Arithmetization of predicates into Halo 2 using application specific trace types

Morgan Thomas
Casper Association
morgan@casper.network

May 11, 2023

Abstract

This note provides an update on the Open Specification Language (OSL) circuit compiler. OSL is a language based on predicate logic which is amenable to compilation to arithmetic constraint systems for use in constructing (zk-)SNARKs. This system provides an alternative to universal zk-VMs and low level *ad hoc* constructions of arithmetic constraint systems, which is potentially more efficient than universal zk-VMs but more cost effective as a development approach than low level *ad hoc* constructions.

Arithmetization is the process of expressing a relation as an arithmetical constraint system suitable for proving membership in the relation using (zk-)SNARKs. There are two kinds of approaches to arithmetization: universal and application-specific. Universal arithmetization uses constraint systems which can capture any NP relation up to some complexity bounds. Application-specific arithmetization uses *ad hoc* constraint systems which each capture one NP relation (again, up to some complexity bounds).

Universal arithmetization typically works by expressing the semantics of a virtual machine architecture as an arithmetical constraint system, yielding a so-called zero knowledge virtual machine (zk-VM). Application specific arithmetization typically works by writing a constraint system more or less directly within some cryptographic application development framework, or else by using a compiler which transforms a relation expressed in some language into an arithmetical constraint system.

In zero knowledge proving for a relation R, one wants to show, for some public x, that there exists some (private) w such that $R(x, w)$. Oftentimes, one models the relation to be arithmetized as a function; in general, for a function f, one wants to show, for some public x and y, that there exists some (private) w such that $f(x, w) = y$. The function formulation is a special case of the relation formulation where one lets $R = \{((x, y), w) : f(x, w) = y\}$. One can also go from the relation formulation to the function formulation, given an algorithm for checking membership in the relation R, that is, a function f such that $f(x, w) = 1$ iff $R(x, w)$.

The goal of this research is to develop tools for creating zk-SNARKs for a relation R without being concerned about how the relation R gets checked. Such tools would allow for working at a higher level of abstraction, separating the concern of *what* is to be proven from *how* it is to be proven. zk-VM approaches require as input a program and inputs to the program, thus requiring the end developer to specify a way to check the statement using a program execution.

As an alternative, this research would allow end developers to specify a relation R by a formula $\Phi(x, w)$ in a version of predicate logic, using a circuit compiler to generate an arithmetical constraint system for the relation R, and an argument translator which takes inputs (x, w) such that $R(x, w)$ and turns them into inputs satisfying the arithmetical constraint system.

This research began with the support of Orbis Labs and continues with the support of Casper Association. Prior art in this line of research [1, 3, 4] defined a version of predicate logic which is strong enough to express NP relations which are practical to arithmetize, and also some methods for arithmetizing formulas in that language on top of PLONKish arithmetization using Halo 2. [5, 6] The language in question is called OSL, which originally stood for Orbis Specification Language and now stands for Open Specification Language (a backronym). Implementation efforts [2] are ongoing and open source. As of this writing, these efforts are closely approaching a point where generating zero knowledge proofs will be possible. The current paper provides an update on the research and development efforts, which have come up with more practical methods of arithmetization than what are described in the prior literature.

OSL is a layer of abstraction over Σ^1_1 formulas. Σ^1_1 formulas are a sublanguage of bounded second order arithmetic [7] extended with $\mathrm{ind}_<$ and max built-in functions, defined as follows:

$$\mathrm{ind}_<(x, y) = \begin{cases} 1 & \text{if } x < y, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

$$\max(x, y) = \begin{cases} x & \text{if } y \leq x, \\ y & \text{otherwise.} \end{cases} \tag{2}$$

Definitions of the syntax and semantics of OSL and Σ^1_1 formulas, as well as an example OSL spec, are available in the appendices to this paper.

Σ^1_1 formulas allow for first-order quantifiers with arbitrary term bounds, and second-order existential quantifiers with arbitrary term bounds. See Appendix D for a formal definition of Σ^1_1 formulas and Appendix E for their denotational semantics. OSL adds rudiments of type theory, including basic functors like list and map type constructors, all of which can be straightforwardly translated into Σ^1_1 formulas. See Appendix B for a formal definition of OSL, and Appendix C for its denotational semantics.

There is a natural connection between Σ^1_1 formulas and computation, in that sequential models of computation are typically natural to express as Σ^1_1 formulas. If a function f is defined within a sequential model of computation, then $f(x, w) = y$ means in other words that there is a sequence of states $s_0, ..., s_n$ such that s_0 is the appropriate initial state for inputs (x, w), s_n is a final state where the program has halted and indicated an output of y, and for all $i \in [0, n)$, the step transition function of the model of computation maps s_i to s_{i+1}. The same can be expressed as a Σ^1_1 formula as long as the step transition function is a first-order relation (i.e., it can be expressed using only first-order bounded quantifiers).

There is also a natural connection between Σ^1_1 formulas and arithmetical constraint systems. The semantics of arithmetical constraint systems are typically natural to express as Σ^1_1 formulas, and it is also very feasible to express a Σ^1_1 formula as an arithmetical constraint system. The free variables in the formula (the ones not bound by quantifiers) correspond to public inputs, and the variables bound by existential quantifiers correspond to private inputs. In a useful variant on this approach, we consider Σ^1_1 formulas with no free variables, and we add a so-called instance quantifier, λ, where $\lambda x.\ \Phi(x)$ has the same semantics as $\Phi(x)$ (where x is free in $\Phi(x)$). What is useful about the instance quantifier concept is that we can use it to apply a bound to an instance variable, as in $\lambda x < \alpha.\ \Phi(x)$.

It is in general not computationally feasible to compute whether a Σ^1_1 formula is true or false. However, it is in general computationally feasible to check whether a first-order (bounded quantifier) formula is satisfied or not on given inputs. Given satisfying values for the instance or second-order existential quantifiers in a Σ^1_1 formula, it is in general computationally feasible to check that the formula is true, since this reduces to the problem of determining that a first-order formula is true after plugging in the given values for the second-order variables.

Prior research [1, 4] described methods for arithmetizing Σ^1_1 formulas which took advantage of the similarities between Σ^1_1 formulas and arithmetical constraint systems. These methods, however, were not feasible to apply in practice. They tended to result in a blow-up in the degrees and numbers of terms in the resulting polynomial constraints, to a point that it was often not feasible to compute these polynomials, let alone generate zk-SNARKs based on them.

The current version of the OSL compiler [2] demonstrates a feasible method of compiling Σ_1^1 formulas to arithmetical constraint systems. This method is based on application specific trace types.

A "trace" describes the steps of a computational process. For a sequential model of computation, a trace is equivalent to a series of states $s_0, ..., s_n$, with s_0 being the initial state and s_n the final state, such that for all $i \in [0, n)$, the step transition relation for the model of computation relates s_i to s_{i+1}. More generally, a trace need not have a sequential structure, but can have a tree structure.

For examples of non-sequential traces, we can consider traces for an arithmetic circuit. An arithmetic circuit, let's say, is a directed acyclic graph, where each non-initial node (each node with an ancestor) has exactly two ancestors and is labeled with either $+$ or $\times$, and there is exactly one final node (a node which is not an ancestor). A trace for an arithmetic circuit is a function from the set of nodes to the set of field elements.

A "trace type" is an object which defines what would be a valid trace for that trace type. For example, a trace type can define valid traces for a specific arithmetic circuit, or valid traces for a specific model of computation. The reason for defining a general trace type abstraction is in order to define application specific trace types for arithmetizing relations using application specific constraint systems. The overall approach is to define a compiler which compiles relations defined in OSL into Σ_1^1 formulas, which are then compiled into application specific trace types, which are then compiled into arithmetical constraint systems.

It would of course be possible to use a universal trace type to arithmetize all NP relations (up to some complexity bound). This is the approach of universal zk-VMs. The hypothesis driving this research into application specific trace types is that given specific knowledge of the relation of interest, one can automatically come up with a trace type which more efficiently arithmetizes the relation of interest.

Overall, the OSL compiler in its current form has the following stages, from first to last:

1. Tokenizing turns the textual representation of OSL code into a sequence of lexical tokens.

2. Parsing turns the tokenized OSL code into an abstract syntax tree (AST).

3. Semantic analysis turns the AST of the OSL code into a valid context, which maps names to their definitions, ensuring that these definitions are interpretable within the context and the semantic rules of OSL.

4. The valid OSL from semantic analysis gets compiled into a Σ_1^1 formula. In this stage, the Σ_1^1 formula uses de Bruijn indices as variable names.

5. The de Bruijn indices in the Σ_1^1 formula get replaced with gensyms, which are globally unique generated names. Formulas using de Bruijn indices are relatively easy to generate, because unintended variable capture is not much of a concern; there is no need to carry state to remember which variable names have been used during the generation process. On the other hand, formulas using gensyms are relatively easy to manipulate. That is why this stage is useful.

6. The Σ_1^1 formula gets converted into a prenex normal form, which brings all quantifiers to the front of the formula.

7. The prenex normal form gets converted into a strong prenex normal form, which brings all instance quantifiers to the front of the formula, followed by all existential quantifiers, followed by all universal quantifiers.

8. The strong prenex normal form gets compiled into a logic circuit. A logic circuit is a structure similar to a Halo 2 [5, 6] circuit, with some key differences, as follows. Logic circuits do not have top-level lookup arguments, but instead just gate constraints and equality constraints. Instead of taking the form $p = 0$, for p a polynomial, gate constraints are quantifier-free logic formulas which can assert equalities and inequalities over terms involving variables, constants, addition, multiplication, the max and $\text{ind}_<$ functions, and lookups.

   Variables in this context are the same as they are in Halo 2 gate constraints: they are of the form $x_{i,j}$, where i is an absolute column index and j is a relative row index (an offset from the current row). Lookups in this context are of the form $c_{n+1}((t_1, c_1), ..., (t_n, c_n))$ where each t_i is a term and each c_i is a column (i being an absolute column index). The output of a lookup term at a row r is the value of column c_{n+1} in the unique row r' such that the vector of column values $(c_1, ..., c_n)$ at row r' is equal to the vector of term values $(t_1, ..., t_n)$ at row r. Lookups are effectively the "missing link" between function calls and lookup arguments, having some characteristics of both function calls and lookup arguments, being the result of compiling function calls and being compiled into lookup arguments in Halo 2 circuits.

   This notion of logic circuits is similar to the notion of logic circuits found in [4], with the difference being that terms related by equalities and inequalities are as just described instead of being plain polynomials as in [4].

9. The logic circuit gets compiled into a trace type. See Section 1 for an explanation of trace types.

10. The trace type gets compiled into a Halo 2 circuit.

For practical purposes, one needs not only a compiler to turn OSL code into Halo 2 circuits, but also a compiler to turn inputs satisfying a predicate defined in OSL into inputs satisfying the resulting Halo 2 circuit. The latter compiler is called the "argument compiler." The OSL compiler only needs to be applied once per relation to be arithmetized, and the argument compiler needs to be applied once per instance to be proven.

As of this writing, the OSL compiler and the argument compiler exist and pass tests, but they use a notion of circuits which is not exactly compatible with Halo 2. The disconnect is that the notion of circuits targeted in the OSL compiler features lookup arguments gated by polynomial expressions, such that the lookup arguments only apply at rows where the gate expression is zero, whereas Halo 2 only supports lookup arguments gated by fixed selector columns. There is nothing that prevents this gap from being bridged efficiently, and work to bridge the gap is underway as of this writing.

In the previously published constructions of Halo 2 circuits from logic circuits, [1, 4], each row of the Halo 2 circuit corresponds to one row of the source logic circuit. Because the constraints in a logic circuit can imply a lot of complex constraints on one row, the complexity of the polynomials in the resulting Halo 2 circuits tends to be exceedingly high in the previously published constructions. The solution to this problem in the trace type based compilation pipeline is to let one row in the logic circuit correspond to many rows in the resulting Halo 2 circuit. In this newer construction, each row of the Halo 2 circuit verifies the evaluation of one subexpression at one row of the logic circuit. A subexpression can be a term, a (sub)formula of a gate constraint, or an assertion that a gate constraint formula is true.

1 Traces and trace types

A "trace" is a value which can satisfy or fail to satisfy a trace type. A trace consists of a statement, a witness, and a map from cases to maps from subexpression ids to subexpression traces. A subexpression trace consists of the value of the subexpression (a field element), the step type id of the subexpression, and the associated advice values (a map from column indexes to field elements). Each row of a logic circuit input matrix corresponds to one case of the corresponding trace. Each information bearing row of the resulting Halo 2 circuit input matrix corresponds to one subexpression trace of one case.

Similar to a Halo 2 circuit, a trace type defines a map from column indices to their column types (fixed, advice, or instance). It defines a set of equality constrainable column indices. It defines equality constraints. Each equality constraint is isomorphic to a set of cell references (which are isomorphic to pairs of column indices and (absolute) row indices). A trace type defines the values at each *case* of each fixed column.

A trace type defines a map from step type ids to step type definitions. A step type definition essentially defines a set of polynomial gate constraints, a set of Halo 2 lookup arguments, and some fixed values. As with the fixed values associated with the trace type overall, the fixed values associated with a step type define one value per *case*, i.e. one value per row of the source logic circuit, as opposed to one value per row of the output Halo 2 circuit. This means that all rows associated with a given case have the same fixed values, as far as those fixed values defined by the step type go.

A trace type defines a set of subexpression ids and a set of subexpression links. Each subexpression link is of the form $\vec{i} \mapsto^t o$, where o is a subexpression id (the output subexpression), $\vec{i}$ is a vector of subexpression ids (the input subexpressions), and t is a step type id.

It is legal for there to be different subexpression links with the same output; this entails that that subexpression's value can validly be determined based on more than one step type and/or more than one input vector. So that each output subexpression value is uniquely defined by the logic circuit input matrix, it should be that in any case, all links that can be instantiated for an output o result in the same value of o.

The motivating example of having multiple links with the same output is short circuiting evaluation. For example, if $x = 0$, then $x \times y = 0$. So we can have three links outputting $x \times y$, one of which takes x as input and works when $x = 0$ and the output is 0, one of which takes y as input and works when $y = 0$ and the output is 0, and one of which takes x and y as input and works for any values of x and y.

A trace type defines a subset of the set of subexpression ids which are the "result subexpression ids." For a trace to satisfy the trace type, each of its result subexpression ids must be true (i.e., have value 1) for each case.

A trace type defines its number of cases (equal to the number of rows of the source logic circuit) and its number of rows. The number of rows is less than or equal to $c \times |S|$, with c the number of cases, and S the set of subexpression ids. In practice, in the current version of the compiler, the number of rows is always equal to $c \times |S|$, but in future versions, the number of rows could be less in some cases, as an optimization, whenever it can happen that not all subexpressions need to be instantiated for all cases.

Finally, a trace type defines various column indices:

- It defines a case number column index, which should be assigned, for each row, the case number that row applies to.

- It defines a step indicator column index, which should be assigned one at each row representing a subexpression, and zero at all other rows.

- It defines a step type id selection vector, which is a vector of column indices, one per step type. At each row with a step indicator value of one, the step type id selection vector should contain exactly one column index assigned the value one, with the rest being assigned zero. At rows with a step indicator value of zero, all of the step type selection vector columns can be assigned zero. The step type selection vector values indicate which step type applies to each row.

- It defines a vector of input column indices, of length equal to the number of inputs to a step type, which get assigned the values of the input subexpressions to the subexpression related to each row which is used to represent a step. Note that all step types have the same number of inputs, and for step types where some of these inputs are not needed, some of these inputs are fed by the "void" step type, which always outputs zero and is linked to itself for all of its own inputs.

- It defines an output column index, which gets assigned the value of the subexpression represented by the row at each row representing a subexpression.
- It defines a case used column index, which gets assigned the value one for each row whose case number is a used case for the trace, and zero for all other rows.

A trace satisfies a trace type if:

- For each used case, all result subexpressions are present.
- For each row with a step indicator value of one, the vector of step type selection vector values contains exactly one 1 and the rest of its values are 0.
- For each row, all applicable step type constraints are satisfied.
- All equality constraints are satisfied.

2 Compiler stages

For full details on the compiler stages, see the code. [2] Here are some concise comments on how the most interesting compiler stages work.

1. The compilation of OSL to Σ_1^1 formulas is largely the same as described in [3].
2. Conversion to strong prenex normal form relies on the fact that instance quantifiers will already be in the front of the Σ_1^1 formula as output by the OSL to Σ_1^1 stage, and this property is preserved by the conversion to prenex normal form. Therefore it suffices to move all existential quantifiers in front of all universal quantifiers. For this, it suffices to understand how to move one existential quantifier in front of one universal quantifier in front of it. This is done by turning a first-order existential quantifier into a second-order quantifier with one argument, and a second-order existential quantifier with n arguments into a second-order quantifier with $n + 1$ arguments. The value of the universal quantifier becomes an argument of the existentially quantified function. This process is similar to the process of Skolemization.
3. The compilation of strong prenex normal forms to logic circuits is similar to the process described in [4], but somewhat simplified by the usage of a strong prenex normal form instead of a prenex normal form. The values of the instance and existential quantifiers are embedded as lookup tables in the logic circuit input matrix, and the set of all applicable combinations of universally quantified variable values is also embedded in the logic circuit input matrix. Each row of the logic circuit checks that the quantifier-free portion of the formula is satisfied for one case, i.e., one of the applicable combinations of universally quantified variable values. To deal with empty quantification, that is, cases where the bound on a universal variable is zero, there are dummy rows inserted at all points in the universal table where empty quantification occurs. There is a dummy row indicator column which identifies these dummy rows. Additional constraints check that the instance and existential function tables define functions (i.e., the output values are unique for a given vector of input values). Additional constraints check that the bounds are satisfied on all function tables and that the universal table is correct.

4. Compilation of a logic circuit to a trace type carries over all of the column types, fixed columns, equality constrainable columns, and equality constraints in the logic circuit. It must define the step types and their constraints, the subexpression links table, and the result subexpression ids. Each gate constraint in the logic circuit gives rise to one result subexpression id. The step types divide into load step types, lookup step types, constant step types, operator step types, the void step type, and the assert step type. The load step types yield the value of a variable at a case. The lookup step types yield the value of a lookup term at a case. Each constant step type yields the value of a particular constant. The operator step types implement the following operators: $+, \times/\wedge, \vee, \neg, \leftrightarrow, =, < /\text{ind}_<, \max$. The operator step types include short circuiting step types for those operators which can be short circuited. Multiplication $\times$ and boolean AND $\wedge$ are the same operator, and they can be short circuited when an input is zero. $\vee$ can be short circuited when an input is one. The void step type always outputs zero and it takes its own output as each of its inputs. The assert step type is used on all of the result subexpression ids and it is satisfied when the input value is 1. Implementing the equality and inequality comparisons, as well as the max function, relies on byte decomposition as described in [1].

5. Compilation of a trace type to a Halo 2 circuit takes all of the step type constraints and gates them on the values of the step type selection vector. It carries over all of the fixed values and equality constraints. It adds equality constraints stating that the value of the case used column is the same for all rows belonging to the same case. It adds lookup arguments checking that the input column values are consistent with the corresponding subexpression output values; that all subexpressions are linked to proper input subexpressions for their step types according to the links table; and that all result subexpressions are present where applicable.

Future directions include:

1. Finishing the integration of the OSL and argument compiler with Halo 2 will allow zk-SNARKs to be created and verified using this system. This will involve compiling out the lookup argument gate constraints; generating Rust code defining the Halo 2 circuit from its representation as a Haskell data structure; and integrating the Haskell argument compiler with the Rust prover code. These efforts are well underway.

2. Adding more optimizations to the compiler pipeline will result in lower circuit complexity. There are too many optimization opportunities to mention them all here. One worth mentioning is that rows can be utilized more efficiently by allowing for cases to use less than the maximum number of rows per case, thus allowing for more cases to fit into the same number of rows. Currently, the compiler makes use of short circuiting step types, which allows for subexpressions to be omitted where they do not need to be evaluated, but the opportunity to use this to pack more cases into the rows is not capitalized upon.

3. Formally verifying the semantics preservation of the OSL and argument compiler will allow users to be confident that if they correctly expressed what they intended to prove in OSL, then the resulting zk-SNARKs do show what they are supposed to show.

The following appendices go into more detail about some of the core concepts of this paper. These appendices are largely repeated from [3, 4].

A Example of OSL

Here is a simple example of an OSL spec. This spec describes the game of Sudoku. Given a circuit compiler for OSL, this spec could be used to generate zk-SNARKs proving that a given Sudoku problem has a solution, without revealing a solution.

$$\text{data Value} \cong \text{Fin}(9). \tag{3}$$

$$\text{data Row} \cong \text{Fin}(9). \tag{4}$$

$$\text{data Col} \cong \text{Fin}(9). \tag{5}$$

$$\text{data Cell} \cong \text{Row} \times \text{Col}. \tag{6}$$

$$\text{data Problem} \cong \text{Cell} \to \text{Maybe}(\text{Value}). \tag{7}$$

$$\text{data Solution} \cong \text{Cell} \to \text{Value}. \tag{8}$$

$$\text{data Square} \cong \text{Fin}(3) \times \text{Fin}(3). \tag{9}$$

$$\text{data SquareCell} \cong \text{Fin}(3) \times \text{Fin}(3). \tag{10}$$

$$\text{def three} : \mathbb{N} := 1_{\mathbb{N}} + 1_{\mathbb{N}} + 1_{\mathbb{N}} \tag{11}$$

$$\begin{aligned}&\text{def getCell} : \text{Square} \to \text{SquareCell} \to \text{Cell}\\ &\quad := \lambda s : \text{Square} \mapsto \lambda c : \text{SquareCell}\\ &\quad \mapsto \text{let } s' : \text{Fin}(3) \times \text{Fin}(3) := \text{from}(\text{Square})(s);\\ &\quad \text{let } c' : \text{Fin}(3) \times \text{Fin}(3) := \text{from}(\text{SquareCell})(c);\\ &\quad \text{to}(\text{Cell})((\text{to}(\text{Row})(\text{cast}(\text{three} \times_{\mathbb{N}} \text{cast}(\pi_1(s')) +_{\mathbb{N}} \text{cast}(\pi_1(c')))),\\ &\qquad \text{to}(\text{Col})(\text{cast}(\text{three} \times_{\mathbb{N}} \text{cast}(\pi_2(s')) +_{\mathbb{N}} \text{cast}(\pi_2(c')))))).\end{aligned} \tag{12}$$

$$\begin{aligned}&\text{def solutionIsWellFormed} : \text{Solution} \to \text{Prop}\\ &\quad := \lambda s : \text{Solution} \mapsto \text{let } f : \text{Cell} \to \text{Value} := \text{from}(\text{Solution})(s);\\ &\quad (\forall r : \text{Row}, \forall v : \text{Value}, \exists c : \text{Col}, f(\text{to}(\text{Cell})((r, c))) = v)\\ &\quad \wedge (\forall c : \text{Col}, \forall v : \text{Value}, \exists r : \text{Row}, f(\text{to}(\text{Cell})((r, c))) = v)\\ &\quad \wedge (\forall r : \text{Square}, \forall v : \text{Value}, \exists c : \text{SquareCell}, f(\text{getCell}(r, c)) = v).\end{aligned} \tag{13}$$

$$\begin{aligned}&\text{def solutionMatchesProblem} : \text{Problem} \to \text{Solution} \to \text{Prop}\\ &\quad := \lambda p : \text{Problem} \mapsto \lambda s : \text{Solution}\\ &\quad \mapsto \text{let } f : \text{Cell} \to \text{Maybe}(\text{Value}) := \text{from}(\text{Problem})(p);\\ &\quad \text{let } g : \text{Cell} \to \text{Value} := \text{from}(\text{Solution})(s);\\ &\quad \forall c : \text{Cell}, f(c) = \text{nothing} \vee f(c) = \text{just}(g(c)).\end{aligned} \tag{14}$$

$$\begin{aligned}&\text{def problemIsSolvable} : \text{Problem} \to \text{Prop}\\ &\quad := \lambda p : \text{Problem} \mapsto \exists s : \text{Solution}, \text{solutionMatchesProblem}(p, s) \wedge \text{solutionIsWellFormed}(s).\end{aligned} \tag{15}$$

B Grammar of OSL

The grammar of OSL specs is defined by a sequent calculus. This syntactic definition of the language characterizes type correctness as well as what is more usually considered part of syntax in programming language theory. In specifying a programming language, the syntax is often first defined by using a context-free grammar, and then the semantic rules that determine which syntactically valid programs denote semantically valid, type-correct programs are separately defined. The definition of OSL combines these separate concerns, which are sometimes viewed as syntax and semantics, into one context-sensitive grammar defined as a sequent calculus.

For an introduction to reading and understanding sequent calculus systems, readers are referred to [8].

Three syntactic sorts exist in OSL: expressions, judgments, and sequents. An expression may, in a given context, denote a type, a value, or a proposition. A judgment expresses that an expression belongs to a type. A sequent expresses either that a context is valid or that a judgment is true in a given context.

The different expression forms are implied by the following sequent calculus definition of OSL. A judgment takes the form $\bar{x} : \bar{A}$, where $\bar{x}$ and $\bar{A}$ are metavariables denoting expressions. A context is a comma-separated list of zero or more declarations, with the different declaration forms being implied by the sequent calculus definition of OSL. An empty context is written as $()$.

Expressions may contain variable names, which are denoted in the following sequent calculus definitions by metavariables written as lowercase Latin letters: $a, b, \ldots$. Metavariables denoting expressions are written either as lowercase Greek letters $(\alpha, \beta, \ldots)$, or as lowercase Latin letters with overlines $(\bar{a}, \bar{b}, \ldots)$. By convention, Greek letters are used for metavariables for expressions denoting propositions, and Latin letters with overlines are used for metavariables for expressions denoting values.

In this sequent calculus definition, Γ is a metavariable denoting an arbitrary context.

A sequent takes either the form Γ ctx (denoting that the context Γ is valid) or the form $\Gamma \vdash \bar{x} : \alpha$ (denoting that in the context Γ, the judgment $\bar{x} : \alpha$ is true).

Given an expression $\bar{x}$, $\text{free}(\bar{x})$ denotes the set of variable names that occur freely in $\bar{x}$. A free occurrence of a variable name is one that is not bound by a quantifier ($\forall$ or $\exists$), a lambda abstraction (λ), or a let binding.

In OSL expressions, $\rightarrow$ associates to the right. Function application is curried, but written in the usual mathematical notation, so that $f(x)$ denotes the application of f to x, $f(x)(y)$ denotes the application of f to x and y, and $f(x, y)$ denotes the same as $f(x)(y)$.

The first rule allows for derivation with no premises that an empty context is valid:

$$\frac{}{() \text{ ctx}}. \tag{16}$$

The following rules introduce basic typing judgments, stating that Prop (the type of propositions) is a type in any context, and types are closed under formation of function types, including permutation types:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \text{Prop} : \text{Type}} \tag{17}$$

$$\frac{\Gamma \vdash \alpha : \text{Type} \qquad \Gamma \vdash \beta : \text{Type}}{\Gamma \vdash (\alpha \rightarrow \beta) : \text{Type}}. \tag{18}$$

$$\frac{\Gamma \vdash \alpha : \text{Type} \qquad \Gamma \vdash \beta : \text{Type}}{\Gamma \vdash (\alpha \leftrightarrow \beta) : \text{Type}}. \tag{19}$$

Every permutation is a function, and its inverse is a function:

$$\frac{\Gamma \vdash f : \alpha \leftrightarrow \beta}{\Gamma \vdash f : \alpha \rightarrow \beta} \qquad \frac{\Gamma \vdash f : \alpha \leftrightarrow \beta}{\Gamma \vdash f^{-1} : \beta \rightarrow \alpha} \tag{20}$$

The following rule enables introduction of a variable of a given type into a context:

$$\frac{\Gamma \vdash \alpha : \text{Type} \qquad x \notin \text{free}(\Gamma)}{(\Gamma, x : \alpha) \text{ ctx}}. \tag{21}$$

The following rules declare the primitive scalar types. In OSL, there are the following primitive scalar types: natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), native field elements ($\mathbb{F}$), and finite sets of n distinct values ($\mathrm{Fin}(n)$, for $n \in \mathbb{N}$). Throughout this paper, $0 \in \mathbb{N}$.

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \mathbb{N} : \mathrm{Type}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \mathbb{Z} : \mathrm{Type}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \mathbb{F} : \mathrm{Type}} \qquad \frac{\Gamma\ \mathrm{ctx} \qquad n \in \mathbb{N}}{\Gamma \vdash \mathrm{Fin}(n) : \mathrm{Type}} \tag{22}$$

The following rules declare the types of the related functions and constants for the scalar types.

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash +_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \times_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \max_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}} \tag{23}$$

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash 0_{\mathbb{N}} : \mathbb{N}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash 1_{\mathbb{N}} : \mathbb{N}} \tag{24}$$

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash +_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \times_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \max_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}} \tag{25}$$

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash 0_{\mathbb{Z}} : \mathbb{Z}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash 1_{\mathbb{Z}} : \mathbb{Z}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash -1_{\mathbb{Z}} : \mathbb{Z}} \tag{26}$$

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash +_{\mathbb{F}} : \mathbb{F} \to \mathbb{F} \to \mathbb{F}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \times_{\mathbb{F}} : \mathbb{F} \to \mathbb{F} \to \mathbb{F}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \max_{\mathbb{F}} : \mathbb{F} \to \mathbb{F} \to \mathbb{F}} \tag{27}$$

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash 0_{\mathbb{F}} : \mathbb{F}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash 1_{\mathbb{F}} : \mathbb{F}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash -1_{\mathbb{F}} : \mathbb{F}} \tag{28}$$

$$\mathrm{Scalar} = \{\mathbb{N}, \mathbb{Z}, \mathbb{F}\} \cup \{\mathrm{Fin}(n) \mid n \in \mathbb{N}\} \tag{29}$$

$$\frac{\Gamma\ \mathrm{ctx} \qquad \alpha, \beta \in \mathrm{Scalar}}{\Gamma \vdash \mathrm{cast} : \alpha \to \beta} \tag{30}$$

Note that casting from a scalar type to a less extensive scalar type is a partial function, where the result is defined only if the argument is in the codomain.

The following rules introduce typing judgments for (Cartesian) products, coproducts (i.e., sum types), and functions, and their related primitives.

$$\frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \alpha \times \beta : \mathrm{Type}} \qquad \frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash (,) : \alpha \to \beta \to \alpha \times \beta} \tag{31}$$

At times, we write a function application $(,)(x, y)$ in the abbreviated form (x, y).

$$\frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \pi_1 : (\alpha \times \beta) \to \alpha} \qquad \frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \pi_2 : (\alpha \times \beta) \to \beta} \tag{32}$$

$$\frac{\Gamma \vdash \bar{f} : \gamma \to \alpha \qquad \Gamma \vdash \bar{g} : \gamma \to \beta}{\bar{f} \times \bar{g} : \gamma \to (\alpha \times \beta)} \tag{33}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \alpha \oplus \beta : \mathrm{Type}} \tag{34}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \iota_1 : \alpha \to (\alpha \oplus \beta)} \qquad \frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \iota_2 : \beta \to (\alpha \oplus \beta)} \tag{35}$$

$$\frac{\Gamma \vdash \bar{f} : \alpha \to \gamma \qquad \Gamma \vdash \bar{g} : \beta \to \gamma \qquad \Gamma \vdash \gamma : \mathrm{FiniteDim}}{\Gamma \vdash \bar{f} \oplus \bar{g} : (\alpha \oplus \beta) \to \gamma} \tag{36}$$

The restriction $\Gamma \vdash \gamma : \mathrm{FiniteDim}$ is present to satisfy a technical restriction in the translation to Σ^1_1 formulas; translating this function is easier when the result is a series of first-order variables.

The following rules define which types are finite-dimensional. The judgment $\Gamma \vdash \alpha : \mathrm{FiniteDim}$ appears to be a type judgment, but like Type itself, FiniteDim is not a type; it can appear only by itself on the right-hand side of a judgment.

$$\frac{\Gamma\ \mathrm{ctx} \qquad \alpha \in \mathrm{Scalar}}{\Gamma \vdash \alpha : \mathrm{FiniteDim}} \qquad \frac{\Gamma \vdash \alpha : \mathrm{FiniteDim}}{\Gamma \vdash \mathrm{Maybe}(\alpha) : \mathrm{FiniteDim}} \tag{37}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{FiniteDim} \qquad \Gamma \vdash \beta : \mathrm{FiniteDim}}{\Gamma \vdash \alpha \times \beta : \mathrm{FiniteDim}} \tag{38}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{FiniteDim} \qquad \Gamma \vdash \beta : \mathrm{FiniteDim}}{\Gamma \vdash \alpha \oplus \beta : \mathrm{FiniteDim}} \tag{39}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{FiniteDim} \qquad (\mathrm{data}\ x \cong \alpha) \in \Gamma}{\Gamma \vdash x : \mathrm{FiniteDim}} \tag{40}$$

Finally, the function rules are:

$$\frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad \Gamma \vdash \beta : \mathrm{Type}}{\Gamma \vdash \alpha \to \beta : \mathrm{Type}} \tag{41}$$

$$\frac{\Gamma, x : \alpha \vdash \bar{y} : \beta}{\Gamma \vdash (\lambda x : \alpha \mapsto \bar{y}) : \alpha \to \beta} \tag{42}$$

$$\frac{\Gamma \vdash \bar{f} : \alpha \to \beta \qquad \Gamma \vdash \bar{x} : \alpha}{\Gamma \vdash \bar{f}(\bar{x}) : \beta}. \tag{43}$$

The following rules allow for the introduction of new data types that are isomorphic to existing data types (similar to newtypes in Haskell):

$$\frac{\Gamma \vdash \alpha : \mathrm{Type} \qquad x \notin \mathrm{free}(\Gamma)}{(\Gamma, \mathrm{data}\ x \cong \alpha)\ \mathrm{ctx}} \qquad \frac{\Gamma\ \mathrm{ctx} \qquad (\mathrm{data}\ x \cong \alpha) \in \Gamma}{\Gamma \vdash x : \mathrm{Type}} \tag{44}$$

$$\frac{\Gamma\ \mathrm{ctx} \qquad (\mathrm{data}\ x \cong \alpha) \in \Gamma}{\Gamma \vdash \mathrm{to}(x) : \alpha \to x} \qquad \frac{\Gamma\ \mathrm{ctx} \qquad (\mathrm{data}\ x \cong \alpha) \in \Gamma}{\Gamma \vdash \mathrm{from}(x) : x \to \alpha}. \tag{45}$$

The following rules enable the introduction of equational definitions:

$$\frac{\Gamma \vdash \bar{a} : \alpha \qquad x \notin \mathrm{free}(\Gamma)}{(\Gamma, \mathrm{def}\ x : \alpha := \bar{a})\ \mathrm{ctx}} \qquad \frac{\Gamma\ \mathrm{ctx} \qquad (\mathrm{def}\ x : \alpha := \bar{a}) \in \Gamma}{\Gamma \vdash x : \alpha} \tag{46}$$

The following rule allows for the formation of let expressions:

$$\frac{\Gamma, \mathrm{def}\ x : \alpha := \bar{y} \vdash \bar{z} : \beta}{\Gamma \vdash (\mathrm{let}\ x : \alpha := \bar{y}; \bar{z}) : \beta}. \tag{47}$$

The following rules provide the typing rules for the useful functors Maybe and List, along with several basic functions for working with these functors:

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{Maybe}(\alpha) : \mathrm{Type}} \qquad \frac{\Gamma \vdash \bar{f} : \alpha \to \beta}{\Gamma \vdash \mathrm{Maybe}(\bar{f}) : \mathrm{Maybe}(\alpha) \to \mathrm{Maybe}(\beta)} \tag{48}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Type}}{\Gamma \vdash \mathrm{just} : \alpha \to \mathrm{Maybe}(\alpha)} \qquad \frac{\Gamma \vdash \alpha : \mathrm{Type}}{\Gamma \vdash \mathrm{nothing} : \mathrm{Maybe}(\alpha)} \tag{49}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{FiniteDim} \qquad \Gamma \vdash \beta : \mathrm{FiniteDim} \qquad \Gamma \vdash \bar{f} : \alpha \to \beta}{\Gamma \vdash \mathrm{maybe}(\bar{f}) : \beta \to \mathrm{Maybe}(\alpha) \to \beta}. \tag{50}$$

The following function is useful for extracting a value from a Maybe when one is known to be present (with undefined result if no value is present):

$$\frac{\Gamma \vdash \alpha : \mathrm{Type}}{\Gamma \vdash \mathrm{exists} : \mathrm{Maybe}(\alpha) \to \alpha}. \tag{51}$$

The rules for List have a circular dependency with the rules for Quantifiable judgments, defined in (93)-(100).

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\alpha) : \mathrm{Type}} \tag{52}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{length} : \mathrm{List}(\alpha) \to \mathbb{N}} \qquad \frac{\Gamma \vdash \alpha : \mathrm{Type}}{\Gamma \vdash \mathrm{nth} : \mathrm{List}(\alpha) \to \mathbb{N} \to \alpha} \tag{53}$$

The result of nth when the index is out of range is undefined.

Because translating the general functor operation of List on morphisms (functions) is difficult, OSL does not include the general functor operation of List on morphisms. However, OSL does include several special cases of the List functor that are useful and easy to translate:

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable} \qquad \Gamma \vdash \beta : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\pi_1) : \mathrm{List}(\alpha \times \beta) \to \mathrm{List}(\alpha)} \tag{54}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable} \qquad \Gamma \vdash \beta : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\pi_2) : \mathrm{List}(\alpha \times \beta) \to \mathrm{List}(\beta)} \tag{55}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable} \qquad (\mathrm{data}\ x \ \cong \alpha) \in \Gamma}{\Gamma \vdash \mathrm{List}(\mathrm{to}(x)) : \mathrm{List}(\alpha) \to \mathrm{List}(x)} \tag{56}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable} \qquad (\mathrm{data}\ x \cong \alpha) \in \Gamma}{\Gamma \vdash \mathrm{List}(\mathrm{from}(x)) : \mathrm{List}(x) \to \mathrm{List}(\alpha)} \tag{57}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\mathrm{length}) : \mathrm{List}(\mathrm{List}(\alpha)) \to \mathrm{List}(\mathbb{N})} \tag{58}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable} \qquad \Gamma \vdash \beta : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\mathrm{Maybe}(\pi_1)) : \mathrm{List}(\mathrm{Maybe}(\alpha \times \beta)) \to \mathrm{List}(\mathrm{Maybe}(\alpha))} \tag{59}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable} \qquad \Gamma \vdash \beta : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\mathrm{Maybe}(\pi_2)) : \mathrm{List}(\mathrm{Maybe}(\alpha \times \beta)) \to \mathrm{List}(\mathrm{Maybe}(\beta))} \tag{60}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Quantifiable}}{\Gamma \vdash \mathrm{List}(\mathrm{Maybe}(\mathrm{length})) : \mathrm{List}(\mathrm{Maybe}(\mathrm{List}(\alpha))) \to \mathrm{List}(\mathrm{Maybe}(\mathbb{N}))}. \tag{61}$$

Lists of numeric types support a sum operation. The following rules define what a numeric type is:

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \mathbb{N} : \mathrm{Num}} \qquad \frac{\Gamma\ \mathrm{ctx}}{\Gamma \vdash \mathbb{Z} : \mathrm{Num}} \tag{62}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Num} \qquad (\mathrm{data}\ a \cong \alpha) \in \Gamma}{\Gamma \vdash a : \mathrm{Num}}. \tag{63}$$

The following rules define the types of the sum operation:

$$\frac{\Gamma \vdash \alpha : \mathrm{Num}}{\Gamma \vdash \mathrm{sum} : \mathrm{List}(\alpha) \to \alpha} \qquad \frac{\Gamma \vdash \alpha : \mathrm{Num}}{\Gamma \vdash \mathrm{sum} : \mathrm{List}(\mathrm{Maybe}(\alpha)) \to \alpha} \tag{64}$$

$$\frac{\Gamma \vdash \alpha : \text{Num}}{\Gamma \vdash \text{sum} : \text{List}(\text{List}(\alpha)) \to \alpha} \qquad \frac{\Gamma \vdash \alpha : \text{Num}}{\Gamma \vdash \text{sum} : \text{List}(\text{List}(\text{Maybe}(\alpha))) \to \alpha}. \tag{65}$$

The following are the map-related rules:

$$\frac{\Gamma \text{ ctx} \qquad \Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \text{Map}(\alpha, \beta) : \text{Type}} \tag{66}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \text{lookup} : \alpha \to \text{Map}(\alpha, \beta) \to \text{Maybe}(\beta)} \tag{67}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \text{keys} : \text{Map}(\alpha, \beta) \to \text{List}(\alpha)} \tag{68}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable} \qquad \Gamma \vdash \gamma : \text{Quantifiable}}{\Gamma \vdash \text{Map}(\pi_1) : \text{Map}(\alpha, \beta \times \gamma) \to \text{Map}(\alpha, \beta)} \tag{69}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable} \qquad \Gamma \vdash \gamma : \text{Quantifiable}}{\Gamma \vdash \text{Map}(\pi_2) : \text{Map}(\alpha, \beta \times \gamma) \to \text{Map}(\alpha, \gamma)} \tag{70}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable} \qquad (\text{data } x \cong \beta) \in \Gamma}{\Gamma \vdash \text{Map}(\text{to}(x)) : \text{Map}(\alpha, \beta) \to \text{Map}(\alpha, x)} \tag{71}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable} \qquad (\text{data } x \cong \beta) \in \Gamma}{\Gamma \vdash \text{Map}(\text{from}(x)) : \text{Map}(\alpha, \beta) \to \text{Map}(\alpha, x)} \tag{72}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Num}}{\Gamma \vdash \text{sum} : \text{Map}(\alpha, \beta) \to \beta}. \tag{73}$$

The following function compositions are defined as primitives, because they are useful and easy to translate into Σ_1^1 formulas:

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash (\text{sum} \circ \text{Map}(\text{length})) : \text{Map}(\alpha, \text{List}(\beta)) \to \mathbb{N}} \tag{74}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Num} \qquad \Gamma \vdash \bar{k} : \alpha}{\Gamma \vdash (\text{sum} \circ \text{List}(\text{lookup}(\bar{k}))) : \text{List}(\text{Map}(\alpha, \beta)) \to \beta}. \tag{75}$$

The following rule defines the types for which equality propositions can be formed in OSL:

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim}}{\Gamma \vdash \alpha : \text{Eq}}. \tag{76}$$

Of note, although Eq appears to be a type, it is not a type. It cannot appear except by itself on the right-hand side of a judgment.

The following rules pertain to the formation of propositions:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \top : \text{Prop}} \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash \bot : \text{Prop}} \tag{77}$$

$$\frac{\Gamma \vdash \bar{x} : \alpha \qquad \Gamma \vdash \bar{y} : \alpha \qquad \Gamma \vdash \alpha : \text{Eq}}{\Gamma \vdash (\bar{x} = \bar{y}) : \text{Prop}} \tag{78}$$

$$\frac{\Gamma \vdash \bar{x} : \alpha \qquad \Gamma \vdash \bar{y} : \alpha \qquad \Gamma \vdash \alpha : \text{Num}}{\Gamma \vdash (\bar{x} \le \bar{y}) : \text{Prop}} \tag{79}$$

$$\frac{\Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \wedge \psi) : \text{Prop}} \qquad \frac{\Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \vee \psi) : \text{Prop}} \tag{80}$$

The following conditional and biconditional formation rules are formulated with reference to $\neg$ to account for the fact that not every proposition can be negated; only first-order formulas can be negated, and the premise of a conditional is implicitly in the negative position. The subformulas of a biconditional are also in positive position, but it suffices to assume that the negations of the subformulas are propositions, because it follows from $\neg\phi$ being a proposition that ϕ is also a proposition, as can be seen by induction on the length of derivations.

$$\frac{\Gamma \vdash \phi : \text{Prop}}{\Gamma \vdash \neg\phi : \text{Prop}} \qquad \frac{\Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \to \psi) : \text{Prop}}. \tag{81}$$

$$\frac{\Gamma \vdash \neg\phi : \text{Prop} \qquad \Gamma \vdash \neg\psi : \text{Prop}}{\Gamma \vdash (\phi \leftrightarrow \psi) : \text{Prop}}. \tag{82}$$

Because OSL allows for only universal quantification over finite types, the following rules define what a finite type is:

$$\frac{\Gamma \text{ ctx} \qquad n \in \mathbb{N}}{\Gamma \vdash \text{Fin}(n) : \text{Finite}} \qquad \frac{\Gamma \vdash \alpha : \text{Finite}}{\Gamma \vdash \text{Maybe}(\alpha) : \text{Finite}} \tag{83}$$

$$\frac{\Gamma \vdash \alpha : \text{Finite} \qquad \Gamma \vdash \beta : \text{Finite}}{\Gamma \vdash \alpha \times \beta : \text{Finite}} \qquad \frac{\Gamma \vdash \alpha : \text{Finite} \qquad \Gamma \vdash \beta : \text{Finite}}{\Gamma \vdash \alpha \oplus \beta : \text{Finite}} \tag{84}$$

$$\frac{\Gamma \vdash \alpha : \text{Finite} \qquad (\text{data } x \cong \alpha) \in \Gamma}{\Gamma \vdash x : \text{Finite}}. \tag{85}$$

Although Finite appears to be a type, it is not a type. It cannot appear except by itself on the right-hand side of a judgment.

Universal quantifiers can be applied only around propositions that do not contain existential quantifiers over infinite types. To capture this notion, the following rules extend the Finite judgment to propositions.

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \top : \text{Finite}} \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash \bot : \text{Finite}} \tag{86}$$

$$\frac{\Gamma \vdash (\bar{x} = \bar{y}) : \text{Prop}}{\Gamma \vdash (\bar{x} = \bar{y}) : \text{Finite}} \qquad \frac{\Gamma \vdash (\bar{x} \leq \bar{y}) : \text{Prop}}{\Gamma \vdash (\bar{x} \leq \bar{y}) : \text{Finite}} \tag{87}$$

$$\frac{\Gamma \vdash \phi : \text{Finite} \qquad \Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Finite} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \wedge \psi) : \text{Finite}} \tag{88}$$

$$\frac{\Gamma \vdash \phi : \text{Finite} \qquad \Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Finite} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \vee \psi) : \text{Finite}} \tag{89}$$

$$\frac{\Gamma \vdash \phi : \text{Finite} \qquad \Gamma \vdash \phi : \text{Prop}}{\Gamma \vdash \neg\phi : \text{Finite}} \tag{90}$$

$$\frac{\Gamma \vdash \phi : \text{Finite} \qquad \Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Finite} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \to \psi) : \text{Finite}} \tag{91}$$

$$\frac{\Gamma \vdash \phi : \text{Finite} \qquad \Gamma \vdash \phi : \text{Prop} \qquad \Gamma \vdash \psi : \text{Finite} \qquad \Gamma \vdash \psi : \text{Prop}}{\Gamma \vdash (\phi \leftrightarrow \psi) : \text{Finite}} \tag{92}$$

Existential quantifiers can quantify only over types that are built by using only finite-dimensional types in the domains of function types. This class of types is called Quantifiable, and is defined by the following rules.

$$\frac{\Gamma \text{ ctx} \qquad \alpha \in \text{Scalar}}{\Gamma \vdash \text{Fin}(n) : \text{Quantifiable}} \qquad \frac{\Gamma \vdash \alpha : \text{Quantifiable}}{\Gamma \vdash \text{Maybe}(\alpha) : \text{Quantifiable}} \tag{93}$$

$$\frac{\Gamma \vdash \alpha : \text{Quantifiable} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \alpha \times \beta : \text{Quantifiable}} \tag{94}$$

$$\frac{\Gamma \vdash \alpha : \text{Quantifiable} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \alpha \oplus \beta : \text{Quantifiable}} \tag{95}$$

$$\frac{\Gamma \vdash \alpha : \text{Quantifiable} \qquad (\text{data } x \cong \alpha) \in \Gamma}{\Gamma \vdash x : \text{Quantifiable}} \tag{96}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \alpha \to \beta : \text{Quantifiable}} \tag{97}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{FiniteDim}}{\Gamma \vdash \alpha \leftrightarrow \beta : \text{Quantifiable}} \tag{98}$$

$$\frac{\Gamma \vdash \alpha : \text{Quantifiable}}{\Gamma \vdash \text{List}(\alpha) : \text{Quantifiable}} \tag{99}$$

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{Quantifiable}}{\Gamma \vdash \text{Map}(\alpha, \beta) : \text{Quantifiable}} \tag{100}$$

The following rules pertain to the formation of quantified propositions:

$$\frac{\Gamma, x : \alpha \vdash \phi : \text{Prop} \qquad \Gamma, x : \alpha \vdash \phi : \text{Finite} \qquad \Gamma \vdash \alpha : \text{Finite}}{\Gamma \vdash (\forall x : \alpha.\ \phi) : \text{Prop}} \tag{101}$$

$$\frac{\Gamma, x : \alpha \vdash \phi : \text{Prop} \qquad \Gamma \vdash \alpha : \text{Quantifiable}}{\Gamma \vdash (\exists x : \alpha.\ \phi) : \text{Prop}}. \tag{102}$$

The following rules allow for the formation of Finite judgments of quantified propositions:

$$\frac{\Gamma \vdash (\forall x : \alpha.\ \phi) : \text{Prop}}{\Gamma \vdash (\forall x : \alpha.\ \phi) : \text{Finite}} \tag{103}$$

$$\frac{\Gamma, x : \alpha \vdash \phi : \text{Prop} \qquad \Gamma, x : \alpha \vdash \phi : \text{Finite} \qquad \Gamma \vdash \alpha : \text{Finite}}{\Gamma \vdash (\exists x : \alpha.\ \phi) : \text{Finite}}. \tag{104}$$

C OSL denotational semantics

The denotational semantics for OSL assigns denotations to expressions relative to denotational contexts. A denotation is a set. A denotational context consists of a context Γ, a universe of sets U, and a partial function $C : \text{Name} \to U$ that assigns denotations to each unbound variable in Γ. Here, Name denotes the set of OSL variable names. OSL contexts consist of three types of declarations: unbound variable declarations $x : \alpha$, equational definitions def $x : \alpha := \bar{y}$, and data declarations data $x \cong \alpha$. The unbound variables in a context Γ are those occurring on the left-hand sides of variable declarations $(x : \alpha) \in \Gamma$.

The universe U is required to be a Grothendieck universe. A Grothendieck universe is a set in which mathematics can be performed. Specifically, by definition, a set U is a Grothendieck universe if and only if:

1. For all x, y, if $x \in U$ and $y \in x$, then $y \in U$.
2. For all $x, y \in U$, $\{x, y\} \in U$.
3. For all $x \in U$, $\mathcal{P}(X) \in U$. $\mathcal{P}(X)$ is the power set of X, i.e.,

$$\mathcal{P}(X) = \{Y \mid Y \subseteq X\}. \tag{105}$$

4. For all sets $I \in U$ and families $\{x_i\}_{i \in I}$ of elements of U,

$$\left(\bigcup_{i \in I} x_i \right) \in U. \tag{106}$$

Any Grothendieck universe U can be used as the universe of a denotational context. A denotational context is written in the form Γ, U, C, where Γ is the context, U is the universe, and C is the map from names to elements of U.

The following sequent calculus delineates the denotational semantics of OSL, by providing rules for the formation of denotational contexts and judgments of denotations relative to denotational contexts. These rules implicitly define, relative to a denotational context, a partial denotation function Δ from expressions to denotations (i.e., elements of the universe U of the related denotational context). The sequent Γ, U, C dctx expresses that Γ, U, C is a valid denotational context. The sequent $\Gamma, U, C \vdash \Delta(\bar{x}) = X$ expresses that, relative to the denotational context Γ, U, C, the denotation of $\bar{x}$ is X (and $X \in U$ is implied).

$$\frac{\Gamma \text{ ctx} \qquad U \text{ is a Grothendieck universe}}{\Gamma, U, \emptyset \text{ dctx}} \tag{107}$$

$$\frac{\Gamma \text{ ctx} \qquad (x : \alpha) \in \Gamma \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad a \in A}{((\Gamma, x : \alpha), U, C \cup \{(x, a)\}) \text{ dctx}} \tag{108}$$

The various sequent rules all preserve the invariant that if Γ, U, C is a denotational context, then $\text{dom}(C)$ is a subset of the set of variables in $\text{free}(\Gamma)$ that are not bound by data or def declarations.

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{Type}) = \{A \in U \mid \exists \alpha, (\Gamma \vdash \alpha : \text{Type}) \land (\Gamma, U, C \vdash \Delta(\alpha) = A)\}} \tag{109}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{Prop}) = \{0, 1\}} \tag{110}$$

In the following rule, B^A represents the set of functions with domain A and codomain B, where a function is represented as the set of ordered pairs of its inputs and outputs.

$$\frac{\Gamma \vdash \alpha : \text{Type} \qquad \Gamma \vdash \beta : \text{Type} \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad \Gamma, U, C \vdash \Delta(\beta) = B}{\Gamma, U, C \vdash \Delta(\alpha \to \beta) = B^A} \tag{111}$$

In the following rule, $A \sim B$ represents the set of permutations with domain A and codomain B.

$$\frac{\Gamma \vdash \alpha : \text{FiniteDim} \qquad \Gamma \vdash \beta : \text{FiniteDim} \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad \Gamma, U, C \vdash \Delta(\beta) = B}{\Gamma, U, C \vdash \Delta(\alpha \leftrightarrow \beta) = A \sim B} \tag{112}$$

The following rule defines denotations of permutation inverses.

$$\frac{\Gamma \vdash \bar{f} : \alpha \leftrightarrow \beta \qquad \Gamma, U, C \vdash \Delta(\bar{f}) = F}{\Gamma, U, C \vdash \Delta(\bar{f}^{-1}) = \{(y, x) \mid (x, y) \in F\}} \tag{113}$$

In the following rules for basic numeric types, the mentioned denotations, $\mathbb{N}, +_{\mathbb{N}}$, and so forth, are defined within U with their usual extensions, with the set of natural numbers being equated with the set of finite von Neumann ordinals, and the set of integers being equated with the set $\{-1, 1\} \times \mathbb{N}$. $\mathbb{F}$ is to be defined as a subset of $\mathbb{N}$, with different arithmetic operations.

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\mathbb{N}) = \mathbb{N}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\mathbb{Z}) = \mathbb{Z}} \qquad \frac{\Gamma, U, C \text{ dctx} \qquad n \in \mathbb{N}}{\Gamma, U, C \vdash \Delta(\text{Fin}(n)) = \{0, ..., n-1\}} \tag{114}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(+_{\mathbb{N}}) = +_{\mathbb{N}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(+_{\mathbb{Z}}) = +_{\mathbb{Z}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(+_{\mathbb{F}}) = +_{\mathbb{F}}} \tag{115}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\times_{\mathbb{N}}) = \times_{\mathbb{N}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\times_{\mathbb{Z}}) = \times_{\mathbb{Z}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\times_{\mathbb{F}}) = \times_{\mathbb{F}}} \tag{116}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\times_{\mathbb{N}}) = \times_{\mathbb{N}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\times_{\mathbb{Z}}) = \times_{\mathbb{Z}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\times_{\mathbb{F}}) = \times_{\mathbb{F}}} \tag{117}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(0_{\mathbb{N}}) = 0_{\mathbb{N}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(1_{\mathbb{N}}) = 1_{\mathbb{N}}} \tag{118}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(0_{\mathbb{Z}}) = 0_{\mathbb{Z}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(1_{\mathbb{Z}}) = 1_{\mathbb{Z}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(-1_{\mathbb{Z}}) = -1_{\mathbb{Z}}} \tag{119}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(0_{\mathbb{Z}}) = 0_{\mathbb{F}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(1_{\mathbb{Z}}) = 1_{\mathbb{F}}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(-1_{\mathbb{Z}}) = -1_{\mathbb{F}}} \tag{120}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{cast}) = \{(x, x) \mid x \in \mathbb{Z}\} \cup \{(x, (1, x)) \mid x \in \mathbb{Z}\}} \tag{121}$$

$$\frac{\Gamma, U, C \text{ dctx} \qquad n \in \mathbb{N}}{\Gamma, U, C \vdash \Delta(\text{fin}(n)) = n} \tag{122}$$

The following rules provide denotations for product types and related functions:

$$\frac{\Gamma \vdash \alpha : \text{Type} \qquad \Gamma \vdash \beta : \text{Type} \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad \Gamma, U, C \vdash \Delta(\beta) = B}{\Gamma, U, C \vdash \Delta(\alpha \times \beta) = A \times B}. \tag{123}$$

Here, the set $A \times B$ is defined in the usual manner, as the set of ordered pairs (a, b) where $a \in A$ and $b \in B$. Ordered pairs are defined in the usual manner, with the pair (a, b) being encoded as the set $\{\{a\}, \{a, b\}\}$ (which contains sufficient information to discern the elements of the pair and which one is first).

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\pi_1) = \{((x, y), x) \mid \exists A, B \in \Delta(\text{Type}), x \in A \wedge y \in B\}} \tag{124}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\pi_2) = \{((x, y), y) \mid \exists A, B \in \Delta(\text{Type}),\ x \in A \wedge y \in B\}} \tag{125}$$

$$\frac{\Gamma \vdash \bar{f} : \gamma \to \alpha \qquad \Gamma \vdash \bar{G} : \gamma \to \beta \qquad \Gamma, U, C \vdash \Delta(\bar{f}) = F \qquad \Gamma, U, C \vdash \Delta(\bar{g}) = G}{\Gamma, U, C \vdash \Delta(\bar{f} \times \bar{g}) = \{(x, (y, z)) \mid (x, y) \in F \wedge (x, z) \in G\}} \tag{126}$$

The following rules provide denotations for coproduct types and related functions:

$$\frac{\Gamma \vdash \alpha : \text{Type} \qquad \Gamma \vdash \beta : \text{Type} \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad \Gamma, U, C \vdash \Delta(\beta) = B}{\Gamma, U, C \vdash \Delta(\alpha \oplus \beta) = (\{0\} \times A) \cup (\{1\} \times B)} \tag{127}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\iota_1) = \{(x, (0, x)) \mid \exists A \in \Delta(\text{Type}),\ x \in A\}} \tag{128}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\iota_2) = \{(x, (1, x)) \mid \exists A \in \Delta(\text{Type}),\ x \in A\}} \tag{129}$$

$$\frac{\Gamma \vdash \bar{f} : \alpha \to \gamma \qquad \Gamma \vdash \bar{G} : \beta \to \gamma \qquad \Gamma, U, C \vdash \Delta(\bar{f}) = F \qquad \Gamma, U, C \vdash \Delta(\bar{g}) = G}{\Gamma, U, C \vdash \Delta(\bar{f} \oplus \bar{g}) = \{((0, x), y) \mid (x, y) \in F\} \cup \{((1, x), y) \mid (x, y) \in G\}}. \tag{130}$$

For a function $C : \text{Name} \to U$, the notation $C[v \mapsto x]$, where $v \in \text{Name}$ and $x \in U$, denotes the partial function:

$$C[v \mapsto x](u) = \begin{cases} x & \text{if } u = v, \\ C(u) & \text{otherwise.} \end{cases} \tag{131}$$

The following rules provide denotations for lambda abstractions and function applications:

$$\frac{\Gamma, v : \alpha \vdash \bar{y} : \beta \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad \Gamma, U, C \vdash \Delta(\beta) = B}{\Gamma, U, C \vdash \Delta(\lambda v : \alpha \mapsto \bar{y}) = \{(x, y) \mid x \in A \wedge y \in B \wedge (\Gamma, U, C[v \mapsto x]) \vdash \Delta(\bar{y}) = y)\}} \tag{132}$$

$$\frac{\Gamma \vdash \bar{f} : \alpha \to \beta \qquad \Gamma \vdash \bar{x} : \alpha \qquad \Gamma \vdash \Delta(\bar{f}) = F \qquad \Gamma \vdash \Delta(\bar{x}) = x \qquad (x, y) \in F}{\Gamma, U, C \vdash \Delta(\bar{f}(\bar{x})) = y}. \tag{133}$$

The following rules define denotations of names introduced by equational definitions and data declarations:

$$\frac{\Gamma, U, C \vdash \Delta(\alpha) = A \qquad (\text{data } x \cong \alpha) \in \Gamma}{\Gamma, U, C \vdash \Delta(x) = A} \qquad \frac{\Gamma, U, C \vdash \Delta(\bar{y}) = Y \qquad (\text{def } x : \alpha := \bar{y}) \in \Gamma}{\Gamma, U, C \vdash \Delta(x) = Y}. \tag{134}$$

The following rules define denotations of to/from isomorphisms, which are quite trivial, because these isomorphisms are always the identity function over the denotation set:

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{to}(x)) = \{(x, x) \mid \exists A \in \Delta(\text{Type}), x \in A\}} \tag{135}$$

$$\frac{\Gamma, U, C \vdash \qquad (\text{data } x \cong \alpha) \in \Gamma}{\Gamma, U, C \vdash \Delta(\text{from}(x)) = \{(x, x) \mid x \in A\}}. \tag{136}$$

The following rule defines denotations of let expressions:

$$\frac{\Gamma, U, C \vdash \Delta(\bar{y}) = Y \qquad \Gamma, U, C[x \mapsto Y] \vdash \Delta(\bar{z}) = Z}{\Gamma, U, C \vdash \Delta(\text{let } x : \alpha := \bar{y}; \bar{z}) = Z}. \tag{137}$$

The following rules define denotations of the Maybe functor and related functions:

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{rclcl} \Gamma, U, C \vdash \Delta(\text{Maybe}) & = & \{(A, \{0\} \cup (\{1\} \times A)) & \mid & A \in \Delta(\text{Type})\} \\ & \cup & \{(f, \{(0,0)\} \cup (\{1\} \times f)) & \mid & \exists \alpha, \beta, (\Gamma \vdash \alpha : \text{Type}) \\ & & & \wedge & (\Gamma \vdash \beta : \text{Type}) \\ & & & \wedge & (\Gamma, U, C \vdash \Delta(\alpha \to \beta) = D) \\ & & & \wedge & f \in D\} \end{array}} \tag{138}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{just}) = \{(x, (1, x)) \mid \exists A \in \Delta(\text{Type}), x \in A\}} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{nothing}) = 0}. \tag{139}$$

For a function $f : A \times B \to C$, let $\text{curry}(f) : A \to B \to C$ be a function defined as follows:

$$\text{curry}(f) = \{(a, \{(b, f(a, b)) \mid b \in B\}) \mid a \in A\}. \tag{140}$$

Similarly, for a function $f : A \times B \times C \to D$, let $\text{curry}^2(f) : A \to B \to C \to D$ be a function defined as follows:

$$\text{curry}^2(f) = \text{curry}(\text{curry}(f)). \tag{141}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{lcl}\Gamma, U, C \vdash \Delta(\text{maybe}) = \text{curry}^2(\{(f, d, x, y) & | & \exists\alpha,\ \Gamma \vdash \alpha : \text{FiniteDim} \\ & \wedge & \exists\beta,\ \Gamma \vdash \beta : \text{FiniteDim} \\ & \wedge & f \in \Delta(\alpha \to \beta) \\ & \wedge & d \in \Delta(\beta) \\ & \wedge & x \in \Delta(\text{Maybe}(\alpha)) \\ & \wedge & ((x = \text{nothing} \wedge y = d) \vee (x, y) \in f)\})\end{array}} \tag{142}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{exists}) = \{((1, x), x) \mid \exists A \in \Delta(\text{Type}), x \in A\}} \tag{143}$$

The following rules define denotations of the List functor and related functions:

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{lcl}\Gamma, U, C \vdash \Delta(\text{List}) & = & \{(A, \mathbb{N} \times A^{\mathbb{N}}) & | & A \in \Delta(\text{Type})\} \\ & \cup & \{(f, (\text{id}_{\mathbb{N}}, g \mapsto f \circ g)) & | & \exists\alpha, \beta, f \in \Delta(\alpha \to \beta) \\ & & & \wedge & \Gamma \vdash \alpha : \text{Quantifiable} \\ & & & \wedge & \Gamma \vdash \beta : \text{Quantifiable}\}\end{array}} \tag{144}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\text{length}) = \{((\ell, f), \ell) \mid \exists\alpha, (\Gamma \vdash \alpha : \text{Quantifiable}) \wedge (\ell, f) \in \Delta(\text{List}(\alpha))\}} \tag{145}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{lcl}\Gamma, U, C \vdash \Delta(\text{nth}) = \text{curry}(\{((\ell, f), i, x) & | & \exists\alpha, (\Gamma \vdash \alpha : \text{Quantifiable}) \\ & \wedge & (\ell, f) \in \Delta(\text{List}(\alpha)) \\ & \wedge & i \in \mathbb{N} \wedge (i, x) \in f \wedge i < \ell\}).\end{array}} \tag{146}$$

The following rules define denotations of the Map functor and related functions:

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{l}\Gamma, U, C \vdash \Delta(\text{Map}) = \\ \quad \text{curry}(\{(A, B, \mathbb{N} \times A^{\mathbb{N}} \times B^{A}) \mid \exists\alpha, (\Gamma \vdash \alpha : \text{FiniteDim}) \wedge \exists\beta, (\Gamma \vdash \beta : \text{Quantifiable}) \\ \qquad \wedge (\Gamma, U, C \vdash \Delta(\alpha) = A) \wedge (\Gamma, U, C \vdash \Delta(\beta) = B)\}) \\ \quad \cup \{(f, \text{id}_{\mathbb{N}} \times \text{id}_{A^{\mathbb{N}}} \times (g \mapsto f \circ g)) \\ \qquad \mid \exists\alpha, \Gamma \vdash \alpha : \text{FiniteDim} \wedge \exists\beta, (\Gamma \vdash \beta : \text{Quantifiable}) \wedge \exists\gamma, (\Gamma \vdash \gamma : \text{Quantifiable}) \\ \qquad \wedge (\Gamma, U, C \vdash \Delta(\alpha) = A) \wedge f \in \Delta(\beta \to \gamma)\}\end{array}} \tag{147}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{l}\Gamma, U, C \vdash \Delta(\text{lookup}) = \text{curry}(\{(i, (\ell, k, v), x) \\ \quad \mid \exists\alpha, (\Gamma \vdash \alpha : \text{FiniteDim}) \wedge \exists\beta, (\Gamma \vdash \beta : \text{FiniteDim}) \\ \quad \wedge (\ell, k, v) \in \Delta(\text{Map}(\alpha, \beta)) \\ \quad \wedge (x = \text{nothing} \vee \exists j \in \mathbb{N}, j < \ell \wedge (j, i) \in k \wedge (i, x') \in v \wedge x = \text{just}(x')\}))\end{array}} \tag{148}$$

$$\frac{\Gamma, U, C \text{ dctx}}{\begin{array}{l}\Gamma, U, C \vdash \Delta(\text{keys}) = \{((\ell, k, v), (\ell, k)) \\ \quad \mid \exists\alpha, (\Gamma \vdash \alpha : \text{FiniteDim}) \wedge \exists\beta, (\Gamma \vdash \beta : \text{Quantifiable}) \\ \quad \wedge (\ell, k, v) \in \Delta(\text{Map}(\alpha, \beta))\}.\end{array}} \tag{149}$$

This semantics does not define a denotation for the sum function, because that function is polymorphic, and it will not necessarily work the same on all values regardless of the type that those values are interpreted as belonging to. Therefore, this semantics merely defines the denotations of applications of the sum function:

$$\frac{\Gamma \vdash \bar{x} : \mathrm{List}(\alpha) \qquad \Gamma \vdash \alpha : \mathrm{Num} \qquad \Gamma, U, C \vdash \Delta(\bar{x}) = (\ell, f)}{\Gamma, U, C \vdash \Delta(\mathrm{sum}(\bar{x})) = \sum_{i=0}^{\ell-1} f(i)} \tag{150}$$

$$\frac{\Gamma \vdash \bar{x} : \mathrm{List}(\mathrm{Maybe}(\alpha)) \qquad \Gamma \vdash \alpha : \mathrm{Num} \qquad \Gamma, U, C \vdash \Delta(\bar{x}) = (\ell, f)}{\Gamma, U, C \vdash \Delta(\mathrm{sum}(\bar{x})) = \sum\limits_{i=0}^{\ell-1} \begin{cases} 0 & \text{if } f(i) = 0, \\ \pi_2(f(i)) & \text{otherwise} \end{cases}} \tag{151}$$

$$\frac{\Gamma \vdash \bar{x} : \mathrm{List}(\mathrm{List}(\alpha)) \qquad \Gamma \vdash \alpha : \mathrm{Num} \qquad \Gamma, U, C \vdash \Delta(\bar{x}) = (\ell, f)}{\Gamma, U, C \vdash \Delta(\mathrm{sum}(\bar{x})) = \sum_{i=0}^{\ell-1} \sum_{j=0}^{\pi_1(f(i))-1} \pi_2(f(i))(j)} \tag{152}$$

$$\frac{\Gamma \vdash \bar{x} : \mathrm{List}(\mathrm{List}(\mathrm{Maybe}(\alpha))) \qquad \Gamma \vdash \alpha : \mathrm{Num} \qquad \Gamma, U, C \vdash \Delta(\bar{x}) = (\ell, f)}{\Gamma, U, C \vdash \Delta(\mathrm{sum}(\bar{x})) = \sum\limits_{i=0}^{\ell-1} \sum\limits_{j=0}^{\pi_1(f(i))-1} \begin{cases} 0 & \text{if } \pi_2(f(i))(j) = 0, \\ \pi_2(\pi_2(f(i))(j)) & \text{otherwise} \end{cases}} \tag{153}$$

$$\frac{\Gamma \vdash \bar{x} : \mathrm{Map}(\alpha, \beta) \qquad \Gamma, U, C \vdash \Delta(\bar{x}) = (\ell, k, v)}{\Gamma, U, C \vdash \Delta(\mathrm{sum}(\bar{x})) = \sum_{i=0}^{\ell-1} v(k(i))} \tag{154}$$

$$\frac{\Gamma \vdash \beta : \mathrm{Num} \qquad \Gamma \vdash \bar{x} : \mathrm{Map}(\alpha, \mathrm{List}(\beta)) \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta((\mathrm{sum} \circ \mathrm{Map}(\mathrm{length}))(\bar{x})) = \Delta(\mathrm{sum}(\mathrm{Map}(\mathrm{length})(\bar{x})))} \tag{155}$$

$$\frac{\Gamma \vdash \beta : \mathrm{Num} \qquad \Gamma \vdash \bar{x} : \mathrm{List}(\mathrm{Map}(\alpha, \beta)) \qquad \Gamma \vdash \bar{k} : \alpha \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta((\mathrm{sum} \circ \mathrm{List}(\mathrm{lookup}(\bar{k})))(\bar{x})) = \Delta(\mathrm{sum}(\mathrm{List}(\mathrm{lookup}(\bar{k}))(\bar{x})))}. \tag{156}$$

The following rules define denotations of propositions, which are truth values (1 = true, 0 = false):

$$\frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\top) = 1} \qquad \frac{\Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\bot) = 0} \tag{157}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{FiniteDim} \qquad \Gamma \vdash \bar{x} : \alpha \qquad \Gamma \vdash \bar{y} : \alpha \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\bar{x} = \bar{y}) = \begin{cases} 1 & \text{if } \Delta(\bar{x}) = \Delta(\bar{y}), \\ 0 & \text{otherwise} \end{cases}} \tag{158}$$

$$\frac{\Gamma \vdash \alpha : \mathrm{Num} \qquad \Gamma \vdash \bar{x} : \alpha \qquad \Gamma \vdash \bar{y} : \alpha \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\bar{x} \leq \bar{y}) = \begin{cases} 1 & \text{if } \Delta(\bar{x}) \leq \Delta(\bar{y}), \\ 0 & \text{otherwise} \end{cases}} \tag{159}$$

$$\frac{\Gamma \vdash \phi : \mathrm{Prop} \qquad \Gamma \vdash \psi : \mathrm{Prop} \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\phi \wedge \psi) = \Delta(\phi) \cdot \Delta(\psi)} \tag{160}$$

$$\frac{\Gamma \vdash \phi : \mathrm{Prop} \qquad \Gamma \vdash \psi : \mathrm{Prop} \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\phi \vee \psi) = \Delta(\phi) + \Delta(\psi) - (\Delta(\phi) \cdot \Delta(\psi))} \tag{161}$$

$$\frac{\Gamma \vdash \phi : \mathrm{Prop} \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\neg\phi) = 1 - \Delta(\phi)} \tag{162}$$

$$\frac{\Gamma \vdash \neg\phi : \mathrm{Prop} \qquad \Gamma \vdash \psi : \mathrm{Prop} \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\phi \to \psi) = \Delta(\neg\phi \vee \psi)} \tag{163}$$

$$\frac{\Gamma \vdash \neg\phi : \mathrm{Prop} \qquad \Gamma \vdash \neg\psi : \mathrm{Prop} \qquad \Gamma, U, C \text{ dctx}}{\Gamma, U, C \vdash \Delta(\phi \leftrightarrow \psi) = \begin{cases} 1 & \text{if } \exists v \in \{0,1\}.\ (\Gamma, U, C \vdash \Delta(\phi) = v) \wedge (\Gamma, U, C \vdash \Delta(\psi) = v), \\ 0 & \text{otherwise.} \end{cases}} \tag{164}$$

$$\frac{\Gamma, x : \alpha \vdash \phi : \text{Prop} \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad v \in \{0,1\}^A \qquad \forall i \in A, (\Gamma, U, C[x \mapsto i] \vdash \Delta(\phi) = v(i))}{\Gamma, U, C \vdash \Delta(\forall x : \alpha, \phi) = \begin{cases} 1 & \text{if } v = A \times \{1\} \\ 0 & \text{otherwise} \end{cases}} \tag{165}$$

$$\frac{\Gamma, x : \alpha \vdash \phi : \text{Prop} \qquad \Gamma, U, C \vdash \Delta(\alpha) = A \qquad v \in \{0,1\}^A \qquad \forall i \in A, (\Gamma, U, C[x \mapsto i] \vdash \Delta(\phi) = v(i))}{\Gamma, U, C \vdash \Delta(\exists x : \alpha, \phi) = \begin{cases} 0 & \text{if } v = A \times \{0\} \\ 1 & \text{otherwise.} \end{cases}} \tag{166}$$

D Grammar of Σ_1^1 formulas

The first step in defining Σ_1^1 formulas is to define a language of terms. A term is a syntactic object which denotes a number, given a context which bestows values to the variables it contains. Terms are defined by the following recursive definition.

1. For each positive integer i, x_i is a term. x_i is called a first-order variable.
2. For each positive integer i and all sequences of terms $\tau_1, ..., \tau_n$, $f_i^n(\tau_1, ..., \tau_n)$ is a term. $f_i^n(\tau_1, ..., \tau_n)$ is called a function application. Notice that in this definition, second order variables contain their arity as part of their name.
3. For all terms τ, μ:
   (a) $(\tau + \mu)$ is a term.
   (b) $(\tau \cdot \mu)$ is a term.
   (c) $\text{ind}_<(\tau, \mu)$ is a term. $\text{ind}_<$ is called the comparison indicator function; it returns 1 when τ is less than μ and 0 otherwise.
   (d) $\max(\tau, \mu)$ is a term, denoting the greatest of the denotations of τ and μ.
4. 0 is a term. 1 is a term. -1 is a term. These are called constant symbols.

First-order formulas (over the language of rings) are defined by the following recursive definition.

1. $\perp$ is a formula (always denoting falsehood), and $\top$ is a formula (always denoting truth).
2. For all terms τ, μ,
$$(\tau = \mu) \tag{167}$$
is a first-order formula. $\tau = \mu$ is called an atomic formula or an equation.
3. For all first-order formulas ϕ,
$$\neg\phi \tag{168}$$
is a first-order formula. $\neg\phi$ is called a negation.
4. For all first-order formulas ϕ, ψ,
$$(\phi \wedge \psi) \tag{169}$$
is a first-order formula. $(\phi \wedge \psi)$ is called a conjunction.

5. For all first-order formulas ϕ, ψ,
$$(\phi \vee \psi) \tag{170}$$
is a first-order formula. $(\phi \vee \psi)$ is called a disjunction.

6. For all first-order formulas ϕ, ψ,
$$(\phi \rightarrow \psi) \tag{171}$$
is a first-order formula. $(\phi \rightarrow \psi)$ is called an implication.

7. For all first-order formulas ϕ, ψ,
$$(\phi \leftrightarrow \psi) \tag{172}$$
is a first-order formula, called a biconditional.

8. For all first-order formulas ϕ and terms β,
$$\forall < \beta.\ \phi \tag{173}$$
is a first-order formula. $\forall$ is called the universal quantifier. β is called the quantifier bound.

9. For all first-order formulas ϕ and terms β,
$$\exists < \beta.\ \phi \tag{174}$$
is a first-order formula. $\exists$ is called the first-order existential quantifier. β is called the quantifier bound.

Σ^1_1 formulas without instance quantifiers are defined by the following recursive definition.

1. Every first-order formula is a Σ^1_1 formula without instance quantifiers.
2. For all Σ^1_1 formulas without instance quantifiers ϕ and terms γ and non-empty sequences of terms $\beta_1, ..., \beta_n$,
$$\exists_f < \gamma(< \beta_1, ..., < \beta_n).\ \phi \tag{175}$$
is a Σ^1_1 formula without instance quantifiers. $\exists_f$ is called the second-order existential quantifier. n is called the arity of the function. $\beta_1, ..., \beta_n$ are called the bounds of the dimensions of the domain of the function. γ is called the bound of the codomain of the function.

Σ^1_1 formulas are defined by the following recursive definition.

1. Every Σ^1_1 formula without instance quantifiers is a Σ^1_1 formula.
2. For all Σ^1_1 formula ϕ and terms γ and non-empty sequences of term $\beta_1, ..., \beta_n$,
$$\lambda < \gamma(< \beta_1, ..., < \beta_n).\ \phi \tag{176}$$
is a Σ^1_1 formula. λ is called the instance quantifier. n, γ, and $\beta_1, ..., \beta_n$ are called as in the preceding clause.

E Denotational semantics of Σ^1_1 formulas

Relative to a suitable model, it is possible to define whether any given Σ^1_1 formula without instance quantifiers is true or false. For this context, a model, by definition, is a tuple

$$M = (R, \cdot, +, 0, 1, -1, <, F, S), \tag{177}$$

where:

1. $(R, \cdot, +, 0, 1, -1)$ is a ring:
   (a) R is a set.
   (b) $\cdot : R \times R \to R$ is a binary operation.
   (c) $+ : R \times R \to R$ is a binary operation.
   (d) $0, 1, -1 \in R$.
   (e) $+$ is associative and commutative, meaning, for all $x, y, z \in R$,
   $$(x + y) + z = x + (y + z), \tag{178}$$
   $$x + y = y + x. \tag{179}$$
   (f) Each element of R has an additive inverse, meaning, for each $x \in R$ there is $y \in R$ such that $x + y = 0$.
   (g) $\cdot$ is associative, meaning, for all $x, y, z \in R$,
   $$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{180}$$
   (h) 0 is the additive identity, meaning, for each $x \in R$,
   $$x + 0 = x = 0 + x. \tag{181}$$
   (i) 1 is the multiplicative identity, meaning, for each $x \in R$,
   $$x \cdot 1 = x = 1 \cdot x. \tag{182}$$
   (j) -1 is the additive inverse of 1, meaning $1 + (-1) = 0$.
   (k) The distributive law holds, meaning, for all $x, y, z \in R$,
   $$x \cdot (y + z) = (x \cdot y) + (x \cdot z), \tag{183}$$
   $$(y + z) \cdot x = (y \cdot x) + (z \cdot x). \tag{184}$$
2. $< \subseteq R \times R$ is a strict total ordering relation on R with a least element:
   (a) $<$ is antisymmetric, meaning there is no $x \in R$ such that $x < x$.
   (b) $<$ is transitive, meaning for all $x, y, z \in R$, if $x < y$ and $y < z$ then $x < z$.
   (c) $<$ is connected, meaning for all $x, y \in R$, if $x \neq y$ then either $x < y$ or $y < x$.
   (d) $<$ has a least element, meaning there exists a $z \in R$ such that for all $x \in R$, if $x \neq z$ then $z < x$. (Such a z is necessarily unique.)

3. $F : \mathbb{Z}^+ \to R$ is a partial function mapping de Bruijn indices naming first-order variables to their corresponding values (if any).

4. $S : \mathbb{Z}^+ \to \sum_{i\in\mathbb{Z}^+}(R^i \to R)$ is a partial function mapping de Bruijn indices naming second-order variables to their corresponding values (if any). The values are partial functions of variable arity from R to R. Here $\sum_{i\in\mathbb{Z}^+}$ denotes the indexed coproduct or disjoint union operation with i ranging over the positive integers.

Let $M = (R, \cdot, +, 0, 1, -1, <, F, S)$ be a model. Let $y \in R$. Define the model

$$M[x_1 \mapsto y] = (R, \cdot, +, 0, 1, <, F', S) \tag{185}$$

by letting

$$\begin{array}{rcl} F'(1) & = & y, \\ F'(n+1) & = & F(n). \end{array} \tag{186}$$

Let n be a positive integer. Let $g : R^n \to R$ be a partial function. Define the model

$$M[f_1 \mapsto g] = (R, \cdot, +, 0, 1, F, S') \tag{187}$$

by letting

$$\begin{array}{rcl} S'(1) & = & g, \\ S'(n+1) & = & S(n). \end{array} \tag{188}$$

The definitions just given of $M[x_1 \mapsto y]$ and $M[f_1 \mapsto g]$ explain how to update a model with a new variable mapping at the least de Bruijn index, pushing up all the existing de Bruijn index mappings. These operations are useful for dealing with quantification in the denotational semantics which follows below.

Also helpful for defining the denotational semantics will be an extension of the $[i]$ notation previously defined. Previously, $[i]$ was defined as $\{1, ..., i\}$ for a positive integer i. Generalizing this to a ring R for an arbitrary model M, let $[i]$ be defined as $\{x \in R \mid z \leq x \leq i\}$, where z is the least element of R under $<$ and $\leq$ is the non-strict version of $<$.

Given a model,

$$M := (R, \cdot, +, 0, 1, -1, <, F, S), \tag{189}$$

it is possible to define the denotations of terms and formulas, such as by the following recursive definition. The denotation of a term is an element of R, whereas the denotation of a formula is a truth value. The set of truth values is the set $\{0, 1\}$, where 0 represents false and 1 represents true. Due to the partiality of F, S, and the functions in the codomain of S, not every term or formula has a denotation in every model. The following recursive clauses define the denotation $\delta_M(\tau)$ for each term τ which has a denotation in M and the denotation $\delta_M(\phi)$ for each term ϕ which has a denotation in M.

1. For all positive integers i,
$$\delta_M(x_i) := F(i), \tag{190}$$
if $F(i)$ is defined.

2. For all positive integers i and non-empty sequences of terms $\tau_1, ..., \tau_n$,
$$\delta_M(f_i(\tau_1, ..., \tau_n)) := S(i)(\delta_M(\tau_1), ..., \delta_M(\tau_n)), \tag{191}$$

if $S(i)$ is defined, and $\delta_M(\tau_i)$ is defined for each i, and

$$S(i)(\delta_M(\tau_1), ..., \delta_M(\tau_n)) \tag{192}$$

is defined.

3. For all terms τ, μ,
$$\delta_M(\tau + \mu) := \delta_M(\tau) + \delta_M(\mu), \tag{193}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined.

4. For all terms τ, μ,
$$\delta_M(\tau \cdot \mu) := \delta_M(\tau) \cdot \delta_M(\mu), \tag{194}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined.

5. For all terms τ, μ,
$$\delta_M(\mathrm{ind}_<(\tau, \mu)) := \begin{cases} 1 & \text{if } \delta_M(\tau) < \delta_M(\mu), \\ 0 & \text{otherwise.} \end{cases} \tag{195}$$

6. For all τ, μ,
$$\delta_M(\max(\tau, \mu)) := \begin{cases} \delta_M(\tau) & \text{if } \delta_M(\tau) < \delta_M(\mu), \\ 0 & \text{otherwise.} \end{cases} \tag{196}$$

7. $\delta_M(0) := 0$.

8. $\delta_M(1) := 1$.

9. $\delta_M(-1) := -1$.

10. For all terms τ, μ,
$$\delta_M(\tau = \mu) := 1 \tag{197}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined and
$$\delta_M(\tau) = \delta_M(\mu). \tag{198}$$

11. For all terms τ, μ,
$$\delta_M(\tau = \mu) := 0 \tag{199}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined and
$$\delta_M(\tau) \neq \delta_M(\mu). \tag{200}$$

12. For all first-order formulas ϕ,
$$\delta_M(\neg\phi) := 1 - \delta_M(\phi), \tag{201}$$
if $\delta_M(\phi)$ is defined.

13. For all first-order formulas ϕ, ψ,
$$\delta_M(\phi \wedge \psi) := \min\{\delta_M(\phi), \delta_M(\psi)\}, \tag{202}$$
if $\delta_M(\phi)$ and $\delta_M(\psi)$ are defined.

14. For all first-order formulas ϕ, ψ,

$$\delta_M(\phi \vee \psi) := \max\{\delta_M(\phi), \delta_M(\psi)\}, \tag{203}$$

if $\delta_M(\phi)$ and $\delta_M(\psi)$ are defined.

15. For all terms β and first-order formulas ϕ,

$$\delta_M(\forall < \beta.\ \phi) := \min(\{\delta_{M[x_1 \mapsto i]}(\phi) | i \in [\delta_M(\beta)]\} \cup \{1\}), \tag{204}$$

if $\delta_{M[x_1 \mapsto i]}(\phi)$ is defined for each i.

16. For all terms β and first-order formulas ϕ,

$$\delta_M(\exists < \beta.\ \phi) := \max(\{\delta_{M[x_1 \mapsto i]}(\phi) | i \in [\delta_M(\beta)]\} \cup \{0\}), \tag{205}$$

if $\delta_{M[x_1 \mapsto i]}(\phi)$ is defined for each i.

17. For all terms γ and non-empty sequences of terms $\beta_1, ..., \beta_n$ and Σ^1_1 formulas without instance quantifiers ϕ,

$$\delta_M(\exists_f < \gamma(< \beta_1, ..., < \beta_n).\ \phi) := \max_{g \in [\delta_{M[x_{n-i} \mapsto v_i]_{i=0}^{n-1}}(\gamma)]^{\prod_{j \in [n]} v_j \in [\delta_{M[x_{j-1-i}]_{i=0}^{j-1}}(\beta_j)]}} \delta_{M[f_1 \mapsto g]}(\phi), \tag{206}$$

if $\delta_{M_n[f_1 \mapsto g]}(\phi)$ is defined for each g.

Note: this truth condition cannot assign a truth value to the formula when $\delta_M(\gamma) < 1$. In such a case, the only value that g can take is the empty set, and if the formula ϕ references f_1, then $\delta_{M[f_1 \mapsto \emptyset]}(\phi)$ is undefined. Since we do not care about cases where ϕ does not reference all quantified variables, it is not important for our purposes that this truth condition fails to apply when $\delta_M(\gamma) < 1$. The same comments apply when $\delta_M(\beta_i) < 1$ for some i.

Extending the denotational semantics to Σ^1_1 formulas in general requires allowing a denotation of a formula to possibly be a function. For all terms γ and non-empty sequences of terms $\beta_1, ..., \beta_n$ and Σ^1_1 formulas ϕ,

$$\delta_M(\lambda < \gamma(< \beta_1, ..., < \beta_n).\ \phi) := \max_{g \in [\delta_{M[x_{n-i} \mapsto v_i]_{i=0}^{n-1}}(\gamma)]^{\prod_{j \in [n]} v_j \in [\delta_{M[x_{j-1-i}]_{i=0}^{j-1}}(\beta_j)]}} \delta_{M[f_1 \mapsto g]}(\phi), \tag{207}$$

if $\delta_{M[f_1 \mapsto g]}(\phi)$ is defined for each g.

References

[1] Morgan Thomas, Orbis Labs. *Arithmetization of Σ_1^1 relations in Halo 2.* Cryptology ePrint Archive, Report 2022/777. `https://eprint.iacr.org/2022/777`

[2] Casper Assocation, Orbis Labs. *Open Specification Language.* 2022–2023. `https://github.com/Polytopoi/osl`

[3] Morgan Thomas, Orbis Labs. *Orbis Specification Language: a type theory for zk-SNARK programming.* Cryptology ePrint Archive, Report 2022/1003. `https://eprint.iacr.org/2022/1003`

[4] Anthony Hart, Morgan Thomas, Orbis Labs. *Arithmetization of Σ_1^1 relations with polynomial bounds in Halo 2.* Cryptology ePrint Archive, Report 2022/1003. `https://eprint.iacr.org/2022/1005`

[5] The Electric Coin Company. *The halo2 Book.* 2021. `https://zcash.github.io/halo2/index.html`

[6] The Electric Coin Company. *halo2.* 2022. `https://github.com/zcash/halo2`

[7] Samuel Buss. *Bounded Arithmetic.* 1985. Doctoral dissertation, Princeton University. `https://mathweb.ucsd.edu/~sbuss/ResearchWeb/BAthesis/Buss_Thesis_OCR.pdf`

[8] Edward Z. Yang. *Logitext Tutorial.* Accessed May 2022. `http://logitext.mit.edu/logitext.fcgi/tutorial`