Security Reasoning via Substructural Dependency Tracking

HEMANT GOUNI, Carnegie Mellon University, USA
FRANK PFENNING, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA

Substructural type systems provide the ability to speak about *resources*. By enforcing usage restrictions on *inputs* to computations they allow programmers to reify limited system units—such as memory—in types. We demonstrate a new form of resource reasoning founded on constraining *outputs* and explore its utility for practical programming. In particular, we identify a number of disparate programming features explored largely in the security literature as various fragments of our unified framework. These encompass capabilities, quantitative information leakage, sandboxing in the style of the Linux `seccomp` interface, authorization protocols, and more. We furthermore explore its connection to conventional input-based resource reasoning, casting it as an internal treatment of the constructive Kripke semantics of substructural logics. We verify the *capability*, *quantity*, and *protocol* safety of our system through a single logical relations argument. In doing so, we take the first steps towards obtaining the ultimate multitool for security reasoning.

CCS Concepts: • **Security and privacy → Logic and verification**; *Information flow control*; • **Theory of computation → Modal and temporal logics**; **Linear logic**; Proof theory.

Additional Key Words and Phrases: capabilities, quantitative reasoning, protocol specifications, security types, substructural types, graded types, linear logic, ordered logic, adjoint logic, modalities, effects, coeffects

ACM Reference Format:
Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2026. Security Reasoning via Substructural Dependency Tracking. *Proc. ACM Program. Lang.* 10, POPL, Article 27 (January 2026), 29 pages. https://doi.org/10.1145/3776669

1 Introduction

Resource constraints are ubiquitous in real-world computing systems. For instance, all programs must manage memory. Programs which make use of concurrency need to contend with interthread sharing and synchronization. Correct management of these resources is essential to the correctness of the program. Substructural type systems have recently become a favored choice for managing these concerns due to their ability to reify limited resources in types.

Affine types as featured in Rust and Swift exemplify this character, prohibiting multiple usage.[1] After data at affine type is used, it drops out of scope and cannot be further used. From a memory management view, this precludes temporal safety issues like double-free and use-after-free. The treatment of units of allocated memory *ephemerally* within the language rather than *persistently* models their *limited* semantics, reflecting that of hardware memory. Various facets of substructurality model changes over time in different forms, emerging as a collection of disciplines for programmatic reasoning about resources.

[1]Practically, borrowing complicates the situation. See Marshall and Orchard [2024] for an overview.

Authors' Contact Information: Hemant Gouni, Carnegie Mellon University, Pittsburgh, USA, hsgouni@cs.cmu.edu; Frank Pfenning, Carnegie Mellon University, Pittsburgh, USA, fp@cs.cmu.edu; Jonathan Aldrich, Carnegie Mellon University, Pittsburgh, USA, jonathan.aldrich@cs.cmu.edu.

ACM 2475-1421/2026/1-ART27
https://doi.org/10.1145/3776669

It has been known from prior work on indexed modalities and coeffects [Gaboardi et al. 2016; Girard et al. 1992] that linearity is connected to variable usage, and therefore to computational *inputs*. Concurrent developments in substructural proof theory [Reed 2009; Reed and Pfenning 2010] have produced an *output-based* account of linearity as a semantic device for bridging logics, but have not explored its computational character. Here we commit fully to the output-based perspective as a programming language in its own right, uncovering an *effectful* interpretation of resources. The resulting type system is endowed with a unique collection of reasoning facilities including capabilities, quantitative information flow, privilege changing protocols, and more considered primarily—but not incidentally—by the literature on secure programming. We begin with a tutorial, starting with a conventional affine type system and inverting its mechanics to produce ours.

1.1 Input-Based Affinity: Looking Forwards

It will turn out that the directionality of resource restriction is the key distinction between input- and output-based resources. We'll start with the affine fragment of *bounded linear logic* [Girard et al. 1992]. Here the exponential modality from linear logic—represented by square brackets `[ ]` in what follows—is indexed with a maximum allowable usage count.[2] Consider the example in Figure 1.

```
1 val f : nat [ α α α ] -> nat
2 let f x = x + x
3 let f x = x + x * x
4 let f x = x + x * x + x ⚠
```

Fig. 1. Bounded Affine Logic: Natural Number Arithmetic

Each α in the type of `f` indicates one allowed usage of its first argument. In other words, every time the implementer of `f` desires to use `x`, they may expend one α token for that usage. So the implementations on lines 2 and 3 work perfectly well, using `x` two and three times respectively. But line 4 produces a type error, because no α tokens remain after using `x` thrice.

Observe the perspective from which usage is constrained here. The *inputs* to the computation—here, `x`—declare the structure of their future usage. This fundamental nature is present in all substructural type systems, with restrictions propagating forwards from inputs dictating how the computation may proceed. It seems there is a void left, then, that takes the shape of *backwards propagating* restrictions which operate based on the *result* of the computation.

1.2 Output-Based Affinity: Looking Backwards

Our primary design goal is for resource restrictions to propagate *backwards* from the output specified for a computation, rather than *forwards* from its inputs. Accordingly, we must record information about how the computation *proceeded* so it can be compared to what is expected at the output. We adapt the prior example to this paradigm, shown in Figure 2.

```
1 val f : [ α ] nat -> [ α α α ] nat
2 let f x = x + x
3 let f x = x + x * x
4 let f x = x + x * x + x ⚠
```

Fig. 2. Our System: The Same Thing?

[2]Our presentation of bounded linear logic uses slightly unconventional syntax for pedagogy; assume the usual natural number index is represented in unary form via αs.

Despite the syntactic similarity, we depart wholly from the machinery of bounded affine logic here. Exactly the same terms as in the prior example are well-typed under this one, so where lies the difference? Rather than `x` declaring that it may be used at most thrice as before, it now declares that it induces one α upon usage. Where before the output was unannotated, it now limits the computation to being dependent on at most three αs. That is, **the output type of the computation dictates the resources it could have used.** This may seem a surface-level distinction but is in reality significant, and is part of what provides us our unique tools for controlling resources.

The other portion comes from explicating the semantics of `[ α ] nat`. This type has little in common with the syntactically similar form in Figure 1. Whereas `nat [ α α α ]` is a constraint on variable use, `[ α ] nat` marks the effect of *running* `x`: it is a computation! It is easy to miss this, because we did not explicitly mark running (or forcing) `x`. We remedy this in Figure 3.

```
1 val f : [ α ] nat -> [ α α α ] nat
2 let f x = !x + !x
3 let f x = !x + !x * !x
4 let f x = !x + !x * !x + !x ⚠
```

Fig. 3. Our System: Forcing

`!x` *forces* the computation associated with `x`. Forcing an argument at type `[ α ] nat` induces an α resource and turns it into a `nat`. Figure 3 can be seen as the "desugared" version of Figure 2. Now that we know we are tracking computations rather than variable use, however, we can write a well-typed version of the implementation on line 4.

```
1 val f : [ α ] nat -> [ α α α ] nat
2 let f x = bind y to !x in y + y * y + y
```

On line 2 we run `x` *once* and bind the result to `y`. We then use `y` the way we used `x` in Figure 2. This time, however, type checking succeeds. Why? Consider how many αs were introduced in the body. Each forcing of `x` corresponds to one count of α, which is why line 4 in Figure 3 produced a type error. In the above, there is only one forcing of `x` and therefore only one α produced. So the resource bound $\alpha\ \alpha\ \alpha$ is not exceeded. This is because **the system tracks the running of computations, rather than the usage of variables.** In the affine case this allows it to capture *how often* some action may occur, e.g. for enforcing rate limits and establishing quantitative information leakage bounds. We will show that this choice induces the *type-theoretic structure* of an *effect*.

Why the focus on computations? It turns out that computations are special even to ordinary affine type systems. Consider the polarized lambda calculus [Levy 1999], which separates types that involve computation ("negative types") from types that characterize values ("positive types"). Perhaps surprisingly, affine typing does not strongly constrain positive types because they can be duplicated via pattern-matching followed by reconstruction. This admits contraction—the property that affine types are not supposed to provide! For instance, Figure 4 shows how to write a function which duplicates `nat`s under the input-based affine type system from Figure 1. Notice that dup is

```
val dup : nat [ α ] -> nat × nat
let dup n = match n with
            | Zero -> (Zero, Zero)
            | Succ n1 -> let (n2, n3) = dup n1 in (Succ n2, Succ n3)
```

Fig. 4. Bounded Affine Logic: Admissibility of Contraction

not only affine but linear in its argument. We could further write a linearly valid function which witnesses the admissibility of weakening, or deletion, for `nat`s. 'Violating' substructural restrictions in this way is possible for any type which can be pattern matched—precisely the *positive types*—but not for those which cannot, like functions and other negative types. This bias is reflected in our framework's tracking of computations. We will obtain stronger justification for our system's behavior as we discuss prior work and formal properties which are consequences of this design.

Output-based substructural types offer a distinct suite of programming tools complementing those arising from input restrictions. We'll touch on the systematic differences which permit certain invariants (e.g. aliasing restrictions) to be expresed naturally on one side or the other, largely directing our attention to the strengths of our regime. For a hint at where we're headed, signatures like `[ low_ops* authorize high_ops* ]` will be able to constrain computation in an *ordered* way while *mixing in* affine, strict, and structural constraints. We will take a tour of these and the rest of the zoo of reasoning tools—largely as applicable to secure programming—which spring out of the simple constructions of our setting.

1.3 A Preview of the Rest

The basic mechanisms we have introduced so far constitute the bulk of what we will use in this work. The largest remaining piece is *mode variability*, which enables us to interoperate between the linear, affine, ordered, strict, and structural modes of resource reasoning. We discuss this extension in Section 2 and use it to justify our focus on security reasoning. We then formalize our intuitions so far, touching on the core typing rules in Section 3 and remarking on the essential role played by polarity. Section 4 presents an array of examples, including modeling sandboxing in the style of Linux's `seccomp` system call. We ground each example in the metatheoretic properties we have shown for various facets of our system, including *capability safety*, *quantity safety*, and *protocol safety*. These are explained in detail in Section 5. Section 6 reviews related work, discussing connections to constructive Kripke semantics and graded modalities. We conclude in Section 7.

(1) **We present an output-based interpretation of resources which displays effectful structure.** Furthermore, **substructural reasoning is *unified* in our system.** Incorporating recent advances from the substructural typing literature, programs written under our framework may liberally swap between structural, affine, linear, strict, ordered, and more. Being able to exploit ordering is uncommon and falls outside the domain of coeffectful accounts of substructurality.
(2) We show a collection of examples enabled by the output-based setting. We survey **capabilities, quantitative information leakage, sandboxing and the Linux `seccomp` interface, authorization protocols, lightweight typestate,** and more.
(3) We show soundness by way of a logical relations arguments. It establishes **capability, quantity, and protocol safety** simultaneously. This serves to formally unify the techniques we surveyed by way of uniting their soundness.
(4) We contextualize our work within the rich body of work on substructural type systems and point out how **issues and insights from other settings are mirrored and offered clarity by ours.** In particular, we discuss connections to type systems for information flow, constructive Kripke semantics, graded modal type theory, and Adjoint Logic. We justify here the use of concepts from substructural reasoning to discuss our system, showing how to reconstruct our language from prior work.

By the end of this paper we will have laid the groundwork for resource-aware machinery which regulates outputs rather than inputs. We will have shown its utility in service of practical programming concerns, and detailed how these arise from its unique foundations.

2 Further Exposition and Background

We review the remaining structural rules supported by our framework. Importantly, we will be able to move between them as programming concerns demand; just as it is rare that one wishes to work in a *purely* linear or ordered type system in the typical substructural setting, so it is in ours. Using these notions, we discuss the foundations of our work in information flow.

2.1 The Remaining Structural Rules and Their Modes

Fleshing out our setup, we step through each supported structural rule in turn and briefly detail its mechanics, deferring a full discussion to Section 3. Afterwards we describe our approach to composing them via *mode switching*, giving each rule a corresponding *mode*.

2.1.1 Contraction / Affinity. In Section 1 we considered collections of *affine*, or quantity-sensitive, resources of the form $[\ \alpha\ \alpha\ \alpha\ ]$. This is because running a computation inducing an α once, for instance, is not considered the same as running it twice (thereby inducing two αs). So we have $[\ \alpha\ ] \neq [\ \alpha\ \alpha\ ]$. In general, affinity will be useful for constraining *how much* computation may take place, and corresponds to relaxing the typical structural rule of contraction. We will see in Section 3 that a non-standard formulation of contraction will be needed due to our ability to mix modes, in particular order-sensitivity. This is drawn from prior work [Kanovich et al. 2018, 2019] investigating an ordered sequent calculus.

```
val ord : [ o₁ ] nat -> [ o₂ ] nat -> [ o₂ o₁ ] nat
let ord x y = !y / !x
let ord x y = !x / !y ⚠
```

Fig. 5. Ordering

2.1.2 Mobility / Ordering. As noted, we may also consider collections of *order-sensitive* resources. We have been operating under the assumption that $[\ \rho_1\ \rho_2\ ] = [\ \rho_2\ \rho_1\ ]$, but it is reasonable to choose to relax this. This causes the *order* of computations to become significant, as Figure 5 shows. The order in which the computation o_1 in x and o_2 in y are run, assuming / is evaluated left-to-right, is reflected in the return type of ord. We will show in Section 4 that discriminating by order leads to lightweight protocol reasoning in the output-based setting. Ordering typically corresponds to relaxing the structural rule of exchange, but as with contraction we will need to deploy a modified variant in our setting. This will be called *mobility*, nomenclature from Roshal and Pfenning [2025]. The ability to (selectively) drop ordering is rare in prior work; Section 3 will discuss the choices taken to make it available here.

```
val strict : [ σ ] nat -> [ σ ] nat
let strict x = !x
let strict x = 5 ⚠
```

Fig. 6. Strictness

2.1.3 Weakening / Strictness[3]. The final reasoning tool is strictness, which *forces* certain resources to be induced. This is connected to the structural rule of weakening, which states that if we have some expression using resources $[\ \sigma_1\ \sigma_3\ ]$, then it can also be said to use resources $[\ \sigma_1\ \sigma_2\ \sigma_3\ ]$.

[3]Also called *relevance*, but we avoid this term because a logic lacking the rule of weakening does not correspond to the Relevance Logic of Anderson et al. [1992]. *Strict* is preferred instead for its meaning in *strictness analysis*.

That is, the former can be transformed into the latter via weakening. Importantly, we may insert σ_2 anywhere we like. We used weakening implicitly in Figure 3, where an extra α was weakened on line 2. Figure 6 shows that without this ability, typable programs must force x at least once.

The ordinary rule of weakening in structural type systems applies to in-scope variables x, allowing them to be added to the set of bound variables while preserving typing. We weaken here not on term variables, but on *resource variables*. The latter fundamentally diverges from the former for its programming consequences. It will turn out that *capabilities* [Linden 1976] emerge from the absence of weakening, a point we will demonstrate in Section 4 and formally justify in Section 5.

```
val mix : [ α ] nat -> [ σ ] nat -> [ α α σ σ ] nat
let mix x y = !y + !x + !y
let mix x y = !x + !x + !x + !y + !y ⚠      (* α over-produced *)
let mix x y = !x + !y ⚠                     (* σ under-produced *)
```

Fig. 7. Mixing

2.1.4 Mixing Modes. We may choose which structural rules we wish to work under. Each resource ρ carries with it the set of structural rules it respects as its *mode*. For instance, assume ρ respects at least contraction and o respects no structural rules. Then we can turn $[\ o\ \rho\ \rho\ o\ ]$ into $[\ o\ \rho\ o\ ]$, contracting ρ. But neither can be transformed to $[\ \rho\ o\ \rho\ ]$ because o is not mobile or contractible. Figure 7 uses an affine α and a strict σ in the same program.

Taking inspiration from prior work on LNL [Benton and Wadler 1996] and Adjoint Logic [Pruiksma et al. 2018], the latter of which provides the basic structure for programs to mediate between input-based linearity, affinity, and strictness, **we additionally allow for moving between substructural modes in a sound way.** In short, it is perfectly acceptable to track *more* information than the program needs (say, regarding quantity) and simply discard that information later. This is *upshifting* in the adjoint setting, and is connected to universal quantification in ours. *Downshifting* is trickier and occurs through existential quantification. We will compare our mechanisms to those in Adjoint Logic in Section 3.

2.2 Substructural Information Flow

Our framework is an extension of Gouni et al. [2025a]'s *structural information flow*. Rather than the resource reasoning light in which we have cast it so far, **another valid view of this work is as a generalization of the standard machinery for information flow.** Information flow reasoning is typically conducted within a semilattice of security annotations. Such a structure is defined by a partial order $\sqsubseteq$ and a join $\sqcup$ (or meet $\sqcap$) operator over security annotations satisfying, crucially, two equations: $x \sqcup x = x$, or idempotency, and $x \sqcup y = y \sqcup x$, or commutativity. Within the structural information flow framework, sets of resource variables $[\ \delta_1\ \delta_1\ \delta_2\ ]$ are interpreted as mathematical sets. So defining $\sqcup$ as set union $\cup$ of these sets, the two laws above are immediately satisfied. $\sqsubseteq$ is then defined as subset inclusion $\subseteq$. From an information flow tracking perspective, the presence of δ_1 in a $[\ \delta_1\ \delta_2\ ]$ annotation proclaims the dependency of the computation in question on the input δ_1. For a concrete example, look to Figure 8. The annotation [pwd ...] states that the annotated expression is affected by password data, appearing in line 4 after calling check. There is an error on line 5 because it depends on a **bool** of password data but only declares alice.

Idempotency and commutativity of $\sqcup$ correspond respectively to contraction and exchange (or mobility). Weakening is given by subsumption, or the preservation of typing, over the lattice preorder $\sqsubseteq$. If all that is desired is to check *whether* some computation depends on some data, then having all three structural properties is quite convenient. However, in this work we consider

```
let alc : [ alice ] string = ...
let pass : [ pwd ] string = ...
let check : [ δ ] string -> [ δ pwd ] bool = fun att -> !att == !pass
let _ : [ pwd alice ] bool = check alc
let _ : [ alice ] bool = check alc ⚠
```

Fig. 8. Information Flow

how that dependency occurs, accounting for quantity, order, and strictness. Relaxing the algebraic properties of lattice join and the accompanying preorder does not fundamentally change non-interference, the core information flow property which ensures that dependency tracking is faithful. Non-interference is a *hyperproperty*, or determined by multiple related executions [McLean 1996]. Working substructurally instead adds extra *trace* properties—those of singular program executions—on top. These are capability, quantity, and protocol safety, to be shown in Section 5. **Our system owes its affinity for security reasoning to its roots in information flow, particularly the additional security-relevant properties gained from generalizing it.** This basis in information flow is what we refer to as *dependency tracking*, avoiding use of the term *information flow* because most of the properties of interest here are not characterized by non-interference. We have extended Gouni et al. [2025a]'s non-interference result to our more general setting, but this does not serve as a focal point of the current work.

3 Quantified Resource Type Theory

We first discuss the precise mechanics of the structural rules summarized in Section 2.1. We then look to the type system itself—abbreviated Qrtt—and note the strong influence of polarity on it, using it to guide our exposition.

3.1 Structural Rules

The syntax for resources is shown in Figure 9. We start with resource contexts ϕ, which are simply a collection of resource variables; we previously wrote these as [ρ_1 ρ_2 ρ_3]. They form a monoid with a unital element $\circ$ and an associative multiplication operation given by juxtaposition. Each resource variable is annotated with a mode m, which we have left unwritten thus far and will omit wherever the mode has already been specified or isn't requisite to understanding. Modes specify collections of active structural rules of which we have *five*; using ordered resources requires a slight generalization from the three we covered in the last section [Kanovich et al. 2019]. We have weakening, backwards + forwards contraction, and backwards + forwards mobility, splitting contraction and mobility in two. Section 4 will show the split structural rules' separate variability [Roshal and Pfenning 2025] to be useful.

The structural rules are also presented in Figure 9. Rather than applying directly to the typing relation, as is typical, each rule induces an inequality of ϕs to be used later by typing. Starting with weakening, the presence of w in a resource ρ's mode allows Weaken to be used to obtain it. Assuming w $\in$ ρ_1, or that ρ_1 is weakenable, it will be possible to start from an expression e : [ρ_2] **int** and obtain e : [ρ_2 ρ_1] **int**. The $\Delta \vdash \rho^m$ in the premise of the rule is for scoping; we will return to this in Section 3.2.

Contraction in Qrtt is regulated by cb and cf, and we show rules corresponding to each of these tokens. The primary difference between Contract-Back and Contract-Fwd is in choosing *which side to contract to.* Reading the first bottom-up, contraction collapses onto the left resource variable, corresponding to going from e : [ρ_1 ρ_2 ρ_1] **int** where cb $\in$ ρ_1 to e : [ρ_1 ρ_2] **int**. Within the second it collapses onto the right, which assuming cf $\in$ ρ_1 instead would allow us

Substructural Modes m Resource Contexts ϕ Resource Variables $\rho_1, \rho_2, \ldots$

$$\text{Substructural Modes } m \subseteq \{\,\textsc{w}, \textsc{cb}, \textsc{cf}, \textsc{mb}, \textsc{mf}\,\}$$
$$\text{Resource Contexts } \phi ::= \circ \mid \rho^m \mid \phi_1\ \phi_2$$

$$\textsc{Weaken}\ \frac{\phi_1\ \rho^m\ \phi_1' \sqsubseteq_\Delta \phi_2 \qquad \Delta \vdash \rho^m \qquad \textsc{w} \in m}{\phi_1\ \phi_1' \sqsubseteq_\Delta \phi_2} \qquad \textsc{Contract-Back}\ \frac{\phi_1\ \rho^m\ \phi_1'\ \phi_1'' \sqsubseteq_\Delta \phi_2 \qquad \textsc{cb} \in m}{\phi_1\ \rho^m\ \phi_1'\ \rho^m\ \phi_1'' \sqsubseteq_\Delta \phi_2}$$

$$\textsc{Contract-Fwd}\ \frac{\phi_1\ \phi_1'\ \rho^m\ \phi_1'' \sqsubseteq_\Delta \phi_2 \qquad \textsc{cf} \in m}{\phi_1\ \rho^m\ \phi_1'\ \rho^m\ \phi_1'' \sqsubseteq_\Delta \phi_2} \qquad \textsc{Move-Back}\ \frac{\phi_1\ \rho^m\ \phi_1'\ \phi_1'' \sqsubseteq_\Delta \phi_2 \qquad \textsc{mb} \in m}{\phi_1\ \phi_1'\ \rho^m\ \phi_1'' \sqsubseteq_\Delta \phi_2}$$

Fig. 9. Structural Rules (Selected)

to obtain e : ⟦ $\rho_2\ \rho_1$ ⟧ **int**. In other words, **our formulation of contraction is *directed* and *non-local*** and therefore non-standard [Kanovich et al. 2019]. To illustrate the difference consider the typical rule of Contraction below, which requires contracted antecedents to be *adjacent*.

$$\textsc{Contraction}\ \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \qquad \textsc{Exchange}\ \frac{\Gamma, B, A, \Gamma' \vdash C}{\Gamma, A, B, \Gamma' \vdash C}$$

We cannot abide by this restriction, however, due to an issue which occurs when context entries are ordered, that is, lacking mobility. Briefly, in the absence of commutativity ⟦ $\rho\ \rho$ ⟧ might be locally contracted to ⟦ ρ ⟧, but instantiating ρ = ⟦ a b ⟧ (to be covered in Section 3.2.3) and attempting to locally contract ⟦ a b a b ⟧ fails. The issue is fully described in Appendix B of Gouni et al. [2025b] and analogous to the failure of local soundness seen in Kanovich et al. [2019, §8]. We use the same remedy as there. **Exchange is likewise directed and non-local and therefore called *mobility*** in our setting. It is regulated by the mb and mf tokens. Move-Back is activated by mb and moves the resource variable left by an arbitrary distance; this corresponds to going from e : ⟦ $\rho_2\ \rho_3\ \rho_1$ ⟧ **int** where mb $\in \rho_1$ to e : ⟦ $\rho_1\ \rho_2\ \rho_3$ ⟧ **int**. Contrast this with Exchange above, which again requires adjacency. We look now to the type system constructed around these rules.

3.2 Core Type System

Selected core syntax and typing rules for Qrtt are laid out in Figure 10. Our formalization is based on Gouni et al. [2025a] and follows a call-by-value semantics. The form of the typing judgement is $\Delta; \Gamma \vdash e : A \mid \phi$ and can be read as "Under resource variables Δ and term variables Γ the expression e has type A using resources ϕ." Starting with the simplest rules, T-Unit incurs no resource cost to introduce a `unit`. The rule for variables is similarly uninteresting, which is as expected for our flavor of substructural reasoning where computational inputs are untracked.

3.2.1 *Negative Connectives.*

The rules for negatives, including resource types $[A \cdot \phi]$ and function types $A \longrightarrow B$, are more interesting. Reading from premises to conclusion, T-Consume packs the judgemental resources ϕ of an expression e into its type A to introduce a type $[A \cdot \phi]$ with resources encapsulated inside it. We previously wrote [**int** $\cdot\ \rho_1\ \rho_1$] as ⟦ $\rho_1\ \rho_1$ ⟧ **int**, where $\phi = \rho_1\ \rho_1$ and A = **int**. T-Produce works in the opposite direction, starting from an expression e at $[A \cdot \phi_1]$, ejecting the resources ϕ_1 from the type back into the typing judgement, and concatenating them to the ϕ_2 resources already there. The distinction between a ϕ in the typing judgement versus one

Type A, B Expr e Term Vars $x_1, x_2, \ldots \in \Gamma$ Resource Vars $\rho_1, \rho_2, \ldots \in \Delta$

$$\text{Types } A, B ::= \texttt{unit} \mid [A \cdot \phi] \mid A \to B$$
$$\text{Expressions } e ::= \langle\rangle \mid x \mid \#e \mid \,!\,e \mid \lambda(x.e) \mid \texttt{ap}(e_1; e_2)$$

$$\textsc{T-Unit}\ \frac{}{\Delta; \Gamma \vdash \langle\rangle : \texttt{unit} \mid \circ} \qquad \textsc{T-Var}\ \frac{}{\Delta; \Gamma, x : A \vdash x : A \mid \circ} \qquad \textsc{T-Consume}\ \frac{\Delta; \Gamma \vdash e : A \mid \phi}{\Delta; \Gamma \vdash \#e : [A \cdot \phi] \mid \circ} \qquad \textsc{T-Produce}\ \frac{\Delta; \Gamma \vdash e : [A \cdot \phi_1] \mid \phi_2}{\Delta; \Gamma \vdash \,!\,e : A \mid \phi_1\ \phi_2}$$

$$\textsc{T-Lam}\ \frac{\Delta; \Gamma, x : A_1 \vdash e : A_2 \mid \circ \qquad \Delta \vdash A_1}{\Delta; \Gamma \vdash \lambda(x.e) : A_1 \to A_2 \mid \circ} \qquad \textsc{T-Ap}\ \frac{\Delta; \Gamma \vdash e : A_1 \to A_2 \mid \phi \qquad \Delta; \Gamma \vdash e_1 : A_1 \mid \phi_1}{\Delta; \Gamma \vdash \texttt{ap}(e; e_1) : A_2 \mid \phi_1\ \phi}$$

Fig. 10. Typing Rules for Variables and Negative Connectives

inside a type is that the first is *being used at present*, whereas the second is waiting for an operation which will *cause it to start being used.* The order of resources in the typing judgement is listed in reverse order of their execution, so T-Produce concatenates the later-used ϕ_1 on to the front of ϕ_2.

Consumption and production correspond respectively to suspending a computation and forcing it. This computational interpretation arises from the syntax: operationally, $\#e$ pauses the computation and $!\,e$ forces it. Crucially, observe from the conclusion of T-Consume that when the computation is suspended, it incurs no resource usage. **In general, introduction forms for negative types will have no judgemental resources because their contents are suspended and not executing.** As Section 5 will show, our safety properties require this. The treatment of effects in an explicitly polarized setting [Levy 1999] is analogous; in particular, we can decompose $[A \cdot \phi]$ as $U(\bigcirc_\phi(F(A)))$, where $\bigcirc_\phi$ is a computation connective tracking ϕ. This indicates that $[A \cdot \phi]$ is a *strong monad* of the variety deployed by Moggi [1989], or alternately a ϕ-indexed *lax* modality [Fairtlough and Mendler 1997]. Resources are tracked using the *structure of effects.*

The resource type is of particular importance because it provides the all-important case for the *subsumption rule*, which connects the structural rules in Figure 9 to the rest of the type system. The relevant rule and case of subsumption are reproduced here. Reading T-Sub from top-to-bottom,

$$\textsc{T-Sub}\ \frac{\Delta; \Gamma \vdash e : A_1 \mid \phi \qquad A_1 \sqsubseteq_\Delta A_2}{\Delta; \Gamma \vdash e : A_2 \mid \phi} \qquad \textsc{S-Resource}\ \frac{A_1 \sqsubseteq_\Delta A_2 \qquad \phi_1 \sqsubseteq_\Delta \phi_2}{[A_1 \cdot \phi_1] \sqsubseteq_\Delta [A_2 \cdot \phi_2]}$$

this rule states that if we have an expression e at some type A_1, then we can obtain that same expression at A_2 so long as the former is a subtype of the latter. S-Resource is the case of this subtyping relation which states that $[A_1 \cdot \phi_1]$ is a subtype of $[A_2 \cdot \phi_2]$ if A_1 and A_2 are subtypes and the resource contexts ϕ_1 and ϕ_2 are compatible. Specifically, **if ϕ_1 can be *rewritten* to ϕ_2 via weakening, contraction, and mobility then they are compatible under subtyping.** Besides this, the subtyping relation is entirely standard and uninteresting; all other types are covariant with respect to it besides the antecents of function types, which are contravariant.

Finally, the rules for $A_1 \to A_2$ are nearly standard. It is a negative type, so T-Lam concludes with no used resources indicated in the typing judgement. This is achieved by forcing its body in the

$$\begin{aligned} \text{Types } A, B &::= \ldots \mid A + B \mid A \otimes B \\ \text{Expressions } e &::= \ldots \mid \mathtt{l} \cdot e \mid \mathtt{r} \cdot e \mid \mathtt{case}\ e\ \{\ \mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\ \} \\ &\quad \mid \langle e_1, e_2 \rangle \mid \mathtt{split}\ e_1\ \mathtt{into}\ \langle x_1, x_2 \rangle\ \mathtt{in}\ e_2 \end{aligned}$$

$$\frac{\Delta; \Gamma \vdash e : A_1 \mid \phi \qquad \Delta \vdash A_2}{\Delta; \Gamma \vdash \mathtt{l} \cdot e : A_1 + A_2 \mid \phi}\ \textsc{T-InjL} \qquad \frac{\Delta; \Gamma \vdash e_1 : A_1 \mid \phi_1 \qquad \Delta; \Gamma \vdash e_2 : A_2 \mid \phi_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : A_1 \otimes A_2 \mid \phi_2\ \phi_1}\ \textsc{T-Pair}$$

$$\frac{\Delta; \Gamma \vdash e : A_2 \mid \phi \qquad \Delta \vdash A_1}{\Delta; \Gamma \vdash \mathtt{r} \cdot e : A_1 + A_2 \mid \phi}\ \textsc{T-InjR} \qquad \frac{\Delta; \Gamma \vdash e : A_1 \otimes A_2 \mid \phi \qquad \Delta; \Gamma, x_1 : A_1, x_2 : A_2 \vdash e' : A \mid \phi'}{\Delta; \Gamma \vdash \mathtt{split}\ e\ \mathtt{into}\ \langle x_1, x_2 \rangle\ \mathtt{in}\ e' : A \mid \phi'\ \phi}\ \textsc{T-Split}$$

$$\frac{\Delta; \Gamma \vdash e : A_1 + A_2 \mid \phi \qquad \Delta; \Gamma, x_1 : A_1 \vdash e_1 : A \mid \phi' \qquad \Delta; \Gamma, x_2 : A_2 \vdash e_2 : A \mid \phi'}{\Delta; \Gamma \vdash \mathtt{case}\ e\ \{\ \mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\ \} : A \mid \phi'\ \phi}\ \textsc{T-Case}$$

Fig. 11. Typing Rules for Positive Connectives

premise to consume its resources, if needed, before forming a lambda. Note that Figure 8 didn't use the $\#e$ syntax associated with T-Consume while introducing a function on line 3. The language in the examples deploys a syntactic convenience that automatically applies consumption to function bodies. T-Ap collates the resource usages from the function and argument expressions together, ensuring that they are joined in the (reverse) order of evaluation.

The premise $\Delta \vdash A_1$ in T-Lam is for scoping, to ensure that the argument type does not introduce any resource variables not contained in Δ. This is the same reason Weaken contains the premise $\Delta \vdash \rho^m$. Were this premise absent, it would be trivial to use T-Sub to weaken ill-scoped resource variables into the type A_2 in its conclusion. Qrtt maintains a property called *regularity* to ensure that all types A are well-scoped, in addition to resource contexts ϕ and expressions e. Expressions must be scoped not only under Δ but also under Γ, because they can contain term variables.

3.2.2 Positive Connectives. We look now to the positive connectives in Figure 11, covering sums and positive products. Looking to the introduction rules T-InjL, T-InjR, and T-Pair, there is an immediate difference from those in Figure 10. Resources may leak freely through positive types! This is because **positive types do not interact with resources due to being computationally inert.** This too is in line with the reading of our resources as effectful, since introduction forms for positive types do not interact with effects (or rather, with potentially effectful computations) [Levy 1999]. Additionally, echoing the reasoning from Figure 4, ordinary substructurality does not interact strongly with positive types so our setting shouldn't either. **We phrase this bifurcation along polarity as a design decision, but it is *forced* by the core safety properties discussed in Section 5.** The attention paid to computational behavior makes Qrtt sensitive to polarity. Section 6.1 will show that prior work is not.

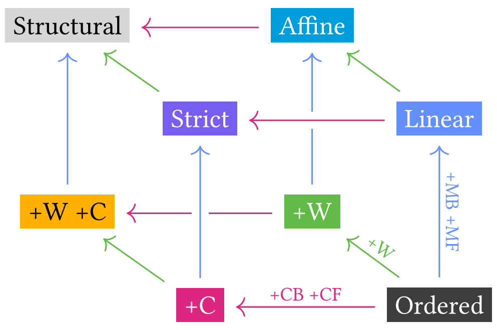
T-InjL and T-InjR once again include scoping premises for ensuring regularity. Each checks that the other element of the sum type—that not witnessed by the expression in the premise—is well-scoped. The choices of concatenated resource contexts in the conclusions of T-Pair, T-Split, and T-Case are given by their order of evaluation. Besides these points the rules for positives are entirely standard, as might be expected from our prior comments.

$$\text{Types } A, B ::= \ \ldots \mid \forall(\rho^m.A) \mid \exists(\rho^m.A)$$
$$\text{Expressions } e ::= \ \ldots \mid \Lambda(\rho^m.e) \mid e[\phi] \mid \mathsf{pack}[\phi](e) \mid \mathsf{open}(e_1; \rho^m, x.e_2)$$

T-All
$$\frac{\Delta, \rho^m; \Gamma \vdash e : A \mid \circ}{\Delta; \Gamma \vdash \Lambda(\rho^m.e) : \forall(\rho^m.A) \mid \circ}$$

T-Inst
$$\frac{\Delta; \Gamma \vdash e : \forall(\rho^m.A) \mid \phi' \qquad \Delta \vdash \phi \qquad \phi \geq m}{\Delta; \Gamma \vdash e[\phi] : [\phi/\rho^m]A \mid \phi'}$$

T-Pack
$$\frac{\Delta; \Gamma \vdash e : [\phi/\rho^m]A \mid \phi' \qquad \Delta \vdash \phi \qquad \phi \geq m}{\Delta; \Gamma \vdash \mathsf{pack}[\phi](e) : \exists(\rho^m.A) \mid \phi'}$$



T-Open
$$\frac{\Delta; \Gamma \vdash e_1 : \exists(\rho^m.A_1) \mid \phi \qquad \Delta, \rho^m; \Gamma, x : A_1 \vdash e_2 : A_2 \mid \phi' \qquad \Delta \vdash A_2 \qquad \Delta \vdash \phi'}{\Delta; \Gamma \vdash \mathsf{open}(e_1; \rho^m, x.e_2) : A_2 \mid \phi' \ \phi}$$

Fig. 12. Typing Rules and Mode Lattice for Quantifiers

3.2.3 Quantification. We left out one negative and one positive connective from Figure 10 and Figure 11 respectively. This is remedied in Figure 12. The universal quantifier $\forall(\rho^m.A)$ over resource variables ρ^m is negative; existential quantification $\exists(\rho^m.A)$ is positive. Being negative, introduction of the former via T-All concludes at an empty resource context. The term $\Lambda(\rho^m.e)$ acts analogously to a lambda but for resource variables rather than term variables, introducing the bound ρ^m into Δ so that e may use it. One way of using it is through T-Inst, which substitutes a well-scoped ϕ for ρ^m in $\forall(\rho^m.A)$. The second premise establishing well-scopedness is required to ensure regularity and restricts ϕ to only mentioning resource variables in Δ.

The third premise of T-Inst subjects ϕ to a further restriction: $\phi \geq m$ requires that all resource variables in ϕ have a mode *greater than or equal to m*, or the mode of the quantified variable. Recalling that m is a set of rule tokens r, this means we must have $m' \supseteq m$ for all modes m' in ϕ. Consequently, **resource variables can only be instantiated to equally or less restrictive modes.** It is permissible to track *more* information than needed, such as quantity and ordering, and drop it later by introducing more substructural equivalences. In terms of Figure 12 it is acceptable to move along the arrows, e.g. from linear to structural. Of course, we are able to *separately* vary the left and right variants of contraction and mobility so there are 5 total dimensions rather than 3 [Roshal and Pfenning 2025]. Section 4.4 will show the extra modes induced to be practically useful. Note that we have been implicitly using universals to introduce polymorphic resources in function types, for instance in Figure 8.

T-Pack offers another way for expressions to use resource variables bound in Δ. pack introduces *existential* quantification, rather than universal. Being positive, it is transparent to any resources ϕ' propagating from its body e. Reading from premises to conclusion, this rule states that if e can be type checked under a ϕ in its type, then it is acceptable to offer it at nearly the same type, but with the ϕ abstracted away by some variable ρ^m. The third premise subjects this to exactly the same restrictions as for using universal quantifiers, but imposed oppositely. Where for instantiation we were *substituting away* a resource variable ρ^m for a context ϕ, here we *factor out* a resource context ϕ and bind a variable ρ^m in its place. This means **existentially bound resource variables ρ^m are equally or more restrictive than their instantiation ϕ.** This is useful for e.g. treating a wholly structural ϕ linearly within some local scope. In terms of Figure 12 this corresponds to following the arrows backwards to find permissible transitions between modes. T-Open extracts from e_1 the existentially quantified resource ρ^m and the type A within which it appears and binds them to be used in e_2. This is a largely standard rule for existential quantification, and the $\phi' \ \phi$ in its conclusion is given by the evaluation order of e_1 followed by e_2.

4 Examples

Our system has three primary facets, corresponding to each class of structural rule: weakening, contraction, and mobility. We'll step through each in turn, highlighting the properties conferred. Varying mobility will turn out to lead to the most interesting and powerful patterns due to its separate forward and backward flavors. Assume as a convenience that **`let`** definitions automatically consume *ambient* resources—those not captured in types, represented by ϕ in the typing judgement—in their bodies, if there are any. So we never have to wrap `#(...)` around the body of a **`let`**.

4.1 Capabilities | `-W`

We start with *capabilities*, an elegant technique from the security literature for managing permissions. Linden [1976] provides the following definition of a capability, emphasis ours:

> A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. **Modification of a capability [...] is not allowable**; however, unlike the case for tickets, reproduction of a capability is legal.

Restated, a capability is a token of authority to access system resources which is (1) *communicable* and (2) *unforgeable* [Gusmeroli et al. 2013]. Unlike access control lists, which require always checking permissions against a central authority, capabilities act as sufficient authorization merely by existence. For instance, a Unix file descriptor is a capability: it represents access to some resource which, once granted by the operating system, can be freely passed to other processes without further authorization between the grantee and the operating system.[4]

Figure 13 sets up an example of this pattern in our system. We begin by introducing a resource `c` which is **`strict`**, or lacks weakening. We introduce `c` via an existential, written here using ML-style module syntax. **`sig`** ... **`end`** denotes the type $\exists$(`c.{ authenticate : string -> ... }`). The record syntax `{ }` is shorthand for a $\otimes$ of definitions, here just a single one. **`struct`** ... **`end`** gives its implementation `pack[[ ]](...)`. The existential is initialized on line 5 with an empty set of resources for convenience; the requirement $\phi \geq$ **`strict`** is vacuously satisfied when ϕ is empty.

The single definition `authenticate` exported by the existential provides the final piece of machinery. It takes as argument a **`string`** containing a passphrase and returns either a bare **`unit`**, if authorization was unsuccessful, or one encapsulated inside a resource type containing `c` otherwise. **The resource `c` is a capability.** Why? It is *communicable*, because resources may be freely passed to and returned from functions at the type level. *Unforgeability* arises from strictness: successfully calling `authenticate` is the only way to obtain `c`. Possession of it attests *a priori* that authorization succeeded, without deference to any other authority.

[4]While the *permission to access* can be transmitted, the file descriptor itself—or its number—will likely change between processes. See `SCM_RIGHTS` in `unix(7)`.

```
 1  module Auth : sig
 2    strict resource c
 3    val authenticate : string -> unit + [ c ] unit
 4  end = struct
 5    resource c = [ ]
 6    ...                                (* implementation here *)
 7  end
```

```
 8  open Auth importing c, authenticate in
 9    val secured : forall ρ . [ c ρ ] int -> ...
10
11    case authenticate "argaven" with
12    | Left _  -> ...                   (* failure case *)
13    | Right tok -> secured #(!tok; 1)  (* c affirms authentication *)
14    end
```

Fig. 13. Capabilities

The minimal working example on lines 8-14 leverages this. On line 8 we open the existential within which c and authenticate are declared. Note that **we can open once for each principal we're interested in, giving each a fresh capability.** We attempt an authentication on line 11 and case on the result. If we were unsuccessful we enter the Left branch where nothing is returned besides a unit, so we handle the error. The Right branch represents success, giving us a tok : [c] unit. Here we force tok to release its capability into the ambient resources—the ϕ in the typing judgement—and sequence with a 1, then suspend again and pass everything to secured. This satisfies the argument type of secured, which requires the resource c to be present. Though it is not needed in this example, to account for cases where the argument of secured may witness more resources than just c, the argument type additionally includes a polymorphic resource ρ. We suspect the syntactic overhead of forcing, sequencing, and suspension on display here can be solved with implicit arguments or simply higher-level syntax which desugars to what is written; this is left as future work. Our examples choose to adhere closely to QRTT as formalized.

The form of capability reasoning shown here is sound. A well-typed expression may only present c in its type if its computation witnesses c. Corollary 4.1 states exactly this. We require here an *instrumented operational semantics* which keeps track of resources as a matter of evaluation; this will be covered fully in Section 5. This section uses specialized notation $e \longmapsto^* v \triangleright \phi$ in stating safety properties. Importantly, keep in mind that this is not the evaluation judgement. It can be read "Closing e causes it to evaluate to a value v while witnessing resources ϕ." The *closing* condition is important, because the corollary is stated for an e typed under a non-empty context Γ, meaning it may contain free variables and cannot be directly evaluated. Closing it means substituting away its free variables for closed expressions of appropriate types. Section 5 will explain precisely how.

Corollary 4.1 (Capability Safety). *If $\Delta; \Gamma \vdash e : A \mid \phi_1$ where $\rho^m \in \phi_1$ and $w \notin m$ then $e \longmapsto^* v \triangleright \phi_2$ where $\rho^m \in \phi_2$.*

Instantiated to our scenario Corollary 4.1 can be read, recalling that forcing statically releases resources into the typing judgement, as "If e is well-typed at [c] A and c is not weakenable then e can be forced to generate an evaluation trace which contains c." This theorem follows the guidance given in Snyder [1981, p. 4] for formal capability soundness, though does not translate exactly because our functional setting departs from theirs.

```
 1  val pass : [ pwd ] string
 2
 3  val mk_checker : [ pwd ] string -> sig
 4    affine resource pwd_bit                    (* = [ pwd ] *)
 5    val check : string -> [ pwd_bit ] bool     (* impl as in Figure 8 *)
 6  end
```

```
 7  open mk_checker pass importing pwd_bit, check in
 8   let _ : [ pwd_bit ] bool = check "faxe"
 9   let _ : [ pwd_bit pwd_bit ] bool = check "faxe" && check "tibe"
10   let _ : [ pwd_bit ] bool = check "faxe" && check "tibe" ⚠
```

Fig. 14. Quantitative Information Flow

4.2 Quantitative Information Flow | -C

Removing contraction provides quantity-sensitivity. This can be applied towards a range of practical problems, of which we will show two particularly simple cases. The first is a rudimentary form of *quantitative information flow*. Recall check from Figure 8, which tracks the dependency of its return value on password data. Leaking password data in general is off limits, but perhaps it is acceptable to leak a certain *amount* of it. Figure 14 develops this idea, introducing a function mk_checker which takes a **string** at the previous structural resource pwd and returns an existential. The latter introduces an **affine resource** pwd_bit backed by pwd. The exported check function now returns a **bool** annotated with pwd_bit instead of pwd. Assuming check is implemented as before we are able to turn a [pwd] into a pwd_bit here because T-Pack factors out the former for the latter.

We deploy this machinery on lines 7-10. For convenience we assume a lifted && which works on […] **bool**s. Since pwd_bit is **affine** and only appears on the return value of check it indicates the number of applications of that function in some computation. On line 8 we invoke it once, which produces one pwd_bit. On the next it is invoked twice, producing pwd_bit twice. The last line fails typing because we invoke check twice but only one occurrence of pwd_bit appears in its annotation. So the effect is that **we can control *exactly how much* data some computation is permitted to be dependent on—one or two bits of pass here—by tying an affine resource to the output of a particular function.** Recall from T-Open that the return type of the body of the **open** expression must not depend on pwd_bit. We can exchange the pwd_bit resource back into pwd once we are finished verifying a program block via a function like finish : [pwd_bit] **bool** -> [pwd] **bool** exported from the existential alongside check. We hide it here for brevity.

The reasoning we have shown here is again formally sound, on two axes this time. **We analyze the form of quantitative information flow shown via intersecting concerns of *quantity safety* and *non-interference*.** Corollary 4.2 establishes the first, and can be interpreted in our setting as "if e is a well-typed at [pwd_bit pwd_bit pwd_bit] A and pwd_bit lacks contraction then forcing e generates an evaluation trace which contains at most three instances of pwd_bit."

Corollary 4.2 (Quantity Safety). *If* $\Delta; \Gamma \vdash e : A \mid \phi_1$ *where* $\overline{\rho^m}^n \in \phi_1$ *and* $\textsc{cb}, \textsc{cf} \notin m$ *then* $e \longmapsto^* v \triangleright \phi_2$ *where* $\overline{\rho^m}^k \in \phi_2$ *and* $n \geq k$.

Gouni et al. [2025a] establishes *non-interference*, the fundamental theorem of information flow, for the fully structural fragment of Qrtt. The idea is that if we have a function e whose input resources ϕ_1 are not represented in its output ϕ_2 then it behaves as the constant function. In other words, computations cannot depend on information from inputs not accounted for by their

```
1  module Serial : sig
2    linear resource lc
3    val init : [ lc ] unit
4    val incr : forall ρ . [ lc ρ ] unit -> [ lc ρ ] unit
5    val decr : forall ρ . [ lc ρ ] unit -> [ lc ρ ] unit
6  end = ...
```

```
7  open Serial importing lc, incr, decr in
8    let _ : [ lc ] unit = incr init
9    let _ : [ lc ] unit = incr (decr (incr init))
10   let _ : [ lc ] unit = !(incr init); !(decr init) ⚠
```

Fig. 15. Linear Capabilities

type. Taken together, this theorem and the preceding one establish the soundness of Figure 14. Non-interference ensures that types capture dependencies on password data, while quantity safety ensures that the number of pwd_bits appearing in the type is accurate to evaluation.

The structural version of non-interference suffices in this instance because we use it only to determine that data cannot affect a computation without inducing its attached resources; since this holds in the more permissive structural setting, it should also be true in the more restrictive substructural one. Corollary 4.2 does most of the difficult work, ensuring that resources that *do occur* are faithful to evaluation. For completeness, we have extended the prior non-interference result to full QRTT in Appendix E of Gouni et al. [2025b]. As noted in Section 2.2, however, it is not the focus of this paper. Finally, the approach to tracking quantitative information leakage displayed here is significantly more restrained than methods based on e.g. Shannon entropy [Denning 1982], min-entropy [G. Smith 2009], or differential privacy [Reed and Pierce 2010]. The advantage of our technique is in its ability to locally strengthen information flow tracking while integrating seamlessly with the other reasoning tools provided by the language. Extending our approach to more sophisticated notions of leakage is left as future work.

4.3 Linear Capabilities | -C -W

The excerpt with which we began our discussion of capabilities stated that *reproduction of a capability is legal.* What if it wasn't? An interesting form of concurrency reasoning emerges [Castegren and Wrigstad 2016]. Figure 15 provides the building blocks for speaking about an **int** cell of memory to which writes must be synchronized, specifically increment and decrement operations. Our goal is to thread a linear capability through invocations of these operations such that they are forced to be executed in a serial, or well-defined, way.

We start by declaring a **linear resource** lc, which is the capability in question. Reading just the resource annotations in each type signature, init performs initialization, providing the lc used to jump-start the computation. The function incr takes in this capability and gives it back, and so does decr. A **forall** ρ appears in the signature of each to account for potential extraneous resources, as in Figure 13. Progressing to the usages on lines 7-10, on line 8 we increment the cell, ending with lc. On the next we tack on an extra decr and incr in series, resulting again in lc. However, as line 10 shows it is not permissible to compose incr and decr in "parallel" under the same type, passing the capability along two separate paths corresponding to incr init and decr init. Two instances of lc will be produced instead, giving the type [lc lc] **unit**. Note that sequencing can be encoded via function application. Having more than one lc isn't strictly

```
module DropProto : sig
  { wea } resource drop
  { mob.back con.back wea } resource high    (* mob & con inverted here! *)
  resource low
  val drop : [ drop ] unit
  val op_hi1 : [ high ] unit
  val op_hi2 : [ high ] unit
  val op_lo : [ low ] unit
end = ...
```

```
open DropProto importing drop, high, low, drop, op_hi, op_lo in
  let _ : [ high drop low ] unit = !op_hi1; !op_hi2; !drop; !op_lo
  let _ : [ high drop low ] unit = !op_lo; !op_hi1
  let _ : [ high drop low ] unit = !op_hi1; !drop; !op_lo; !op_hi2 ⚠
```

Fig. 16. Modeling seccomp

wrong depending on the situation: we might interpret `[ lc lc ]` **unit** as the maximum permitted degree of concurrent execution being two. That said, under just one permitted `lc` the program is structured serially as expected, as chains of `incr` and `decr` operations.

The above example bears a resemblance to the *exclusive* capabilities from Castegren and Wrigstad [2016]. They leverage a non-interference theorem between concurrent computations possessing disjoint capabilities to ensure safety. This is the same basic principle by which the soundness of our example is ensured, in addition to quantity safety. As in Section 4.1, we can create (or **open**) a module for each unit of concurrency. Within each unit serialization requirements will be enforced, but disjoint units might execute concurrently. We only show the type mechanics here; prior work [Stork et al. 2014] has explored runtime support for concurrency control via capabilities. We look now to order-sensitivity and protocols applied to modeling security—rather than concurrency—constraints.

4.4 Privilege-Dropping Protocols and seccomp | -MF -CF

Ordered reasoning can provide a mechanism akin to `seccomp`, a sandboxing technology available on Linux and used by sensitive software like webservers and browsers. `seccomp()` is a system call which drops privileges for the calling process: it establishes a contract with the kernel stating that the process is hereafter to use only a limited set of system calls. If this contract is violated, the process is killed. This fits a useful pattern in much software where many more permissions are needed at initialization—e.g. reading and creating files—than will be needed later.

$$\cdots \xrightarrow{\quad\text{high privilege}\quad} \texttt{seccomp} \xrightarrow{\quad\text{low privilege}\quad} \cdots$$

Figure 16 begins by declaring a **resource** `drop` which has only **wea**kening, not **con**traction or **mob**ility. Working in a setting with relaxed mobility fragments contraction into forward and backwards shards, reflected in the substructural specification for the **resource** `high`. The declaration `mob.back con.back wea` provides it *backward mobility* and *backward contraction* in addition to weakening, and only those. Finally, we declare a structural **resource** `low`. To see how these all work together, consider the subtyping $\boxed{?} \sqsubseteq_{\texttt{drop,low,high}}$ `[ high drop low ]`. Valid and invalid fillings of $\boxed{?}$ are shown in Figure 17.

[high high drop low]   [high low drop low]   [low drop high] ⚠   [high drop]
[high low drop]   [low low]   [low high]   [drop high] ⚠

Fig. 17. Compatible Resource Usages for the Privilege-Dropping Specification

These indicate resource usages which may be expressed by computations typed under the specification [high drop low], except for those colored in **red**, which cannot be. The crucial restriction is that a high usage may not appear after drop. So [drop high] produces an error. This is because high is specified to appear before drop as the supertype, and backward-mobility and contractibility only permit it to move *right*, but not *left*. This is reversed from our exposition of the structural rules in Section 3.1, because **the typing judgement lists resources in reverse temporal order, but we use the usual temporal ordering in the surface syntax appearing in Figure 16.** So while multiple uses of high to the left of drop can be contracted, any following drop are stuck to the right of it. The intuition is that high is forced to stay to the back of drop.

The examples in Figure 16 make use of this. On line 11 we use two high operations, then use drop followed by a low operation. The next uses a low operation followed by a high one. This is also acceptable because drop is never called. The last fails to typecheck under the expected protocol because a high operation op_hi2 is used after the privilege to do so is dropped. All these examples are sound under Corollary 4.3.

Corollary 4.3 (Protocol Safety). *If* $\Delta; \Gamma \vdash e : A \mid \phi_1$ *then* $e \longmapsto^* v \triangleright \phi_2$ *where* $\phi_2 \sqsubseteq_\Delta \phi_1$.

Specialized to our case, it states that "if e is well-typed at resources [high drop low] then evaluating e produces some ϕ_2 such that $\phi_2 \sqsubseteq_\Delta$ [high drop low]." In other words, ϕ_2 must be structured according to the possible fillings of $\boxed{?}$ above. Corollary 4.3 is the fully general form of the safety theorem and implies Corollary 4.2 and Corollary 4.1; we use it here due to the extensive mode mixing in use, rather than one specialized to ordered reasoning.

4.5 Authorization Protocols | -MB -CB -W

Rather than privilege *dropping*, one can also reason about privilege *escalation* or authorization protocols. The picture is exactly inverted from the prior section, where an operation denoting authorization enters a *higher* privilege level and unlocks sensitive operations. Consequently, where before the drop resource was weakenable, the auth resource we shall introduce cannot be.

```
module AuthProto : sig
  { } resource auth
  { mob.fwd con.fwd wea } resource high
  resource low
  val auth : [ auth ] unit
  val priv : [ high ] unit
  val pub : [ low ] unit
end = ...
```

```
open AuthProto importing auth, high, low, auth, priv, pub in
  let _ : [ low auth high ] unit = !pub; !pub; !auth; !priv
  let _ : [ low auth high ] unit = !pub; !priv ⚠
  let _ : [ low auth high ] unit = !pub; !priv; !auth; !priv ⚠
```

Fig. 18. Modeling authorization

Figure 18 begins by declaring an `ordered resource auth`. The `resource high` is now *forward*, rather than *backward*, mobile and contractible. As before, we have a structural `resource low`. Going by the typical semantics of an authorization flow, a good guess for our desired resource annotation here is `[ low auth high ]`. Observe that `low` and `high` have exchanged positions from in Section 4.4. Let's replicate the exercise of seeing which resource contexts are permissible as subtypes of this one: $\boxed{?} \sqsubseteq_{\mathtt{auth,low,high}}$ `[ low auth high ]`. Valid and invalid fillings of $\boxed{?}$ are shown in Figure 19.

```
[ low low auth high ]   [ low auth high low ]   [ high auth low ] ⚠   [ auth high ]
     [ low auth ]             [ low low ] ⚠       [ low high ] ⚠    [ high auth ] ⚠
```

Fig. 19. Compatible Resource Usages for the Authorization Specification

Observe that where simply not producing `drop` was permitted before, the contexts `[ low low ]` and `[ low high ]` now give an error. This is because `auth` cannot be weakened into them to make a valid subtype. The pattern here is that `auth` must be present, and `high` must follow `auth`. The latter is because `high` is specified to appear after `auth` in the supertype, and forward-mobility and contractibility only permit it to be moved *left*, but not *right*. This is again reversed from Section 3.1. Uses of `high` before `auth` are stuck to its left, so `high` is forced to appear `fwd` of `auth`.

The usages on lines 10-12 of Figure 18 echo this. On line 10 we use two public operations, followed by `authoriz`ing, followed by a `private` operation. This adheres to the protocol. The next line violates it, attempting to use a private operation without any authorization. The next line contains an authorization, but a private operation is used *before* it, and we get another error. In a realistic setting `auth` would of course be produced by a function being passed authentication information, perhaps a capability as in Section 4.1. We isolate protocol reasoning here. This reasoning is sound under Corollary 4.3 for the same reasons as for Section 4.4.

It is reasonable to wonder whether we can force the presence of `auth` *only* if `high` operations are used. An extension of QRTT with unions and subtyping over them, perhaps in the style of Algebraic Subtyping [Dolan 2016], would likely be able to capture this via *unions* of permissible protocols. We leave this to future work.

4.6 More Examples

Now that we have a number of basic tools at our disposal, we quickly review encodings for additional reasoning mechanisms. These arise as variations of what we have already seen.

(1) **Capability Revocation | `-MF -CF -W`:** Section 4.1 avoided the issue of capability *revocation*: the problem of removing access permissions after they have already been delegated via a capability. We can leverage the privilege dropping strategy seen in Section 4.4 to prevent certain functions from being used after a revocation event occurs. This means we can revoke capabilities without relying on a dynamic access control policy, as conventionally required.

(2) **Rate Limiting | `-MB -C -W`:** We can fully remove contractibility from the schema for authorization in Section 4.5 to produce an approach to rate limiting. It can be specified that some affine operation may only done after an authorization, and only in limited quantity. So each authorization 'unlocks' a fixed quantity of extra computational budget.

(3) **Lightweight Typestate | `-MB -CB -MF -CF -W`:** It is possible to *combine* the schemas for privilege-dropping and authorization protocols to guard resources on *both sides*. We first introduce a backward contractible and mobile resource `op_l` and a forward contractible and mobile resource `op_r`. We then introduce two ordered resources `open` and `close`, produced by operations `open` and `close`. We imagine functions `read : ... -> [ op_l op_r ] string` and `write : ... -> [ op_l op_r ] unit` and use a resource context `[ open op_l op_r close ]`

as our specification. Under this regime, it is impossible to call `read` or `write` before `open` because there will be a stray `op_r` stuck before the `open` resource. Likewise, it is impossible to call `read` or `write` after `close` due to `op_l` being stuck after the `close` resource. As with capabilities we can speak about operations on *specific* files via existential resources representing them. We show here the common open-use-close protocol, but this pattern can be chained for more complex cases.

(4) **Authorization Logic | `-C` `-W`:** It is possible to replicate within our system part of the *authorization logic* BL found in Garg and Pfenn [2006]. BL reasons about authority via a monadic P says A construct which permits a principal P to affirm its belief in a proposition A describing e.g. access to a file or other resource. We can emulate this in our setting by attaching resource variables representing principals to types capturing access permissions. So if `c1` stands for a principal P_1, the type `[ c1 ] file` can be interpreted as P_1 giving its approval for the holder to access the indicated file. Making `c1` into a capability permits modeling more complex prerequisites for P_1 to provide its affirmation. Further work by Morgenstern et al. [2011] extends BL with limited-use affirmations via linearity, and we can follow a similar recipe to Section 4.3 to account for this extension.

We expect that more examples can be captured beyond what we have covered or suggested. Though we annotated each usage site manually with its expected resources here, the impredicative encoding of existentials from Gouni et al. [2025a] can be used to specify valid resource outputs in the module signature—akin to fixing the output type of T-Open. Note that the preceding examples make an effort to adhere closely to Qrtt as presented in Section 3, so the syntactic overhead of e.g. forcing and suspension is expected. Work on a higher-level language is left to the future. With the hindsight of our examples, we review the strengths of output-based resource reasoning.

4.7 Versus Input-Based Resource Reasoning

Where ordinary resource reasoning works in terms of computations *consuming* resources from their inputs, our system instead speaks about computations *producing* resources into their outputs. This enables us in Section 4.1 to realize the absence of weakening as a form of capability reasoning, since capabilities are fundamentally a matter of a resource—an authorization token—being *produced* by some computation—the authorization function—rather than *consumed*. The connection discussed in Section 4.2 between contraction and a coarse form of quantitative information reasoning also takes advantage of the production perspective, under the view that we wish to track the amount of information being *generated*. The producer-consumer distinction precisely characterizes effects versus coeffects [Gaboardi et al. 2016], thus we treat resources from an *effectful* perspective.

It is interesting to note that the output-based method separates the behavior of resources under usage, such as affinity or strictness, from their usage constraints; only the former must be specified *a priori*, unlike typical input-based systems. Their exact usage constraints may vary by the context in which they are used, according to the specified type. So two distinct subexpressions may mention the same resources under e.g. different protocols in the style of Section 4.4 with minimal friction. We suspect this will lead to better support for incremental programming because resource constraints are not assumed before the program is written, but can be thought of as bubbled up from its parts. Of course, constraints can still be specified top-down by simply fixing the desired types.

From a theoretical perspective it is highly significant that our system does not alter the structure of the typing context as substructural type theories typically do, which has presented difficulties in the face of more advanced type features such as value dependency [McBride 2016]. We believe the output-based setup is more straightforwardly open to extension with complex features. We proceed now to substantiate the soundness theorems referenced during the preceding discussion.

$$\text{V-LAM}\quad \frac{}{\lambda(x.e)\ \mathsf{val}} \qquad \text{E-AP-CONG1}\quad \frac{e_1 \mid \phi \longmapsto e_1' \mid \phi'}{\mathsf{ap}(e_1; e_2) \mid \phi \longmapsto \mathsf{ap}(e_1'; e_2) \mid \phi'} \qquad \text{E-AP-CONG2}\quad \frac{e_1\ \mathsf{val} \qquad e_2 \mid \phi \longmapsto e_2' \mid \phi'}{\mathsf{ap}(e_1; e_2) \mid \phi \longmapsto \mathsf{ap}(e_1; e_2') \mid \phi'}$$

$$\text{E-AP-}\beta\quad \frac{e_2\ \mathsf{val}}{\mathsf{ap}(\lambda(x.e); e_2) \mid \phi \longmapsto [e_2/x]e \mid \phi} \qquad \text{E-TRACK}\quad \frac{}{e^{\phi} \mid \phi' \longmapsto e \mid \phi\ \phi'}$$

Fig. 20. Tagged Operational Semantics / Dynamics (Selected Rules)

5 Metatheory

We first review the operational semantics used in the statements of the corollaries in Section 4. A logical relations argument is given which uses these semantics to show our core safety properties.

5.1 Tagged Operational Semantics

We have repeatedly referred to collections of resources ϕ in our development as having the type- and proof-theoretic *structure* of *effects*. As we saw in Section 4, these resources must be generated during evaluation for the statements of Corollary 4.1, Corollary 4.2, and Corollary 4.3 to make sense. While typical languages with effectful semantics have specialized constructs like `print` or `tick` which produce the appropriate effect when executed, it is unclear which construct(s) in QRTT should produce resources under evaluation. Our language is pure and does not have conventionally effectful constructs like `print`, yet we still want to assign computational meaning to resources.

A tempting but incorrect move is to annotate the introduction form $\#e$ for a resource type $[A \cdot \phi]$ with the ϕ in the type, turning it into $\#_{\phi}e$. We could then create a semantics where executing $!\#_{\phi}e$, assuming $\Delta; \Gamma \vdash e : A \mid \phi$, dynamically produces the resources ϕ. However, this is not sound: stacking another introduction and elimination onto the expression as in $!\#_{\phi}!\#_{\phi}e$ would induce a run-time effect $\phi\ \phi$, even though the type will have been unchanged.

The first step is to realize that *closed programs statically possess no ambient resources.* That is, if $\varnothing; \varnothing \vdash e : A \mid \phi$ then ϕ must be empty by well-scoping or regularity (mentioned at the end of Section 3.2.1) since $\Delta = \varnothing$ contains no resource variables. There could be resources encapsulated inside A, say, if A is a quantifier, but by well-scoping these cannot be present in the ambient ϕ outside their binders. Consequently, e must not witnesss any resources under evaluation, because this would violate what is dictated by typing. The crucial insight is that the resource variables available in Δ regulate which effects should be witnessed.

The next step is to ask how a resource is statically produced once we have it in scope in Δ. It turns out that given $\Delta; \Gamma \vdash e : A \mid \phi$, the contents of Γ are all that can be used to contribute *computationally* to ϕ. This is due to subtleties arising from weakening. Firstly, observe that there is no impetus to witness weakenable resources dynamically because they may have been obtained "for free" statically via weakening, rather than as a matter of some computation running. So these are computationally *irrelevant.* Further, those not obtained via weakening must have been produced by e.g. forcing assumptions or applying and forcing assumed functions which produce them, as we did in Section 4. The *only* way to obtain non-weakenable resources is by assuming into Γ computations which produce them, and using those. So all computationally relevant resources—those which we should witness as part of evaluation—emerge from assumptions. **In-scope resource variables and the term variables mentioning them determine the set of effects to be tracked operationally.**

$$
\begin{aligned}
e \overset{*}{\in} A \mid \phi\ [\Delta] &\triangleq \exists \phi_1 \sqsubseteq_\Delta \phi\ .\ v\ \mathsf{val},\ e \mid \circ \longmapsto^* v \mid \phi_1,\ v \in A\ [\Delta] \\
\langle\rangle \in \mathtt{unit}\ [\Delta] &\triangleq \top \\
\mathtt{l} \cdot v \in A_1 + A_2\ [\Delta] &\triangleq v \in A_1\ [\Delta] \\
\mathtt{r} \cdot v \in A_1 + A_2\ [\Delta] &\triangleq v \in A_2\ [\Delta] \\
\langle v_1, v_2 \rangle \in A_1 \otimes A_2\ [\Delta] &\triangleq v_1 \in A_1\ [\Delta], v_2 \in A_2\ [\Delta] \\
\mathsf{pack}[\phi](v) \in \exists(\rho^m.A)\ [\Delta] &\triangleq \Delta \vdash \phi, \phi \geq m, v \in [\phi/\rho^m]A\ [\Delta] \\
v \in [A \cdot \phi']\ [\Delta] &\triangleq\ !v \overset{*}{\in} A \mid \phi'\ [\Delta] \\
v \in A_1 \to A_2\ [\Delta] &\triangleq v_1 \in A_1\ [\Delta],\ v_1\ \mathsf{val} \implies \mathsf{ap}(v; v_1) \overset{*}{\in} A_2 \mid \circ\ [\Delta] \\
v \in \forall(\rho^m.A)\ [\Delta] &\triangleq \Delta \vdash \phi,\ \phi \geq m \implies v[\phi] \overset{*}{\in} [\phi/\rho^m]A \mid \circ\ [\Delta]
\end{aligned}
$$

Fig. 21. Logical Relation for Capability, Quantity, and Protocol Safety

The goal is to transform *each* assumption in Γ into a primitive effect by the resources it produces when used. E-Track in Figure 20 helps us do this. We introduce a new syntax for *tagged* expressions e^ϕ which, when evaluated, produce a run-time ϕ which is concatenated with any previously produced run-time resources ϕ'. The idea is to tag assumptions to give them an effect-like run-time semantics. Importantly, e^ϕ is *not* represented in the typing rules. Its absence ensures that well-typed programs do not contain it, which could otherwise be abused to miscount resources. **The safety proof will instrument programs with tagged expressions, and it must be the only machinery which is capable of doing so.** For an assumption $x : [[A \cdot \phi_2] \cdot \phi_1]$ we might associate a tagged expression to it of the form $\#(\#(e^{\phi_2})^{\phi_1})$. Forcing and evaluating it fully would dynamically produce $\phi_2\ \phi_1$. Observe that the resource context ϕ in the evaluation judgement is again temporally reversed, to mirror the typing judgement in Section 3.2. Our operational semantics otherwise follow a standard call-by-value strategy, as shown by the rules for evaluating function applications. The contained dynamic resource context ϕ grows throughout evaluation.

At first blush, it is odd to add a construct to the language which is not typable. One could imagine an alternative where we modify Qrtt by specially postulating primitive effects, and track those. Committing to certain primitives in this way loses much generality and elegance, however. Our strategy takes inspiration from the *semantic* approach used by Timany et al. [2024] in reasoning about programs which use unsafe constructs. The latter are not soundly typable, but their soundness can nevertheless be formally accounted for by analyzing their run-time behavior. This is the essence of our approach, generalizing the *labelled transition relation* of Plotkin and Power [2001] and others to account for 'effects' not readily visible in the syntax of typed programs. We have not yet specified *how* assumptions get tagged; the safety proof provides clarity.

5.2 Safety Properties

Figure 21 shows the setup for the logical relation, comprised of two mutually recursive sub-relations. We start out with the starred $e \overset{*}{\in} A \mid \phi\ [\Delta]$ relation on the first line, read as "e is *semantically* in A using resources ϕ scoped under Δ." Its definition states that e must evaluate to a value v producing some resources ϕ_1 substructurally compatible with ϕ where $v \in A\ [\Delta]$. In other words, the logical relation forces any expressions valid under it to be dynamically faithful to their declared resources. It is now clear why the syntax $\#e$ for $[A \cdot \phi]$ must suspend: since it encapsulates resources, if we were to evaluate inside it, resources would be dynamically produced which aren't yet expected by the logical relation. And more broadly, since negative (positive) types (don't) suspend [Levy 1999] they must (not) encapsulate resources. Observe that the use of $\sqsubseteq_\Delta$ handles weakening: any weakenable resources expected but not witnessed during evaluation will simply be assumed here.

The $v \in A\ [\Delta]$ relation, defined on values, interprets each type into its meaning as defined by the operational semantics. It is bifurcated, as with the type system itself, between positive and negative. The meaning of positive types is given by their introduction forms, and that of negative types is given by their behavior under elimination. For instance, `unit` is defined simply by being of the form $\langle\rangle$, and pairs by the form $\langle v_1, v_2\rangle$ where each of v_1 and v_2 satisfy the value relation at the right or left types of the pair. Functions, being negative, are defined not by their evaluated forms but by their behavior under application; they defer to the starred relation because the application is not a value and must be evaluated. The definition of each individual connective follows from its polarity.

We step through the core elements of the safety proof. Membership in the logical relation is preserved by *reverse evaluation*: any e' that evaluates to some e in the logical relation without use of resources is itself in the logical relation. The effect is that the logical relation cares only about the behavior of programs under evaluation, not their precise syntactic form.

Lemma 5.1 (Reverse Closure). *If $e \overset{*}{\in} A \mid \phi\ [\Delta]$ and $e' \mid \circ \longmapsto^* e \mid \circ$ then $e' \overset{*}{\in} A \mid \phi\ [\Delta]$.*

Proof. By use of evaluation in the starred relation and transitivity of evaluation. □

Membership is also monotonic across both resource contexts and types. The second theorem will be needed to account for the soundness of the subsumption rule T-Sub.

Lemma 5.2 (Monotonicity). *If $e \overset{*}{\in} A \mid \phi\ [\Delta]$ and $\phi \sqsubseteq_\Delta \phi'$ then $e \overset{*}{\in} A \mid \phi'\ [\Delta]$.*

Proof. By use of $\sqsubseteq_\Delta$ in the starred relation and transitivity of $\sqsubseteq_\Delta$. □

Lemma 5.3 (Type Monotonicity). *If $e \overset{*}{\in} A \mid \phi\ [\Delta]$ and $A \sqsubseteq_\Delta A'$ then $e \overset{*}{\in} A' \mid \phi\ [\Delta]$.*

Proof. By induction on A and application of Lemma 5.2. □

Before we proceed, we generalize the logical relation. The machinery we have introduced only operates on *closed* expressions, but to connect it to the static type system, we must account for expressions with *free variables*. This is because the typing operates on *open* expressions which mention free variables. We begin by defining *closing substitutions* which replace free resource variables ρ^m and free term variables x with appropriate forms. Specifically, define:

(1) $\delta \in \Delta \rightsquigarrow \Delta'$ by sending:
  (a) $\rho^m \in \Delta$ to a resource context ϕ such that $\Delta' \vdash \phi$ and $\phi \geq m$.
  (b) $\rho^m \notin \Delta$ to ρ^m.
(2) $\gamma \in \Gamma\ [\Delta \rightsquigarrow \Delta']$ to mean that if we have $\delta \in \Delta \rightsquigarrow \Delta'$ then there is a map sending $x : A \in \Gamma$ to a value v such that under the value relation we have $v \in \widehat{\delta}(A)\ [\Delta']$.
(3) $\Delta\ \Gamma \gg_{\Delta'} e \overset{*}{\in} A \mid \phi$ to mean that for some Δ' if $\delta \in \Delta \rightsquigarrow \Delta'$ and $\gamma \in \Gamma\ [\Delta \rightsquigarrow \Delta']$, the latter indexed with δ, then under the starred relation we have $\widehat{\delta}(\widehat{\gamma}(e)) \overset{*}{\in} \widehat{\delta}(A) \mid \widehat{\delta}(\phi)\ [\Delta']$.

δ replaces resource variables ρ^m contained in its domain Δ with resource contexts ϕ closed under a given Δ'. Similarly, γ replaces term variables x with closed values logically related at the variable's type, which is itself is closed using δ. $\widehat{\delta}$ and $\widehat{\gamma}$ are variants of δ and γ applied *simultaneously* to all open variables in their arguments. That is, they take the mappings in δ from resource variables ρ^m to resource contexts ϕ and in γ from term variables x to values v and create simultaneous substitutions $[\phi_1 \ldots \phi_n / {\rho_1}^{m_1} \ldots {\rho_n}^{m_n}]$ and $[v_1 \ldots v_n / x_1 \ldots x_n]$. Importantly, note that while $\widehat{\delta}$ and $\widehat{\gamma}$ can appear in type and expression position, they are not program syntax! They are only used to execute simultaneous substitutions. The generalized logical relation in item 3 is defined by assuming closing maps δ and γ, then using $\widehat{\delta}$ and $\widehat{\gamma}$ on its expression e and $\widehat{\delta}$ on its type A and resource context ϕ, and finally invoking the starred relation we saw from Figure 21.

The fundamental theorem constitutes the core of the proof and connects well-typedness to membership in the logical relation. With the generalized logical relation in hand, the statement is straightforward: given a well-typed open expression we can obtain an instance of the generalized logical relation. Expanding the latter, this means using the closing substitutions δ and γ from the generalized logical relation on the open expression from the typing judgement and ensuring that the now-closed expression behaves as defined by the starred logical relation. So it must dynamically witness resources according to its declared type and resource signature.

Theorem 5.4 (Fundamental Theorem). *Fix Δ_0. If $\Delta_0, \Delta; \Gamma \vdash e : A \mid \phi$ then $\Delta\ \Gamma \gg_{\Delta_0} e \overset{*}{\in} A \mid \phi$.*

Proof. By induction on the height of a derivation of $\Delta_0, \Delta; \Gamma \vdash e : A \mid \phi$. □

Recall our comment from Section 5.1 that the set of effects we wish to track under evaluation is determined by the *in-scope resource variables.* This is what δ does. The theorem fixes a Δ_0 to define *which resources we're concerned with.* Δ_0 is the set of resources mapped into by δ. So all resources witnessed dynamically under the logical relation will be in Δ_0, since types and resource contexts will be closed under it by δ. Recall a further comment from the prior section that *for an assumption $x : [[A \cdot \phi_2] \cdot \phi_1]$ we might associate a tagged expression to it of the form* $\#(\#(e^{\phi_2})^{\phi_1})$. This is the exact function of γ, which **introduces tracked expressions.** The expression e from the typing judgement is instrumented with tags when $\widehat{\gamma}$ replaces variables with values of corresponding type in the logical relation, which witness the resources indicated in their type when used. We illustrate the mechanics here with a concrete example.

Assume we have ${\rho_1}^m, {\rho_2}^m; x : [[\texttt{unit} \cdot {\rho_1}^m] \cdot {\rho_2}^m] \vdash\ !x : [\texttt{unit} \cdot {\rho_1}^m] \mid {\rho_2}^m$. We take the entire resource typing context as our base Δ_0 for simplicity, setting $\Delta_0 = {\rho_1}^m, {\rho_2}^m$. This allows us to sidestep δ since the typing judgement is already closed under ${\rho_1}^m, {\rho_2}^m$. To close our term variable context, γ might map $x \mapsto \#(\#(\langle\rangle^{{\rho_1}^m})^{{\rho_2}^m})$. This works since a value in the logical relation at x's type $[[\texttt{unit}\cdot{\rho_1}^m]\cdot{\rho_2}^m]$ may witness resources ${\rho_2}^m$ and ${\rho_1}^m$ when successively forced. Upon substituting we get $\widehat{\gamma}(!x) = !\#(\#(\langle\rangle^{{\rho_1}^m})^{{\rho_2}^m})$. Evaluation gives $!\#(\#(\langle\rangle^{{\rho_1}^m})^{{\rho_2}^m}) \mid \circ \longmapsto^* \#(\langle\rangle^{{\rho_1}^m}) \mid {\rho_2}^m$, producing a dynamic resource usage of ${\rho_2}^m$ where the typing predicted the same. This shows how the operational semantics, in concert with the closing map γ on term variables x, tracks the usage of assumptions within the main expression e from the typing judgement. The fundamental theorem ultimately demonstrates that if the assumptions behave as the expression expects—producing the required resources under evaluation—then so does the expression itself. The corollary from which the rest in Section 4 emerge follows immediately from the fundamental theorem.

Corollary 5.5 (Safety). *If $\Delta; \Gamma \vdash e : A \mid \phi_1$ and $\gamma : \Gamma\ [\circ \rightsquigarrow \Delta]$ then $\widehat{\gamma}(e) \mid \circ \longmapsto^* v \mid \phi_2$ where v val and $\phi_2 \sqsubseteq_\Delta \phi_1$.*

Proof. Immediate from Theorem 5.4 and the definition of the logical relation.

(1) Apply Theorem 5.4 to the typing assumption to obtain $\circ\ \Gamma \gg_\Delta e \overset{*}{\in} A \mid \phi_1$.
(2) Define $\delta \in \circ \rightsquigarrow \Delta$ to be the identity map sending all ρ^m to ρ^m.
(3) Apply γ to δ, calling the result γ_1.
(4) Apply item 1 to δ and γ_1 to obtain $\widehat{\delta}(\widehat{\gamma_1}(e)) \overset{*}{\in} \widehat{\delta}(A) \mid \widehat{\delta}(\phi_1)\ [\Delta]$.
(5) Since δ is the identity map $\widehat{\delta}(\widehat{\gamma_1}(e))$ is equivalent to $\widehat{\gamma}(e)$, recalling the definition of γ_1, and likewise for $\widehat{\delta}(A)$ and $\widehat{\delta}(\phi_1)$. So we have $\widehat{\gamma}(e) \overset{*}{\in} A \mid \phi_1\ [\Delta]$.

(6) We have from the definition of the starred logical relation that $\widehat{\gamma}(e) \mid \circ \longmapsto^* v \mid \phi_2$ where v val and $\phi_2 \sqsubseteq_\Delta \phi_1$.
(7) So we have $\widehat{\gamma}(e) \mid \circ \longmapsto^* v \mid \phi_2$ where v val and $\phi_2 \sqsubseteq_\Delta \phi_1$. □

If we have a well-typed expression and its free variables can be substantiated with semantically valid witnesses, then that expression will dynamically produce the expected resources. γ's signature indicates that we use the entire resource typing context Δ as the closing context for the logical relation—that is, we don't need to mention a resource variable substitution δ in the statement of the theorem, since we fix our perspective to the current scope. This forces us to define δ to be the identity map. We can obtain Corollary 4.3 directly from this one by defining $e \longmapsto^* v \triangleright \phi_2$ as "$\gamma : \Gamma[\circ \rightsquigarrow \Delta]$ implies $\widehat{\gamma}(e) \mid \circ \longmapsto^* v \mid \phi_2$ and v val." Corollary 4.1 and Corollary 4.2 are then immediate, as specializations of the previous. Full proofs are in Appendix E of Gouni et al. [2025b].

6 Related Work

We pause here and take stock of prior work and its relationship to the formalism just discussed. We discuss work in constructive Kripke semantics for substructural logics, contextualize our setup against the backdrop of literature on graded monads and modal type theory, and finish with a discussion on trace-based type and effect systems.

6.1 Programming with Constructive Kripke Semantics

Besides information flow, *resource semantics* [Reed and Pfenning 2010] provide another lens through which to view our system. Resource semantics have been deployed as a constructive technique for embedding intuitionistic substructural logics into intuitionistic logic. In bridging logics, they act as a *constructive Kripke semantics*. A typical resource semantics annotates each in-scope assumption with a *single* resource variable ρ, and whenever it is used (that is, whenever it appears) the resource context in the succedent adds another ρ. Formally we have $A_1[\rho_1], \ldots, A_n[\rho_n] \vdash C[\phi]$. Consider in Figure 22 a list of instances of this judgement for the linear case adapted from Pfenning [2023].

The inferences on lines 3 and 6 are unacceptable by the $\otimes$R rule since each uses assumptions not declared in the output resource variable environment. However, the inference on line 4 is acceptable despite ignoring the $R[\rho_3]$ assumption. This seems to violate the linear restriction that assumptions always be used, but recall that we are working with an *embedding* of linear logic *into* intuitionistic logic, which does not restrict its assumptions as such. That assumptions must be used is enforced on the *output environment* ϕ by the rules defining the embedding. The rule $\multimap$R for introducing linear implication requires that its body B witness the resource ρ for its assumption A. Observe that unlike T-Lam in Figure 10 this rule permits resource variables to leak freely through it. This choice contravenes Corollary 5.5: functions are negative so evaluation does not traverse inside them, yet here we would expect to observe resources from their bodies dynamically.

$$\otimes\text{R}\quad \frac{\Gamma \vdash A[\phi_1] \qquad \Gamma \vdash B[\phi_2]}{\Gamma \vdash A \otimes B[\phi_1\ \phi_2]}$$

$$\multimap\text{R}\quad \frac{\Gamma, A[\rho] \vdash B[\phi\ \rho]}{\Gamma \vdash A \multimap B[\phi]}$$

$P[\rho_1], Q[\rho_2] \vdash P \otimes Q[\rho_1\ \rho_2]$	✅	(1)
$P[\rho_1], Q[\rho_2] \vdash P \otimes Q[\rho_2\ \rho_1]$	✅	(2)
$P[\rho_1], Q[\rho_2] \vdash P \otimes Q[\rho_1]$	✗	(3)
$P[\rho_1], Q[\rho_2], R[\rho_3] \vdash P \otimes Q[\rho_1\ \rho_2]$	✅	(4)
$P[\rho_1], Q[\rho_2], R[\rho_3] \vdash P \otimes Q \otimes R[\rho_1\ \rho_2\ \rho_3]$	✅	(5)
$P[\rho_1], Q[\rho_2], R[\rho_3] \vdash P \otimes Q \otimes R[\rho_2\ \rho_3]$	✗	(6)

Fig. 22. Linear Resource Semantics

Despite the difference, $\multimap$R's formulation shows that **resource semantics are responsible for the output-based reasoning we noted in Section 1.** This picture needs slightly more generalization to arrive at our system. Each assumption in a resource semantics is judgementally annotated with just one resource variable ρ, but cannot carry multiple nor mention them in types. We may *internalize* resource variables—or give them their own connective—using the *satisfaction operator* from hybrid logic, shown in Figure 23 as @I and @E [Reed 2009]. Hybrid logic [Prior 1968] is an approach to syntactically internalizing worlds and quantification from the Kripke semantics of modal logics. It is deployed here to the same end but towards substructural logics.

$$\text{@I}\ \frac{\Gamma \vdash M : A\,[\phi]}{\Gamma \vdash M : @_{\phi}A\,[\phi']} \qquad \text{@E}\ \frac{\Gamma \vdash M : @_{\phi}A\,[\phi']}{\Gamma \vdash M : A\,[\phi]} \qquad \text{@E-NEW}\ \frac{\Gamma \vdash M : @_{\phi}A\,[\phi']}{\Gamma \vdash M : A\,[\phi\ \phi']}$$

Fig. 23. Satisfaction from Hybrid Logic

Figure 23 also adds program terms M. The setup for satisfaction $@_{\phi}$ looks quite similar to that for $[A \cdot \phi]$ because it is: resource types are underlyingly a kind of satisfaction. Just like $\multimap$R, however, the formulation of satisfaction given here is unsatisfactory. The first issue is that @E forgets resources—or information—ϕ' being depended upon, causing the non-interference theorem to fail; Gouni et al. [2025a] points this out. This can be remedied by replacing @E with @E-new as is done there. The more poignant and subtle issue is that our safety properties again fail unless $@_{\phi}$ is exactly a negative connective. Importantly, it must suspend its contents, following Levy [1999]. The reason for this is precisely that given for $[A \cdot \phi]$'s design in Section 5. @E-new does not remedy this because it does not give satisfaction any syntax which could induce suspension. Addressing this leads to the effectful structure we noted regarding $[A \cdot \phi]$ in Section 3. So the journey from hybrid logic and resource semantics to our system is completely determined by metatheoretic issues driven by our attention paid to computational behavior.

We deploy resource semantics as a programming language and explore their inherent resource reasoning abilities, rather than using them as a bridge to conventional regimes. It is due to this departure from the latter that invariants from conventional linearity such as non-aliasing become unnatural. It would be strange if these were naturally enforced by a bare resource semantics, because this would preclude needing embeddings like in Figure 22 to do so. On that note, the basis of our language as a resource semantics allows it to *meta*-express intuitionistic linear logic. This justifies our use of the terms affine, linear, ordered, and strict to refer to the activation of various combinations of structural rules within our system: they correspond to the usual rules working on typing contexts and can be leveraged to enforce ordinary substructurality. We outline the translation in Appendix A of Gouni et al. [2025b], deferring to Reed [2009] for the full result.

6.1.1 Quantification and Adjoint Logic's Shifts. Our language's status as a hybrid logic provides us important insights about quantifiers. Universal and existential quantification and their interaction with modes bears a striking similarity to the upshift and downshift connectives found in *Adjoint Logic* [Pruiksma et al. 2018]. There, upshift $\uparrow A$ and downshift $\downarrow A$ respectively act to make A either less or more restricted with respect to weakening and contraction (*adding* structural rules imposes *less* restrictions). Upshift is reflected in our system via universal quantification—by eliminating it—and downshift by existential quantification—by introducing it. Each of upshift and downshift is of the same polarity as its counterpart in our system. The Adjoint Logic analogues to our $\phi \geq m$ premises—essentially, that the antecedents and succedents with which shifted propositions interact are at compatible modes—are also checked while eliminating upshifts and introducing downshifts.

This is no mistake, and arises directly from the foundations of our language in hybrid logic and resource semantics. Hybrid logic, as mentioned, internalizes the Kripke semantics of modal logic, specifically worlds and quantification over them. Resource semantics use these syntactic worlds—which we call resource variables—to bridge substructural reasoning into a structural setting. Resource semantics can be thought of as internalizing Kripke models for substructural logics [Kamide 2002]. **Our quantification connectives and their mode-varying behavior, which has not been available in prior work on resource semantics, can be thought of as a syntactic reification of the Kripke semantics of shifts in Adjoint Logic.** Briefly, shifting a type *upwards* to be less restricted semantically involves a bounded universal quantification over all resources less restricted than the higher mode being shifted to, which it is permitted to use in its implementation. Shifting a type *downwards* to be more restricted involves a bounded existential quantification witnessing that the resources it used in its implementation are less restricted than the lower mode it shifted to. We leave a full development of this idea to future work.

6.2 Graded Monads and Modalities

The second body of prior work to which ours bears significant resemblance is that on graded coeffects and effects. In the latter case, *grading* refers to indexing effects with partially ordered monoids [Orchard, Wadler, et al. 2020]. Our resource contexts $[\ \rho_1\ \rho_1\ \rho_2\ ]$ are partially ordered monoids where the order is induced by weakening. Contraction and mobility relate to partial idempotency and commutativity, set up to preserve associativity. The unital element is the empty context $[\ ]$. In Section 3 we pointed out that a strong monad emerged from the resource type $[A \cdot \phi]$. In other words, **our system is close to one of graded effects.** We say *close* here because it is unclear to what extent the quantification and mode shifting behavior available in our system can be accounted for, and because the *directed* nature of contraction and mobility in QRTT precludes them from corresponding precisely to idempotency and commutativity *equations*.

Theories of graded modal types [Atkey 2018; Gaboardi et al. 2016; Ghica and A. I. Smith 2014; Orchard, Liepelt, et al. 2019; Petricek et al. 2014] generally focus on resource reasoning as a graded *coeffect* rather than as a graded *effect*, despite supporting the latter. Indeed, Orchard, Liepelt, et al. [2019] note that they "focus mainly on graded [coeffects], which [are] heavily integrated with linearity." Coeffects are connected to computational *inputs* [Petricek et al. 2014], whereas effects naturally speak about the *outputs* of a computation.

Literature which positions graded effects in service of resource reasoning includes Orchard and Yoshida [2016], which establishes the relationship between graded effects and session types. The latter are Curry-Howard correspondent to the linear sequent calculus [Toninho et al. 2012]. Danielsson [2008] uses a natural number graded monad analogous to that available in the purely affine fragment of our system to establish basic time complexity results of lazy functional programs. Hughes et al. [2025] recently developed an account of region-based memory management using effects, explicitly intending to avoid conventional substructural typing for better compatibility with C-like languages. These are all towards specific applications, however; isolating the new form of substructurality is not the object of study. Beyond these works there are extensive prior accounts of graded and indexed effects [Fujii et al. 2016; Katsumata 2014; McDermott and Uustalu 2022; Orchard, Wadler, et al. 2020]. Their connection to resource reasoning in the sense of substructurality has remained largely unexplored, though, with priority given instead to increasing the expressive power of standard effects. **Whereas most prior work focuses on the *general* framework of graded effects, we focus on *a particular class* of graded effects which provides a reimagination of conventional substructural reasoning and richer insights therein.** The soundness theorems in Section 5 are also more interesting than for graded systems studied previously [Orchard, Liepelt, et al. 2019], which largely cover syntactic type safety.

Finally, we are able to leverage our simplified setting both for more powerful abstraction facilities than in prior work and for smoother integration between our different modes of reasoning. On the first point, **QRTT supports existential and higher-rank quantification over grades whereas prior theoretical developments have focused on prenex polymorphism** [Orchard, Liepelt, et al. 2019].[5] This was essential in Section 4 for exposing resource variables with module-defined semantics. And from a practical perspective, prior work accounting for arbitrary grades is not user-extensible [Moon et al. 2021; Orchard, Liepelt, et al. 2019]; the language implementation must be modified with extra SMT encodings supporting new grades. Our unified framework accounts for a full range of examples under a single, simple set of mechanics.

6.3 Trace-Based Types

A final category of related work is in type and effect systems reasoning over traces of events produced by a program. Our resource contexts can be thought of as such traces, particularly within the ordered setting. Skalka and S. Smith [2004] employ an operational semantics much like that in Section 5 to track the history of operations executed by a program, likewise approximating them via a type system to capture temporal safety properties. Koskinen and Terauchi [2014] generalize this idea to encompass infinite traces, accounting for a number of *liveness* properties falling outside the focus of the current work on *safety*. More recently, Zhou et al. [2024] explored *symbolic finite automata* reified within types as a medium for constraining program execution. Their setting is richer in that program variables can be mentioned in traces, yet potentially also less expressive in that our per-resource mode variability may lie outside the class of formal languages supported there. We suspect our system offers strong logical justification to much prior work in this area, but leave establishing the precise nature of the correspondence to future work.

7 Conclusion

We have noted future work throughout, and summarize and supplement our comments here. The main points of future work are design considerations for a source-level language, some of which have been surfaced in the process of writing the examples in Section 4. These are not merely syntactic issues, but e.g. heuristics for precluding writing the forcing ! and suspension # operations. We already hinted at one such mechanic in Section 3.2.1 and Section 4 when we assumed that functions and `let` definitions would suspend their bodies if ambient resources were present. Beyond this, developing algorithmic type checking and inference is essential for a practical implementation. Adding recursive types would also be attractive, and present complications for certain fragments of the system: for instance, affine resources cannot be used straightforwardly with non-termination.

This work has presented a *highly general* output-based, effect-flavored approach to substructural reasoning, contrasting with the input-based, coeffectful approaches largely seen before. We have shown a simple, powerful type system for programming under this regime, invoking proof-theoretic ideas and connections to the literature to justify our design choices. We have shown it to be useful for encoding a wide range of programming problems, and expect it to be applicable to many more.

Data Availability Statement

Referenced appendices, auxiliary definitions, and full proofs are available at Gouni et al. [2025b].

Acknowledgements

We thank the anonymous reviewers for their thorough feedback; any remaining oversights or errors are ours. We would also like to thank Corinthia Aberlé, Harrison Grodin, Juli O'Connor, and

[5]Note that the Granule language has implemented higher-ranked quantification over grades as unpublished work.

Tesla Zhang for insightful discussions. This research was supported by the Department of Defense and the National Science Foundation under Grant Nos. H98230-23-C-0275 and CCF-1901033. The lead author was supported by a Sansom Endowed Presidential Fellowship.

References

Alan R. Anderson et al.. 1992. *Entailment, Vol. II: The Logic of Relevance and Necessity.* Princeton University Press.

Robert Atkey. 2018. "Syntax and Semantics of Quantitative Type Theory." In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (LICS '18), 56–65. doi:10.1145/3209108.3209189.

Nick Benton and Philip Wadler. 1996. "Linear Logic, Monads and the Lambda Calculus." In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science* (LICS '96), 420. doi:10.1109/LICS.1996.561458.

Elias Castegren and Tobias Wrigstad. 2016. "Reference Capabilities for Concurrency Control." In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (Leibniz International Proceedings in Informatics (LIPIcs)). Vol. 56, 5:1–5:26. doi:10.4230/LIPIcs.ECOOP.2016.5.

Nils Anders Danielsson. 2008. "Lightweight semiformal time complexity analysis for purely functional data structures." In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '08), 133–144. doi:10.1145/1328438.1328457.

Dorothy E. Denning. 1982. *Cryptography and data security.* Addison-Wesley.

Stephen Dolan. 2016. "Algebraic subtyping." Ph.D. Dissertation. University of Cambridge.

Matt Fairtlough and Michael Mendler. 1997. "Propositional Lax Logic." *Information and Computation*, 137, 1, 1–33. doi:10.1006/inco.1997.2627.

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. "Towards a Formal Theory of Graded Monads." In: *Foundations of Software Science and Computation Structures*, 513–530. doi:10.1007/978-3-662-49630-5_30.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. "Combining effects and coeffects via grading." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (ICFP 2016), 476–489. doi:10.1145/2951913.2951939.

Deepak Garg and Frank Pfenn. 2006. "Non-Interference in Constructive Authorization Logic." In: *Proceedings of the 19th IEEE Workshop on Computer Security Foundations* (CSFW '06), 283–296. doi:10.1109/CSFW.2006.18.

Dan R. Ghica and Alex I. Smith. 2014. "Bounded Linear Types in a Resource Semiring." In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, 331–350. doi:10.1007/978-3-642-54833-8_18.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. "Bounded linear logic: a modular approach to polynomial-time computability." *Theoretical Computer Science*, 97, 1, 1–66. doi:https://doi.org/10.1016/0304-3975(92)90386-T.

Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. Oct. 2025a. "Structural Information Flow: A Fresh Look at Types for Non-interference." *Proc. ACM Program. Lang.*, 9, OOPSLA2, Article 414, (Oct. 2025), 27 pages. doi:10.1145/3764116.

Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2025b. *Technical Report: Security Reasoning via Substructural Dependency Tracking*. Zenodo. (2025). doi:10.5281/zenodo.17772505.

Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. 2013. "A capability-based security approach to manage access control in the Internet of Things." *Mathematical and Computer Modelling*, 58, 5, 1189–1205. doi:10.1016/j.mcm.2013.02.006.

Jack Hughes, Michael Vollmer, and Mark Batty. 2025. "Spegion: Implicit and Non-Lexical Regions with Sized Allocations." In: *39th European Conference on Object-Oriented Programming (ECOOP 2025)* (Leibniz International Proceedings in Informatics (LIPIcs)). Vol. 333, 15:1–15:26. doi:10.4230/LIPIcs.ECOOP.2025.15.

Norihiro Kamide. Sept. 2002. "Kripke Semantics for Modal Substructural Logics." *J. of Logic, Lang. and Inf.*, 11, 4, (Sept. 2002), 453–470. doi:10.1023/A:1019915908844.

Max Kanovich, Stepan Kuznetsov, Vivek Nigam, and Andre Scedrov. 2018. "A Logical Framework with Commutative and Non-commutative Subexponentials." In: *Automated Reasoning*, 228–245. doi:10.1007/978-3-319-94205-6_16.

Max Kanovich, Stepan Kuznetsov, Vivek Nigam, and Andre Scedrov. 2019. "Subexponentials in non-commutative linear logic." *Mathematical Structures in Computer Science*, 29, 8, 1217–1249. doi:10.1017/S0960129518000117.

Shin-ya Katsumata. 2014. "Parametric effect monads and semantics of effect systems." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '14), 633–645. doi:10.1145/2535838.2535846.

Eric Koskinen and Tachio Terauchi. 2014. "Local temporal reasoning." In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (CSL-LICS '14) Article 59, 10 pages. doi:10.1145/2603088.2603138.

Paul Blain Levy. 1999. "Call-by-Push-Value: A Subsuming Paradigm." In: *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications* (TLCA '99), 228–242. doi:10.1007/3-540-48959-2_17.

Theodore A. Linden. Dec. 1976. "Operating System Structures to Support Security and Reliable Software." *ACM Comput. Surv.*, 8, 4, (Dec. 1976), 409–445. doi:10.1145/356678.356682.

Daniel Marshall and Dominic Orchard. Apr. 2024. "Functional Ownership through Fractional Uniqueness." *Proc. ACM Program. Lang.*, 8, OOPSLA1, Article 131, (Apr. 2024), 31 pages. doi:10.1145/3649848.

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.* Ed. by Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella. Springer International Publishing, 207–233. doi:10.1007/978-3-319-30936-1_12.

Dylan McDermott and Tarmo Uustalu. 2022. "Flexibly Graded Monads and Graded Algebras." In: *Mathematics of Program Construction: 14th International Conference (MPC 2022)*, 102–128. doi:10.1007/978-3-031-16912-0_4.

J. McLean. 1996. "A general theory of composition for a class of "possibilistic" properties." *IEEE Transactions on Software Engineering*, 22, 1, 53–67. doi:10.1109/32.481534.

Eugenio Moggi. 1989. *Computational lambda-calculus and monads.* IEEE Press, 14–23. doi:https://doi.org/10.1109/LICS.1989.39155.

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. "Graded Modal Dependent Type Theory." In: *Programming Languages and Systems*, 462–490. doi:10.1007/978-3-030-72019-3_17.

Jamie Morgenstern, Deepak Garg, and Frank Pfenning. 2011. "A Proof-Carrying File System with Revocable and Use-Once Certificates." In: *Security and Trust Management - 7th International Workshop, (STM 2011), Revised Selected Papers* (Lecture Notes in Computer Science). Vol. 7170, 40–55. doi:10.1007/978-3-642-29963-6_5.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. July 2019. "Quantitative program reasoning with graded modal types." *Proc. ACM Program. Lang.*, 3, ICFP, Article 110, (July 2019), 30 pages. doi:10.1145/3341714.

Dominic Orchard, Philip Wadler, and Harley Eades III. 2020. "Unifying graded and parameterised monads." In: *Eighth Workshop on Mathematically Structured Functional Programming (MSFP).* doi:https://doi.org/10.4204/EPTCS.317.2.

Dominic Orchard and Nobuko Yoshida. 2016. "Effects as sessions, sessions as effects." Principles of Programming Languages, 568–581. doi:10.1145/2837614.2837634.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. "Coeffects: a calculus of context-dependent computation." International Conference on Functional Programming, 123–135. doi:10.1145/2628136.2628160.

Frank Pfenning. 2023. "Lecture Notes on Resource Semantics." https://www.cs.cmu.edu/~fp/courses/15836-f23/lectures/19-resource.pdf.

Gordon Plotkin and John Power. 2001. "Adequacy for Algebraic Effects." In: *Foundations of Software Science and Computation Structures*, 1–24. doi:10.1007/3-540-45315-6_1.

A. N. Prior. 1968. ""Now"." *Noûs*, 2, 2, 101–119. doi:10.2307/2214699.

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

Jason Reed. 2009. "A hybrid logical framework." Ph.D. Dissertation. Carnegie Mellon University.

Jason Reed and Frank Pfenning. 2010. "Focus-preserving embeddings of substructural logics in intuitionistic logic." Unpublished manuscript. https://www.cs.cmu.edu/~fp/papers/substruct10.pdf.

Jason Reed and Benjamin C. Pierce. 2010. "Distance makes the types grow stronger: a calculus for differential privacy." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (ICFP '10), 157–168. doi:10.1145/1863543.1863568.

Sophia Roshal and Frank Pfenning. 2025. Unpublished communications. https://www.cs.cmu.edu/~fp/courses/15417-s25/lectures/18-adjord.pdf.

Christian Skalka and Scott Smith. 2004. "History Effects and Verification." In: *Programming Languages and Systems*, 107–128. doi:10.1007/978-3-540-30477-7_8.

Geoffrey Smith. 2009. "On the Foundations of Quantitative Information Flow." In: *Foundations of Software Science and Computational Structures*, 288–302. doi:10.1007/978-3-642-00596-1_21.

L. Snyder. Mar. 1981. "Formal Models of Capability-Based Protection Systems." *IEEE Trans. Comput.*, 30, 3, (Mar. 1981), 172–181. doi:10.1109/TC.1981.1675753.

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. Mar. 2014. "Æminium: A Permission-Based Concurrent-by-Default Programming Language Approach." *ACM Trans. Program. Lang. Syst.*, 36, 1, Article 2, (Mar. 2014), 42 pages. doi:10.1145/2543920.

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Nov. 2024. "A Logical Approach to Type Soundness." *J. ACM*, 71, 6, Article 40, (Nov. 2024), 75 pages. doi:10.1145/3676954.

Bernardo Toninho, Luis Caires, and Frank Pfenning. 2012. "Functions as Session-Typed Processes." In: *Foundations of Software Science and Computational Structures*, 346–360. doi:10.1007/978-3-642-28729-9_23.

Zhe Zhou, Qianchuan Ye, Benjamin Delaware, and Suresh Jagannathan. June 2024. "A HAT Trick: Automatically Verifying Representation Invariants using Symbolic Finite Automata." *Proc. ACM Program. Lang.*, 8, PLDI, Article 203, (June 2024), 25 pages. doi:10.1145/3656433.

Received 2025-07-10; accepted 2025-11-06