April 1,2006

Toward a Universal Policy Logic
Version 0.1

ICS-16763-TR-07-003
SRI Project No. 16763
Contract No. FA8750-05-C-0230

Prepared by
Mark-Oliver Stehr
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025-3493

Prepared for
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

Acknowledgment and Disclaimer:
This material is based upon work supported by the Defense Advanced Projects Agency and the United State Air Force under Contract Number FA8750-05-C-0230.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Defense Advanced Research Projects Agency and the United States Air Force.

1 Introduction

Policies are becoming increasingly important in coping with the complexity of today's networked information systems. Policies are used in various domains such as security, power management, routing, and spectrum access. By guiding the behavior of a system, policies can lead to better adaptation to environmental conditions and users' goals, better coordination among system components and their environment, and ultimately to improved performance of the entire system.

Our approach favors the use of a *declarative policy language* that offers the promise of separating policies from low-level code with all associated advantages such as increased flexibility, improved maintainability, independent analysis, and accreditation. Policies at different layers can be expected to become increasingly widespread and intertwined (e.g., to capture crosslayer interactions), so that a common foundation for a *universal policy logic (UPL)* and an associated policy engine is an important next step.

To develop a *common notion of policies across various application domains* it is essential to first understand the key objective of what policies attempt to accomplish, namely, that policies are defining the space of permitted behaviors of a system. The strategic algorithms driving the dynamics and evolution of the system take into account applicable policies to remain within the space of permitted behaviors, but these algorithms do not have to be part of the policy itself. In a declarative approach, the permitted behavior would correspond to solutions/models of a logical specification that represents the set of applicable policies. Since the space of possible solutions is infinite in most application domains, it is essential that the use of policies is not restricted to checking candidate solutions. Instead, a policy engine needs to be capable of returning an infinite set of solutions, for example, by using a finitary symbolic representation. In addition, richer interactions with the algorithmic strategies are needed so that the set of possible solutions can be narrowed down incrementally.

This paper presents the logical foundations for a policy reasoner based on a *typed first-order logic*. This logic does not necessarily coincide with the language that is used to express policies, although from our experience a large variety of policies can be naturally represented in a typed first-order logic. Typically, policies are subject to various syntactic constraints, but to capture the entire picture of policy processing, their interaction with queries,

and the reasoning process it is useful to consider the full logic with minimal syntactic restrictions. In general, we envision that a *semantics-preserving representation mapping* can be defined from a policy language with user-friendly syntax into the typed first-order logic used in this paper. Because of expressiveness of first-order logic, the proof system needs to be part of a well-defined operational semantics, which relies on the enrichment of the logic with information that can guide the reasoning process. Hence, certain forms of first-order formulas that can be equipped with a natural operational semantics will be syntactically promoted to rules, whereas other first-order logic formulas will be used as conditions of those rules, which can be in turn processed by the very same operational semantics.

One design goal behind our universal policy logic is to offer at least the expressiveness of *description logic-based languages*, such as OWL-DL, which are used in the semantic web community and commonly used to describe ontologies in various domains. Although standard first-order logic is sufficiently expressive in principle, we have added two concepts to support a representation that is not only very direct but also useful for computation and reasoning. These concepts are (1) *(finite) cardinality construct* that allows us to explicitly refer to the number of objects with a given property, and (2) a *description construct* that gives us an explicit notation for an object with certain properties.

2 Preliminaries

For the syntax and semantics of the language we assume a set of *booleans* $\mathbb{B} = \{\mathbb{T}, \mathbb{F}\}$ and a standard set-theoretic number hierarchy $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ of *integers*, *rationals*, *reals*, and *complex numbers*.

3 Syntax

Assuming infinite sets of *type variables* (written C or X, Y, Z), *(ordinary) variables* (written c, f, g, h or x, y, z), and *propositional variables* (written p, q), we define the syntactic categories of types, terms, formulas, and rules as follows. We should point out that the three sets of variables do not have to be distinct, but the assumption of disjoint sets is consistent with our

formalization and helps to improve readability.

The three syntactic categories of types, terms, and formulas have the following inductive definition. Reflecting the possible dependency of terms on formulas, the definition is mutually inductive for these two categories. The syntactic category of rules is then defined on top of all of these.

3.1 Types

Simple types (written S, S_1, S_2, S' etc.) are inductively defined as follows. A simple type can be a type variable C, an *atomic data type* `Data`, a *product type* $S_1 * S_2$, a *list type* $[S]$, an *empty list type* $[]$, a *(finite) set type* $\{S\}$, or an *empty set type* $\{\}$. *Types* (written T, T_1, T_2, T' etc.) are defined as follows. A type can be a *propositional type* `Prop`, a *predicate type* $S \to$ `Prop`, or a *function type* $S_1 \to S_2$.

3.2 Terms

Terms (written M, N, M', N' etc.) are inductively defined as follows. A term can be an ordinary variable x, a *function application* $f\ M$, a *pair construction* (M, N), a *boolean constant* $b \in \mathbb{B}$, a *rational constant* $r \in \mathbb{Q}$ (which can be an *integer*), an *operator application* $uop\ M$ and $M\ bop\ N$ for $uop \in \{-, \texttt{sqrt}\}$ and $bop \in \{+, -, *, /, \texttt{div}, \texttt{mod}\}$, the *empty list constant* $[]$, a *singleton list construct* $[M]$, a *list concatenation construct* $M \mid_l N$, the *empty set constant* $\{\}$, a *singleton set construct* $\{M\}$, a *set union construct* $M \mid_s N$, a *description construct*[1] $(\epsilon x : S)P$, and a *(finite) cardinality construct* $(\kappa x : T)P$, where P ranges over formulas. The last two constructs *bind* x in P.

3.3 Formulas

An *atomic formula* can be a propositional variable p, a *predicate application* $p\ M$ (which we distinguish from the following built-in sort constraints), a *boolean sort constraint* `Bool` M, an *integer sort constraint* `Int` M, a *real sort constraint* `Real` M, an *equality constraint* $M \equiv N$, a *non-equality constraint* $M \not\equiv N$, a *strict ordering constraint* $M < N$, a *non-strict ordering constraint*

[1] also known as Hilbert's ϵ-operator

$M \leq N$, a *set membership constraint* $M \in_s N$, or a *list membership constraint* $M \in_l N$.

Formulas (written P, Q, P', Q' etc.) are inductively defined as follows. A formula can be an *atomic formula*, a *negation* `not` P, a *conjunction* P `and` Q, a *disjunction* P `or` Q, a *universally quantified formula* $(\forall \bar{x} : \bar{S})P$, or an *existentially quantified formula* $(\exists \bar{x} : \bar{S})P$, where $\bar{x} : \bar{S}$ stands for a set of *binders* $\bar{x}_1 : \bar{S}_1, \ldots, \bar{x}_n : \bar{S}_n$. Quantifiers *bind* these variables in P. Introducing syntactic sugar for formulas, we also define the *implication* P `implies` Q as `not` P `or` Q, and the *equivalence* P `iff` Q as (P `implies` Q) `and` (Q `implies` P).

3.4 Rules

(Logical) rules are defined as follows, assuming that H and Q range over formulas. A logical rule can be

1. an *equational rule* $(\forall \bar{x} : \bar{S})M = M' \Leftarrow Q$,
2. an *equivalence rule* $(\forall \bar{x} : \bar{S})H \Leftrightarrow P \Leftarrow Q$,
3. a *forward rule* $(\forall \bar{x} : \bar{S})Q \Rightarrow H$, or
4. a *backward rule* $(\forall \bar{x} : \bar{S})H \Leftarrow Q$.

A logical rule or a formula is said to be a *sentence*.

Model-theoretically (see Section 6.4), rules will be regarded as first-order formulas (involving equality, equivalences, and implications as defined above), but in the proof system and hence the operational semantics, rules and formulas have different capabilities. In brief, the operational semantics of equational and equivalence rules is based on (conditional) rewriting, whereas the operational semantics of forward and backward rules is based on forward and backward chaining, respectively.

3.5 Contexts

A *context*, written Γ, is a set of type declarations, type definitions, subtyping declarations, ordinary declarations, and ordinary definitions, as defined

below. The empty context is written as $\emptyset$. The set-theoretic union $\Gamma \cup \Gamma'$ of two contexts Γ and Γ' is also written as Γ, Γ'.

A *type declaration* is of the form C : `Type` and said to *declare* the type variable C as a type. A *type definition* is of the form $C = S$ and said to *define* the type variable C (which is not declared by this construct) to be equal to S. A *(ordinary) declaration* is of the form $c : S$ and said to *declare* the variable c to be of type S. An *inductive type declaration* is of the form $\texttt{ind}(C)$, an *inductive propositional declaration* is of the form $\texttt{ind}(p)$, and a *constructor declaration* is of the form $\texttt{ctor}(x)$. In these cases we say that C is an *inductive type variable*, p is an *inductive propositional variable*, and x is a *constructor variable*, respectively.

3.6 Variable Conventions

Following standard conventions, we identify types, terms, formulas, rules, and contexts that are equal modulo renaming of bound variables. The *free variables* of a type T, a term M, a sentence ϕ, or a context Γ are written as $FV(T)$, $FV(M)$, $FV(\phi)$, $FV(\Gamma)$, respectively, and defined as the variables associated with occurrences that are not bound. The set of *declared variables* of a context Γ is a subset of $FV(\Gamma)$ and written as $V(\Gamma)$.

3.7 Capture-free Substitution

We furthermore introduce a standard capture-free notion of substitution on the syntactic categories of types, terms, and formulas. A *type substitution*, written $[C := S]$, is defined as the operation of replacing each non-declaring and non-defining occurrence of type variable C by a type S. A *substitution*, written $[c := M]$, stands for the operation of replacing each non-declaring and non-defining occurrence of an ordinary variable c by a term M. Substitutions can be applied to all syntactic categories, and also to contexts and sets of sentences by lifting the definition in the obvious way.

3.8 Placeholders and Substitution

We extend the syntax of terms, formulas, and rules by a distinct placeholder symbol $\bullet$, called a *hole*, for all three syntactic categories. Using U and U'

to uniformly range over rules, formulas, and terms, we have the following rules. We also use the notation $U[\bullet]$ instead of U to express that U contains a hole and $U[U']$ to denote the U with the hole replaced by U'. Whenever we use this notation, we make the implicit assumption that U contains a single unique hole. We furthermore introduce $\Delta[U']$, which is the obvious lifting to this notion to an arbitrary set Δ of sentences. Capturing of free variables of U' is possible with this notation so that certain potential restrictions in this regard need to be explicitly stated.

We also use the notation $U \sqsubseteq U'$ to express U is a subterm or a subformula of U, which needs to be defined in the standard way such that capturing of the free variables of U by surrounding quantifiers in U' is avoided.

4 Type System

To define well-typed contexts, types, terms, and formulas, we present a type system, that is, a mutually inductive definition of derivable *typing judgments* of the form $\Gamma \vdash M : T$, $\Gamma \vdash P : \texttt{Prop}$, or $\Gamma \vdash R : \texttt{Rule}$ and *well-typed context judgments* of the form $\Gamma \vdash$.

A context Γ is said to be *well-typed* iff $\Gamma \vdash$ is derivable. Let Γ be a well-typed context. A type T is said to be *well-typed* under Γ iff $\Gamma \vdash T : \texttt{Type}$ is derivable. A term M is said to be well-typed and of type T iff $\Gamma \vdash M : T$ is derivable. A formula P is said to be *well-typed* iff $\Gamma \vdash P : \texttt{Prop}$ is derivable. A rule R is said to be *well-typed* iff $\Gamma \vdash R : \texttt{Rule}$ is derivable.

The type system is designed to associate with each term a unique type, the only trivial exception being empty lists and sets that can have multiple types. However, we can consider the empty list and set types as their *canonical types*, so that each term has a unique canonical type.

4.1 Typing Types

Well-typed types are defined by the following rules. Recall that here and in the following S,S_1,S_2,S',S'' etc. range over simple types, whereas T,T_1,T_2,T',T'' range over arbitrary types.

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Data} : \texttt{Type}} \quad \text{(DataType)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Prop} : \texttt{Type}} \quad \text{(PropType)}$$

$$\frac{\Gamma \vdash S : \texttt{Type}}{\Gamma \vdash S \to \texttt{Prop} : \texttt{Type}} \quad \text{(PredType)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash C : \texttt{Type}} \text{ if } C : \texttt{Type} \in \Gamma \quad \text{(ConstType)}$$

$$\frac{\Gamma \vdash S_1 : \texttt{Type} \quad \Gamma \vdash S_2 : \texttt{Type}}{\Gamma \vdash S_1 * S_2 : \texttt{Type}} \quad \text{(ProdType)}$$

$$\frac{\Gamma \vdash S_1 : \texttt{Type} \quad \Gamma \vdash S_2 : \texttt{Type}}{\Gamma \vdash S_1 \to S_2 : \texttt{Type}} \quad \text{(FunType)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash [] : \texttt{Type}} \quad \text{(ListType)}$$

$$\frac{\Gamma \vdash S : \texttt{Type}}{\Gamma \vdash [S] : \texttt{Type}} \quad \text{(ListType')}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \{\} : \texttt{Type}} \quad \text{(SetType)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \{S\} : \texttt{Type}} \quad \text{(SetType')}$$

4.2 Typing Terms

Well-typed terms are generated by the following inference rules together with the subsumption rule that will be given later to reflect closure under subtyping.

$$\frac{\Gamma \vdash}{\Gamma \vdash x : T} \text{ if } x : T \in \Gamma \quad \text{(Var)}$$

$$\frac{\Gamma \vdash f : S_1 \to S_2 \quad \Gamma \vdash M : S_1}{\Gamma \vdash f\ M : S_2} \quad \text{(App)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash b : \texttt{Data}} \text{ if } b \in \mathbb{B} \quad \text{(Bool)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash r : \texttt{Data}} \text{ if } r \in \mathbb{Q} \quad \text{(Num)}$$

$$\frac{\Gamma \vdash M : \texttt{Data}}{\Gamma \vdash uop\, M : \texttt{Data}} \quad \text{(UOp)}$$

$$\frac{\Gamma \vdash M : \texttt{Data} \quad \Gamma \vdash N : \texttt{Data}}{\Gamma \vdash M\, bop\, N : \texttt{Data}} \quad \text{(BOp)}$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : S'}{\Gamma \vdash (M, N) : (S, S')} \quad \text{(Pair)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash [] : []} \quad \text{(EmptyList)}$$

$$\frac{\Gamma \vdash S : \texttt{Type}}{\Gamma \vdash [] : [S]} \quad \text{(EmptyList')}$$

$$\frac{\Gamma \vdash S : \texttt{Type} \quad \Gamma \vdash M : S}{\Gamma \vdash [M] : [S]} \quad \text{(SingleList)}$$

$$\frac{\Gamma \vdash M : [] \quad \Gamma \vdash N : []}{\Gamma \vdash M \mid_l N : []} \quad \text{(ConcatList)}$$

$$\frac{\Gamma \vdash M : [S] \quad \Gamma \vdash N : [S]}{\Gamma \vdash M \mid_l N : [S]} \quad \text{(ConcatList')}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \{\} : \{\}} \quad \text{(EmptySet)}$$

$$\frac{\Gamma \vdash S : \texttt{Type}}{\Gamma \vdash \{\} : \{S\}} \quad \text{(EmptySet')}$$

$$\frac{\Gamma \vdash S : \texttt{Type} \quad \Gamma \vdash M : S}{\Gamma \vdash \{M\} : \{S\}} \quad \text{(SingleSet)}$$

$$\frac{\Gamma \vdash M : \{S\} \quad \Gamma \vdash N : \{S\}}{\Gamma \vdash M \mid_s N : \{S\}} \quad \text{(ConcatSet)}$$

$$\frac{\Gamma \vdash M : \{\} \quad \Gamma \vdash N : \{\}}{\Gamma \vdash M \mid_s N : \{\}} \quad \text{(ConcatSet')}$$

$$\frac{\Gamma, x : S \vdash P : S \to \texttt{Prop}}{\Gamma \vdash (\epsilon x : S) P : S} \quad \text{(Desc)}$$

$$\frac{\Gamma, x : S \vdash P : S \to \texttt{Prop}}{\Gamma \vdash (\kappa x : S)P : \texttt{Data}} \quad \text{(Card)}$$

4.3 Typing Formulas

First-order formulas with quantification restricted to simple types are equipped with the type `Prop` by means of the following rules.

$$\frac{\Gamma \vdash}{\Gamma \vdash p : T} \text{ if } p : T \in \Gamma \quad \text{(PropVar)}$$

$$\frac{\Gamma \vdash p : S \to \texttt{Prop} \quad \Gamma \vdash M : S}{\Gamma \vdash p\ M : \texttt{Prop}} \quad \text{(PredApp)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Bool}\ M : \texttt{Prop}} \quad \text{(Bool)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Int}\ M : \texttt{Prop}} \quad \text{(Int)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Rat}\ M : \texttt{Prop}} \quad \text{(Rat)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Real}\ M : \texttt{Prop}} \quad \text{(Real)}$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : S}{\Gamma \vdash (M \equiv N) : \texttt{Prop}} \quad \text{(Equal)}$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : S}{\Gamma \vdash (M \not\equiv N) : \texttt{Prop}} \quad \text{(NotEqual)}$$

$$\frac{\Gamma \vdash M : \texttt{Data} \quad \Gamma \vdash N : \texttt{Data}}{\Gamma \vdash (M < N) : \texttt{Prop}} \quad \text{(Less)}$$

$$\frac{\Gamma \vdash M : \texttt{Data} \quad \Gamma \vdash N : \texttt{Data}}{\Gamma \vdash (M \leq N) : \texttt{Prop}} \quad \text{(LessEq)}$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : \{S\}}{\Gamma \vdash (M \in_s N) : \texttt{Prop}} \quad \text{(InSet)}$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : [S]}{\Gamma \vdash (M \in_l N) : \texttt{Prop}} \quad \text{(InList)}$$

$$\frac{\Gamma \vdash P : \texttt{Prop}}{\Gamma \vdash (\texttt{not}\ P) : \texttt{Prop}} \quad \text{(Neg)}$$

$$\frac{\Gamma \vdash P : \texttt{Prop} \quad \Gamma \vdash Q : \texttt{Prop}}{\Gamma \vdash (P\ \texttt{and}\ Q) : \texttt{Prop}} \quad \text{(And)}$$

$$\frac{\Gamma \vdash P : \texttt{Prop} \quad \Gamma \vdash Q : \texttt{Prop}}{\Gamma \vdash (P\ \texttt{or}\ Q) : \texttt{Prop}} \quad \text{(Or)}$$

$$\frac{\Gamma, \bar{x} : \bar{S} \vdash P : \texttt{Prop}}{\Gamma \vdash (\forall \bar{x} : \bar{S}) P : \texttt{Prop}} \quad \text{(Forall)}$$

$$\frac{\Gamma, \bar{x} : \bar{S} \vdash P : \texttt{Prop}}{\Gamma \vdash (\exists \bar{x} : \bar{S}) P : \texttt{Prop}} \quad \text{(Exists)}$$

4.4 Typing Rules

Based on well-typed formulas, we define well-typed logical rules as follows:

$$\frac{\Gamma, \bar{x} : \bar{S} \vdash M : S \quad \Gamma, \bar{x} : \bar{S} \vdash N : S \quad \Gamma, \bar{x} : \bar{S} \vdash Q : \texttt{Prop}}{\Gamma \vdash (\forall \bar{x} : \bar{S})(M = N \Leftarrow Q) : \texttt{Rule}} \quad \text{(EqRule)}$$

$$\frac{\Gamma, \bar{x} : \bar{S} \vdash P : \texttt{Prop} \quad \Gamma, \bar{x} : \bar{S} \vdash P' : \texttt{Prop} \quad \Gamma, \bar{x} : \bar{S} \vdash Q : \texttt{Prop}}{\Gamma \vdash (\forall \bar{x} : \bar{S})(P \Leftrightarrow P' \Leftarrow Q) : \texttt{Rule}} \quad \text{(IffRule)}$$

$$\frac{\Gamma, \bar{x} : \bar{S} \vdash Q : \texttt{Prop} \quad \Gamma, \bar{x} : \bar{S} \vdash H : \texttt{Prop}}{\Gamma \vdash (\forall \bar{x} : \bar{S})(Q \Rightarrow H) : \texttt{Rule}} \quad \text{(FwRule)}$$

$$\frac{\Gamma, \bar{x} : \bar{S} \vdash H : \texttt{Prop} \quad \Gamma, \bar{x} : \bar{S} \vdash Q : \texttt{Prop}}{\Gamma \vdash (\forall \bar{x} : \bar{S})(H \Leftarrow Q) : \texttt{Rule}} \quad \text{(BwRule)}$$

4.5 Typing Contexts

Well-typed contexts are generated by the following inference rules. Note that in a well-typed context variables can be defined at most once but they can be declared more than once (i.e., overloaded) if their declarations are consistent.

$$\emptyset \vdash \quad \text{(EmptyCtxt)}$$

$$\frac{\Gamma \vdash}{\Gamma, C : \texttt{Type} \vdash} \qquad \text{(TypeDecl)}$$

$$\frac{\Gamma \vdash}{\Gamma, p : \texttt{Prop} \vdash} \quad p \notin V(\Gamma) \qquad \text{(PropDecl)}$$

$$\frac{\Gamma \vdash S \to \texttt{Prop} : \texttt{Type}}{\Gamma, p : S \to \texttt{Prop} \vdash} \quad p \notin V(\Gamma) \qquad \text{(PredDecl)}$$

$$\frac{\Gamma \vdash S : \texttt{Type}}{\Gamma, c : S \vdash} \quad c \notin V(\Gamma) \qquad \text{(ConstDecl)}$$

$$\frac{\Gamma \vdash S_1 \to S_2 : \texttt{Type}}{\Gamma, f : S_1 \to S_2 \vdash} \quad f \notin V(\Gamma) \qquad \text{(FunDecl)}$$

$$\frac{\Gamma \vdash S : \texttt{Type}}{\Gamma, C = S \vdash} \quad C \text{ is not defined in } \Gamma \qquad \text{(TypeDef)}$$

$$\frac{\Gamma \vdash C : \texttt{Type}}{\Gamma, \texttt{ind}(C) \vdash} \qquad \text{(IndTypeDecl)}$$

$$\frac{\Gamma \vdash p : \texttt{Prop}}{\Gamma, \texttt{ind}(p) \vdash} \qquad \text{(IndPropDecl)}$$

$$\frac{\Gamma \vdash p : S \to \texttt{Prop}}{\Gamma, \texttt{ind}(p) \vdash} \qquad \text{(IndPredDecl)}$$

$$\frac{\Gamma \vdash c : C}{\Gamma, \texttt{ctor}(c) \vdash} \qquad \text{(CtorConstDecl)}$$

$$\frac{\Gamma \vdash f : S \to C}{\Gamma, \texttt{ctor}(f) \vdash} \qquad \text{(CtorFunDecl)}$$

4.6 Eliminating Type Definitions

Type definitions can be eliminated by simple substitution as captured by the following two rules:

$$\frac{\Gamma, C = S \vdash \quad \Gamma, [C := S]\Gamma' \vdash}{\Gamma, C = S, \Gamma' \vdash} \quad \text{if } C \notin FV(\Gamma) \qquad \text{(ElimTypeDef)}$$

$$\frac{\Gamma, C = S \vdash \quad \Gamma, [C := S]\Gamma' \vdash [C := S]J}{\Gamma, C = S, \Gamma' \vdash J} \quad \text{if } C \notin FV(\Gamma) \qquad \text{(ElimTypeDef')}$$

5 Policies, Extension, and Composition

A *policy* (Γ, Δ) consists of a well-typed context Γ and a set Δ of well-typed sentences over Γ.

A context Γ' is an *extension* of Γ, written $\Gamma \subseteq \Gamma'$ iff each element of Γ' is an element of Γ. A policy (Γ', Δ') is an *extension* of a policy (Γ, Δ) iff $\Gamma \subseteq \Gamma'$ and $\Delta \subseteq \Delta'$.

Two policies (Γ, Δ) and (Γ', Δ') are *composable* iff the context $\Gamma \cup \Gamma'$ is well-typed. The *composition* of two such policies is defined as the policy $((\Gamma \cup \Gamma), (\Delta \cup \Delta'))$. Composition naturally extends to any number of policies.

A *policy base*, written $\mathcal{B}$, is a set of policies that are pairwise composable. By composition we can *associate* with each policy base $\mathcal{B}$ a policy $(\Gamma_{\mathcal{B}}, \Delta_{\mathcal{B}})$ where $\Gamma_{\mathcal{B}} = \bigcup\{\Gamma \mid (\Gamma, \Delta) \in \mathcal{B} \text{ for some } \Delta\}$ and $\Delta_{\mathcal{B}} = \bigcup\{\Delta \mid (\Gamma, \Delta) \in \mathcal{B} \text{ for some } \Gamma\}$.

6 Model-Theoretic Semantics

The importance of the model-theoretic semantics lies in its simplicity that allows us to understand the meaning of a policy without the need to understand the reasoning process. The use of an expressive set-theoretic semantics enables us to extend the language in the future without fundamentally changing its underlying semantics. Another advantage of using a set-theoretic semantics is that it provides a notion of soundness for the proof system that is easy to verify, and hence is a key for accreditation of a reasoner. Completeness is not required and will never be achieved for the highly expressive policy language that we envision, but progress in reasoning technology can be expected to lead to gradual improvements in this direction. In summary, the model-theoretic semantics enables future extensions of the proof system and hence the reasoner, without being itself subject to changes.

A *interpretation* $\mathcal{I} = (D, D_{\texttt{Data}}, \prec, \bot, I)$ consists of an *interpretation domain* D, which is a set-theoretic universe (i.e., a set with the standard set-theoretic closure properties); a *data subdomain* $D_{\texttt{Data}} \subseteq D$, which includes at least the set-theoretic booleans $\mathbb{B}$ as well as the standard set-theoretic number hierarchy $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ of integers, rationals, reals, and complex numbers; a well-ordering $\prec$ on D; a distinguished *error element* $\bot \notin D$; and a partial

interpretation function I of type variables, ordinary variables, and propositional variables into D. An *assignment* for $\mathcal{I}$ consists of a partial function β of ordinary variables into D.

Given an interpretation $\mathcal{I}$ and an assignment β, the *model-theoretic semantics* is defined by an *interpretation of types* (written $[\![T]\!]_{\mathcal{I}}$), an *interpretation of terms* (written $[\![M]\!]_{\mathcal{I},\beta}$), and *a notion of validity* of formulas (written $\mathcal{I}, \beta \models P$) all subject to the following conditions. For better readability we leave $\mathcal{I}$ and β implicit whenever possible.

6.1 Interpretation of Types

The interpretation $[\![T]\!]_{\mathcal{I}}$ of a type T under $\mathcal{I}$ is inductively defined as follows:

1. The type `Data` is interpreted as $D_{\texttt{Data}}$.
2. A type variable C is interpreted as $I(C)$.
3. A product $S_1 * S_2$ is interpreted as a cartesian product $[\![S_1]\!] \times [\![S_2]\!]$.
4. A list type [] is interpreted as the set containing the empty sequence.
5. A list type $[S]$ is interpreted as the set of finite sequences over $[\![S]\!]$.
6. A set type {} is interpreted as the set containing the empty set.
7. A set type $\{S\}$ is interpreted as the set of finite sets over $[\![S]\!]$.
8. A propositional type `Prop` is interpreted as the powerset of $\{()\}$, where () stands for the empty tuple.
9. A predicate type $S \to$ `Prop` is interpreted as the powerset of $[\![S]\!]$.
10. A function type $S_1 \to S_2$ is interpreted as the set of total functions from $[\![S_1]\!]$ to $[\![S_2]\!]$.

Note that a set type is semantically equivalent to a predicate type.

6.2 Interpretation of Terms

The interpretation $[\![M]\!]_{\mathcal{I},\beta}$ of a term M under $\mathcal{I}$ and β is inductively defined. A term M is interpreted as $\bot$ if it contains a subterm M' such that $[\![M']\!] = \bot$. Otherwise, $[\![M]\!]$ is interpreted as follows and undefined in all other cases.

1. A variable x is interpreted as $\beta(x)$ if $\beta(x)$ is defined.
2. A variable x is interpreted as $I(x)$ if $\beta(x)$ is undefined.
3. A boolean constant $b \in \mathbb{B}$ is interpreted as b.
4. A rational constant $r \in \mathbb{Q}$ (and in particular an integer) is interpreted as r.
5. A function application $f\ M$ is interpreted as $I(f)([\![M]\!])$.
6. A term (M, N) is interpreted as a pair $([\![M]\!], [\![N]\!])$.
7. A term [] is interpreted as the empty sequence.
8. A term $[M]$ is interpreted as the sequence containing a single element $[\![M]\!]$.
9. A term $M \mid_l N$ is interpreted as the concatenation of $[\![M]\!]$ and $[\![N]\!]$ if $[\![M]\!]$ and $[\![N]\!]$ are sequences.
10. A term $\{\}$ is interpreted as the empty set.
11. A term $\{M\}$ is interpreted as the set containing a single element $[\![M]\!]$.
12. A term $M \mid_s N$ is interpreted as the union of $[\![M]\!]$ and $[\![N]\!]$ if $[\![M]\!]$ and $[\![N]\!]$ are finite sets.
13. A term *uop* N is interpreted as $\widehat{uop}\ [\![N]\!]$ if this is defined and as $\bot$ otherwise. Here, $\widehat{uop}$ is the set-theoretic counterpart of *uop*, in particular $\widehat{\texttt{sqrt}}$ is the set-theoretic square root operation on $\mathbb{C}$.
14. A term M *bop* N is interpreted as $[\![M]\!]\ \widehat{bop}\ [\![N]\!]$ if this is defined and $\bot$ otherwise. Here $\widehat{bop}$ is the set-theoretic counterpart of *bop*; in particular $\widehat{\texttt{div}}$ is the integer division (with rounding toward 0) and $\widehat{\texttt{mod}}$ is the modulo operation on integers.

15. A description construct $(\epsilon x : S)P$ is interpreted as the unique $u \in [\![S]\!]$ such that P is valid under the assignment $\beta' = [x := u]\beta$ and u is minimal w.r.t. the well-ordering $\prec$ on D. If such a u does not exist, $(\epsilon x : S)P$ is interpreted as $\bot$.

16. A cardinality construct $(\kappa x : S)P$ is interpreted as the cardinality (as a natural number) of the set of all $u \in [\![S]\!]$ such that P is valid under the assignment $\beta' = [x := u]\beta$. If this set is infinite, $(\kappa x : S)P$ is interpreted as $\bot$.

Above we use the notation $[x := u]\beta$ to denote the (partial) function β', which is identical to β except that $\beta'(x) = u$.

6.3 Validity of Formulas

The validity of a formula P under $\mathcal{I}$ and β, written $\mathcal{I}, \beta \models P$, is inductively defined. It can hold only if P does not contain a subterm M' such that $[\![M']\!] = \bot$ and is then defined by the following cases:

1. A sort constraint `Bool` M is valid iff $[\![M]\!] \in \mathbb{B}$.

2. A sort constraint `Int` M is valid iff $[\![M]\!] \in \mathbb{Z}$.

3. A sort constraint `Rat` M is valid iff $[\![M]\!] \in \mathbb{Q}$.

4. A sort constraint `Real` M is valid iff $[\![M]\!] \in \mathbb{R}$.

5. An equality constraint $M \equiv N$ is valid iff $[\![M]\!] = [\![N]\!]$.

6. A non-equality constraint $M \not\equiv N$ is valid iff $[\![M]\!] \neq [\![N]\!]$.

7. An ordering constraint $M < N$ is valid iff $[\![M]\!] < [\![N]\!]$, where $<$ is the standard strict ordering on $\mathbb{R}$.

8. An ordering constraint $M \leq N$ is valid iff $[\![M]\!] \leq [\![N]\!]$, where $\leq$ is the standard non-strict ordering on $\mathbb{R}$.

9. A membership constraint $M \in_S N$ is valid iff $[\![M]\!]$ is an element of the set $[\![N]\!]$.

10. A membership constraint $M \in_l N$ is valid iff $[\![M]\!]$ is an element of the sequence $[\![N]\!]$.

11. A propositional variable p valid iff $() \in [\![p]\!]$, where $()$ stands for the empty tuple.

12. A predicate application $p\ M$ is valid iff $[\![M]\!] \in [\![p]\!]$.

13. A formula $\texttt{not}\ P$ is valid iff P is not valid.

14. A formula P and Q is valid iff P is valid and Q is valid.

15. A formula P or Q is valid iff P is valid or Q is valid.

16. A quantified formula $(\forall \bar{x} : \bar{S})P$ is valid under an assignment β iff P is valid under all assignments $\beta' = [x := u]\beta$ and $u \in [\![S]\!]$.

17. A quantified formula $(\exists \bar{x} : \bar{S})P$ is valid under an assignment β iff P is valid under some assignment $\beta' = [x := u]\beta$ and $u \in [\![S]\!]$.

6.4 Validity of Logical Rules

The validity of a logical rule R under I and β, written $I, \beta \models R$, is defined by $\mathcal{I}, \beta, \models \tilde{R}$ where $\tilde{R}$ is the formula *corresponding* to R defined as follows:

1. The formula corresponding to an equational rule $(\forall \bar{x} : \bar{S})M = N \Leftarrow Q$ is $(\forall \bar{x} : \bar{S})Q\ \texttt{implies}\ M \equiv N$.

2. The formula corresponding to an equivalence rule $(\forall \bar{x} : \bar{S})H \Leftrightarrow P \Leftarrow Q$ is $(\forall \bar{x} : \bar{S})H\ \texttt{iff}\ Q\ \texttt{implies}\ P$

3. The formula corresponding to a forward rule $(\forall \bar{x} : \bar{S})Q \Rightarrow H$ is $(\forall \bar{x} : \bar{S})Q\ \texttt{implies}\ H$.

4. The formula corresponding to a backward rule $(\forall \bar{x} : \bar{S})H \Leftarrow Q$ is $(\forall \bar{x} : \bar{S})Q\ \texttt{implies}\ H$.

We also define *validity* of a set of sentences Δ under $\mathcal{I}$, written $\mathcal{I} \models \Delta$, by requiring that $\mathcal{I} \models \phi$ for each $\phi \in \Delta$.

6.5 Validity of Contexts

A context element γ is *valid* under $\mathcal{I}$, written $\mathcal{I} \models \gamma$ as follows:

1. A type declaration $C : \texttt{Type}$ is valid iff $[\![C]\!] \in D$.
2. A type definition $C = T$ is valid iff $[\![C]\!] = [\![T]\!]$.
3. An (ordinary) declaration $c : T$ is valid iff $[\![c]\!] \in [\![T]\!]$.

We define *validity* of a context Γ under $\mathcal{I}$, written $\mathcal{I} \models \Gamma$, by requiring that $\mathcal{I} \models \gamma$ for each $\gamma \in \Gamma$ that is not an inductive or a constructor declaration and in addition that $\mathcal{I}$ satisfies the following *constructor constraints*:

1. For all constructors $c : S$ and $c' : S'$ in Γ we require $[\![c]\!] \neq [\![c']\!]$ if $c \neq c'$.
2. For all constructors $c : S, f : S'_1 \to S'_2$ in Γ we require $[\![c]\!] \neq [\![f]\!](x)$ if $c \neq f$ for all x in the domain of f.
3. For all constructors $f : S_1 \to S_2$ and $f' : S'_1 \to S'_2$ in Γ we require $[\![f]\!](x) \neq [\![f']\!](y)$ if $f \neq f'$ for all x, y in the domain of f and f', respectively.
4. For each constructor $f : S_1 \to S_2$ in Γ we require $[\![f]\!](x) \neq [\![f]\!](y)$ if $x \neq y$ for all x, y in the domain of f.

6.6 Validity of Typing Judgments

Validity of a *judgment* $\Gamma \vdash J$, written $\Gamma \models J$, is defined as follows:

1. A typing judgment $\Gamma \vdash T : \texttt{Type}$ is valid iff $[\![T]\!] \in D$ for all models $\mathcal{I}$ of Γ.
2. A typing judgment $\Gamma \vdash M : T$ is valid iff $[\![M]\!] \in [\![T]\!] \cup \{\bot\}$ and $[\![T]\!] \in D$ for all models $\mathcal{I}$ of Γ.

6.7 Semantics of Policies and Composition

The semantics of a single policy is open ended because it can be extended and more generally composed with other policies, which ultimately can have an impact on the meaning of the former policy. Hence, the semantics of a single policy is given by the class of all models that are consistent with it. Once a policy base has been built by composing a suitable set of policies, we consider the set of policies as closed, and hence the policy base can be equipped with a stronger semantics that is minimal in the sense that inductive types and predicates are interpreted in the most restrictive way (generalized closed-world assumption).

We define *validity* of a policy (Γ, Δ) under the interpretation $\mathcal{I}$, written $\mathcal{I} \models (\Gamma, \Delta)$, by requiring $\mathcal{I} \models \Gamma$ and $\mathcal{I} \models \Delta$. In this case we also say that $\mathcal{I}$ is a *model* of the policy (Γ, Δ). The *semantics of a policy* is given by the class of all its models. A policy without a model is said to be *inconsistent*.

We define *prevalidity* of a policy base $\mathcal{B}$ under the interpretation $\mathcal{I}$, written $\mathcal{I} \models^{pre} \mathcal{B}$, by requiring that $\mathcal{I} \models \Gamma_{\mathcal{B}}$ and that $\mathcal{I} \models \Delta_{\mathcal{B}}$. Such an interpretation is also called a *premodel* of $\mathcal{B}$.

Given a policy base $\mathcal{B}$ we define $\sqsubseteq$ as the smallest preorder on premodels of $\mathcal{B}$ such that $\mathcal{I} \sqsubseteq \mathcal{I}'$ if either (1) $[\![C]\!]_{\mathcal{I}} \sqsubseteq [\![C]\!]_{\mathcal{I}'}$ for some inductive type variable C or (2) $[\![p]\!]_{\mathcal{I}} \sqsubseteq [\![p]\!]_{\mathcal{I}'}$ for some inductive propositional variable p.

We define *validity* of a policy base $\mathcal{B}$ under the interpretation $\mathcal{I}$, written $\mathcal{I} \models \mathcal{B}$, by requiring that $\mathcal{I} \models^{pre} \mathcal{B}$ and that $\mathcal{I}$ satisfies the *inductive constraint*, that is, $\mathcal{I}$ is minimal w.r.t. $\sqsubseteq$. A *model* of a policy base $\mathcal{B}$ is then an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{B}$. The *semantics of a policy base* is given by the class of all its models. A policy base without a model is said to be *inconsistent*.

7 Proof System

The proof system contains the inference rules of the type system and the following inference rules for *logical judgments*, which are of the form $\Gamma, \Delta \vdash Q$ for a context Γ, a set of sentences Δ, which are also called *hypotheses*, and a formula Q, which is called the *goal*.

7.1 Underlying Equational Theory

For the presentation of the inference rules we assume the following underlying equational theory so that certain terms and certain formulas are identified.

$M \mid_l (M' \mid_l M'') = (M \mid_l M') \mid_l M''$
$M \mid_l [] = M$

$M \mid_s N = N \mid_s M$
$M \mid_s (M' \mid_s M'') = (M \mid_s M') \mid_s M''$
$M \mid_s M = M$
$M \mid_s \{\} = M$

$M + N \equiv N + M$
$M * N \equiv N * M$

$M \equiv N = N \equiv M$
$M \not\equiv N = N \not\equiv M$
$M \not\equiv N = \texttt{not}\ N \equiv M$
$M \leq N = \texttt{not}\ N < M$

$P\ \texttt{and}\ \texttt{True} = P$
$P\ \texttt{or}\ \texttt{False} = P$

$A\ \texttt{and}\ A = A$
$A\ \texttt{or}\ A = A$

$P\ \texttt{and}\ Q = Q\ \texttt{and}\ P$
$P\ \texttt{or}\ Q = Q\ \texttt{or}\ P$
$P\ \texttt{and}\ (Q\ \texttt{and}\ R) = (P\ \texttt{and}\ Q)\ \texttt{and}\ R$
$P\ \texttt{or}\ (Q\ \texttt{or}\ R) = (P\ \texttt{or}\ Q)\ \texttt{or}\ R$

$(\forall \emptyset) P = P$
$(\exists \emptyset) P = P$
$(\forall \bar{x} : \bar{S})(\forall \bar{x'} : \bar{S'}) P = (\forall \bar{x} : \bar{S}, \bar{x'} : \bar{S'}) P$
$(\exists \bar{x} : \bar{S})(\exists \bar{x'} : \bar{S'}) P = (\exists \bar{x} : \bar{S}, \bar{x'} : \bar{S'}) P$

We continue to use the convention to identify terms and formulas that are equal modulo renaming of bound variables.

We identify judgments by means of the following equations:

$\Gamma, \Delta, \texttt{True} \vdash Q' = \Gamma, \Delta \vdash Q'$
$\Gamma, \Delta, P \texttt{ and } Q \vdash Q' = \Gamma, \Delta, P, Q \vdash Q'$

7.2 Eliminating Type Definitions

The inference rules for eliminating type definitions are extended to logical judgments as follows.

$$\frac{\Gamma, [C := S]\Gamma', [C := S]\Delta \vdash [C := S]Q}{\Gamma, C = S, \Gamma', \Delta \vdash Q} \quad \text{if } C \notin FV(\Gamma) \qquad (\mathsf{ElimTypeDef''})$$

7.3 Simplification of Hypotheses and Goals

In the subsequent subsections several *simplification relations* $\mapsto$ are used on terms, formulas, and sentences. The following inference rules show how they are used to simplify judgments. Generally, simplification rules come in pairs, reflecting the fact that simplification can take place in a hypothesis or in the goal. Except for the basic simplification relation used in the first two rules, simplification has a side condition (written as a subscript) and hence is generally a context-dependent notion.

$$\frac{\Gamma, \Delta[U'] \vdash Q}{\Gamma, \Delta[U] \vdash Q} \quad \text{if } U \mapsto U' \qquad (\mathsf{SimpHyp})$$

$$\frac{\Gamma, \Delta \vdash Q[U']}{\Gamma, \Delta \vdash Q[U]} \quad \text{if } U \mapsto U' \qquad (\mathsf{SimpGoal})$$

$$\frac{\Gamma, \Delta[U'] \vdash Q}{\Gamma, \Delta[U] \vdash Q} \quad \text{if } U \mapsto_S U' \text{ and } \Gamma \vdash M : S \qquad (\mathsf{NonEmptySimpHyp})$$

$$\frac{\Gamma, \Delta \vdash Q[U']}{\Gamma, \Delta \vdash Q[U]} \quad \text{if } U \mapsto_S U' \text{ and } \Gamma \vdash M : S \qquad \text{(NonEmptySimpGoal)}$$

Using $\texttt{ctor}(\bar{x}) \in \Gamma$ as shorthand for $\texttt{ctor}(x) \in \Gamma$ for all $x \in \bar{x}$ we have the following rules:

$$\frac{\Gamma, \Delta[U'] \vdash Q}{\Gamma, \Delta[U] \vdash Q} \quad \text{if } U \mapsto_{\texttt{ctor}(\bar{x})} U' \text{ and } \texttt{ctor}(\bar{x}) \in \Gamma \qquad \text{(CtorSimpHyp)}$$

where $\Delta[\bullet]$ does not bind any variables in $\bar{x}$.

$$\frac{\Gamma, \Delta \vdash Q[U']}{\Gamma, \Delta \vdash Q[U]} \quad \text{if } U \mapsto_{\texttt{ctor}(\bar{x})} U' \text{ and } \texttt{ctor}(\bar{x}) \in \Gamma \qquad \text{(CtorSimpGoal)}$$

where $Q[\bullet]$ does not bind any variables in $\bar{x}$.

$$\frac{\Gamma, \Delta[U'] \vdash Q}{\Gamma, \Delta[U] \vdash Q} \quad \text{if } U \mapsto_P U' \text{ and } P \in \Delta \qquad \text{(CtxtSimpHyp)}$$

where $\Delta[\bullet]$ does not bind any variables of P.

$$\frac{\Gamma, \Delta \vdash Q[U']}{\Gamma, \Delta \vdash Q[U]} \quad \text{if } U \mapsto_P U' \text{ and } P \in \Delta \qquad \text{(CtxtSimpGoal)}$$

where $Q[\bullet]$ does not bind any variables of P.

$$\frac{\Gamma, \Delta[U'] \vdash Q}{\Gamma, \Delta[U] \vdash Q} \quad \text{if } U \mapsto_{\vdash P} U' \text{ and } \Gamma, \Delta \vdash P \qquad \text{(CondSimpHyp)}$$

where $\Delta[\bullet]$ does not bind any variables of P.

$$\frac{\Gamma, \Delta \vdash Q[U']}{\Gamma, \Delta \vdash Q[U]} \quad \text{if } U \mapsto_{\vdash P} U' \text{ and } \Gamma, \Delta \vdash P \qquad \text{(CondSimpGoal)}$$

where $Q[\bullet]$ does not bind any variables of P.

7.3.1 Logical Simplification

We start out with the inductive definition of simplification on formulas, which is given by the following rules and further rules in subsequent subsections.

$\texttt{not not}\, A \mapsto A$

A `and not` $A \mapsto$ `False`
A `or not` $A \mapsto$ `True`

`not False` $\mapsto$ `True`
`not True` $\mapsto$ `False`

A `and False` $\mapsto$ `False`
A `or True` $\mapsto$ `True`

`not`$(A$ `or` $B) \mapsto$ (`not` A) `and` (`not` B)
`not`$(A$ `and` $B) \mapsto$ (`not` A) `or` (`not` B)

`not`$(\exists \bar{x} : \bar{S})A \mapsto (\forall \bar{x} : \bar{S})$ `not` A
`not`$(\forall \bar{x} : \bar{S})A \mapsto (\exists \bar{x} : \bar{S})$ `not` A

A `and` $(B$ `or` $C) \mapsto (A$ `and` $B)$ `or` $(A$ `and` $C)$

$(\forall x : S)P \mapsto_S P$ if $x \notin FV(P)$
$(\exists x : S)P \mapsto_S P$ if $x \notin FV(P)$

7.3.2 Simplification Rules for Equality

The following rules support the simplification of equality constraints applied to built-in and user-defined constants and constructs:

$M \equiv M \mapsto$ `True`
$b \equiv b' \mapsto$ `False` if $b \neq b'$
$r \equiv r' \mapsto$ `False` if $r \neq r'$

$(M, N) \equiv (M', N') \mapsto M \equiv M'$ `and` $N \equiv N'$

$[] \equiv [M'] \mapsto$ `False`
$[] \equiv M' \mid_l N' \mapsto [] \equiv M'$ `and` $[] \equiv N'$
$[M] \equiv [M'] \mapsto M \equiv M'$

$[M] \equiv M' \,|_l\, N' \mapsto []{\equiv}M'$ `and` $[M]{\equiv}N'$ `or` $[M]{\equiv}M'$ `and` $[]{\equiv}N'$
$[M] \,|_l\, N \equiv [M'] \,|_l\, N' \mapsto M{\equiv}M'$ `and` $N{\equiv}N'$
$M \,|_l [N] \equiv M' \,|_l [N'] \mapsto M{\equiv}M'$ `and` $N{\equiv}N'$
$M \,|_l\, N \equiv M \,|_l\, N' \mapsto N{\equiv}N'$
$M \,|_l\, N \equiv M' \,|_l\, N \mapsto M{\equiv}M'$

$\{M\} \,|_s\, N \mapsto_{\vdash M \in_s N} N$

$\{\} \equiv \{M'\} \mapsto$ `False`
$\{\} \equiv M' \,|_s\, N' \mapsto \{\}{\equiv}M'$ `and` $\{\}{\equiv}N'$
$\{M\} \equiv \{M'\} \mapsto M{\equiv}M'$
$\{M\} \equiv \{M'\} \,|_s\, N' \mapsto M{\equiv}M'$ `and` $(N'{\equiv}\{M'\}$ `or` $N'{\equiv}\{M\}$ `or` $N'{\equiv}\{\})$
$\{M\} \,|_s\, N \equiv \{M\} \,|_s\, N' \mapsto N{\equiv}N'$ `or` $N{\equiv}\{M\} \,|_s\, N'$ `or` $\{M\} \,|_s\, N{\equiv}N'$
$\{M\} \,|_s\, N \equiv \{M'\} \,|_s\, N' \mapsto_{\vdash M \equiv M'} N{\equiv}N'$ `or` $N{\equiv}\{M'\} \,|_s\, N'$ `or` $\{M\} \,|_s\, N{\equiv}N'$

$c \equiv fM \mapsto_{\texttt{ctor}(c,f)}$ `False` if $c \neq f$
$fM \equiv fN \mapsto_{\texttt{ctor}(f)} M{\equiv}N$

7.3.3 Simplification Rules for Ordering

Simplification of ordering constraints is defined by the following rules.

$M < M \mapsto$ `False`
$r < r' \mapsto$ `True` if $r < r'$ w.r.t. the standard ordering on $\mathbb{Q}$
$r < r' \mapsto$ `False` if $r' \leq r$ w.r.t. the standard ordering on $\mathbb{Q}$

`not` $M < N \mapsto N \leq M$

$M \equiv N$ `or` $M < N \mapsto M \leq N$

$M \equiv N$ `or` $M < N \mapsto M \leq N$

7.3.4 Simplification Rules for Built-in Predicates

The following simplification rules are for sort constraints.

`Bool` $b \mapsto$ `True`
`Bool` $M \mapsto$ `False` if M is a non-boolean constant
`Int` $i \mapsto$ `True`
`Int` $M \mapsto$ `False` if M is a non-integer constant
`Rat` $r \mapsto$ `True`
`Rat` $M \mapsto$ `False` if M is a non-rational constant

Membership constraints for lists and sets are simplified by the following rules:

$M \in_l [] \mapsto$ `False`
$M \in_l [M'] \mapsto (M \equiv M')$
$M \in_l M' \mid_l N' \mapsto (M \in_l M')$ `or` $(M \in_l N')$

$M \in_s \{\} \mapsto$ `False`
$M \in_s \{M'\} \mapsto (M \equiv M')$
$M \in_s M' \mid_s N' \mapsto (M \in_s M')$ `or` $(M \in_s N')$

7.3.5 Simplification Rules for Built-in Operators

We define the simplification relation for terms by means of the following rules.

$uop\ r \mapsto \widehat{uop}\ r$
$r\ bop\ r' \mapsto r\ \widehat{bop}\ r'$

$0 + M \mapsto M$
$M - 0 \mapsto M$
$0 * M \mapsto 0$
$1 * M \mapsto M$
$M/1 \mapsto M$

$(M + M') * N \mapsto (M * N) + (M' * N)$

$(M - M') * N \mapsto (M * N) - (M' * N)$
$(M + M')/N \mapsto (M/N) + (M'/N)$
$(M - M')/N \mapsto (M/N) - (M'/N)$

7.4 Subsumption Detection

Subsumption detection is based on a direct syntactic *subsumption relation* $\Rrightarrow$ on formulas that is inductively defined as follows.

$A \Rrightarrow A$
$M \equiv N \Rrightarrow M \leq N$
$M \equiv N \Rrightarrow N \leq M$
$M < N \Rrightarrow M \not\equiv N$
$M \equiv r \Rrightarrow M \leq r'$ if $r \leq r'$
$M \equiv r \Rrightarrow M < r'$ if $r < r'$
$r < M \Rrightarrow r' < M$ if $r' \leq r$
$r < M \Rrightarrow r' \leq M$ if $r' \leq r$
$r \leq M \Rrightarrow r' < M$ if $r' < r$
$r \leq M \Rrightarrow r' \leq M$ if $r' \leq r$
$M < r \Rrightarrow M < r'$ if $r \leq r'$
$M < r \Rrightarrow M \leq r'$ if $r \leq r'$
$M \leq r \Rrightarrow M < r'$ if $r < r'$
$M \leq r \Rrightarrow M \leq r'$ if $r \leq r'$
$M < r \Rrightarrow M \not\equiv r'$ if $r = r'$

Subsequently, subsumption is used to extend the simplification relation on formulas:

A `and` $A' \mapsto A$ if $A \Rrightarrow A'$
A' `or` $A \mapsto A'$ if $A \Rrightarrow A'$

A `and` $(A'$ `or` $Q) \mapsto A$ if $A \Rrightarrow A'$
A' `or` $(A$ `and` $Q) \mapsto A'$ if $A \Rrightarrow A'$

The subsumption relation can also be used to prove a goal that is directly subsumed by a hypothesis.

$$\frac{\cdot}{\Gamma, \Delta, A \vdash A'} \text{ if } A \Rrightarrow A' \qquad \text{(Subsumption)}$$

7.5 Inconsistency Detection

Similar to subsumption detection, inconsistency detection is based on a direct syntactic *inconsistency relation* that is inductively defined as a symmetric relation by the following rules.

$A \bowtie \texttt{not}\, A$
$M < N \bowtie N < M$
$M < N \bowtie N \leq M$
$M \equiv N \bowtie N < M$
$M \equiv r \bowtie M \equiv r'$ if $r \neq r'$
$M \equiv r \bowtie M \leq r'$ if $r' < r$
$M \equiv r \bowtie M < r'$ if $r' \leq r$
$r < M \bowtie M < r'$ if $r' \leq r$
$r < M \bowtie M \leq r'$ if $r' \leq r$
$r \leq M \bowtie M \leq r'$ if $r' < r$
$r \leq M \bowtie M < r'$ if $r' \leq r$

$A \bowtie A'$ if $A' \bowtie A$

Detected inconsistencies are then used in the following simplification rule:

$A \texttt{ and } A' \mapsto \texttt{False}$ if $A \bowtie A'$

7.6 Split Detection

We introduce a direct syntactic *splitting relation* that allows us to detect pairs of formulas that exhaustively cover all possibilities and hence can be used for simplification and for case splitting.

The symmetric *splitting relation* is inductively defined as follows:

$A \bigtriangledown \texttt{not}\, A$
$M \leq M' \bigtriangledown M' < M$
$M \leq M' \bigtriangledown M' \leq M$

$A \triangledown A'$ if $A' \triangledown A$

Detected splitting pairs are then used for a simplification relation as follows:

$(A \texttt{ and } B) \texttt{ or } (A' \texttt{ and } B') \mapsto \texttt{True}$ if $A \triangledown A'$

7.7 Closure Generation

The concept of closure is different from simplification. First, closure is applied only to hypotheses and not to goals. Second, closure rules can increase the complexity of the set of hypotheses by adding certain inferred (and hence logically redundant) hypotheses. The rationale behind this is to enable the triggering of further inferences, including further applications of closure or simplification rules, that would not happen otherwise.

$M \equiv M' \texttt{ and } M' \equiv M'' \rightsquigarrow M \equiv M''$
$M \equiv M' \texttt{ and } M' < M'' \rightsquigarrow M < M''$
$M < M' \texttt{ and } M' \equiv M'' \rightsquigarrow M < M''$
$M \equiv M' \texttt{ and } M' \leq M'' \rightsquigarrow M \leq M''$
$M \leq M' \texttt{ and } M' \equiv M'' \rightsquigarrow M \leq M''$
$M < M' \texttt{ and } M' < M'' \rightsquigarrow M < M''$
$M \leq M' \texttt{ and } M' < M'' \rightsquigarrow M < M''$
$M < M' \texttt{ and } M' \leq M'' \rightsquigarrow M < M''$
$M \leq M' \texttt{ and } M' \leq M'' \rightsquigarrow M \leq M''$

For the following rule, recall that because of the underlying equational theory on judgments, A can be a conjunction representing a set of formulas:

$$\frac{\Gamma, \Delta, A, A' \vdash Q}{\Gamma, \Delta, A \vdash Q} \quad \text{if } A \rightsquigarrow A' \text{ and } A' \notin \Delta \qquad \text{(Closure)}$$

7.8 Congruence

Congruence rules allow the inference of equality constraints between terms from equality constraints on subterms. We distinguish two flavors of congruence rules: backward congruence rules that are goal directed, and forward congruence rules that are similar to the closure rules above. Typically, both congruence and transitive closure rules for equality constraints are needed to

prove a given goal.

$$\frac{\Gamma, \Delta \vdash M \equiv N}{\Gamma, \Delta \vdash M'[M] \equiv M'[N]} \qquad \text{(BwCong)}$$

if $M'[\bullet]$ does not bind any free variables of M and N.

$$\frac{\Gamma, \Delta, M'[M] \equiv M'[N] \vdash Q}{\Gamma, \Delta \vdash Q} \qquad \text{(FwCong)}$$

if $M \equiv N \in \Delta$ and $M'[M]$ is a subterm in Δ and
$M'[\bullet]$ does not bind any free variables of M and N.

The side condition of FwCong avoids unnecessary instantiations. Clearly, the number of instantiations is bounded by the set of subterms of Δ.

7.9 Speculative Quantifier Instantiation

Speculative quantifier instantiation instantiates the universal quantifiers of logical rules so that the instantiated version can be potentially applied (forward or backward chaining) or can trigger other rules (opportunistic case splitting).

To be applicable to logical rule conditions we first inductively define $\Rrightarrow'$ by extending the subsumption relation $\Rrightarrow$ to conjunctions of formulas:

A `and` $A' \Rrightarrow' B$ `and` B' if $A \Rrightarrow B$ and $A' \Rrightarrow B'$

We furthermore define $Subst_\Gamma(\bar{x} : \bar{S})$ as the set of *well-typed substitutions* σ for $\bar{x} : \bar{S}$ in Γ, that is, substitutions for free variables in Γ such that $\Gamma \vdash \sigma\bar{x} : \bar{S}$.

Now we have a quantifier instantiation rule for each of our four flavors of logical rules. In all cases, the instantiation is based on a simple syntactic condition, which is a direct subsumption condition for the first two flavors, and a subterm/subformula condition for the other two.

$$\frac{\Gamma, \Delta, \sigma H \Leftarrow \sigma Q' \vdash Q}{\Gamma, \Delta \vdash Q} \qquad \text{(BwRuleInst)}$$

if $(\forall \bar{x} : \bar{S}) H \Leftarrow Q' \in \Delta$ and $\sigma \in Subst_\Gamma(\bar{x} : \bar{S})$ such that $\sigma H \Rrightarrow Q$.

$$\frac{\Gamma, \Delta, A, \sigma Q' \Rightarrow \sigma H \vdash Q}{\Gamma, \Delta, A \vdash Q} \qquad \text{(FwRuleInst)}$$

if $(\forall \bar{x} : \bar{S}) Q' \Rightarrow H \in \Delta$ and $\sigma \in Subst_\Gamma(\bar{x} : \bar{S})$ such that $A \Rrightarrow' \sigma Q'$.

$$\frac{\Gamma, \Delta, \sigma M = \sigma M' \Leftarrow \sigma Q' \vdash Q}{\Gamma, \Delta \vdash Q} \qquad \text{(EqRuleInst)}$$

if $(\forall \bar{x} : \bar{S}) M = M' \Leftarrow Q' \in \Delta$ and $\sigma \in Subst_\Gamma(\bar{x} : \bar{S})$ such that $\sigma M \sqsubseteq Q$.

$$\frac{\Gamma, \Delta, \sigma P \Leftrightarrow \sigma P' \Leftarrow \sigma Q' \vdash Q}{\Gamma, \Delta \vdash Q} \qquad \text{(IffRuleInst)}$$

if $(\forall \bar{x} : \bar{S}) P \Leftrightarrow P' \Leftarrow Q' \in \Delta$ and $\sigma \in Subst_\Gamma(\bar{x} : \bar{S})$ such that $\sigma P \sqsubseteq Q$.

7.10 Forward and Backward Chaining

Our backward chaining and forward chaining inference rules both assume that the logical rule to be applied already has been instantiated using one of the quantifier instantiation inference rules above. Since these inference rules use a notion of subsumption instead of logical equivalence we also say that backward chaining and forward chaining take place modulo subsumption (in spite of the fact that subsumption is not an equivalence relation).

7.10.1 Backward Chaining

Below we find the inference rules for backward chaining, which simply unifies (in the generalized sense of subsumption) the goal against the conclusion, and if successful verifies if the condition holds. Quantifiers must have been already instantiated by means of the quantifier instantiation rules. Note that the check of the condition is formulated as a side condition, to make sure that the inference rule can be applied only if the check is successful, rather than as a premise, which would mean that the goal would be replaced by the condition (in the backward reading of the inference rule), thereby deferring its verification.

$$\frac{\cdot}{\Gamma, \Delta \vdash Q} \quad \text{if } H \Rrightarrow Q \text{ and } \Gamma, \Delta \vdash Q' \qquad \text{(BwChain)}$$

where $H \Leftarrow Q' \in \Delta$.

7.10.2 Forward Chaining

Forward chaining is similar to our previously introduced closure rules for built-in constraints, except that it is based on user-defined forward rules. Again, quantifiers must already have been instantiated. Note that our extended simplification relation $\Rrightarrow'$ is used to check if the existing hypotheses directly imply the validity of the condition.

$$\frac{\Gamma, \Delta, A, H \vdash Q}{\Gamma, \Delta, A \vdash Q} \quad \text{if } A \Rrightarrow' Q' \qquad \text{(FwChain)}$$

where $Q' \Rightarrow H \in \Delta$.

7.11 Rewriting

Rewriting can make use of equalities, equational rules, or equivalence rules to simplify hypotheses or the goal. Just like in the case of forward and backward chaining, we assume that quantifier instantiation already had been performed in the case of logical rules. What distinguishes rewriting from forward and backward chaining is that it can be applied in any context rather than at the top-level formula. Like for backward chaining, conditions are checked in the side condition of the inference rule.

7.11.1 Value Substitution

Value substitution can be seen as a very special case of rewriting. We say that a term that contains only constants and constructors (built-in or declared in Γ) is a *value* in Γ. Now, value substitution replaces a term M by its value M' if there a corresponding equality $M \equiv M'$ (no quantifiers or condition allowed) among the hypotheses. Note that $\equiv$ is the ordinary (undirected) equality and not the (directed) equality that is used in equational rules.

$$\frac{\Gamma, \Delta, A[M'] \vdash Q}{\Gamma, \Delta, A[M] \vdash Q} \qquad \text{(ValueSubstHyp)}$$

where $M \equiv M' \in \Delta$, M is not a value and M' is a value in Γ, and $A[\bullet]$ does not bind any free variables of M or M'.

$$\frac{\Gamma, \Delta \vdash Q[M']}{\Gamma, \Delta \vdash Q[M]} \qquad \text{(ValueSubstGoal)}$$

where $M \equiv M' \in \Delta$, M is not a value and M' is a value in Γ, and $Q[\bullet]$ does not bind any free variables of M or M'.

7.11.2 Equational Rewriting

Equational rewriting uses the equational rules among the hypotheses for simplification. Quantifiers must already have been instantiated by means of the quantifier instantiation rules. Recall that rewriting takes place modulo the underlying equational theory of our proof system, which in particular includes algebraic properties of list and set constructors.

$$\frac{\Gamma, \Delta, A[M'] \vdash Q}{\Gamma, \Delta, A[M] \vdash Q} \text{ if } \Gamma, \Delta, A[M] \vdash Q' \qquad \text{(EqRewHyp)}$$

where $M = M' \Leftarrow Q' \in \Delta$ and
$A[\bullet]$ does not bind any free variables of M or M'.

$$\frac{\Gamma, \Delta \vdash Q[M']}{\Gamma, \Delta \vdash Q[M]} \text{ if } \Gamma, \Delta \vdash Q' \qquad \text{(EqRewGoal)}$$

where $M = M' \Leftarrow Q' \in \Delta$ and
$Q[\bullet]$ does not bind any free variables of M or M'.

7.11.3 Logical Rewriting

Logical rewriting is similar to equational rewriting except that it uses equivalence rules.

$$\frac{\Gamma, \Delta, A[P'] \vdash Q}{\Gamma, \Delta, A[P] \vdash Q} \text{ if } \Gamma, \Delta, A[P] \vdash Q' \qquad \text{(IffRewHyp)}$$

where $P \Leftrightarrow P' \Leftarrow Q' \in \Delta$ and
$A[\bullet]$ does not bind any free variables of P or P'.

$$\frac{\Gamma, \Delta \vdash Q[P']}{\Gamma, \Delta \vdash Q[P]} \text{ if } \Gamma, \Delta \vdash Q' \qquad \text{(IffRewGoal)}$$

where $P \Leftrightarrow P' \Leftarrow Q' \in \Delta$ and
$Q[\bullet]$ does not bind any free variables of P or P'.

7.12 Goal-Directed Structural Rules

The following goal-directed inference rules are based on the shape of their conclusion. Except for ExIntro the rules are deterministic, that is, there is a unique way to generate the new set of goals under the backward reading. The rule ExIntro can be implemented deterministically by using a metavariable to defer the decision for M.

$$\frac{\Gamma \vdash M : S \quad \Gamma, \Delta \vdash [x := M]A}{\Gamma, \Delta \vdash (\exists x : S)A} \tag{ExIntro}$$

$$\frac{\Gamma, \Delta, x : S, A \vdash Q}{\Gamma, \Delta, (\exists x : S)A \vdash Q} \quad x \notin FV(\Gamma, \Delta, Q) \tag{ExElim}$$

$$\frac{\Gamma, \Delta, x : S \vdash A}{\Gamma, \Delta \vdash (\forall x : S)A} \quad x \notin FV(\Gamma, \Delta) \tag{AllIntro}$$

$$\frac{\Gamma, \Delta, A \vdash \texttt{False}}{\Gamma, \Delta \vdash \texttt{not}\ A} \tag{NotIntro}$$

$$\frac{.}{\Gamma, \Delta, A, \texttt{not}\ A \vdash Q} \tag{NotElim}$$

$$\frac{\Gamma, \Delta, \texttt{not}\ A \vdash B}{\Gamma, \Delta \vdash A\ \texttt{or}\ B} \tag{OrIntro}$$

$$\frac{\Gamma, \Delta, A \vdash Q \quad \Gamma, \Delta, B \vdash Q}{\Gamma, \Delta, A\ \texttt{or}\ B \vdash Q} \tag{OrElim}$$

$$\frac{\Gamma, \Delta \vdash A \quad \Gamma, \Delta, A \vdash B}{\Gamma, \Delta \vdash A\ \texttt{and}\ B} \tag{AndIntro}$$

$$\frac{\Gamma, \Delta, A, B \vdash Q}{\Gamma, \Delta, A\ \texttt{and}\ B \vdash Q} \tag{AndElim}$$

Note that some rules are redundant. The rule AndElim is trivially derivable from the underlying equational theory for judgments. Similarly, NotElim can be derived via using simplification. Logical implication has been introduced

as syntactic sugar, and the following implication introduction rule can be derived using OrIntro.

$$\frac{\Gamma, \Delta, A \vdash B}{\Gamma, \Delta \vdash \texttt{not}\ A\ \texttt{or}\ B} \qquad \text{(ImpIntro)}$$

It is noteworthy that no general elimination rules for universal quantifiers or implication exist. Instead, special cases of such rules are covered by our quantifier instantiation and forward/backward chaining and equational/equivalence rewriting rules.

7.13 Rules for the Description Construct

Depending on where the description construct appears, one of two different inferences rules may be applicable. First, the introduction rule deals with a subterm $(\epsilon x : S)A$ that appears in the goal, which requires a witness M for such an x, a decision that again can be deferred using a metavariable. Second, the elimination rule, that covers the case where $(\epsilon x : S)A$ appears as a subterm of one of the hypotheses.

$$\frac{\Gamma \vdash M : S \quad \Gamma, \Delta \vdash [x := M]A \quad \Gamma, \Delta, x : S, A \vdash Q[x]}{\Gamma, \Delta \vdash Q[(\epsilon x : S)A]} \qquad \text{(EpsIntro)}$$

where $x \notin FV(\Gamma, \Delta)$ and $Q[\bullet]$ does not bind any variables in $(\epsilon x : S)A$.

$$\frac{\Gamma, \Delta, x : S, P[x], A \vdash Q}{\Gamma, \Delta, P[(\epsilon x : S)A] \vdash Q} \qquad \text{(EpsElim)}$$

where $x \notin FV(\Gamma, \Delta, Q)$ and $P[\bullet]$ does not bind any variables in $(\epsilon x : S)A$.

Since the description construct can be seen as an existential quantifier pushed into terms, it is instructive to compare these rules with ExIntro and ExElim, respectively. The main difference is that EpsIntro has the stronger premise that the goal can be proved without referring to a *specific* x, that is, $Q[x]$ can be proved assuming any x that satisfies A.

7.14 Opportunistic Case Splitting

The inferences rules for opportunistic case splitting assume that quantifier instantiation rules have been already applied and try to find pairs of preferably but not necessarily complementary formulas, which can be used to perform a case split. The candidates for such formulas are subformulas occurring in the conditions of logical rules. The splitting relation $\triangledown$ will help us to identify possible case splits.

Below we can find an inference rule for case splitting based on each kind of logical rule:

$$\frac{\cdot}{\Gamma, \Delta \vdash Q} \quad \text{if } \Gamma, \Delta, A \vdash Q \text{ and } \Gamma, \Delta, B \vdash Q \qquad \text{(BwRuleCaseSplit)}$$

if $A'' \Leftarrow A$ `and` $A' \in \Delta$ and $B'' \Leftarrow B$ `and` $B' \in \Delta$
and $A \triangledown B$ and $A'' \Rrightarrow Q$ and $B'' \Rrightarrow Q$.

$$\frac{\cdot}{\Gamma, \Delta \vdash Q} \quad \text{if } \Gamma, \Delta, A \vdash Q \text{ and } \Gamma, \Delta, B \vdash Q \qquad \text{(FwRuleCaseSplit)}$$

if A `and` $A' \Rightarrow A'' \in \Delta$ and B `and` $B' \Rightarrow B'' \in \Delta$
and $A \triangledown B$ and $A'' \Rrightarrow Q$ and $B'' \Rrightarrow Q$.

$$\frac{\cdot}{\Gamma, \Delta \vdash Q} \quad \text{if } \Gamma, \Delta, A \vdash Q \text{ and } \Gamma, \Delta, B \vdash Q \qquad \text{(EqRuleCaseSplit)}$$

if $N = N' \Leftarrow A$ `and` $A' \in \Delta$ and $N = N'' \Leftarrow B$ `and` $B' \in \Delta$
and $A \triangledown B$ and N is a subterm of Q.

$$\frac{\cdot}{\Gamma, \Delta \vdash Q} \quad \text{if } \Gamma, \Delta, A \vdash Q \text{ and } \Gamma, \Delta, B \vdash Q \qquad \text{(IffRuleCaseSplit)}$$

if $P \Leftrightarrow P' \Leftarrow A$ `and` $A' \in \Delta$ and $P \Leftrightarrow P'' \Leftarrow B$ `and` $B' \in \Delta$
and $A \triangledown B$ and P is a subformula of Q.

7.15 Proof by Contradiction

The following inference rule allows to conduct an indirect proof. A goal A can be proved by deriving a contradiction from `not` A.

$$\frac{\Gamma, \Delta, \texttt{not}\ A \vdash \texttt{False}}{\Gamma, \Delta \vdash A} \qquad \text{(Contra)}$$

7.16 Rule Processing

Rules are used to prove goals, but they can be themselves subject to various kinds of processing. In most basic rules for simplification and elimination, knowledge about other hypotheses is exploited.

7.16.1 Rule Simplification

The condition in each of the four flavors of logical rules can be simplified if it is directly implied by another hypothesis.

If $A \Rrightarrow A'$ and the rule quantifier does not bind any free variables of A', then we have the following rules for simplifying logical rule conditions:

$$(\forall \bar{x} : \bar{S}) Q' \ \texttt{and}\ A' \Rightarrow H \mapsto_A (\forall \bar{x} : \bar{S}) Q' \Rightarrow H \qquad \text{(SimpFwRules)}$$

$$(\forall \bar{x} : \bar{S}) H \Leftarrow Q' \ \texttt{and}\ A' \mapsto_A (\forall \bar{x} : \bar{S}) H \Leftarrow Q' \qquad \text{(SimpBwRules)}$$

$$(\forall \bar{x} : \bar{S}) P \Leftrightarrow P' \Leftarrow Q' \ \texttt{and}\ A' \mapsto_A (\forall \bar{x} : \bar{S}) P \Leftrightarrow P' \Leftarrow Q' \qquad \text{(SimpIffRules)}$$

$$(\forall \bar{x} : \bar{S}) M = M' \Leftarrow Q' \ \texttt{and}\ A' \mapsto_A (\forall \bar{x} : \bar{S}) M = M' \Leftarrow Q' \qquad \text{(SimpEqRules)}$$

If x is not a free variable in P, P', Q, Q', M, M', then we furthermore have the following rules for eliminating redundant logical rule quantifiers, which require non-emptiness of the type associated with the variable x we want to eliminate:

$$(\forall \bar{x} : \bar{S}, x : S) Q' \Rightarrow H \mapsto_S (\forall \bar{x} : \bar{S}) Q' \Rightarrow H \qquad \text{(SimpFwRules')}$$

$$(\forall \bar{x} : \bar{S}, x : S) H \Leftarrow Q' \mapsto_S (\forall \bar{x} : \bar{S}) H \Leftarrow Q' \qquad \text{(SimpBwRules')}$$

$$(\forall \bar{x} : \bar{S}, x : S) P \Leftrightarrow P' \Leftarrow Q' \mapsto_S (\forall \bar{x} : \bar{S}) P \Leftrightarrow P' \Leftarrow Q' \qquad \text{(SimpIffRules')}$$

$$(\forall \bar{x} : \bar{S}, x : S) M = M' \Leftarrow Q' \mapsto_S (\forall \bar{x} : \bar{S}) M = M' \Leftarrow Q' \qquad \text{(SimpEqRules')}$$

7.16.2 Rule Elimination

A logical rule can be completely eliminated if its condition is inconsistent with any of the hypotheses.

If $A \bowtie A'$ and the rule quantifier does not bind any free variables of A', then we have the following rule elimination rules:

$$(\forall \bar{x} : \bar{S})Q' \texttt{ and } A' \Rightarrow H \mapsto_A \texttt{True} \qquad \text{(ElimFwRules)}$$

$$(\forall \bar{x} : \bar{S})H \Leftarrow Q' \texttt{ and } A' \mapsto_A \texttt{True} \qquad \text{(ElimBwRules)}$$

$$(\forall \bar{x} : \bar{S})P \Leftrightarrow P' \Leftarrow Q' \texttt{ and } A' \mapsto_A \texttt{True} \qquad \text{(ElimIffRules)}$$

$$(\forall \bar{x} : \bar{S})M = M' \Leftarrow Q' \texttt{ and } A' \mapsto_A \texttt{True} \qquad \text{(ElimEqRules)}$$

7.16.3 Forward Rule Splitting

Forward rules with the same head are subject to splitting if the condition is a disjunction. This allows them to be used independently for forward chaining and opportunistic case splitting.

$$(\forall \bar{x} : \bar{S})(Q' \texttt{ or } Q'') \Rightarrow H \mapsto (\forall \bar{x} : \bar{S})Q' \Rightarrow H, (\forall \bar{x} : \bar{S})Q'' \Rightarrow H \qquad \text{(SplitRules)}$$

No corresponding splitting rule exists for backward rules. Instead, backward rules with the same head can be merged by means of the following completion rules.

7.16.4 Backward Rule Completion

The objective of rules completion is to bring an inductive definition given by a set of backward rules into a form that is a logical equivalence, and hence suitable for logical rewriting. Completion will also allow us to convert inductive definitions that are suitable for positive reasoning into definitions that can support both, positive and negative, forms of reasoning.

If $\texttt{ind}(p) \in \Gamma$ we have the following rules:

$$\frac{\Gamma, \Delta, (\forall x : S)p\ x \Leftarrow (\exists \bar{x} : \bar{S})x \equiv M \texttt{ and } Q' \vdash Q}{\Gamma, \Delta, (\forall \bar{x} : \bar{S})p\ M \Leftarrow Q' \vdash Q} \quad \text{(Abstraction)}$$

where $x \notin FV(\Gamma, \Delta)$ and
$(\forall \bar{x} : \bar{S})p\ M \Leftarrow \ldots$ is not of the form $(\forall x : S)p\ x \Leftarrow \ldots$.

$$\frac{\Gamma, \Delta, p \Leftarrow (Q' \texttt{ or } Q'') \vdash Q}{\Gamma, \Delta, p \Leftarrow Q', p \Leftarrow Q'' \vdash Q} \quad \text{(Fusion)}$$

$$\frac{\Gamma, \Delta, (\forall x : S)p\ x \Leftarrow (Q' \texttt{ or } Q'') \vdash Q}{\Gamma, \Delta, (\forall x : S)p\ x \Leftarrow Q', (\forall x : S)p\ x \Leftarrow Q'' \vdash Q} \quad \text{(Fusion')}$$

$$\frac{\Gamma, \Delta, p \Leftrightarrow Q' \vdash Q}{\Gamma, \Delta, p \Leftarrow Q' \vdash Q} \quad \text{(Completion)}$$

if p does not occur positively in Δ.

$$\frac{\Gamma, \Delta, (\forall x : S)p\ x \Leftrightarrow Q' \vdash Q}{\Gamma, \Delta, (\forall x : S)p\ x \Leftarrow Q' \vdash Q} \quad \text{(Completion')}$$

if p does not occur positively in Δ.

8 Operational Semantics

Under the backward reading, the UPL inference rules define a relation $J \rightarrow \mathcal{J}_\wedge$ that holds iff there is an instance of an inference rule that has the set of judgments $\mathcal{J}_\wedge$ as a premise and the judgment J as a conclusion.

A *conjunctive proof state* is a set $\mathcal{J}_\wedge$ of judgemnts. We inductively extend $\rightarrow$ to a relation $\rightarrow_\wedge$ on conjunctive proof states by

$\{J\} \cup \mathcal{J}'_\wedge \rightarrow_\wedge \mathcal{J}_\wedge \cup \mathcal{J}'_\wedge$ if $J \rightarrow \mathcal{J}_\wedge$ and $J \notin \mathcal{J}_\wedge$.

The condition $J \notin \mathcal{J}_\wedge$ prevents trivial non-termination, that is, the application of inference rules that generate proof obligations containing the original goal.

A *disjunctive proof state* is a set of conjunctive proof states. We inductively extend $\rightarrow_\wedge$ to a relation $\rightarrow_\vee$ on disjunctive proof states by

$\{\emptyset, \mathcal{J}_\wedge\} \rightarrow_\vee \emptyset$
$\{\mathcal{J}_\wedge\} \rightarrow_\vee \{\mathcal{J}'_\wedge \mid \mathcal{J}_\wedge \rightarrow_\wedge \mathcal{J}'_\wedge\}$
$\mathcal{J}_\vee \cup \mathcal{J}'_\vee \rightarrow_\vee \mathcal{J}'_\vee \cup \mathcal{J}''_\vee$ if $\mathcal{J}_\vee \rightarrow_\vee \mathcal{J}'_\vee$.

Now the *operational semantics* is given by the transition system with a set of states that is the set of all disjunctive proof states and the above relation $\rightarrow_\vee$ as a transition relation.

It is noteworthy that both the conjunctive as well as the disjunctive proof state can be infinite, but an implementation will typically use schematic representations, for example, judgments involving metavariables, to represent reachable proof states in a finitary way.

9 Application to Spectrum Policies

Until now the logical foundations for policies have been introduced without reference to a particular application domain. We now consider spectrum policies as one particular application of our policy framework. The central question a policy engine needs to answer is if a radio can use a transmission opportunity with certain parameters given that such a transmission is allowed and not disallowed by the policy.

Hence, a *spectrum policy* is a special kind of policy that uses two predicates *allow* and *disallow*. A *spectrum metapolicy* then introduces a new predicate *transmit* in terms of the *allow* and *disallow* predicates of individual policies. A *spectrum policy base* is a set containing any number of spectrum policies and a single metapolicy.

In the simplest case, the policy base contains one policy with its *allow* and *disallow* predicates and a metapolicy defines *transmit* in terms of these:

$$transmit \Leftrightarrow allow \texttt{ and not } disallow$$

More generally, the policy base can contain any number of policies $P_1, \ldots, P_r$, each with it own *allow* and *disallow* predicates, and the metapolicy

$$transmit \Leftrightarrow \begin{array}{l} (allow_1 \texttt{ or } \ldots \texttt{ or } allow_r) \texttt{ and} \\ (\texttt{not } disallow_1 \texttt{ and } \ldots \texttt{ and not } disallow_r) \end{array}$$

Now, a *transmission opportunity* of a spectrum policy base $\mathcal{B}$ is a model $\mathcal{I}$ of $\mathcal{B}$ such that $\mathcal{I} \models transmit$.

For more complicated metapolicies, for example, involving a precedence between spectrum policies, predicates with parameters are important in practice and can be expressed in our framework. Furthermore, the metapolicy is an example of policy invocation, in the sense that the metapolicies invokes all other policies. More generally, such invocations can be part of any policy. In other words, a policy may become a metapolicy for other policies that it invokes.