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

William M. Farmer†
McMaster University

2 May 2007

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

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

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 [2].

The following is the outline of the paper. Section 2 gives an informal overview of Chiron. Section 3 presents Chiron’s official syntax, and section 4 introduces an unofficial compact notation for Chiron. The semantics of Chiron is given in section 5. Several useful operators are defined in section 6. Some of the practical expressivity of Chiron is illustrated in section 7. In section 8, we show that there is a faithful interpretation of NBG in Chiron. The paper concludes in section 9 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 [12, 14, 15]. 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 [8] or [11].)
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, $\mathbf{t}$ representing *true* and $\mathbf{f}$ 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 undefined definite or indefinite descriptions. $\bot$ is not a member of $D_s \cup \{\mathbf{t}, \mathbf{f}\}$.

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

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

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

$$f : D_v \rightarrow D_v.$$  

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 if 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 non-denoting (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 $\alpha$ and $\alpha$ denotes a superclass $\Sigma_\alpha$. If $a$ is defined, i.e., $a$ denotes a class $x$, then $x$ is in $\Sigma_\alpha$. 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

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

where \( \alpha \) and \( \beta \) are types. (Dependent function types are commonly known as dependent product types.) The type \( \gamma \) denotes a superclass of possibly partial functions. A function abstraction of the form

\[ (\text{fun-abs}, (\text{var}, x, \alpha), b), \]

where \( b \) is a term of type \( \beta \), is of type \( \gamma \).

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

\[ (\text{fun-app}, f, a), \]

the application of \( f \) to \( a \), is of the type

\[ (\text{type-app}, \gamma, a), \]

the type formed by applying the type \( \gamma \) 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 $(\text{quote}, e)$, 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 $(\text{quote}, e)$.

There are two ways to refer to a semantic meaning $v$. The first is to directly form a proper expression $e$ not beginning with $\text{eval}$ 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 $(\text{eval}, a, \text{type})$, term $(\text{eval}, a, \alpha)$, or formula $(\text{eval}, a, \text{formula})$ (depending on whether $e$ is a type, a term assigned the type $\alpha$, 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 $\text{quote}$ 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 $(\text{quote}, e)$ which denotes $c$.

For an expression $e$, the term $(\text{quote}, e)$ 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 $(\text{eval}, a, \alpha)$, where $\alpha$ is a type, may be undefined.

3 Syntax

This section presents the syntax of Chiron.

Let $S$ be an infinite set of symbols and $K$ be the set of the 30 symbols called key words in Table 1. Assume $K \subseteq S$. 
Table 1: The Key Words of Chiron.

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

\textbf{Expr-1 (Atomic expression)}

\[ \mathit{s} \in \mathcal{S} \]
\[ \texttt{expr}\,[\mathit{s}] \]

\textbf{Expr-2 (Compound expression)}

\[ \texttt{expr}[\mathit{e}_1, \ldots, \mathit{e}_n] \]
\[ \texttt{expr}[\mathit{e}_1, \ldots, \mathit{e}_n] \]

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 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. \texttt{p-expr}\,[\mathit{e} : \mathit{e}'\,] \texttt{asserts that} \, \mathit{e} \, \in \, \mathcal{E} \, \texttt{is a proper expression to which the expression} \, \mathit{e}' \, \in \, \mathcal{E} \, \texttt{is assigned. An improper expression} \texttt{is an expression that is not a proper expression. Improper expressions are nondenoting.}

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

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

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

\textbf{P-Expr-1 (Operator)}

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

where \( n \geq 0 \).

\textbf{P-Expr-2 (Operator application)}

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

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

\textbf{P-Expr-3 (Variable)}

\[ x \in \mathcal{S}, \text{type}[\alpha] \]
\[ \text{term}[(\text{var}, x, \alpha) : \alpha] \]

\textbf{P-Expr-4 (Type application)}

\[ \text{type}[\alpha], \text{term}[a] \]
\[ \text{type}[(\text{type-app}, \alpha, a)] \]

\textbf{P-Expr-5 (Dependent function type)}

\[ \text{term}[(\text{var}, x, \alpha)], \text{type}[\beta] \]
\[ \text{type}[(\text{dep-fun-type}, (\text{var}, x, \alpha), \beta)] \]
We will use $O, P, Q, \ldots$ to denote operators, $\alpha, \beta, \gamma, \ldots$ to denote types, $a, b, c, \ldots$ to denote terms, and $A, B, C, \ldots$ to denote formulas.
Proposition 3.1 The formation rules assign a unique expression to each proper expression.

Let $O = (\text{op}, s, k_1, \ldots, k_{n+1})$ 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 $k_{n+1} = \text{type}$, $\text{type}[k_{n+1}]$, or $k_{n+1} = \text{formula}$, respectively. A base type is a type operator application of the form $(\text{op-app}, (\text{op}, s, \text{type}))$. An individual constant of type $\alpha$ is a term operator application of the form $(\text{op-app}, (\text{op}, s, \alpha))$. A truth-value constant is the formula operator application $(\text{op-app}, (\text{op}, \text{true}, \text{formula}))$ or $(\text{op-app}, (\text{op}, \text{false}, \text{formula}))$.

A subexpression of an expression is defined inductively as follows:

1. If $e$ is a proper expression, the $e$ 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 $e''$.

$e$ is a proper subexpression of $e'$ if $e$ is a subexpression of $e'$ and $e \neq e'$. Notice that an improper expression has no subexpressions and that a quotation has no proper subexpressions.

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.

4 Compact Notation

In this section we introduce 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.

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.
Table 2: The Built-In Operators of Chiron.

2. A variable \((x : \alpha)\) occurring in the body \(e\) of \((\star x : \alpha . e)\), where \(\star\) is \(\Lambda\), \(\lambda\), \(\exists\), \(\forall\), \(\iota\), or \(\epsilon\) may be written as \(x\) if there is no resulting ambiguity.

3. \((\star x_1 : \alpha_1 \ldots (\star x_n : \alpha_n . e) \ldots)\), where \(\star\) is \(\Lambda\), \(\lambda\), \(\exists\), or \(\forall\), may be written as

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

Similarly, \((\star x_1 : \alpha \ldots (\star x_n : \alpha . e) \ldots)\), where \(\star\) is \(\Lambda\), \(\lambda\), \(\exists\), 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 \(\alpha\), then a term of the form \((\star x : \alpha . e)\), where \(\star\) is \(\Lambda\), \(\lambda\), \(\exists\), \(\forall\), \(\iota\), or \(\epsilon\), may be written as \((\star x . e)\).

5. \([a]_t\) may be shortened to \([a]\) if \(a\) is of type \(E_t\). \([a]_e\) may be shortened to \([a]\) if \(a\) is of type \(E_e\). And \([a]_o\) may be shortened to \([a]\) if \(a\) is of type \(E_o\).
Using the compact notation, expressions can be written in Chiron so that they look very much like expressions written in mathematics textbooks and papers.

5 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.

5.1 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

$$[-[LIAR]_{fo}]$$

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

| Compact Notation | Official Notation |
|------------------|-------------------|
| (s :: k₁, . . . , kₙ₊₁) | (op, s, k₁, . . . , kₙ₊₁) |
| (s :: k₁, . . . , kₙ₊₁)(e₁, . . . , eₙ) | (op-app, (op, s, k₁, . . . , kₙ₊₁), e₁, . . . , eₙ) |
| (x : α) | (var, x, α) |
| α(a) | (type-app, α, a) |
| (λ x : α . β) | (fun-abs, (var, x, α), β) |
| f(a) | (fun-app, f, a) |
| if(A, b, c) | (if, A, b, c) |
| (∃ x : α . B) | (exist, (var, x, α), B) |
| (λ x : α . B) | (def-des, (var, x, α), B) |
| (ε x : α . B) | (indef-des, (var, x, α), B) |
| [e] | (quote, e) |
| [a]_{ty} | (eval, a, type) |
| [a]_{α} | (eval, a, α) |
| [a]_{te} | (eval, a, (op-app(op, class, type))) |
| [a]_{fo} | (eval, a, formula) |

Table 3: Compact Notation
| Compact Notation | Defining Expression |
|------------------|---------------------|
| T                | (true :: formula)( ) |
| F                | (false :: formula)( ) |
| V                | (set :: type)( ) |
| C                | (class :: type)( ) |
| E                | (expr :: type)( ) |
| E_{op}           | (expr-op :: type)( ) |
| E_{ty}           | (expr-type :: type)( ) |
| E_{te}           | (expr-term :: type)( ) |
| E_{fo}           | (expr-formula :: type)( ) |
| (a ∈ b)          | (in :: V, C, formula)(a, b) |
| (α =_{ty} β)     | (type-equal :: type, type, formula)(α, β) |
| (a =_{α} b)      | (term-equal :: C, C, type, formula)(a, b, α) |
| (a = b)          | (a =_{C} b) |
| (A ≡ B)          | (formula-equal :: formula, formula, formula)(A, B) |
| (¬A)             | (not :: formula, formula)(A) |
| (a ̸∈ b)         | (¬(a ∈ b)) |
| (a ̸= b)         | (¬(a = b)) |
| (A ∨ B)          | (or :: formula, formula, formula)(A, B) |
| (∀x : α . A)     | (¬(∃ x : α . (¬A))) |

Table 4: Additional Compact Notation
If the naive semantics is employed for quotation and evaluation, a contradiction is immediately obtained:

\[
[\text{LIAR}]_{\text{fo}} = \neg [\neg [\text{LIAR}]_{\text{fo}}]_{\text{fo}} = \neg \neg \text{LIAR,}
\]

\([\text{LIAR}]_{\text{fo}}\) is thus ungrounded in the sense that its value depends on itself.

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 (\text{eval}) from Chiron but keep quotation (\text{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 \([\text{LIAR}]_{\text{fo}}\) are not assigned any value at all. In his famous paper Outline of a Theory of Truth \[10\], 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 \[9\] and B. van Fraassen’s notion of a super-valuation \[17\]. 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 all evaluations of terms that denote constructions containing the value of \([\text{eval}]\) to be undefined. For example, the value of \([\text{LIAR}]_{\text{fo}}\) would be \(\bot\), the undefined value for formulas. It is important to understand that an evaluation of a term \(a\) containing \text{eval} can be defined as long as \(a\) denotes a construction that does not contain the value of \([\text{eval}]\). Thus the value of an evaluation like \([[[[17]]]]_{\text{eval}}\) would be the value of 17 because the construction representing 17 presumably does not contain the value of \([\text{eval}]\). The official semantics for Chiron defined in this section employs this third approach to blocking the liar paradox.
5.2 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 [8] or [11].

Let \(P = (D^P, \in^P)\) be a prestructure of Chiron. A \textit{class} of \(P\) is a member of \(D^P\). A \textit{set} of \(P\) is a member \(x\) of \(D^P\) such that \(x \in^P y\) for some member \(y\) of \(D^P\). That is, a set is a class that is itself a member of a class. A class is \textit{proper} if it is not a set. A \textit{superclass} of \(P\) is a collection of classes in \(D^P\).

We consider a class, as a collection of sets, to be a superclass. Let \(D^P\) be the domain of sets of \(P\) and \(D^s_P\) be the domain of superclasses of \(P\). The following inclusions hold: \(D^P \subset D^c_P \subset D^s_P\).

A \textit{function} of \(P\) is a member \(f\) of \(D^P\) such that:

1. Each \(p \in^P f\) is an ordered pair \(\langle x, y \rangle\) where \(x, y\) are in \(D^P\).
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 \(D^P\).

Let \(D^f_P\) be the domain of functions of \(P\). For \(f, x \in D^P\), \(f(x)\) denotes the unique \(y\) in \(D^P\) 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 \(D^P\).) For \(\Sigma\) in \(D^P\) and \(x\) in \(D^P\), \(\Sigma[x]\) denotes the class of all \(y\) in \(D^P\) such that, for some \(f\) in both \(\Sigma\) and \(D^f_P\), \(f(x) = y\).

Let \(\top^P\), \(\bot^P\), and \(\bot^P\) be distinct values not in \(D^P\). \(\top^P\) and \(\bot^P\) represent the truth values \textit{true} and \textit{false}, respectively. \(\bot^P\) is the \textit{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 \(D^P\), \(D^P \cup \{\bot\}\), or \(\{\top, \bot\}\) for each \(i\) with \(1 \leq i \leq n + 1\). Let \(D^o_P\) be the domain of operations of \(P\). We assume that \(D^P \cup \{\top^P, \bot^P\}\) and \(D^o_P\) are disjoint.

5.3 Structures

A \textit{structure} for \(L\) is a tuple

\[ S = (D_v, D_c, D_s, D_t, D_o, \in, \top, \bot, \xi, H, I) \]

where:

1. \(P = (D_c, \in)\) is a prestructure of Chiron. \(D_v = D^P_v\), \(D_s = D^P_s\), \(D_t = D^P_t\), \(D_o = D^P_o\), \(\top = \top^P\), \(\bot = \bot^P\), and \(\bot = \bot^P\).

16
2. $\xi$ is a choice function on $D_s$. Hence, for all nonempty superclasses $\Sigma$ in $D_s$, $\xi(\Sigma)$ is a class in $\Sigma$.

3. $H$ is a total, injective mapping of $S$ into $D_v$ such that, for each $s \in S$, $H(s)$ is neither an ordered pair nor the empty set $\emptyset$. Let $\hat{H}$ be the total, injective mapping of $E$ into $D_v$ defined inductively by:
   a. If $s \in S$, then $\hat{H}(s) = H(s)$.
   b. If $e = () \in E$, then $\hat{H}(e) = \emptyset$.
   c. If $e = (e_1, \ldots, e_n) \in E$ where $n \geq 1$, then
      $$\hat{H}(e) = (\hat{H}(e_1), \hat{H}((e_2, \ldots, e_n))).$$

For $e \in E$, $\hat{H}(e)$ called the construction of $e$. It represents the syntactic structure of the expression $e$. The construction is a operator construction, type construction, term construction, or formula construction if it represents an operator, type, term, or formula, respectively.

4. For each operator $O = (\text{op}, s, k_1, \ldots, k_n, k_{n+1})$, $I(O)$ is an $n$-ary operation $o$ in $D_v$ from $D_{s_1} \times \cdots \times D_{s_n}$ into $D_{s_{n+1}}$ where $D_i = D_k$ if $k_i = \text{type}$, $D_i = D_c \cup \{\bot\}$ if $k_i = \text{type}[k_{i+1}]$, and $D_i = \{\text{T}, \text{F}\}$ if $k_i = \text{formula}$ for each $i$ with $1 \leq i \leq n + 1$ such that:
   a. If $O \notin L$, then, for all $(d_1, \ldots, d_n) \in D_{s_1} \times \cdots \times D_{s_n}$, $I(O)(d_1, \ldots, d_n)$ is $D_c$, $\bot$, or $\text{F}$ if $k_{n+1} = \text{type}$, $\text{type}[k_{n+1}]$, or $k_{n+1} = \text{formula}$, respectively.
   b. $I((\text{op}, \text{true}, \text{formula}))(()) = \text{T}$.
   c. $I((\text{op}, \text{false}, \text{formula}))(()) = \text{F}$.
   d. $I((\text{op}, \text{set}, \text{type}))(()) = D_v$, the universal class that contains all sets.
   e. $I((\text{op}, \text{class}, \text{type}))(()) = D_c$, the universal superclass that contains all classes.
   f. $I((\text{op}, \text{expr}, \text{type}))(()) = E$, the range of $\hat{H}$ (i.e., the class of constructions).
   g. $I((\text{op}, \text{expr-op}, \text{type}))(()) = E_{o_p}$, the range of $\hat{H}$ restricted to operators (i.e., the class of operator constructions).
   h. $I((\text{op}, \text{expr-type}, \text{type}))(()) = E_{o_y}$, the range of $\hat{H}$ restricted to types (i.e., the class of type constructions).
   i. $I((\text{op}, \text{expr-term}, \text{type}))(()) = E_{e_t}$, the range of $\hat{H}$ restricted to terms (i.e., the class of term constructions).
j. \( I((\text{op}, \text{expr-formula}, \text{type}))(\) \( ) = E_{\text{fo}}, \) the range of \( \tilde{H} \) restricted to formulas (i.e., the class of formula constructions).

k. If \( x \) is a member of \( D_v \) and \( y \) is a member of \( D_c \), then

\[
I((\text{op}, \text{in}, V, C, \text{formula}))(x, y)
\]

is \( T \) if \( x \) is a member of \( y \) and is \( F \) otherwise.

l. If \( \Sigma, \Sigma' \) are members of \( D_s \), then

\[
I((\text{op}, \text{type-equal}, \text{type}, \text{type}, \text{formula}))(\Sigma, \Sigma')
\]

is \( T \) if \( \Sigma \) and \( \Sigma' \) are identical and is \( F \) otherwise.

m. If \( x, y \) are members of \( D_c \) and \( \Sigma \) is a member of \( D_s \), then

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

is \( T \) if \( x, y \) are identical members of \( \Sigma \) and is \( F \) otherwise.

n. If \( t, t' \) are members of \( \{ t, f \} \), then

\[
I((\text{op}, \text{formula-equal}, \text{formula}, \text{formula}, \text{formula}))(t, t')
\]

is \( T \) if \( t \) and \( t' \) are identical and is \( F \) otherwise.

o. If \( t \) is a member of \( \{ t, f \} \), then

\[
I((\text{op}, \text{not}, \text{formula}, \text{formula}))(t)
\]

is \( T \) if \( t \) is \( F \) and is \( F \) otherwise.

p. If \( t, t' \) are members of \( \{ t, f \} \), then

\[
I((\text{op}, \text{or}, \text{formula}, \text{formula}, \text{formula}))(t, t')
\]

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

Fix a structure

\[
S = (D_v, D_c, D_s, D_t, D_o, \in, T, F, \bot, \xi, H, I)
\]

for \( L \). A variable assignment into \( S \) is a mapping that assigns a member of \( D_c \) to each term \((\text{var}, x, \alpha)\). Given a variable assignment \( \varphi \) into \( S \), a term \((\text{var}, x, \alpha)\), and a value \( d \in D_c \), let \( \varphi[(\text{var}, x, \alpha) \mapsto d] \) be the variable assignment \( \varphi' \) into \( S \) such that \( \varphi'((\text{var}, x, \alpha)) = d \) and \( \varphi'((\text{var}, y, \beta)) = \varphi((\text{var}, y, \beta)) \) for all terms \((\text{var}, y, \beta)\) different from \((\text{var}, x, \alpha)\). Let \( \text{var-ass}(S) \) be the collection of variable assignments into \( S \).
5.4 Valuations

A \textit{valuation} for $S$ is a possibly partial mapping $V$ from $\mathcal{E} \times \text{var-ass}(S)$ into $D_0 \cup D_b \cup \{T,F,\bot\}$ such that, for all $e \in \mathcal{E}$ and $\varphi \in \text{var-ass}(S)$, if $V_\varphi(e)$ is defined, then $V_\varphi(e) \in D_o$ if $e$ is an operator, $V_\varphi(e) \in D_b$ if $e$ is a type, $V_\varphi(e) \in D_c \cup \{\bot\}$ if $e$ is a term, and $V_\varphi(e) \in \{T,F\}$ if $e$ is a formula. (We write $V_\varphi(e)$ instead of $V(e, \varphi)$.)

Let the \textit{standard valuation} for $S$ be the valuation $V$ for $S$ defined inductively by the following statements:

1. Let $e \in \mathcal{E}$ be improper. Then $V_\varphi(e)$ is undefined.
2. Let $O = (\text{op}, s, k_1, \ldots, k_n, k_{n+1})$ be proper. Then $V_\varphi(O) = I(O)$.
3. Let $e = (\text{op-app}, O, e_1, \ldots, e_n)$ be proper where $O = (\text{op}, s, k_1, \ldots, k_n, k_{n+1})$. 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 $\text{type}[k_i]$, then
   \[ V_\varphi(e) = V_\varphi(O)(V_\varphi(e_1), \ldots, V_\varphi(e_n)) \]
   Otherwise $V_\varphi(e)$ is $D_c$ if $k_{n+1} = \text{type}$, $\bot$ if $\text{type}[k_{n+1}]$, and $F$ if $k_{n+1} = \text{formula}$.
4. Let $a = (\text{var}, x, \alpha)$ be proper. If $\varphi(a)$ is in $V_\varphi(\alpha)$, then $V_\varphi(a) = \varphi(a)$. Otherwise $V_\varphi(a) = \bot$.
5. Let $\beta = (\text{type-app}, \alpha, a)$ be proper. If $V_\varphi(a) \neq \bot$, then $V_\varphi(\beta) = V_\varphi(\alpha)[V_\varphi(a)]$. Otherwise $V_\varphi(\beta) = D_c$.
6. Let $\gamma = (\text{dep-fun-type}, (\text{var}, x, \alpha), \beta)$ be proper. Then $V_\varphi(\gamma)$ is the superclass of all $f$ in $D_f$ such that:
   a. For all sets $d$ in $V_\varphi(\alpha)$, if $f(d)$ is defined, then $f(d)$ is in $V_\varphi[(\text{var}, x, \alpha) \rightarrow d(\beta)]$.
   b. For all sets $d$ not in $V_\varphi(\alpha)$, $f(d)$ is undefined.
7. Let $b = (\text{fun-app}, f, a)$ 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 $f = (\text{fun-abs}, (\text{var}, x, \alpha), b)$ be proper. If
   \[ g = \{ (d, d') \mid d \text{ is a set in } V_\varphi(\alpha) \text{ and } d' = V_\varphi[(\text{var}, x, \alpha) \rightarrow d(\beta)] \text{ is a set} \} \]

19
is in $D_f$, then $V_\varphi(f) = g$. Otherwise $V_\varphi(f) = \bot$.

9. Let $a = (if, A, b, c)$ be proper. If $V_\varphi(A) = T$, then $V_\varphi(a) = V_\varphi(b)$. Otherwise $V_\varphi(a) = V_\varphi(c)$.

10. Let $A = (exist, (var, x, \alpha), B)$ be proper. If there is some $d$ in $V_\varphi(\alpha)$ such that $V_\varphi[(var, x, \alpha) \rightarrow d](B) = T$, then $V_\varphi(A) = T$. Otherwise, $V_\varphi(A) = F$.

11. Let $a = (def-des, (var, x, \alpha), B)$ be proper. If there is a unique $d$ in $V_\varphi(\alpha)$ such that $V_\varphi[(var, x, \alpha) \rightarrow d](B) = T$, then $V_\varphi(a) = d$. Otherwise, $V_\varphi(a) = \bot$.

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

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

14. Let $b = (eval, a, k)$ be proper. If (1) $V_\varphi(a)$ is in $E_{ty}$ and $k = type$, $V_\varphi(a)$ is in $E_{te}$ and $type[k]$, or $V_\varphi(a)$ is in $E_{fo}$ and $k = formula$; (2) the “tree” $V_\varphi(a)$ does not contain the “leaf” $H(eval)$; and (3) $V_\varphi(\hat{H}^{-1}(V_\varphi(a)))$ is in $V_\varphi(k)$ if $type[k]$, then $V_\varphi(b) = V_\varphi(\hat{H}^{-1}(V_\varphi(a)))$. Otherwise $V_\varphi(b)$ is $D_e$ if $k = type$, $\bot$ if $type[k]$, and $F$ if $k = formula$.

**Proposition 5.1** The standard valuation exists for every structure for $L$.

**Proposition 5.2** Let $V$ be the standard valuation for $S$, $e \in E$, and $\varphi$ be a variable assignment into $S$.

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 $\alpha$, then $V_\varphi(e)$ is in $V_\varphi(\alpha) \cup \{\bot\}$.

5. If $e$ is a formula, then $V_\varphi(e)$ is in $\{T,F\}$.
5.5 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 [non-denoting] in $M$ with respect to a variable assignment $\varphi$ if $V_\varphi(e)$ is defined [undefined]. A term $a$ is defined [undefined] in $M$ with respect to $\varphi$ if $V_\varphi(a)$ is in $D_c [V_\varphi(a) = \perp]$. If an expression $e$ is denoting in $M$ with respect to $\varphi$, then its value in $M$ with respect to $\varphi$ 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 $A$ of $L$ is valid, written $|= A$, if $V_\varphi(A) = T$ for all standard models $M = (S, V)$ for $L$ and all $\varphi \in \text{var-ass}(S)$. An expression $e$ is closed if, for all standard models $M = (S, V)$, $V_\varphi(e)$ does not depend on $\varphi$, then its value in $M$ with respect to $\varphi$ is $V_\varphi(e)$.

A sentence of $L$ is a formula of $L$ that is closed.

5.6 Notes concerning the Semantics of Chiron

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 that represents the syntactic structure of the expression as a tree of symbols.

2. Two variables $(\text{var}, x, \alpha)$ and $(\text{var}, x', \alpha')$ are considered to be identical only if $x$ and $x'$ are identical symbols and $\alpha$ and $\alpha'$ are identical types, that is, only if $[(\text{var}, x, \alpha)]$ and $[(\text{var}, x', \alpha')]$ have identical values. In particular, if $\alpha$ and $\alpha'$ are not identical types, then $(\text{var}, x, \alpha)$ and $(\text{var}, x, \alpha')$ are not identical variables even if $\alpha$ and $\alpha'$ have the same value with respect to every variable assignment.

3. 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 $\perp$ is also not possible. However, indirect quantification over “definable” operations, “definable” superclasses, “definable” members of $D_c \cup \{\perp\}$, or truth values can be done via variables of type $E_{\text{op}}$, $E_{\text{ty}}$, $E_{\text{te}}$, or $E_{\text{fo}}$, respectively. As in standard NBG set theory, only classes are first-class values in Chiron.

4. 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.
5. Dependent function types, function abstractions, existential quantifications, definition descriptions, and indefinite descriptions are \textit{variable binding} expressions. They bind variables in the traditional, naive way. It would be possible to use other more sophisticated variable binding mechanisms such as de Bruijn notation [1] or nominal datatypes [13, 16].

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

7. Suppose a built-in operator
\[
(\text{op}, \text{lub}, \text{type}, \text{type}, \text{type})
\]
is added to Chiron that denotes an operation that, given superclasses \(\Sigma_1\) and \(\Sigma_2\), returns a superclass that is the least upper bound of \(\Sigma_1\) and \(\Sigma_2\). Then formation rule P-Expr-8 could be modified so that a conditional term \((\text{if}, A, b, c)\) is assigned the type
\[
(\text{op-app}, (\text{op}, \text{lub}, \text{type}, \text{type}, \text{type}), \beta, \gamma)
\]
where \(\beta\) and \(\gamma\) are the types of \(b\) and \(c\), respectively.

8. A \textit{Gödel number} [7] is a number that encodes an expression. Analogously, a \textit{Gödel set} is a set that encodes an expression. Employing this terminology, the construction that represents an expression \(e \in \mathcal{E}\) is the \textit{Gödel set of } \(e\) which is denoted by \((\text{quote}, e)\). Hence, “Gödel numbering” is built into Chiron.

9. When an evaluation \(b = (\text{eval}, a, k)\) is “semantically well-formed”, the value of \(b\) is, roughly speaking, \textit{the value of the value of } \(a\).

10. An “ungrounded expression” is considered to be an “undefined” expression. For example, the value of an ungrounded formula like \([\text{LIAR}]_o\) is \(\mathsf{f}\).

11. Operators of the form \((s :: \alpha)\) are applied to an empty tuple of arguments. The value of \((s :: \alpha)\) 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 \((s :: \alpha)\) is \(v\) instead of \(o\).
A value $x \in \mathcal{E}$ is a construction that represents an expression of Chiron. We will sometimes abuse terminology and say $x \in \mathcal{E}$ is an *expression*. Similarly, we will say that $x \in \mathcal{E}_{\text{op}}$, $x \in \mathcal{E}_{\text{ty}}$, $x \in \mathcal{E}_{\text{te}}$, or $x \in \mathcal{E}_{\text{fo}}$ is an *operator*, *type*, *term*, or *formula*, respectively.

### 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 (op, set, type) ((set :: type) in compact notation) and then define its meaning as part of the definition of a standard model. The second is to construct a universally quantified formula that defines the meaning of the operator. We will use this latter approach to define several useful logical, set-theoretic, and syntactic operators.

#### 6.1 Logical Operators

1. **Conjunction**
   Operator: $(\text{and} :: \text{formula}, \text{formula}, \text{formula})$
   Definition:
   $$\forall e_1, e_2 : \mathcal{E}_{\text{fo}} \cdot (\text{and} :: \text{formula}, \text{formula}, \text{formula})([e_1], [e_2]) \equiv \neg(\neg[e_1] \lor \neg[e_2])$$
   Compact notation:
   $$(A \land B) \text{ means } (\text{and} :: \text{formula}, \text{formula}, \text{formula})(A, B).$$

2. **Implication**
   Operator: $(\text{implies} :: \text{formula}, \text{formula}, \text{formula})$
   Definition:
   $$\forall e_1, e_2 : \mathcal{E}_{\text{fo}} \cdot (\text{implies} :: \text{formula}, \text{formula}, \text{formula})([e_1], [e_2]) \equiv \neg[e_1] \lor [e_2]$$
   Compact notation:
   $$(A \supset B) \text{ means } (\text{implies} :: \text{formula}, \text{formula}, \text{formula})(A, B).$$
3. **Definedness in a Type**
Operator: (defined-in :: C, type, formula)
Definition:
\[
\forall e_1 : E_{\text{te}}, e_2 : E_{\text{ty}} \cdot (\text{defined-in} :: C, \text{type}, \text{formula})([e_1], [e_2]) \equiv
\]
\[
[e_1] = [e_2] \equiv [e_1]
\]

Compact notation:
- \((a \downarrow \alpha)\) means (defined-in :: C, type, formula)(a, \alpha).
- \((a \uparrow \alpha)\) means \(\neg(a \downarrow \alpha)\).
- \((a \downarrow)\) means \((a \downarrow C)\).
- \((a \uparrow)\) means \(\neg(a \downarrow)\).

4. **Quasi-Equality**
Operator: (quasi-equal :: C, C, formula)
Definition:
\[
\forall e_1, e_2 : E_{\text{te}} \cdot (\text{quasi-equal} :: C, C, \text{formula})([e_1], [e_2]) \equiv
\]
\[
([e_1] \downarrow \lor [e_2] \downarrow) \supset [e_1] \equiv [e_2]
\]

Compact notation:
- \((a \simeq b)\) means (quasi-equal :: C, C, formula)(a, b).
- \((a \not\simeq b)\) means \(\neg(a \simeq b)\).

5. **Type Order**
Operator: (type-le :: type, type, formula)
Definition:
\[
\forall e_1, e_2 : E_{\text{ty}} \cdot (\text{type-le} :: \text{type}, \text{type}, \text{formula})([e_1], [e_2]) \equiv
\]
\[
(\forall x : [e_1] \cdot (x \downarrow [e_2]))
\]

Compact notation:
- \((\alpha \ll \beta)\) means (type-le :: type, type, formula)(\alpha, \beta).
6. Canonical Undefined Term
Operator: (undefined :: C)
Definition:

\[(\text{undefined :: C})(\_ : C. \_ \neq x)\]

Compact notation:

\[\bot_C \text{ means } (\text{undefined :: C})(\_).\]

6.2 Set-Theoretic Operators

1. Pair
Operator: (pair :: V, V, V)
Definition:

\[\forall e_1, e_2 : \text{Ete}. (\text{pair :: V, V, V})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \simeq\]
\[\iota z : V. (\llbracket e_1 \rrbracket \downarrow v) \land (\llbracket e_2 \rrbracket \downarrow v) \land\]
\[\forall w : V. w \in z \supset (w = \llbracket e_1 \rrbracket \lor w = \llbracket e_2 \rrbracket)\]

Compact notation:

\[\{a, b\} \text{ means } (\text{pair :: V, V, V})(a, b)\]
\[\{a\} \text{ means } \{a, a\}\]

2. Ordered Pair
Operator: (ord-pair :: V, V, V)
Definition:

\[\forall e_1, e_2 : \text{Ete}. (\text{ord-pair :: V, V, V})(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \simeq\]
\[\{\{\llbracket e_1 \rrbracket\}, \{\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket\}\}\]

Compact notation:

\[\langle a, b \rangle \text{ means } (\text{ord-pair :: V, V, V})(a, b)\]
\[\langle a_1, \ldots, a_n \rangle \text{ means } \langle a_1, \langle a_2, \ldots, a_n \rangle \rangle \text{ for } n \geq 3\]
\[\[] \text{ means } \emptyset.\]
\[\llbracket a_1, \ldots, a_n \rrbracket \text{ means } \langle a_1, [a_2, \ldots, a_n] \rangle \text{ for } n \geq 1\]
3. **Subclass**  
Operator: (subclass :: C, C, formula)  
Definition:  
\[
\forall e_1, e_2 : E_t \cdot (\text{subclass :: C, C, formula})([e_1], [e_2]) \equiv \\
\forall z : V . z \in [e_1] \supset z \in [e_2]
\]
Compact notation:  
\[a \subseteq b \text{ means (subclass :: C, C, formula)}(a, b)\]  
\[a \subset b \text{ means } a \subseteq b \land a \neq b\]

4. **Empty set**  
Operator: (empty-set :: V)  
Definition:  
\[
(\text{empty-set :: V})( ) \simeq \iota y : V . \forall z : V . z \notin y
\]
Compact notation:  
\[\emptyset \text{ means (empty-set :: V)}( )\]

5. **Universal class**  
Operator: (universal-class :: C)  
Definition:  
\[
(\text{universal-class :: C})( ) \simeq \iota y : C . \forall z : V . z \in y
\]
Compact notation:  
\[U \text{ means (universal-set :: C)}( )\]

6. **Union**  
Operator: (union :: C, C, C)  
Definition:  
\[
\forall e_1, e_2 : E_t \cdot (\text{union :: C, C, C})([e_1], [e_2]) \simeq \\
\iota z : C . \forall w : V . w \in z \equiv (w \in [e_1] \lor w \in [e_2])
\]
Compact notation:  
\[a \cup b \text{ means (union :: C, C, C)}(a, b)\]
7. **Intersection**
Operator: \((\text{intersection} :: C, C, C)\)
Definition:

\[
\forall e_1, e_2 : E_{te} . (\text{intersection} :: C, C, C)([e_1], [e_2]) \simeq \\
\iota z : C . \forall w : V . w \in z \equiv (w \in [e_1] \land w \in [e_2])
\]

Compact notation:

\(a \cap b\) means \((\text{intersection} :: C, C)(a, b)\)

8. **Complement**
Operator: \((\text{compl} :: C, C)\)
Definition:

\[
\forall e : E_{te} . (\text{compl} :: C, C)([e]) \simeq \\
\iota y : C . \forall z : V . z \in y \equiv z \not\in [e]
\]

Compact notation:

\(\pi\) means \((\text{compl} :: C, C)(a)\)

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

\[
\forall e : E_{te} . (\text{head} :: V, V)([e]) \simeq \\
\iota x_1 : V . \exists x_2 : V . [e] = \langle x_1, x_2 \rangle
\]

Compact notation:

\(\text{hd}(a)\) means \((\text{head} :: V, V)(a)\)

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

\[
\forall e : E_{te} . (\text{tail} :: V, V)([e]) \simeq \\
\iota x_2 : V . \exists x_1 : V . [e] = \langle x_1, x_2 \rangle
\]
Compact notation:
\( \text{tl}(a) \) means \((\text{tail} :: V, V)(a)\)

11. **Type Product**
Operator: \((\text{type-prod} :: \text{type}, \text{type}, \text{type})\)
Definition:
\[
\forall e_1, e_2 : E_ty, \forall x : C . (x \downarrow (\text{type-prod} :: \text{type}, \text{type})([e_1][e_2])) \equiv \\
(\text{hd}(x) \downarrow [e_1]) \land (\text{tl}(x) \downarrow [e_2])
\]
Compact notation:
\((\alpha \times \beta) \) means \((\text{type-prod} :: \text{type}, \text{type}, \text{prod})(\alpha, \beta)\).

6.3 **Syntactic Operators**
1. **Proper Expression Checker**
Operator: \((\text{is-p-expr} :: E, \text{formula})\)
Definition:
\[
\forall e : E . (\text{is-p-expr} :: E, \text{formula})(e) \equiv \\
(e \downarrow E_{op}) \lor (e \downarrow E_{ty}) \lor (e \downarrow E_{te}) \lor (e \downarrow E_{fo})
\]
Compact notation:
\(\text{is-p-expr}(e) \) means \((\text{is-p-expr} :: E, \text{formula})(e)\)

2. **First Component Selector for a Proper Expression**
Operator: \((\text{1st-comp} :: E, E)\)
Definition:
\[
\forall e : E . (\text{1st-comp} :: E, E)(e) \simeq \\
\text{if(is-p-expr}(e), \text{hd}(\text{tl}(e)), \bot_C)
\]
Compact notation:
\(\text{1st-comp}(e) \) means \((\text{1st-comp} :: E, E)(e)\)

Note: The second, third, fourth, \ldots component selectors for proper expressions are defined in a similar way: 2nd-comp, 3rd-comp, 4th-comp, \ldots
3. **Operator Checker**  
Operator: \((\text{is-op} :: \text{E, formula})\)  
Definition:

\[
\forall e : \text{E} \cdot (\text{is-op} :: \text{E, formula})(e) \equiv 
\text{is-p-expr}(e) \land \text{hd}(e) = [\text{op}]
\]

Compact notation:

\(\text{is-op}(e)\) means \((\text{is-op} :: \text{E, formula})(e)\)

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

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

\[
\forall e : \text{E} \cdot (\text{is-disj} :: \text{E, formula})(e) \equiv 
\text{is-op-app}(e) \land 1\text{-st-comp}(e) = [\text{or}]
\]

Compact notation:

\(\text{is-disj}(e)\) means \((\text{is-disj} :: \text{E, formula})(e)\)

Note: Checkers for other kinds of operator applications are defined in a similar way: \(\text{is-in}, \text{is-type-eqn}, \text{is-term-eqn}, \text{is-formula-eqn}, \text{is-neg}, \text{is-conj}, \text{is-impl}, \text{is-defined-in}, \text{is-quasi-eqn}, \text{is-type-ineq}, \text{is-undefined-term}, \text{is-pair}, \text{is-ord-pair}, \text{is-subclass}, \text{is-empty-set}, \text{is-union}, \text{is-intersec}, \text{is-compl}, \text{is-hd}, \text{is-tl}, \text{is-type-prod}\).

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

\[
\forall e : \text{E} \cdot (\text{1st-arg} :: \text{E, E})(e) \simeq 
\text{if}(\text{is-op-app}(e), 2\text{nd-comp}(e), \bot_C)
\]
Compact notation:

\[ \text{1st-arg}(e) \text{ means (1st-arg :: E,E)}(e) \]

Note: The second, third, fourth, \ldots argument selectors for operator application are defined in a similar way: \(\text{2nd-arg}, \text{3rd-arg}, \text{4th-arg}, \ldots\).

6. **Variable Binder Checker**

Operator: \((\text{is-binder :: E, formula})\)

Definition:

\[ \forall e : E. \ (\text{is-binder :: E, formula})(e) \equiv \]
\[ \text{is-dep-fun-type}(e) \lor \text{is-fun-abs}(e) \lor \text{is-exist}(e) \lor \]
\[ \text{is-def-des}(e) \lor \text{is-indef-des}(e) \]

Compact notation:

\[ \text{is-binder}(e) \text{ means } (\text{is-binder :: E, formula})(e) \]

7. **Variable Selector for Variable Binder**

Operator: \((\text{binder-var :: E,E})\)

Definition:

\[ \forall e : E. \ (\text{binder-var :: E,E})(e) \simeq \]
\[ \text{if} (\text{is-binder}(e), \text{1st-comp}(e), \bot_C) \]

Compact notation:

\[ \text{binder-var}(e) \text{ means } (\text{binder-var :: E,E})(e) \]

Note: Name and body selectors for a variable binder named \text{binder-name} and \text{binder-body} are defined in a similar way.

8. **Function Redex Checker**

Operator: \((\text{is-fun-redex :: E, formula})\)

Definition:

\[ \forall e : E. \ (\text{is-fun-redex :: E, formula})(e) \equiv \]
\[ \text{is-fun-app}(e) \land \text{is-fun-abs(1st-comp(e))} \]
Compact notation:

\[ \text{is-fun-redex}(e) \text{ means } (\text{is-fun-redex :: E, formula})(e) \]

Note: A dependent function type redex checker named \text{is-fun-type-redex} is defined in a similar way.

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

\[ \forall e : E. \ (\text{is-redex :: E, formula})(e) \equiv \text{is-fun-type-redex}(e) \lor \text{is-fun-redex}(e) \]

Compact notation:

\[ \text{is-redex}(e) \text{ means } (\text{is-redex :: E, formula})(e) \]

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

\[ \forall e : E. \ (\text{redex-var :: E, E})(e) \simeq \]
\[ \text{if}(\text{is-redex}(e), \text{binder-var}(1\text{-st-comp}(e)), \bot_C) \]

Compact notation:

\[ \text{redex-var}(e) \text{ means } (\text{redex-var :: E, E})(e) \]

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

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

\[ \forall e_1, e_2 : E. \ (\text{free-in :: E, E, formula})(e_1, e_2) \equiv \]
\[ \text{is-var}(e_1) \land \]
\[ ((\text{is-var}(e_2) \land e_1 = e_2) \lor \]
\[ (\text{is-binder}(e_2) \land e_1 \neq \text{binder-var}(e_2) \land \]
\[ \text{free-in}(e_1, \text{binder-body}(e_2))) \lor \]
\[ ((\text{is-op}(e_2) \lor \text{is-op-app}(e_2) \lor \text{is-type-app}(e_2) \lor \]
\[ \text{is-fun-app}(e_2) \lor \text{is-if}(e_2) \lor \text{is-eval}(e_2)) \land \]
\[ \text{free-in-list}(e_1, \text{tl}(e_2))) \]
Compact notation:
\[ \text{free-in}(e_1, e_2) \text{ means } (\text{free-in :: } E, E, \text{formula})(e_1, e_2) \]

Auxiliary operator: \( (\text{free-in-list :: } E, E, \text{formula}) \)

Definition:
\[
\forall e_1, e_2 : E . (\text{free-in-list :: } E, E, \text{formula})(e_1, e_2) \equiv \\
\neg \text{is-empty-set}(e_2) \land \\
(\text{free-in}(e_1, \text{hd}(e_2)) \lor \text{free-in-list}(e_1, \text{tl}(e_2)))
\]

Compact notation:
\[ \text{free-in-list}(e_1, e_2) \text{ means } (\text{free-in-list :: } E, E, \text{formula})(e_1, e_2) \]

12. **Closed Proper Expression**
Operator: \( (\text{closed :: } E, \text{formula}) \)

Definition:
\[
\forall e : E . (\text{closed :: } E, \text{formula})(e) \equiv \\
\text{is-p-expr}(e) \land (\forall e' : E . \text{is-var}(e') \supset \neg \text{free-in}(e', e))
\]

Compact notation:
\[ \text{closed}(e) \text{ means } (\text{closed :: } E, \text{formula})(e) \]

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

Definition:
\[
\forall e_1, e_2, e_3 : E . (\text{free-for :: } E, E, E, \text{formula})(e_1, e_2, e_3) \equiv \\
(e_1 \downarrow E_{te}) \land \text{is-var}(e_2) \land \\
\text{is-var}(e_3) \lor \\
\neg \text{free-in}(e_2, e_3) \lor \\
(\neg \text{free-in}(\text{binder-var}(e_3), e_1) \land \\
\text{free-for}(e_1, e_2, \text{binder-body}(e_3)))) \lor \\
((\text{is-op}(e_3) \lor \text{is-op-app}(e_3) \lor \\
\text{is-type-app}(e_3) \lor \text{is-fun-app}(e_3) \lor \\
\text{is-if}(e_3) \lor \text{is-eval}(e_3)) \land \\
\text{free-for-list}(e_1, e_2, \text{tl}(e_3))))
\]
Compact notation:

\[
\text{free-for}(e_1, e_2, e_3) \text{ means } (\text{free-for} :: E, E, \text{formula})(e_1, e_2, e_3)
\]

Auxiliary operator: (free-for-list :: E, E, formula)
Definition:

\[
\forall e_1, e_2, e_3 : E . \ (\text{free-for-list} :: E, E, \text{formula})(e) \equiv
\begin{align*}
\text{is-empty-set}(e_3) & \lor \\
(\text{free-for}(e_1, e_2, \text{hd}(e_3)) & \land \text{free-for-list}(e_1, e_2, \text{tl}(e_3)))
\end{align*}
\]

Compact notation:

\[
\text{free-for-list}(e_1, e_2, e_3) \text{ means } (\text{free-for-list} :: E, E, \text{formula})(e_1, e_2, e_3)
\]

14. Substitution for a Variable in an Expression
Operator: (sub :: E, E, E)
Definition:

\[
\forall e_1, e_2, e_3 : E . \ (\text{sub} :: E, E, E)(e_1, e_2, e_3) \simeq
\begin{align*}
\text{if}(e_1 \downarrow E_{e_0}) & \land \text{is-var}(e_2), \\
\text{if}(\text{is-var}(e_3) & \land e_2 = e_3, \\
\text{sub}(e_1, e_2, \text{binder-body}(e_3))), \\
\text{if}(\text{is-op}(e_3) & \lor \text{is-op-app}(e_3) \lor \\
\text{is-type-app}(e_3) & \lor \text{is-fun-app}(e_3) \lor \\
\text{is-if}(e_3) & \lor \text{is-eval}(e_3)), \\
[\text{hd}(e_3), \text{sub-list}(e_1, e_2, \text{tl}(e_3))], \\
e_3))
\end{align*}
\]

Compact notation:

\[
\text{sub}(e_1, e_2, e_3) \text{ means } (\text{sub} :: E, E, E)(e_1, e_2, e_3)
\]
Auxiliary operator: \((\text{sub-list} :: \mathbb{E}, \mathbb{E}, \mathbb{E})\)

Definition:

\[
\forall e_1, e_2, e_3 : \mathbb{E}. \ (\text{sub-list} :: \mathbb{E}, \mathbb{E}, \mathbb{E})(e) \simeq \\
\text{if}(\text{is-empty-set}(e_3), \\
\emptyset, \\
[\text{sub}(e_1, e_2, \text{hd}(e_3)), \text{sub-list}(e_1, e_2, \text{tl}(e_3))])
\]

Compact notation:

\[
\text{sub-list}(e_1, e_2, e_3) \text{ means } (\text{sub-list} :: \mathbb{E}, \mathbb{E}, \mathbb{E})(e_1, e_2, e_3)
\]

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

\[A \lor \neg A\]

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

\[
\forall e : \mathbb{E}_{fo}. \ [e] \lor \neg [e].
\]

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:

\[
\forall e_1, e_2 : \mathbb{E}_{fo}.
\]

\[
([e_1] \land [e_2] \land \text{is-impl}(e_2) \land e_1 = \text{1st-arg}(e_2)) \supset [\text{2nd-arg}(e_2)]_{fo}
\]

Deduction and computation rules are naturally represented as transformers [6], 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:

\[
\lambda x : \mathbb{E}_{fo} \times \mathbb{E}_{fo}.
\]

\[
\text{if}(\text{is-impl}(\text{tl}(x)) \land \text{hd}(x) = \text{1st-arg}(\text{tl}(x)), \text{2nd-arg}(\text{tl}(x)), \bot_{\mathcal{C}})
\]
Let \((\text{modus-ponens} :: \Lambda x : E_{fo} \times E_{fo} \cdot E_{fo})\) be an individual constant that is defined to be this function abstraction, and let \(mp\) mean
\[
(\text{modus-ponens} :: \Lambda x : E_{fo} \times E_{fo} \cdot E_{fo})().
\]
Then the following formula is an alternate expression of the law of modus ponens:
\[
\forall e_1, e_2 : E_{fo} \cdot ([e_1] \land [e_2] \land mp(\langle e_1, e_2 \rangle) \downarrow) \supset [mp(\langle e_1, e_2 \rangle)]
\]

### 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 \((x, \alpha, b, a)\), 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 functions is:
\[
\forall e : E_{te} \cdot
(is-redex(e) \land free-for(redex-arg(e), redex-var(e), redex-body(e)))
\supset
[e] \simeq [\text{sub(redex-arg(e), redex-var(e), redex-body(e))}]_{te}
\]
The beta reduction law for functions is applied to an application \(a\) of a function abstraction by instantiating the variable \((e : E_{te})\) with the quotation \([a]\).

### 7.4 Liar Paradox

We will formalize in Chiron the liar paradox mentioned in section 5.1. Assume that \(\text{nat}\) is a type and \(0, 1, 2, \ldots\) denote terms such that \(\text{nat}\) denotes an infinite set \(\{0, 1, 2, \ldots\}\). Assume also that \(\text{num}\) is a type that denotes the set \(\{[0], [1], [2], \ldots\}\). And finally assume that \(\text{enum}\) is a term that denotes a
function which is a bijection from \{[0], [1], [2], \ldots\} to the set \(F\) of all functions of type \((A x : E . E)\) that are definable by a closed function abstraction of the form \((\lambda x : E . b)\).

The following function abstraction denotes some \(f \in F\):

\[
\lambda x : E . \\
[\text{[op-app]}], \\
[\text{[op]}, [\text{not}], [\text{formula}], [\text{formula}]], \\
[\text{[eval]}], \\
[\text{[fun-app]}, [\text{fun-app}], [\text{enum}], x, x], \\
[\text{formula}]]
\]

Therefore, for some \(i\) of type \text{nat}, \text{enum}([i]) = f. Then

\[
\text{enum}([i])([i]) = f([i]) = [-[\text{enum}([i])([i])]]_{fo}
\]

Therefore, if LIAR is the term \text{enum}([i])([i]), then

\[
\text{LIAR} = [-[\text{LIAR}]]_{fo}.
\]

8 Relationship to NBG

We show in this section that there a faithful interpretation of NBG set theory in Chiron. Loosely speaking, this means Chiron is a conservative extension of NBG. It 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 \(V\) of variables, a unary predicate symbol \(V\), binary predicates symbols \(=\) and \(\in\), and some complete set of logical connectives (say \(\neg, \lor, \exists\)). Assume \(V \subseteq S\), i.e., each variable of \(L_{nbg}\) is a symbol of Chiron. Let \(\mathcal{F}_{nbg}\) 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. A variable assignment into \(N\) is a mapping that assigns a member of \(D^N\) to each variable \(x \in V\). Let \text{var-ass}(N) be the collection of variable assignments in \(N\). The valuation for \(N\) is a total mapping \(W^N\) from the \(\mathcal{F}_{nbg} \times \text{var-ass}(N)\) to the set \{\(t, f\)\} of truth values. A formula \(A\) of \(L_{nbg}\) is valid, written \(\models_{nbg} A\), if \(W^N_A = t\) for all models \(N\) of NBG and all \(\varphi \in \text{var-ass}(N)\).\(^\dagger\)

\(^\dagger\)As above, we write \(W^N(A, \varphi)\) as \(W^N_{\varphi}(A)\).
Let $L$ be any language of Chiron. Suppose $\Phi$ 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 $L$. $\Phi$ is a translation from $nbg$ to Chiron if $\Phi$ maps the sentences of $L_{nbg}$ to sentences of $L$. $\Phi$ is an interpretation of $nbg$ in Chiron if $\Phi$ maps the sentences of $nbg$ to sentences of $L$. $\Phi$ is a faithful interpretation of $nbg$ in Chiron if $\Phi$ is an interpretation of $nbg$ in Chiron and, for all sentences $A$ of $L_{nbg}$, $\models_{nbg} A$ implies $\models \Phi(A)$. That is, $\Phi$ is an interpretation of $nbg$ in Chiron if $\Phi$ is a meaning-preserving translation from $nbg$ to Chiron. $\Phi$ is a faithful interpretation of $nbg$ in Chiron if $\Phi$ is a conservative interpretation in the sense that Chiron is a conservative extension of the image of $nbg$ under $\Phi$.

Let $\Phi$ 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 $x \in V$, then $\Phi(x) = (x : C)$.
2. If $V(x)$ is a formula of $L_{nbg}$, then $\Phi(V(x)) = (\Phi(x) \uparrow V)$.
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 $(A \lor B)$ is a formula of $L_{nbg}$, then $\Phi((A \lor B)) = (\Phi(A) \lor \Phi(B))$.
7. If $(\exists x . A)$ is a formula of $L_{nbg}$, then $\Phi((\exists x . A)) = (\exists x : C . \Phi(A))$.

**Lemma 8.1** 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

$$S = (D_v, D_c, D_s, D_t, D_o, \in, t, f, \bot, \xi, H, I),$$

such that $(D^N, \in^N)$ is identical to the prestructure $(D_c, \in)$.

1. For all $d \in D^N$, $V^N(d)$ iff $d$ is in $I(V)$.
2. For all $d, d' \in D^N$, $d =^N d'$ iff $d I(=) d'$.
3. For all $x \in V$ and $\varphi \in \text{var-assign}(N)$, $W^N_\varphi(x) = V^N_\varphi(\Phi(x))$ where $\varphi' \in \text{var-assign}(S)$ such that $\varphi'(x : C) = \varphi(x)$.
4. For all formulas $A$ of $L_{nbg}$ and $\varphi \in \text{var-assign}(N)$, $W^N_\varphi(A) = V^N_\varphi(\Phi(A))$ where $\varphi \in \text{var-assign}(S)$ such that $\varphi'(x : C) = \varphi(x)$ for all $x \in V$.  

37
Proof  Clauses 1–3 are obvious. Clause 4 is easily proved by induction on the structure of the formulas of $L_{\text{nbg}}$ since the logical connectives $\neg, \vee, \exists$ of $L_{\text{nbg}}$ have the same meanings as the negation operator, the disjunction operator, and existential quantification, respectively, in Chiron. □

**Theorem 8.2** For all sentences $A$ of $L_{\text{nbg}}$, 

$$\models_{\text{NBG}} A \iff \models \Phi(A).$$

*That is, $\Phi$ 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

$$S = (D_v, D_c, D_s, D_f, D_o, \in, t, f, \bot, \xi, H, I),$$

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

$$S = (D_v, D_c, D_s, D_f, D_o, \in, t, f, \bot, \xi, H, I),$$

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 4 of Lemma 8.1. □

9  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, 6].
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.

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 [10]. 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 [9], while the second defines the values gaps according to a valuation scheme based on B. van Fraassen’s notion of a supervaluation [17] that Kripke describes in [10, p. 711].

Valuations

The notion of a valuation for a structure was defined in section 5.4. Fix a structure $S$ for $L$. Let $\text{val}(S)$ be the collection of valuations for $S$. Given $U, V \in \text{val}(S)$, $U$ is a subvaluation of $V$, written $U \subseteq V$, if, for all $e \in E$ and $\varphi \in \text{var-ass}(S)$, $U\varphi(e)$ is defined implies $V\varphi(e)$. A valuation functional for $S$ is a mapping from $\text{val}(S)$ into $\text{val}(S)$. Let $\Psi$ be a valuation functional for $S$. A fixed point of $\Psi$ is a $V \in \text{val}(S)$ such that $\Psi(V) = V$. $\Psi$ is monotone if $U \subseteq V$ implies $\Psi(U) \subseteq \Psi(V)$ for all $U, V \in \text{val}(S)$.

Theorem 9.1 Let $\Psi$ be a monotone valuation functional for $S$. Then $\Psi$ has a fixed point.

Proof The construction of a fixed point of $\Psi$ is similar to the construction of the fixed point Kripke gives in [10, pp. 703–705]. □

$\Psi^S_1$ is the valuation functional for $S$ defined by the following rules where $V \in \text{val}(S)$ and $V' = \Psi^S_1(V)$. 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 $V$. $\Psi^S_1$ 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 $e \in E$ be improper. Then $V'_\varphi(e)$ is undefined.

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

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

\[ O = (\text{op}, s, k_1, \ldots, k_n, k_{n+1}). \]

a. Let $V'_\varphi(k_i)$ be defined for all $i$ with $1 \leq i \leq n$ and 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 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 $k = \text{type}$, $\bot$ if type[$k$], and $f$ if $k = \text{formula}$.

b. Let $V'_\varphi(k_i)$ be undefined for some $i$ with $1 \leq i \leq n$ and 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 $a = (\text{var}, x, \alpha)$ be proper.

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

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

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

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

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

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

a. Let $V'_\varphi(\alpha)$ be defined, and let $V'_{\varphi[(\text{var}, x, \alpha) \rightarrow d]}(\beta)$ be defined for all sets $d$ in $V'_\varphi(\alpha)$. Then $V'_\varphi(\gamma)$ is the superclass of all $f$ in $D_t$ such that:

i. For all sets $d$ in $V'_\varphi(\alpha)$, if $f(d)$ is defined, then $f(d)$ is in $V'_{\varphi[(\text{var}, x, \alpha) \rightarrow d]}(\beta)$.

ii. For all sets $d$ not in $V'_\varphi(\alpha)$, $f(d)$ is undefined.
10. Let \( a \) be undefined, or let \( V'_{\varphi}(a) \) be undefined for some set \( d \) in \( V''_{\varphi}(\alpha) \). Then \( V'_{\varphi}(\gamma) \) is undefined.

7. Let \( b = (\text{fun-app}, f, a) \) 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 \( f = (\text{fun-abs}, (\text{var}, x, \alpha), b) \) be proper.
   a. Let \( V''_{\varphi}(\alpha) \) be defined, and let \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](b) \) be defined for all sets \( d \) in \( V''_{\varphi}(\alpha) \). If
     \[
     g = \{ (d, d') \mid d \text{ is a set in } V''_{\varphi}(\alpha) \text{ and } d' = V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](b) \text{ is a set} \}
     \]
     is in \( D_\Gamma \), then \( V''_{\varphi}(f) = g \). Otherwise \( V''_{\varphi}(f) = \bot \).
   b. Let \( V''_{\varphi}(\alpha) \) be undefined, or let \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](b) \) be undefined for some set \( d \) in \( V''_{\varphi}(\alpha) \). Then \( V''_{\varphi}(f) \) is undefined.

9. Let \( a = (\text{if}, A, b, c) \) be proper.
   a. Let \( V''_{\varphi}(A), V''_{\varphi}(b), V''_{\varphi}(c) \) be defined. If \( V''_{\varphi}(A) = \top \), 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 \( A = (\text{exist}, (\text{var}, x, \alpha), B) \) be proper.
    a. Let \( V''_{\varphi}(\alpha) \) be defined, and let \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](B) \) be defined for all \( d \) in \( V''_{\varphi}(\alpha) \). If there is some \( d \) in \( V''_{\varphi}(\alpha) \) such that \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](B) = \top \), then \( V''_{\varphi}(A) = \top \). Otherwise, \( V''_{\varphi}(A) = \bot \).
    b. Let \( V''_{\varphi}(\alpha) \) be undefined, or let \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](B) \) be undefined for some \( d \) in \( V''_{\varphi}(\alpha) \). Then \( V''_{\varphi}(A) \) is undefined.

11. Let \( a = (\text{def-des}, (\text{var}, x, \alpha), B) \) be proper.
    a. Let \( V''_{\varphi}(\alpha) \) be defined, and let \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](B) \) be defined for all \( d \) in \( V''_{\varphi}(\alpha) \). If there is a unique \( d \) in \( V''_{\varphi}(\alpha) \) such that \( V''_{\varphi}[\text{var}, x, \alpha)\rightarrow_d](B) = \top \), then \( V''_{\varphi}(a) = d \). Otherwise, \( V''_{\varphi}(a) = \bot \).
b. Let \( V'_e(\alpha) \) be undefined, or let \( V'_{\phi[[\text{var},x,\alpha] \rightarrow d]}(B) \) be undefined for some \( d \) in \( V'_\phi(\alpha) \). Then \( V'_\phi(a) \) is undefined.

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

a. Let \( V'_e(\alpha) \) be defined, and let \( \mathcal{V}_{\phi[[\text{var},x,\alpha] \rightarrow d]}(B) \) be defined for all \( d \) in \( V'_\phi(\alpha) \). If there is some \( d \) in \( V'_\phi(\alpha) \) such that \( V'_{\phi[[\text{var},x,\alpha] \rightarrow d]}(B) = \top \), then \( V'_\phi(a) = \xi(\Sigma) \) where \( \Sigma \) is the superclass of all \( d \) in \( V'_\phi(\alpha) \) such that \( V'_{\phi[[\text{var},x,\alpha] \rightarrow d]}(B) = \top \). Otherwise, \( V'_\phi(a) = \bot \).

b. Let \( V'_e(\alpha) \) be undefined, or let \( \mathcal{V}_{\phi[[\text{var},x,\alpha] \rightarrow d]}(B) \) be undefined for some \( d \) in \( V'_\phi(\alpha) \). Then \( V'_\phi(a) \) is undefined.

13. Let \( a = (\text{quote}, e) \) be proper. Then \( V'_\phi(a) = \hat{H}(e) \).

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

a. Let \( V'_e(a) \) be defined and \( V'_e(k) \) be defined if \( \text{type}[k] \).

i. Let \( \mathcal{V}_{\phi[[\text{type},e]]}(a) \) be in \( E_{\text{ty}} \) and \( k = \text{type}, V'_e(a) \) be in \( E_{\text{te}} \) and \( \text{type}[k], \) or \( V'_e(a) \) be in \( E_{\text{fo}} \) and \( k = \text{formula} \).

A. Let \( V'_e(\hat{H}^{-1}(V'_e(a))) \) be defined. If \( k \in \{\text{type}, \text{formula}\} \) or \( \text{type}[k] \) and \( V'_e(\hat{H}^{-1}(V'_e(a))) \) is in \( V'_e(k) \), then \( V'_e(b) = V'_e(\hat{H}^{-1}(V'_e(a))) \). Otherwise \( V'_e(b) \) is \( \bot \).

B. Let \( V'_e(\hat{H}^{-1}(V'_e(a))) \) be undefined. Then \( V'_e(b) \) is undefined.

ii. Let \( V'_e(a) \) not be in \( E_{\text{ty}} \) or \( k \neq \text{type}, V'_e(a) \) not be in \( E_{\text{te}} \) or not \( \text{type}[k] \), and \( V'_e(a) \) not be in \( E_{\text{fo}} \) or \( k \neq \text{formula} \). Then \( V'_e(b) \) is \( D_c \) if \( k = \text{type}, \bot \) if \( \text{type}[k] \), and \( F \) if \( k = \text{formula} \).

b. Let \( V'_e(a) \) be undefined or \( V'_e(k) \) be undefined if \( \text{type}[k] \). Then \( V'_e(b) \) is undefined.

**Lemma 9.2** \( \Psi^S \) is monotone.

**Proof** Let \( U, V \in \text{val}(S) \) such that \( U \subseteq V \). Assume \( U' \) and \( V' \) mean \( (\Psi^S(U))_\phi \) and \( (\Psi^S(V))_\phi \), respectively. We must show that, for all \( e \in \mathcal{E} \) and \( \phi \in \text{var-ass}(S) \), if \( U'_\phi(e) \) is defined, then \( U'_\phi(e) = V'_\phi(e) \). Our proof will be by induction on the number of symbols in \( e \).

There are three cases:

1. \( e \) is improper. Then \( U'_\phi(e) \) is undefined by the definition of \( \Psi^S \).
2. \( e = (\text{eval}, a, k) \) 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 \( \Psi^S_1 \), 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 \( U \subseteq V \), \( U'_\varphi(e_1) = V'_\varphi(e_2) \). Hence, \( U'_\varphi(e) = V'_\varphi(e) \).

b. \( U'_\varphi(e) \) and \( V'_\varphi(e) \) both equal \( D \) if \( k = \text{type} \), \( \bot \) if \( k = \text{type}[k] \), and \( F \) if \( k = \text{formula} \). 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, \( U'_\varphi(e) = V'_\varphi(e) \).

\( \square \)

**Corollary 9.3** \( \Psi^S_1 \) has a fixed point.

**Proof** By Lemma 9.2, \( \Psi^S_1 \) is monotone. Therefore, by Theorem 9.1, \( \Psi^S_1 \) has a fixed point. \( \square \)

\( \Psi^S_2 \) is the valuation functional for \( S \) defined by the following three rules where \( V \in \text{val}(S) \) and \( V' = \Psi^S_2(V) \). \( \Psi^S_2 \) defines value gaps according to the supervaluation scheme.

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

2. Let \( p\text{-expr}_L[e] \) such that \( e \) is not an evaluation. If there is a value \( d \) such that, for all total valuations \( V^* \) with \( V \sqsubseteq V^* \), \( (\Psi^S_1(V^*))_\varphi(e) = d \), then \( V'_\varphi(e) = d \). Otherwise \( V'_\varphi(e) \) is undefined.

3. Let \( p\text{-expr}_L[b] \) with \( b = (\text{eval}, a, k) \). This rule is exactly the same as the \( \Psi^S_1 \) rule for evaluations.

**Lemma 9.4** \( \Psi^S_2 \) is monotone.

**Proof** The proof is exactly the same as the proof of Lemma 9.2 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, $U'_\varphi(e) = V'_\varphi(e)$.

$\square$

**Corollary 9.5** $\Psi^S_2$ has a fixed point.

**Proof** By Lemma 9.4, $\Psi^S_2$ is monotone. Therefore, by Theorem 9.1, $\Psi^S_2$ has a fixed point. $\square$

**Models**

The valuation functionals $\Psi^S_1$ and $\Psi^S_2$ 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 $\Psi^S_1$. 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 $\Psi^S_2$.

**Theorem 9.6** 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 9.3, $\Psi^S_1$ has a fixed point $V_1$. Similarly, by Corollary 9.5, $\Psi^S_2$ 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 $L$. $\square$

The weak Kleene semantics defined by $\Psi^S_1$ is “strict” in the sense that, if any proper subexpression $e$ of a proper expression $e'$ is non-denoting, then $e'$ itself is non-denoting. The supervaluation semantics defined by $\Psi^S_2$ is not strict in this sense. For example, the value of an application of the operator $(op, or, formula, formula, formula)$ to a pair of formulas $(A, B)$ is $\top$ if the value of $A$ is $\top$ and $B$ is non-denoting or visa versa.
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 $O = (s :: \text{formula}, \text{formula})$ 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 $U'_\varphi(O(e)) = T \neq F = V'_\varphi(O(e))$. 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:

$$\forall e : \text{Efo} . \llbracket e \rrbracket \lor \lnot \llbracket e \rrbracket$$

This formula is false in the weak Kleene semantics because, if $e$ represents a nondenoting formula, then $\llbracket e \rrbracket \lor \lnot \llbracket e \rrbracket$ 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 $\llbracket e \rrbracket \lor \lnot \llbracket e \rrbracket$ is true because $\llbracket e \rrbracket \lor \lnot \llbracket e \rrbracket$ is true no matter what value is assigned to $\llbracket e \rrbracket$. 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 $\llbracket e \rrbracket$ or $\lnot \llbracket e \rrbracket$.

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] 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.

[2] W. M. Farmer. Chiron: A multi-paradigm logic. *Studies in Logic, Grammar and Rhetoric*. Special issue: R. Matuszewski, P. Rudnicki, and A. Zalewska, eds, *From Insight to Proof*, forthcoming.

[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 and M. von Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.

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

[8] 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.

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

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

[11] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, fourth edition, 1997.
[12] I. L. Novak. A construction for models of consistent systems. *Fundamenta Mathematicae*, 37:87–110, 1950.

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

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

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

[16] 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.

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