %)
CP (cp)	`tps`	170 (14.5 %)	504 (42.9 %)	500 (42.6 %)
	`cp2scp`	53 (6.8 %)	293 (37.8 %)	429 (55.4 %)
	`scp2cp`	59 (17.4 %)	113 (33.2 %)	168 (49.4 %)
Linear sequent calculus (seq)	`cut_elim`	266 (24.4 %)	335 (30.7 %)	489 (44.9 %)
Linear sequent / natural deduction calculi (seq / nd)	`seq2nd`	75 (7.4 %)	447 (44.4 %)	485 (48.2 %)
	`chk2seq` / `syn2seq`	64 (10.8 %)	116 (19.5 %)	414 (69.7 %)

Graded Assumptions. The numerical monoid

$$\mathcal{G} = (\mathbb{N}, \{(\langle x, y\rangle, z) : x + y = z\}, 0)$$

characterizes allocation for graded contexts in the spirit of Orchard *et al.* [39]. Gradedness is a generalization of linearity: an assumption of multiplicity n must be used precisely n times. We thus consider as harmless—as in the linear case—assumptions of multiplicity 0.

6 Related Work

6.1 Explicit Contexts and HOAS

Crary [15] introduced the idea of mixing hypothetical and categorical judgments in a HOAS setting, where HOAS is reserved for syntax while typing contexts are treated as explicit objects in judgments such as typing rules. He also showed a translation between explicit (intuitionistic) contexts and implicit ones. This idea had been heavily used in the Twelf formalization of SML [34]. A similar approach underlines the implementation of the two-level approach (see for example [19]).

6.2 Resource Algebras

The idea of generalizing the linear discipline into an algebraic structure owes itself to two threads.

Bounded Linear Logic. In the early 1990s, Girard introduced *bounded linear logic* (BLL) [25], where a family of modalities $!_x A$ indicates that A may be reused up to x times. The resource polynomials of BLL were generalized using semirings in several works including [24], where ring addition controls contraction and multiplication bounds function usage. This allows one to track various properties—such as bounded reuse and strictness—within a single system. This approach was further generalized as graded modal logic in the design of the typed functional language Granule [39]. These ideas have not been widely adopted in the mechanized meta-theory community. Semiring annotations are present in quantitative type theory [2], but for a different purpose, namely to combine linear and dependent types. At the same time, researchers have mechanized some of the meta-theory of those type theories, (*e.g.*, [13]), thus endorsing a context management style similar to ours.

Separation Algebras. The heap semantics of the logic of bunched implications [44] was first abstracted into a partial commutative cancellative monoid by Calcagno *et al.* [6]. This was further refined by Dockins *et al.* [18] in view of their Coq implementation in the VST project; they embrace partiality by switching to a relational presentation of the monoidal operation and add axioms to exclude degenerate algebras while relaxing the unit conditions. Other slightly different axiomatizations are considered in [42] and [32].

6.3 Let's Split

When addressing the challenge multiplicative rules present to encoding contexts, the prevailing approach in mechanized meta-theory within mainstream proof assistants has been to physically partition the context, seen as a list or a finite map. Limiting ourselves to the meta-theory of the π-calculus, this approach has been used in [22, 27] and even extended to the intrinsically-typed approach [14, 46, 52].

Alternatively, binding contexts may be seen as dictionaries, *i.e.*, finite maps that can be split since they are known to have disjoint domains (either by α-renaming or by explicit conditions). The inspiration is again the heap model of separation logic. One example is the Coq library by Castro *et al.* [9], where splitting makes the context undefined when it would result in duplicated entries. The library is based on the formalization of finite maps in MathComp. Another is the pedagogical implementation of separation logic for the SF curriculum [11], where a finite map is a dependent packaging of a partial function with a proof of finiteness of its domain. Both libraries encompass an extensive collection of low-level lemmas and tactics, in both cases around 100 lemmas and 100 lines of Ltac code,

Another domain where splitting has been the dominant paradigm is the meta-theory of linear sequent calculi, namely cut-elimination and focusing. In contrast with binding contexts, contexts are here truly multisets, whose structure is generally imposed over the context seen as a list. The idea

Table 2. Summary of approaches to substructural context modeling

Paper	Application	Result	System	Syntax	Contexts
[1]	MA linear λ-calculus	type preservation (TPS)	Agda	scoped de Bruijn (DB)	leftover typing
[50]	explicit substitutions	TPS / confluence	Twelf	HOAS	tags
[13]	graded dependent type theory	TPS w.r.t. heap semantics	Coq	locally nameless	tags
[60]	π-calculus	TPS for capabilities types	Agda	scoped DB	leftover typing
[22]	π-calculus	TPS for linear type system	Isabelle / HOL	DB à la Gordon	list split
[27]	polymorphic π-calculus	TPS for session types	Coq	locally nameless	finite maps
[52]	functional session-typed calculus	TPS / session fidelity	Agda	intrinsically-typed DB	list split
[14]	π-calculus with dependent pairs	TPS by construction	Agda	intrinsically-typed DB	list split
[46]	π-calculus	TPS for session types	Agda	intrinsically-typed co-DB	split via bunched logic
[9]	π-calculus	TPS for session types	Coq	locally nameless	finite maps
[47]	π-calculus	TPS for session types	Beluga	HOAS	linearity predicate
[58]	sequents	cut elimination / focusing	Coq	parametric HOAS	lists with bag equivalence
[21, 33]	sequents	cut elimination / focusing	Coq	Hybrid	list with permutations
[12]	sequents	cut elimination	Abella	HOAS	multiset over lists
[28]	linear λ-calculus	type uniqueness	Abella	HOAS	multiset over lists

is then to see a context split as list concatenation modulo exchange. This is realized in different ways: Xavier *et al.* [58] use bag equivalence implemented as equal number of occurrences. Other authors rely on permutations, either assuming a structural rule of exchange [21, 33] or localizing permutations in multiplicative rules (*e.g.*, [7]). Chaudhuri *et al.* [12] instead encode multisets via a non-deterministic "cons" operation, by means of which merging and permutation of contexts are defined. The former is used as expected in multiplicative rules and the latter in additive ones. A library of some 80 lemmas about those predicates is provided. Other combinations have also been investigated [28].

6.4 Let's Stay Together

Keeping the context intact has been explored in various flavours.

Tags. The use of *type qualifiers*, which are first-class tags occurring in the syntax of types and terms encoding introduction rules, was pioneered by Walker [55] and adapted by Vascocelos [53] to the concurrent setting. As previously mentioned, our approach to meta-theory is built on the ideas of Schack-Nielsen and Schürmann [50], which were formulated to provide an efficient realization of the explicit substitution calculus underlying Celf's [49] operational semantics.

Wood and Atkey [56] annotate variables with values from a skew semiring denoting those variables' usage by terms. This also permits them to keep the context intact by simply updating the status of the variable. Their work extends McBride's kits and traversals technique to the quantitative / linear setting. This allows one to isolate properties required to form binding-respecting traversals of simply typed λ-terms, so that renaming and substitution arise as specific instantiations. In that setting, usage annotations on contexts are vectors, usage-preserving maps of contexts are matrices, and the linearity properties of the maps induced by matrices are exactly the lemmas needed for showing that traversals (and hence renaming, sub-usaging, and substitution) preserve typing and usages. It remains open how this technique can be applied to process calculi such as CP, or for proving meta-theory more generally.

Linearity Predicates. Crary [16], building on previously unpublished work by Pfenning, introduced the idea of separating a typing derivation from the check that it satisfies a given property, namely of being linear. While this fits well with existing non-substructural proof assistants and applies to different settings [47], it seems hard to generalize to more exotic substructural systems.

Leftovers. This is based on the idea that a linear term consumes some of the resources available in its context, while leaving behind leftovers which can then be fed to another program. Though the idea originated within linear functional and logic programming [30, 35], Allais [1] was the first to employ the idea for proving (in Agda) subject reduction for the linear λ-calculus. This was later extended to the π-calculus (with respect to capabilities types) by Zalakain and Dardha [60]. The Agda development is parameterized over a usage algebra following the definition of Dockins *et al.* [18]. It would be interesting to draw a quantitative comparison with our approach, although, as usual, the de Bruijn encoding in the cited papers tends to overwhelm the development. Note however that CARVe can be easily used to implement the leftover style, by appropriate updating of the linear tag; in fact, Allais uses annotations similar to ours.

The different approaches covered in subsection 6.3 and subsection 6.4 are summarized in Table 2.

6.5 Substructural Frameworks

A rather different approach is to let the logical framework where we encode our system under study be substructural. Examples include LLF [10], based on linear hereditary Harrop formulæ, Lincx [23] for a linearization of contextual LF,

HLF [45] for supporting hybrid LF, and Celf [49] for implementing Concurrent LF.

In all these systems, using HOAS and hypothetical judgments, resource contexts are *implicit*, bypassing (for the user) all the issues connected to resource management. This leads to elegant encoding of object logics such as linear sequent calculi, MiniML with references, security protocols, the π-calculus, and session types. However, none of these frameworks, bar Celf, have been implemented and—even in Celf's case—there is little support for verifying the meta-theory of the encoded systems.

7 Conclusion

In this work, we have presented CARVe, a flexible infrastructure for managing substructural contexts explicitly, which is fully implemented in the proof assistant Beluga. We have showcased the infrastructure's versatility by using it to mechanize a broad range of substructural systems and corresponding proofs. We anticipate that our development will prove useful for formalizing other substructural languages not considered in this paper.

Future Work. While monoids suffice for our study, the modularity of our approach should lend itself well to encoding systems using more intricate multiplicity structures. Specifically, while our study considered only substructural systems that control strengthening and weakening, modeling ordered type systems based on non-commutative logic (which restrict exchange) will require a richer resource algebra. In the future, we also intend to extend CARVe to model subexponential and adjoint modalities.

A further quantitative comparison of CARVe with alternative approaches would also offer insights into the relative strengths and limitations of our infrastructure.

Finally, it would be worthwhile to implement CARVe in a mainstream proof assistant to better leverage structuring mechanisms such as polymorphism, module systems, and type classes. This would allow us to go beyond the current limitations of Beluga and directly realize a library for binding contexts where arbitrary keys carry arbitrary payloads given some resource algebra. Additionally, integrating code generation into the implementation could facilitate the application of CARVe to new systems.

8 Data Availability Statement

An artifact accompanying this paper is available online [59].

Acknowledgments

The authors thank Ryan Kavanagh for discussions that contributed to the development of this work, and the anonymous reviewers for their constructive and insightful feedback.

This work was funded by the Natural Sciences and Engineering Research Council of Canada and the Fonds de recherche du Québec — Nature et technologies.

A Further Signatures

For reference, we include below the LF signatures of all constructs and lemmas left undefined in subsection 4.2 (simplified for readability).

Identity property of merge:

```
LF cx_eq : lctx N → lctx N → type =
| cx/refl : cx_eq Δ Δ;

rec merge_id : (Ψ:ctx) [Ψ ⊢ merge Δ1 Δ2 Δ]
  → [Ψ ⊢ exh Δ1] → [Ψ ⊢ cx_eq Δ2 Δ] = ... ;
```

Associativity of merge:

```
LF mg_assoc : merge _ _ _ → merge _ _ _ → type =
| mg-assoc : merge Δ2 Δ3 Δ23 → merge Δ1 Δ23 Δ
  → {M1:merge Δ12 Δ3 Δ} {M2:merge Δ1 Δ2 Δ12}
     mg_assoc M1 M2;

rec merge_assoc : (Ψ:ctx)
  {M1:[Ψ ⊢ merge Δ12 Δ3 Δ]} {M2:[Ψ ⊢ merge Δ1 Δ2 Δ12]}
  [Ψ ⊢ mg_assoc M1 M2] = ... ;
```

Lemma 3.1 (1):

```
rec subst_exh : (Ψ:ctx) [Ψ ⊢ wf_subst Δ σ Γ]
  → [Ψ ⊢ exh Γ] → [Ψ ⊢ exh Δ] = ... ;
```

Lemma 3.1 (2):

```
LF subst-merge : wf_subst _ _ _ → merge _ _ _ → type =
| sub-mg : wf_subst Δ1 σ Γ1 → wf_subst Δ2 σ Γ2
  → merge Δ1 Δ2 Δ
  → {S:wf_subst Δ σ Γ} {M:merge Γ1 Γ2 Γ}
  subst-merge S M;

rec subst_merge : (Ψ:ctx) {S:[Ψ ⊢ wf_subst Δ σ Γ]}
  {M:[Ψ ⊢ merge Γ1 Γ2 Γ]} [Ψ ⊢ subst-merge S M] = ... ;
```

Lemma 3.1 (3):

```
LF subst-upd :
  wf_subst _ _ _ → upd _ _ _ _ _ _ _ _ _ → type =
| sub-up : wf_subst Δ1 σ Γ' → chk Δ2 _ A
  → merge Δ1 Δ2 Δ
  → {S:wf_subst Δ σ Γ} {U:upd Γ _ _ _ A A 𝟙 𝟘 Γ'}
    subst-upd S U;

rec subst_upd : (Ψ:ctx) {S:[Ψ ⊢ wf_subst Δ σ Γ]}
  {U:[Ψ ⊢ upd Γ _ _ _ A A 𝟙 𝟘 Γ']}
  [Ψ ⊢ subst-upd S U] = ... ;
```

Pruning lemma:

```
inductive PruneChk : (Ψ:ctx)
  {CH:[Ψ,x:obj ⊢ chk Δ[..] M C[]]} ctype =
| Prune-Chk : [Ψ ⊢ chk Δ M C[]]
  → {CH:[Ψ,x:obj ⊢ chk Δ[..] M[..] C[]]}
     PruneChk [Ψ,x:obj ⊢ CH];

rec prune_chk : {CH:[Ψ,x:obj ⊢ chk Δ[..] M C[]]}
  PruneChk [Ψ,x:obj ⊢ CH] = ... ;
```

References

[1] Guillaume Allais. 2017. Typing with leftovers: A mechanization of intuitionistic multiplicative-additive linear logic. In *Proc. TYPES 2017 (LIPIcs, Vol. 104)*, Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi (Eds.). 1:1–1:22. https://doi.org/10.4230/LIPICS.TYPES.2017.1

[2] Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proc. LICS '18*. 56–65. https://doi.org/10.1145/3209108.3209189

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

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

[5] Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *Proc. CONCUR '10 (Lect. Notes Comput. Sci., Vol. 6269)*, Paul Gastin and François Laroussinie (Eds.). Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

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

[7] Étienne Callies and Olivier Laurent. 2021. Click and coLLecT: An interactive linear logic prover. In *Proc. TLLA '21*. https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271501

[8] 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. 2024. The concurrent calculi formalisation benchmark. In *Proc. COORDINATION '24 (Lect. Notes Comput. Sci., Vol. 14676)*, Ilaria Castellani and Francesco Tiezzi (Eds.). 149–158. https://doi.org/10.1007/978-3-031-62697-5_9

[9] David Castro, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the meta-theory of session types. In *Proc. TACAS '20 (Lect. Notes Comput. Sci., Vol. 12079)*, Armin Biere and David Parker (Eds.). 278–285. https://doi.org/10.1007/978-3-030-45237-7_17

[10] Iliano Cervesato and Frank Pfenning. 2002. A linear logical framework. *Inf. Comput.* 179, 1 (2002), 19–75. https://doi.org/10.1006/INCO.2001.2951

[11] Arthur Charguéraud. 2024. *Separation Logic Foundations*. Software Foundations, Vol. 6. Electronic textbook. Version 2.2.

[12] Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. 2019. Formalized meta-theory of sequent calculi for linear logics. *Theor. Comput. Sci.* 781 (2019), 24–38. https://doi.org/10.1016/j.tcs.2019.02.023

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

[14] Luca Ciccone and Luca Padovani. 2020. A dependently typed linear π-calculus in Agda. In *Proc. PPDP '20*. 1–14. https://doi.org/10.1145/3414080.3414109

[15] Karl Crary. 2009. Explicit contexts in LF (extended abstract). *Electron. Notes Theor. Comput. Sci.* 228 (2009), 53–68. https://doi.org/10.1016/j.entcs.2008.12.116

[16] Karl Crary. 2010. Higher-order representation of substructural logics. In *Proc. ICFP '10*. 131–142. https://doi.org/10.1145/1863543.1863565

[17] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (Oct. 2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

[18] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A fresh look at separation algebras and share accounting. In *Proc. APLAS '09 (Lect. Notes Comput. Sci., Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

[19] Amy P. Felty and Alberto Momigliano. 2012. Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. *J. Autom. Reason.* 48, 1 (2012), 43–105. https://doi.org/10.1007/S10817-010-9194-X

[20] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015. The next 700 challenge problems for reasoning with higher-order abstract syntax representations. *J. Autom. Reason.* 55, 4 (2015), 307–372. https://doi.org/10.1007/s10817-015-9327-3

[21] Amy P. Felty, Carlos Olarte, and Bruno Xavier. 2021. A focused linear logical framework and its application to metatheory of object logics. *Math. Struct. Comput. Sci.* 31, 3 (2021), 312–340. https://doi.org/10.1017/S0960129521000323

[22] Simon J. Gay. 2001. A framework for the formalisation of pi calculus type systems in Isabelle/HOL. In *Proc. TPHOLs '01 (Lect. Notes Comput. Sci., Vol. 2152)*, Richard J. Boulton and Paul B. Jackson (Eds.). Springer, 217–232. https://doi.org/10.1007/3-540-44755-5_16

[23] Aïna Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. 2017. LINCX: A linear logical framework with first-class contexts. In *Proc. ESOP '17 (Lect. Notes Comput. Sci., Vol. 10201)*, Hongseok Yang (Ed.). Springer, 530–555. https://doi.org/10.1007/978-3-662-54434-1_20

[24] Dan R. Ghica and Alex I. Smith. 2014. Bounded linear types in a resource semiring. In *Programming Languages and Systems (Lect. Notes Comput. Sci., Vol. 8410)*, Zhong Shao (Ed.). 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

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

[26] Jean-Yves Girard. 1987. Linear logic. *Theor. Comput. Sci.* 50, 1 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[27] Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2016. An extensible approach to session polymorphism. *Math. Struct. Comput. Sci.* 26, 3 (2016), 465–509. https://doi.org/10.1017/S0960129514000231

[28] Terrance Gray and Gopalan Nadathur. 2024. Binding contexts as partitionable multisets in Abella. In *Proc. LFMTP '24 (Electron. Proc. Theor. Comput. Sci., Vol. 404)*, Florian Rabe and Claudio Sacerdoti Coen (Eds.). Open Publishing Association, 19–34. https://doi.org/10.4204/EPTCS.404.2

[29] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *J. ACM* 40, 1 (January 1993), 143–184. https://doi.org/10.1145/138027.138060

[30] Joshua S. Hodas and Dale Miller. 1994. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.* 110, 2 (1994), 327–365. https://doi.org/10.1006/inco.1994.1036

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

[32] Jonas Braband Jensen and Lars Birkedal. 2012. Fictional separation logic. In *Proc. ESOP '12 (Lect. Notes Comput. Sci., Vol. 7211)*, Helmut Seidl (Ed.). Springer, 377–396. https://doi.org/10.1007/978-3-642-28869-2_19

[33] Olivier Laurent. 2017. Yalla. https://github.com/olaure01/yalla/.

[34] Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of standard ML. In *Proc. POPL '07*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 173–184. https://doi.org/10.1145/1190216.1190245

[35] Ian Mackie. 1994. Lilac: A functional programming language based on linear logic. *J. Funct. Program.* 4, 4 (1994), 395–433. https://doi.org/10.1017/S0956796800001131

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

[37] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-

319-30936-1_12

[38] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Logic* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

[39] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

[40] Brigitte Pientka and Jana Dunfield. 2008. Programming with proofs and explicit contexts. In *Proc. PPDP '08*. 163–173. https://doi.org/10.1145/1389449.1389469

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

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

[43] Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. 2018. Adjoint logic. (April 2018). https://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf (unpublished manuscript).

[44] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. 2004. Possible worlds and resources: The semantics of BI. *Theor. Comput. Sci.* 315, 1 (2004), 257–305. https://doi.org/10.1016/J.TCS.2003.11.020

[45] Jason Reed. 2006. Hybridizing a logical framework. In *Proc. HyLo@FLoC '06 (Electron. Notes Theor. Comput. Sci., Vol. 174)*, Patrick Blackburn, Thomas Bolander, Torben Braüner, Valeria de Paiva, and Jørgen Villadsen (Eds.). Elsevier, 135–148. https://doi.org/10.1016/J.ENTCS.2006.11.030

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

[47] Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. 2023. Mechanizing session-types using a structural view: Enforcing linearity without linearity. *Proc. ACM Program. Lang.* 7, OOPSLA (2023), 235:374–235:399. https://doi.org/10.1145/3622810

[48] Anders Schack-Nielsen. 2011. *Implementing Substructural Logical Frameworks*. Ph.D. Dissertation. Copenhagen, Denmark.

[49] Anders Schack-Nielsen and Carsten Schürmann. 2008. Celf - A logical framework for deductive and concurrent systems (system description). In *Proc. IJCAR '08 (Lect. Notes Comput. Sci., Vol. 5195)*, Alessandro Armando, Peter Baumgartner, and Gilles Dowek (Eds.). Springer, 320–326. https://doi.org/10.1007/978-3-540-71070-7_28

[50] Anders Schack-Nielsen and Carsten Schürmann. 2010. Curry-style explicit substitutions for the linear and affine lambda calculus. In *Proc. IJCAR '10 (Lect. Notes Comput. Sci., Vol. 6173)*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer, 1–14. https://doi.org/10.1007/978-3-642-14203-1_1

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

[52] Peter Thiemann. 2019. Intrinsically-typed mechanized semantics for session types. In *Proc. PPDP '19*. 1–15. https://doi.org/10.1145/3354166.3354184

[53] Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Inf. Comput.* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

[54] Philip Wadler. 2012. Propositions as sessions. In *Proc. ICFP '12*. 273–286. https://doi.org/10.1145/2364527.2364568

[55] David Walker. 2005. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 1, 3–43.

[56] James Wood and Robert Atkey. 2021. A linear algebra approach to linear metatheory. In *Proc. 2nd Joint Int. Workshop on Linearity & Trends in Linear Logic and Applications (Electron. Proc. Theor. Comput. Sci., Vol. 353)*, Ugo Dal Lago and Valeria de Paiva (Eds.). 195–212. https://doi.org/10.4204/eptcs.353.10

[57] James Wood and Robert Atkey. 2022. A framework for substructural type systems. In *Proc. ESOP '22 (Lect. Notes Comput. Sci., Vol. 13240)*, Ilya Sergey (Ed.). 376–402. https://doi.org/10.1007/978-3-030-99336-8_14

[58] Bruno Xavier, Carlos Olarte, Giselle Reis, and Vivek Nigam. 2017. Mechanizing focused linear logic in Coq. In *Proc. LSFA '17 (Electron. Notes Theor. Comput. Sci., Vol. 338)*, Sandra Alves and Renata Wasserman (Eds.). Elsevier, 219–236. https://doi.org/10.1016/J.ENTCS.2018.10.014

[59] Daniel Zackon, Chuta Sano, Alberto Momigliano, and Brigitte Pientka. 2024. *Split Decisions: Explicit Contexts for Substructural Languages (artifact)*. https://doi.org/10.5281/zenodo.14271731

[60] Uma Zalakain and Ornela Dardha. 2021. π with leftovers: A mechanisation in Agda. In *Proc. FORTE '21 (Lect. Notes Comput. Sci., Vol. 12719)*, Kirstin Peters and Tim A. C. Willemse (Eds.). 157–174. https://doi.org/10.1007/978-3-030-78089-0_9

Received 2024-09-17; accepted 2024-11-19