A Unified View of Modalities in Type Systems

ANDREAS ABEL, Gothenburg University, Sweden
JEAN-PHILIPPE BERNARDY, Gothenburg University, Sweden

We propose to unify the treatment of a broad range of modalities in typed lambda calculi. We do so by defining a generic structure of modalities, and show that this structure arises naturally from the structure of intuitionistic logic, and as such finds instances in a wide range of type systems previously described in literature. Despite this generality, this structure has a rich metatheory, which we expose.

CCS Concepts: • **Theory of computation → Type theory**; **Type structures**; *Program verification*; Operational semantics.

Additional Key Words and Phrases: linear types, modal logic, subtyping.

ACM Reference Format:
Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 5, ICFP, Article 1 (September 2020), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In logic, *modalities* are qualifiers that apply to statements. In the sentence "it *may* rain today", "may" is a modality which qualifies "it rains today". Modalities constitute a fruitful topic of research in logic, and, through the Curry-Howard correspondence, in programming language theory as well, where modalities qualify *types*. Girard [1987] famously proposed to decompose the function type constructor into a *linear* function type constructor and an *exponential* modality named "!". Beside this notorious example, modalities have found plenty of varied applications, including in privacy [Reed and Pierce 2010] and distributed computing [Murphy et al. 2005].

In this paper, we propose to unify the treatment of a broad range of modalities. We do so by defining a generic structure of modalities (Section 2), which finds instances in a wide range of systems (surveyed in Section 4). By framing a range of systems as instances of the same framework, the similarities and differences between them appear more clearly. We go further, and observe that the modality structure arises naturally from the structure of (higher-order) intuitionistic logic, or, equivalently, lambda calculus. More precisely, the operations on modalities reflect the way contexts are combined in the typing rules (Section 3), and the modality laws are dictated by the need to respect cut-elimination (or substitution, see Section 5).

In addition to the substitution lemma (Theorem 5.2), we then develop the meta-theory for the predicative polymorphic lambda calculus with modalities (hereafter called Λ^p). We provide a modality-respecting abstract machine (Section 6), and a parametric relational semantics (Section 7). We instantiate this semantics in Section 8 to show "free theorems" for some terms and types of Λ^p.

Our aim is to provide a high-utility framework while restricting the structure of modalities as little as possible. This way, we hope that Λ^p can be used as a basic framework for future work on

Authors' addresses: Andreas Abel, Department of Computer Science and Engineering, Gothenburg University, Rännvägen 6b, Göteborg, 41296, Sweden, andreas.abel@gu.se; Jean-Philippe Bernardy, Department of Philosophy, Linguistics and and Theory of Science, Gothenburg University, Box 100, Göteborg, SE-405 30, Sweden, jean-philippe.bernardy@gu.se.

2475-1421/2020/9-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

modalities in programming language theory and logic. We additionally hope that our work will inform the inclusion of modality structures in languages with types and proof assistants.

Note: The supplementary material contains a version of this paper extended with proofs.

2 THE RINGOID OF MODALITIES

The modality structure is a ring-like (ringoid) structure, which parameterises our calculus Λ^{p}.

Definition 2.1. A modality ringoid is a 6-tuple consisting of a set M, three binary operations: addition (+), multiplication (·) and meet (∧); and two elements zero (0) and unit (1); with the following structure:

- $(M, +, 0)$ forms a commutative monoid: addition is associative and commutative, with 0 as identity element.
- $(M, \cdot, 1)$ forms a monoid: multiplication is associative, with 1 as identity element.
- Multiplication distributes over addition: $p(q + r) = pq + pr$ and $(p + q)r = pr + qr$.
- 0 is an absorbing element for multiplication: $p \cdot 0 = 0 \cdot p = 0$.
- $(M, \wedge)$ forms a semilattice: meet is associative, commutative and idempotent.
- Multiplication distributes over meet (like for addition).
- Addition distributes over meet: $(p \wedge q) + r = (p + r) \wedge (q + r)$.

We do *not* rule out $0 = 1$; a one-element set is a ringoid with a trivial structure. An interesting special case are lattices M where addition coincides with meet and multiplication is join (∨); then, 0 is absorbing for ∧ and serves as top element (Section 4.3). Yet in general, neither 0 nor 1 need to be bounds of the semilattice.

We will detail the use of operations together with the typing rules, but it is easy to form an intuitive understanding right away. Addition is used to combine modalities of two components of a term which are both run, for example the function and the argument of an application. Its unit (0) correspond to no usage. Multiplication arises from function composition and is the principal means to combine modalities (qualifying a single type with several modalities); its unit (1) corresponds to the identity function, or, more generally, a single (non-qualified) use, like a plain variable occurrence. The meet can be used to match modalities between the branches of a conditional, via weakening. Commutativity of addition means that we do not have an ordered logic [Lambek 1958; Polakow and Pfenning 1999]. In general, the laws are those necessary to ensure preservation of modalities under evaluation (Theorems 5.2 and 6.2).

Definition 2.2. $(p \leqslant q) \stackrel{\text{def}}{=} (p = p \wedge q)$. This is the standard partial order arising from semi-lattices.

Note that addition and multiplication are monotone with respect to $(\leqslant)$, as a consequence of the corresponding distributivity. Theorem 3.1 shows that this order entails convertibility: if $p \leqslant q$ then p is less specific than q.

Modality expressions are formed from modality variables (ranged over by m), modality constants (elements of M), and formal sums, products, and meets. We overload the metasyntactic variables p, q and r to also range over modality expressions.

Definition 2.3. A *modality context* or usage map is defined as a map from variable names to modality expressions. When writing a modality context, we typically omit variables mapped to zero, and we may write 0 for the constant mapping $x \mapsto 0$. Usage maps are ranged by the metasyntactic variables γ, δ and ζ. Such contexts are used qualify a whole *typing* context.

We lift addition, meet and scaling by q to act pointwise on modality contexts: $(\gamma + \delta)(x) = \gamma(x) + \delta(x)$ and $(\gamma \wedge \delta)(x) = \gamma(x) \wedge \delta(x)$ and $(q \cdot \gamma)(x) = q \cdot \gamma(x)$. Modality contexts form a *left* *module* [McBride 2016] to ringoid M in the sense that scaling satisfies the following laws:

$$\begin{array}{lll} 1 \cdot \gamma = \gamma & (p \cdot q) \cdot \gamma = p \cdot (q \cdot \gamma) & p \cdot 0 = 0 \\ 0 \cdot \gamma = 0 & (p + q) \cdot \gamma = p \cdot \gamma + q \cdot \gamma & p \cdot (\gamma + \delta) = p \cdot \gamma + p \cdot \delta \\ & (p \wedge q) \cdot \gamma = p \cdot \gamma \wedge q \cdot \gamma & p \cdot (\gamma \wedge \delta) = p \cdot \gamma \wedge p \cdot \delta \end{array}$$

Modules are generalisations of vector spaces where the scalars q come from rings rather than fields; and *left* indicates that scaling is written as multiplication from the left.

3 PREDICATIVE POLYMORPHIC LAMBDA CALCULUS WITH MODALITIES

In this section we introduce our main object of study, a core functional programming language named Λ^{p} with predicative polymorphism $\forall\alpha.B$, modal function types ${}^{p}A \to B$, modal boxing $p\langle A\rangle$, and modality polymorphism $\forall m.B$. The language is chosen to be as simple as possible while being sufficient to illustrate how modalities work, and serves as a vehicle to illustrate applications in Section 4. In particular, Λ^{p} is lacking recursion on type and term level, but allows us to represent some inductive data types via the usual Church encoding. It is total and strongly normalising. This has two benefits: it simplifies the discourse—admitting a standard set-theoretic interpretation—and it means that the system can be used as a consistent logic.

Types $A, B, C \in \mathsf{Ty}$ are given by the following grammar. Herein, modality expressions p are formed over a fixed modality ringoid Mod that should be considered a parameter of the language.

$$A, B, C ::= K \mid 1 \mid \alpha \mid \forall\alpha.A \mid \forall m.A \mid {}^{p}A \to B \mid A + B \mid A \times B \mid p\langle A\rangle$$

Let Ty_0 denote the set of *monomorphic types*, for short *monotypes*. These are types that are free of polymorphism, i. e., contain no sub-expression of the form $\forall\alpha.B$. Restricting type variables α to stand for monotypes makes Λ^{p} *predicative*; in particular, the instantiation order $B[A/\alpha] < \forall\alpha.B$ is well-founded (where A monotype). A measure certifying well-foundedness is the lexicographic product of first, the number of type quantifiers $\forall\alpha$ and second, the size of the type expression. Well-foundedness of the instantiation order facilitates a direct set-theoretic interpretation of type quantification as an infinite product indexed by the monotypes.

Let further Ty_0^0 denote the set of *closed monotypes*, i. e., types that neither contain type quantification nor type variables. The letter K ranges over a set TyConst of uninterpreted base types; these monotypes will be used in the semantics to freely interpret type variables beyond the monotypes formed from $1, +, \times, \to$ and $p\langle\_\rangle$, which have a fixed meaning. For holding specific data, the base types K are unusable for lack of constructors and operations, however, we can define some data types from $1, +$, and $\times$, and even function space and polymorphism (Church encodings). For instance, the Boolean type is represented as $\mathsf{Bool} = 1 + 1$.

The domain of function types ${}^{p}A \to B$ is qualified with an arbitrary modality expression p. An omitted modality implicitly stands for 1, and thus we will see that $A \to B$ is often a linear function type, be we may still write $A \multimap B$ to emphasise linearity. Besides using modal function types, we can also qualify a type directly by applying a modality to it ($p\langle A\rangle$). It will become obvious from the typing rules that the types ${}^{p}A \to B$ and $p\langle A\rangle \to B$ are isomorphic. Regardless, we chose to include both ways to qualify types, for pedagogical purposes: they each have their advantages in this respect. In a language with generalised algebraic data types one would instead define $p\langle A\rangle$ as a data type with a constructor of type ${}^{p}A \to p\langle A\rangle$. For a monotype A, the Church encoding $\forall\alpha.({}^{p}A \to \alpha) \multimap \alpha$ is also isomorphic to $p\langle A\rangle$, a fact that can be demonstrated using parametricity (see Section 8.1).

Terms and typing. The terms of the language offer a couple of notable points. First, the eliminator of pairs is a let, binding the components to variables. For some modalities the projections fst or

$$\frac{}{0\Gamma, x : {}^{1}A \vdash x : A}\,\textsc{var} \qquad \frac{\delta\Gamma \vdash t : A \qquad \gamma \leqslant \delta}{\gamma\Gamma \vdash t : A}\,\textsc{wk} \qquad \frac{\gamma\Gamma, x : {}^{q}A \vdash t : B}{\gamma\Gamma \vdash \lambda^{q} x.t : {}^{q}A \to B}\,\textsc{abs}$$

$$\frac{\gamma\Gamma \vdash t : {}^{q}A \to B \qquad \delta\Gamma \vdash u : A}{(\gamma + q\delta)\Gamma \vdash t\,{}^{q}u : B}\,\textsc{app} \qquad \frac{\gamma(\Gamma, \alpha) \vdash t : B}{\gamma\Gamma \vdash \Lambda\alpha.t : \forall\alpha.B}\,\textsc{t-abs} \qquad \frac{\gamma\Gamma \vdash t : \forall\alpha.B \qquad \Gamma \vdash A}{\gamma\Gamma \vdash t \cdot A : B[A/\alpha]}\,\textsc{t-app}$$

$$\frac{\gamma(\Gamma, m) \vdash t : B}{\gamma\Gamma \vdash \Lambda m.t : \forall m.B}\,\textsc{m-abs} \qquad \frac{\gamma\Gamma \vdash t : \forall m.B \qquad \Gamma \vdash q}{\gamma\Gamma \vdash t \cdot q : B[q/m]}\,\textsc{m-app} \qquad \frac{}{0\Gamma \vdash () : 1}\,\textsc{1-intro}$$

$$\frac{\gamma\Gamma \vdash t : A_1 + A_2 \qquad \delta\Gamma, x_i : {}^{q}A_i \vdash u_i : C \qquad q \leqslant 1}{(q\gamma + \delta)\Gamma \vdash \mathsf{case}\ {}^{q}t\ \mathsf{of}\ \{\mathsf{inj}_1\, x_1 \mapsto u_1; \mathsf{inj}_2\, x_2 \mapsto u_2\} : C}\,+\textsc{-elim} \qquad \frac{\gamma\Gamma \vdash t : A_i}{\gamma\Gamma \vdash \mathsf{inj}_i\, t : A_1 + A_2}\,+\textsc{-intro}$$

$$\frac{\gamma\Gamma \vdash t : A \qquad \delta\Gamma \vdash u : B}{(\gamma + \delta)\Gamma \vdash (t, u) : A \times B}\,\times\textsc{-intro} \qquad \frac{\gamma\Gamma \vdash t : A \times B \qquad \delta\Gamma, x : {}^{q}A, y : {}^{q}B \vdash u : C}{(q\gamma + \delta)\Gamma \vdash \mathsf{let}\,(x, y) = {}^{q}t\ \mathsf{in}\ u : C}\,\times\textsc{-elim}$$

$$\frac{\gamma\Gamma \vdash t : A}{p\gamma\Gamma \vdash [{}^{p}t] : p\langle A\rangle}\,p\langle\cdot\rangle\textsc{-intro} \qquad \frac{\gamma\Gamma \vdash u : p\langle A\rangle \qquad \delta\Gamma, x : {}^{qp}A \vdash t : C}{(q\gamma + \delta)\Gamma \vdash \mathsf{let}\,[{}^{p}x] = {}^{q}u\ \mathsf{in}\ t : C}\,p\langle\cdot\rangle\textsc{-elim}$$

Fig. 1. Typing rules of Λ^{p}

snd are definable from let, but not always (see Section 8.2). Second, we allow eliminating qualified terms ${}^{q}t$—this is useful because it is not always possible to construct a term with an exact modality of 1. Omitted modality annotations default to 1.

$$\begin{array}{llr}
t, u ::= & x \mid \lambda^{q} x.t \mid t\,{}^{q}u & \text{variables; functions} \\
& \mid \Lambda\alpha.t \mid t \cdot A \mid \Lambda m.A \mid t \cdot q & \text{polymorphism} \\
& \mid \mathsf{inj}_1\, t \mid \mathsf{inj}_2\, t \mid \mathsf{case}\ {}^{q}t\ \mathsf{of}\ \{\mathsf{inj}_1\, x_1 \mapsto u_1; \mathsf{inj}_2\, x_2 \mapsto u_2\} & \text{sums} \\
& \mid () \mid (t, u) \mid \mathsf{let}\,(x, y) = {}^{q}t\ \mathsf{in}\ u & \text{tuples} \\
& \mid [{}^{p}t] \mid \mathsf{let}\,[{}^{p}x] = {}^{q}t\ \mathsf{in}\ u & \text{qualification}
\end{array}$$

As usual, let the Boolean constants be true $= \mathsf{inj}_1\,()$ and false $= \mathsf{inj}_2\,()$.

Contexts bind free variables of all sorts:

$$\Gamma, \Delta ::= [\,] \mid \Gamma, x{:}A \mid \Gamma, \alpha \mid \Gamma, m$$

The typing judgement has the form $\gamma\Gamma \vdash t : A$, meaning that t has type A (with an implicit modality 1) in context Γ, and t uses the variables $x : \Gamma(x)$ with modalities $\gamma(x)$. Let the notation $\gamma\Gamma, x : {}^{q}A$ stand for $(\gamma, qx)(\Gamma, x : A)$. We write $\Gamma \vdash q$ to mean that a modality expression q is well-formed in a context Γ. This means exactly that its free (modality) variables are all bound by Γ. Likewise, a monotype A whose free (type and modality) variables are bound by Γ is noted $\Gamma \vdash A$. The typing rules are shown in Fig. 1. Some comments:

First, a variable occurrence always corresponds to usage 1. This ensures stability of usage under variable substitution. Other rules ensure that the modalities of introduction and elimination match. For example, abstraction introduces a variables with modality q and application multiplies the usage of the argument by q.

Second, we allow usage weakening: one can always use a variable in a more specific modality than the one which is available. In fact, we always have convertibility in this direction.

Theorem 3.1 (Convertibility). *If $p \leqslant q$, then there is a term of type ${}^{p}A \to q\langle A\rangle$ for any A.*

The proof can be found in the long version of the paper, provided as supplementary material. Hereafter this will be noted with pictograms. (⇨📄)

Third, the existence of meet ensures compositionality for case branches. That is, if we have branches with differing usages ($\delta_i\Gamma, x : {}^{q}A_i \vdash u_i : C$), then we can always find a single modality context $\delta = \bigwedge_i \delta_i$ such that $\delta \leqslant \delta_i$ for every i, and combine the branches using weakening.

Besides, we require the scrutinee of case analysis to be available with modality 1, or more relaxed. This constraint captures the fact that case analysis observes information contained in the scrutinee, namely whether we have inj_1 or inj_2, and at the same time we want irrelevance Theorem 7.10 to be a property of our system. We further discuss this issue in Section 10.

Additionally, we imbue our system with the ability to (universally) quantify over modalities. Such universally quantified variables may then be constrained by adding convertibility assumptions. Any equality constraint $p \leqslant q$ can be expressed instead using the type $\forall\alpha.{}^{p}\alpha \to q\langle\alpha\rangle$, and the equality $p = q$ is equivalent to the two inequalities $p \leqslant q$ and $q \leqslant p$. This means that a user of this system is able to apply the general structure to special cases; some of which we present in Section 4.

4 APPLICATIONS

In this section we survey several systems featuring modalities, and show how they are instances of ours (or sometimes what the difference is). By doing so we illustrate various ways to specialise the modality ringoid structure. We do not aim for exhaustivity, but rather at showing how varied applications can be.

As a prelude, we remark that if modalities are ignored (for example by letting all modalities be equal to 1), then Λ^{p} degenerates to the usual polymorphic lambda calculus (with sum and products).

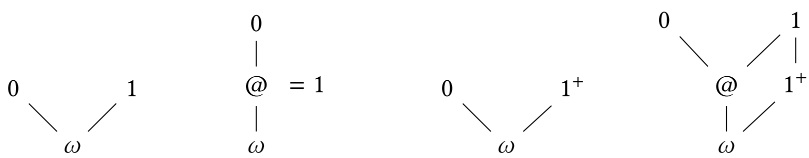
4.1 Substructural Type Systems

Λ^{p} provides a uniform calculus for substructural typing (see for example Walker [2005] for an introduction to substructural typing).

4.1.1 Linear types. The first obvious application of our system is linearity. Indeed, the unit modality precisely corresponds to linear usages. In our system, a 0-qualified function is necessarily constant, and so contrary to linear logic this modality is always supported specially. To conveniently support all other non-linear usages, one can add single a modality for unrestricted usages, which we note here ω instead of the traditional exclamation mark for typographical reasons. We have $\omega \leqslant 1$, meaning that if we have any number of allowed usages, we also have in particular one usage allowed. When specialised this way, Λ^{p} becomes nearly equivalent to the core language of Linear Haskell [Bernardy et al. 2018] — with the addition of support for 0.

Most of the operations are fixed by the algebraic restrictions, but one can refer to Table 1 in case of doubt. Instead of a table, we use a Hasse diagram to represent the meet (Fig. 2). Checking the laws is routine, and thus we omit the proofs here (and in the rest of the section).

4.1.2 Affine types. If one so wishes, the above system can be refined to support affine types by adding a modality @ corresponding to either 0 or 1 usages. This time, @ can play the role of 1. The semilattice is changed as in Fig. 2.



(a) Linear (b) Affine (c) Relevant (d) Combined

Fig. 2. Hasse diagrams for various substructural type system lattices. The modality @ corresponds to 0 or 1 uses, 1^+ corresponds to 1 use or more, ω corresponds to any number of uses.

$(+)$	0	ω	$@$	1	1^+
0	0	ω	$@$	1	1^+
ω	ω	ω	ω	1^+	1^+
$@$	$@$	ω	ω	1^+	1^+
1	1	1^+	1^+	1^+	1^+
1^+	1^+	1^+	1^+	1^+	1^+

$(\cdot)$	0	ω	$@$	1	1^+
0	0	0	0	0	0
ω	0	ω	ω	ω	ω
$@$	0	ω	$@$	$@$	ω
1	0	ω	$@$	1	1^+
1^+	0	ω	ω	1^+	1^+

Table 1. Addition and multiplication rules for usual substructural modalities

4.1.3 Relevant types. Dually we can instead let the unit modality represent "at least one usage", capturing relevant type systems. If write $1^+$to minimise confusions, the characteristic equation of this system is $1^+ + 1^+ = 1^+$.

4.1.4 Combined system. Another useful setup is one where the modalities zero, linear, affine, relevant, and unrestricted are all present (and different). In such a situation the system will keep track of all cases simultaneously, and the operation tables are more involved (Table 1).

4.1.5 Quantitative typing. A generalisation of all the above systems is what can be called quantitative typing, where one has a modality for each set of accepted usage. That is, the set of modalities Mod is the powerset of natural numbers, with $0 = \{0\}$, $1 = \{1\}$ and the following operations:

$$
\begin{aligned}
p \wedge q &= p \cup q \\
p + q &= \{x + y \mid x \in p, y \in q\} \\
p \cdot q &= \{x \cdot y \mid x \in p, y \in q\}
\end{aligned}
$$

This is the most precise substructural instance, tracking exactly which set of usages are acceptable. It is a useful theoretical device, however, even in their simplest form modality expressions for this structure can be large, and thus it is often preferable not to track usages so precisely.

In all the above cases ω (even under its other name $\mathbb{N}$) is the extremum of the meet-lattice. Variables associated with this modality can be used in unrestricted fashion. Conversely, to produce a term to substitute in an ω-variable, one can only use ω-variables.

4.2 Sensitivity Analysis for Differential Privacy

Another application of affine-like type systems is differential privacy, where one is interested in publishing statistically anonymised data without revealing individual secrets. Here, the role of the type system is to ensure that if a certain amount of noise is introduced in the inputs of a program, then at least the same amount is present in the outputs.

For this purpose, Reed and Pierce [2010] equip every type A with a *metric* $d_A : A \times A \longrightarrow \mathbb{R}^{\infty}_{\geqslant 0}$, where $\mathbb{R}^{\infty}_{\geqslant 0}$ shall denote the set of non-negative reals augmented with positive infinity.

Then a function f from A to B is defined to be c-sensitive if it does not increase distances by a factor greater than c; as defined by the metrics for A and B: $d_B(f(x), f(y)) \leqslant_{\mathbb{R}} c \cdot d_A(x, y)$. Consequently, under the assumption that distance 0 means equality at all types, 0-sensitive functions are necessarily constant. Conversely ∞-sensitive functions impose no restriction on the argument. Because of the inequality, c-sensitivity is subject to subsumption: if $c' \geqslant_{\mathbb{R}} c$ and f is c-sensitive then f is also c'-sensitive.

We can cast this system into our framework by letting the modality carrier set be $\mathbb{R}^{\infty}_{\geqslant 0}$, with the usual arithmetic operations and the meet be the *maximum* of its arguments— which implies that the order on modalities is the opposite of the usual order on $\mathbb{R}$: $(\leqslant) = (\geqslant_{\mathbb{R}})$. It is easy to check that the obtained system is equivalent to that of Reed and Pierce, with exceptions detailed below.

Reed and Pierce then proceed to define a sensitivity-aware type system, and metrics for every type. With this in place they show that evaluation preserve sensitivity, and they do this by using a special-purpose step-indexed metric logical relation.

But with our general setting, we do not have to do any special preservation proof: we already (Theorem 6.2) know that the system is type-preserving for any modality ringoid, and thus any assignment of types to metrics will do for this purpose. All we need to do is to ensure that primitive functions are metric-respecting on the types that they mention. For example, assuming the usual arithmetic meaning, and the absolute value as metric for reals (Real), real addition can be typed with Real $\multimap$ Real $\multimap$ Real and multiplication by a positive constant k with ^{k}Real $\rightarrow$ Real.

The instance our general framework described above departs from the Reed-Pierce system in one respect: Reed and Pierce allow case analysis on any modality (sensitivity) $r \geqslant_{\mathbb{R}} 0$, whereas we demand $r \leqslant 1$. In consequence, they additionally sustain a function $f : {}^{r}(A + B) \longrightarrow (r\langle A\rangle + r\langle B\rangle)$, for every non-zero r, and in particular ^{r}Bool $\longrightarrow$ Bool. This apparently means that metrics are not preserved in their system, but as we see it, they save the day by defining the metric on sum types to be infinite when the tags differ. Thus, we can use the same metric for sum types and safely add f as a primitive function, recovering the equivalence between the systems. Regardless, it unclear that this metric on sum types is useful. For example, the later system of Gaboardi et al. [2013], concerned particularly on the relation between a linear type system and differential privacy, features no (dynamic) sum type—the lengths of lists are tracked statically.

Shall we say that a reviewer disagrees?

4.3 Informational applications

In this section, we describe applications which we group under the loose term "informational", in the sense that it does not matter how many times variables are used, but rather *in which context* they are used. Technically, the addition is relegated to play the same role as the meet $((+) = (\wedge))$. We also constrain the multiplication so that it acts as the join (dual to the meet) of the lattice. This means that multiplication must be idempotent ($a \cdot a = a$), and absorption laws must be respected:

$$a \cdot (a \wedge b) = a \tag{1}$$

$$a \wedge (a \cdot b) = a \tag{2}$$

(In fact, (1) is a consequence of (2) and the other laws.) We illustrate these properties on several examples below. In the rest of this section we may write $(\vee)$ in place of $(\cdot)$ to emphasise the lattice duality of operations.

4.3.1 Irrelevance. Irrespective of any additional modality structure, 0 represents no usage of a variable; and thus, if $\gamma(x) = 0$ and $\gamma\Gamma \vdash t : A$ then t cannot use x.

It is however useful to analyse the role of 0 in the informational setting. Here, 0 is also the unit of $(\wedge)$, and as such the top of the lattice: $p \leqslant 0$ for every p. Consequently, we have the following derivation, chaining $0\langle\cdot\rangle$-INTRO and weakening with $\delta \leqslant 0$:

$$\dfrac{\dfrac{\gamma\Gamma \vdash t : A}{0\Gamma \vdash [^0 t] : 0\langle A\rangle}\ 0\langle\cdot\rangle\text{-INTRO}}{\delta\Gamma \vdash [^0 t] : 0\langle A\rangle}\ \text{WK}$$

It says that if tasked to construct $0\langle A\rangle$ in any usage context δ it suffices to construct A for any (other) usage γ. Even if $\gamma(x) = 0$, we can choose $\delta(x)$ to be any modality we like. Borrowing the striking metaphor of Pfenning [2001], the variables of $\gamma\Gamma$ are *resurrected* inside the 0-box. In fact in this system the modality 0 represents irrelevance, in the sense of Pfenning [2001]. The key property of the system (equality ignores irrelevant arguments) is captured by Theorem 7.10.

4.3.2 Information-flow security. One application of type systems is to ensure that certain parts of a program do not have access to private (high security) information. Several type systems have been proposed to explicitly support this feature, notably the seminal work of Abadi et al. [1999].

The principal property of such systems is that the output of a program does not depend on secret inputs. This a property holds for Λ^{p} (Theorem 7.10), if we consider that any modality p above 1 in the lattice is secret. The simplest security lattice has a single secret level H (high) which can be represented by 0 and a single public level L (low) represented by 1. The construction generalises however to any lattice of informational modalities as specified above: no further specialisation is required nor desirable.

We can convince ourselves intuitively that addition should coincide with the meet: if we need a variable in two parts of a term, we must assume the worst and require the most public level, given by the meet. Dually, if a function t offers a at least a level of secrecy p for its parameter, and constructing its argument u offers a level of secrecy of at least q for a given variable x, then the whole application offers the maximum level of secrecy $p \vee q$ for x.

$$\dfrac{\vdash t : {}^{p}A \to B \qquad x : {}^{q}X \vdash u : A}{x : {}^{p\vee q}X \vdash t\,{}^{p}u : B}\quad \text{Example application}$$

Generalising to arbitrary contexts, we obtain exactly the generic application rule with $(\cdot) = (\vee)$ and $(+) = (\wedge)$. Contrary to the convention of much literature on information-flow security, including Abadi et al. [1999], our security levels are *relative* to the level of the program under current execution, which works at level 1. Indeed, the variable x above appears to become more public when constructing u. As with irrelevance before, an inaccessible variable may become accessible again in a secret context.

We are not aware of a security type system which corresponds exactly to our the informational instance of our framework, but some are very close [Algehed 2018]. Regardless, as further witness of the capability of the system to support security applications, and inspired by Algehed et al. [2019], we give an implementation of a chat server which serves as the prototype of a system which is communicating with many agents operating at different security levels. Whether agents can communicate is provided by a policy, which essentially takes the form of a decidable partial order corresponding to the security lattice. In this example we use a Haskell-like syntax and also assume that the language is extended with usual features such as data types.

We use the *CanFlow c c′* type, to capture that $c \leqslant c'$. This is done by giving the corresponding (polymorphic) conversion function:

type $\mathit{CanFlow}\ c\ c' = \forall\alpha.c\langle\alpha\rangle \to c'\langle\alpha\rangle$

We have a number of *Client*s sending messages to *Chan*nels, which they can also connect to. Every connected client receives the messages sent this way. Clients and channel types are indexed by the modality corresponding to their security level.

```
data Chan   (c :: M)
data Client (c :: M)
```

The security policy is represented by the following three functions, which are parameters of the program. They essentially act as functions testing the modality order, but they operate on the *Client* and *Chan* types and are given suggestive names.

```
canRead   :: Client c → Chan c′ → Maybe (CanFlow c′ c)
canWrite  :: Client c → Chan c′ → Maybe (CanFlow c c′)
testEqual :: Chan   c → Chan c′ → Maybe (CanFlow c c′)
```

Messages are secure pieces of information, and as such are annotated with the corresponding level c. Their type is thus $c\langle String\rangle$. A client can only be sent messages at the correct level, which is represented by the next (and last) parameter to the program:

```
clientWrite :: Client c → c⟨String⟩ → IO ()
```

The body of the server can then be implemented given the above primitives. A subscription of a given channel by a given client is represented by the following data, witnessing the level compatibilities:

```
data Subscription where
  Subscribed :: Chan c → Client c′ → CanFlow c c′ → Subscription
```

The server handles two kind of events: subscription and sending a message. At this stage the compatibility between levels is not guaranteed; it is the task of the server to do so.

```
data Event where
  SubscribeEvent :: Client c → Chan c′ → Event
  WriteEvent     :: Client c → Chan c′ → c⟨String⟩ → Event
```

The server maintains a list of *Subscription*s. Its main job is to test level compatibilities and act accordingly:

```
mainStep :: [Subscription] → IO [Subscription]
mainStep cs = do
  ev ← readEvent
  case ev of (SubscribeEvent client chan) → case canRead client chan of
                 Nothing   → return cs    -- request declined
                 (Just ok) → return (Subscribed chan client ok : cs)
             (WriteEvent client chan msg) → case canWrite client chan of
                 Nothing   → return cs    -- request declined
                 (Just f1) → do forM cs
                                  λ(Subscribed ch rcvClient f2) → case testEqual chan ch of
                                     Nothing   → return ()   -- not a matching channel
                                     (Just f3) → clientWrite rcvClient ((f2 ∘ f3 ∘ f1) msg)
                                  return cs
```

Supporting security features via generic abstraction features of type systems have been proposed before, but so far this has been done via quantification over types [Bowman and Ahmed 2015; Tse and Zdancewic 2004]. It has additionally been shown that non-interference is a consequence of the generic parametricity of type-theory [Algehed and Bernardy 2019].

However, modalities are in much direct correspondence to the security levels found in the information-flow security literature [Abadi et al. 1999], and thus we believe that this is a natural application of generic modal type system.

4.3.3 Necessity and possibility. The necessity modality $\Box$ can be captured in STLC by adding the following rules:

$$\frac{\Box\Gamma \vdash t : A}{\Box\Gamma \vdash t : \Box A}\Box\text{-Intro} \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t : A}\Box\text{-Elim}$$

The elimination rule says that if A holds necessarily, it holds. This corresponds to the conversion of $\Box A$ to $1A$, and in turn it follows from the lattice containing the relation $\Box \leqslant 1$. Hence $\Box$ acts like an "categorically true" modality. According to the introduction rule, to hold *necessarily* ($\Box A$), A must hold under only necessary assumptions (no non-necessary assumptions are allowed). Recall that our introduction rule for $\Box\langle A\rangle$ is:

$$\frac{\Gamma \vdash t : A}{\Box\Gamma \vdash [^{\Box}t] : \Box\langle A\rangle}$$

which is a strengthening of the meaning of $\Box$, because we forget that assumptions are *necessary* in the premise, and thus, *a priori* fewer terms can be shown to inhabit $\Box A$ by using our rule. However, according to our assumptions we also have $\Box \cdot \Box = \Box$, and thus we can derive:

$$\frac{\dfrac{\Box\Gamma \vdash t : A}{\Box\Box\Gamma \vdash [^{\Box}t] : \Box\langle A\rangle}}{\Box\Gamma \vdash [^{\Box}t] : \Box\langle A\rangle}$$

This shows the admissibility of the introduction rule. Additionally the law $\Box \cdot \Box = \Box$ makes the type $^{\Box}A \longrightarrow (\Box \cdot \Box)\langle A\rangle$ inhabited— an often desired property of necessity in the literature. Classically, possibility ($\Diamond$) is the De Morgan dual of necessity ($\neg\Box A \leftrightarrow \Diamond\neg A$), however this does not work in intuitionistic logic. Thus, a better option may be a specific modality $\Diamond$ occupying a dual position in the lattice wrt. to $\Box$; starting from the calculus of Pfenning and Davies [2001].

4.3.4 Beliefs. Logical systems are sometimes used to describe the beliefs of various agents. Such systems can be rather intricate, and we do not claim that our modality framework can capture all the intricacies previously studied in the literature. Yet we can note that one area of application is the ability to model agents with inconsistent beliefs, while retaining the overall consistency of the system. We recall that information can travel in the ($\leqslant$) direction, and thus we have $^{p}A \longrightarrow {}^{q}B \longrightarrow (p \vee q)\langle A \wedge B\rangle$. In consequence the beliefs at level p may contradict those at level q, but only agents at level $p \vee q$ or above will consider this contradiction as their own belief. In particular both the p and q levels can remain locally consistent.

4.3.5 Distributed computing. Another application is to use modalities to represent the location of code. This idea was proposed by Murphy et al. [2005], and can be imported in our framework. The system of Murphy et al. is syntactically far from ours. While we have a type $p\langle A\rangle$ to represent truth of A at location p, they use a different judgement altogether. Thus in this respect our system is more general. Additionally, while our system is intuitionistic, theirs is classical (featuring first-class continuations). Yet, the idea that modalities can represent a (set of) computers is an available interpretation for our system. Additionally one can also use the logical aspect of the system, to reason about what is true at different locations.

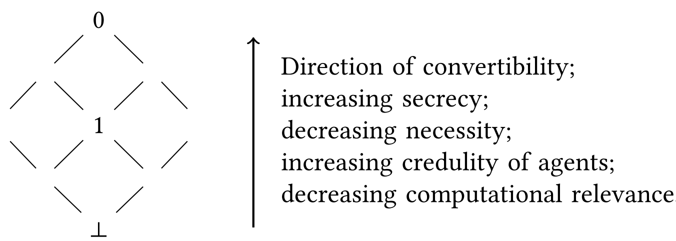


Fig. 3. Hasse diagram for informational modalities. The (partial) ordering can be interpreted in various ways depending on the application.

4.3.6 Summary of informational aspect. All the above aspects can be conveniently combined in a single lattice, as shown in Fig. 3. The 1 modality represents the point of view of the program. Modalities above it correspond to (partially) inaccessible information due to secrecy, possibility and partial irrelevance. Modalities below it correspond to (excessively) public information and (partial) necessity. Unrelated modalities correspond to independent agents, with whom no communication of data (or proofs of a proposition) is possible. The various interpretations (secrecy, necessity, etc.) can be made depending on the application.

4.4 Combining informational and quantitative aspects

Having a single system supporting all possible applications yields the usual benefit of reuse: the generic applications can be coded in generic contexts and applied in several. A somewhat more subtle benefit is that one can combine several applications in a single program: for example one can have an system which combines aspects of differential privacy and information-flow secrecy[1] (by, say, having several dimensions of differential privacy, themselves organised in a lattice). This can be done using a product of modalities, as Orchard et al. [2019] suggests. This would mean that informational and quantitative aspects are both checked, but separately; i. e., when counting occurrences, convertibility is ignored and *vice versa*.

However, it is also possible to construct a more fine-grained ringoid, with modalities capturing situations such as “one public usage or three private ones”. We can model this using a set of generators of for secrecy (capabilities), and counting how we can use those.

This modality ringoid can be built in two stages. First, we build the structure of exact numbers of usage at given security levels, L. This number acts as a generalisation of $\mathbb{N}$ in the initial quantitative structure of Section 4.1.5. Assuming a lattice K of capabilities/security levels as in the informational examples, we let $L = \mathit{MultiSet}(K)$ and

$$\begin{array}{ll} 0_L = \emptyset & l_1 +_L l_2 = l_1 \uplus l_2 \\ 1_L = \{\!\{1\}\!\} & l_1 \cdot_L l_2 = \{k_1 \vee k_2 \mid k_1 \in l_1, k_2 \in l_2\} \end{array}$$

We inductively define a partial order $\leqslant_L$ on L capturing that any single usage can be relaxed using the underlying order on K:

$$\frac{}{\emptyset \leqslant_L \emptyset} \qquad \frac{k_1 \leqslant_K k_2}{\{\!\{k_1\}\!\} \leqslant_K \{\!\{k_2\}\!\}} \qquad \frac{l_1 \leqslant_L l_2 \quad m_1 \leqslant_L m_2}{(l_1 \uplus m_1) \leqslant_L (l_2 \uplus m_2)}$$

Finally, a modality p is a $\leqslant_L$-downward closed subset of L; each $m \in p$ presents one alternative of exact capabilities m to assign to a variable. Formally, the modality ringoid of possible numbers

[1] broadly similar to the system of Ebadi et al. [2015]

of usage $M = \{p \subseteq L \mid (l \leqslant_L l' \wedge l' \in p) \to l \in p\}$ is the powerset of L, quotiented by $(\leqslant_L)$-closure: if m is allowable and l is less restrictive, then l is also allowable. The operations are defined as in Section 4.1.5:

$$\begin{array}{ll} & p \wedge_M q = p \cup q \\ 0_M = \{0_L\} & p +_M q = \{l +_L l' \mid l \in p, l' \in q\} \\ 1_M = \{1_L\} & p \cdot_M q = \{l \cdot_L l' \mid l \in p, l' \in q\} \end{array}$$

5 SUBSTITUTION LEMMA

In the theory of lambda calculi, subject reduction states that term reductions (such as β reduction) preserve types. Subject reduction rests on the substitution lemma, which states that types are preserved under substitution. We will prove type preservation in the setting of an abstract machine (Theorem 6.2), but we are particularly interested in the substitution lemma, because it is the simplest setting which shows why the modalities need to have the structure shown in Definition 2.1.

There are three substitutions in Λ^{p}: one for modality expressions $[p/m]$, one for types $[A/\alpha]$ and one for terms $[u/x]$. The first two are straightforward and standard, and in the rest of the section we consider only substitution on terms. Traditionally, a parallel substitution σ is a map from a context Γ to a context Δ. There is one term $\sigma(x)$ for each variable x in Δ, each of them typeable in Γ, formally $\Gamma \vdash \sigma(x) : \Delta(x)$. This can be written in compact form as $\Gamma \vdash \sigma : \Delta$. Then the substitution lemma states that applying the substitution changes the typing of a term from a context Δ to a context Γ:

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash t : A}{\Gamma \vdash t[\sigma] : A}$$

In the rest of the section we show how substitutions and the substitution lemma extend to qualified contexts. Each of the terms $\sigma(x)$ is typed in a different modality context, $\Psi(x)$, formally $\Psi(x)\Gamma \vdash \sigma(x) : \Delta(x)$. In compact form, we can write the type of a substitution $\Psi\Gamma \vdash \sigma : \Delta$. It is interesting to observe that Ψ is a map of variables (in Δ) to modality contexts (for Γ). That is, we have a matrix of modalities, whose indices are variables in Γ and Δ.

When applying substitution, every occurrence of a variable x in Δ is replaced by $\sigma(x)$, and thus a usage of $1x$ is replaced by $\Psi(x)$. Therefore the modality for a variable y in $t[\sigma]$ is $(\sum_{x \in \Delta} \delta(x)\Psi(x, y))$. We see that Ψ acts linearly on δ, and thus in the following we treat Ψ as a linear operator on modality contexts [Atkey and Wood 2019]. We write $\Psi : \Delta \multimap \Gamma$ to reflect this fact, and write $\delta\Psi$ for application to δ. The postfix notation witnesses that substitution acts on the right of modalities.

Lemma 5.1. *Operator application is (1) associative with modality multiplication, $(q\gamma)\Psi = q(\gamma\Psi)$, and (2) it distributes over context addition ($(\gamma + \delta)\Psi = \gamma\Psi + \delta\Psi$) and (3) meet ($(\gamma \wedge \delta)\Psi = \gamma\Psi \wedge \delta\Psi$).*

Proof. (1) rests on associativity of $(\cdot)$ and distributivity of $(\cdot)$ over $(+)$. (2) additionally relies on $(+)$ being associative and commutative; likewise for (3) *mutatis mutandis.* □

We are now ready to state and prove our result:

Theorem 5.2 (Substitution lemma). *Given a modality operator $\Psi : \Delta \multimap \Gamma$, a substitution $\Psi\Gamma \vdash \sigma : \Delta$ and a typed term $\delta\Delta \vdash t : A$ then $(\delta\Psi)\Gamma \vdash t[\sigma] : A$.*

6 ABSTRACT MACHINE

In this section we construct an abstract call-by-name machine for Λ^{p}. The main purpose of the machine is to show that modalities are preserved under execution. Machine states will be presented in the form $\gamma h \Vdash^r t \cdot \vec{e}$ where h is a heap with modality context γ, a head t, and a stack of eliminations $\vec{e}$ and a corresponding stack of modalities r. Each entry e is a function argument ${}^q u$ or an eliminator whose scrutinee is replaced by a hole "?", e. g., let $(x, y) = {}^q ?$ in u. Thus, $t \cdot \vec{e}$ is a spine representation

$$
\begin{array}{rlcrl}
(\gamma + rx)h \Vdash^{r} & x \cdot \vec{e} & \longrightarrow & \gamma h \Vdash^{r} & h(x) \cdot \vec{e} \\
\gamma h \Vdash^{r} & (t\,{}^{q}u) \cdot \vec{e} & \longrightarrow & \gamma h \Vdash^{r} & t \cdot {}^{q}u \cdot \vec{e} \\
(\gamma + rq|u|)h \Vdash^{r} & \lambda\,{}^{q}x.t \cdot {}^{q}u \cdot \vec{e} & \longrightarrow & \gamma h, x \mapsto rqu \Vdash^{r} & t \cdot \vec{e} \\
\gamma h \Vdash^{r} & \mathsf{let}\,[{}^{q}x] = {}^{q}t\,\mathsf{in}\,u \cdot \vec{e} & \longrightarrow & \gamma h \Vdash^{rq} & t \cdot \mathsf{let}\,[{}^{q}x] = {}^{q}?\,\mathsf{in}\,u \cdot \vec{e} \\
(\gamma + rq|v|)h \Vdash^{rq} & [{}^{q}v] \cdot \mathsf{let}\,[{}^{q}x] = {}^{q}?\,\mathsf{in}\,u \cdot \vec{e} & \longrightarrow & \gamma h, x \mapsto rqv \Vdash^{r} & u \cdot \vec{e} \\
\gamma h \Vdash^{r} & (t \cdot p) \cdot \vec{e} & \longrightarrow & \gamma h \Vdash^{r} & t \cdot (p \cdot \vec{e}) \\
\gamma h \Vdash^{r} & \Lambda m.t \cdot (p \cdot \vec{e}) & \longrightarrow & \gamma h \Vdash^{r} & t[p/m] \cdot \vec{e} \\
\gamma h \Vdash^{r} & (t \cdot A) \cdot \vec{e} & \longrightarrow & \gamma h \Vdash^{r} & t \cdot (A \cdot \vec{e}) \\
\gamma h \Vdash^{r} & \Lambda\alpha.t \cdot (A \cdot \vec{e}) & \longrightarrow & \gamma h \Vdash^{r} & t[A/\alpha] \cdot \vec{e}
\end{array}
$$

$$
\begin{array}{lrll}
 & \gamma h & \Vdash^{rq} & \mathsf{let}\,(x,y) = {}^{q}t\,\mathsf{in}\,u \cdot \vec{e} \\
\longrightarrow & \gamma h & \Vdash^{rq} & t \cdot \mathsf{let}\,(x,y) = {}^{q}?\,\mathsf{in}\,u \cdot \vec{e} \\[1ex]
 & (\gamma + rq(|u| + |t|))h & \Vdash^{rq} & (t,u) \cdot \mathsf{let}\,(x,y) = {}^{q}?\,\mathsf{in}\,v \cdot \vec{e} \\
\longrightarrow & \gamma h, x \mapsto rqt, y \mapsto rqu & \Vdash^{r} & v \cdot \vec{e} \\[1ex]
 & \gamma h & \Vdash^{rq} & \mathsf{case}\,{}^{q}t\,\mathsf{of}\,\{\mathsf{inj}_1\,x_1 \mapsto u_1; \mathsf{inj}_2\,x_2 \mapsto u_2\} \cdot \vec{e} \\
\longrightarrow & \gamma h & \Vdash^{rq} & t \cdot \mathsf{case}\,{}^{q}?\,\mathsf{of}\,\{\mathsf{inj}_1\,x_1 \mapsto u_1; \mathsf{inj}_2\,x_2 \mapsto u_2\} \cdot \vec{e} \\[1ex]
 & (\gamma + rq|v|)h & \Vdash^{rq} & (\mathsf{inj}_i\,v) \cdot \mathsf{case}\,{}^{q}?\,\mathsf{of}\,\{\mathsf{inj}_1\,x_1 \mapsto u_1; \mathsf{inj}_2\,x_2 \mapsto u_2\} \cdot \vec{e} \\
\longrightarrow & \gamma h, x_i \mapsto rqv & \Vdash^{r} & u_i \cdot \vec{e}
\end{array}
$$

Note: $|u|$ denotes the modality context of u, given by the typing judgement.

Fig. 4. Machine transitions.

of Λ^{p}-terms: a head t subsequently eliminated by the $\vec{e}$, with the first elimination, the top of the stack, applied first. We write $\vec{e}(t)$ for the thus reconstructed Λ^{p}-term. When t is in weak head normal form, it interacts with the first elimination, implementing a call-by-name weak head reduction that adds new bindings to the heap. Besides "reduction" steps, the machine performs administrative steps which decompose the head further into spine form, and dereferencing when the head is a variable. (See Fig. 4.)

The annotation r is a stack of modalities, obtained from the scrutinee qualifications ${}^{q}?$ of each let and case in $\vec{e}$, and as such is functionally dependent on $\vec{e}$. This stack may occur in modality expressions, and then it shall be interpreted as a product of its components. This product is the modality qualifying t. In sum, $\gamma h \Vdash^{r} t \cdot \vec{e}$ can be read as "γh provides what is needed to produce ${}^{r}t$, and continue with $\vec{e}$".

The states are well-typed, such that $\gamma\Gamma \vdash \vec{e}(t) : C$. Thus, strictly speaking, machine states also contain types and contexts. In fact $\gamma = \delta + r\zeta$, such that $\zeta\Gamma \vdash t : A$, and $\delta\Gamma$ is the context of $\vec{e}$. When we want to emphasise typing we write the machine state in the form $(h : \gamma\Gamma) \Vdash^{r} (t \cdot \vec{e} : C)$, however we generally leave types implicit to avoid bloat.

Remark 1. *An alternative would be to work with* intrinsically *typed terms [Allais et al. 2018; Benton et al. 2012]. This would mean there would not be a need to add explicit annotations for type and modality contexts. In this situation the* wk *rule would be represented explicitly as a term constructor.*

However, we introduced terms as extrinsically *typed, and thus, in fact, machine states manipulate typing derivations.*

Thereby, there is an embedding-projection relation between well-typed terms and machine states. The projection of an arbitrary state $(h : \gamma\Gamma) \Vdash^r (t \cdot \vec{e} : C)$ is $\gamma\Gamma \vdash \vec{e}(t) : C$. Conversely an arbitrary term $\gamma\Gamma \vdash t : C$ can be embedded into the machine state $(h : \gamma\Gamma) \Vdash^1 (t \cdot - : C)$ if a suitable heap h can be constructed. In particular, if Γ is empty, then the empty heap is suitable.

We work with a *global* (immutable) heap using variable names as pointers. This means that (1) whenever we put variable bindings on the heap we assume fresh names and (2) importantly, the heap uses *absolute* modalities. This contrasts to the *relative* modalities used in the typing rules, where, for instance, irrelevant variables can be resurrected or private variables become public in private contexts. Once pushed in the heap, a private value will stay private the rest of the run of the program. For quantitative applications, this means that the program will never over- or under-consume the initial budget of resources that it started with.

We have a notion of well-typed, and in fact well-qualified, heaps. Whereas typing rules for terms keep track of the modalities of the inputs, for heaps we track the modalities of the outputs. Thus we write $h : \gamma\Gamma$ when h provides $\gamma\Gamma$. While Γ has the form of a context, in this role it contains no modality nor type variables, and thus has only closed types. The variables of the heap are provided with a certain modality, but the term associated with a new provided variable may use old variables.

$$\frac{}{\{\} : []} \qquad\qquad \frac{h : (q\delta + \gamma)\Gamma \qquad \delta\Gamma \vdash t : A}{(h, x \mapsto t) : (\gamma\Gamma, x :^q A)}$$

Hence, in the heap construction rule the heap h does not provide x, but provides instead all the variables needed to construct ${}^q t$, with the appropriate modalities. Hereafter we use the $\gamma h, x \mapsto qu$ notation to mean $(\gamma, qx)(h, x \mapsto u)$ in a similar style to what we used for contexts.

Definition 6.1 (Machine Transitions). Depending on the head t, the machine makes transitions as given in Fig. 4. If the typing of a sub-term u is such that $\delta\Delta \vdash u : A$, then we write $|u|$ for δ.

Additionally we have a rule for weakening: $(\zeta + r\gamma)h \Vdash^r t \cdot \vec{e} \longrightarrow (\zeta + r\delta)h \Vdash^r t \cdot \vec{e}$, with the side condition $\gamma \leqslant \delta$, whose modality contexts γ and δ come from the weakening rule present on the left-hand-side state.

Theorem 6.2 (Modality Preservation). *If* $(h : \gamma\Gamma) \Vdash^r (t \cdot \vec{e} : C) \longrightarrow (h' : \gamma'\Gamma') \Vdash^{r'} (t' \cdot \vec{e}' : C')$ *then*

(1) $C = C'$, *and*
(2) *if* $h : \gamma\Gamma$ *then* $h' : \gamma'\Gamma'$.

Because we have well-typed states ($\gamma\Gamma \vdash \vec{e}(t) : C$), then the first item implies type preservation. The second item implies that modalities are preserved: if we start from a state where the heap provides what evaluating the term demands, it will remain so after a machine transition.

7 RELATIONAL SEMANTICS

We adopt the standard semantics of typed lambda-calculus, which interprets closed types as sets and closed terms as elements. This first semantics (Section 7.1) models modality polymorphism with modality abstraction and application, but ignores the effects of the modality on types. Such effects will be taken into account by the relational model (Sections 7.2 to 7.8).

7.1 Modality-oblivious set-theoretic model

The interpretation of closed types A by sets $(\!| A |\!)$ is parameterised by an interpretation $(\!| K |\!)$ of the type constants $K \in \mathsf{TyConst}$. On the right-hand-sides of the following equations, we refer to the set-theoretic cartesian product $\mathcal{A} \times \mathcal{B}$, the disjoint union $\mathcal{A} + \mathcal{B}$, the function space $\mathcal{A} \to \mathcal{B}$ and a (possibly infinite) product $\prod_{i:I} \mathcal{A}_i$ of sets. The latter can be seen as a dependent function space, thus, we eliminate it with application $f(j) : \mathcal{A}_j$ (given $f : \prod_{i:I} \mathcal{A}_i$ and $j : I$).

$$\begin{array}{lll@{\qquad}lll}
\llparenthesis 1 \rrparenthesis & = & \{()\} & \llparenthesis {}^{p}A \to B \rrparenthesis & = & \llparenthesis A \rrparenthesis \to \llparenthesis B \rrparenthesis \\
\llparenthesis A + B \rrparenthesis & = & \llparenthesis A \rrparenthesis + \llparenthesis B \rrparenthesis & \llparenthesis p\langle A \rangle \rrparenthesis & = & \llparenthesis A \rrparenthesis \\
\llparenthesis A \times B \rrparenthesis & = & \llparenthesis A \rrparenthesis \times \llparenthesis B \rrparenthesis & \llparenthesis \forall \alpha . B \rrparenthesis & = & \prod_{A:\mathsf{Ty}_0^0} \llparenthesis B[A/\alpha] \rrparenthesis \\
 & & & \llparenthesis \forall m . B \rrparenthesis & = & \prod_{p:\mathsf{Mod}} \llparenthesis B[p/m] \rrparenthesis
\end{array}$$

In the interpretation of predicative polymorphism $\forall \alpha . B$, the product ranges over all small monotypes $A : \mathsf{Ty}_0^0$. Modality polymorphism $\forall m . B$ is interpreted by a product over all modality constants $p : \mathsf{Mod}$. Note that $\llparenthesis B \rrparenthesis$ is defined by lexicographic recursion on the pair whose first component is the number of quantifiers in B and the second the syntactic size of B.

Contexts Γ are interpreted as sets $\llparenthesis \Gamma \rrparenthesis$ of finite maps η such that $\eta(x) : \llparenthesis \Gamma(x)[\eta] \rrparenthesis$—which is $\llparenthesis A[\eta] \rrparenthesis$—for all $(x{:}A) \in \Gamma$, further $\eta(m) : \mathsf{Mod}$ for all $m \in \mathrm{dom}(\Gamma)$ and $\eta(\alpha) : \mathsf{Ty}_0^0$ for all $\alpha \in \mathrm{dom}(\Gamma)$. In $\llparenthesis \Gamma(x)[\eta] \rrparenthesis$, we mean by $A[\eta]$ the parallel substitution in A of all type and modality bindings contained in η. Similarly, $q[\eta]$ shall denote the parallel substitution in q of all modality bindings contained in η.

Now, given $\eta : \llparenthesis \Gamma \rrparenthesis$, we can interpret a typed term $\gamma\Gamma \vdash t : A$ as an element $\llparenthesis t \rrparenthesis_\eta : \llparenthesis A[\eta] \rrparenthesis$ in the standard way.

$$\begin{array}{lll@{\qquad}lll}
\llparenthesis x \rrparenthesis_\eta & = & \eta(x) & & & \\
\llparenthesis \lambda {}^{q}x . t \rrparenthesis_\eta (a : \llparenthesis A[\eta] \rrparenthesis) & = & \llparenthesis t \rrparenthesis_{\eta[x \mapsto a]} & \llparenthesis t\, {}^{q}u \rrparenthesis_\eta & = & \llparenthesis t \rrparenthesis_\eta (\llparenthesis u \rrparenthesis_\eta) \\
\llparenthesis \Lambda \alpha . t \rrparenthesis_\eta (A : \mathsf{Ty}_0^0) & = & \llparenthesis t \rrparenthesis_{\eta[\alpha \mapsto A]} & \llparenthesis t \cdot A \rrparenthesis_\eta & = & \llparenthesis t \rrparenthesis_\eta (A\eta) \\
\llparenthesis \forall m . t \rrparenthesis_\eta (p : \mathsf{Mod}) & = & \llparenthesis t \rrparenthesis_{\eta[m \mapsto p]} & \llparenthesis t \cdot q \rrparenthesis_\eta & = & \llparenthesis t \rrparenthesis_\eta (q\eta) \\
\llparenthesis \mathsf{inj}_i\, t \rrparenthesis_\eta & = & \iota_i \llparenthesis t \rrparenthesis_\eta & \llparenthesis () \rrparenthesis_\eta & = & () \\
\llparenthesis [{}^{q}t] \rrparenthesis_\eta & = & \llparenthesis t \rrparenthesis_\eta & \llparenthesis (t, u) \rrparenthesis_\eta & = & (\llparenthesis t \rrparenthesis_\eta, \llparenthesis u \rrparenthesis_\eta)
\end{array}$$

In the case for $\lambda {}^{q}x . t$, we assume $\gamma\Gamma, x : {}^{q}A \vdash t : B$. For disjoint sum types, we make use of the injections $\iota_i : \mathcal{A}_i \to \mathcal{A}_1 + \mathcal{A}_2$ and the copairing $[f_1, f_2] : \mathcal{A}_1 + \mathcal{A}_2 \to \mathcal{B}$ of functions $f_i : \mathcal{A}_i \to \mathcal{B}$.

$$\begin{array}{lll@{\qquad}l}
\llparenthesis \mathsf{case}\ {}^{p}t \ \mathsf{of}\ \{\mathsf{inj}_1\, x_1 \mapsto u_1; \mathsf{inj}_2\, x_2 \mapsto u_2\} \rrparenthesis_\eta & = & [f_1, f_2] \llparenthesis t \rrparenthesis_\eta & \text{where } \gamma\Gamma \vdash t : A_1 + A_2 \text{ and} \\
 & & & f_i(a : \llparenthesis A_i \eta \rrparenthesis) = \llparenthesis u_i \rrparenthesis_{\eta[x_i \mapsto a]} \\
\llparenthesis \mathsf{let}\, (x_1, x_2) = {}^{q}t \ \mathsf{in}\ u \rrparenthesis_\eta & = & \llparenthesis u \rrparenthesis_{\eta[x_1 \mapsto a_1][x_2 \mapsto a_2]} & \text{where } (a_1, a_2) = \llparenthesis t \rrparenthesis_\eta \\
\llparenthesis \mathsf{let}\, [{}^{q}x] = t \ \mathsf{in}\ u \rrparenthesis_\eta & = & \llparenthesis u \rrparenthesis_{\eta[x \mapsto \llparenthesis t \rrparenthesis_\eta]} &
\end{array}$$

Remark 2. *As for machine states, the interpretation works on typed terms, and thus the whole typing derivation should be written, but we write only the term for concision. However in this case, different typing derivations for the same term yield the same semantics. In term notation, the semantics of (invisible) weakening would read* $\llparenthesis t \rrparenthesis_\eta = \llparenthesis t \rrparenthesis_\eta$*, and thus we omitted it above.*

The model uses sets and pointwise definition of functions, but it can be easily reformulated in point-free style and then be generalised to an arbitrary cartesian-closed category with infinite products and distributive coproducts. Closed types and contexts would then be interpreted as objects and terms $\gamma\Gamma \vdash t : A$ as morphisms from $\llparenthesis \Gamma \rrparenthesis$ to $\llparenthesis A \rrparenthesis$.

7.2 Relational model for parametricity and usage-tracking: framework

On top of the set-theoretic interpretation, we define a logical relation to express three kinds of program properties:

(1) Parametricity: Programs cannot inspect types.
(2) Modality irrelevance: Programs cannot inspect modalities (Theorem 7.9).
(3) And most interestingly, program properties implied by modalities (Theorem 7.10, Section 8).

Our model combines aspects of the parametricity interpretation [Reynolds 1983], classified sets [Abadi et al. 1999; Kavvos 2019], and resource indexing [Atkey and Wood 2018; Brunel et al. 2014].

As for Abadi et al. [1999], types are interpreted by a *family* of relations indexed by worlds $w : W$ (instead of just a single relation). We let $\mathsf{Rel}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{P}(\mathcal{A}_1 \times \mathcal{A}_2)$ denote the set of relations between $\mathcal{A}_1$ and $\mathcal{A}_2$, and $\mathsf{WRel}(\mathcal{A}_1, \mathcal{A}_2)$ denote the contravariant ($w \leqslant w' \to R^{w'} \subseteq R^w$) families $W \to \mathsf{Rel}(\mathcal{A}_1, \mathcal{A}_2)$.

Each type A is interpreted as a family of relations $[\![A]\!]_{\sigma;\rho} \in \mathsf{WRel}((\!| A\sigma_1 |\!), (\!| A\sigma_2 |\!))$. Herein $\sigma = (\sigma_1, \sigma_2)$ is a pair of finite maps σ_i, each of them mapping type variables α to closed monotypes $A : \mathsf{Ty}_0^0$ and modality variables m to modality constants $q : \mathsf{Mod}$. The finite map ρ maps each type variable α to a family of relations in $\mathsf{WRel}((\!| \sigma_1(\alpha) |\!), (\!| \sigma_2(\alpha) |\!))$, and each modality variable m to a modality constant p which can be different from both $\sigma_1(m)$ and $\sigma_2(m)$.

The set of worlds W is equipped with a preordered commutative monoid structure whose (monotone) operation is written $\bullet$ and its unit ε. In a first approximation, $(\bullet, \varepsilon)$ can be thought of as $(+, 0)$ from the modality ringoid. To gain some intuition for the role of W, we consider how it can be instantiated in specific cases. However, we stress that Λ^{p} is fully generic in this respect: every program is susceptible to be interpreted in either of the following ways, depending on the application.

Security levels. For Abadi et al. [1999] each world $w : W$ stands for a security level, and the relation $\mathcal{R}^w$ will identify values that an agent of clearing level w is not allowed to distinguish (“see”). One extreme is the discrete relation that hides nothing and allows one to distinguish everything (full information); the other extreme is the full relation that identifies any two values and thus hides everything (no information). The index set W may be (pre)ordered, putting levels w into a hierarchy. The higher the clearing of an agent w, the more it is allowed to see, thus, the fewer values become related by the indistinguishability relations. Thus $\mathcal{R}^w$ is contravariant in w, i. e., $w \leqslant w'$ implies $\mathcal{R}^{w'} \subseteq \mathcal{R}^w$.

Sensitivity. For Reed and Pierce [2010], indices $w : W$ are non-negative reals, and $a \mathrel{\mathcal{R}^w} b$ shall mean that the distance between a and b is at most w (for a suitable metric). (To avoid clutter, we write relations infix.) Here, $\mathcal{R}^w$ is covariant on w in the natural order on reals. We still have contravariance, because we set $w \leqslant w'$ to be $w \geqslant_{\mathbb{R}} w'$, the opposite of the natural order.

Quantitative analysis. For quantitative analyses [Atkey 2018; Brunel et al. 2014; Ghica and Smith 2014], a world $w : W$ in $a \mathrel{\mathcal{R}^w} b$ denotes the *resources* needed to construct a or b. *Insufficient* resources w prevent $a \mathrel{\mathcal{R}^w} b$ from holding, and *excessive* resources may have the same effect if we model strict *linearity* rather than just *affinity*. A world w could be a multiset of elementary resources that are composed to build a (and b would be built from another copy of the same resources). Such multisets form indeed a commutative monoid with $w \bullet w'$ denoting the multiset union $w \uplus w'$ and ε the empty multiset $\emptyset$. The preorder $w \leqslant w'$ may be simply equality when we insist on exact resource consumption.

Uncertainty about resources can be expressed by letting a world w be a set of multisets m. To satisfy $a \mathrel{\mathcal{R}^w} b$, we are allowed to choose one multiset $m \in w$, but need to consume it fully to build our object a (and build b from the same m). The monoid structure is then given by $\varepsilon = \{\emptyset\}$ (the set containing just the empty multiset) and $w \bullet w' = \{m \uplus m' \mid m \in w \text{ and } m' \in w'\}$. The preorder $w \leqslant w'$ shall be $w \supseteq w'$, meaning that going up in the preorder we eliminate alternatives. Then $\mathcal{R}^w$ is contravariant in w.

7.3 Relational interpretation of simple types

Let us now return to the definition of the semantics $[\![A]\!]$. In order to define $[\![A]\!]_{\sigma;\rho}$ in a concise way, we introduce some constructions on relation families. First, observe that WRel inherits all logical connectives by pointwise definition, for instance, we can define $\top^w$ to be the full relation, yielding true if applied to any two points. Likewise, we can define finite and infinite intersection ($\cap$ and $\bigcap$) of relation families pointwise via conjunction and universal quantification, and similar finite and infinite union ($\cup$ and $\bigcup$) via disjunction and existential quantification.

Further, recall the standard product and function space on relations. Let $\mathcal{R} : \mathsf{Rel}(\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{S} : \mathsf{Rel}(\mathcal{B}_1, \mathcal{B}_2)$.

$$\begin{array}{lcl} \mathcal{R} \times \mathcal{S} & : & \mathsf{Rel}(\mathcal{A}_1 \times \mathcal{B}_1, \mathcal{A}_2 \times \mathcal{B}_2) \\ & = & \{((a_1, b_1), (a_2, b_2)) \mid (a_1, a_2) \in \mathcal{R} \text{ and } (b_1, b_2) \in \mathcal{S}\} \\ \mathcal{R} + \mathcal{S} & : & \mathsf{Rel}(\mathcal{A}_1 + \mathcal{B}_1, \mathcal{A}_2 + \mathcal{B}_2) \\ & = & \{(\iota_1 a_1, \iota_1 a_2) \mid (a_1, a_2) \in \mathcal{R}\} \cup \{(\iota_2 b_1, \iota_2 b_2) \mid (b_1, b_2) \in \mathcal{S}\} \\ \mathcal{R} \to \mathcal{S} & : & \mathsf{Rel}(\mathcal{A}_1 \to \mathcal{B}_1, \mathcal{A}_2 \to \mathcal{B}_2) \\ & = & \{(f_1, f_2) \mid (f_1(a_1), f_2(a_2)) \in \mathcal{S} \text{ for all } (a_1, a_2) \in \mathcal{R}\} \end{array}$$

These constructions extend pointwise to families of relations WRel.

Then, we interpret linear type constructors as operations on relation families that actually inspect the index w. Let $\mathcal{R} : \mathsf{WRel}(\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{S} : \mathsf{WRel}(\mathcal{B}_1, \mathcal{B}_2)$. In the following, we use just a as shorthand for the pair (a_1, a_2); likewise for b.

$$\begin{array}{lclcl} 1^w & = & \{((),()) \mid w \leqslant \varepsilon\} & : & \mathsf{Rel}(1, 1) \\ (\mathcal{R} \otimes \mathcal{S})^w & = & \bigcup_{w \leqslant w_a \bullet w_b} (\mathcal{R}^{w_a} \times \mathcal{S}^{w_b}) & : & \mathsf{Rel}(\mathcal{A}_1 \times \mathcal{B}_1, \mathcal{A}_2 \times \mathcal{B}_2) \\ (\mathcal{R} \multimap \mathcal{S})^w & = & \bigcap_{w_b \leqslant w \bullet w_a} (\mathcal{R}^{w_a} \to \mathcal{S}^{w_b}) & : & \mathsf{Rel}(\mathcal{A}_1 \to \mathcal{B}_1, \mathcal{A}_2 \to \mathcal{B}_2) \end{array}$$

In the following, let us interpret these definitions for different analyses.

Unit. The unit set 1 contains no information, and thus its inhabitant () can be constructed in a world w such that: $w \leqslant \varepsilon$. In terms of security, the empty tuple is (by its very nature) always indistinguishable from itself. Thus the indistinguishability relation may hold at all security levels, as there is never a need to look into the empty tuple, regardless the capabilities an agent is equipped with. Thus, $w \leqslant \varepsilon$ does not place any restriction on w. This suggests that for a complete lattice W of capabilities, the unit ε should be the top element $\top$, making $w \leqslant \varepsilon$ vacuously true. For sensitivity analysis, any two inhabitants of the unit set have distance 0, thus, the unit ε of monoid W is the real 0, and the condition $w \leqslant \varepsilon$ equivalent to $w \geqslant_{\mathbb{R}} 0$, is vacuously true. When worlds are sets of multisets, as in quantitative analysis, $w \leqslant \varepsilon$ expresses that the sets w contains the empty multiset. This means that no resources are required, but the empty resource bag needs to be one of our possibilities.

Tensor product. Recall that $(a_1, b_1)\ (\mathcal{R} \otimes \mathcal{S})^w\ (a_2, b_2)$ holds iff. there are w_a, w_b such that $a_1\ \mathcal{R}^{w_a}\ a_2$ and $b_1\ \mathcal{R}^{w_b}\ b_2$ and $w \leqslant w_a \bullet w_b$. In the quantitative interpretation, we need to break down the resources w available for the construction of the pair into resources w_a for the first component and w_b for the second component. This is expressed by the condition $w \leqslant w_a \bullet w_b$. For security analysis, the capability to access a pair should include the capability to access both components. Turning this statement around, a pair is only indistinguishable from another pair if both respective components are so. On a capability lattice, $w_a \bullet w_b$ would be the meet $w_a \wedge w_b$, breaking the condition $w \leqslant w_a \wedge w_b$ into the pair of conditions $w \leqslant w_a$ and $w \leqslant w_b$. In sensitivity analysis [Reed and Pierce 2010], the distance of pairs is the sum of distances of its respective components.

This means that two pairs are within distance w if its components are within distances w_a and w_b and $w \geqslant_{\mathbb{R}} w_a + w_b$. The composition $w_a \bullet w_b$ is thus addition.

Lemma 7.1 (Unit law). *$\mathcal{R} \otimes 1$ is isomorphic to $\mathcal{R}$.*

Lemma 7.2 (Symmetry). *$\mathcal{R} \otimes \mathcal{S}$ is isomorphic to $\mathcal{S} \otimes \mathcal{R}$.*

Sum. The disjoint sum $(\mathcal{R} + \mathcal{S})^w$ is defined directly in terms of $\mathcal{R}^w$ and $\mathcal{S}^w$ at the same world w. In this interpretation, making the choice does not cost any resources. Conversely, this correctly models that any value of a closed (non-abstract) data type can be constructed in any modality context, and lives at the bottom of the informational lattice.

In other words, the indistinguishability relation for the sum type only inherits the indistinguishability from the components. Put plainly, if a and b are identified, so are $\iota_i(a)$ and $\iota_i(b)$. Different injections are *always* distinguished, thus, the bit of information associated to the choice of injection is visible to all informational levels.

For sensitivity analysis, the distance of different injections is ∞, thus, they are not related by any $\mathcal{R}^w$ since we restrict worlds to $< \infty$. The genericity of our semantics takes the burden of choice from us; otherwise, we could have been tempted to include a world ∞ where everything is related, but then we would have needed a special case for sum types. In fact, a world ∞ would contain no information, thus, it is anyway redundant.

Linear function space. Recall that $f_1\ (\mathcal{R} \multimap \mathcal{S})^w\ f_2$ iff for all $a_1\ \mathcal{R}^{w_a}\ a_2$ and all w_b with $w_b \leqslant w \bullet w_a$ we have $f_1(a_1)\ \mathcal{S}^{w_b}\ f_2(a_2)$. Thus, the definition of $\mathcal{R} \multimap \mathcal{S}$ states that a function can be constructed from resources w if for any argument that brings its own resources w_a the function result can be constructed with resources w_b with $w_b \leqslant w \bullet w_a$. Read differently, the resources for a function application is the composition of the resources for both function and argument. Functions stemming from closed terms do not need own resources, thus, they start with ε, but as a curried function is applied to its arguments one after another, it accumulates the resources coming with each argument to eventually construct a result from all the gathered resources. Technically, the construction of $\multimap$ can be derived from the fact that $(\mathcal{S} \multimap \_)$ should be a right adjoint to $(\_ \otimes \mathcal{S})$ to allow currying and uncurrying.

Lemma 7.3 (Currying). *$\mathcal{R} \otimes \mathcal{S} \multimap \mathcal{T}$ is isomorphic to $\mathcal{R} \multimap (\mathcal{S} \multimap \mathcal{T})$.*

From the security perspective, access to a function and access to its argument should be sufficient to get access to the result. Thus, the definition of $(\multimap)$, invoking $w_b \leqslant w \bullet w_a$, does the correct thing for access control to functions.

Reed and Pierce [2010] define the distance of two 1-sensitive functions f, f' as the supremum of their distance at each point in their domain. In our notation that would mean that $(f, f') \in (\mathcal{R} \multimap \mathcal{S})^w$ iff $(f(a), f'(a)) \in \mathcal{S}^w$ for all a. This is a consequence of the definition of $\multimap$ for reflexive a, meaning $(a, a) \in \mathcal{R}^0$. Our definition requires more generally that $(a, a') \in \mathcal{R}^{w_a}$ should imply $(f(a), f'(a')) \in \mathcal{S}^{w+w_a}$. This could be equivalent to Reed and Pierce given that 1-sensitivity implies $(f(a), f(a')) \in \mathcal{S}^{w_a}$ and the triangle inequality $\mathcal{S}^w \circ \mathcal{S}^{w_a} \subseteq \mathcal{S}^{w+w_a}$ could be proven on homogeneous relations. However, our relations are heterogeneous, and the triangle law is ill-formed in general. Our definition thus properly generalises the one of Reed and Pierce.

7.4 Non-Idempotent Intersection

In order to characterise the interpretation of modal boxing $p\langle A \rangle$, we introduce the operation $\mathcal{R} \owedge \mathcal{S}$ for relation families $\mathcal{R}, \mathcal{S} : \mathsf{WRel}(\mathcal{A}_1, \mathcal{A}_2)$. It is similar to $\mathcal{R} \otimes \mathcal{S}$, only that it is akin to a non-idempotent *intersection* type rather than a product.

$$a \in (\mathcal{R} \owedge \mathcal{S})^w :\Longleftrightarrow \exists w_r, w_s.\ w \leqslant w_r \bullet w_s \wedge a \in \mathcal{R}^{w_r} \wedge a \in \mathcal{S}^{w_s}$$

Here, we split the resources w for a into w_r and w_s to build the *same* a twice, once in $\mathcal{R}$ and once in $\mathcal{S}$. Another way to write the non-idempotent intersection is:

$$(\mathcal{R} \circledcap \mathcal{S})^w = \bigcup_{w \leqslant w_r \bullet w_s} (\mathcal{R}^{w_r} \cap \mathcal{S}^{w_s})$$

Non-idempotent intersection has unit $\circledtop : \mathsf{WRel}(\mathcal{A}_1, \mathcal{A}_2)$ defined by

$$a \in \circledtop^w \iff w \leqslant \varepsilon.$$

This is similar to family 1 only that it can be used at any type, not just the unit type.

Lemma 7.4 (Distribution properties of non-idempotent intersection).
(1) $\mathcal{R} \circledcap (\mathcal{S} \cup \mathcal{T}) = (\mathcal{R} \circledcap \mathcal{S}) \cup (\mathcal{R} \circledcap \mathcal{T})$ *and* $\mathcal{R} \circledcap \bigcup_i \mathcal{S}_i = \bigcup_i (\mathcal{R} \circledcap \mathcal{S}_i)$.
(2) $(\mathcal{R}_1 \cap \mathcal{R}_2) \circledcap (\mathcal{S}_1 \cap \mathcal{S}_2) \subseteq (\mathcal{R}_1 \circledcap \mathcal{S}_1) \cap (\mathcal{R}_2 \circledcap \mathcal{S}_2)$ *and* $(\bigcap_{i:I} \mathcal{R}_i) \circledcap (\bigcap_{i:I} \mathcal{S}_i) \subseteq \bigcap_{i:I} (\mathcal{R}_i \circledcap \mathcal{S}_i)$.

7.5 Subexponentials

The interpretation of modalities via subexponentials

$$!^p_{A_1,A_2} : \mathsf{WRel}(\llparenthesis A_1 \rrparenthesis, \llparenthesis A_2 \rrparenthesis) \to \mathsf{WRel}(\llparenthesis A_1 \rrparenthesis, \llparenthesis A_2 \rrparenthesis)$$

is a parameter to our model, however, we require that $\mathsf{WRel}(\llparenthesis A_1 \rrparenthesis, \llparenthesis A_2 \rrparenthesis)$ is almost a left module to ringoid Mod via action $(p, \mathcal{R}) \mapsto !^p_{A_1,A_2} \mathcal{R}$. More precisely, the following laws must hold:

$$\begin{array}{llll}
!^1 \; \mathcal{R} = \mathcal{R} & \qquad & !^{pq} \; \mathcal{R} & \subseteq \; !^p !^q \mathcal{R} \\
!^0 \; \mathcal{R} \subseteq \circledtop & & !^{p+q} \mathcal{R} & \subseteq \; !^p \mathcal{R} \circledcap !^q \mathcal{R} \\
!^{p \wedge q} \mathcal{R} \subseteq \; !^p \mathcal{R} \cap !^q \mathcal{R} & & !^p \; (\mathcal{R} \cap \mathcal{S}) & \subseteq \; !^p \mathcal{R} \cap !^p \mathcal{S} \\
!^p \; \circledtop = \circledtop & & !^p \; (\mathcal{R} \circledcap \mathcal{S}) & = \; !^p \mathcal{R} \circledcap !^p \mathcal{S}
\end{array}$$

Note that the distribution of the meet entails monotonicity: $!^p \mathcal{R} \subseteq !^q \mathcal{R}$ for $p \leqslant q$ (which is defined as $p \wedge q = p$).

Beside the above properties we require subexponentials to distribute over sums and products as follows:

$$\begin{array}{lll}
!^p 1 & = & 1 \\
!^p (\mathcal{R} \otimes \mathcal{S}) & = & !^p \mathcal{R} \otimes !^p \mathcal{S} \\
!^p (\mathcal{R} + \mathcal{S}) & \subseteq & !^p \mathcal{R} + !^p \mathcal{S} \quad \text{if } p \leqslant 1
\end{array}$$

Finally, to model box-introduction ($p\langle\cdot\rangle$-intro), we require $!^p$ to be *functorial* in the following sense:

$$(\mathcal{R} \multimap \mathcal{S})^\varepsilon \subseteq (!^p \mathcal{R} \multimap !^p \mathcal{S})^\varepsilon$$

In other words, $\bigcap_w (\mathcal{R}^w \to \mathcal{S}^w) \subseteq \bigcap_w ((!^p \mathcal{R})^w \to (!^p \mathcal{S})^w)$. In the following, we provide some insights into the operator $!^p$ by spelling it out for some instances of modal type systems.

7.5.1 Sensitivity analysis. In sensitivity analysis with $p : \mathbb{R}^\infty_{\geqslant 0}$, the scaling modality $!^p$ inflates distances by a factor of p; in our setting,

$$\begin{array}{lll}
(!^p \mathcal{R})^w & = & \mathcal{R}^{w/p} \qquad \text{for } p > 0 \\
(!^0 \mathcal{R})^w & = & \top^w.
\end{array}$$

In particular, $(a_1, a_2) \in (!^\infty \mathcal{R})^w$ iff $(a_1, a_2) \in \mathcal{R}^0$, stating that two points can only be related in $!^\infty \mathcal{R}$ if they had distance 0 in $\mathcal{R}$. This can be interpreted that all non-identical points in $\mathcal{R}$ are infinitely apart in $!^\infty \mathcal{R}$, thus, the space $!^\infty \mathcal{R}$ is discrete. In particular, unrestricted functions $(!^\infty \mathcal{R}) \multimap \mathcal{S}$ are c-sensitive for any c, and thus, devoid of any sensitivity information.

A corner case is $p = 0$ which means multiplying all distances by 0, making all finitely apart points equal. Since w cannot be infinity, we can consistently set $(a_1, a_2) \in (!^0 \mathcal{R})^w$ to be true. More concisely, $!^0\mathcal{R} = \top = ⓣ$, the latter holding since $w \leqslant \varepsilon$ is vacuously true ($w \geqslant_{\mathbb{R}} 0$ is true). We see that $!^0$ lumps all points together, and the space $!^0\mathcal{R}$ is codiscrete. One-sensitivity for a function in $(!^0\mathcal{R}) \multimap \mathcal{S}$ means that it needs to keep the lump together, thus, unless the codomain $\mathcal{S}$ is codiscrete, it cannot use its argument relevantly.

Note that action $!^p$ is contravariant in p w. r. t. the natural order on $\mathbb{R}^{\infty}_{\geqslant 0}$ due to p occuring in the denominator (and $\mathcal{R}$ being covariant w. r. t. the natural order). Thus $!^p$ is covariant in p w. r. t. the modality order.

Lemma 7.5 (Soundness of scaling). *The operator $!^p$ has the required properties.*

7.5.2 Security. In the security case, modalities form a *distributive* lattice, and so it is isomorphic to a lattice generated by set inclusion over a carrier set C, corresponding to capabilities. Thus we represent a world as a subset of C, and define $!^p\mathcal{R}^w = \mathcal{R}^{w \setminus p}$, where we consider here p as its representation as a subset of C. The operations on modalities are thus $(\wedge) = (+) = (\cap)$ and $(\cdot) = (\cap)$, and 0 corresponds to C and 1 to $\{\}$. As suggested above, $\varepsilon = C$ and $(\leqslant) = (\subseteq)$. Intuitively, the fewer capabilities one has, the more things become equal, according to contravariance of WRel.

Lemma 7.6 (Soundness of capabilities).

7.5.3 Quantitative analysis. In quantitative analysis, p is a set of natural numbers. Let our worlds w be sets of multisets of resources. Here, a multiset $m \in w$ should be one possibility of available resources that have to be consumed exactly, but w offers several resource bags to choose from. Let $0_W = \{\emptyset\}$ and $w_1 +_W w_2 = \{m_1 \uplus m_2 \mid m_1 \in w_1 \text{ and } m_2 \in w_2\}$. This allows us to define $n \cdot w = \underbrace{w +_W \ldots +_W w}_{n \text{ times}}$ for $n \in \mathbb{N}$.

Modalities p act on worlds w via $p \cdot w = \bigcup_{n \in p}(n \cdot w)$. It is easy to see that W is a left module to ringoid M under action $(p, w) \mapsto p \cdot w$:

$$
\begin{array}{lll}
 & (p \wedge q) \cdot w & = p \cdot w \cup q \cdot w \\
1 \cdot w = w & pq \cdot w & = p \cdot (q \cdot w) \\
0 \cdot w = 0_W & (p + q) \cdot w & = p \cdot w +_W q \cdot w \\
p \cdot 0_W = 0_W & p \cdot (w_1 +_W w_2) & = p \cdot w_1 +_W p \cdot w_2
\end{array}
$$

This action allows us to define the subexponential:

$$(!^p\mathcal{R})^w = \bigcup_{w \leqslant p \cdot w'} \mathcal{R}^{w'}$$

Lemma 7.7 (Soundness of boxing).

7.6 Relational interpretation of polymorphism

Given families of sets $(\mathcal{A}_A)_{A:\mathsf{Ty}^0_0}$ and $(\mathcal{B}_B)_{B:\mathsf{Ty}^0_0}$ and a family of relations $(\mathcal{R}_{AB} : \mathsf{WRel}(\mathcal{A}_A, \mathcal{B}_B))_{A,B:\mathsf{Ty}^0_0}$ let

$$\left(\prod_{A,B:\mathsf{Ty}^0_0} \mathcal{R}_{AB}\right)^w = \{(f, g) \mid f : \prod_{A:\mathsf{Ty}^0_0} \mathcal{A}_A \text{ and } g : \prod_{B:\mathsf{Ty}^0_0} \mathcal{B}_B \text{ and } (f(A), g(B)) \in \mathcal{R}^w_{AB}\}.$$

We use an analogous definition for Mod instead of Ty^0_0. In fact, any index set I could replace Ty^0_0.

7.7 Definition of the relational interpretation

The relation family $[\![A]\!]_{\sigma;\rho} : \mathsf{WRel}((\!| A\sigma_1 |\!), (\!| A\sigma_2 |\!))$ is defined by induction on A as follows. Herein, the relational interpretation $[\![K]\!]$ of type constants K remains a parameter.

$$
\begin{array}{lcl}
[\![K]\!]_{\sigma;\rho} & = & [\![K]\!] \\
[\![\alpha]\!]_{\sigma;\rho} & = & \rho(\alpha) \\
[\![1]\!]_{\sigma;\rho} & = & 1 \\
[\![A \times B]\!]_{\sigma;\rho} & = & [\![A]\!]_{\sigma;\rho} \otimes [\![B]\!]_{\sigma;\rho} \\
[\![A + B]\!]_{\sigma;\rho} & = & [\![A]\!]_{\sigma;\rho} + [\![B]\!]_{\sigma;\rho} \\
[\![{}^{p}A \to B]\!]_{\sigma;\rho} & = & !^{p[\rho]}_{A[\sigma]}[\![A]\!]_{\sigma;\rho} \multimap [\![B]\!]_{\sigma;\rho} \\
[\![p\langle A\rangle]\!]_{\sigma;\rho} & = & !^{p[\rho]}_{A[\sigma]}[\![A]\!]_{\sigma;\rho} \\
[\![\forall\alpha.B]\!]_{\sigma;\rho} & = & \prod_{A_1,A_2:\mathsf{Ty}_0^0} \bigcap_{\mathcal{R}:\mathsf{WRel}((\!| A_1 |\!),(\!| A_2 |\!))} [\![B]\!]_{\sigma[\alpha\mapsto A];\rho[\alpha\mapsto\mathcal{R}]} \\
[\![\forall m.B]\!]_{\sigma;\rho} & = & \prod_{p_1,p_2:\mathsf{Mod}} \bigcap_{q:\mathsf{Mod}} [\![B]\!]_{\sigma[m\mapsto p];\rho[m\mapsto q]}
\end{array}
$$

Comparing to parametricity semantics. If we let $W = 1$ the unit set of worlds, our semantics defines a single relation for each type and is very similar to the usual parametricity interpretation of types. However, there are important differences:

The usual *identity extension lemma*, which implies that $[\![B]\!]^{\varepsilon}$ for a closed type B is equality, fails due to irrelevance. Given an irrelevant modality p, we have true $[\![p\langle\mathsf{Bool}\rangle]\!]$ false. Interpreting p as *secret* this expresses that for the public eye, the content of the box $p\langle\mathsf{Bool}\rangle$ is unobservable.

Further, the relation $[\![B]\!]^{\varepsilon}$ is not always reflexive. A counterexample is $B = \forall\alpha.\alpha$: If every monotype $A : \mathsf{Ty}_0^0$ is inhabited, we have an element $*_A : (\!| A |\!)$ for each A, thus $* := (*_A)_A : (\!| B |\!)$. However, $* \; [\![B]\!] \; *$ can be refuted by using the empty relation $\emptyset_A : \mathsf{Rel}((\!| A |\!), (\!| A |\!))$ on A to instantiate α. Then, we are obliged to show $*_A \; \emptyset_A \; *_A$, which is false by definition of the empty relation.

The counterexample to reflexivity uses a semantic element $* : (\!| \forall\alpha.\alpha |\!)$ that does not represent a λ-term. In Section 7.8 we will show that reflexivity *does* hold for all elements $(\!| t |\!)$ that represent a term.

7.8 Fundamental lemma

To state the fundamental lemma of logical relations, which establishes the soundness of the relational model, we need to extend the interpretation of types to contexts: for every qualified context $\gamma\Gamma$, we have $[\![\gamma\Gamma]\!]_{\sigma;\rho} \in \mathsf{WRel}((\!| \Gamma\sigma_1 |\!), (\!| \Gamma\sigma_2 |\!))$. The idea is to interpret context extension in the same way as the product of types: $[\![\gamma\Gamma, x : {}^{p}A]\!]_{\sigma;\rho} = [\![\gamma\Gamma]\!]_{\sigma;\rho} \otimes !^{p}[\![A]\!]_{\sigma;\rho}$

There is a formal mismatch by doing so: variables in Γ are accessed by name and not by index. This issue could be solved by defining a named version of $\otimes$, but doing so is straightforward and uninformative, and thus we do not go through this tedium. The introduction of modalities or types in Γ is dealt with by demanding that σ_i maps all variables in Γ to monotypes ($\sigma_i(\alpha) \in \mathsf{Ty}_0^0$), and ρ to matching relations ($\rho(\alpha) \in \mathsf{WRel}((\!| \sigma_1(\alpha) |\!), (\!| \sigma_2(\alpha) |\!))$).

Theorem 7.8 (Fundamental lemma). *If* $(\xi_1, \xi_2) \in [\![\gamma\Gamma]\!]^{w}_{\sigma;\rho}$ *then* $((\!| t |\!)_{(\sigma_1,\xi_1)}, (\!| t |\!)_{(\sigma_2,\xi_2)}) \in [\![A]\!]^{w}_{\sigma;\rho}$.

Proof. By induction on $\gamma\Gamma \vdash t : A$. The proof relies in particular on several lemmas connecting the qualifications of contexts to relations:

- $[\![(\gamma \wedge \delta)\Gamma]\!]_{\sigma;\rho} = [\![\gamma\Gamma]\!]_{\sigma;\rho} \cap [\![\delta\Gamma]\!]_{\sigma;\rho}$.
- $[\![(\gamma + \delta)\Gamma]\!]_{\sigma;\rho} = [\![\gamma\Gamma]\!]_{\sigma;\rho} \oslash [\![\delta\Gamma]\!]_{\sigma;\rho}$.
- $[\![p\Gamma]\!]_{\sigma;\rho} = !^{p}[\![\Gamma]\!]_{\sigma;\rho}$.

The first property entails soundness of weakening, as $\gamma \leqslant \delta$ then implies $[\![\gamma\Gamma]\!]_{\sigma;\rho} \subseteq [\![\delta\Gamma]\!]_{\sigma;\rho}$.

The functoriality of $!^p$ ensures the soundness of $p\langle\cdot\rangle$-INTRO. Further, the proof utilises the distribution properties of subexponentials in the elimination rules. E. g., for +-ELIM, $!^q[\![A_1 + A_2]\!] \subseteq !^q[\![A_1]\!] + !^q[\![A_2]\!]$ is used to distribute the q-scaling of the eliminatee t to the alternatives, to be bound to variable $x : {}^qA_i$ in the branches. Similarly, $!^q$ needs to distribute over tensor (in $\times$-ELIM) and $!^p$ (in $p\langle\cdot\rangle$-ELIM). □

A first corollary of the fundamental lemma is modality irrelevance, which states that the behaviour of *terms* is independent of the modality appearing in terms.

THEOREM 7.9 (MODALITY IRRELEVANCE). *If* $\vdash t : \forall m.\mathsf{Bool}$*, then* $f(p_1) = f(p_2)$ *for* $f = (\!| t |\!)$.

PROOF. The relational semantics of modality quantification gives us $f\ [\![\forall m.\mathsf{Bool}]\!]^\varepsilon\ f$, which yields by definition $f(p_1)\ [\![\mathsf{Bool}]\!]^\varepsilon\ f(p_2)$ for *all* modalities p_1, p_2. To conclude, it remains to observe that $[\![\mathsf{Bool}]\!]^\varepsilon$ is the identity relation. □

The second corollary is irrelevance to 0-qualified inputs. The term *irrelevance* was coined by Pfenning [2001] for proof systems, but in the literature on security it is spoken of *non-interference* for the equivalent property.

THEOREM 7.10 (IRRELEVANCE AND NON-INTERFERENCE). *If* $f : (\!| A \to {}^0B \to \mathsf{Bool} |\!)$, $\vdash u : A$, *and* $b_1, b_2 \in (\!| B |\!)$ *then* $f (\!| u |\!) b_1 = f (\!| u |\!) b_2$.

PROOF. We use an instance of the semantics with the trivial one-point world set ($W = 1$). Irrelevance comes from taking $!^0 R = \top$ and $!^p R = R$ if $p \neq 0$. (Checking the subexponential laws is routine.) The fundamental lemma gives $f[\![A \to {}^0B \to \mathsf{Bool}]\!]^\varepsilon f$ and by definition $f a_1 b_1\ [\![\mathsf{Bool}]\!]^\varepsilon\ f a_2 b_2$, which means $f a_1 b_1 = f a_2 b_2$, whenever $a_1\ [\![A]\!]^\varepsilon\ a_2$ and $b_1\ [\![!^0 B]\!]^\varepsilon\ b_2$. By another instance of the fundamental lemma we get $(\!| u |\!)\ [\![A]\!]\ (\!| u |\!)$. The definition of subexponential make the second requirement vacuous. □

Additionally, in security applications, one often generalises non-interference to any modality p that represents a secret security level, meaning that $p \leqslant 1$ can be ruled out. For us, this generalisation can be done at the level of Λ^{p} programs: because such a modality p is universally quantified at the outside, one can always instantiate p with 0; unless it is also constrained to be observable ($p \leqslant 1$) — in which case the constraint cannot be satisfied.

Example 7.11. For example, the server of Section 4.3.2 takes as parameters a series of security levels $c_1, c_2, \ldots, c_n$ constrained by a policy, eventually realised as terms of type *CanFlow* $c_i\ c_j$ — itself defined as $\forall\alpha.c_i\langle\alpha\rangle \to c_j\langle\alpha\rangle$. We can show that the server is secure, in the sense that it does not observe any of the messages which it handles (but only forwards them to suitably trusted clients). To carry out the proof, we must first check if we can substitute every c_i by 0. The question which arises is then if the policy can be realised, namely, can we construct the terms of type $c_i\langle\alpha\rangle \to c_j\langle\alpha\rangle$? The answer is yes, because the types reduce to $0\langle\alpha\rangle \to 0\langle\alpha\rangle$.

As a negative example, we can attempt to prove that the messages coming from c_1 are private to all other clients c_i, for $i \neq 1$. We substitute c_1 by 0 and check if we can construct $c_i\langle\alpha\rangle \to c_1\langle\alpha\rangle$. This is now impossible (because $c_1 = 0$ and c_i is arbitrary). Hence, the result holds only if *CanFlow* $c_i\ c_1$ is not found in the policy, for every $c_i \neq c_1$.

8 FREE THEOREMS

Let us apply the relational semantics to establish some properties of terms and types of Λ^{p}.

8.1 On Church encodings

An application of parametricity is the adequacy of Church encodings [Böhm and Berarducci 1985]. We investigate one instance that goes back to Reynolds [1983] and also appears as one of Wadler's "free theorems" [1989, Section 3.8]:

$$A \cong \forall\beta.\, (A \to \beta) \to \beta$$

The type on the right is the Church encoding of the not very interesting data type $\mathsf{Wrap}\, A$ that has a single constructor wrap with a single argument of type A, basically just wraps the elements of A. Unsurprisingly, this wrapping is not expected to make a difference, hence, the isomorphism.

A more refined picture on Church encodings makes use of linearity: First, constructors are naturally linear functions since their intended semantics is to form a new cell containing all their arguments exactly once. Secondly, some constructors appear exactly once in certain data structures, e. g., the zero in Peano natural numbers or the nil in lists. In our case, there exactly one occurrence of constructor wrap in any value of type $\mathsf{Wrap}\, A$, hence, a refined encoding of the wrapping type is:

$$\mathsf{Wrap}\, A = \forall\beta.\, (A \multimap \beta) \multimap \beta$$

We shall now demonstrate that this variant is still isomorphic to A. In fact, the original proof [Hasegawa 1994] via parametricity carries over to our setting, with some modifications:

1. We restrict to monotypes A since we need to instantiate β by A in the proof.
2. We have to tread more carefully since we do not have the identity extension lemma, i. e., we cannot assume that the relation $\llbracket A \rrbracket$ associated to a closed type A is set-theoretic equality. However, we will treat $\llbracket A \rrbracket$ as the *definition* of equality on type A and formulate our argument modulo the relation $\llbracket A \rrbracket$.

Maybe surprisingly, the proof does not require any reference to resources: we work with a single world and thus a single relation for each type. The quantitative aspects enter the picture in that the two directions of the isomorphism given by Wadler can be assigned *linear* types:

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{wrap} & : & A \multimap \mathsf{Wrap}\, A & \mathsf{unwrap} & : & \mathsf{Wrap}\, A \multimap A \\
\mathsf{wrap}\; a & = & \Lambda\beta.\lambda(k : A \multimap \beta).\, k\, a & \mathsf{unwrap}\; f & = & f\, A\, (\lambda x.\, x)
\end{array}$$

It is easy to see that $\mathsf{unwrap} \circ \mathsf{wrap}$ is the identity, but in the other direction we have to show that $\mathsf{wrap}\,(\mathsf{unwrap}\; t)$ which is $\Lambda\beta.\lambda k.\, k\,(t\, A\, (\lambda x.\, x))$ has the same meaning as t for any $\vdash t : \mathsf{Wrap}\, A$. To this end, we use the abstraction theorem for t *twice*.

First, since $\vdash t\, A\, (\lambda x.\, x) : A$, the abstraction theorem gives us for $a_0 := \llparenthesis\, t\, A\, (\lambda x.\, x)\, \rrparenthesis$ reflexivity $a_0 \llbracket A \rrbracket a_0$. With $f := \llparenthesis\, t\, \rrparenthesis$ and $\mathrm{id} := \llparenthesis\, \lambda x.\, x\, \rrparenthesis$ this means reflexivity for $f(A)(\mathrm{id})$ which we shall need below.

Secondly, for our goal $\llparenthesis\, \Lambda\beta.\lambda k.\, k\,(t\, A\, (\lambda x.\, x))\, \rrparenthesis = \llparenthesis\, t\, \rrparenthesis$ it is sufficient to show that for any closed monotype B and every function $k : \llparenthesis\, A\, \rrparenthesis \to \llparenthesis\, B\, \rrparenthesis$ we have $k(f(A)(\mathrm{id})) = f(B)(k)$. We use the abstraction theorem on $\vdash t : \forall\beta.\, (A \multimap \beta) \multimap \beta$ replacing β by the types A and B and the relation $\mathcal{R} : \mathsf{Rel}(\llparenthesis\, A\, \rrparenthesis, \llparenthesis\, B\, \rrparenthesis)$ defined by:

$$a\; \mathcal{R}\; b \;:\Longleftrightarrow\; \forall a'.\, a \llbracket A \rrbracket a' \implies k(a') = b.$$

The generalisation to all a' that are related to a replaces the identity extension lemma that would say that $\llbracket A \rrbracket$ is equality. The abstraction theorem can give us $f(A)(\mathrm{id})\; \mathcal{R}\; f(B)(k)$ which implies our goal $k(f(A)(\mathrm{id})) = f(B)(k)$ with $a = a' = a_0 = f(A)(\mathrm{id})$, exploiting reflexivity of a_0. However, the instantiation of the abstraction theorem requires us to show that $\mathrm{id} \llbracket A \multimap \beta \rrbracket k$. To this end, assume $a \llbracket A \rrbracket a'$ and conclude $\mathrm{id}(a)\; \mathcal{R}\; k(a')$ by the very definition of $\mathcal{R}$. □

Instantiating type A by $p\langle A\rangle$ we get the isomorphism:

$$p\langle A\rangle \cong \forall\beta.\, (p\langle A\rangle \multimap \beta) \multimap \beta \cong \forall\beta.\, ({}^{p}A \to \beta) \multimap \beta$$

This would give us subexponentials via Church encoding, albeit raising the type level from monotype to polytype.

8.2 Permutations

Atkey and Wood [2018] show that any list transformation List $K \multimap$ List K with an abstract type K is a permutation. To this end, they use lists modulo permutation as worlds, thus, $W =$ List K with the standard monoid structure (empty list and append) and $l \leqslant l'$ if l is a permutation of l'. The relational semantics is induced by $k_1 \ [\![K]\!]^w \ k_2 \iff w = [k_1] = [k_2]$, i. e., w is the singleton list containing k_1 that is equal to k_2.

Here, we will show the simpler fact that every term

$$\vdash t : (K \times K) \multimap (K \times K)$$

implements a permutation, thus, $(\!| t |\!)$ is either identity id or a swap of the elements of the pair. We use the same worlds, lists modulo permutation, but choose to represent them directly as multisets (e. g., could be implemented as $K \to \mathbb{N}$). The monoidal structure is multiset union, and $\leqslant$ is just equality. The relational semantics is constructed from $k_1 \ [\![K]\!]^w \ k_2 \iff w = \{\!| k_1 |\!\} = \{\!| k_2 |\!\}$.

The fundamental theorem for t gives $f \ [\![(K \times K) \multimap (K \times K)]\!]^\emptyset \ f$ with $f = (\!| t |\!)$. Assuming $k_1, k_2 : (\!| K |\!)$, we get $f(k_1, k_2) \ [\![K \times K]\!]^{\{\!| k_1, k_2 |\!\}} \ f(k_1, k_2)$. With $(k_1', k_2') = f(k_1, k_2)$ this yields $k_i' \ [\![K]\!]^{w_i} \ k_i'$ for $i = 1, 2$ and $w_1 \bullet w_2 = \{\!| k_1, k_2 |\!\}$. Inferring $w_i = \{\!| k_i' |\!\}$, we conclude $\{\!| k_1', k_2' |\!\} = \{\!| k_1, k_2 |\!\}$, leaving only the solutions $k_i' = k_i$ (identity) or $k_i' = k_{2-i}$ (swap). □

Small modifications of this proof show the impossibility of duplication or projection:

$$\nvdash t_d : K \multimap (K \times K)$$
$$\nvdash t_p : (K \times K) \multimap K$$

Applying the multiset semantics, we end up with absurd proof obligations like $\{\!| k |\!\} = \{\!| k_1, k_2 |\!\}$. □

We have shown these results for an abstract type K, but we can immediately generalise them to polymorphic types:

$$\vdash t \ : \forall\alpha.\ (\alpha \times \alpha) \multimap (\alpha \times \alpha) \text{ implies } (\!| t |\!) \in \{\mathsf{id}, \mathsf{swap}\}$$
$$\nvdash t_d : \forall\alpha.\ \alpha \multimap (\alpha \times \alpha)$$
$$\nvdash t_p : \forall\alpha.\ (\alpha \times \alpha) \multimap \alpha$$

The two applications of the fundamental theorem show just the tip of the iceberg. Many more "free" theorems wait to be discovered.

9 RELATED WORK

We have already extensively specific related systems in Section 4, and concentrate here on generalising work. The idea of generalising the structure of modalities to some ring-like structure can be traced to bounded linear logic [Girard et al. 1992]. This idea was then refined by Lago and Hofmann [2009] and made explicit by Ghica and Smith [2014], but, in all three cases the ring structure is only used to place an *upper bound* on resource usage. The observation that the ring structure can place more general constraints constraints is, to our knowledge, due to McBride [2016], who also combined dependent types into the mix. According to McBride, types consume no resources, and thus there is no constraint on the occurrences of variables bound by a type-former (such as Π).

Downstream, Atkey [2018] further refined the system and gave it categorical semantics, however this system appears to lack interest in the weakening rule WK, which is important for us.

The idea of further generalising the semantics comes from Atkey and Wood [2018], who suggest using a promonoidal category for the equivalent of our set of worlds W. As we see it, this means introducing a relation P generalising $w \leqslant w_1 \bullet w_2$ and a predicate J generalising $w \leqslant \varepsilon$ with a suitable axiomatisation expressing monotonicity, associativity, commutativity, and unit laws. The use of linear operators in the substitution lemma can also be attributed to Atkey and Wood [2019], but in later work.

Regardless, none of the above systems seems to aim at a maximal generality for the modality structure, whereas this is our goal. Brunel et al. [2014] propose the same ringoid structure as ours (additionally demanding a greatest element ∞). However, they leave out sum types, thus lacking the interaction between observability and case analysis. They offer an abstract machine interpretation, but it does not track modalities. Petricek et al. [2014] considers an even more generic structure (structured coeffects correspond to whole modality contexts). However they also present a specialised "flat" variant, which is closer to our ringoid. Yet it remains subtly different, requiring $p \wedge q \leqslant p + q$ instead of the monotonicity of $(+)$.

The present paper stands alone in the following respects. (1) It explicitly shows how the system subsumes several others. (2) It explores lesser trodden areas of semantics for modalities: a modality-preserving abstract machine and a modality-aware relational semantics, which implies irrelevance. (3) It leverages quantification over modalities, so that specialised systems can be constructed within the system, and consequently irrelevance works for any modality p above 1.

The work of Orchard et al. [2019] is perhaps one of the pieces of work nearest to ours, and as such deserves a detailed comparison. One of the main difference is that of focus: Orchard et al. describe a complete system, and thus focus more on user-facing features, such as a type-checker — which we do not present here. In contrast, we offer a more detailed meta-theory, including in particular a relational semantics. We also analyse the interaction between observability and case analysis (See also Section 10), which is not discussed by Orchard et al. [2019], even though their calculus Gr features patterns.

As we do, Orchard et al. aim at using modalities for several purposes, which they support by having builtin modality constructors and operations for several purposes (*Private* and *Public* for security applications, intervals for quantitative analyses, etc.). We argue here that these are not needed, because the user can quantify over modalities with constraints, which can be expressed within Λ^{p} (Example 7.11).

In addition to graded necessity, supporting co-effects, (corresponding to our qualified types), Gr also supports graded possibility[2], to support *effects*, such as IO. The relationship between the two is studied by Gaboardi et al. [2016], but we would prefer the approach taken by Bernardy et al. [2018], who use a double-negation encoding to encode possibility, keeping the calculus simpler.

10 DISCUSSION AND CONCLUSIONS

Constraint on case analysis. A non-forced design choice in our system is witnessed by the constraint $q \leqslant 1$ in the rule for case analysis. While all other choices (including the use of 0 and 1 in the variable rule, addition and multiplication in application in application and the use of ordering ($\leqslant$) in weakening) are necessary for (modality-)preservation to go through, in the substitution lemma and in the abstract machine, we could just as well remove the $q \leqslant 1$ with no consequence on this preservation.

To further analyse our design choice, let us imagine that we replace the constraint $q \leqslant 1$ by $q \leqslant \theta$, for some fixed modality θ. Then, recall (1) that the bit of information corresponding to which tag is present (inj_1 or inj_2) is accessible in the branches of a case analysis, and (2) that closed values of a closed (non-abstract) type can be constructed in empty contexts. Together (1) and (2) mean that the calculus would allow promotion of concrete data (in particular Booleans) from θ to any modality, by pattern matching:

[2] A terminology borrowed from alethic logic, see Section 4.3.3

$$
\begin{aligned}
&\mathit{promote} : \theta\langle \mathit{Bool}\rangle \multimap p\langle \mathit{Bool}\rangle \\
&\mathit{promote}\ [\mathit{true}] = [\mathit{true}] \\
&\mathit{promote}\ [\mathit{false}] = [\mathit{false}]
\end{aligned}
$$

(In contrast, $\lambda\,^{\theta}x.[^{p}x] : {}^{\theta}\alpha \to p\langle\alpha\rangle$ would be well-typed, for any abstract type α, only if $\theta \leqslant p$.) Thus θ is the modality of data which can be duplicated (for quantitative and sensitivity application) and revealed (for informational application).

If we let $\theta = 0$, then all (concrete) data becomes observable by the current program, and irrelevance (Theorem 7.10) no longer holds. In general if $1 \leqslant \theta$ (and $\theta \neq 1$) then the current program may not return $x : {}^{\theta}\mathrm{Bool}$ directly, but by case analysis can observe it, promote it and then return it, essentially bypassing the restriction. Thus we find that $1 \leqslant \theta$ should be ruled out as a design point.

If we have $\theta \leqslant 1$ (and $\theta \neq 1$), then we have a situation where the current program can return some variable $x : {}^{1}\mathrm{Bool}$, but it cannot itself observe it by case analysis. This choice is justified for systems where information must be strictly conserved (say quantum logic), and it does not violate non-interference. In fact our meta-theory is fully compatible with this choice. Letting $\theta = \bot$, the bottom of the lattice, may even appear the most natural choice, because it rules out any promotion. However it would mean that the payload (A_i) of a sum type $A_1 + A_2$ would also be required to have modality $\bot$, restricting the usefulness of sum types. To recover their flexibility, sum types would need to be modified and come at least with an extra modality annotation.

To avoid such complications, and following Girard [1987] who does allow the promotion $\mathit{Bool} \multimap\ !\mathit{Bool}$, we decided to simply let $\theta = 1$.

Regardless, with $\theta = 1$, one *can* qualify explicitly any bit of information with a modality p, by using the type $p\langle\mathrm{Bool}\rangle$. This bit will be only accessible to the current program if $p \leqslant 1$.

Relative versus absolute modalities. In several systems [Atkey 2018; McBride 2016] the typing judgement is annotated with a current modality. This has the benefit that modalities have an absolute, fixed meaning. On the other hand, typing is less compositional: whether a term is well-typed or not depends additionally in what context it occurs. In particular, in 0-context terms, no modality check happens. However, even relative modalities can be given an absolute meaning for evaluation, as our abstract machine shows. In this respect we drew inspiration from Bernardy et al. [2018], however their development is based on a big-step evaluation relation [Launchbury 1993] and a lot more involved than ours.

Extending to dependent types. Even though we exposed our ideas in the context of a relatively simple system (polymorphic lambda calculus), we are confident that they can be exported to related systems with dependent types [Barendregt 1992]. An open question is whether McBride's idea (allow arbitrary occurrences of variables in types) is fully compatible with our development.

Non-commutative modalities. Our metatheory does not require modality product to be commutative, but none of our examples leverages this generality. To our knowledge, the structure in not exploited yet in the literature. Non-commutativity could be useful to represent that several operations need to be performed in a specific order. This way, a particular protocol could be enforced using modalities. We leave this uncharted area for future work.

ACKNOWLEDGMENTS

We warmly thank anonymous reviewers for many insightful comments.

This material is based upon work supported by the Swedish Research Council (Vetenskapsrådet) under Grants No. 621-2014-4864 *Termination Certificates for Dependently-Typed Programs and Proofs* *via Refinement Types*, and No. 2014-39, which funds the Centre for Linguistic Theory and Studies in Probability (CLASP) in the Department of Philosophy, Linguistics, and Theory of Science at the University of Gothenburg.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM Press, 147–160. https://doi.org/10.1145/292540.292555

Maximilian Algehed. 2018. A Perspective on the Dependency Core Calculus. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 24–28. https://doi.org/10.1145/3264820.3264823

Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple noninterference from parametricity. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 89:1–89:22. https://doi.org/10.1145/3341693

Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. 2019. Optimising Faceted Secure Multi-Execution. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019.* 1–16. https://doi.org/10.1109/CSF.2019.00008

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 90:1–90:30. https://doi.org/10.1145/3236785

Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 9-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM Press, 56–65. https://doi.org/10.1145/3209108.3209189

Robert Atkey and James Wood. 2018. Context constrained computation. In *Third Workshop on Type-Driven Development (TyDe'18).* https://personal.cis.strath.ac.uk/james.wood.100/pub/context-constrained.pdf

Robert Atkey and James Wood. 2019. Linear metatheory via linear algebra. In *25th International Conference on Types for Proofs and Programs, TYPES 2019, Oslo, Norway, June 11-14, 2019, Book of Abstracts*, Marc Bezem (Ed.). http://www.ii.uib.no/~bezem/program.pdf

Hendrik Pieter Barendregt. 1992. Lambda calculi with types. *Handbook of logic in computer science* 2 (1992), 117–309. https://doi.org/10.1.1.26.4391

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning* 49, 2 (2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Corrado Böhm and Alessandro Berarducci. 1985. Automatic Synthesis of Typed Lambda-Programs on Term Algebras. *Theoretical Computer Science* 39 (1985), 135–154. https://doi.org/10.1016/0304-3975(85)90135-5

William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM Press, 101–113. https://doi.org/10.1145/2784731.2784733

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus, See [Shao 2014], 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now it's Getting Personal. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 69–81. https://doi.org/10.1145/2676726.2677005

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013.* 357–370. https://doi.org/10.1145/2429069.2429113

Marco Gaboardi, Shin-ya Katsumata, Dominic A. 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, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 476–489. https://doi.org/10.1145/2951913.2951939

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring, See [Shao 2014], 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

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 (1992), 1 – 66. https://doi.org/10.1016/0304-3975(92)90386-T

Ryu Hasegawa. 1994. Categorical Data Types in Parametric Polymorphism. *Mathematical Structures in Computer Science* 4, 1 (1994), 71–109. https://doi.org/10.1017/S0960129500000372

G. A. Kavvos. 2019. Modalities, cohesion, and information flow. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 20:1–20:29. https://doi.org/10.1145/3290333

Ugo Dal Lago and Martin Hofmann. 2009. Bounded Linear Logic, Revisited. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009, Proceedings (Lecture Notes in Computer Science)*, Pierre-Louis Curien (Ed.), Vol. 5608. Springer, 80–94. https://doi.org/10.1007/978-3-642-02273-9_8

Joachim Lambek. 1958. The mathematics of sentence structure. *Amer. Math. Monthly* 65, 3 (1958), 154–170. https://doi.org/10.1080/00029890.1958.11989160

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL*. 144–154.

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 (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Tom Murphy, Karl Crary, and Robert Harper. 2005. Distributed Control Flow with Classical Modal Logic. In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*. 51–69. https://doi.org/10.1007/11538363_6

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

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

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. IEEE Computer Society Press, 221–230. https://doi.org/10.1109/LICS.2001.932499

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

Jeff Polakow and Frank Pfenning. 1999. Natural Deduction for Intuitionistic Non-communicative Linear Logic. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science)*, Jean-Yves Girard (Ed.), Vol. 1581. Springer, 295–309. https://doi.org/10.1007/3-540-48959-2_21

Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM Press, 157–168. https://doi.org/10.1145/1863543.1863568

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

Zhong Shao (Ed.). 2014. *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Lecture Notes in Computer Science, Vol. 8410. Springer. https://doi.org/10.1007/978-3-642-54833-8

Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM Press, 115–125. https://doi.org/10.1145/1016850.1016868

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM Press, 347–359. https://doi.org/10.1145/99370.99404

David Walker. 2005. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 1.