Chiron: A Set Theory with Types, Undefinedness, Quotation, and Evaluation*

William M. Farmer†
McMaster University

15 March 2009

Abstract

Chiron is a derivative of von-Neumann-Bernays-Gödel (NBG) set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. Unlike traditional set theories such as Zermelo-Fraenkel (ZF) and NBG, Chiron is equipped with a type system, lambda notation, and definite and indefinite description. The type system includes a universal type, dependent types, dependent function types, subtypes, and possibly empty types. Unlike traditional logics such as first-order logic and simple type theory, Chiron admits undefined terms that result, for example, from a function applied to an argument outside its domain or from an improper definite or indefinite description. The most noteworthy part of Chiron is its facility for reasoning about the syntax of expressions. *Quotation* is used to refer to a set called a construction that represents the syntactic structure of an expression, and *evaluation* is used to refer to the value of the expression that a construction represents. Using quotation and evaluation, syntactic side conditions, schemas, syntactic transformations used in deduction and computation rules, and other such things can be directly expressed in Chiron. This paper presents the syntax and semantics of Chiron and illustrates its use with some simple examples.

*Published as SQRL Report No. 38, McMaster University, 2007 (revised 2009).

†Address: Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario L8S 4K1, Canada. E-mail: wmfarmer@mcmaster.ca.

Contents

1 Introduction **4**

2 Overview **5**
- 2.1 NBG Set Theory . . . 5
- 2.2 Values . . . 6
- 2.3 Expressions . . . 7
- 2.4 Dependent Function Types . . . 8
- 2.5 Undefinedness . . . 8
- 2.6 Quotation and Evaluation . . . 9

3 Syntax **9**
- 3.1 Expressions . . . 10
- 3.2 Compact Notation . . . 14
- 3.3 Quasiquotation . . . 16

4 Semantics **19**
- 4.1 The Liar Paradox . . . 19
- 4.2 Prestructures . . . 20
- 4.3 Structures . . . 22
- 4.4 Valuations . . . 24
- 4.5 Models . . . 27
- 4.6 Theories . . . 28
- 4.7 Notes concerning the Semantics of Chiron . . . 28

5 Relationship to NBG **31**

6 Operator Definitions **33**
- 6.1 Logical Operators . . . 33
- 6.2 Set-Theoretic Operators . . . 36
- 6.3 Syntactic Operators . . . 43
- 6.4 Substitution Operators . . . 47
- 6.5 Kernel Theory . . . 67

7 Examples **67**
- 7.1 Law of Excluded Middle . . . 67
- 7.2 Modus Ponens . . . 67
- 7.3 Beta Reduction . . . 68
- 7.4 Liar Paradox . . . 69

8 Conclusion **69**

Acknowledgments **70**

A Appendix: Alternate Semantics **70**
A.1 Valuations 70
A.2 Models 75
A.3 Discussion 76

References **76**

List of Tables

1 The Key Words of Chiron. 10
2 The Built-In Operators of Chiron. 14
3 Compact Notation 15
4 Additional Compact Notation 16

1 Introduction

The usefulness of a logic is often measured by its expressivity: the more that can be expressed in the logic, the more useful the logic is. By a *logic*, we mean a language (or a family of languages) that has a formal syntax and a precise semantics with a notion of logical consequence. (A logic may also have, but is not required to have, a proof system.) By this definition, a theory in a logic—such as Zermelo-Fraenkel (ZF) set theory in first-order order—is itself a logic. But what do we mean by *expressivity*? There are actually two notions of expressivity. The *theoretical expressivity* of a logic is the measure of what ideas can be expressed in the logic without regard to how the ideas are expressed. The *practical expressivity* of a logic is the measure of how readily ideas can be expressed in the logic.

To illustrate the difference between these two notions, let us compare two logics, standard first-order logic (FOL) and first-order logic without function symbols (FOL^-). Since functions can be represented using either predicate symbols or function symbols, FOL and FOL^- clearly have exactly the same theoretical expressivity. For example, if three functions are represented as unary function symbols f, g, h in FOL, these functions can be represented as binary predicate symbols p_f, p_g, p_h in FOL^-. The statement that the third function is the composition of the first two functions is expressed in FOL by the formula

$$\forall\, x \,.\, h(x) = f(g(x)),$$

while it is expressed in FOL^- by the more verbose formula

$$\forall\, x, z \,.\, p_h(x, z) \equiv \exists\, y \,.\, p_g(x, y) \wedge p_f(y, z).$$

The verbosity that comes from using predicate symbols to represent functions progressively increases as the complexity of statements about functions increases. Hence, FOL^- has a significantly lower level of practical expressivity than FOL does.

Traditional general-purpose logics—such as predicate logics like first-order logic and simple type theory and set theories like ZF and von-Neumann-Bernays-Gödel (NBG) set theory—are primarily intended to be theoretical tools. They are designed to be used *in theory*, not *in practice*. They are thus very expressive theoretically, but not very expressive practically. For example, in the languages of ZF and NBG, there is no vocabulary for forming a term $f(a)$ that denotes the application of a set f representing a function to a set a representing an argument to f. Moreover, even if such an application operator were added to ZF or NBG, there is no special mechanism for handling

"undefined" applications. As a result, statements involving functions and undefinedness are much more verbose and indirect than they need to be, and reasoning about functions and undefinedness is usually performed in the metalogic instead of in the logic itself.

Chiron is a set theory that has a much higher level of practical expressivity than traditional set theories. It is intended to be a general-purpose logic that, unlike traditional logics, is designed to be used in practice. It integrates NBG set theory, elements of type theory, a scheme for handling undefinedness, and a facility for reasoning about the syntax of expressions. This paper presents the syntax and semantics of Chiron and illustrates its use with some simple examples. A quicker, more informal presentation of the syntax and semantics of Chiron is found in [6].

The following is the outline of the paper. Section 2 gives an informal overview of Chiron. Section 3 presents Chiron's official syntax and an unofficial compact notation for Chiron. The semantics of Chiron is given in section 4. In section 5, we show that there is a faithful interpretation of NBG in Chiron. A large group of useful operators are defined in section 6 including the operators needed for the substitution of a term for the occurrences of a free variable. Some of the practical expressivity of Chiron is illustrated in section 7. The paper concludes in section 8 with a brief summary and a list of future tasks. An appendix presents two alternate semantics for Chiron based on value gaps.

2 Overview

This section gives an informal overview of Chiron. A formal definition of the syntax and semantics of Chiron is presented in subsequent sections.

2.1 NBG Set Theory

NBG set theory is closely related to the more well-known ZF set theory. The underlying logic of both NBG and ZF is first-order logic, and NBG and ZF both share the same intuitive model of the iterated hierarchy of sets. However, in contrast to ZF, variables in NBG range over both sets and proper classes. Thus, the universe of sets V and total functions from V to V like the cardinality function can be represented as terms in NBG even though they are proper classes. There is a faithful interpretation of ZF in NBG [13, 16, 17]. This means that ZF can be embedded in NBG in a meaning preserving way and that ZF is consistent iff NBG is consistent. (A good introduction to NBG is found in [9] or [12].)

Chiron is a derivative of NBG. It is an enhanced version of STMM [3], a version of NBG with types and undefinedness. Chiron has a much richer syntax and more complex semantics than NBG, but the models for Chiron contain exactly the same values (i.e., classes) as the models for NBG. Moreover, there is a faithful interpretation of NBG in Chiron—which means that there is a meaning preserving embedding of NBG in Chiron such that Chiron is a conservative extension of the image of NBG under the embedding. That is, Chiron adds new reasoning machinery to NBG without compromising the underlying semantics of NBG.

2.2 Values

A *value* is a set, class, superclass, truth value, undefined value, or operation. A *class* is an element of a model of NBG set theory. Each class is a collection of classes. A *set* is a class that is a member of a class. A class is thus a collection of sets. A class is *proper* if it not a set. A *superclass* is a collection of classes that need not be a class itself. Summarizing, the domain D_{v} of sets is a proper subdomain of the domain D_{c} of classes, and D_{c} is a proper subdomain of the domain D_{s} of superclasses. D_{v} is the universal class (the class of all sets), and D_{c} is the universal superclass (the superclass of all classes).

There are two truth values, T representing *true* and F representing *false*. The truth values are not members of D_{s}. There is also an *undefined value* $\bot$ which serves as the value of various sorts of undefined terms such as undefined function applications and improper definite or indefinite descriptions. $\bot$ is not a member of $D_s \cup \{\mathrm{T}, \mathrm{F}\}$.

An *operation* is a mapping over superclasses, the truth values, and the undefined value. More precisely, for $n \geq 0$, an *n-ary operation* is a total mapping

$$o : D_1 \times \cdots \times D_n \to D_{n+1}$$

where D_i is D_{s}, $D_{\mathrm{c}} \cup \{\bot\}$, or $\{\mathrm{T}, \mathrm{F}\}$ for all i with $1 \leq i \leq n+1$. An operation is not a member of $D_s \cup \{\mathrm{T}, \mathrm{F}, \bot\}$. A *function* is a class of ordered pairs that represents a (possibly partial) mapping

$$f : D_{\mathrm{v}} \rightharpoonup D_{\mathrm{v}}.$$

Operations are not classes, but many operations can be represented by functions (which are classes).

2.3 Expressions

An *expression* is a tree whose leaves are *symbols*. There are four special sorts of expressions: *operators*, *types*, *terms*, and *formulas*. An expression is *proper* is it is one of these special sorts of expressions, and an expression is *improper* if it is not proper. Proper expressions denote values, while improper expressions are nondenoting (i.e., they do not denote anything).

Operators denote operations. Many sorts of syntactic entities can be formalized in Chiron as operators. Examples include logical connectives; individual constants, function symbols, and predicate symbols from first-order logic; base types and type constructors including dependent type constructors; and definedness operators. Like a function or predicate symbol in first-order logic, an operator in Chiron is not meaningful unless it is applied.

Types are used to restrict the values of operators and variables and to classify terms by their values. They denote superclasses. Terms are used to describe classes. They denote classes or the undefined value $\bot$. A term is *defined* if it denotes a class and is *undefined* if it denotes $\bot$. Every term is assigned a type. Suppose a term a is assigned a type α and α denotes a superclass Σ_α. If a is defined, i.e., a denotes a class x, then x is in Σ_α. Formulas are used to make assertions. They denote truth values.

The proper expressions are categorized according to their first (leftmost) symbols:

1. Operator and operator applications (op, op-app).
2. Variables (var).
3. Type applications and dependent function types (type-app, dep-fun-type).
4. Function applications and abstractions (fun-app, fun-abs).
5. Conditional terms (if).
6. Existential quantifications (exist).
7. Definite and indefinite descriptions (def-des, indef-des).
8. Quotations and evaluations (quote, eval).

2.4 Dependent Function Types

A *dependent function type* is a type of the form

$$\gamma = (\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$$

where α and β are types. (Dependent function types are commonly known as *dependent product types*.) The type γ denotes a superclass of possibly partial functions. A function abstraction of the form

$$(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b),$$

where b is a term of type β, is of type γ.

The dependent function type γ is a generalization of the more common function type $\alpha \to \beta$. If f is a term of type $\alpha \to \beta$ and a is a term of type α, then the application $f(a)$ is of type β—which does not depend on the value of a. In Chiron, however, if f is a term of type γ and a is a term of type α, then the term

$$(\mathsf{fun\text{-}app}, f, a),$$

the application of f to a, is of the type

$$(\mathsf{type\text{-}app}, \gamma, a),$$

the type formed by applying the type γ to a—which generally depends on the value of a.

2.5 Undefinedness

An expression is *undefined* if it has no prescribed meaning or if it denotes a value that does not exist. There are several sources of undefined expressions in Chiron:

- Nondenoting operator, type, and function applications.
- Nonexistent function abstractions.
- Improper definite and indefinite descriptions.
- Out of range variables and evaluations.

Undefined expressions are handled in Chiron according to the *traditional approach to undefinedness* [4]. The value of an undefined term is the undefined value $\bot$, but the value of an undefined type or formula is D_{c} (the universal superclass) or F, respectively. That is, the values for undefined types, terms, and formulas are D_{c}, $\bot$, and F, respectively. Commonly used in mathematical practice, the traditional approach to undefinedness enables statements involving partial functions and definite and indefinite descriptions to be expressed very concisely [4].

2.6 Quotation and Evaluation

A *construction* is a set that represents the syntactic structure of an expression. A term of the form (quote, e), where e is an expression, denotes the construction that represents e. Thus a proper expression e has two different meanings:

1. The *semantic meaning* of e is the value denoted by e itself.
2. The *syntactic meaning* of e is the construction denoted by (quote, e).

There are two ways to refer to a semantic meaning v. The first is to directly form a proper expression e not beginning with eval that denotes v. The second is to form a term a that denotes the construction that represents a proper expression e that denotes v and then form the type $(\mathsf{eval}, a, \mathsf{type})$, term $(\mathsf{eval}, a, \alpha)$, or formula $(\mathsf{eval}, a, \mathsf{formula})$ (depending on whether e is a type, a term assigned the type α, or a formula) which denotes v.

Likewise there are two ways to refer to a syntactic meaning c. The first is to directly form a term a not beginning with quote that denotes c. The second is to form an expression e such that the construction c represents the syntactic structure of e and then form the expression (quote, e) which denotes c.

For an expression e, the term (quote, e) denotes the syntactic meaning of e and is thus always defined (even when e is an undefined term or an improper (nondenoting) expression). However, a term $(\mathsf{eval}, a, \alpha)$, where α is a type, may be undefined.

3 Syntax

This section presents the syntax of Chiron which is inspired by the S-expression syntax of the Lisp family of programming languages.

op	type	formula	op-app	var
type-app	dep-fun-type	fun-app	fun-abs	if
exist	def-des	indef-des	quote	eval
set	class	expr	expr-sym	expr-op
expr-type	expr-term	expr-term-type	expr-formula	in
type-equal	term-equal	formula-equal	not	or

Table 1: The Key Words of Chiron.

3.1 Expressions

Let $\mathcal{S}$ be a fixed well-ordered, countably infinite set of symbols and $\mathcal{K}$ be the set of the 30 symbols called *key words* in Table 1. Assume $\mathcal{K} \subseteq \mathcal{S}$.

The two formation rules below inductively define the notion of an *expression* of Chiron. $\mathbf{expr}[e]$ asserts that e is an expression.

Expr-1 (Atomic expression)

$$\frac{s \in \mathcal{S}}{\mathbf{expr}[s]}$$

Expr-2 (Compound expression)

$$\frac{\mathbf{expr}[e_1], \ldots, \mathbf{expr}[e_n]}{\mathbf{expr}[(e_1, \ldots, e_n)]}$$

where $n \geq 0$.

Hence, an expression is an S-expression (with commas in place of spaces) that exhibits the structure of a tree whose leaves are symbols in $\mathcal{S}$. Let $\mathcal{E}$ denote the set of expressions of Chiron.

The *length* of an expression $e \in \mathcal{E}$, written $|e|$, is defined recursively by the following statements:

1. If $s \in \mathcal{S}$, $|s| = 1$.
2. If $e_1, \ldots, e_n \in \mathcal{E}$, $|(e_1, \ldots, e_n)| = 1 + |e_1| + \cdots |e_n|$.

Notice that $|(\,)| = 1$ and, in general, $|e|$ equals the number of symbols and parenthesis pairs occurring in e. The *complexity* of an expression $e \in \mathcal{E}$, written $c(e)$, is the pair $(m, n) \in \mathbf{N} \times \mathbf{N}$ of natural numbers such that:

1. m is the number of occurrences of the symbol eval in e.

2. n is the length of e.

For $c(e_1) = (m_1, n_1)$ and $c(e_2) = (m_2, n_2)$, $c(e_1) < c(e_2)$ means either $m_1 < m_2$ or ($m_1 = m_2$ and $n_1 < n_2$).

The set of 13 formation rules below defines the notion of a *proper expression* of Chiron. A proper expression denotes a class, a truth value, the undefined value, or an operation. Each proper expression is assigned an expression. $\mathbf{p\text{-}expr}[e : e']$ asserts that $e \in \mathcal{E}$ is a proper expression to which the expression $e' \in \mathcal{E}$ is assigned. An *improper expression* is an expression that is not a proper expression. Improper expressions are nondenoting.

There are four sorts of proper expressions. An *operator* is a proper expression to which the expression op is assigned. A *type* is a proper expression to which the expression type is assigned. A *term* is a proper expression to which a type is assigned. And a *formula* is a proper expression to which the expression $\mathsf{formula}$ is assigned. When a is a term, α is a type, and $\mathbf{p\text{-}expr}[a : \alpha]$ holds, a is said to be a *term of type* α. As we mentioned earlier, operators denote operations, types denote superclasses, terms denote classes or the undefined value $\bot$, and formulas denote the truth values T and F.

$\mathbf{operator}[o]$ means $\mathbf{p\text{-}expr}[o : \mathsf{op}]$, $\mathbf{type}[\alpha]$ means $\mathbf{p\text{-}expr}[\alpha : \mathsf{type}]$, $\mathbf{term}[a]$ means $\mathbf{p\text{-}expr}[a : \alpha]$ for some type α, and $\mathbf{formula}[A]$ means $\mathbf{p\text{-}expr}[A : \mathsf{formula}]$. $\mathbf{term}[a : \alpha]$ means $\mathbf{p\text{-}expr}[a : \alpha]$ and $\mathbf{type}[\alpha]$, i.e., a is a term of type α. An expression k is a *kind*, written $\mathbf{kind}[k]$, if $k = \mathsf{type}$, $\mathbf{type}[k]$, or $k = \mathsf{formula}$. Thus kinds are the expressions assigned to types, terms, and formulas. A proper expression e is said to be an *expression of kind* k if $k = \mathsf{type}$ and e is a type, $\mathbf{type}[k]$ and e is a term of type k, or $k = \mathsf{formula}$ and e is a formula.

The following formation rules define the 13 proper expression categories of Chiron:

P-Expr-1 (Operator)

$$\frac{s \in \mathcal{S}, \mathbf{kind}[k_1], \ldots, \mathbf{kind}[k_{n+1}]}{\mathbf{operator}[(\mathsf{op}, s, k_1, \ldots, k_{n+1})]}$$

where $n \geq 0$.

P-Expr-2 (Operator application)

$$\frac{\mathbf{operator}[(\mathsf{op}, s, k_1, \ldots, k_{n+1})], \mathbf{expr}[e_1], \ldots, \mathbf{expr}[e_n]}{\mathbf{p\text{-}expr}[(\mathsf{op\text{-}app}, (\mathsf{op}, s, k_1, \ldots, k_{n+1}), e_1, \ldots, e_n) : k_{n+1}]}$$

where $n \geq 0$ and ($k_i = \mathsf{type}$ and $\mathbf{type}[e_i]$), ($\mathbf{type}[k_i]$ and $\mathbf{term}[e_i]$), or ($k_i = \mathsf{formula}$ and $\mathbf{formula}[e_i]$) for all i with $1 \leq i \leq n$.

P-Expr-3 (Variable)

$$\frac{x \in \mathcal{S}, \mathbf{type}[\alpha]}{\mathbf{term}[(\mathsf{var}, x, \alpha) : \alpha]}$$

P-Expr-4 (Type application)

$$\frac{\mathbf{type}[\alpha], \mathbf{term}[a]}{\mathbf{type}[(\mathsf{type\text{-}app}, \alpha, a)]}$$

P-Expr-5 (Dependent function type)

$$\frac{\mathbf{term}[(\mathsf{var}, x, \alpha)], \mathbf{type}[\beta]}{\mathbf{type}[(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)]}$$

P-Expr-6 (Function application)

$$\frac{\mathbf{term}[f : \alpha], \mathbf{term}[a]}{\mathbf{term}[(\mathsf{fun\text{-}app}, f, a) : (\mathsf{type\text{-}app}, \alpha, a)]}$$

P-Expr-7 (Function abstraction)

$$\frac{\mathbf{term}[(\mathsf{var}, x, \alpha)], \mathbf{term}[b : \beta]}{\mathbf{term}[(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b) : (\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)]}$$

P-Expr-8 (Conditional term)

$$\frac{\mathbf{formula}[A], \mathbf{term}[b : \beta], \mathbf{term}[c : \gamma]}{\mathbf{term}[(\mathsf{if}, A, b, c) : \delta]}$$

where $\delta = \begin{cases} \beta & \text{if } \beta = \gamma \\ (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{class}, \mathsf{type})) & \text{otherwise} \end{cases}$

P-Expr-9 (Existential quantification)

$$\frac{\mathbf{term}[(\mathsf{var}, x, \alpha)], \mathbf{formula}[B]}{\mathbf{formula}[(\mathsf{exist}, (\mathsf{var}, x, \alpha), B)]}$$

P-Expr-10 (Definite description)

$$\frac{\mathbf{term}[(\mathsf{var}, x, \alpha)], \mathbf{formula}[B]}{\mathbf{term}[(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B) : \alpha]}$$

P-Expr-11 (Indefinite description)

$$\frac{\mathbf{term}[(\mathsf{var}, x, \alpha)], \mathbf{formula}[B]}{\mathbf{term}[(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B) : \alpha]}$$

P-Expr-12 (Quotation)

$$\frac{\mathbf{expr}[e]}{\mathbf{term}[(\mathsf{quote}, e) : (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{expr}, \mathsf{type}))]}$$

P-Expr-13 (Evaluation)

$$\frac{\mathbf{term}[a], \mathbf{kind}[k]}{\mathbf{p\text{-}expr}[(\mathsf{eval}, a, k) : k]}$$

We will use $s, t, u, v, w, x, y, z, \ldots$ to denote symbols; $O, P, Q, \ldots$ to denote operators; $\alpha, \beta, \gamma, \ldots$ to denote types; $a, b, c, \ldots$ to denote terms; $A, B, C, \ldots$ to denote formulas; and $k, k', \ldots$ to denote kinds.

Proposition 3.1 *The formation rules assign a unique expression to each proper expression.*

Let $O = (\mathsf{op}, s, k_1, \ldots, k_{n+1})$ be an operator. The symbol s is called the *symbol* of O, and the list $k_1, \ldots, k_{n+1}$ of kinds is called the *signature* of O. O is a *type operator*, *term operator*, or *formula operator* if $k_{n+1} = \mathsf{type}$, $\mathbf{type}[k_{n+1}]$, or $k_{n+1} = \mathsf{formula}$, respectively. A *base type* is a a type operator application of the form $(\mathsf{op\text{-}app}, (\mathsf{op}, s, \mathsf{type}))$. An *individual constant* of type α is a term operator application of the form $(\mathsf{op\text{-}app}, (\mathsf{op}, s, \alpha))$.

Two operators $(\mathsf{op}, s^1, k_1^1, \ldots, k_{m+1}^1)$ and $(\mathsf{op}, s^2, k_1^2, \ldots, k_{n+1}^2)$ are *similar* if $s^1 = s^2$, $m = n$, and either $k_i^1 = k_i^2$ or both $\mathbf{type}[k_i^1]$ and $\mathbf{type}[k_i^2]$ for all i with $1 \leq i \leq m+1$. Two variables $(\mathsf{var}, x, \alpha)$ and (var, y, β) are *similar* if $x = y$. An expression e is *eval-free* if it does not contain the symbol eval.

A *subexpression* of an expression is defined inductively as follows:

1. If e is a proper expression, then e is a subexpression of itself.

2. If $e = (s, e_1, \ldots, e_n)$ is a proper expression such that s is not quote, then e_i is a subexpression of e for each proper expression e_i with $1 \leq i \leq n$.

3. If e is a subexpression of e' and e' is a subexpression of e'', then e is a subexpression of e''.

1.	(op, set, type)
2.	(op, class, type)
3.	(op, expr, type)
4.	(op, expr-sym, type)
5.	(op, expr-op, type)
6.	(op, expr-type, type)
7.	(op, expr-term, type)
8.	(op, expr-term-type, (op-app, (op, expr-type, type)), type)
9.	(op, expr-formula, type)
10.	(op, in, (op-app, (op, set, type)), (op-app, (op, class, type)), formula)
11.	(op, type-equal, type, type, formula)
12.	(op, term-equal, (op-app, (op, class, type)), (op-app, (op, class, type)), type, formula)
13.	(op, formula-equal, formula, formula, formula)
14.	(op, not, formula, formula)
15.	(op, or, formula, forumla, formula)

Table 2: The Built-In Operators of Chiron.

e is a *proper subexpression* of e' if e is a subexpression of e' and $e \neq e'$. Notice that (1) a subexpression is always a proper expression, (2) an improper expression has no subexpressions, (3) a quotation has no proper subexpressions, and (4) if e_1 is a proper subexpression of e_2, then $c(e_1) < c(e_2)$, i.e., the complexity of a proper subexpression of an expression is strictly less than the expression itself.

A *language* of Chiron is a set of operators that contains the 15 "built-in" operators in Table 2. In the remainder of this paper, let L be a language of Chiron.

3.2 Compact Notation

We introduce in this subsection a compact notation for proper expressions—which we will use in the rest of the paper whenever it is convenient. The first group of definitions in Table 3 defines the compact notation for each of the 13 proper expression categories.

The next group of definitions in Table 4 defines additional compact notation for the built-in operators and the universal quantifier.

Compact Notation	**Official Notation**
$(s :: k_1, \ldots, k_{n+1})$	$(\mathsf{op}, s, k_1, \ldots, k_{n+1})$
$(s :: k_1, \ldots, k_{n+1})(e_1, \ldots, e_n)$	$(\mathsf{op\text{-}app}, (\mathsf{op}, s, k_1, \ldots, k_{n+1}), e_1, \ldots, e_n)$
$(x : \alpha)$	$(\mathsf{var}, x, \alpha)$
$\alpha(a)$	$(\mathsf{type\text{-}app}, \alpha, a)$
$(\Lambda\, x : \alpha\ .\ \beta)$	$(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$
$f(a)$	$(\mathsf{fun\text{-}app}, f, a)$
$(\lambda\, x : \alpha\ .\ b)$	$(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$
$\mathsf{if}(A, b, c)$	(if, A, b, c)
$(\exists\, x : \alpha\ .\ B)$	$(\mathsf{exist}, (\mathsf{var}, x, \alpha), B)$
$(\iota\, x : \alpha\ .\ B)$	$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$
$(\epsilon\, x : \alpha\ .\ B)$	$(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$
$\lceil e \rceil$	(quote, e)
$[\![a]\!]_k$	(eval, a, k)
$[\![a]\!]_{\mathrm{ty}}$	$(\mathsf{eval}, a, \mathsf{type})$
$[\![a]\!]_{\mathrm{te}}$	$(\mathsf{eval}, a, (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{class}, \mathsf{type})))$
$[\![a]\!]_{\mathrm{fo}}$	$(\mathsf{eval}, a, \mathsf{formula})$

Table 3: Compact Notation

We will often employ the following abbreviation rules when using the compact notation:

1. A matching pair of parentheses in an expression may be dropped if there is no resulting ambiguity.

2. A variable $(x : \alpha)$ occurring in the body e of $(\star\, x : \alpha\ .\ e)$, where $\star$ is Λ, λ, $\exists$, $\forall$, ι, or ϵ may be written as x if there is no resulting ambiguity.

3. $(\star\, x_1 : \alpha_1 \ldots (\star\, x_n : \alpha_n\ .\ e) \cdots)$, where $\star$ is Λ, λ, $\exists$, or $\forall$, may be written as

   $$(\star\, x_1 : \alpha_1, \ldots, x_n : \alpha_n\ .\ e).$$

   Similarly, $(\star\, x_1 : \alpha \ldots (\star\, x_n : \alpha\ .\ e) \cdots)$, where $\star$ is Λ, λ, $\exists$, or $\forall$, may be written as

   $$(\star\, x_1, \ldots, x_n : \alpha\ .\ e).$$

4. If we fix the type of a variable symbol x, say to α, then an expression of the form $(\star\, x : \alpha\ .\ e)$, where $\star$ is Λ, λ, $\exists$, $\forall$, ι, or ϵ, may be written as $(\star\, x\ .\ e)$.

Compact Notation	**Defining Expression**
V	$(\mathsf{set} :: \mathsf{type})(\,)$
C	$(\mathsf{class} :: \mathsf{type})(\,)$
E	$(\mathsf{expr} :: \mathsf{type})(\,)$
E_{sy}	$(\mathsf{expr\text{-}sym} :: \mathsf{type})(\,)$
E_{op}	$(\mathsf{expr\text{-}op} :: \mathsf{type})(\,)$
E_{ty}	$(\mathsf{expr\text{-}type} :: \mathsf{type})(\,)$
E_{te}	$(\mathsf{expr\text{-}term} :: \mathsf{type})(\,)$
E_a	$(\mathsf{expr\text{-}term\text{-}type} :: \mathsf{E}_{\mathrm{ty}}, \mathsf{type})(a)$
E_{fo}	$(\mathsf{expr\text{-}formula} :: \mathsf{type})(\,)$
$(a \in b)$	$(\mathsf{in} :: \mathsf{V}, \mathsf{C}, \mathsf{formula})(a, b)$
$(\alpha =_{\mathrm{ty}} \beta)$	$(\mathsf{type\text{-}equal} :: \mathsf{type}, \mathsf{type}, \mathsf{formula})(\alpha, \beta)$
$(a =_\alpha b)$	$(\mathsf{term\text{-}equal} :: \mathsf{C}, \mathsf{C}, \mathsf{type}, \mathsf{formula})(a, b, \alpha)$
$(a = b)$	$(a =_{\mathsf{C}} b)$
$(A \equiv B)$	$(\mathsf{formula\text{-}equal} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})(A, B)$
$(\neg A)$	$(\mathsf{not} :: \mathsf{formula}, \mathsf{formula})(A)$
$(a \notin b)$	$(\neg(a \in b))$
$(a \neq b)$	$(\neg(a = b))$
$(A \vee B)$	$(\mathsf{or} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})(A, B)$
$(\forall\, x : \alpha \,.\, A)$	$(\neg(\exists\, x : \alpha \,.\, (\neg A)))$

Table 4: Additional Compact Notation

5. If we fix the signature of an operator symbol s, say to $k_1, \ldots, k_{n+1}$, then an operator application of the form $(s :: k_1, \ldots, k_{n+1})(e_1, \ldots, e_n)$ may be written as $s(e_1, \ldots, e_n)$ and an operator application of the form $(s :: k)(\,)$ may be written as s.

6. $[\![a]\!]_{\mathrm{ty}}$, $[\![a]\!]_{\mathrm{te}}$, and $[\![a]\!]_{\mathrm{fo}}$ may be shortened to $[\![a]\!]$ if a is of type E_{ty}, E_{te}, and E_{fo}, respectively.

Using the compact notation, expressions can be written in Chiron so that they look very much like expressions written in mathematics textbooks and papers.

3.3 Quasiquotation

Quasiquotation is a parameterized form of quotation in which the parameters serve as holes in a quotation that are filled with the values of expressions.

It is a very powerful syntactic device for specifying expressions and defining macros. Quasiquotation was introduced by W. Quine in 1940 in the first version of his book *Mathematical Logic* [15]. It has been extensively employed in the Lisp family of programming languages [1].[1]

We will introduce quasiquotation into Chiron as a notational definition. Unlike quotation, quasiquotation will not be part of the official Chiron syntax. The meaning of a quasiquotation will be an expression that denotes a construction.

The three formation rules below inductively define the notion of a *marked expression* of Chiron. **m-expr**$[m]$ asserts that m is a marked expression.

M-Expr-1

$$\frac{s \in \mathcal{S}}{\textbf{m-expr}[s]}$$

M-Expr-2

$$\frac{\textbf{term}[a]}{\textbf{m-expr}[\lfloor a \rfloor]}$$

M-Expr-3

$$\frac{\textbf{m-expr}[m_1], \ldots, \textbf{m-expr}[m_n]}{\textbf{m-expr}[(m_1, \ldots, m_n)]}$$

where $n \geq 0$.

A marked expression of the form $\lfloor a \rfloor$ is called an *evaluated component*.

Proposition 3.2 *Every expression is a marked expression.*

A *quasiquotation* of Chiron is a syntactic entity of the form

$(\mathsf{quasiquote}, m)$

where m is a marked expression.[2] A compact notation for quasiquotation can be easily defined as indicated by the following example: Let $\lceil f(\lfloor a \rfloor) \rceil$ be the compact notation for the quasiquotation

$(\mathsf{quasiquote}, (\mathsf{fun\text{-}app}, f, \lfloor a \rfloor))$.

[1] In Lisp, the standard symbol for quasiquotation is the backquote (`) symbol, and thus in Lisp, quasiquotation is usually called *backquote*.

[2] Quasiquotations correspond to backquotes in Lisp as follows. The symbol quasiquote corresponds to the backquote symbol (`) and a marked expression $\lfloor a \rfloor$ in a quasiquotation corresponds to $,a$. Thus the quasiquotation $(\mathsf{quasiquote}, (a, b, \lfloor c \rfloor))$ corresponds to the backquote `` `(a b ,c) ``.

Here we are using $\lceil m \rceil$ when m is an expression to mean the quotation of m and when m is a marked expression containing evaluated components to mean the quasiquotation of m.

We next define the semantics of quasiquotation. It assumes a knowledge of the semantics of Chiron given in section 4 and the defined-in and ord-pair operators defined in section 6. Let F be the function, mapping marked expressions to terms, recursively defined by:

1. If m is a symbol $s \in \mathcal{S}$, then $F(m) = (\mathsf{quote}, s)$.
2. If m is an evaluated component $\lfloor a \rfloor$, then $F(m) = a$.
3. If m is a marked expression $(m_1, \ldots, m_n)$ where $n \geq 0$, then

$$F(m) = [F(m_1), \ldots, F(m_n)].$$

Note that, as defined in section 6.2, $[a_1, \ldots, a_n]$ denotes an n-tuple of sets. For a quasiquotation $q = (\mathsf{quasiquote}, m)$, define $G(q) = F(m)$.

Proposition 3.3

1. *If e is an expression, then $G((\mathsf{quasiquote}, e))$ is a term such that*

$$\models G((\mathsf{quasiquote}, e)) = (\mathsf{quote}, e).$$

2. *If $\{\lfloor a_1 \rfloor, \ldots, \lfloor a_n \rfloor\}$ is the set of evaluated components occurring in a quasiquotation q, then*

$$\models (a_1 \downarrow \mathsf{E} \wedge \cdots \wedge a_n \downarrow \mathsf{E}) \supset G(q) \downarrow \mathsf{E}.$$

Note that, as defined in section 6.1, $a \downarrow \alpha$ asserts that the value of the term a is in the value of type α.

From now on, we will consider a quasiquotation q to be an alternate notation for the term $G(q)$.

Many expressions involving syntax can be expressed very succinctly using quasiquotation. For example, the function abstraction given in section 7.4 can be expressed as

$$\lambda\, e : \mathsf{E}\ .\ \lceil \neg [\![\mathsf{enum}(\lfloor e \rfloor)(\lfloor e \rfloor)]\!]_{\mathrm{fo}} \rceil.$$

Quasiquotation also provides an alternate style of expressing schemas in Chiron. For example, LEM in section 7.1 can be express as

$$\forall\, e : \mathsf{E}_{\mathrm{fo}}\ .\ [\![\lceil \lfloor e \rfloor \vee \neg \lfloor e \rfloor \rceil]\!]_{\mathrm{fo}}$$

as opposed to

$$\forall\, e : \mathsf{E}_{\mathrm{fo}} \,.\, [\![e]\!] \vee \neg [\![e]\!]$$

as shown in section 7.1.

4 Semantics

This section presents the official semantics of Chiron which is based on *standard models*. Two alternate semantics based on other kinds of models are given in the Appendix.

4.1 The Liar Paradox

Using quotation and evaluation, it is possible to express the *liar paradox* in Chiron. That is, it is possible to construct a term LIAR whose value equals the value of

$$\ulcorner \neg [\![\mathsf{LIAR}]\!]_{\mathrm{fo}} \urcorner.$$

(See section 7.4 for details.) LIAR denotes a construction representing a formula that says in effect "I am a formula that is false".

If the naive semantics is employed for quotation and evaluation, a contradiction is immediately obtained:

$$\begin{aligned} [\![\mathsf{LIAR}]\!]_{\mathrm{fo}} &= [\![\ulcorner \neg [\![\mathsf{LIAR}]\!]_{\mathrm{fo}} \urcorner]\!]_{\mathrm{fo}} \\ &= \neg [\![\mathsf{LIAR}]\!]_{\mathrm{fo}} \end{aligned}$$

$[\![\mathsf{LIAR}]\!]_{\mathrm{fo}}$ is thus *ungrounded* in the sense that its value depends on itself. This simple argument is essentially the proof of Tarski's 1935 theorem on the undefinability of truth [18, 19]. The theorem says that $[\![x]\!]_{\mathrm{fo}}$ cannot serve as a truth predicate over all formulas.

Any reasonable semantics for Chiron needs a way to block the liar paradox and similar ungrounded expressions. We will briefly describe three approaches.

The first approach is to remove evaluation (eval) from Chiron but keep quotation (quote). This would eliminate ungrounded expressions. However, it would also severely limit Chiron's facility for reasoning about the syntax of expressions. It would be possible using quotation to construct terms that denote constructions, but without evaluation it would not be possible to employ these terms as the expressions represented by the constructions. Some of the power of evaluation could be replaced by introducing functions

that map types of constructions to types of values. An example would be a function that maps numerals to natural numbers.

A second approach is to define a semantics with "value gaps" so that expressions like $[\![\mathsf{LIAR}]\!]_{\mathrm{fo}}$ are not assigned any value at all. In his famous paper *Outline of a Theory of Truth* [11], S. Kripke presents a framework for defining semantics with *truth-value gaps* using various evaluation schemes. Kripke's approach can be easily generalized to allow *value gaps* for types and terms as well as for formulas. In the Appendix we define two value-gap semantics for Chiron using Kripke's framework with valuation schemes based on the weak Kleene logic [10] and B. van Fraassen's notion of a *super-valuation* [21]. Kripke-style value-gap semantics are very interesting, if not enlightening, but they are exceeding difficult to work with in practice. The main problem is that there is no mechanism to assert that an expression has no value (see the Appendix for details).

The third approach is to consider any evaluation of a term that denotes a construction representing a non-eval-free expression to be undefined. For example, the value of $[\![\mathsf{LIAR}]\!]_{\mathrm{fo}}$ would be F, the undefined value for formulas. It is important to understand that an evaluation of a term a containing the symbol eval can be defined as long as a denotes a construction representing an expression that does not contain eval. Thus the value of an evaluation like $[\![[\![\ulcorner\ulcorner 17\urcorner\urcorner]\!]_{\mathrm{te}}]\!]_{\mathrm{te}}$ would be the value of 17 because the expression 17 presumably does not contain eval. The official semantics for Chiron defined in this section employs this third approach to blocking the liar paradox.

4.2 Prestructures

A *prestructure* of Chiron is a pair $(D, \in)$, where D is a nonempty domain and $\in$ is a membership relation on D, that satisfies the axioms of NBG set theory as given, for example, in [9] or [12].

Let $P = (D^P_{\mathrm{c}}, \in^P)$ be a prestructure of Chiron. A *class* of P is a member of D^P_{c}. A *set* of P is a member x of D^P_{c} such that $x \in^P y$ for some member y of D^P_{c}. That is, a set is a class that is itself a member of a class. A class is *proper* if it is not a set. A *superclass* of P is a collection of classes in D^P_{c}. We consider a class, as a collection of sets, to be a superclass. Let D^P_{v} be the domain of sets of P and D^P_{s} be the domain of superclasses of P. The following inclusions hold: $D^P_{\mathrm{v}} \subset D^P_{\mathrm{c}} \subset D^P_{\mathrm{s}}$.

A *function* of P is a member f of D^P_{c} such that:

1. Each $p \in^P f$ is an ordered pair $\langle x, y\rangle$ where x, y are in D^P_{v}.
2. For all $\langle x_1, y_1\rangle, \langle x_2, y_2\rangle \in^P f$, if $x_1 = x_2$, then $y_1 = y_2$.

Notice that a function of P may be partial, i.e., there may not be an ordered pair $\langle x, y\rangle$ in a function for each x in D_{v}^P.

Let D_{f}^P be the domain of functions of P. For f, x in D_{c}^P, $f(x)$ denotes the unique y in D_{v}^P such that f is in D_{f}^P and $\langle x, y\rangle \in^P f$. ($f(x)$ is undefined if there is no such unique y in D_{v}^P.) For Σ in D_{s}^P and x in D_{c}^P, $\Sigma[x]$ denotes the class of all y in D_{v}^P such that, for some f in both Σ and D_{f}^P, $f(x) = y$.

Let T^P, F^P, and $\bot^P$ be distinct values not in D_{s}^P. T^P and F^P represent the truth values *true* and *false*, respectively. $\bot^P$ is the *undefined value* that represents values that are undefined.

For $n \geq 0$, an *n-ary operation* of P is a total mapping from $D_1 \times \cdots \times D_n$ to D_{n+1} where D_i is D_{s}^P, $D_{\mathrm{c}}^P \cup \{\bot^P\}$, or $\{\mathrm{T}^P, \mathrm{F}^P\}$ for each i with $1 \leq i \leq n+1$. Let D_{o}^P be the domain of operations of P. We assume that $D_{\mathrm{s}}^P \cup \{\mathrm{T}^P, \mathrm{F}^P, \bot^P\}$ and D_{o}^P are disjoint.

Let $D_{\mathrm{e},1}^P = \omega = \{0, 1, 2, \ldots\}$, the set of finite cardinals in D_{v}. (We assume that each $n \in \omega$ is not an ordered pair.) Then let $D_{\mathrm{e},2}^P$ be the subset of D_{v}^P defined inductively as follows:

1. The empty set $\emptyset$ is a member of $D_{\mathrm{e},2}^P$.
2. If $x \in D_{\mathrm{e},1}^P \cup D_{\mathrm{e},2}^P$ and $y \in D_{\mathrm{e},2}^P$, then the ordered pair $\langle x, y\rangle$ of x and y is a member of $D_{\mathrm{e},2}^P$.

Finally, let $D_{\mathrm{e}}^P = D_{\mathrm{e},1}^P \cup D_{\mathrm{e},2}^P$. The members of D_{e}^P are called *constructions*. They are sets with the form of trees whose leaves are finite cardinals. Their purpose is to represent the syntactic structure of expressions. The symbols in expressions are represented by the finite cardinals in constructions. A construction is a *symbol construction*, *operator construction*, *type construction*, *term construction*, or *formula construction* if it represents a symbol, operator, type, term, or formula, respectively.

Let H^P be the bijective mapping of $\mathcal{E}$ onto D_{e}^P defined recursively by:

1. If $e = s_n \in \mathcal{S}$ (the n-th member of $\mathcal{S}$) where $n \geq 0$, then $H^P(e) = n+1$, the $(n+1)$-th finite cardinal.
2. If $e = (\,) \in \mathcal{E}$, then $H^P(e) = \emptyset$, the empty set (and 0-th finite cardinal).
3. If $e = (e_1, \ldots, e_n) \in \mathcal{E}$ where $n \geq 1$, then

$$H^P(e) = \langle H^P(e_1), H^P((e_2, \ldots, e_n))\rangle.$$

$H^P(e)$ is called the *construction* of the expression e.

4.3 Structures

A *structure* for L is a tuple

$$S = (D_{\mathrm{v}}, D_{\mathrm{c}}, D_{\mathrm{s}}, D_{\mathrm{f}}, D_{\mathrm{o}}, D_{\mathrm{e}}, \in, \mathrm{T}, \mathrm{F}, \bot, H, \xi, I)$$

where:

1. $P = (D_c, \in)$ is a prestructure of Chiron. $D_{\mathrm{v}} = D_{\mathrm{v}}^P$, $D_{\mathrm{s}} = D_{\mathrm{s}}^P$, $D_{\mathrm{f}} = D_{\mathrm{f}}^P$, $D_{\mathrm{o}} = D_{\mathrm{o}}^P$, $D_{\mathrm{e}} = D_{\mathrm{e}}^P$, $\mathrm{T} = \mathrm{T}^P$, $\mathrm{F} = \mathrm{F}^P$, $\bot = \bot^P$, and $H = H^P$.

2. ξ is a choice function on D_{s}. Hence, for all nonempty superclasses Σ in D_{s}, $\xi(\Sigma)$ is a class in Σ.

3. For each operator $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$, $I(O)$ is an n-ary operation o in D_{o} from $D_1 \times \cdots \times D_n$ into D_{n+1} where $D_i = D_{\mathrm{s}}$ if $k_i = \mathsf{type}$, $D_i = D_{\mathrm{c}} \cup \{\bot\}$ if $\mathbf{type}[k_i]$, and $D_i = \{\mathrm{T}, \mathrm{F}\}$ if $k_i = \mathsf{formula}$ for each i with $1 \leq i \leq n+1$ such that:

   a. If O_1 and O_2 are similar, then $I(O_1) = I(O_2)$.

   b. If $O \notin L$, then, for all $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$, $I(O)(d_1, \ldots, d_n)$ is D_c, $\bot$, or F if $k_{n+1} = \mathsf{type}$, $\mathbf{type}[k_{n+1}]$, or $k_{n+1} = \mathsf{formula}$, respectively.

   c. $I((\mathsf{op}, \mathsf{set}, \mathsf{type}))(\,) = D_{\mathrm{v}}$, the universal class that contains all sets.

   d. $I((\mathsf{op}, \mathsf{class}, \mathsf{type}))(\,) = D_{\mathrm{c}}$, the universal superclass that contains all classes.

   e. $I((\mathsf{op}, \mathsf{expr}, \mathsf{type}))(\,) = D_{\mathrm{e}}$, the set of constructions (whose members represent expressions).

   f. $I((\mathsf{op}, \mathsf{expr\text{-}sym}, \mathsf{type}))(\,) = D_{\mathrm{e,sy}}$, the subset of D_{e} whose members represent symbols.

   g. $I((\mathsf{op}, \mathsf{expr\text{-}op}, \mathsf{type}))(\,) = D_{\mathrm{e,op}}$, the subset of D_{e} whose members represent operators.

   h. $I((\mathsf{op}, \mathsf{expr\text{-}type}, \mathsf{type}))(\,) = D_{\mathrm{e,ty}}$, the subset of D_{e} whose members represent types.

   i. $I((\mathsf{op}, \mathsf{expr\text{-}term}, \mathsf{type}))(\,) = D_{\mathrm{e,te}}$, the subset of D_{e} whose members represent terms.

   j. If x is a member of $D_{\mathrm{c}} \cup \{\bot\}$, then

      $$I((\mathsf{op}, \mathsf{expr\text{-}term\text{-}type}, \mathsf{E}_{\mathrm{ty}}, \mathsf{type}))(x)$$

      is the subset of $D_{\mathrm{e,te}}$ whose members represent terms of the type represented by x if x is in $D_{\mathrm{e,ty}}$ and is $D_{\mathrm{e,te}}$ otherwise.

k. $I((\mathsf{op}, \mathsf{expr\text{-}formula}, \mathsf{type}))(\,) = D_{\mathrm{e,fo}}$, the subset of D_{e} whose members represent formulas.

l. If x and y are members of $D_{\mathrm{c}} \cup \{\bot\}$, then

$$I((\mathsf{op}, \mathsf{in}, \mathsf{V}, \mathsf{C}, \mathsf{formula}))(x, y)$$

is T if x is a member of y (and hence x is a member of D_{v}) and is F otherwise.

m. If Σ, Σ' are members of D_{s}, then

$$I((\mathsf{op}, \mathsf{type\text{-}equal}, \mathsf{type}, \mathsf{type}, \mathsf{formula}))(\Sigma, \Sigma')$$

is T if Σ and Σ' are identical and is F otherwise.

n. If x, y are members of $D_{\mathrm{c}} \cup \{\bot\}$ and Σ is a member of D_{s}, then

$$I((\mathsf{op}, \mathsf{term\text{-}equal}, \mathsf{C}, \mathsf{C}, \mathsf{type}, \mathsf{formula}))(x, y, \Sigma)$$

is T if x, y are identical members of Σ and is F otherwise.

o. If t, t' are members of $\{\mathrm{T}, \mathrm{F}\}$, then

$$I((\mathsf{op}, \mathsf{formula\text{-}equal}, \mathsf{formula}, \mathsf{formula}, \mathsf{formula}))(t, t')$$

is T if t and t' are identical and is F otherwise.

p. If t is a member of $\{\mathrm{T}, \mathrm{F}\}$, then

$$I((\mathsf{op}, \mathsf{not}, \mathsf{formula}, \mathsf{formula}))(t)$$

is T if t is F and is F otherwise.

q. If t, t' are members of $\{\mathrm{T}, \mathrm{F}\}$, then

$$I((\mathsf{op}, \mathsf{or}, \mathsf{formula}, \mathsf{formula}, \mathsf{formula}))(t, t')$$

is T if at least one of t and t' is T and is F otherwise.

Fix a structure

$$S = (D_{\mathrm{v}}, D_{\mathrm{c}}, D_{\mathrm{s}}, D_{\mathrm{f}}, D_{\mathrm{o}}, D_{\mathrm{e}}, \in, \mathrm{T}, \mathrm{F}, \bot, H, \xi, I)$$

for L. An *assignment* into S is a mapping that assigns a class in D_{c} to each symbol in $\mathcal{S}$. Given an assignment φ into S, a symbol $s \in \mathcal{S}$, and a class $d \in D_{\mathrm{c}}$, let $\varphi[s \mapsto d]$ be the assignment φ' into S such that $\varphi'(s) = d$ and $\varphi'(t) = \varphi(t)$ for all symbols $t \in \mathcal{S}$ different from s. Let $\mathsf{assign}(S)$ be the collection of assignments into S.

4.4 Valuations

A *valuation* for S is a possibly partial mapping V from $\mathcal{E} \times \mathsf{assign}(S)$ into $D_\mathrm{o} \cup D_\mathrm{s} \cup \{\mathrm{T}, \mathrm{F}, \bot\}$ such that, for all $e \in \mathcal{E}$ and $\varphi \in \mathsf{assign}(S)$, if $V_\varphi(e)$ is defined, then $V_\varphi(e) \in D_\mathrm{o}$ if e is an operator, $V_\varphi(e) \in D_\mathrm{s}$ if e is a type, $V_\varphi(e) \in D_\mathrm{c} \cup \{\bot\}$ if e is a term, and $V_\varphi(e) \in \{\mathrm{T}, \mathrm{F}\}$ if e is a formula. (We write $V_\varphi(e)$ instead of $V(e, \varphi)$.)

Let the *standard valuation for* S be the valuation V for S defined recursively by the following statements:

1. Let $e \in \mathcal{E}$ be improper. Then $V_\varphi(e)$ is undefined.

2. Let $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$ be proper. Then $V_\varphi(O) = I(O)$.

3. Let $e = (\mathsf{op\text{-}app}, O, e_1, \ldots, e_n)$ be proper where $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$. Let

   $$d = V_\varphi(O)(V_\varphi(e_1), \ldots, V_\varphi(e_n)).$$

   If $V_\varphi(e_i)$ is in $V_\varphi(k_i)$ or $V_\varphi(e_i) = \bot$ for all i such that $1 \leq i \leq n$ and $\mathbf{type}[k_i]$ and d is in $V_\varphi(k_{n+1})$ or $d = \bot$ when $\mathbf{type}[k_{n+1}]$, then $V_\varphi(e) = d$. Otherwise $V_\varphi(e)$ is D_c if $k_{n+1} = \mathsf{type}$, $\bot$ if $\mathbf{type}[k_{n+1}]$, and F if $k_{n+1} = \mathsf{formula}$.

4. Let $a = (\mathsf{var}, x, \alpha)$ be proper. If $\varphi(x)$ is in $V_\varphi(\alpha)$, then $V_\varphi(a) = \varphi(x)$. Otherwise $V_\varphi(a) = \bot$.

5. Let $\beta = (\mathsf{type\text{-}app}, \alpha, a)$ be proper. If $V_\varphi(a) \neq \bot$, then $V_\varphi(\beta) = V_\varphi(\alpha)[V_\varphi(a)]$. Otherwise $V_\varphi(\beta) = D_\mathrm{c}$.

6. Let $\gamma = (\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$ be proper. Then $V_\varphi(\gamma)$ is the superclass of all g in D_f such that:

   a. For all sets d in $V_\varphi(\alpha)$, if $g(d)$ is defined, then $g(d)$ is in $V_{\varphi[x \mapsto d]}(\beta)$.
   b. For all sets d not in $V_\varphi(\alpha)$, $g(d)$ is undefined.

7. Let $b = (\mathsf{fun\text{-}app}, f, a)$ be proper. If $V_\varphi(f) \neq \bot$, $V_\varphi(a) \neq \bot$, and $V_\varphi(f)(V_\varphi(a))$ is defined, then $V_\varphi(b) = V_\varphi(f)(V_\varphi(a))$. Otherwise $V_\varphi(b) = \bot$.

8. Let $f = (\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$ be proper. If

   $$g = \{\langle d, d' \rangle \mid d \text{ is a set in } V_\varphi(\alpha) \text{ and } d' = V_{\varphi[x \mapsto d]}(b) \text{ is a set}\}$$

   is in D_f, then $V_\varphi(f) = g$. Otherwise $V_\varphi(f) = \bot$.

9. Let $a = (\mathsf{if}, A, b, c)$ be proper. If $V_\varphi(A) = \textsc{t}$, then $V_\varphi(a) = V_\varphi(b)$. Otherwise $V_\varphi(a) = V_\varphi(c)$.

10. Let $A = (\mathsf{exist}, (\mathsf{var}, x, \alpha), B)$ be proper. If there is some d in $V_\varphi(\alpha)$ such that $V_{\varphi[x \mapsto d]}(B) = \textsc{t}$, then $V_\varphi(A) = \textsc{t}$. Otherwise, $V_\varphi(A) = \textsc{f}$.

11. Let $a = (\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$ be proper. If there is a unique d in $V_\varphi(\alpha)$ such that $V_{\varphi[x \mapsto d]}(B) = \textsc{t}$, then $V_\varphi(a) = d$. Otherwise, $V_\varphi(a) = \bot$.

12. Let $a = (\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$ be proper. If there is some d in $V_\varphi(\alpha)$ such that $V_{\varphi[x \mapsto d]}(B) = \textsc{t}$, then $V_\varphi(a) = \xi(\Sigma)$ where Σ is the superclass of all d in $V_\varphi(\alpha)$ such that $V_{\varphi[x \mapsto d]}(B) = \textsc{t}$. Otherwise, $V_\varphi(a) = \bot$.

13. Let $a = (\mathsf{quote}, e)$ be proper. $V_\varphi(a) = H(e)$.

14. Let $b = (\mathsf{eval}, a, k)$ be proper. If (1) $V_\varphi(a)$ is in $D_{\mathrm{e,ty}}$ and $k = \mathsf{type}$, $V_\varphi(a)$ is in $D_{\mathrm{e,te}}$ and $\mathbf{type}[k]$, or $V_\varphi(a)$ is in $D_{\mathrm{e,fo}}$ and $k = \mathsf{formula}$; (2) $H^{-1}(V_\varphi(a))$ is eval-free; and (3) $V_\varphi(H^{-1}(V_\varphi(a)))$ is in $V_\varphi(k)$ if $\mathbf{type}[k]$, then $V_\varphi(b) = V_\varphi(H^{-1}(V_\varphi(a)))$. Otherwise $V_\varphi(b)$ is D_{c} if $k = \mathsf{type}$, $\bot$ if $\mathbf{type}[k]$, and $\textsc{f}$ if $k = \mathsf{formula}$.

Lemma 4.1 *The standard valuation for S is well defined.*

Proof We will show that, for all $e \in \mathcal{E}$ and $\varphi \in \mathsf{assign}(S)$, $V_\varphi(e)$ is well defined. Our proof is by induction on the complexity of e. There are 14 cases corresponding to the 14 clauses of the definition of V.

Case 1: e is improper. $V_\varphi(e)$ is alway undefined in this case.

Case 2: $e = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$ is proper. $V_\varphi(e) = I(e)$ is well defined since I is a well-defined component of S.

Cases 3-12: e is a proper expression that is not an operator, quotation, or evaluation. For each case, $V_\varphi(e)$ depends on well-defined components of S and a collection of values $V_{\varphi'}(e')$ where e' ranges over a set of subexpressions of e and φ' is φ or ranges over an infinite subset of $\mathsf{assign}(S)$. Each such $V_{\varphi'}(e')$ is well defined by the induction hypothesis because e' is a subexpression of e, and hence, $c(e') < c(e)$. Therefore, $V_\varphi(e)$ is well-defined.

Case 13: $e = (\mathsf{quote}, e')$ is proper. $V_\varphi(e) = H(e')$ is well defined since H is a well-defined component of S.

Case 14: $e = (\mathsf{eval}, a, k)$ is proper. $V_\varphi(e)$ depends on one, two, or three of the values of $V_\varphi(a)$, $V_\varphi(k)$, and $V_\varphi(H^{-1}(V_\varphi(a)))$. $V_\varphi(a)$ and $V_\varphi(k)$ when $\mathbf{type}[k]$ holds are well defined by the induction hypothesis because a and k are subexpressions of e, and hence, $c(a) < c(e)$ and $c(k) < c(e)$. $V_\varphi(H^{-1}(V_\varphi(a)))$ when $H^{-1}(V_\varphi(a)) \in \mathcal{E}$ is well defined by the induction hypothesis because $H^{-1}(V_\varphi(a))$ is eval-free and e contains at least one occurrence of the symbol eval, and hence, $c(H^{-1}(V_\varphi(a))) < c(e)$. Therefore, $V_\varphi(e)$ is well-defined.

□

Theorem 4.2 *Let V be the standard valuation for S. Then the following statements hold for all $e \in \mathcal{E}$ and $\varphi \in \mathsf{assign}(S)$:*

1. *$V_\varphi(e)$ is defined iff e is proper.*
2. *If e is an n-ary operator, then $V_\varphi(e)$ is an n-ary operation in D_o.*
3. *If e is a type, then $V_\varphi(e)$ is in D_s.*
4. *If e is a term of type α, then $V_\varphi(e)$ is in $V_\varphi(\alpha) \cup \{\bot\}$.*
5. *If e is a formula, then $V_\varphi(e)$ is in $\{\mathrm{T}, \mathrm{F}\}$.*

Proof By Lemma 4.1, the standard valuation for S is well defined. Parts 1, 2, 3, and 5 of the theorem follow immediately from the definitions of a structure and the standard valuation for a structure.

Our proof of part 4 is by induction on the length of the term e.

Case 1: $e = (\mathsf{op\text{-}app}, O, e_1, \ldots, e_n)$ where $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$. Then e is of type k_{n+1}. By the definition of V, $V_\varphi(e)$ is in $V_\varphi(k_{n+1}) \cup \{\bot\}$.

Case 2: $e = (\mathsf{var}, x, \alpha)$. Then e is of type α. By the definition of V, $V_\varphi(e)$ is in $V_\varphi(\alpha) \cup \{\bot\}$.

Case 3: $e = (\mathsf{fun\text{-}app}, f, a)$ where f is of type α. Then e is of type $\alpha(a)$. By the induction hypothesis, $V_\varphi(f)$ is in $V_\varphi(\alpha) \cup \{\bot\}$. By the definition of V, if $V_\varphi(e) = V_\varphi(f)(V_\varphi(a)) \neq \bot$, then $V_\varphi(f) \neq \bot$ and $V_\varphi(a) \neq \bot$, and so $V_\varphi(e)$ is in $V_\varphi(\alpha)[V_\varphi(\alpha)] = V_\varphi(\alpha(a))$. Therefore, $V_\varphi(e)$ is in $V_\varphi(\alpha(a)) \cup \{\bot\}$.

Case 4: $e = (\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$ where b is of type β. Then e is of type $\gamma = (\Lambda\, x : \alpha\ .\ \beta)$. By the induction hypothesis, $V_{\varphi'}(b)$ is in $V_{\varphi'}(\beta) \cup \{\bot\}$ for all $\varphi' \in \mathsf{assign}(S)$. Suppose $g = V_\varphi(e) \neq \bot$. For all sets d in $V_\varphi(\alpha)$, if $g(d)$ is defined, $g(d) = V_{\varphi[x \mapsto d]}(b)$ is a set in $V_{\varphi[x \mapsto d]}(\beta)$. For all sets d not in $V_\varphi(\alpha)$, $g(d)$ is undefined. Therefore, by the definition of V, $V_\varphi(e)$ is in $V_\varphi(\gamma) \cup \{\bot\}$.

Case 5: $e = (\mathsf{if}, A, b, c)$ where b is of type β and c is of type γ. Then e is of type

$$\delta = \begin{cases} \beta & \text{if } \beta = \gamma \\ \mathsf{C} & \text{otherwise} \end{cases}$$

By the induction hypothesis, $V_\varphi(b)$ is in $V_\varphi(\beta) \cup \{\bot\}$ and $V_\varphi(c)$ is in $V_\varphi(\gamma) \cup \{\bot\}$. $V_\varphi(\beta) \subseteq V_\varphi(\mathsf{C})$ and $V_\varphi(\gamma) \subseteq V_\varphi(\mathsf{C})$. Therefore, by the definition of V, $V_\varphi(e)$ is in $V_\varphi(\delta) \cup \{\bot\}$.

Case 6: $e = (\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$. Then e is of type α. By the definition of V, $V_\varphi(e)$ is in $V_\varphi(\alpha) \cup \{\bot\}$.

Case 7: $e = (\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$. Then e is of type α. By the definition of V, $V_\varphi(e)$ is in $V_\varphi(\alpha) \cup \{\bot\}$.

Case 8: $e = (\mathsf{quote}, e)$. Then e is of type E. By the definition of H, I, and V, $V_\varphi(e)$ is in $D_\mathrm{e} = V_\varphi(\mathsf{E})$.

Case 9: $e = (\mathsf{eval}, a, k)$. Then e is of type k. By the definition of V, $V_\varphi(e)$ is in $V_\varphi(k) \cup \{\bot\}$.

□

4.5 Models

A *model for* L is a pair $M = (S, V)$ where S is a structure for L and V is a valuation for S. Let $M = (S, V)$ be a model for L. An expression e is *denoting* [*nondenoting*] in M with respect to an assignment φ if $V_\varphi(e)$ is defined [undefined]. A denoting term a is *defined* [*undefined*] in M with respect to φ if $V_\varphi(a)$ is in D_c [$V_\varphi(a) = \bot$]. If an expression e is denoting in M with respect to φ, then its *value* in M with respect to φ is $V_\varphi(e)$.

A model $M = (S, V)$ for L is a *standard model* for L if V is the standard valuation for S. The official semantics of Chiron is based on standard models. A formula A is *valid in* M, written $M \models A$, if $V_\varphi(A) = \mathrm{T}$ for all

$\varphi \in \mathsf{assign}(S)$. A is *valid*, written $\models A$, if $M \models A$ for all standard models M. A proper expression e is *semantically closed* if, for all standard models $M = (S, V)$, $V_\varphi(e)$ does not depend on φ. A *sentence* is a formula that is semantically closed. A *standard model* of a set Γ of formulas is a standard model M such that $M \models A$ for all $A \in \Gamma$.

4.6 Theories

A *theory* of Chiron is pair $T = (L, \Gamma)$ where L is a language of Chiron and Γ is a set of sentences called the *axioms* of T. A *standard model* of T is a standard model for L that is a standard model of Γ. A formula A is *valid in* T, written $T \models A$, if $M \models A$ for all standard models M of T.

Let $T_i = (L_i, \Gamma_i)$ be a theory of Chiron for $i = 1, 2$. T_2 is an *extension* of T_1, written $T_1 \leq T_2$, if $L_1 \subseteq L_2$ and $\Gamma_1 \subseteq \Gamma_2$.

4.7 Notes concerning the Semantics of Chiron

Fix a standard model $M = (S, V)$ for L.

1. An improper expression never has a value, but its quotation (as well as the quotation of any proper expression) always has a value. The latter is a set called a construction that represents the syntactic structure of the expression as a tree of finite cardinals.

2. The components of an operator's signature will normally be semantically closed. This is the case for all the built-in operators of Chiron as well as all the operators defined in section 6.

3. Dependent function types, function abstractions, existential quantifications, definition descriptions, and indefinite descriptions are *variable binding* expressions. According to the semantics of Chiron, a variable binding $(\star\, x : \alpha\ .\ e)$, where $\star$ is Λ, λ, $\exists$, ι, or ϵ, binds all the "free" variables occurring in e that are similar to $(x : \alpha)$. Variables are bound in the traditional, naive way. It would be possible to use other more sophisticated variable binding mechanisms such as de Bruijn notation [2] or nominal datatypes [14, 20].

4. The type α of a variable $(\mathsf{var}, x, \alpha)$ restricts the value of a variable in two ways. First, if $(\mathsf{var}, x, \alpha)$ immediately follows Λ, λ, $\exists$, ι, or ϵ in a variable binding expression, then the values assigned to x for the body of the variable binding expression are restricted to the values in the superclass denoted by α. Second, $V_\varphi((\mathsf{var}, x, \alpha)) = \varphi(x)$ iff $\varphi(x)$ is in

the superclass denoted by α. ($V_\varphi((\mathsf{var}, x, \alpha)) = \bot$ if $\varphi(x)$ is not in the superclass denoted by α.)

5. The notions of a free variable, substitution for a variable, variable capturing, etc. can be formalized in Chiron as defined operators (see subsection 6.4.) As a result, syntactic side conditions can be expressed directly within Chiron formulas. However, these notions are more complicated in Chiron than in traditional predicate logic due to the presence of evaluation. For example, when the value of $(e : \mathsf{E}_{\mathrm{te}})$ equals the value of $\lceil (y : \mathsf{C}) \rceil$ with $x \neq y$, the variable $(y : \mathsf{C})$ is free in

   $$\forall\, x : \mathsf{C} \,.\, x = [\![(e : \mathsf{E}_{\mathrm{te}})]\!]_{\mathrm{te}}$$

   because this formula is semantically equivalent to

   $$\forall\, x : \mathsf{C} \,.\, x = (y : \mathsf{C}).$$

6. Since variables denote classes, they can be called *class variables*. There are no operation, superclass, or truth value variables in Chiron. Thus direct quantification over operations, superclasses, and truth values is not possible. Direct quantification over the undefined value $\bot$ is also not possible. However, indirect quantification over "definable" operations, "definable" superclasses, "definable" members of $D_{\mathrm{c}} \cup \{\bot\}$, or truth values can be done via variables of type E_{op}, E_{ty}, E_{te}, or E_{fo}, respectively. As in standard NBG set theory, only classes are first-class values in Chiron.

7. Since sets and classes are superclasses, a type may denote a set or a proper class. In particular, a type may denote the *empty set*. That is, types in Chiron are allowed to be empty. Empty types result, for example, from a type application $\alpha(a)$ where a denotes a value that is not in the domain of any function in the superclass that α denotes.

8. α is a *subtype* of β in M if $V_\varphi(\alpha) \subseteq V_\varphi(\beta)$ for all $\varphi \in \mathsf{assign}(S)$. For example, E is a subtype of V in every standard model. C is the *universal type* by virtue of denoting D_c, the universal superclass. Every type is a subtype of C in every standard model.

9. An application of a term denoting a function to an undefined term is itself undefined. That is, function application is strict with respect to undefinedness. In contrast, the application of term operators is not necessarily strict with respect to undefinedness.

10. Suppose $f(a)$ is a function application where f is a term of type $(\Lambda\, x : \alpha\ .\ \beta)$. Then a could be any term of any type whatsoever. However, if the value of a is not a set in the value of α, then $f(a)$ is undefined. Similarly, suppose

$$(s :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_{n+1})(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n)$$

is an operator application. Then a could be any term of any type whatsoever. However, if the value of a is a class not in the value of α, then the value of this operator application is D_{c} if $k_{n+1} = \mathsf{type}$, $\perp$ if $\mathbf{type}[k_{n+1}]$, and F if $k_{n+1} = \mathsf{formula}$.

11. Suppose a built-in operator

$$(\mathsf{op}, \mathsf{lub}, \mathsf{type}, \mathsf{type}, \mathsf{type})$$

is added to Chiron that denotes an operation that, given superclasses Σ_1 and Σ_2, returns a superclass that is the least upper bound of Σ_1 and Σ_2. Then formation rule P-Expr-8 could be modified so that a conditional term (if, A, b, c) is assigned the type

$$(\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{lub}, \mathsf{type}, \mathsf{type}, \mathsf{type}), \beta, \gamma)$$

where β and γ are the types of b and c, respectively.

12. A *Gödel number* [8] is a number that encodes an expression. Analogously, a *Gödel set* is a set that encodes an expression. Employing this terminology, the construction that represents an expression $e \in \mathcal{E}$ is the *Gödel set* of e which is denoted by (quote, e). Hence, "Gödel numbering" is built into Chiron.

13. When an evaluation $b = (\mathsf{eval}, a, k)$ is "semantically well-formed", the value of b is, roughly speaking, *the value of the value of* a.

14. An "ungrounded expression" is considered to be an undefined expression. For example, the value of an ungrounded formula like $[\![\mathsf{LIAR}]\!]_{\mathrm{fo}}$ is F.

15. Operators of the form $(s :: k)$ are applied to an empty tuple of arguments. The value of $(s :: k)$ is a 0-ary operation o such that $o(\,) = v$ for some value v. We will sometimes abuse terminology and say that the value of $(s :: k)$ is v instead of o.

16. A value $x \in D_{\mathrm{e}}$ is a construction that represents an expression of Chiron. We will sometimes abuse terminology and say $x \in D_{\mathrm{e}}$ is an *expression*. Similarly, we will sometimes say that $x \in D_{\mathrm{e,sy}}$, $x \in D_{\mathrm{e,op}}$, $x \in D_{\mathrm{e,ty}}$, $x \in D_{\mathrm{e,te}}$, or $x \in D_{\mathrm{e,fo}}$ is a *symbol*, *operator*, *type*, *term*, or *formula*, respectively.

5 Relationship to NBG

We show in this section that there is a faithful interpretation of NBG set theory in Chiron. Loosely speaking, this means Chiron is a conservative extension of NBG. That is, Chiron adds new reasoning machinery to NBG without compromising the underlying semantics of NBG.

NBG is usually formulated as a theory in first-order logic over a language L_{nbg} containing an infinite set $\mathcal{V}$ of variables, a unary predicate symbol V, binary predicates symbols $=$ and $\in$, and some complete set of logical connectives (say $\neg, \vee, \exists$). Assume $\mathcal{V} \subseteq \mathcal{S}$, i.e., each variable of L_{nbg} is a symbol of Chiron. Let $\mathcal{F}_{\mathrm{nbg}}$ denote the set of formulas of L_{nbg}. A *model* of NBG is a structure $N = (D^N, V^N, =^N, \in^N)$ for L_{nbg} that satisfies the axioms of NBG. An *assignment* into N is a mapping that assigns a member of D^N to each variable $x \in \mathcal{V}$. Let $\mathsf{assign}(N)$ be the collection of assignments into N. The *valuation* for N is a total mapping W^N from $\mathcal{F}_{\mathrm{nbg}} \times \mathsf{assign}(N)$ to the set $\{\mathrm{T}, \mathrm{F}\}$ of truth values. A formula A of L_{nbg} is *valid*, written $\models_{\mathrm{nbg}} A$, if $W^N_\varphi(A) = \mathrm{T}$ for all models N of NBG and all $\varphi \in \mathsf{assign}(N)$.[3]

Let L be any language of Chiron. Suppose Φ is a total function that maps the terms (variables) of L_{nbg} to terms of L and the formulas of L_{nbg} to formulas of L. Φ is a *translation from* NBG *to* Chiron if Φ maps the sentences of L_{nbg} to sentences of L. Φ is an *interpretation of* NBG *in* Chiron if Φ is a translation from NBG to Chiron and, for all sentences A of L_{nbg}, $\models_{\mathrm{nbg}} A$ implies $\models \Phi(A)$. That is, Φ is an interpretation of NBG in Chiron if Φ is a meaning-preserving translation from NBG to Chiron. Φ is a *faithful interpretation of* NBG *in* Chiron if Φ is an interpretation of NBG in Chiron and, for all sentences A of L_{nbg}, $\models \Phi(A)$ implies $\models_{\mathrm{nbg}} A$. That is, Φ is a faithful interpretation of NBG in Chiron if Chiron is a conservative extension of the image of NBG under Φ.

Let Φ be the total function, mapping the variables of L_{nbg} to variables of L and the formulas of L_{nbg} to formulas of L, recursively defined by:

1. If $x \in \mathcal{V}$, then $\Phi(x) = (x : \mathsf{C})$.

[3] As above for a valuation V, we write $W^N(A, \varphi)$ as $W^N_\varphi(A)$.

2. If $V(x)$ is a formula of L_{nbg}, then $\Phi(V(x)) = (\Phi(x) =_{\mathsf{V}} \Phi(x))$.

3. If $(x = y)$ is a formula of L_{nbg}, then $\Phi((x = y)) = (\Phi(x) = \Phi(y))$.

4. If $(x \in y)$ is a formula of L_{nbg}, then $\Phi((x \in y)) = (\Phi(x) \in \Phi(y))$.

5. If $(\neg A)$ is a formula of L_{nbg}, then $\Phi((\neg A)) = (\neg \Phi(A))$.

6. If $(A \vee B)$ is a formula of L_{nbg}, then $\Phi((A \vee B)) = (\Phi(A) \vee \Phi(B))$.

7. If $(\exists\, x\ .\ A)$ is a formula of L_{nbg}, then $\Phi((\exists\, x\ .\ A)) = (\exists\, x : \mathsf{C}\ .\ \Phi(A))$.

Lemma 5.1 *Let* $N = (D^N, V^N, =^N, \in^N)$ *be a model of* NBG *and* $M = (S, V)$ *be a standard model for* L*, where*

$$S = (D_{\mathrm{v}}, D_{\mathrm{c}}, D_{\mathrm{s}}, D_{\mathrm{f}}, D_{\mathrm{o}}, D_{\mathrm{e}}, \in, \mathrm{T}, \mathrm{F}, \bot, H, \xi, I),$$

such that $(D^N, \in^N)$ *is identical to the prestructure* $(D_{\mathrm{c}}, \in)$*.*

1. *For all* $d \in D^N$*,* $V^N(d)$ *iff* d *is in* $I(\mathsf{V})$*.*

2. *For all* $d, d' \in D^N$*,* $d =^N d'$ *iff* $d\ I(=)\ d'$*.*

3. *For all* $d, d' \in D^N$*,* $d \in^N d'$ *iff* $d\ I(\in)\ d'$*.*

4. $\mathsf{assign}(N) = \mathsf{assign}(M)$*.*

5. *For all formulas* A *of* L_{nbg} *and* $\varphi \in \mathsf{assign}(N)$*,* $W^N_\varphi(A) = V_\varphi(\Phi(A))$*.*

Proof Clauses 1–4 are obvious. Clause 5 is easily proved by induction on the structure of the formulas of L_{nbg} since the logical connectives $\neg, \vee, \exists$ of L_{nbg} have the same meanings as the negation operator, the disjunction operator, and existential quantification, respectively, in Chiron. □

Theorem 5.2 *For all sentences* A *of* L_{nbg}*,*

$$\models_{\mathrm{NBG}} A \quad \textit{iff} \quad \models \Phi(A).$$

That is, Φ *is a faithful interpretation of* NBG *in Chiron.*

Proof For every model $N = (D^N, V^N, =^N, \in^N)$ of NBG, there is a standard model $M = (S, V)$ for L, where

$$S = (D_{\mathrm{v}}, D_{\mathrm{c}}, D_{\mathrm{s}}, D_{\mathrm{f}}, D_{\mathrm{o}}, D_{\mathrm{e}}, \in, \mathrm{T}, \mathrm{F}, \bot, H, \xi, I),$$

such that $(D^N, \in^N)$ is identical to the prestructure $(D_c, \in)$. Likewise, for every standard model $M = (S, V)$ for L, where

$$S = (D_v, D_c, D_s, D_f, D_o, D_e, \in, \mathrm{T}, \mathrm{F}, \bot, H, \xi, I),$$

there is a model $N = (D^N, V^N, =^N, \in^N)$ of NBG such that the prestructure $(D_c, \in)$ is identical to $(D^N, \in^N)$. The theorem follows from this observation and clause 5 of Lemma 5.1. □

6 Operator Definitions

There are two ways of assigning a meaning to an operator. The first is to make the operator one of the built-in operators like $(\mathsf{op}, \mathsf{set}, \mathsf{type})$ ($(\mathsf{set} :: \mathsf{type})$ in compact notation) and then define its meaning as part of the definition of a standard model. The second is to construct one or more sentences that define the meaning of the operator. We will use this latter approach to define several useful logical, set-theoretic, syntactic, and substitution operators.

6.1 Logical Operators

1. **Truth**
   Operator: $(\mathsf{true} :: \mathsf{formula})$
   Definition:

   $$(\mathsf{true} :: \mathsf{formula})(\,) \equiv \mathsf{C} =_{\mathrm{ty}} \mathsf{C}.$$

   Compact notation:

   T means $(\mathsf{true} :: \mathsf{formula})(\,)$.

2. **Falsehood**
   Operator: $(\mathsf{false} :: \mathsf{formula})$
   Definition:

   $$(\mathsf{false} :: \mathsf{formula})(\,) \equiv \mathsf{V} =_{\mathrm{ty}} \mathsf{C}.$$

   Compact notation:

   F means $(\mathsf{false} :: \mathsf{formula})(\,)$.

3. **Conjunction**
   Operator: $(\mathsf{and} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})$
   Definition:

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{fo}} \,.\, (\mathsf{and} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})([\![e_1]\!], [\![e_2]\!]) \equiv \neg(\neg[\![e_1]\!] \vee \neg[\![e_2]\!]).$$

   Compact notation:

$$(A \wedge B) \text{ means } (\mathsf{and} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})(A, B).$$

   Note: This definition is a schema expressed in Chiron as a universal sentence whose quantified variables range over constructions representing formulas. Several of the operators that follow will also be defined as schemas of this kind.

4. **Implication**
   Operator: $(\mathsf{implies} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})$
   Definition:

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{fo}} \,.\, (\mathsf{implies} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})([\![e_1]\!], [\![e_2]\!]) \equiv \neg[\![e_1]\!] \vee [\![e_2]\!].$$

   Compact notation:

$$(A \supset B) \text{ means } (\mathsf{implies} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})(A, B).$$

5. **Definedness in a Type**
   Operator: $(\mathsf{defined\text{-}in} :: \mathsf{C}, \mathsf{type}, \mathsf{formula})$
   Definition:

$$\forall\, e_1 : \mathsf{E}_{\mathrm{te}}, e_2 : \mathsf{E}_{\mathrm{ty}} \,.\, (\mathsf{defined\text{-}in} :: \mathsf{C}, \mathsf{type}, \mathsf{formula})([\![e_1]\!], [\![e_2]\!]) \equiv [\![e_1]\!] =_{[\![e_2]\!]} [\![e_1]\!].$$

   Compact notation:

$(a \downarrow \alpha)$ means $(\mathsf{defined\text{-}in} :: \mathsf{C}, \mathsf{type}, \mathsf{formula})(a, \alpha)$.
$(a \uparrow \alpha)$ means $\neg(a \downarrow \alpha)$.
$(a\!\downarrow)$ means $(a \downarrow \mathsf{C})$.
$(a\!\uparrow)$ means $\neg(a\!\downarrow)$.

6. **Quasi-Equality**
   Operator: $(\mathsf{quasi\text{-}equal} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})$
   Definition:

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}} \,.\, (\mathsf{quasi\text{-}equal} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})([\![e_1]\!], [\![e_2]\!]) \equiv \\ ([\![e_1]\!]\downarrow \vee [\![e_2]\!]\downarrow) \supset [\![e_1]\!] = [\![e_2]\!].$$

   Compact notation:

$(a \simeq b)$ means $(\mathsf{quasi\text{-}equal} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})(a, b)$.

$(a \not\simeq b)$ means $\neg(a \simeq b)$.

7. **Empty Type**
   Operator: $(\mathsf{empty\text{-}type} :: \mathsf{type})$
   Definition:

$$\neg(\exists\, x : \mathsf{C} \,.\, x \downarrow (\mathsf{empty\text{-}type} :: \mathsf{type})(\,)).$$

   Compact notation:

∇ means $(\mathsf{empty\text{-}type} :: \mathsf{type})(\,)$.

8. **Type Order**
   Operator: $(\mathsf{type\text{-}le} :: \mathsf{type}, \mathsf{type}, \mathsf{formula})$
   Definition:

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{ty}} \,.\, (\mathsf{type\text{-}le} :: \mathsf{type}, \mathsf{type}, \mathsf{formula})([\![e_1]\!], [\![e_2]\!]) \equiv \\ \forall\, e : \mathsf{E}_{\mathrm{sy}} \,.\, (\neg\mathsf{free\text{-}in}(e, e_1) \wedge \neg\mathsf{free\text{-}in}(e, e_2)) \supset \\ [\![\lceil \forall\, \lfloor e \rfloor : \lfloor e_1 \rfloor \,.\, \lfloor e \rfloor \downarrow \lfloor e_2 \rfloor \rceil]\!]_{\mathrm{fo}}.$$

   Compact notation:

$(\alpha \ll \beta)$ means $(\mathsf{type\text{-}le} :: \mathsf{type}, \mathsf{type}, \mathsf{formula})(\alpha, \beta)$.

   Note: This definition is a schema of a more complex form than the schemas we have seen previously in this section. It utilizes the free-in operator defined in subsection 6.4 and quasiquotation.

9. **Canonical Undefined Term**
   Operator: $(\mathsf{undefined} :: \mathsf{C})$
   Definition:

$$(\mathsf{undefined} :: \mathsf{C})(\,) \simeq \iota\, x : \mathsf{C} \,.\, x \neq x.$$

   Compact notation:

   $\bot_{\mathsf{C}}$ means $(\mathsf{undefined} :: \mathsf{C})(\,)$.

6.2 Set-Theoretic Operators

1. **Empty set**
   Operator: $(\mathsf{empty\text{-}set} :: \mathsf{V})$
   Definition:

$$(\mathsf{empty\text{-}set} :: \mathsf{V})(\,) \simeq \iota\, u : \mathsf{V} \,.\, \forall\, v : \mathsf{V} \,.\, v \notin u.$$

   Compact notation:

   $\emptyset$ means $(\mathsf{empty\text{-}set} :: \mathsf{V})(\,)$.

2. **Pair**
   Operator: $(\mathsf{pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})$
   Definition:

$$\forall\, u, v : \mathsf{V} \,.\, (\mathsf{pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(u, v) \simeq \\ \quad \iota\, w : \mathsf{V} \,.\, \forall\, w' : \mathsf{V} \,.\, w' \in w \equiv (w' = u \vee w' = v).$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}} \,.\, (\llbracket e_1 \rrbracket\! \uparrow \vee \llbracket e_2 \rrbracket\! \uparrow) \supset \\ \quad (\mathsf{pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)\! \uparrow .$$

   Compact notation:

   $\{a, b\}$ means $(\mathsf{pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(a, b)$.

   $\{a\}$ means $\{a, a\}$.

   Note: The second defining sentence asserts that pair is strict with respect to undefinedness. Most of the operators defined in the rest of this subsection are similarly strict with respect to undefinedness.

3. **Ordered Pair**
   Operator: $(\mathsf{ord\text{-}pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})$
   Definition:

$$\forall\, u, v : \mathsf{V} \;.\; (\mathsf{ord\text{-}pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(u, v) \simeq \{\{u\}, \{u, v\}\}.$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}} \;.\; (\llbracket e_1 \rrbracket\!\uparrow \vee \llbracket e_2 \rrbracket\!\uparrow) \supset (\mathsf{ord\text{-}pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)\!\uparrow .$$

   Compact notation:

$\langle a, b \rangle$ means $(\mathsf{ord\text{-}pair} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(a, b)$.
$\langle a_1, \ldots, a_n \rangle$ means $\langle a_1, \langle a_2, \ldots, a_n \rangle \rangle$ for $n \geq 3$.
$[\,]$ means $\emptyset$.
$[a_1, \ldots, a_n]$ means $\langle a_1, [a_2, \ldots, a_n] \rangle$ for $n \geq 1$.

4. **Subclass**
   Operator: $(\mathsf{subclass} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})$
   Definition:

$$\forall\, x, y : \mathsf{C} \;.\; (\mathsf{subclass} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})(x, y) \equiv \forall\, u : \mathsf{V} \;.\; u \in x \supset u \in y.$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}} \;.\; (\llbracket e_1 \rrbracket\!\uparrow \vee \llbracket e_2 \rrbracket\!\uparrow) \supset (\mathsf{subclass} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \equiv \mathsf{F}.$$

   Compact notation:

$a \subseteq b$ means $(\mathsf{subclass} :: \mathsf{C}, \mathsf{C}, \mathsf{formula})(a, b)$.
$a \subset b$ means $a \subseteq b \wedge a \neq b$.

5. **Universal class**
   Operator: $(\mathsf{universal\text{-}class} :: \mathsf{C})$
   Definition:

$$(\mathsf{univeral\text{-}class} :: \mathsf{C})(\,) \simeq \iota\, y : \mathsf{C} \;.\; \forall\, u : \mathsf{V} \;.\; u \in y.$$

Compact notation:

U means $(\mathsf{universal\text{-}set} :: \mathsf{C})()$.

6. **Union**
Operator: $(\mathsf{union} :: \mathsf{C}, \mathsf{C}, \mathsf{C})$
Definition:

$$\forall\, x, y : \mathsf{C}\ .\ (\mathsf{union} :: \mathsf{C}, \mathsf{C}, \mathsf{C})(x, y) \simeq \\ \quad \iota\, z : \mathsf{C}\ .\ \forall\, u : \mathsf{V}\ .\ u \in z \equiv (u \in x \vee u \in y).$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}}\ .\ (\llbracket e_1 \rrbracket\!\uparrow \vee \llbracket e_2 \rrbracket\!\uparrow) \supset \\ \quad (\mathsf{union} :: \mathsf{C}, \mathsf{C}, \mathsf{C})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)\!\uparrow\ .$$

Compact notation:

$a \cup b$ means $(\mathsf{union} :: \mathsf{C}, \mathsf{C}, \mathsf{C})(a, b)$.

7. **Intersection**
Operator: $(\mathsf{intersection} :: \mathsf{C}, \mathsf{C}, \mathsf{C})$
Definition:

$$\forall\, x, y : \mathsf{C}\ .\ (\mathsf{intersection} :: \mathsf{C}, \mathsf{C}, \mathsf{C})(x, y) \simeq \\ \quad \iota\, z : \mathsf{C}\ .\ \forall\, u : \mathsf{V}\ .\ u \in z \equiv (u \in x \wedge u \in y).$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}}\ .\ (\llbracket e_1 \rrbracket\!\uparrow \vee \llbracket e_2 \rrbracket\!\uparrow) \supset \\ \quad (\mathsf{intersection} :: \mathsf{C}, \mathsf{C}, \mathsf{C})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)\!\uparrow\ .$$

Compact notation:

$a \cap b$ means $(\mathsf{intersection} :: \mathsf{C}, \mathsf{C}, \mathsf{C})(a, b)$.

8. **Complement**
Operator: $(\mathsf{complement} :: \mathsf{C}, \mathsf{C})$
Definition:

$$\forall\, x : \mathsf{C}\ .\ (\mathsf{complement} :: \mathsf{C}, \mathsf{C})(x) \simeq \\ \quad \iota\, y : \mathsf{C}\ .\ \forall\, u : \mathsf{V}\ .\ u \in y \equiv u \notin x.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ \llbracket e \rrbracket\!\uparrow \supset \\ \quad (\mathsf{complement} :: \mathsf{C}, , \mathsf{C})(\llbracket e \rrbracket)\!\uparrow\ .$$

Compact notation:

$\overline{a}$ means $(\mathsf{complement} :: \mathsf{C}, \mathsf{C})(a)$.

9. **Head**
Operator: $(\mathsf{head} :: \mathsf{V}, \mathsf{V})$
Definition:

$$\forall\, w : \mathsf{V} \,.\, (\mathsf{head} :: \mathsf{V}, \mathsf{V})(w) \simeq \\ \quad \iota\, u : \mathsf{V} \,.\, \exists\, v : \mathsf{V} \,.\, w = \langle u, v \rangle.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \,.\, [\![e]\!]\!\uparrow \supset \\ \quad (\mathsf{head} :: \mathsf{V}, , \mathsf{V})([\![e]\!])\!\uparrow \,.$$

Compact notation:

$\mathsf{hd}(a)$ means $(\mathsf{head} :: \mathsf{V}, \mathsf{V})(a)$.

10. **Tail**
Operator: $(\mathsf{tail} :: \mathsf{V}, \mathsf{V})$
Definition:

$$\forall\, w : \mathsf{V} \,.\, (\mathsf{tail} :: \mathsf{V}, \mathsf{V})(w) \simeq \\ \quad \iota\, v : \mathsf{V} \,.\, \exists\, u : \mathsf{V} \,.\, w = \langle u, v \rangle.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \,.\, [\![e]\!]\!\uparrow \supset \\ \quad (\mathsf{tail} :: \mathsf{V}, , \mathsf{V})([\![e]\!])\!\uparrow \,.$$

Compact notation:

$\mathsf{tl}(a)$ means $(\mathsf{tail} :: \mathsf{V}, \mathsf{V})(a)$.

11. **Append**
Operator: $(\mathsf{append} :: \mathsf{V}, \mathsf{V}, \mathsf{V})$
Definition:

$$\forall\, x, y : \mathsf{V} \,.\, (\mathsf{append} :: \mathsf{V}, \mathsf{V}, \mathsf{V})(x, y) \simeq \\ \quad \mathsf{if}(x = \emptyset, y, \langle \mathsf{hd}(x), \mathsf{append}(\mathsf{tl}(x), y) \rangle).$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}} \,.\, ([\![e_1]\!]\!\uparrow \vee [\![e_2]\!]\!\uparrow) \supset \\ \quad (\mathsf{append} :: \mathsf{V}, \mathsf{V}, \mathsf{V})([\![e_1]\!], [\![e_2]\!])\!\uparrow \,.$$

Compact notation:

$a \hat{} b$ means $(\mathsf{append} :: \mathsf{V}, \mathsf{V})(a, b)$.

12. **Type Product**
Operator: $(\mathsf{type\text{-}prod} :: \mathsf{type}, \mathsf{type}, \mathsf{type})$
Definition:

$$\begin{array}{l}\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{ty}}, \forall\, e : \mathsf{E}_{\mathrm{sy}}\ . \\ \quad (\neg \mathsf{free\text{-}in}(e, e_1) \wedge \neg \mathsf{free\text{-}in}(e, e_2)) \supset \\ \quad [\![\lceil \forall\, \lfloor e \rfloor : \mathsf{C}\ . \\ \qquad \lfloor e \rfloor \downarrow (\mathsf{type\text{-}prod} :: \mathsf{type}, \mathsf{type}, \mathsf{type})(\lfloor e_1 \rfloor, \lfloor e_2 \rfloor) \equiv \\ \qquad \mathsf{hd}(\lfloor e \rfloor) \downarrow \lfloor e_1 \rfloor \wedge \mathsf{tl}(\lfloor e \rfloor) \downarrow \lfloor e_2 \rfloor \rceil]\!]_{\mathrm{fo}}.\end{array}$$

Compact notation:

$(\alpha \times \beta)$ means $(\mathsf{type\text{-}prod} :: \mathsf{type}, \mathsf{type}, \mathsf{prod})(\alpha, \beta)$.

Note: This definition is a schema similar to the schema used to define the type-le operator.

13. **Binary Relation**
Operator: $(\mathsf{bin\text{-}rel} :: \mathsf{C}, \mathsf{formula})$
Definition:

$$\begin{array}{l}\forall\, x : \mathsf{C}\ .\ (\mathsf{bin\text{-}rel} :: \mathsf{C}, \mathsf{formula})(x) \equiv \\ \quad \forall\, w : \mathsf{V}\ .\ w \in x \supset \exists\, u, v : \mathsf{V}\ .\ w = \langle u, v \rangle.\end{array}$$

$$\begin{array}{l}\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!] \uparrow \supset \\ \quad (\mathsf{bin\text{-}rel} :: \mathsf{C}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.\end{array}$$

14. **Univocal**
Operator: $(\mathsf{univocal} :: \mathsf{C}, \mathsf{formula})$
Definition:

$$\begin{array}{l}\forall\, x : \mathsf{C}\ .\ (\mathsf{univocal} :: \mathsf{C}, \mathsf{formula})(x) \equiv \\ \quad \forall\, u, v, v' : \mathsf{V}\ .\ (\langle u, v \rangle \in x \wedge \langle u, v' \rangle \in x) \supset v = v'.\end{array}$$

$$\begin{array}{l}\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!] \uparrow \supset \\ \quad (\mathsf{univocal} :: \mathsf{C}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.\end{array}$$

15. **Function**
Operator: $(\mathsf{fun} :: \mathsf{C}, \mathsf{formula})$
Definition:

$$\forall\, x : \mathsf{C} \;.\; (\mathsf{fun} :: \mathsf{C}, \mathsf{formula})(x) \equiv \\ \quad \mathsf{bin\text{-}rel}(x) \wedge \mathsf{univocal}(x).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\!\uparrow \supset \\ \quad (\mathsf{fun} :: \mathsf{C}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

16. **Domain of a Class**
Operator: $(\mathsf{dom} :: \mathsf{C}, \mathsf{C})$
Definition:

$$\forall\, x : \mathsf{C} \;.\; (\mathsf{dom} :: \mathsf{C}, \mathsf{C})(x) \simeq \\ \quad \iota\, y : \mathsf{C} \;.\; \forall\, u : \mathsf{V} \;.\; u \in y \equiv (\exists\, v : \mathsf{V} \;.\; \langle u, v \rangle \in x).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\!\uparrow \supset \\ \quad (\mathsf{dom} :: \mathsf{C}, \mathsf{C})([\![e]\!])\!\uparrow .$$

17. **Sum Class**
Operator: $(\mathsf{sum} :: \mathsf{C}, \mathsf{C})$
Definition:

$$\forall\, x : \mathsf{C} \;.\; (\mathsf{sum} :: \mathsf{C}, \mathsf{C})(x) \simeq \\ \quad \iota\, y : \mathsf{C} \;.\; \forall\, u : \mathsf{V} \;.\; u \in y \equiv (\exists\, v : \mathsf{V} \;.\; u \in v \wedge v \in x).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\!\uparrow \supset \\ \quad (\mathsf{sum} :: \mathsf{C}, \mathsf{C})([\![e]\!])\!\uparrow .$$

18. **Power Class**
Operator: $(\mathsf{power} :: \mathsf{C}, \mathsf{C})$
Definition:

$$\forall\, x : \mathsf{C} \;.\; (\mathsf{power} :: \mathsf{C}, \mathsf{C})(x) \simeq \\ \quad \iota\, y : \mathsf{C} \;.\; \forall\, u : \mathsf{V} \;.\; u \in y \equiv u \subseteq x.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\!\uparrow \supset \\ \quad (\mathsf{power} :: \mathsf{C}, \mathsf{C})([\![e]\!])\!\uparrow .$$

19. **Type is Class Checker**
Operator: $(\mathsf{type\text{-}is\text{-}class} :: \mathsf{type}, \mathsf{formula})$
Definition:

$$\forall\, e_1 : \mathsf{E}_{\mathrm{ty}} \,.\, (\mathsf{type\text{-}is\text{-}class} :: \mathsf{type}, \mathsf{formula})([\![e_1]\!]) \equiv$$
$$\forall\, e, e' : \mathsf{E}_{\mathrm{sy}} \,.\, (\neg \mathsf{free\text{-}in}(e, e_1) \wedge \neg \mathsf{free\text{-}in}(e', e_1)) \supset$$
$$[\![\lceil \exists\, \lfloor e \rfloor : \mathsf{C} \,.\, \forall\, \lfloor e' \rfloor : \mathsf{C} \,.\, \lfloor e' \rfloor \downarrow \lfloor e_1 \rfloor \equiv \lfloor e' \rfloor \in \lfloor e \rfloor \rceil]\!]_{\mathrm{fo}}.$$

20. **Type is Set Checker**
Operator: $(\mathsf{type\text{-}is\text{-}set} :: \mathsf{type}, \mathsf{formula})$
Definition:

$$\forall\, e_1 : \mathsf{E}_{\mathrm{ty}} \,.\, (\mathsf{type\text{-}is\text{-}set} :: \mathsf{type}, \mathsf{formula})([\![e_1]\!]) \equiv$$
$$\forall\, e, e' : \mathsf{E}_{\mathrm{sy}} \,.\, (\neg \mathsf{free\text{-}in}(e, e_1) \wedge \neg \mathsf{free\text{-}in}(e', e_1)) \supset$$
$$[\![\lceil \exists\, \lfloor e \rfloor : \mathsf{V} \,.\, \forall\, \lfloor e' \rfloor : \mathsf{C} \,.\, \lfloor e' \rfloor \downarrow \lfloor e_1 \rfloor \equiv \lfloor e' \rfloor \in \lfloor e \rfloor \rceil]\!]_{\mathrm{fo}}.$$

21. **Type to Term**
Operator: $(\mathsf{type\text{-}to\text{-}term} :: \mathsf{type}, \mathsf{C})$
Definition:

$$\forall\, e_1 : \mathsf{E}_{\mathrm{ty}}, \forall\, e, e' : \mathsf{E}_{\mathrm{sy}} \,.$$
$$(\neg \mathsf{free\text{-}in}(e, e_1) \wedge \neg \mathsf{free\text{-}in}(e', e_1)) \supset$$
$$[\![\lceil (\mathsf{type\text{-}to\text{-}term} :: \mathsf{type}, \mathsf{C})(\lfloor e_1 \rfloor) \simeq$$
$$\iota\, \lfloor e \rfloor : \mathsf{C} \,.\, \forall\, \lfloor e' \rfloor : \mathsf{C} \,.\, \lfloor e' \rfloor \downarrow \lfloor e_1 \rfloor \equiv \lfloor e' \rfloor \in \lfloor e \rfloor \rceil]\!]_{\mathrm{fo}}.$$

22. **Term to Type**
Operator: $(\mathsf{term\text{-}to\text{-}type} :: \mathsf{C}, \mathsf{type})$
Definition:

$$\forall\, x : \mathsf{C} \,.$$
$$\forall\, y : \mathsf{C} \,.\, y \downarrow (\mathsf{term\text{-}to\text{-}type} :: \mathsf{C}, \mathsf{type})(x) \equiv y \in x.$$
$$\forall\, e : \mathsf{E}_{\mathrm{te}} \,.\, [\![e]\!]\uparrow \supset$$
$$(\mathsf{term\text{-}to\text{-}type} :: \mathsf{C}, \mathsf{type})([\![e]\!]) =_{\mathrm{ty}} \mathsf{C}.$$

23. **Sets Type**
Operator: $(\mathsf{sets} :: \mathsf{type}, \mathsf{type})$
Definition:

$$\forall\, e : \mathsf{E}_{\mathrm{ty}} \,.\, (\mathsf{sets} :: \mathsf{type}, \mathsf{type})([\![e]\!]) =_{\mathrm{ty}}$$
$$\mathsf{term\text{-}to\text{-}type}(\mathsf{power}(\mathsf{type\text{-}to\text{-}term}([\![e]\!]))).$$

6.3 Syntactic Operators

1. **Proper Expression Checker**
   Operator: $(\mathsf{is\text{-}p\text{-}expr} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E} \;.\; (\mathsf{is\text{-}p\text{-}expr} :: \mathsf{E}, \mathsf{formula})(e) \equiv$$
$$e \downarrow \mathsf{E}_{\mathrm{op}} \vee e \downarrow \mathsf{E}_{\mathrm{ty}} \vee e \downarrow \mathsf{E}_{\mathrm{te}} \vee e \downarrow \mathsf{E}_{\mathrm{fo}}.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\uparrow \supset$$
$$(\mathsf{is\text{-}p\text{-}expr} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

2. **First Component Selector for a Proper Expression**
   Operator: $(\mathsf{1st\text{-}comp} :: \mathsf{E}, \mathsf{E})$
   Definition:

$$\forall\, e : \mathsf{E} \;.\; (\mathsf{1st\text{-}comp} :: \mathsf{E}, \mathsf{E})(e) \simeq$$
$$\mathsf{if}(\mathsf{is\text{-}p\text{-}expr}(e), \mathsf{hd}(\mathsf{tl}(e)), \bot_{\mathsf{C}}).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\uparrow \supset$$
$$(\mathsf{1st\text{-}comp} :: \mathsf{E}, \mathsf{E})([\![e]\!])\uparrow .$$

   Note: The second, third, fourth, ... component selectors for proper expressions are defined in a similar way: 2nd-comp, 3rd-comp, 4th-comp, ...

3. **Operator Checker**
   Operator: $(\mathsf{is\text{-}op} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E} \;.\; (\mathsf{is\text{-}op} :: \mathsf{E}, \mathsf{formula})(e) \equiv$$
$$\mathsf{is\text{-}p\text{-}expr}(e) \wedge \mathsf{hd}(e) = \lceil \mathsf{op} \rceil.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \;.\; [\![e]\!]\uparrow \supset$$
$$(\mathsf{is\text{-}op} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

   Note: Checkers for the other 12 proper expression categories are defined in a similar way: is-op-app, is-var, is-type-app, is-dep-fun-type, is-fun-app, is-fun-abs, is-if, is-exist, is-def-des, is-indef-des, is-quote, is-eval.

4. **Disjunction Checker**
   Operator: $(\mathsf{is\text{-}disj} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{is\text{-}disj} :: \mathsf{E}, \mathsf{formula})(e) \equiv$$
$$\quad \mathsf{is\text{-}op\text{-}app}(e) \wedge \mathsf{1st\text{-}comp}(\mathsf{1st\text{-}comp}(e)) = \lceil \mathsf{or} \rceil.$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\!\uparrow \supset$$
$$\quad (\mathsf{is\text{-}disj} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

   Note: Checkers for other kinds of operator applications are defined in a similar way: is-in, is-type-eqn, is-term-eqn, is-formula-eqn, is-neg, is-conj, is-impl, is-defined-in, is-quasi-eqn, is-type-ineq, is-undefined-term, is-empty-set, is-pair, is-ord-pair, is-subclass, is-univ-class, is-union, is-intersec, is-compl, is-hd, is-tl, is-type-prod, is-bin-rel, is-univocal, is-fun, is-dom, is-sum, is-power.

5. **First Argument Selector for an Operator Application**
   Operator: $(\mathsf{1st\text{-}arg} :: \mathsf{E}, \mathsf{E})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{1st\text{-}arg} :: \mathsf{E}, \mathsf{E})(e) \simeq$$
$$\quad \mathsf{if}(\mathsf{is\text{-}op\text{-}app}(e), \mathsf{2nd\text{-}comp}(e), \bot_{\mathsf{C}}).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\!\uparrow \supset$$
$$\quad (\mathsf{1st\text{-}arg} :: \mathsf{E}, \mathsf{E})([\![e]\!])\!\uparrow .$$

   Note: The second, third, fourth, ... argument selectors for operator application are defined in a similar way: 2nd-arg, 3rd-arg, 4th-arg, ...

6. **Variable Binder Checker**
   Operator: $(\mathsf{is\text{-}binder} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{is\text{-}binder} :: \mathsf{E}, \mathsf{formula})(e) \equiv$$
$$\quad \mathsf{is\text{-}dep\text{-}fun\text{-}type}(e) \vee \mathsf{is\text{-}fun\text{-}abs}(e) \vee \mathsf{is\text{-}exist}(e) \vee$$
$$\quad \mathsf{is\text{-}def\text{-}des}(e) \vee \mathsf{is\text{-}indef\text{-}des}(e).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\!\uparrow \supset$$
$$\quad (\mathsf{is\text{-}binder} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

7. **Variable Selector for Variable Binder**
   Operator: $(\mathsf{binder\text{-}var} :: \mathsf{E}, \mathsf{E})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{binder\text{-}var} :: \mathsf{E}, \mathsf{E})(e) \simeq \mathsf{if}(\mathsf{is\text{-}binder}(e), \mathsf{1st\text{-}comp}(e), \bot_{\mathsf{C}}).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\uparrow\ \supset (\mathsf{binder\text{-}var} :: \mathsf{E}, \mathsf{E})([\![e]\!])\uparrow\ .$$

   Note: Selectors for a binder name and a binder body are defined in a similar way: binder-name and binder-body.

8. **Function Redex Checker**
   Operator: $(\mathsf{is\text{-}fun\text{-}redex} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{is\text{-}fun\text{-}redex} :: \mathsf{E}, \mathsf{formula})(e) \equiv \mathsf{is\text{-}fun\text{-}app}(e) \wedge \mathsf{is\text{-}fun\text{-}abs}(\mathsf{1st\text{-}comp}(e)).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\uparrow\ \supset (\mathsf{is\text{-}fun\text{-}redex} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

   Note: A dependent function type redex checker named is-dep-fun-type-redex is defined in a similar way.

9. **Redex Checker**
   Operator: $(\mathsf{is\text{-}redex} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{is\text{-}redex} :: \mathsf{E}, \mathsf{formula})(e) \equiv \mathsf{is\text{-}dep\text{-}fun\text{-}type\text{-}redex}(e) \vee \mathsf{is\text{-}fun\text{-}redex}(e).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\uparrow\ \supset (\mathsf{is\text{-}redex} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

10. **Variable Selector for a Redex**
Operator: $(\mathsf{redex\text{-}var} :: \mathsf{E}, \mathsf{E})$
Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{redex\text{-}var} :: \mathsf{E}, \mathsf{E})(e) \simeq \mathsf{if}(\mathsf{is\text{-}redex}(e), \mathsf{binder\text{-}var}(\mathsf{1st\text{-}comp}(e)), \bot_{\mathsf{C}}).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\uparrow \supset (\mathsf{redex\text{-}var} :: \mathsf{E}, \mathsf{E})([\![e]\!])\uparrow\ .$$

Note: Body and argument selectors for a redex named redex-body and redex-arg are defined in a similar way.

11. **Variable Similarity**
Operator: $(\mathsf{var\text{-}sim} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})$
Definition:

$$\forall\, e_1, e_2 : \mathsf{E}\ .\ (\mathsf{var\text{-}sim} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(e_1, e_2) \equiv \mathsf{is\text{-}var}(e_1) \wedge \mathsf{is\text{-}var}(e_2) \wedge \mathsf{1st\text{-}comp}(e_1) = \mathsf{1st\text{-}comp}(e_2).$$

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{te}}\ .\ ([\![e_1]\!]\uparrow \vee [\![e_2]\!]\uparrow) \supset (\mathsf{var\text{-}sim} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})([\![e_1]\!], [\![e_2]\!]) \equiv \mathsf{F}.$$

Compact notation:

$e_1 \sim e_2$ means $(\mathsf{var\text{-}sim} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(e_1, e_2)$.
$e_1 \not\sim e_2$ means $\neg(e_1 \sim e_2)$.

12. **Eval-Free Checker**
Operator: $(\mathsf{is\text{-}eval\text{-}free} :: \mathsf{E}, \mathsf{formula})$
Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{is\text{-}eval\text{-}free} :: \mathsf{E}, \mathsf{formula})(e) \equiv (e \downarrow \mathsf{E}_{\mathrm{sy}} \wedge e \neq \lceil \mathsf{eval} \rceil) \vee e = [\,] \vee (e \downarrow \mathsf{E} \wedge \mathsf{is\text{-}eval\text{-}free}(\mathsf{hd}(e)) \wedge \mathsf{is\text{-}eval\text{-}free}(\mathsf{tl}(e))).$$

$$\forall\, e : \mathsf{E}_{\mathrm{te}}\ .\ [\![e]\!]\uparrow \supset (\mathsf{is\text{-}eval\text{-}free} :: \mathsf{E}, \mathsf{formula})([\![e]\!]) \equiv \mathsf{F}.$$

6.4 Substitution Operators

We define now the operators related to the substitution of a term for the free occurrences of a variable. Each operator will be defined only for quotations. As a result, the defining sentences will be an infinite set organized into a finite number of traditional sentence schemas.

1. **Good Evaluation Arguments**
   Operator: $(\mathsf{gea} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})$
   Definition:

   $\neg(\mathsf{gea} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil e_1 \rceil, \lceil e_2 \rceil)$

   where e_1 is a non eval-free expression, e_1 is a type and e_2 is not type, e_1 is a term and e_2 is not a type, or e_1 is a formula and e_2 is not $\mathsf{formula}$.

   $(\mathsf{gea} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil \alpha \rceil, \lceil \mathsf{type} \rceil)$

   where α is eval-free.

   $(\mathsf{gea} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil \alpha \rceil)$

   where a is eval-free.

   $(\mathsf{gea} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil A \rceil, \lceil \mathsf{formula} \rceil)$

   where A is eval-free.

2. **Free Variable Occurrence in an Expression**
   Operator: $(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})$
   Definition:

   $\neg(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil e_1 \rceil, \lceil e_2 \rceil)$

   where e_1 is not a symbol or e_2 is not a proper expression.

   $(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil (s :: k_1, \ldots, k_{n+1}) \rceil) \equiv$
   $\mathsf{free\text{-}in}(\lceil x \rceil, \lceil k_1 \rceil) \vee \cdots \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil k_{n+1} \rceil)$

   where $n \geq 0$.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil O(e_1, \ldots, e_n) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil O \rceil) \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil e_1 \rceil) \vee \cdots \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil e_n \rceil)$

$\quad$ where $O(e_1, \ldots, e_n)$ is proper and $n \geq 0$.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil (x : \alpha) \rceil)$.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil (y : \alpha) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil \alpha \rceil)$

$\quad$ where $x \neq y$.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil (\star x : \alpha \ . \ e) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil \alpha \rceil)$

$\quad$ where $(\star x : \alpha \ . \ e)$ is proper and $\star$ is Λ, λ, $\exists$, ι, or ϵ.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil (\star y : \alpha \ . \ e) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil \alpha \rceil) \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil e \rceil)$

$\quad$ where $x \neq y$, $(\star x : \alpha \ . \ e)$ is proper, and $\star$ is Λ, λ, $\exists$, ι, or ϵ.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil \alpha(a) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil \alpha \rceil) \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil a \rceil)$.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil f(a) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil f \rceil) \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil a \rceil)$.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil \mathsf{if}(A, b, c) \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil A \rceil) \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil b \rceil) \vee \mathsf{free\text{-}in}(\lceil x \rceil, \lceil c \rceil)$.

$\neg(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil \lceil e \rceil \rceil)$

$\quad$ where e is any expression.

$(\mathsf{free\text{-}in} :: \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil x \rceil, \lceil [\![a]\!]_k \rceil) \equiv$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil a \rceil) \vee$
$\quad \mathsf{free\text{-}in}(\lceil x \rceil, \lceil k \rceil) \vee$
$\quad (\mathsf{gea}(a, \lceil k \rceil) \wedge \mathsf{free\text{-}in}(\lceil x \rceil, a))$.

Note: For an evaluation (eval, a, k), a variable may be free in the term a, in the kind k when k is a type, and in the expression that the evaluation represents.

3. **Syntactically Closed Proper Expression**
   Operator: $(\mathsf{syn\text{-}closed} :: \mathsf{E}, \mathsf{formula})$
   Definition:

$$\forall\, e : \mathsf{E}\ .\ (\mathsf{syn\text{-}closed} :: \mathsf{E}, \mathsf{formula})(e) \equiv \mathsf{is\text{-}p\text{-}expr}(e) \wedge \forall\, e' : \mathsf{E}_{\mathrm{sy}}\ .\ \neg \mathsf{free\text{-}in}(e', e).$$

4. **Substitution for a Variable in an Expression**
   Operator: $(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})$
   Definition:

$$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil e_1 \rceil, \lceil e_2 \rceil, \lceil e_3 \rceil) = \lceil e_3 \rceil$$

   where e_1 is not a term, e_2 is not a symbol, or e_3 is not a proper expression.

$$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (s :: k_1, \ldots, k_{n+1}) \rceil) = \lceil (s :: \lfloor \widehat{k}_1 \rfloor, \ldots, \lfloor \widehat{k}_{n+1} \rfloor) \rceil$$

   where $n \geq 0$ and $\widehat{k}_i = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil k_i \rceil)$ for all i with $1 \leq i \leq n+1$.

$$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil O(e_1, \ldots, e_n) \rceil) = \lceil \lfloor \widehat{O} \rfloor (\lfloor \widehat{e}_1 \rfloor, \ldots, \lfloor \widehat{e}_n \rfloor) \rceil$$

   where $O(e_1, \ldots, e_n)$ is proper, $n \geq 0$,
   $\widehat{O} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil O \rceil)$, and
   $\widehat{e}_i = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil e_i \rceil)$ for all i with $1 \leq i \leq n$.

$$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (x : \alpha) \rceil) = \mathsf{if}(a \downarrow [\![\widehat{\alpha}]\!]_{\mathrm{ty}}, \lceil a \rceil, \lceil \bot_{\mathsf{C}} \rceil)$$

   where $\widehat{\alpha} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$.

$$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (y : \alpha) \rceil) = \lceil (y : \lfloor \widehat{\alpha} \rfloor) \rceil$$

   where $x \neq y$ and $\widehat{\alpha} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (\star x : \alpha \,.\, e) \rceil) =$

$\lceil \star x : \lfloor \widehat{\alpha} \rfloor \,.\, e \rceil$

where $(\star x : \alpha \,.\, e)$ is proper; $\star$ is Λ, λ, $\exists$, ι, or ϵ; and $\widehat{\alpha} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (\star y : \alpha \,.\, e) \rceil) =$

$\lceil \star y : \lfloor \widehat{\alpha} \rfloor \,.\, \lfloor \widehat{e} \rfloor \rceil$

where $x \neq y$; $(\star y : \alpha \,.\, e)$ is proper; $\star$ is Λ, λ, $\exists$, ι, or ϵ; $\widehat{\alpha} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$; and $\widehat{e} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil e \rceil)$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \alpha(b) \rceil) =$

$\lceil \lfloor \widehat{\alpha} \rfloor (\lfloor \widehat{b} \rfloor) \rceil$

where $\widehat{\alpha} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$ and $\widehat{b} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil f(b) \rceil) =$

$\lceil \lfloor \widehat{f} \rfloor (\lfloor \widehat{b} \rfloor) \rceil$

where $\widehat{f} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil f \rceil)$ and $\widehat{b} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \mathsf{if}(A, b, c) \rceil) =$

$\lceil \mathsf{if}(\lfloor \widehat{A} \rfloor, \lfloor \widehat{b} \rfloor, \lfloor \widehat{c} \rfloor) \rceil$

where $\widehat{A} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil A \rceil)$, $\widehat{b} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$, and $\widehat{c} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil c \rceil)$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \lceil e \rceil \rceil) =$

$\lceil \lceil e \rceil \rceil$

where e is any expression.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil [\![b]\!]_{\mathrm{ty}} \rceil) =$

$\mathsf{if}(\mathsf{gea}([\![\widehat{b}]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil), \widehat{b}', \lceil \mathsf{C} \rceil)$

where $\widehat{b} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$ and $\widehat{b}' = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, [\![\widehat{b}]\!]_{\mathrm{te}})$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil [\![b]\!]_\alpha \rceil) =$
$\quad \mathsf{if}(\mathsf{gea}([\![\widehat{b}]\!]_{\mathrm{te}}, \widehat{\alpha}) \wedge [\![[\![\widehat{b}]\!]_{\mathrm{te}}]\!]_{\mathrm{te}} \downarrow [\![\widehat{\alpha}]\!]_{\mathrm{ty}}, \widehat{b}', \lceil \bot_{\mathsf{C}} \rceil)$

$\quad$ where $\widehat{b} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$, $\widehat{\alpha} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$, and $\widehat{b}' = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, [\![\widehat{b}]\!]_{\mathrm{te}})$.

$(\mathsf{sub} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil [\![b]\!]_{\mathrm{fo}} \rceil) =$
$\quad \mathsf{if}(\mathsf{gea}([\![\widehat{b}]\!]_{\mathrm{te}}, \lceil \mathsf{formula} \rceil), \widehat{b}', \lceil \mathsf{F} \rceil)$

$\quad$ where $\widehat{b} = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$ and $\widehat{b}' = \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, [\![\widehat{b}]\!]_{\mathrm{te}})$.

Note: For an evaluation (eval, a, k), a substitution for a variable is performed on a to obtain a' and on k when k is a type to obtain k' and then on the expression represented by the new evaluation resulting from these substitutions.

5. **Free for a Variable in an Expression**
Operator: $(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})$
Definition:

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil e_1 \rceil, \lceil e_2 \rceil, \lceil e_3 \rceil)$

$\quad$ where e_1 is not a term, e_2 is not a symbol, or e_3 is not a proper expression.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (s :: k_1, \ldots, k_{n+1}) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil k_1 \rceil) \wedge \cdots \wedge \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil k_{n+1} \rceil)$

$\quad$ where $n \geq 0$.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil O(e_1, \ldots, e_n) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil O \rceil) \wedge$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil e_1 \rceil) \wedge \cdots \wedge \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil e_n \rceil)$

$\quad$ where $O(e_1, \ldots, e_n)$ is proper and $n \geq 0$.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (y : \alpha) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil (\star x : \alpha \,.\, e) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$

$\quad$ where $(\star x : \alpha \,.\, e)$ is proper and $\star$ is Λ, λ, $\exists$, ι, or ϵ.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \star y : \alpha \,.\, e \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil) \wedge$
$\quad (\neg \mathsf{free\text{-}in}(\lceil x \rceil, \lceil e \rceil) \vee$
$\quad\ (\neg \mathsf{free\text{-}in}(\lceil y \rceil, \lceil a \rceil) \wedge \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil e \rceil)))$

$\quad$ where $x \neq y$, $(\star y : \alpha \,.\, e)$ is proper, and $\star$ is Λ, λ, $\exists$, ι, or ϵ.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \alpha(b) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil) \wedge \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil).$

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil f(b) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil f \rceil) \wedge \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil).$

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \mathsf{if}(A, b, c) \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil A \rceil) \wedge$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil) \wedge$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil c \rceil).$

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil \lceil e \rceil \rceil)$

$\quad$ where e is any expression.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil [\![b]\!]_{\mathrm{ty}} \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil) \wedge$
$\quad (\mathsf{gea}([\![\widehat{b}]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil) \supset \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, [\![\widehat{b}]\!]_{\mathrm{te}})).$

$\quad$ where $\widehat{b}$ is $\mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$.

$(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil [\![b]\!]_{\alpha} \rceil) \equiv$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil) \wedge$
$\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil) \wedge$
$\quad ((\mathsf{gea}([\![\widehat{b}]\!]_{\mathrm{te}}, \widehat{\alpha}) \wedge [\![[\![\widehat{b}]\!]_{\mathrm{te}}]\!]_{\mathrm{te}} \downarrow [\![\widehat{\alpha}]\!]_{\mathrm{ty}}) \supset \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, [\![\widehat{b}]\!]_{\mathrm{te}})).$

$\quad$ where $\widehat{b}$ is $\mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil)$ and $\widehat{\alpha}$ is $\mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil \alpha \rceil)$.

$$
\begin{array}{l}
(\mathsf{free\text{-}for} :: \mathsf{E}, \mathsf{E}, \mathsf{E}, \mathsf{formula})(\lceil a \rceil, \lceil x \rceil, \lceil [\![b]\!]_{\mathrm{fo}} \rceil) \equiv \\
\quad \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil) \wedge \\
\quad (\mathsf{gea}([\![\widehat{b}]\!]_{\mathrm{te}}, \lceil \mathsf{formula} \rceil) \supset \mathsf{free\text{-}for}(\lceil a \rceil, \lceil x \rceil, [\![\widehat{b}]\!]_{\mathrm{te}})). \\
\quad \text{where } \widehat{b} \text{ is } \mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil b \rceil).
\end{array}
$$

The next several lemmas show that the operators defined in this section have their intended meaning.

Lemma 6.1 (Good Evaluation Arguments) *Let $M = (S, V)$ be a standard model for L. For all assignments φ into S, types α, terms a of type α, terms b, and formulas A, then:*

1. $V_\varphi(\mathsf{gea}(\lceil \alpha \rceil, \lceil \mathsf{type} \rceil)) = \mathrm{T}$ *implies* $V_\varphi([\![\lceil \alpha \rceil]\!]_{\mathrm{ty}} =_{\mathrm{ty}} \alpha) = \mathrm{T}$.
2. $V_\varphi(\mathsf{gea}(b, \lceil \mathsf{type} \rceil)) = \mathrm{F}$ *implies* $V_\varphi([\![b]\!]_{\mathrm{ty}} =_{\mathrm{ty}} \mathsf{C}) = \mathrm{T}$.
3. $V_\varphi(\mathsf{gea}(\lceil a \rceil, \lceil \alpha \rceil)) = \mathrm{T}$ *implies* $V_\varphi([\![\lceil a \rceil]\!]_\alpha = \mathsf{if}(a \downarrow \alpha, a, \bot_{\mathsf{C}})) = \mathrm{T}$.
4. $V_\varphi(\mathsf{gea}(b, \lceil \alpha \rceil)) = \mathrm{F}$ *implies* $V_\varphi([\![b]\!]_\alpha \uparrow) = \mathrm{T}$.
5. $V_\varphi(\mathsf{gea}(\lceil A \rceil, \lceil \mathsf{formula} \rceil)) = \mathrm{T}$ *implies* $V_\varphi([\![\lceil A \rceil]\!]_{\mathrm{fo}} \equiv A) = \mathrm{T}$.
6. $V_\varphi(\mathsf{gea}(b, \lceil \mathsf{formula} \rceil)) = \mathrm{F}$ *implies* $V_\varphi([\![b]\!]_{\mathrm{fo}} \equiv \mathsf{F}) = \mathrm{T}$.

Proof Follows from the definition of the standard valuation V applied to evaluations. □

Let $k[e]$ be type if e is a type, C if e is a term, and $\mathsf{formula}$ if e is a formula.

Lemma 6.2 (Free Variable) *Let $M = (S, V)$ be a standard model for L. For all assignments φ into S, symbols x, and proper expressions e, if*

$$V_\varphi(\neg \mathsf{free\text{-}in}(\lceil x \rceil, \lceil e \rceil)) = \mathrm{T},$$

then

$$V_\varphi(e) = V_{\varphi[x \mapsto d]}(e) \text{ for all } d \in D_{\mathrm{c}}.$$

Proof Fix an assignment φ into S, a symbol x, and a proper expression e. We will show that, if

$$V_\varphi(\neg\textsf{free-in}(\lceil x\rceil, \lceil e\rceil)) = \mathrm{T} \quad [\text{designated } H(\lceil e\rceil)],$$

then

$$V_\varphi(e) = V_{\varphi[x\mapsto d]}(e) \text{ for all } d \in D_\mathrm{c} \quad [\text{designated } C(\lceil e\rceil)].$$

Our proof is by induction on the complexity of e. There are 11 cases corresponding to the 11 sentence schemas used to define $\textsf{free-in}(\lceil e_1\rceil, \lceil e_2\rceil))$ when e_1 is a symbol and e_2 is a proper expression.

Case 1: $e = (s :: k_1, \ldots, k_{n+1})$. $C(\lceil e\rceil)$ follows immediately since $V_\varphi(e)$ does not depend on φ.

Case 2: $e = O(e_1, \ldots, e_n)$ and $O = (s :: k_1, \ldots, k_{n+1})$. By the definition of $\textsf{free-in}$, $H(\lceil e\rceil)$ implies (1) $H(\lceil k_i\rceil)$ for all i with $1 \le i \le n+1$ and $\mathbf{type}[k_i]$ and (2) $H(\lceil e_i\rceil)$ for all i with $1 \le i \le n$. By the induction hypothesis, this implies (1) $C(\lceil k_i\rceil)$ for all i with $1 \le i \le n+1$ and $\mathbf{type}[k_i]$ and (2) $C(\lceil e_i\rceil)$ for all i with $1 \le i \le n$. $C(\lceil O\rceil)$ holds by Case 1. Therefore, $C(\lceil e\rceil)$ holds since $V_\varphi(e)$ is defined in terms of $V_\varphi(O)$, $V_\varphi(k_i)$ for all i with $1 \le i \le n+1$ and $\mathbf{type}[k_i]$, and $V_\varphi(e_i)$ for all i with $1 \le i \le n$.

Case 3: $e = (x : \alpha)$. By the definition of $\textsf{free-in}$, $H(\lceil e\rceil)$ is false. Therefore, the lemma is true for this case.

Case 4: $e = (y : \alpha)$ where $x \neq y$. By the definition of $\textsf{free-in}$, $H(\lceil e\rceil)$ implies $H(\lceil \alpha\rceil)$. By the induction hypothesis, this implies $C(\lceil \alpha\rceil)$. Therefore, $C(\lceil e\rceil)$ holds since $V_\varphi(e)$ is defined in terms of $\varphi(y)$ and $V_\varphi(\alpha)$.

Case 5: $e = (\star x : \alpha \,.\, e')$ and $\star$ is Λ, λ, $\exists$, ι, or ϵ. By the definition of $\textsf{free-in}$, $H(\lceil e\rceil)$ implies $H(\lceil \alpha\rceil)$. By the induction hypothesis, this implies $C(\lceil \alpha\rceil)$. Then

$$V_{\varphi[x\mapsto d']}(e') = V_{\varphi[x\mapsto d][x\mapsto d']}(e')$$

for all $d, d' \in D_\mathrm{c}$. Therefore, $C(\lceil e\rceil)$ holds since $V_\varphi(e)$ is defined in terms of $V_\varphi(\alpha)$ and $V_{\varphi[x\mapsto d']}(e')$ where $d' \in D_\mathrm{c}$.

Case 6: $e = (\star y : \alpha \ . \ e')$, $x \neq y$, and $\star$ is Λ, λ, $\exists$, ι, or ϵ. By the definition of $\mathsf{free\text{-}in}$, $H(\lceil e \rceil)$ implies $H(\lceil \alpha \rceil)$ and $H(\lceil e' \rceil)$. By the induction hypothesis, this implies $C(\lceil \alpha \rceil)$ and $C(\lceil e' \rceil)$. Then

$$V_{\varphi[y \mapsto d']}(e') = V_{\varphi[y \mapsto d'][x \mapsto d]}(e') = V_{\varphi[x \mapsto d][y \mapsto d']}(e')$$

for all $d, d' \in D_{\mathrm{c}}$. Therefore, $C(\lceil e \rceil)$ holds since $V_\varphi(e)$ is defined in terms of $V_\varphi(\alpha)$ and $V_{\varphi[y \mapsto d']}(e')$ where $d' \in D_{\mathrm{c}}$.

Case 7: $e = \alpha(a)$. Similar to Case 4.

Case 8: $e = f(a)$. Similar to Case 4.

Case 9: $e = \mathsf{if}(A, b, c)$. Similar to Case 4.

Case 10: $e = \lceil e' \rceil$. $C(\lceil e \rceil)$ follows immediately since $V_\varphi(e)$ does not depend on φ.

Case 11: $e = \lceil [\![a]\!]_k \rceil$. By the definition of $\mathsf{free\text{-}in}$, $H(\lceil e \rceil)$ implies (1) $H(\lceil a \rceil)$, (2) $H(\lceil k \rceil)$ when $\mathbf{type}[k]$, and (3) if $V_\varphi(\mathsf{gea}(a, \lceil k \rceil)) = \mathrm{T}$, then $V_\varphi(\neg \mathsf{free\text{-}in}(\lceil x \rceil, a)) = \mathrm{T}$. By the induction hypothesis, (1) and (2) imply $C(\lceil a \rceil)$ and $C(\lceil k \rceil)$ when $\mathbf{type}[k]$. If $V_\varphi(\mathsf{gea}(a, \lceil k \rceil)) = \mathrm{T}$, (3) implies $H(\lceil e' \rceil)$ for $e' = H^{-1}(V_\varphi(a))$, and so by the induction hypothesis, $C(\lceil e' \rceil)$ holds. Therefore, $C(\lceil e \rceil)$ holds since $V_\varphi(e)$ is defined in terms of $V_\varphi(a)$, $V_\varphi(k)$ when $\mathbf{type}[k]$, and $V_\varphi(e')$. If $V_\varphi(\mathsf{gea}(a, \lceil k \rceil)) = \mathrm{F}$, $C(\lceil e \rceil)$ follows immediately since, by Lemma 6.1, $V_\varphi(e)$ does not depend on φ.

□

An expression e is *semi-eval-free* if each quotation in e, that is not in the second argument of an evaluation, is eval-free.

Lemma 6.3 (Substitution A) *Let $M = (S, V)$ be a standard model for L. For all assignments φ into S, terms a, symbols x, and proper expressions e the following statements hold:*

1. *If e is an operator, type, term, or formula, then $H^{-1}(V_\varphi(\mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil e \rceil)))$ is an operator similar to e, type, term, or formula, respectively.*

2. *If a is eval-free and e is semi-eval-free, then $H^{-1}(V_\varphi(\mathsf{sub}(\lceil a \rceil, \lceil x \rceil, \lceil e \rceil)))$ is eval-free.*

3. *If e is eval-free and $V_\varphi(\neg\mathsf{free\text{-}in}(\lceil x\rceil,\lceil e\rceil)) = \mathrm{T}$, then*

$$V_\varphi(\mathsf{sub}(\lceil a\rceil,\lceil x\rceil,\lceil e\rceil) = \lceil e\rceil) = \mathrm{T}.$$

4. *If e is a type, term, or formula, e is semi-eval-free, and $V_\varphi(\neg\mathsf{free\text{-}in}(\lceil x\rceil,\lceil e\rceil)) = \mathrm{T}$, then*

   a. *$H^{-1}(V_\varphi(\mathsf{sub}(\lceil a\rceil,\lceil x\rceil,\lceil e\rceil)))$ is eval-free and*
   b. *$V_\varphi(\llbracket\mathsf{sub}(\lceil a\rceil,\lceil x\rceil,\lceil e\rceil)\rrbracket_{k[e]}) = V_\varphi(e)$.*

Proof Fix an assignment φ into S and a term a, a symbol x, and a proper expression e that is an operator, type, term, or formula. Let $S(\lceil e'\rceil)$ mean $\mathsf{sub}(\lceil a\rceil,\lceil x\rceil,\lceil e'\rceil)$ and $E(\lceil e'\rceil)$ mean $H^{-1}(V_\varphi(S(\lceil e'\rceil)))$ for any expression e'.

Part 1 We must show that, if e is an operator, type, term, or formula, then $E(\lceil e\rceil)$ is an operator similar to e, type, term, or formula, respectively. Our proof is by induction on the complexity of e. There are 13 cases corresponding to the 13 sentence schemas used to define $\mathsf{sub}(\lceil e_1\rceil,\lceil e_2\rceil,\lceil e_3\rceil)$ when e_1 is a term, e_2 is a symbol, and e_3 is a proper expression.

Case 1: $e = (s :: k_1,\ldots,k_{n+1})$. By the first sentence schema of the definition of sub, $V_\varphi(S(\lceil\mathsf{type}\rceil)) = V_\varphi(\lceil\mathsf{type}\rceil)$ and $V_\varphi(S(\lceil\mathsf{formula}\rceil)) = V_\varphi(\lceil\mathsf{formula}\rceil)$, and so $E(\lceil\mathsf{type}\rceil) = \mathsf{type}$ and $E(\lceil\mathsf{formula}\rceil) = \mathsf{formula}$. By the second sentence schema of the definition of sub,

$$\begin{aligned}
& E(\lceil e\rceil)\\
= \;& H^{-1}(V_\varphi(S(\lceil e\rceil)))\\
= \;& H^{-1}(V_\varphi(\lceil(s :: \lfloor S(\lceil k_1\rceil)\rfloor,\ldots,\lfloor S(\lceil k_{n+1}\rceil)\rfloor)\rceil))\\
= \;& H^{-1}(V_\varphi(\lceil(s :: E(\lceil k_1\rceil),\ldots,E(\lceil k_{n+1}\rceil))\rceil))\\
= \;& (s :: E(\lceil k_1\rceil),\ldots,E(\lceil k_{n+1}\rceil)).
\end{aligned}$$

By the induction hypothesis, for all i with $1 \le i \le n+1$ and $\mathbf{type}[k_i]$, $E(\lceil k_i\rceil)$ is a type. Therefore, if e is an operator, then $E(\lceil e\rceil)$ is an operator similar to e.

Case 2: $e = O(e_1,\ldots,e_n)$ and $O = (s :: k_1,\ldots,k_{n+1})$. By the third sentence schema of the definition of sub,

$$\begin{aligned}
& E(\lceil e\rceil)\\
= \;& H^{-1}(V_\varphi(S(\lceil e\rceil)))
\end{aligned}$$

$$
\begin{aligned}
&= H^{-1}(V_\varphi(\ulcorner\lfloor S(\ulcorner O\urcorner)\rfloor(\lfloor S(\ulcorner e_1\urcorner)\rfloor,\ldots,\lfloor S(\ulcorner e_n\urcorner)\rfloor)\urcorner))\\
&= H^{-1}(V_\varphi(\ulcorner E(\ulcorner O\urcorner)(E(\ulcorner e_1\urcorner),\ldots,E(\ulcorner e_n\urcorner))\urcorner))\\
&= E(\ulcorner O\urcorner)(E(\ulcorner e_1\urcorner),\ldots,E(\ulcorner e_n\urcorner)).
\end{aligned}
$$

By the induction hypothesis, (1) $E(\ulcorner O\urcorner)$ is an operator similar to O and (2) for all i with $1 \leq i \leq n$, if e_i is a type, term, or formula, then $E(\ulcorner e_i\urcorner)$ is a type, term, or formula, respectively. Therefore, if e is a type, term, or formula, then $E(\ulcorner e\urcorner)$ is a type, term, or a formula, respectively.

Cases 3–13. Similar to Case 2.

Part 2 Assume a is eval-free and e is semi-eval-free. We must show $E(\ulcorner e\urcorner)$ is eval-free. Our proof is by induction on the complexity of e. There are 13 cases as for Part 1.

Cases 1–2, 4–9. An eval symbol in $E(\ulcorner e\urcorner)$ can be introduced explicitly in $S(e)$ and implicitly in $S(e)$ via a subcomponent $S(e')$ of $S(e)$ where e' is a component of e. In these cases, no eval symbols are explicitly introduced in $S(e)$ by the definition of sub, and no eval symbols are implicitly introduced in $S(e)$ by the induction hypothesis. Therefore, $E(\ulcorner e\urcorner)$ is eval-free.

Case 3. $e = (x : \alpha)$. $E(\ulcorner e\urcorner)$ is either a or $\bot_{\mathsf{C}}$. Therefore, $E(\ulcorner e\urcorner)$ is eval-free since a is eval-free by assumption and $\bot_{\mathsf{C}}$ is obviously eval-free.

Case 10. $e = \ulcorner e'\urcorner$. $E(\ulcorner e\urcorner) = e$. Therefore, $E(\ulcorner e\urcorner)$ is eval-free since e is semi-eval-free by assumption and hence eval-free.

Cases 11–13. $e = [\![b]\!]_k$ and either (1) $H^{-1}(V_\varphi([\![S(\ulcorner b\urcorner)]\!]_{\mathrm{te}}))$ is eval-free and $E(\ulcorner e\urcorner)$ is $E(S([\![S(\ulcorner b\urcorner)]\!]_{\mathrm{te}})$ or (2) $E(\ulcorner e\urcorner)$ is T, $\bot_{\mathsf{C}}$, or F. If (1), $E(S([\![S(\ulcorner b\urcorner)]\!]_{\mathrm{te}})$ is eval-free by the induction hypothesis, and if (2), T, $\bot_{\mathsf{C}}$, and F are obviously eval-free. Therefore, $E(\ulcorner e\urcorner)$ is eval-free.

Part 3 Assume e is an eval-free type, term, or formula and

$$V_\varphi(\neg\mathsf{free\text{-}in}(\ulcorner x\urcorner,\ulcorner e\urcorner)) = \mathrm{T} \quad [\text{designated } H(\ulcorner e\urcorner)].$$

We must show that

$$V_\varphi(S(\ulcorner e\urcorner) = \ulcorner e\urcorner) = \mathrm{T} \quad [\text{designated } C(\ulcorner e\urcorner)].$$

Our proof is by induction on the complexity of e. There are 13 cases as for Parts 1 and 2.

Cases 1. $e = (s :: k_1, \ldots, k_{n+1})$. $H(\ulcorner e \urcorner)$ implies $H(\ulcorner k_i \urcorner)$ for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$. From these hypotheses, $C(\ulcorner k_i \urcorner)$ follows for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$ by the induction hypothesis. $C(\ulcorner k_i \urcorner)$ also holds for all i with $1 \leq i \leq n+1$ and $k_i = \mathsf{type}$ or $k_i = \mathsf{formula}$. From these conclusions, $C(\ulcorner e \urcorner)$ follows by the definition of sub.

Cases 2, 4–9. Similar to Case 1.

Case 3. $e = (x : \alpha)$. The hypothesis $H(\ulcorner e \urcorner)$ is false in this case.

Case 10. $e = \ulcorner e' \urcorner$. $C(\ulcorner e \urcorner)$ is always true in this case by the definition of sub.

Cases 11–13. $e = [\![b]\!]_k$. The hypothesis that e is eval-free is false is these cases.

Part 4 Assume e is a type, term, or formula, e is semi-eval-free, and

$V_\varphi(\neg \mathsf{free\text{-}in}(\ulcorner x \urcorner, \ulcorner e \urcorner)) = \mathrm{T}$ [designated $H(\ulcorner e \urcorner)$].

We must show that:

1. $H^{-1}(V_\varphi(S(\ulcorner e \urcorner)))$ is eval-free [designated $C_1(\ulcorner e \urcorner)$].
2. $V_\varphi([\![S(\ulcorner e \urcorner)]\!]_{k[e]}) = V_\varphi(e)$ [designated $C_2(\ulcorner e \urcorner)$].

Our proof is by induction on the complexity of e. There are 12 cases corresponding to the 13 sentence schemas used to define $\mathsf{sub}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner, \ulcorner e_3 \urcorner)$ when e_1 is a term, e_2 is a symbol, and e_3 is a type, term, or formula.

Case 1: $e = O(e_1, \ldots, e_n)$ and $O = (s :: k_1, \ldots, k_{n+1})$. By the definition of $\mathsf{free\text{-}in}$, $H(\ulcorner e \urcorner)$ implies (1) $H(\ulcorner k_i \urcorner)$ for all i with $1 \leq i \leq n+1$ and (2) $H(\ulcorner e_i \urcorner)$ for all i with $1 \leq i \leq n$. By the induction hypothesis, this implies (1) $C_1(\ulcorner k_i \urcorner)$ and $C_2(\ulcorner k_i \urcorner)$ for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$ and (2) $C_1(\ulcorner e_i \urcorner)$ and $C_2(\ulcorner e_i \urcorner)$ for all i with $1 \leq i \leq n$. Then

$$\begin{aligned}
& V_\varphi([\![S(\ulcorner e \urcorner)]\!]_{k[e]}) \\
= \; & V_\varphi([\![S(\ulcorner O(e_1, \ldots, e_n) \urcorner)]\!]_{k[e]}) \\
= \; & V_\varphi([\![\ulcorner \lfloor S(\ulcorner O \urcorner) \rfloor (\lfloor S(\ulcorner e_1 \urcorner) \rfloor, \ldots, \lfloor S(\ulcorner e_n \urcorner) \rfloor) \urcorner]\!]_{k[e]}) \\
= \; & V_\varphi(\overline{S(\ulcorner O \urcorner)}([\![S(\ulcorner e_1 \urcorner)]\!]_{k[e_1]}, \ldots, [\![S(\ulcorner e_n \urcorner)]\!]_{k[e_n]})) \\
= \; & V_\varphi(O(e_1, \ldots, e_n)) \\
= \; & V_\varphi(e)
\end{aligned}$$

where

$$\overline{S(\lceil O \rceil)} = (s :: \overline{S(k_1)}, \ldots, \overline{S(k_{n+1})})$$

and

$$\overline{S(k_i)} = \begin{cases} [\![S(\lceil k_i \rceil)]\!]_{\mathrm{ty}} & \text{if } \mathbf{type}[k_i] \\ k_i & \text{if } k_i = \mathsf{type} \text{ or } k_i = \mathsf{formula}. \end{cases}$$

The third line is by the definition of sub; the fourth is by Lemma 6.1, $C_1(\lceil k_i \rceil)$ for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$, and $C_1(\lceil e_i \rceil)$ for all i with $1 \leq i \leq n$; and the fifth holds since (1) $C_2(\lceil k_i \rceil)$ holds for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$ and (2) $C_2(\lceil e_i \rceil)$ holds for all i with $1 \leq i \leq n$. Therefore, $C_1(\lceil e \rceil)$ and $C_2(\lceil e \rceil)$ hold.

Case 2: $e = (x : \alpha)$. The hypothesis $H(\lceil e \rceil)$ is false in this case.

Case 3: $e = (y : \alpha)$ where $x \neq y$. By the definition of free-in $H(\lceil e \rceil)$ implies $H(\lceil \alpha \rceil)$. By the induction hypothesis, this implies $C_1(\lceil \alpha \rceil)$ and $C_2(\lceil \alpha \rceil)$. Then

$$\begin{aligned} & V_\varphi([\![S(\lceil e \rceil)]\!]_{k[e]}) \\ = \; & V_\varphi([\![S(\lceil (y : \alpha) \rceil)]\!]_{\mathrm{te}}) \\ = \; & V_\varphi([\![\lceil (y : \lfloor S(\alpha) \rfloor) \rceil]\!]_{\mathrm{te}}) \\ = \; & V_\varphi((y : [\![S(\lceil \alpha \rceil)]\!]_{\mathrm{ty}})) \\ = \; & V_\varphi((y : \alpha)) \\ = \; & V_\varphi(e). \end{aligned}$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and $C_1(\lceil \alpha \rceil)$; and the fifth is by $C_2(\lceil \alpha \rceil)$. Therefore, $C_1(\lceil e \rceil)$ and $C_1(\lceil e \rceil)$ hold.

Case 4–8: Similar to Case 3.

Case 9. $e = \lceil e' \rceil$. $C_1(\lceil e \rceil)$ and $C_2(\lceil e \rceil)$ are always true by the definition of sub and the assumption that e is semi-eval-free.

Case 10: $e = [\![b]\!]_{\mathrm{ty}}$. Let $V_\varphi(\mathsf{gea}(c, d)) = \mathrm{T}$ be designated by $G(c, d)$. By the definition of free-in, $H(\lceil e \rceil)$ implies (1) $H(\lceil b \rceil)$ and

(2) $G(b, \lceil \mathsf{type} \rceil)$ implies $H(b)$. By the induction hypothesis, this implies (1) $C_2(\lceil b \rceil)$ and (2) $G(b, \lceil \mathsf{type} \rceil)$ implies $C_1(b)$ and $C_2(b)$.

$$
\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi([\![S(\lceil [\![b]\!]_{\mathrm{ty}} \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil), S([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}), \lceil \mathsf{C} \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}(b, \lceil \mathsf{type} \rceil), S(b), \lceil \mathsf{C} \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}(b, \lceil \mathsf{type} \rceil), [\![S(b)]\!]_{\mathrm{ty}}, \mathsf{C})) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}(b, \lceil \mathsf{type} \rceil), [\![b]\!]_{\mathrm{ty}}, \mathsf{C})) \\
= \; & V_\varphi([\![b]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi(e)
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by $C_2(\lceil b \rceil)$; the fifth by Lemma 6.1 and $G(b, \lceil \mathsf{type} \rceil)$ implies $C_1(b)$; the sixth is by $G(b, \mathsf{type})$ implies $C_2(b)$; and the seventh is by the definition of the standard valuation applied to evaluations. Therefore, $C_1(\lceil e \rceil)$ and $C_2(\lceil e \rceil)$ hold.

Case 11: $e = [\![b]\!]_\alpha$. Let $V_\varphi(\mathsf{gea}(c, d) \wedge [\![c]\!]_{\mathrm{te}} \downarrow [\![d]\!]_{\mathrm{ty}}) = \mathrm{T}$ be designated by $G(c, d)$. By the definition of $\mathsf{free\text{-}in}$, $H(\lceil e \rceil)$ implies (1) $H(\lceil b \rceil)$, (2) $H(\lceil \alpha \rceil)$, and (3) $G(b, \lceil \alpha \rceil)$ implies $H(b)$. By the induction hypothesis, this implies (1) $C_2(\lceil b \rceil)$, (2) $C_2(\lceil \alpha \rceil)$, and (3) $G(b, \lceil \alpha \rceil)$ implies $C_1(b)$ and $C_2(b)$.

$$
\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_\alpha) \\
= \; & V_\varphi([\![S(\lceil [\![b]\!]_\alpha \rceil)]\!]_\alpha) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, S(\lceil \alpha \rceil)) \wedge [\![[\![S(\lceil b \rceil)]\!]_{\mathrm{te}}]\!]_{\mathrm{te}} \downarrow [\![S(\lceil \alpha \rceil)]\!]_{\mathrm{ty}}, \\
& \quad S([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}), \lceil \bot_{\mathsf{C}} \rceil)]\!]_\alpha) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}(b, \lceil \alpha \rceil) \wedge [\![b]\!]_{\mathrm{te}} \downarrow \alpha, S(b), \lceil \bot_{\mathsf{C}} \rceil)]\!]_\alpha) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}(b, \lceil \alpha \rceil) \wedge [\![b]\!]_{\mathrm{te}} \downarrow \alpha, [\![S(b)]\!]_\alpha, \bot_{\mathsf{C}})) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}(b, \lceil \alpha \rceil) \wedge [\![b]\!]_{\mathrm{te}} \downarrow \alpha, [\![b]\!]_\alpha, \bot_{\mathsf{C}})) \\
= \; & V_\varphi([\![b]\!]_\alpha) \\
= \; & V_\varphi(e)
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by $C_2(\lceil b \rceil)$ and $C_2(\lceil \alpha \rceil)$; the fifth by Lemma 6.1 and $G(b, \lceil \alpha \rceil)$ implies $C_1(b)$; the sixth is by $G(b, \lceil \alpha \rceil)$ implies $C_2(b)$; and the seventh is by the definition of

the standard valuation applied to evaluations. Therefore, $C_1(\ulcorner e \urcorner)$ and $C_2(\ulcorner e \urcorner)$ hold.

Case 12: $e = [\![b]\!]_{\mathrm{fo}}$. Let $V_\varphi(\mathsf{gea}(c,d)) = \mathrm{T}$ be designated by $G(c,d)$. By the definition of $\mathsf{free\text{-}in}$, $H(\ulcorner e \urcorner)$ implies (1) $H(\ulcorner b \urcorner)$ and (2) $G(b, \ulcorner \mathsf{formula} \urcorner)$ implies $H(b)$. By the induction hypothesis, this implies (1) $C_2(\ulcorner b \urcorner)$ and (2) $G(b, \ulcorner \mathsf{formula} \urcorner)$ implies $C_1(b)$ and $C_2(b)$.

$$
\begin{aligned}
& V_\varphi([\![S(\ulcorner e \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi([\![S(\ulcorner [\![b]\!]_{\mathrm{fo}} \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner), S([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}), \ulcorner \mathsf{F} \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}(b, \ulcorner \mathsf{formula} \urcorner), S(b), \ulcorner \mathsf{F} \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}(b, \ulcorner \mathsf{formula} \urcorner), [\![S(b)]\!]_{\mathrm{fo}}, \mathsf{F})) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}(b, \ulcorner \mathsf{formula} \urcorner), [\![b]\!]_{\mathrm{fo}}, \mathsf{F})) \\
= \; & V_\varphi([\![b]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi(e)
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by $C_2(\ulcorner b \urcorner)$; the fifth by Lemma 6.1 and $G(b, \ulcorner \mathsf{formula} \urcorner)$ implies $C_1(b)$; the sixth is by $G(b, \mathsf{formula})$ implies $C_2(b)$; and the seventh is by the definition of the standard valuation applied to evaluations. Therefore, $C_1(\ulcorner e \urcorner)$ and $C_2(\ulcorner e \urcorner)$ hold.

□

Lemma 6.4 (Substitution B) *Let $M = (S, V)$ be a standard model for L. For all assignments φ into S, terms a, symbols x, and types, terms, or formulas e, if*

1. *a is eval-free,*
2. *e is semi-eval-free,*
3. *$V_\varphi(a) \neq \bot$, and*
4. *$V_\varphi(\mathsf{free\text{-}for}(\ulcorner a \urcorner, \ulcorner x \urcorner, \ulcorner e \urcorner)) = \mathrm{T}$,*

then

$$V_\varphi([\![\mathsf{sub}(\ulcorner a \urcorner, \ulcorner x \urcorner, \ulcorner e \urcorner)]\!]_{k[e]}) = V_{\varphi[x \mapsto V_\varphi(a)]}(e).$$

Proof Fix an assignment φ into S and a term a, a symbol x, and a proper expression e that is a type, term, or formula. Assume a is eval-free, e is semi-eval-free, and $V_\varphi(a) \neq \bot$. Let $S(\ulcorner e \urcorner)$ mean $\mathsf{sub}(\ulcorner a \urcorner, \ulcorner x \urcorner, \ulcorner e \urcorner)$. We will show that, if

$$V_\varphi(\mathsf{free\text{-}for}(\ulcorner a \urcorner, \ulcorner x \urcorner, \ulcorner e \urcorner)) = \mathrm{T} \quad [\text{designated } H(\ulcorner e \urcorner)]$$

and then

$$V_\varphi([\![S(\ulcorner e \urcorner)]\!]_{k[e]}) = V_{\varphi[x \mapsto V_\varphi(a)]}(e) \quad [\text{designated } C(\ulcorner e \urcorner)].$$

Our proof is by induction on the complexity of e. There are 12 cases corresponding to the 12 sentence schemas used to define $\mathsf{sub}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner, \ulcorner e_3 \urcorner))$ when e_1 is a term, e_2 is a symbol, e_3 is a type, term, or formula.

Case 1: $e = O(e_1, \ldots, e_n)$ and $O = (s :: k_1, \ldots, k_{n+1})$. By the definition of $\mathsf{free\text{-}for}$, $H(\ulcorner e \urcorner)$ implies (1) $H(\ulcorner k_i \urcorner)$ for all i with $1 \leq i \leq n+1$ and (2) $H(\ulcorner e_i \urcorner)$ for all i with $1 \leq i \leq n$. By the induction hypothesis, this implies (1) $C(\ulcorner k_i \urcorner)$ for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$ and (2) $C(\ulcorner e_i \urcorner)$ for all i with $1 \leq i \leq n$. Then

$$\begin{aligned}
& V_\varphi([\![S(\ulcorner e \urcorner)]\!]_{k[e]}) \\
= \; & V_\varphi([\![S(\ulcorner O(e_1, \ldots, e_n) \urcorner)]\!]_{k[e]}) \\
= \; & V_\varphi([\![\ulcorner \lfloor S(\ulcorner O \urcorner) \rfloor (\lfloor S(\ulcorner e_1 \urcorner) \rfloor, \ldots, \lfloor S(\ulcorner e_n \urcorner) \rfloor) \urcorner]\!]_{k[e]}) \\
= \; & V_\varphi(\overline{S(\ulcorner O \urcorner)}([\![S(\ulcorner e_1 \urcorner)]\!]_{k[e_1]}, \ldots, [\![S(\ulcorner e_n \urcorner)]\!]_{k[e_n]})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(O(e_1, \ldots, e_n)) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e)
\end{aligned}$$

where

$$\overline{S(\ulcorner O \urcorner)} = (s :: \overline{S(k_1)}, \ldots, \overline{S(k_{n+1})})$$

and

$$\overline{S(k_i)} = \begin{cases} [\![S(\ulcorner k_i \urcorner)]\!]_{\mathrm{ty}} & \text{if } \mathbf{type}[k_i] \\ k_i & \text{if } k_i = \mathsf{type} \text{ or } k_i = \mathsf{formula}. \end{cases}$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); and the fifth holds since (1) O and $\overline{S(\ulcorner O \urcorner)}$ are similar by Lemma 6.3(1), (2) $C(\ulcorner k_i \urcorner)$ holds for all i with $1 \leq i \leq n+1$ and $\mathbf{type}[k_i]$, and (3) $C(\ulcorner e_i \urcorner)$ holds for all i with $1 \leq i \leq n$. Therefore, $C(\ulcorner e \urcorner)$ holds.

Case 2: $e = (x : \alpha)$. By the definition of free-for, $H(\lceil e \rceil)$ implies $H(\lceil \alpha \rceil)$. By the induction hypothesis, this implies $C(\lceil \alpha \rceil)$. Then

$$\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_{k[e]}) \\
= \; & V_\varphi([\![S(\lceil (x : \alpha) \rceil)]\!]_{\mathrm{te}}) \\
= \; & V_\varphi([\![\mathsf{if}(a \downarrow [\![S(\alpha)]\!]_{\mathrm{ty}}, \lceil a \rceil, \lceil \bot_{\mathsf{C}} \rceil)]\!]_{\mathrm{te}}) \\
= \; & V_\varphi(\mathsf{if}(a \downarrow [\![S(\alpha)]\!]_{\mathrm{ty}}, [\![\lceil a \rceil]\!]_{\mathrm{te}}, [\![\lceil \bot_{\mathsf{C}} \rceil]\!]_{\mathrm{te}})) \\
= \; & V_\varphi(\mathsf{if}(a \downarrow [\![S(\alpha)]\!]_{\mathrm{ty}}, a, \bot_{\mathsf{C}})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}((x : \mathsf{C}) \downarrow \alpha, (x : \mathsf{C}), \bot_{\mathsf{C}})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}((x : \alpha)) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e).
\end{aligned}$$

The third line is by the definition of sub; the fourth and fifth are by Lemma 6.1; the sixth is by $C(\lceil \alpha \rceil)$; and the seventh is by the definition of the standard valuation applied to variables. Therefore, $C(\lceil e \rceil)$ holds.

Case 3: $e = (y : \alpha)$ where $x \neq y$. By the definition of free-for $H(\lceil e \rceil)$ implies $H(\lceil \alpha \rceil)$. By the induction hypothesis, this implies $C(\lceil \alpha \rceil)$. Then

$$\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_{k[e]}) \\
= \; & V_\varphi([\![S(\lceil (y : \alpha) \rceil)]\!]_{\mathrm{te}}) \\
= \; & V_\varphi([\![\lceil (y : \lfloor S(\alpha) \rfloor) \rceil]\!]_{\mathrm{te}}) \\
= \; & V_\varphi((y : [\![S(\lceil \alpha \rceil)]\!]_{\mathrm{ty}})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}((y : \alpha)) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e).
\end{aligned}$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); and the fifth is by $C(\lceil \alpha \rceil)$. Therefore, $C(\lceil e \rceil)$ holds.

Case 4: $e = (\star x : \alpha \,.\, e')$ and $\star$ is Λ, λ, $\exists$, ι, or ϵ. By the definition of free-for, $H(\lceil e \rceil)$ implies $H(\lceil \alpha \rceil)$. By the induction hypothesis, this implies $C(\lceil \alpha \rceil)$. Then

$$\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_{k[e]}) \\
= \; & V_\varphi([\![S(\lceil (\star x : \alpha \,.\, e') \rceil)]\!]_{k[e]}) \\
= \; & V_\varphi([\![\lceil (\star x : \lfloor S(\lceil \alpha \rceil) \rfloor \,.\, e') \rceil]\!]_{k[e]}) \\
= \; & V_\varphi((\star x : [\![S(\lceil \alpha \rceil)]\!]_{\mathrm{ty}} \,.\, e'))
\end{aligned}$$

$$
\begin{aligned}
&= V_{\varphi[x\mapsto V_\varphi(a)]}((\star\, x : \alpha\ .\ e')) \\
&= V_{\varphi[x\mapsto V_\varphi(a)]}(e).
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); and the fifth is by $C(\lceil\alpha\rceil)$ and the fact that

$$V_{\varphi[x\mapsto d]}(e') = V_{\varphi[x\mapsto V_\varphi(a)][x\mapsto d]}(e')$$

for all $d \in D_{\mathrm{c}}$. Therefore, $C(\lceil e\rceil)$ holds.

Case 5: $e = (\star y : \alpha\ .\ e')$, $x \neq y$, and $\star$ is Λ, λ, $\exists$, ι, or ϵ. By the definition of free-for, $H(\lceil e\rceil)$ implies $H(\lceil\alpha\rceil)$ and either $(*)$ $V_\varphi(\neg\mathsf{free\text{-}in}(\lceil x\rceil, \lceil e\rceil)) = \mathrm{T}$ or $(**)$ $V_\varphi(\neg\mathsf{free\text{-}in}(\lceil y\rceil, \lceil a\rceil)) = \mathrm{T}$ and $H(\lceil e'\rceil)$. By the induction hypothesis, this implies $C(\lceil\alpha\rceil)$ and, in case $(**)$, $C(\lceil e'\rceil)$. Then

$$
\begin{aligned}
&V_\varphi([\![S(\lceil e\rceil)]\!]_{k[e]}) \\
&= V_\varphi([\![S(\lceil(\star y : \alpha\ .\ e')\rceil)]\!]_{k[e]}) \\
&= V_\varphi([\![\lceil(\star y : \lfloor S(\lceil\alpha\rceil)\rfloor\ .\ \lfloor S(\lceil e'\rceil)\rfloor\rceil]\!]_{k[e]}) \\
&= V_\varphi((\star y : [\![S(\lceil\alpha\rceil)]\!]_{\mathrm{ty}}\ .\ [\![S(\lceil e'\rceil)]\!]_{k[e']})) \\
&= V_{\varphi[x\mapsto V_\varphi(a)]}((\star y : \alpha\ .\ e')) \\
&= V_{\varphi[x\mapsto V_\varphi(a)]}(e).
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); and the fifth is by $C(\lceil\alpha\rceil)$ and separate arguments for the two cases $(*)$ and $(**)$. In case $(*)$,

$$
\begin{aligned}
&V_{\varphi[y\mapsto d]}([\![S(\lceil e'\rceil)]\!]_{k[e]}) \\
&= V_{\varphi[y\mapsto d]}(e') \\
&= V_{\varphi[y\mapsto d][x\mapsto\varphi(a)]}(e') \\
&= V_{\varphi[x\mapsto\varphi(a)][y\mapsto d]}(e')
\end{aligned}
$$

for all $d \in D_{\mathrm{c}}$ by Lemma 6.3(4) and Lemma 6.2. In case $(**)$,

$$
\begin{aligned}
&V_{\varphi[y\mapsto d]}([\![S(\lceil e'\rceil)]\!]_{k[e]}) \\
&= V_{\varphi[y\mapsto d][x\mapsto\varphi[y\mapsto d](a)]}(e') \\
&= V_{\varphi[y\mapsto d][x\mapsto\varphi(a)]}(e') \\
&= V_{\varphi[x\mapsto\varphi(a)][y\mapsto d]}(e')
\end{aligned}
$$

for all $d \in D_{\mathrm{c}}$ by $C(\lceil e'\rceil)$ and Lemma 6.2. Therefore, $C(\lceil e\rceil)$ holds.

Case 6: $e = \alpha(a)$. Similar to Case 3.

Case 7: $e = f(a)$. Similar to Case 3.

Case 8: $e = \mathsf{if}(A, b, c)$. Similar to Case 3.

Case 9: $e = \lceil e' \rceil$. Then

$$\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_{k[e]}) \\
= \; & V_\varphi([\![S(\lceil \lceil e' \rceil \rceil)]\!]_{\mathrm{te}}) \\
= \; & V_\varphi([\![\lceil \lceil e' \rceil \rceil]\!]_{\mathrm{te}}) \\
= \; & V_\varphi(\lceil e' \rceil) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\lceil e' \rceil) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e)
\end{aligned}$$

The third line is by the definition of sub; the fourth is by Lemma 6.1; and the fifth by the fact that $V_\varphi(e)$ does not depend on φ. Therefore, $C(\lceil e \rceil)$ holds.

Case 10: $e = [\![b]\!]_{\mathrm{ty}}$. Let $V_\varphi(\mathsf{gea}(c, d)) = \mathrm{T}$ be designated by $G(c, d)$. By the definition of $\mathsf{free\text{-}for}$, $H(\lceil e \rceil)$ implies (1) $H(\lceil b \rceil)$ and (2) $G([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil)$ implies $H([\![S(\lceil b \rceil)]\!]_{\mathrm{te}})$. By the induction hypothesis, this implies (1) $C(\lceil b \rceil)$ and (2) $G([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil)$ implies $C([\![S(\lceil b \rceil)]\!]_{\mathrm{te}})$.

$$\begin{aligned}
& V_\varphi([\![S(\lceil e \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi([\![S(\lceil [\![b]\!]_{\mathrm{ty}} \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil), S([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}), \lceil \mathsf{C} \rceil)]\!]_{\mathrm{ty}}) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil), [\![S([\![S(\lceil b \rceil)]\!]_{\mathrm{te}})]\!]_{\mathrm{ty}}, \mathsf{C})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}(\mathsf{gea}([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil), [\![[\![S(\lceil b \rceil)]\!]_{\mathrm{te}}]\!]_{\mathrm{ty}}, \mathsf{C})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}(\mathsf{gea}(b, \lceil \mathsf{type} \rceil), [\![b]\!]_{\mathrm{ty}}, \mathsf{C})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}([\![b]\!]_{\mathrm{ty}}) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e)
\end{aligned}$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); the fifth is by $G([\![S(\lceil b \rceil)]\!]_{\mathrm{te}}, \lceil \mathsf{type} \rceil)$ implies $C([\![S(\lceil b \rceil)]\!]_{\mathrm{te}})$; the sixth is by $C(\lceil b \rceil)$; and the seventh is by the definition of the standard valuation applied to evaluations. . Therefore, $C(\lceil e \rceil)$ holds.

Case 11: $e = [\![b]\!]_\alpha$. Let $V_\varphi(\mathsf{gea}(c,d) \wedge [\![c]\!]_{\mathrm{te}} \downarrow [\![d]\!]_{\mathrm{ty}}) = \mathrm{T}$ be designated by $G(c,d)$. By the definition of free-for, $H(\ulcorner e \urcorner)$ implies (1) $H(\ulcorner b \urcorner)$, (2) $H(\ulcorner \alpha \urcorner)$, and (3) $G([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, S(\ulcorner \alpha \urcorner))$ implies $H([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})$. By the induction hypothesis, this implies (1) $C(\ulcorner b \urcorner)$, (2) $C(\ulcorner \alpha \urcorner)$, and (3) $G([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, S(\ulcorner \alpha \urcorner))$ implies $C([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})$.

$$
\begin{aligned}
& V_\varphi([\![S(\ulcorner e \urcorner)]\!]_\alpha) \\
= \; & V_\varphi([\![S(\ulcorner [\![b]\!]_\alpha \urcorner)]\!]_\alpha) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, S(\ulcorner \alpha \urcorner)) \wedge [\![[\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}]\!]_{\mathrm{te}} \downarrow [\![S(\ulcorner \alpha \urcorner)]\!]_{\mathrm{ty}}, \\
& \quad S([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}), \ulcorner \bot_{\mathsf{C}} \urcorner)]\!]_\alpha) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, S(\ulcorner \alpha \urcorner)) \wedge [\![[\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}]\!]_{\mathrm{te}} \downarrow [\![S(\ulcorner \alpha \urcorner)]\!]_{\mathrm{ty}}, \\
& \quad [\![S([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})]\!]_\alpha, \bot_{\mathsf{C}})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, S(\ulcorner \alpha \urcorner)) \wedge \\
& \quad [\![[\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}]\!]_{\mathrm{te}} \downarrow [\![S(\ulcorner \alpha \urcorner)]\!]_{\mathrm{ty}}, [\![[\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}]\!]_\alpha, \bot_{\mathsf{C}})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}(\mathsf{gea}(b, \ulcorner \alpha \urcorner \wedge [\![b]\!]_{\mathrm{te}} \downarrow \alpha), [\![b]\!]_\alpha, \bot_{\mathsf{C}})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}([\![b]\!]_\alpha) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e)
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); the fifth is by $G([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, S(\ulcorner \alpha \urcorner))$ implies $C([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})$; the sixth is by $C(\ulcorner b \urcorner)$ and $C(\ulcorner \alpha \urcorner)$; and the seventh is by the definition of the standard valuation applied to evaluations. Therefore, $C(\ulcorner e \urcorner)$ holds.

Case 12: $e = [\![b]\!]_{\mathrm{fo}}$. Let $V_\varphi(\mathsf{gea}(c,d)) = \mathrm{T}$ be designated by $G(c,d)$. By the definition of free-for, $H(\ulcorner e \urcorner)$ implies (1) $H(\ulcorner b \urcorner)$ and (2) $G([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner)$ implies $H([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})$. By the induction hypothesis, this implies (1) $C(\ulcorner b \urcorner)$ and (2) $G([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner)$ implies $C([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})$.

$$
\begin{aligned}
& V_\varphi([\![S(\ulcorner e \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi([\![S(\ulcorner [\![b]\!]_{\mathrm{fo}} \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi([\![\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner), S([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}), \ulcorner \mathsf{F} \urcorner)]\!]_{\mathrm{fo}}) \\
= \; & V_\varphi(\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner), [\![S([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})]\!]_{\mathrm{fo}}, \mathsf{F})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}(\mathsf{gea}([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner), [\![[\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}]\!]_{\mathrm{fo}}, \mathsf{F})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(\mathsf{if}(\mathsf{gea}(b, \ulcorner \mathsf{formula} \urcorner), [\![b]\!]_{\mathrm{fo}}, \mathsf{F})) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}([\![b]\!]_{\mathrm{fo}}) \\
= \; & V_{\varphi[x \mapsto V_\varphi(a)]}(e)
\end{aligned}
$$

The third line is by the definition of sub; the fourth is by Lemma 6.1 and Lemma 6.3(2); the fifth is by $G([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}}, \ulcorner \mathsf{formula} \urcorner)$ implies $C([\![S(\ulcorner b \urcorner)]\!]_{\mathrm{te}})$; the sixth is by $C(\ulcorner b \urcorner)$; and the seventh is by the definition of the standard valuation applied to evaluations. Therefore, $C(\ulcorner e \urcorner)$ holds.

□

6.5 Kernel Theory

The *kernel theory* of Chiron, written T_{ker}, is the theory (L, Γ) where:

1. L is the union of the set $\mathcal{O}_1$ of built-in operators and the set $\mathcal{O}_2$ of the operators defined in this section.
2. Γ is the set of defining sentences given in this section for the operators in $\mathcal{O}_2$.

7 Examples

7.1 Law of Excluded Middle

In many traditional logics, e.g., first-order logic, the law of excluded middle (LEM) is expressed as a formula schema

$$A \vee \neg A$$

where A can be any formula. In Chiron LEM can be expressed as a single formula:

$$\forall\, e : \mathsf{E}_{\mathrm{fo}} \,.\, [\![e]\!] \vee \neg [\![e]\!].$$

7.2 Modus Ponens

Like the law of excluded middle, other laws of logic are expressed as formula schemas. In Chiron these laws can be expressed as single formulas. For example, the following Chiron formula expresses the law of modus ponens:

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{fo}} \,.\, ([\![e_1]\!] \wedge [\![e_2]\!] \wedge \mathsf{is\text{-}impl}(e_2) \wedge e_1 = \mathsf{1st\text{-}arg}(e_2)) \supset [\![\mathsf{2nd\text{-}arg}(e_2)]\!]_{\mathrm{fo}}.$$

Deduction and computation rules are naturally represented as *transformers* [7], algorithms that map expressions to expressions. Transformers can

be directly formalized in Chiron. For example, the following function abstraction, which maps a pair of formulas to a formula, formalizes the modus ponens rule of inference:

$$\lambda\, x : \mathsf{E}_{\mathrm{fo}} \times \mathsf{E}_{\mathrm{fo}} \,.\; \mathsf{if}(\mathsf{is\text{-}impl}(\mathsf{tl}(x)) \wedge \mathsf{hd}(x) = \mathsf{1st\text{-}arg}(\mathsf{tl}(x)), \mathsf{2nd\text{-}arg}(\mathsf{tl}(x)), \bot_{\mathsf{C}}).$$

Let $(\mathsf{modus\text{-}ponens} :: (\Lambda\, x : \mathsf{E}_{\mathrm{fo}} \times \mathsf{E}_{\mathrm{fo}} \,.\, \mathsf{E}_{\mathrm{fo}}))$ be an individual constant that is defined to be this function abstraction, and let $\mathsf{mp}(a)$ mean

$$(\mathsf{modus\text{-}ponens} :: (\Lambda\, x : \mathsf{E}_{\mathrm{fo}} \times \mathsf{E}_{\mathrm{fo}} \,.\, \mathsf{E}_{\mathrm{fo}}))(a).$$

Then the following formula is an alternate expression of the law of modus ponens:

$$\forall\, e_1, e_2 : \mathsf{E}_{\mathrm{fo}} \,.\, (\llbracket e_1 \rrbracket \wedge \llbracket e_2 \rrbracket \wedge \mathsf{mp}(\langle e_1, e_2 \rangle)\downarrow) \supset \llbracket \mathsf{mp}(\langle e_1, e_2 \rangle) \rrbracket.$$

7.3 Beta Reduction

There are two laws of beta reduction in Chiron, one for the application of a dependent function type and one for the application of a function abstraction. Without quotation and evaluation, the latter beta reduction law would be expressed as the formula schema

$$(\lambda\, x : \alpha \,.\, b)(a) \simeq b[(x : \alpha) \mapsto a]$$

where a is free for $(x : \alpha)$ in b. Notice that this schema includes four schema variables (x, α, b, a), a substitution instruction, and a syntactic side condition.

Using constructions, quotation, and evaluation, both laws of beta reduction can be formalized in Chiron as *single formulas.* For example, the law of beta reduction for function abstractions is:

$$\begin{array}{l}\forall\, e : \mathsf{E}_{\mathrm{te}} \,. \\ \quad (\mathsf{is\text{-}redex}(e) \wedge \\ \quad\; \mathsf{free\text{-}for}(\mathsf{redex\text{-}arg}(e), \mathsf{1st\text{-}comp}(\mathsf{redex\text{-}var}(e)), \mathsf{redex\text{-}body}(e))) \\ \quad\quad \supset \\ \quad \llbracket e \rrbracket \simeq \llbracket \mathsf{sub}(\mathsf{redex\text{-}arg}(e), \mathsf{1st\text{-}comp}(\mathsf{redex\text{-}var}(e)), \mathsf{redex\text{-}body}(e)) \rrbracket_{\mathrm{te}}.\end{array}$$

The beta reduction law for function abstractions is applied to an application a of a function abstraction by instantiating the variable $(e : \mathsf{E}_{\mathrm{te}})$ with the quotation $\lceil a \rceil$.

7.4 Liar Paradox

We will formalize in Chiron the liar paradox mentioned in section 4.1. Assume that nat is a type and $0, 1, 2, \ldots$ denote terms such that nat denotes an infinite set $\{0, 1, 2, \ldots\}$. Assume also that num is a type that denotes the set $\{\ulcorner 0\urcorner, \ulcorner 1\urcorner, \ulcorner 2\urcorner, \ldots\}$. And finally assume that enum is a term that denotes a function which is a bijection from $\{\ulcorner 0\urcorner, \ulcorner 1\urcorner, \ulcorner 2\urcorner, \ldots\}$ to the set F of all functions of type $(\Lambda\, e : \mathsf{E}\ .\ \mathsf{E})$ that are definable by a syntactically closed function abstraction of the form $(\lambda\, e : \mathsf{E}\ .\ b)$.

The following function abstraction denotes some $f \in F$:

$$
\begin{array}{l}
\lambda\, e : \mathsf{E}\ . \\
\quad [\ulcorner \mathsf{op\text{-}app} \urcorner, \\
\quad\quad [\ulcorner \mathsf{op} \urcorner, \ulcorner \mathsf{not} \urcorner, \ulcorner \mathsf{formula} \urcorner, \ulcorner \mathsf{formula} \urcorner], \\
\quad\quad [\ulcorner \mathsf{eval} \urcorner, \\
\quad\quad\quad [\ulcorner \mathsf{fun\text{-}app} \urcorner, [\ulcorner \mathsf{fun\text{-}app} \urcorner, \ulcorner \mathsf{enum} \urcorner, e], e], \\
\quad\quad\quad \ulcorner \mathsf{formula} \urcorner]].
\end{array}
$$

Therefore, for some i of type nat, $\mathsf{enum}(\ulcorner i \urcorner) = f$. Then

$$
\begin{array}{rcl}
\mathsf{enum}(\ulcorner i \urcorner)(\ulcorner i \urcorner) & = & f(\ulcorner i \urcorner) \\
 & = & \ulcorner \neg [\![\mathsf{enum}(\ulcorner i \urcorner)(\ulcorner i \urcorner)]\!]_{\mathrm{fo}} \urcorner.
\end{array}
$$

Therefore, if LIAR is the term $\mathsf{enum}(\ulcorner i \urcorner)(\ulcorner i \urcorner)$, then

$$\mathsf{LIAR} = \ulcorner \neg [\![\mathsf{LIAR}]\!]_{\mathrm{fo}} \urcorner.$$

8 Conclusion

In this paper we have presented the syntax and semantics of a set theory named Chiron that is intended to be a practical, general-purpose logic for mechanizing mathematics. Several simple examples are given that illustrate Chiron's facility for reasoning about the syntax of expressions. This paper is a first step in a long-range research program to design, analyze, and implement Chiron. In the future we plan to:

1. Design a proof system for Chiron.
2. Implement Chiron and its proof system.

3. Develop a series of applications to demonstrate Chiron's reach and level of effectiveness. As a first step, we have shown how biform theories can be formalized in Chiron [5]. A *biform theory* is a theory in which both formulas and algorithms can serve as axioms [5, 7].

Acknowledgments

The author is grateful to Marc Bender and Jacques Carette for many valuable discussions on the design and use of Chiron. Over the course of these discussions, Dr. Carette convinced the author that Chiron needs to include a powerful facility for reasoning about the syntax of expressions.

A Appendix: Alternate Semantics

This appendix presents two alternate semantics for Chiron based on S. Kripke's framework for defining semantics with *truth-value gaps* which is described in his famous paper *Outline of a Theory of Truth* [11]. Both semantics use *value gaps* for types and terms as well as for formulas. The first defines the value gaps according to weak Kleene logic [10], while the second defines the values gaps according to a valuation scheme based on B. van Fraassen's notion of a *supervaluation* [21] that Kripke describes in [11, p. 711].

A.1 Valuations

The notion of a valuation for a structure was defined in section 4.4. Fix a structure S for L. Let $\mathsf{val}(S)$ be the collection of valuations for S. Given $U, V \in \mathsf{val}(S)$, U is a *subvaluation* of V, written $U \sqsubseteq V$, if, for all $e \in \mathcal{E}$ and $\varphi \in \mathsf{assign}(S)$, $U_\varphi(e)$ is defined implies $U_\varphi(e) = V_\varphi(e)$. A *valuation functional* for S is a mapping from $\mathsf{val}(S)$ into $\mathsf{val}(S)$. Let Ψ be a valuation functional for S. A *fixed point* of Ψ is a $V \in \mathsf{val}(S)$ such that $\Psi(V) = V$. Ψ is *monotone* if $U \sqsubseteq V$ implies $\Psi(U) \sqsubseteq \Psi(V)$ for all $U, V \in \mathsf{val}(S)$.

Theorem A.1 *Let Ψ be a monotone valuation functional for S. Then Ψ has a fixed point.*

Proof The construction of a fixed point of Ψ is similar to the construction of the fixed point Kripke gives in [11, pp. 703–705]. □

Ψ_1^S is the valuation functional for S defined by the following rules where $V \in \mathsf{val}(S)$ and $V' = \Psi_1^S(V)$. There is a rule for the category of improper expressions and a rule for each of the 13 categories of proper expressions. Note that only part (a) of the rule 14 (the rule for evaluation) makes use of V. Ψ_1^S defines value gaps according to the weak Kleene logic valuation scheme in which a proper expression is denoting only if all of its proper subexpressions are also denoting.

1. Let $e \in \mathcal{E}$ be improper. Then $V'_\varphi(e)$ is undefined.

2. Let $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$ be proper. Then $V'_\varphi(O) = I(O)$.

3. Let $e = (\mathsf{op\text{-}app}, O, e_1, \ldots, e_n)$ be proper where

   $$O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1}).$$

   a. Let $V'_\varphi(k_i)$ be defined for all i with $1 \leq i \leq n$ and $\mathbf{type}[k_i]$, and let $V'_\varphi(e_1), \ldots, V'_\varphi(e_n)$ be defined. If $V'_\varphi(e_i)$ is in $V'_\varphi(k_i)$ or $V'_\varphi(e_i) = \bot$ for all i such that $1 \leq i \leq n$ and $\mathbf{type}[k_i]$, then

   $$V'_\varphi(e) = V'_\varphi(O)(V'_\varphi(e_1), \ldots, V'_\varphi(e_n)).$$

   Otherwise $V'_\varphi(e)$ is D_c if $k = \mathsf{type}$, $\bot$ if $\mathbf{type}[k]$, and F if $k = \mathsf{formula}$.

   b. Let $V'_\varphi(k_i)$ be undefined for some i with $1 \leq i \leq n$ and $\mathbf{type}[k_i]$, or let $V'_\varphi(e_i)$ be undefined for some i with $1 \leq i \leq n$. Then $V'_\varphi(e)$ is undefined.

4. Let $a = (\mathsf{var}, x, \alpha)$ be proper.

   a. Let $V'_\varphi(\alpha)$ be defined. If $\varphi(x)$ is in $V'_\varphi(\alpha)$, then $V'_\varphi(a) = \varphi(x)$. Otherwise $V'_\varphi(a) = \bot$.

   b. Let $V'_\varphi(\alpha)$ be undefined. Then $V'_\varphi(a)$ is undefined.

5. Let $\beta = (\mathsf{type\text{-}app}, \alpha, a)$ be proper.

   a. Let $V'_\varphi(\alpha)$ and $V'_\varphi(a)$ be defined. If $V'_\varphi(a) \neq \bot$, then $V'_\varphi(\beta) = V'_\varphi(\alpha)[V'_\varphi(a)]$. Otherwise $V'_\varphi(\beta) = D_\mathrm{c}$.

   b. Let $V'_\varphi(\alpha)$ or $V'_\varphi(a)$ be undefined. Then $V'_\varphi(\beta)$ is undefined.

6. Let $\gamma = (\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$ be proper.

a. Let $V'_\varphi(\alpha)$ be defined, and let $V'_{\varphi[x\mapsto d]}(\beta)$ be defined for all sets d in $V'_\varphi(\alpha)$. Then $V'_\varphi(\gamma)$ is the superclass of all f in D_f such that:
   i. For all sets d in $V'_\varphi(\alpha)$, if $f(d)$ is defined, then $f(d)$ is in $V'_{\varphi[x\mapsto d]}(\beta)$.
   ii. For all sets d not in $V'_\varphi(\alpha)$, $f(d)$ is undefined.

b. Let $V'_\varphi(\alpha)$ be undefined, or let $V'_{\varphi[x\mapsto d]}(\beta)$ be undefined for some set d in $V'_\varphi(\alpha)$. Then $V'_\varphi(\gamma)$ is undefined.

7. Let $b = (\mathsf{fun\text{-}app}, f, a)$ be proper.

   a. Let $V'_\varphi(f)$ and $V'_\varphi(a)$ be defined. If $V'_\varphi(f) \neq \bot$ and $V'_\varphi(a) \neq \bot$, then $V'_\varphi(b) = V'_\varphi(f)(V'_\varphi(a))$. Otherwise $V'_\varphi(b) = \bot$.
   b. Let $V'_\varphi(f)$ or $V'_\varphi(a)$ be undefined. Then $V'_\varphi(b)$ is undefined.

8. Let $f = (\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$ be proper.

   a. Let $V'_\varphi(\alpha)$ be defined, and let $V'_{\varphi[x\mapsto d]}(b)$ be defined for all sets d in $V'_\varphi(\alpha)$. If

$$g \;=\; \{\langle d, d'\rangle \mid d \text{ is a set in } V'_\varphi(\alpha) \text{ and } d' = V'_{\varphi[x\mapsto d]}(b) \text{ is a set}\}$$

   is in D_f, then $V'_\varphi(f) = g$. Otherwise $V'_\varphi(f) = \bot$.
   b. Let $V'_\varphi(\alpha)$ be undefined, or let $V'_{\varphi[x\mapsto d]}(b)$ be undefined for some set d in $V'_\varphi(\alpha)$. Then $V'_\varphi(f)$ is undefined.

9. Let $a = (\mathsf{if}, A, b, c)$ be proper.

   a. Let $V'_\varphi(A), V'_\varphi(b), V'_\varphi(c)$ be defined. If $V'_\varphi(A) = \textsc{t}$, then $V'_\varphi(a) = V'_\varphi(b)$. Otherwise $V'_\varphi(a) = V'_\varphi(c)$.
   b. Let one of $V'_\varphi(A), V'_\varphi(b), V'_\varphi(c)$ be undefined. Then $V'_\varphi(a)$ is undefined.

10. Let $A = (\mathsf{exist}, (\mathsf{var}, x, \alpha), B)$ be proper.

    a. Let $V'_\varphi(\alpha)$ be defined, and let $V'_{\varphi[x\mapsto d]}(B)$ be defined for all d in $V'_\varphi(\alpha)$. If there is some d in $V'_\varphi(\alpha)$ such that $V'_{\varphi[x\mapsto d]}(B) = \textsc{t}$, then $V'_\varphi(A) = \textsc{t}$. Otherwise, $V'_\varphi(A) = \textsc{f}$.

b. Let $V'_\varphi(\alpha)$ be undefined, or let $V'_{\varphi[x\mapsto d]}(B)$ be undefined for some d in $V'_\varphi(\alpha)$. Then $V'_\varphi(A)$ is undefined.

11. Let $a = (\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$ be proper.

    a. Let $V'_\varphi(\alpha)$ be defined, and let $V'_{\varphi[x\mapsto d]}(B)$ be defined for all d in $V'_\varphi(\alpha)$. If there is a unique d in $V'_\varphi(\alpha)$ such that $V'_{\varphi[x\mapsto d]}(B) = \textsc{t}$, then $V'_\varphi(a) = d$. Otherwise, $V'_\varphi(a) = \bot$.
    b. Let $V'_\varphi(\alpha)$ be undefined, or let $V'_{\varphi[x\mapsto d]}(B)$ be undefined for some d in $V'_\varphi(\alpha)$. Then $V'_\varphi(a)$ is undefined.

12. Let $a = (\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$ be proper.

    a. Let $V'_\varphi(\alpha)$ be defined, and let $V'_{\varphi[x\mapsto d]}(B)$ be defined for all d in $V'_\varphi(\alpha)$. If there is some d in $V'_\varphi(\alpha)$ such that $V'_{\varphi[x\mapsto d]}(B) = \textsc{t}$, then $V'_\varphi(a) = \xi(\Sigma)$ where Σ is the superclass of all d in $V'_\varphi(\alpha)$ such that $V'_{\varphi[x\mapsto d]}(B) = \textsc{t}$. Otherwise, $V'_\varphi(a) = \bot$.
    b. Let $V'_\varphi(\alpha)$ be undefined, or let $V'_{\varphi[x\mapsto d]}(B)$ be undefined for some d in $V'_\varphi(\alpha)$. Then $V'_\varphi(a)$ is undefined.

13. Let $a = (\mathsf{quote}, e)$ be proper. Then $V'_\varphi(a) = H(e)$.

14. Let $b = (\mathsf{eval}, a, k)$ be proper.

    a. Let $V'_\varphi(a)$ be defined and $V'_\varphi(k)$ be defined if $\mathbf{type}[k]$.
        i. Let $V'_\varphi(a)$ be in $D_{\mathrm{e,ty}}$ and $k = \mathsf{type}$, $V'_\varphi(a)$ be in $D_{\mathrm{e,te}}$ and $\mathbf{type}[k]$, or $V'_\varphi(a)$ be in $D_{\mathrm{e,fo}}$ and $k = \mathsf{formula}$.
            A. Let $V_\varphi(H^{-1}(V'_\varphi(a)))$ be defined. If $k \in \{\mathsf{type}, \mathsf{formula}\}$ or $\mathbf{type}[k]$ and $V_\varphi(H^{-1}(V'_\varphi(a)))$ is in $V'_\varphi(k)$, then $V'_\varphi(b) = V_\varphi(H^{-1}(V'_\varphi(a)))$. Otherwise $V'_\varphi(b)$ is $\bot$.
            B. Let $V_\varphi(H^{-1}(V'_\varphi(a)))$ be undefined. Then $V'_\varphi(b)$ is undefined.
        ii. Let $V'_\varphi(a)$ not be in $D_{\mathrm{e,ty}}$ or $k \neq \mathsf{type}$, $V'_\varphi(a)$ not be in $D_{\mathrm{e,te}}$ or not $\mathbf{type}[k]$, and $V'_\varphi(a)$ not be in $D_{\mathrm{e,fo}}$ or $k \neq \mathsf{formula}$. Then $V'_\varphi(b)$ is D_{c} if $k = \mathsf{type}$, $\bot$ if $\mathbf{type}[k]$, and $\textsc{f}$ if $k = \mathsf{formula}$.
    b. Let $V'_\varphi(a)$ be undefined or $V'_\varphi(k)$ be undefined if $\mathbf{type}[k]$. Then $V'_\varphi(b)$ is undefined.

Lemma A.2 *Ψ_1^S is monotone.*

Proof Let $U, V \in \mathsf{val}(S)$ such that $U \sqsubseteq V$. Assume U'_φ and V'_φ mean $(\Psi_1^S(U))_\varphi$ and $(\Psi_1^S(V))_\varphi$, respectively. We must show that, for all $e \in \mathcal{E}$ and $\varphi \in \mathsf{assign}(S)$, if $U'_\varphi(e)$ is defined, then $U'_\varphi(e) = V'_\varphi(e)$. Our proof will be by induction on the number of symbols in e.

There are three cases:

1. *e is improper.* Then $U'_\varphi(e)$ is undefined by the definition of Ψ_1^S.

2. *$e = (\mathsf{eval}, a, k)$ is proper.* If either $U'_\varphi(a)$ or $U'_\varphi(k)$ is undefined, then $U'_\varphi(e)$ is undefined. So assume $U'_\varphi(a)$ and $U'_\varphi(k)$ are defined. By the induction hypothesis, $U'_\varphi(a) = V'_\varphi(a)$ and $U'_\varphi(k) = V'_\varphi(k)$. Assume $U'_\varphi(e)$ is defined. By the definition of Ψ_1^S, there are two subcases:

   a. *For some $e_1, e_2 \in \mathcal{E}$, $U'_\varphi(e) = U_\varphi(e_1)$ and $V'_\varphi(e) = V_\varphi(e_2)$.* Since $U'_\varphi(a) = V'_\varphi(a)$, $e_1 = e_2$, and since $U \sqsubseteq V$, $U_\varphi(e_1) = V_\varphi(e_2)$. Hence, $U'_\varphi(e) = V'_\varphi(e)$.

   b. *$U'_\varphi(e)$ and $V'_\varphi(e)$ both equal D_c if $k = \mathsf{type}$, $\perp$ if* $\mathbf{type}[k]$, and F *if $k = \mathsf{formula}$.* Hence, $U'_\varphi(e) = V'_\varphi(e)$.

3. *e is proper but not an evaluation.* Assume $U'_\varphi(e)$ is defined. Then $U'_\varphi(e')$ is defined for each subexpression e' of e. By the induction hypothesis, $U'_\varphi(e') = V'_\varphi(e')$ for each such subexpression e' of e. Hence, $U'_\varphi(e) = V'_\varphi(e)$.

□

Corollary A.3 *Ψ_1^S has a fixed point.*

Proof By Lemma A.2, Ψ_1^S is monotone. Therefore, by Theorem A.1, Ψ_1^S has a fixed point. □

Ψ_2^S is the valuation functional for S defined by the following three rules where $V \in \mathsf{val}(S)$ and $V' = \Psi_2^S(V)$. Ψ_2^S defines value gaps according to the supervaluation scheme.

1. Let $e \in \mathcal{E}$ be improper. Then $V'_\varphi(e)$ is undefined.

2. Let $\mathbf{p\text{-}expr}_L[e]$ such that e is not an evaluation. If there is a value d such that, for all total valuations V^* with $V \sqsubseteq V^*$, $(\Psi_1^S(V^*))_\varphi(e) = d$, then $V'_\varphi(e) = d$. Otherwise $V'_\varphi(e)$ is undefined.

3. Let $\mathbf{p\text{-}expr}_L[b]$ with $b = (\mathsf{eval}, a, k)$. This rule is exactly the same as the Ψ_1^S rule for evaluations.

Lemma A.4 *Ψ_2^S is monotone.*

Proof The proof is exactly the same as the proof of Lemma A.2 except for the argument for the third case:

3. *e is proper but not an evaluation.* Assume $U'_\varphi(e)$ is defined. Then there is a value d such that, for all total valuations U^* with $U \sqsubseteq U^*$, $U^*_\varphi(e) = d$. Since $U \sqsubseteq V$, it follows that, for all total valuations V^* with $V \sqsubseteq V^*$, $V^*_\varphi(e) = d$. Hence, $U'_\varphi(e) = V'_\varphi(e)$.

□

Corollary A.5 *Ψ_2^S has a fixed point.*

Proof By Lemma A.4, Ψ_2^S is monotone. Therefore, by Theorem A.1, Ψ_2^S has a fixed point. □

A.2 Models

The valuation functionals Ψ_1^S and Ψ_2^S define two semantics, which we will refer to as the *weak Kleene semantics* and the *supervaluation semantics*, respectively. Clearly, the supervaluation semantics allows more expressions to be denoting than the weak Kleene semantics.

A *weak Kleene model* for L is a model $M = (S, V)$ where S is a structure for L and V is a valuation for S that is a fixed point of Ψ_1^S. A *supervaluation model* for L is a model $M = (S, V)$ where S is a structure for L and V is a valuation for S that is a fixed point of Ψ_2^S.

Theorem A.6 *Let L be a language of Chiron. For each structure S for L there exists a weak Kleene model and a supervaluation model for L.*

Proof Let L be a language of Chiron and S be a structure for L. By Corollary A.3, Ψ_1^S has a fixed point V_1. Similarly, by Corollary A.5, Ψ_2^S has a fixed point V_2. Therefore, $M = (S, V_1)$ is a weak Kleene model for L, and $M = (S, V_2)$ is a supervaluation model for L. □

The weak Kleene semantics defined by Ψ_1^S is "strict" in the sense that, if any proper subexpression e of a proper expression e' is nondenoting, then e' itself is nondenoting. The supervaluation semantics defined by Ψ_2^S is not strict in this sense. For example, the value of an application of the operator

$$(\mathsf{op}, \mathsf{or}, \mathsf{formula}, \mathsf{forumla}, \mathsf{formula})$$

to a pair of formulas (A, B) is T if the value of A is T and B is nondenoting or visa versa.

A.3 Discussion

There are various Kripke-style value-gap semantics for Chiron; the weak Kleene and supervaluation semantics are just two examples. The weak Kleene semantics is a conservative example: every expression that could be nondenoting is nondenoting. On the other hand, the supervaluation semantics is much more liberal: many expressions that are nondenoting in the weak Kleene semantics are denoting in the supervaluation semantics.

It is not possible to define a denoting formula checker in any Kripke-style value-gap semantics. If the operator $O = (s :: \mathsf{formula}, \mathsf{formula})$ were a denoting formula checker, then $O(e)$ would be true whenever e is denoting and false whenever e is nondenoting. However, such an operator breaks the monotonicity lemmas proved above because, if $U'_\varphi(e)$ is undefined but $V'_\varphi(e)$ is defined, then $U'_\varphi(O(e)) = \textsc{f} \neq \textsc{t} = V'_\varphi(O(e))$. Similarly, it is not possible to define denoting type and term checkers in a Kripke-style value-gap semantics.

The lack of available checkers for denoting types, terms, and formulas makes reasoning in Kripke-style value-gap semantics very difficult. For example, consider the formalization of the law of excluded middle given in subsection 7.1:

$$\forall\, e : \mathsf{E}_{\mathrm{fo}} \,.\, [\![e]\!] \vee \neg[\![e]\!].$$

This formula is false in the weak Kleene semantics because, if e represents a nondenoting formula, then $[\![e]\!] \vee \neg[\![e]\!]$ is nondenoting. Since it is false, we cannot use it as a basis for proof by cases.

This formula is true in the supervaluations semantics because, if e represents a nondenoting formula, then $[\![e]\!] \vee \neg[\![e]\!]$ is true because $[\![e]\!] \vee \neg[\![e]\!]$ is true no matter what value is assigned to $[\![e]\!]$. Even though this formula is true, we cannot use it as a basis for proof by cases because, if e is the liar paradox, we can derive a contradiction from either $[\![e]\!]$ or $\neg[\![e]\!]$.

We expect that reasoning in the official semantics for Chiron will be much easier than in any Kripke-style value-gap semantics for Chiron.

References

[1] A. Bawden. Quasiquotation in lisp. In O. Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999. Technical report BRICS-NS-99-1, University of Aarhus, 1999.

[2] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[3] W. M. Farmer. STMM: A Set Theory for Mechanized Mathematics. *Journal of Automated Reasoning*, 26:269–289, 2001.

[4] W. M. Farmer. Formalizing undefinedness arising in calculus. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning—IJCAR 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 475–489. Springer-Verlag, 2004.

[5] W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer-Verlag, 2007.

[6] W. M. Farmer. Chiron: A multi-paradigm logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1–19. University of Białystok, 2007.

[7] W. M. Farmer and M. von Mohrenschildt. An overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.

[8] K. Gödel. Über formal unentscheidbare Sätze der Principia Mthematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[9] K. Gödel. *The Consistency of the Axiom of Choice and the Generalized Continuum Hypothesis with the Axioms of Set Theory*, volume 3 of *Annals of Mathematical Studies*. Princeton University Press, 1940.

[10] S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[11] S. Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[12] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, fourth edition, 1997.

[13] I. L. Novak. A construction for models of consistent systems. *Fundamenta Mathematicae*, 37:87–110, 1950.

[14] A. M. Pitts. Nominal Logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[15] W. V. O. Quine. *Mathematical Logic: Revised Edition*. Harvard University Press, 2003.

[16] J. B. Rosser and H. Wang. Non-standard models for formal logics. *Journal of Symbolic Logic*, 15:113–129, 1950.

[17] J. Shoenfield. A relative consistency proof. *Journal of Symbolic Logic*, 19:21–28, 1954.

[18] A. Tarski. Der wahrheitsbegriff in den formalisierten sprachen. *Studia Philosophica*, 1:261–405, 1935.

[19] A. Tarski. The concept of truth in formalized languages. In J. Corcoran, editor, *Logic, Semantics, Meta-Mathematics*. Hackett, 1983.

[20] C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In R. Niewenhuis, editor, *Automated Deduction—CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, 2005.

[21] B. C. van Fraassen. Singular terms, truth-value gaps, and free logic. *Journal of Philosophy*, 63:481–495, 1966.