Chiron: A Multi-Paradigm Logic

William M. Farmer

McMaster University
Hamilton, Ontario, Canada
wmfarmer@mcmaster.ca

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. It supports several reasoning paradigms by integrating NBG set theory with elements of type theory, a scheme for handling undefinedness, and a facility for reasoning about the syntax of expressions. This paper gives a quick, informal presentation of the syntax and semantics of Chiron and then discusses some of the benefits Chiron provides as a multi-paradigm logic.

1 Introduction

One of the great challenges of the information age is to mechanize mathematics. Mechanizing mathematics does not mean replacing mathematicians with machines. It means building software systems to help mathematics practitioners—engineers, scientists, mathematicians, etc.—to do mathematics. The principal objective of *mechanized mathematics* is to develop widely accessible *mechanized mathematics systems* (MMSs) that support various parts of the mathematics process. Contemporary MMSs include computer theorem proving systems such as Coq [5], Isabelle [23], and PVS [22], computer algebra systems such as Axiom [18], Maple [3], and Mathematica [27].

A key component of an MMS is its underlying logic. 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.[1] (A logic may also have a proof system, but it is not a required constituent.) By this definition, a theory in a logic—such as Zermelo-Fraenkel (ZF) set theory or von-Neumann-Bernays-Gödel (NBG) set theory in first-order logic—is itself a logic. Of course, a predicate logic—such as first-order logic or simple type theory (classical higher-order logic) [10]—is also a logic.

A practical, general-purpose MMS needs a well-designed logic that is both theoretically expressive and practically expressive. 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. Traditional "off-the-shelf" logics,

[1] The underlying logics of contemporary computer algebra systems have a formal syntax but usually lack a precise semantics.

ISBN 978-83-7431-128-1 ISSN 0860–150X

like those we mentioned in the previous paragraph, lack the practical expressivity that it needed for an MMS. This is because they are designed to be used *in theory*, not *in practice*.

For example, ZF set theory has very high theoretical expressivity. Indeed most mathematicians believe that essentially all of mathematics can be formalized in ZF—at least in theory. However, ZF does not contain an operator for forming a term that denotes the application of a set f representing a function to a set a that represents an argument to f. Moreover, even if such an application operator were added to ZF, there is no special mechanism for handling "undefined" applications. As a result, statements involving functions and undefinedness are much more verbose and indirect when expressed in ZF than they need to be, and reasoning about functions and undefinedness is usually performed in the metalogic of ZF instead of in ZF itself. (For examples of this kind and other kinds of reasoning that, for the sake of convenience, is usually done in the metalogic of ZF, see K. Kunen's *Set Theory* [19].)

Chiron is a derivative of NBG set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. It is designed to have both high theoretical and high practical expressivity. As a derivative of NBG, it has the same theoretical expressivity as the ZF and NBG set theories. However, Chiron has a much higher level of practical expressivity than traditional logics. It achieves this in large part by integrating several reasoning paradigms.

A *reasoning paradigm* is a style of reasoning analogous to a *programming paradigm* [25], a style of computer programming. A reasoning paradigm includes such things as background assumptions about semantics, favored modes of expression, and sanctioned reasoning rules. Just as programming languages are designed to support certain programming paradigms, logics are designed to support certain reasoning paradigms. Here is a short list of some reasoning paradigms that are especially relevant to mechanized mathematics:

1. *Classical.* The classical paradigm assumes that formulas have only two possible truth values: true and false. Ideas are expressed using the usual machinery of predicate logic: terms that denote values, predicates applied to terms, the standard propositional connectives, and the existential and universal quantifiers. Reasoning is performed using the standard laws of predicate logic such as modus ponens and universal instantiation including nonconstructive principles such as the law of excluded middle. The classical paradigm is dominant in mathematical practice. Most traditional logics—including first-order logic, simple type theory, ZF, and NBG—support it.
2. *Constructive.* On the surface, the constructive paradigm is very similar to the classical paradigm. However, the underlying semantics is quite different and only constructive principles of reasoning are permitted. The constructive paradigm is not widely followed in mathematical practice. However, the underlying logics of several computer theorem proving systems support the constructive paradigm. For example, the logic of Coq, the Calculus of Inductive Constructions [2, 6], supports this paradigm.
3. *Assured Definedness.* The assured definedness paradigm is based on the assumption that every term is defined and thus denotes some value. As a result,

definedness checking is unnecessary and all functions are considered to be total. Nearly all logics used for mechanized mathematics support the assured definedness paradigm including first-order logic, simple type theory, ZF, and NBG. However, this paradigm is not commonly employed in mathematical practice.

4. *Permitted Undefinedness.* The converse of assured definedness, the permitted undefinedness paradigm allows terms to be undefined and thus to be nondenoting. This paradigm is very widely used in mathematical practice. Although traditional logics do not support this reasoning paradigm, we have shown that, if a traditional logic is modified slightly, it can support this paradigm [8, 9, 11, 13].
5. *Set theory.* The set theory paradigm reduces all mathematical reasoning to reasoning about sets. It is deeply entrenched in mathematical practice. ZF and NBG are two of the most popular set-theoretic logics that support this paradigm. They are both based on the same intuitive model of the iterated hierarchy of sets. The key difference between them is that ZF does not admit proper classes, while NBG does. (Proper classes are collections of sets that are too large to be sets themselves. They include useful entities like the class of all sets and the cardinality function.)
6. *Type theory.* The type theory paradigm uses a hierarchy of *types* to classify expressions and their values. Most type theories provide strong support for reasoning with functions. Type theories—both classical and constructive—are popular logics for mechanized mathematics systems. In particular, several leading computer theorem proving systems [1, 14, 17, 20, 22, 23] are based Church's type theory [4], a version of simple type theory which employs lambda-notation. Type theories are seldom used in mathematical practice outside of the computing field.
7. *Formalized Syntax.* The formalized syntax paradigm integrates reasoning about the syntax of expressions with reasoning about what the expressions mean. It is usually used in the metalogic of a logic rather than in the logic itself. For example, sets of axioms and rules of inference are often expressed using syntactic templates called formula schemas. They are formula-like expressions containing syntactic variables that represent collections of formulas with similar syntactic structure, but they usually are not official formulas of the logic. An example is the well-known axiom schema for the induction principle for the natural numbers:

   $$(A[x \mapsto 0] \wedge (\forall\, x\ .\ A \supset A[x \mapsto S(x)])) \supset \forall\, x\ .\ A$$

   where the syntactic variable A ranges over formulas. The formalized syntax paradigm can be used directly in a logic with natural number arithmetic using Gödel's *arithmetization of syntax* via Gödel numbering [15].

Mathematical practice is very rich and varied. There are usually many ways to attack a mathematical problem and many ways to express a solution. We maintain that a logic for mechanized mathematics can benefit from being able to support several reasoning paradigms. Chiron supports in an integrated manner the following five reasoning paradigms:

1. Classical.
2. Permitted undefinedness.
3. Set theory.
4. Type theory.
5. Formalized syntax.

The design of Chiron is novel, but the lion's share of the ideas behind the design come from traditional predicate logic, set theory, and type theory.

This paper gives a quick, informal presentation of the syntax and semantics of Chiron and then discusses some of the benefits Chiron provides as a multi-paradigm logic. The syntax and semantics of Chiron are presented together in section 2. A compact notation for Chiron is given in section 3. Section 4 discusses Chiron's support for its five reasoning paradigms. The paper ends with some concluding remarks in section 5.

2 Syntax and Semantics

We will present the syntax and semantics of Chiron together. The presentation of the semantics will be informal. The reader can find separate formal presentations of the syntax and semantics in [12].

2.1 Values

The official semantics of Chiron is based on *standard models*. (Two alternate semantics based on other kinds of models are given in [12].) A standard model is an elaboration of a model of NBG set theory. The basic values or elements in a model of NBG are classes (which include sets and proper classes). The values of a standard model include other values besides classes, but classes are the most important. We will not give a precise definition of a standard model; the reader can find a proper definition of this notion in [12]. We will instead present the semantics of Chiron with respect to an arbitrary standard model M. As we move along, we will reveal the ingredients of M as needed.

M includes *values* of several kinds: sets, classes, superclasses, truth values, the undefined value, and operations. M is derived from a structure, consisting of a nonempty domain D_{c} of classes and a membership relation $\in$ on D_{c}, that satisfies the axioms of NBG set theory as given, for example, in [16] or [21].

A *class* of M is a member of D_{c}. A *set* of M is a member x of D_{c} such that $x \in y$ for some member y of D_{c}. That is, a set is a class that is itself a member of a class. A class is thus a collection of sets. A class is *proper* if it is not a set. A *superclass* of M is a collection of classes in D_{c}. We consider a class, as a collection

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

Table 1. The Key Words of Chiron.

of sets, to be a superclass itself. Let D_{v} be the domain of sets of M and D_{s} be the domain of superclasses of M. The following inclusions hold: $D_{\mathrm{v}} \subset D_{\mathrm{c}} \subset D_{\mathrm{s}}$. D_{v} is the universal class (the class of all sets), and D_{c} is the universal superclass (the superclass of all classes).

T, F, and $\perp$ are distinct values of M not in D_{s}. T and F represent the truth values *true* and *false*, respectively. $\perp$ is the *undefined value* which serves as the value of undefined terms. For $n \geq 0$, an *n-ary operation* of M is a total mapping from $D_1 \times \cdots \times D_n$ to D_{n+1} where D_i is D_{s}, $D_{\mathrm{c}} \cup \{\perp\}$, or $\{\mathrm{T}, \mathrm{F}\}$ for each i with $1 \leq i \leq n+1$. Let D_{o} be the domain of operations of M. $D_{\mathrm{s}} \cup \{\mathrm{T}, \mathrm{F}, \perp\}$ and D_{o} are assumed to be disjoint.

2.2 Expressions

Let $\mathcal{S}$ be a fixed infinite set of symbols that includes the 30 *key words* in Table 1. The key words are used to classify expressions, identify different categories of expressions, and name the built-in operators (see below).

An *expression* of Chiron is defined inductively by:

1. Each symbol $s \in \mathcal{S}$ is an expression.
2. If $e_1, \ldots, e_n$ are expressions where $n \geq 0$, then $(e_1, \ldots, e_n)$ is an expression.

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 $\mathcal{E}$ be the set of expressions of Chiron.

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 of M, while improper expressions are nondenoting (i.e., they do not denote anything). Operators are used to construct expressions. They denote operations. 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 $\perp$. Formulas are used to make assertions. They denote truth values.

A term is *defined* if it denotes a class and is *undefined* if it denotes $\perp$. Every term is assigned a type. Suppose a term a is assigned a type α. Then a is said to be a *term of type* α. Suppose further α denotes a superclass Σ_α. If a is defined, i.e., a denotes a class x, then x is in Σ_α. The value of a nondenoting term is the

1.	(op, true, formula)
2.	(op, false, formula)
3.	(op, set, type)
4.	(op, class, type)
5.	(op, expr, type)
6.	(op, expr-op, type)
7.	(op, expr-type, type)
8.	(op, expr-term, 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.

undefined value $\bot$, but the value of an nondenoting type or formula is D_{c} (the universal superclass) or F, respectively. That is, the values for nondenoting types, terms, and formulas are D_{c}, $\bot$, and F, respectively.

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.

2.3 Operators

A *kind* is the key word type, a type, or the key word formula. Kinds are expressions used to define the signatures of operators. If $s \in \mathcal{S}$ and $k_1, \ldots, k_{n+1}$ are kinds where $n \geq 0$, then

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

is an n-ary *operator*. s is the *name* and $k_1, \ldots, k_{n+1}$ is the *signature* of the operator. The operator is a *type operator*, *term operator*, or *formula operator* if $k_{n+1} = \mathsf{type}$, k_{n+1} is a type, or $k_{n+1} = \mathsf{formula}$, respectively.

A *language* of Chiron is a set of operators that contains the 15 "built-in" operators in Table 2. Each standard model is *for* a particular language of Chiron. Let L be the language for which M is a model.

An n-ary operator $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$ denotes 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 k_i is a type, and $D_i = \{\mathrm{T}, \mathrm{F}\}$ if $k_i = \mathsf{formula}$ for each i with $1 \leq i \leq n+1$ such that:

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

2. If O is a built-in operator, then o is a particular operation of M specified by the definition of a standard model.

Like a function or predicate symbol in first-order logic, an operator in Chiron is not meaningful unless it is applied. Suppose $O = (\mathsf{op}, s, k_1, \ldots, k_n, k_{n+1})$ is an operator and $e_1, \ldots, e_n$ are expressions such that $k_i = \mathsf{type}$ and e_i is a type, k_i is a type and e_i is a term, or $k_i = \mathsf{formula}$ and e_i is a formula for all i with $1 \leq i \leq n$. Then

$$(\mathsf{op\text{-}app}, O, e_1, \ldots, e_n)$$

is an expression called an *operator application* that is a type if $k_{n+1} = \mathsf{type}$, a term of type k_{n+1} if k_{n+1} is a type, and a formula if $k_{n+1} = \mathsf{formula}$. If the value of e_i is $\perp$ or is in the value of k_i for all i for which k_i is a type, then the operator application denotes the result of applying the n-ary operation denoted by O to the n values denoted by $e_1, \ldots, e_n$. Otherwise, the operator application denotes D_{c}, $\perp$, or F if O is a type, term, or formula operator, respectively.

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.

Example 1. $O = (\mathsf{op}, \mathsf{class}, \mathsf{type})$ is a built-in 0-ary type operator. The operator application $(\mathsf{op\text{-}app}, O)$ is the type of classes. It denotes the universal superclass D_{c}. Similarly, the operator application $(\mathsf{op\text{-}app}, O')$, where $O' = (\mathsf{op}, \mathsf{set}, \mathsf{type})$, is the type of sets. It denotes the universal class D_{v}. □

Example 2. The operator

$$O = (\mathsf{op}, \mathsf{in}, (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{set}, \mathsf{type})), (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{class}, \mathsf{type})), \mathsf{formula})$$

is a built-in binary formula operator. If a and b are terms denoting the classes x and y, then the operator application

$$(\mathsf{op\text{-}app}, O, a, b)$$

is a formula that asserts $x \in y$. By our remark above, the value of this formula is F if x is not a set. □

Example 3. Suppose N is the type of natural numbers. Let

$$O = (\mathsf{op}, \mathsf{matrix}, \mathsf{N}, \mathsf{N}, \mathsf{type}, \mathsf{type})$$

be a ternary type operator such that, if a, b are terms of type N that denote the natural numbers m, n and α is a type, then the operator application

$$\gamma = (\mathsf{op\text{-}app}, O, a, b, \alpha)$$

denotes the type of $m \times n$ matrices of elements of type α. Since the type γ depends on the values of a and b, γ is a *dependent type* and O is a *dependent type constructor*. □

2.4 Variables

If $x \in \mathcal{S}$ and α is a type, then

$$(\mathsf{var}, x, \alpha)$$

is a term of type α called a *variable*. Variables are used mainly in conjunction with the five variable binders in Chiron: dependent function type, function abstraction, existential quantification, definite description, and indefinite description. 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 [7] or nominal datatypes [24, 26].

As in traditional predicate logic, a variable may occur either *bound* or *free* in a proper expression. The value of a bound occurrence of a variable $(\mathsf{var}, x, \alpha)$ is restricted to classes in the superclass denoted by α. The value of a free occurrence is either a class in this same superclass or $\bot$. The latter case happens, for example, when α denotes the empty set, i.e., the empty collection of classes.

Two variables $(\mathsf{var}, x, \alpha)$ and $(\mathsf{var}, x', \alpha')$ are considered to be identical only if x and x' are identical symbols and α and α' are identical types. In particular, if α and α' are not identical types, then $(\mathsf{var}, x, \alpha)$ and $(\mathsf{var}, x, \alpha')$ are not identical variables even if α and α' denote the same superclass.

2.5 Logical Connectives

Chiron includes the usual logical connectives. The propositional connectives for negation and disjunction are represented by the built-in formula operators named not and or. Truth value constants for true and false are represented by applications of the built-in 0-ary operators named true and false. If $(\mathsf{var}, x, \alpha)$ is a variable and B is a formula, then

$$(\mathsf{exist}, (\mathsf{var}, x, \alpha), B)$$

is a formula called an *existential quantification*. It is for expressing existential assertions. There are three sorts of equality—one for each of types, terms, and formulas—represented by the built-in operators named type-equal, term-equal, and formula-equal. (The universal quantifier will be introduced in the next section by a notational definition, and other propositional connectives can be introduced by operator definitions (see [12]).)

Example 4. Let O be the built-in ternary operator named term-equal:

$$\begin{array}{l}(\mathsf{op}, \mathsf{term\text{-}equal}, (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{class}, \mathsf{type})),\\ \qquad\qquad\quad (\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{class}, \mathsf{type})),\\ \qquad\qquad\quad \mathsf{type},\\ \qquad\qquad\quad \mathsf{formula}).\end{array}$$

(We will soon see why the operator for term equality is ternary instead of binary.) Also let a, b be terms, α be a type that denotes the superclass Σ_α, and C be the type of classes. $(\mathsf{op\text{-}app}, O, a, b, \alpha)$ asserts that a and b denote the same class in Σ_α.

$(\mathsf{op\text{-}app}, O, a, b, \mathsf{C})$ asserts that a and b denote the same class. $(\mathsf{op\text{-}app}, O, a, a, \alpha)$ asserts that a is *defined in* α, that is, that a denotes a class in Σ_α. And $(\mathsf{op\text{-}app}, O, a, a, \mathsf{C})$ asserts that a is defined, that is, that a denotes some class. □

If A is a formula and b, c are terms, then

$$(\mathsf{if}, A, b, c)$$

is an if-then-else term called a *conditional term*. Its value is the value of b if A is true and is the value of c if A is false. If b and c are of the same type α, then the conditional term is also of type α; otherwise its type is the type of classes. An if-then-else term could also be constructed by an operator application, but the term's type would have to be the last member of the signature of the operator. Thus, the typing of conditional terms is more convenient than the typing of operation applications representing if-then-else terms.

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. As in standard NBG set theory, only classes are first-class values in Chiron.

2.6 Functions

A *function* of M is a class f of M such that:

1. Each $p \in f$ is an ordered pair $\langle x, y \rangle$ where x, y are in D_v.
2. For all $\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle \in f$, if $x_1 = x_2$, then $y_1 = y_2$.

A function represents a mapping from sets to sets. A function 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. Every function that is a set is partial, and every function that is total is a proper class. Since functions are classes, they are first-class values, unlike operations.

Let D_f be the domain of functions of M. For f, x in D_c, $f(x)$ denotes the unique y in D_v such that f is in D_f and $\langle x, y \rangle \in f$. ($f(x)$ is undefined if there is no such unique y in D_v.) For Σ in D_s and x in D_c, $\Sigma[x]$ denotes the class of all y in D_v such that, for some f in both Σ and D_f, $f(x) = y$.

If α is a type and a is a term, then

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

is a type called a *type application*. If α denotes the superclass Σ and a denotes the class x, then the type application denotes $\Sigma[x]$. If a is undefined, the type application denotes D_c.

If $(\mathsf{var}, x, \alpha)$ is a variable and β is a type, then

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

is a type called a *dependent function type*. It denotes the superclass of functions $f \in D_\mathrm{f}$ such that:

1. For all sets z in the superclass denoted by α, if $f(z)$ is defined, then $f(z)$ is in the superclass denoted by β when the value of $(\mathsf{var}, x, \alpha)$ is z.
2. For all sets z not in the superclass denoted by α, $f(z)$ is undefined.

Dependent function types are commonly known as *dependent product types*.

If f is a term of type α and a is term, then

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

is a term of type $(\mathsf{type\text{-}app}, \alpha, a)$ called a *function application*. If the value of f is a function g, the value of a is a set z, and $g(z)$ is defined, then the value of the function application is $g(z)$. Otherwise the function application is undefined.

If $(\mathsf{var}, x, \alpha)$ is a variable and b is a term of type β, then

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

is a term of type $(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$ called a *function abstraction*. If the collection g of pairs $\langle z, z' \rangle$, where z is a set in the value of α and z' is a set equal to the value of b when the value of $(\mathsf{var}, x, \alpha)$ is z, is a function in D_{f}, then g is the value of the function abstraction. Otherwise the function abstraction is undefined.

The dependent function type

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

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.

Example 5. This example continues Example 3. Suppose R is the type of real numbers and v is the variable $(\mathsf{var}, x, \mathsf{N})$. Then

$$\gamma = (\mathsf{dep\text{-}fun\text{-}type}, v, (\mathsf{op\text{-}app}, O, v, v, \mathsf{R}))$$

is a dependent function type that denotes the superclass of functions that map a natural number n to an $n \times n$ matrix of real numbers. Suppose $\mathsf{zero\text{-}matrix}$ is a term of type γ that denotes the function that maps a natural number n to the $n \times n$ zero matrix. If a term a of type N denotes the natural number 17, then

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

is a term of type $(\mathsf{type\text{-}app}, \gamma, a)$ that denotes the 17×17 zero matrix. Notice that $(\mathsf{type\text{-}app}, \gamma, a)$ and $(\mathsf{op\text{-}app}, O, a, a, \mathsf{R})$ are different types that denote the same superclass, namely, the type of 17×17 matrices of real numbers. □

2.7 Definite and Indefinite Description

If $(\mathsf{var}, x, \alpha)$ is a variable and B is a formula, then

$$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$$

is a term called a *definite description* and

$$(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$$

is a term called an *indefinite description*. The definite description denotes *the* value of $(\mathsf{var}, x, \alpha)$ that satisfies B; it denotes $\perp$ if there is no value or more than one value of $(\mathsf{var}, x, \alpha)$ that satisfies B. The indefinite description denotes *some* value of $(\mathsf{var}, x, \alpha)$ that satisfies B; it denotes $\perp$ if there is no value of $(\mathsf{var}, x, \alpha)$ that satisfies B.

Definite descriptions are quite common in mathematics but often occur in a disguised form. For example,"the limit of $\mathsf{sin}\frac{1}{x}$ as x approaches 0" is a definite description. Indefinite description is also employed in mathematics but less so than definite description. Definite and indefinite description works very smoothly in a logic, like Chiron, in which terms are allowed to be undefined [11].

In traditional logic, definite descriptions are usually formed using an *iota operator* (ι), while indefinite descriptions are usually formed using an *epsilon operator* (ϵ).

2.8 Quotation and Evaluation

M contains a total, injective mapping $H : \mathcal{S} \to D_{\mathrm{v}}$ such that, for each $s \in \mathcal{S}$, $H(s)$ is neither an ordered pair nor the empty set $\emptyset$. $\widehat{H}$ is the total, injective mapping of $\mathcal{E}$ into D_{v} defined inductively by:

1. If $s \in \mathcal{S}$, then $\widehat{H}(s) = H(s)$.
2. If $e = (\,) \in \mathcal{E}$, then $\widehat{H}(e) = \emptyset$.
3. If $e = (e_1, \ldots, e_n) \in \mathcal{E}$ where $n \geq 1$, then
$$\widehat{H}(e) = \langle \widehat{H}(e_1), \widehat{H}((e_2, \ldots, e_n)) \rangle.$$

$\widehat{H}(e)$ is a set, called the *construction* of e, that represents the syntactic structure of the expression e. Notice that a construction has the same structure as a list data structure in the Lisp programming language with the empty set playing the role of nil.

A construction is an *operator construction*, *type construction*, *term construction*, or *formula construction* if it represents an operator, type, term, or formula, respectively. The class of constructions is represented by the application of the built-in 0-ary operator named expr. That is,

$$(\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{expr}, \mathsf{type}))$$

is the type of expression constructions. Similarly, the class of operator constructions, type constructions, term constructions, or formulas constructions is represented by the application of the built-in 0-ary operators named $\mathsf{expr\text{-}op}$, $\mathsf{expr\text{-}type}$, $\mathsf{expr\text{-}term}$, or $\mathsf{expr\text{-}formula}$, respectively.

If e is any expression, proper or improper, then

(quote, e)

is a term of type

$(\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{expr}, \mathsf{type}))$

called a *quotation*. The value of the quotation is $\widehat{H}(e)$, the construction of 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).

Example 6. Suppose zero and one are terms that denote the natural numbers 0 and 1, respectively. Define a *(binary) numeral* to be an expression $(a_1, \ldots, a_n)$ where $n \geq 1$ and a_i is zero or one for each i with $1 \leq i \leq n$. As defined, a numeral is an improper expression, and thus it denotes nothing. However, if n is a numeral, then (quote, n) is a proper expression that denotes the construction of n. Since constructions are sets that can be manipulated just like any other sets, numerals can be manipulated (e.g., added or multiplied) in Chiron by manipulating their quotations. (Of course, in order to know what these manipulations mean arithmetically, we would need to define a term that denotes an appropriate function from numerals to natural numbers.) This example shows that quotation enables the manipulation of syntactic objects to be performed in Chiron using all of the machinery that Chiron provides. □

If a is a term and k is a kind, then

(eval, a, k)

is an expression called an *evaluation* that is a type if $k = \mathsf{type}$, a term of type k if k is a type, and a formula if $k = \mathsf{formula}$. Roughly speaking, if a denotes a construction that represents an expression e, then the evaluation denotes the value of e.

Suppose $k = \mathsf{type}$. If a denotes a construction that represents a type α in which the symbol eval does not occur, then the evaluation denotes the value of α. Otherwise the evaluation denotes D_{c}. Suppose k is a type. If a denotes a construction that represents a term b in which the symbol eval does not occur and the value of b is in the value of k, then the evaluation denotes the value of b. Otherwise the evaluation denotes $\bot$. And suppose $k = \mathsf{formula}$. If a denotes a construction that represents a formula A in which the symbol eval does not occur, then the evaluation denotes the value of A. Otherwise the evaluation denotes F. In each of the three cases the condition concerning the presence of the symbol eval is needed to block the liar paradox and similar semantically ungrounded expressions (see [12]).

See Example 7 in section 4.5 for an illustration of how the law of beta reduction can be formalized in Chiron using evaluation. Quotation and evaluation in Chiron are inspired by the quote and eval operators in the Lisp programming language.

3 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 we defined above.

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)
$(\exists\, x : \alpha\ .\ B)$	$(\mathsf{exist}, (\mathsf{var}, x, \alpha), B)$
$(\iota\, x : \alpha\ .\ B)$	$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$
$(\epsilon\, x : \alpha\ .\ B)$	$(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$
$\ulcorner e \urcorner$	(quote, e)
$[\![a]\!]_{\mathrm{ty}}$	$(\mathsf{eval}, a, \mathsf{type})$
$[\![a]\!]_{\alpha}$	$(\mathsf{eval}, a, \alpha)$
$[\![a]\!]_{\mathrm{te}}$	$(\mathsf{eval}, a, (\mathsf{op\text{-}app}(\mathsf{op}, \mathsf{class}, \mathsf{type})))$
$[\![a]\!]_{\mathrm{fo}}$	$(\mathsf{eval}, a, \mathsf{formula})$

Table 3. Compact Notation

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.
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

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

   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 a term of the form $(\star\, x : \alpha\ .\ e)$, where $\star$ is Λ, λ, $\exists$, $\forall$, ι, or ϵ, may be written as $(\star\, x\ .\ e)$.

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

Compact Notation	Defining Expression
T	$(\mathsf{true} :: \mathsf{formula})()$
F	$(\mathsf{false} :: \mathsf{formula})()$
V	$(\mathsf{set} :: \mathsf{type})()$
C	$(\mathsf{class} :: \mathsf{type})()$
E	$(\mathsf{expr} :: \mathsf{type})()$
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_{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

4 Reasoning Paradigms

We will now briefly describe how Chiron supports the five reasoning paradigms listed at the end of the Introduction.

4.1 Classical

Chiron fully supports the classical paradigm. As we mentioned in section 2.5, Chiron has the usual logical connectives: propositional connectives for negation and disjunction, an existential quantifier, constants for true and false, and an equality for each of types, term, and formulas. Chiron also includes an if-then-else term constructor. The universal quantifier is introduced by a notational definition, and other propositional connectives can be introduced by operator definitions (see [12]).

4.2 Set Theory

Chiron also fully supports the set theory paradigm. As a derivative of NBG set theory, Chiron can be used to reason about both sets and proper classes. The usual set-theoretic operators such as union, intersection, complement, etc. and constants such as the empty set and the universal class can be introduced by operator definitions (see [12]). Unlike traditional set theories formalized in first-order logic, Chiron is equipped with a rich language for forming typed terms that denote sets and classes. This language includes mechanisms for expressing function application and abstraction and definite and indefinite description.

4.3 Permitted Undefinedness

Undefined expressions are handled in Chiron according to the *traditional approach to undefinedness* [11]. Terms are allowed to be undefined. This means that a term that has no natural denotation—such as an application of a function to an argument outside of its domain—denotes the undefined value $\bot$ instead of a class. An undefined type—such as the application of a type to an undefined term—denotes the universal superclass D_{c}. And an undefined formula—such as an out-of-range formula evaluation—denotes F.

As mentioned in Example 4, the assertion that a term a is defined in a type α is explicitly expressed by the formula $(a =_\alpha a)$. This same assertion can be implicitly expressed in various ways. For example, if a is a term of type α, the formula $(a = b)$ implies that a is defined in α. The use of implicit definedness assertions enables statements involving partial functions and definite and indefinite descriptions to be expressed very concisely [11].

4.4 Type Theory

Chiron can be used as a type theory. Every term has a unique type determined by its syntax. The Chiron type system contains three sorts of types:

1. A *type operator application* constructs a type from a list of proper expressions which may be types, terms, or formulas. Many kinds of types can be formed in this way as shown by the following examples. An application of a type operator with the signature type introduces a base type such as V, C, or E. An application of a type operator with a signature $\mathsf{type}, \ldots, \mathsf{type}$ of length $n+1$ where $n \geq 1$ introduces a type constructed from a list of n other types. An application of a type operator with a signature that contains a type introduces a dependent type.
2. A *dependent function type* constructs a dependent function type from a variable and a type.
3. A *type application* constructs a type from a type (which is usually a dependent function type) and a term (to which the type is applied).

Each type in Chiron denotes a superclass that is a subcollection of D_{c}, the universal superclass that contains all classes. Thus, viewed semantically, types are allowed to freely "overlap". We say that a type α is a *subtype* of another type β if the value of α is a subcollection of the value of β. It is possible for a type to be empty, that is, to denote the empty set. For example, if α is a type that denotes a superclass Σ that does not contain any functions and a is a defined term, then the type $\alpha(a)$ is empty.

Although each term has a unique type assigned to it, a term can be *defined in* many types. Suppose a is a term of type α. If a is defined, it is defined in α, the type C of classes, and every other type that denotes a superclass containing the value of a. If a is undefined, it is not defined in any type.

4.5 Formalized Syntax

Quotation is used in Chiron to refer to a set called a construction that represents the syntactic structure of an expression. Analogously to a *Gödel number* that encodes an expression as a number, a quotation in Chiron is a *Gödel set* that encodes an expression as a set. Constructions can be probed and formed using the set-theoretic machinery of Chiron. Evaluation is used in Chiron to refer to the value of the expression that a construction represents. To avoid the liar paradox and similar semantically ungrounded expressions, evaluation can only be applied to terms that denote constructions representing expressions that do not contain the symbol eval.

Quotation and evaluation together enable many common syntactic devices that are usually expressed metalogically—such as syntactic side conditions, schemas, and syntactic transformations used in deduction and computation rules—to be expressed directly in Chiron. We will give two illustrations: (1) how schemas can be formalized in Chiron and (2) how metalogical reasoning can be "reflected" in Chiron.

Schemas A *schema* is an expression that contains special variables called *schema variables* that range over expressions. An instance of a schema is obtained by simultaneously replacing each schema variable with an appropriate expression. A schema is thus a representation of the set of its instances. Schemas are used in traditional logic to describe deduction rules and sets of axioms. However, schemas usually cannot be directly formalized in a logic but are instead expressed in the logic's metalogic.

Schemas can be directly formalized in Chiron. Schema variables are formalized by variables of type E, the type of expression constructions, or by variables of the subtypes of E: E_{op}, E_{ty}, E_{te}, and E_{fo}. Syntactic side conditions are formalized as conditions about constructions. Syntactic transformations are formalized as applications of functions that map constructions to constructions. Evaluation is used to state the value of the result of the transformation. And instances of a formalized schema are obtained by instantiating the schema variables with quoted expressions.

Example 7. 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. The expressions x, α, b, a are schema variables, "a is free for $(x : \alpha)$ in b" is a syntactic side condition, and $b[(x : \alpha) \mapsto a]$ is the syntactic transformation that substitutes a for each free occurrence of x in b.

Using constructions, quotation, and evaluation, a general law of beta reduction can be formalized in Chiron as a *single formula* that merges the law of beta reduction for dependent function types and the law of beta reduction for functions:

$$\begin{array}{l}\forall\, e : \mathsf{E}_{\mathrm{te}}\ . \\ \quad (\mathsf{is\text{-}redex}(e) \wedge \mathsf{free\text{-}for}(\mathsf{redex\text{-}arg}(e), \mathsf{redex\text{-}var}(e), \mathsf{redex\text{-}body}(e))) \\ \qquad \supset \\ \quad [\![e]\!]_{\mathrm{te}} \simeq [\![\mathsf{sub}(\mathsf{redex\text{-}arg}(e), \mathsf{redex\text{-}var}(e), \mathsf{redex\text{-}body}(e))]\!]_{\mathrm{te}}\end{array}$$

Here is-redex, free-for, etc. are defined operators without their signatures. The syntactic side condition (corresponding to "a is free for $(x : \alpha)$ in b") is incorporated into the hypothesis of the formula's implication, while the syntactic transformation (corresponding to $b[(x : \alpha) \mapsto a]$) is incorporated into the conclusion. The beta reduction law is applied to an application α of a dependent function type or an application a of a function abstraction by instantiating the variable $(e : \mathsf{E}_{\mathrm{te}})$ with the quotation $\ulcorner\alpha\urcorner$ or $\ulcorner a\urcorner$. See [12] for details. Many other axiom schemas and rules of inference rules can be formalized in Chiron using schemas of this style. □

Metalogical Reflection Reasoning that is normally done in the metalogic of a logic—such as defining and applying a proof system—can be "reflected" in Chiron using its facility to reason about syntax. For example, a *theory* T can be defined as a set of formula constructions. A schema for a set of logical axioms for Chiron can be defined as a Chiron-style schema. A rule of inference for Chiron can be defined as Chiron-style schema that involves a *provability relation* $\vdash$ between theories and formula constructions. Then a formula $T \vdash a$ would assert that the formula represented by the construction a is provable from the formulas represented by the constructions in T using the Chiron-style schemas that define the logical axioms and rules of inference for Chiron.

5 Conclusion

Chiron is a logic designed to be a logical foundation for mechanized mathematics. Derived from NBG set theory, it is based on familiar principles from predicate logic, set theory, and type theory. It supports in an integrated manner five reasoning paradigms that are commonly employed either in mathematical practice or in contemporary MMSs. As a result, Chiron has a high level of practical expressivity—it offers the user easy access to a rich mixture of popular reasoning styles.

The design of Chiron is the first step of a long-range research program. The second step is to design a proof system for Chiron. Part of this task will be to determine Chiron's exact relationship to NBG. It is our conjecture that there is a faithful interpretation of NBG in Chiron. The third step is to develop an implementation of Chiron and its proof system. The final, and most important step, will be to develop a series of applications to demonstrate Chiron's reach and level of effectiveness.

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. The author is also grateful to Volker Sorge for reading a preliminary draft of the paper.

References

1. P. B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfennig, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.
2. G. Bancerek and P. Rudnicki. Inductively defined types. In P. Martin Löf and G. Mints, editors, *COLOG-88: Proceedings of the International Conference on computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.
3. B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, 1991.
4. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
5. Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.0*, 2006. Available at `http://coq.inria.fr/doc/`.
6. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
7. 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.
8. W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.
9. W. M. Farmer. STMM: A Set Theory for Mechanized Mathematics. *Journal of Automated Reasoning*, 26:269–289, 2001.
10. W. M. Farmer. The seven virtues of simple type theory. SQRL Report No. 18, McMaster University, 2003. Revised 2006.
11. 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.
12. W. M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University, 2007.
13. W. M. Farmer and J. D. Guttman. A set theory with support for partial functions. *Studia Logica*, 66:59–78, 2000.
14. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
15. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
16. 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.

17. M. J. C. Gordon and T. F. Melham. *Introduction to* HOL*: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
18. R. D. Jenks and R. S. Sutor. *Axiom: The Scientific Computation System*. Springer-Verlag, 1992.
19. K. Kunen. *Set Theory: An Introduction to Independence Proofs*. North-Holland, 1980.
20. Lemma 1 Ltd. *ProofPower: Description*, 2004.
Available at `http://www.lemma-one.com/ProofPower/doc/doc.html`.
21. E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, fourth edition, 1997.
22. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.
23. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
24. A. M. Pitts. Nominal Logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
25. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
26. 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.
27. S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1991.