Programming in Logic without Prolog

M.H. van Emden

Department of Computer Science
University of Victoria, Canada
vanemden@cs.uvic.ca

Abstract. Logic can be made useful for programming and for databases independently of logic programming. To be useful in this way, logic has to provide a mechanism for the definition of new functions and new relations on the basis of those given in the interpretation of a logical theory. We provide this mechanism by creating a compositional semantics on top of the classical semantics. In this approach, verification of computational results relies on a correspondence between logic interpretations and a class definition in languages like Java or C++. The advantage of this approach is the combination of an expressive medium for the programmer with, in the case of C++, optimal use of computer resources.

Keywords First-order predicate logic, compositional semantics, relations, recursive definitions, object-oriented programming languages.

1 Introduction

“What can logic do for programming?” This question could have been asked as early as 1950 when it was first noticed that valuable time on one of the few computers had been wasted due to a programming error. The question has not been used as a starting point in the development of programming languages. When logic was connected for the first time with programming it was in the form of logic programming, arising as a special form of automatic theorem-proving with the resolution inference rule. Given that many alternatives need to be excluded before arriving at logic programming, it is likely that it is not the whole answer to the question of what logic can do for programming. To make sure that we don’t prematurely exclude interesting possibilities, let us see how far we can get without any inference rule. On the other hand, in this paper we do not try to do everything, so we restrict the scope of “logic” to mean first-order predicate logic.

A good starting point for the use of logic for programming is that logic formulas have symbols that are interpreted as functions or as relations. On the programming side we note that code is organized into subroutines, which can take the form of procedures or function subroutines. A desirable property of subroutines is that they are free from side-effects: that function subroutines only interact with their environment by delivering a result and procedures only do so by modifying one or more parameters. Our approach to programming in logic relies on a correspondence between such side effect-free subroutines on the one hand and functions or relations in interpretations of logic theories on the other.
Most of a programmer’s work consists of writing new subroutines in terms of existing ones. To be useful for programming, logic has to provide a mechanism for defining new functions and relations in terms of existing ones. It may seem that the semantics of logic does not provide such a mechanism. However, this problem is solved by reformulating the classical semantics as *compositional semantics*. In compositional semantics the meaning of a composite syntactic construct is defined as a combination of the meanings of the constituent parts of the construct. In this way we obtain a meaning assigned to a term with variables. This meaning is a function. We also obtain a meaning assigned to formula with free variables. This meaning is a relation. Compositional semantics makes any term and formula with (free) variables available for the definition of a new function or relation.

What do we do with this definition mechanism? To go beyond a theoretical exercise there needs to be a way to get a computer program verified in some way by a specification in logic.

Let us consider interpretations for a theory of logic. Classical examples are theories for commonly used structures of algebra, such as monoids, groups, rings, and fields. For each of these, the axioms are formalized as a theory of logic. A given algebraic structure is then by definition a monoid, group, ring, or field according to whether it satisfies, in the sense of model theory, the defining logical theory. Although these theories and this methodology were established before the birth of computers, it is a topic of current research to establish a link between them and efficient implementations of algorithms [3].

An interpretation for a theory \( T \) consists of a universe of discourse \( D \), a function over \( D \) as interpretation for every function symbol in \( T \), and a relation over \( D \) for every relation symbol in \( T \). A properly structured program consists almost entirely of definitions of function subroutines and of procedures. In a language like Java or C++ classes are used to organize function subroutines and procedures into meaningful groups. This suggests to us that the desired connection between logic and the operation of a computer can be made by writing a class in such a way that it is sufficiently similar to an interpretation of a logic theory that the behaviour of the (side-effect free) subroutines of the class is described by the corresponding function or relation symbol of the theory. Although classes are used, this is not object-oriented programming, where side-effects of subroutines are not merely regrettable lapses, but constitute the essence.

To illustrate how this correspondence between a logical theory and a class can be used consider the following example. Let the theory consist of the axioms for Euclidean domains in the sense of abstract algebra. The constants are 0 and 1 for the additive and multiplicative identities. The function symbols are those for addition, multiplication, and subtraction. The integers are the most familiar example. Especially interesting are the ones leading to a finite universe of discourse, such as the modular numbers. To be specific, consider a C++ class that emulates an interpretation for the axioms for Euclidean domains by having instances named `zero` and `one` and subroutines for addition, multiplication, and subtraction. Let us add to the theory for Euclidean domains a relation symbol
gcd such that gcd(a, b, c) holds iff c is the greatest common denominator of a
and b. The definition can be made to correspond to a C++ procedure using an
implementation of the C++ class for Euclidean domains. Such a procedure is a
program for a computer on which C++ is implemented. Because the procedure
relies on the C++ class, the correctness of the implementations of addition, multi-
plification, and subtraction of the Euclidean domain class imply correctness of
the implementation of gcd.

Plan of the paper As preparation for compositional semantics of logic, and to
establish terminology and notation, we begin with two reviews: material on re-
lations in Section 2; on model-theoretic semantics in Section 3. Much of this
material is standard, but some concepts, such as the quotient of one function
by another, are rarely found in the literature. This particular one is crucial to
the paper. The basic observation underlying our definitions of relations is that
any formula defines a relation as the set of tuples that, when assigned to the
tuple of free variables, makes the formula true, given a fixed interpretation of
function and relation symbols. In this way formulas denote relations; we call this
the compositional semantics of logic. It is the topic of Section 3.2 In Section 4
we apply the compositional semantics to the introduction of new function and
relation symbols and define their interpretations. This section does not treat the
recursive case, which is the topic of Section 5. In Section 6 we discuss the design
of a C++ class so that it can be regarded as a specification of an interpretation
for a given theory.

2 Notation and terminology for functions and relations

Not all authoritative texts agree on the notations of set theory needed in this
paper. We also need some concepts that are usually not covered in introduc-
tions like the present one. Therefore we collect in this section the necessary material,
termology and notation.

Functions The set of functions that take arguments in a set S and have values
in a set T is denoted S → T. This set is said to be the type of a function
f ∈ (S → T). We write f(a) for the element of T that is the value of f for
argument a ∈ S; f(a) may also be written as fa. If an expression E with a single
free variable x is used to define the values of f ∈ (S → T), then the mapping of
f is (a ∈ S) ↦ E[a].

If S′ ⊆ S, then the projection (or restriction) f↓S′ of f ∈ S → T on S′ is
the function in S′ → T such that (f↓S′)(a) = f(a) for all a ∈ S′.

Suppose we have f ∈ S → T and g ∈ T → U. Then the composition g ◦ f
of f and g is the function h ∈ S → U defined by x ↦ g(f(x)). Suppose now
that we are given f ∈ S → T and h ∈ S → U, is there a g ∈ T → U such that
h = g ◦ f? The answer, depending on f and h, may be that there is no such g,
or one, or more than one. We therefore define h/f, the quotient of h by f, to be
{g ∈ T → U | f ◦ g = h}. 

Tuples Often an n-tuple over a set D is thought of as an object \((d_0, \ldots, d_{n-1})\) in which an element of D is associated with each of the indexes 0, \ldots, n-1. It is convenient to view such a d as a function of type \(\{0, \ldots, n-1\} \to D\). This formulation allows us to consider tuples of which the index set is a set other than \(\{0, \ldots, n-1\}\). Hence we define a tuple as an element of the function set \(I \to T\), where I is an arbitrary countable set to serve as index set. \(I \to T\) is the type of the tuple.

Example If \(t\) is tuple in \(\{x, y, z\} \to R\), then we may have \(t_x = 1.1, t_y = 1.21, \text{ and } t_z = 1.331\). A more compact notation would be welcome; we use \(t = \begin{pmatrix} x & y & z \\ 1.1 & 1.21 & 1.331 \end{pmatrix}\), where the order of columns is immaterial.

Example \(t \in \{0, 1, 2\} \to \{a, b, c\}\), where \(t = \begin{pmatrix} 2 & 1 & 0 \\ a & c & b \end{pmatrix}\). In cases like this, where the index set is of the form \(\{0, \ldots, n-1\}\), we use the compact notation \(t = (b, c, c)\), using the conventional order of the index set.

Relations A relation is a set of tuples with the same type. This type is the type of the relation.

Example \(sum = \{(x, y, z) \in (\{0, 1, 2\} \to R) \mid x + y = z\}\) is a relation of type \(\{0, 1, 2\} \to R\). Compare this relation to the relation \(\sigma = \{s \in (\{x, y, z\} \to R) \mid s_x + s_y = s_z\}\). As their types are different, they are different relations; \((2, 2, 4) \in sum\) is not the same tuple as \(s \in \sigma\) where \(s = \begin{pmatrix} x & y & z \\ 2 & 2 & 4 \end{pmatrix}\).

Definition 1. If \(r\) is a relation with type \(I \to T\), then the projection \(\pi_{I'}(r)\) of \(r\) on \(I' \subseteq I\) is

\[
\{f' \in I' \to T \mid \exists f \in r. (f|_{I'}) = f'\}.
\]

If \(r_0\) and \(r_1\) are relations with types \(I_0 \to T\) and \(I_1 \to T\), respectively, then the join \(r_0 \bowtie r_1\) of \(r_0\) and \(r_1\) is

\[
\{f \in (I_0 \cup I_1) \to T \mid (f|_{I_0}) \in r_0 \text{ and } (f|_{I_1}) \in r_1\}.
\]

Definition 2. Let \(H \subseteq (S \to U)\) be a relation and let \(f \in S \to T\) be a tuple. Then the quotient \(H/f\) of \(H\) by \(f\) is defined as the relation \(\cup\{h/f \mid h \in H\}\) of type \(T \to U\).

Example With \(S = \{0, 1, 2\}\), \(U = \mathcal{R}\), \(T = \{x, y\}\), \(sum = \{h \in \{0, 1, 2\} \to \mathcal{R} \mid h_0 + h_1 = h_2\}\), and \(f = (x, x, y)\) we have

\[
sum/(x, x, y) = \{(s \in \{x, y\}) \to \mathcal{R} \mid s_y = 2s_x\}.
\]

Here quotient on relations is used to define on the basis of the \(sum\) relation the relation indexed by \(\{x, y\}\) in which the argument indexed by \(y\) is double the one indexed by \(x\).
3 Semantics for first-order predicate logic

Conventional semantics is primarily concerned with the justification of inference systems. The use of predicate logic for the definition of functions or relations is secondary, if considered at all. As a result, conventional semantics centres around the concept of satisfaction: under what conditions is a formula satisfied by a given interpretation of the relation symbols and constants under a given assignment of individuals to the variables. Because of the emphasis on satisfaction, we refer to this kind of semantics as satisfaction semantics. Compositional semantics, in contrast with satisfaction semantics, defines the meaning of a complex term or formula as a composition of the meanings of the constituent terms or formulas.

3.1 Satisfaction semantics

Our language of logical formulas is determined by a set \( V \) of variables, a set \( F \) of function symbols, and a set \( R \) of relation symbols. The role constant symbols is played by 0-ary function symbols.

To avoid lexical details we give the syntax in an abstract form.

A term is a variable, a constant symbol, or an expression consisting of a \( k \)-ary function symbol and a tuple of \( k \) terms.

An atom (or atomic formula) is an expression consisting of a \( k \)-ary relation symbol and a tuple of \( k \) terms.

A conjunction is a formula consisting of a set of formulas.

An existential quantification is a formula consisting of a variable and a formula.

A negation is a pair consisting of a formula and an indication that the pair is a negation.

An interpretation \( M \) for the language consists of a set \( D \) called the universe of discourse (with elements called individuals) of the interpretation, a function that maps every \( n \)-ary function symbol in \( F \) to a function of type \( D^n \rightarrow D \), and a function that maps every \( n \)-ary relation symbol in \( R \) to a subset \( M(p) \) of \( D^n \).

The interpretation \( M \) is extended to assign the meaning \( M(t) \) to every variable-free term \( t \) and extended to determine whether a variable-free formula is true.

We first define the meaning of variable-free atoms.

- \( M(f(t_0, \ldots, t_{n-1})) = (M(f))(M(t_0), \ldots, M(t_{n-1})) \).
- A variable-free atom \( p(t_0, \ldots, t_{k-1}) \) is satisfied by an interpretation iff \( (M(t_0), \ldots, M(t_{k-1})) \in M(p) \).
- A conjunction \( \{ F_0, \ldots, F_{n-1} \} \) of variable-free formulas is satisfied by \( M \) iff \( F_i \) is satisfied by \( M \), for all \( i \in \{0, \ldots, n-1\} \).
- A variable-free formula that is the negation of \( F \) is satisfied by \( M \) iff \( F \) is not satisfied by \( M \).
We now consider meanings of formulas that contain variables. Let $A$ be an assignment, which is a function in $V \rightarrow D$, assigning an individual in $D$ to every variable. In other words, $A$ is a tuple of elements of $D$ indexed by $V$. As meanings of terms with variables depend on $A$, we write $M_A$ for the function mapping a term to a domain element. $M_A$ is defined as follows.

- $M_A(t) = A(t)$ if $t$ is a variable
- $M_A(c) = M(c)$ if $c$ is a constant
- $M_A(f(t_0, \ldots, t_{n-1})) = (M(f))(M_A(t_0), \ldots, M_A(t_{n-1}))$.
- $p(t_0, \ldots, t_{k-1})$ is satisfied by $M$ with $A$ iff 
  $(M_A(t_0), \ldots, M_A(t_{k-1})) \in M(p)$

Now that satisfaction of atoms is defined, we can continue with:

- $\{F_0, \ldots, F_{n-1}\}$ is satisfied by $M$ with $A$ iff the formulas $F_i$ are satisfied by $M$ with $A$, for all $i = 0, \ldots, n-1$.
- If $F$ is a formula, then $\exists x.F$ is satisfied by $M$ and $A$ iff there is a $d \in D$ such that $F$ is satisfied with $M$ with $A \mid x \mapsto d$ where $A \mid x \mapsto d$ is an assignment that maps $x$ to $d$ and maps the other variables according to $A$.
- $\neg F$ is satisfied by $M$ with $A$ iff formula $F$ is not thus satisfied.

The meaning of formulas contain disjunction, implication, or existential quantification is obtained by eliminating these according to the usual rules.

### 3.2 Compositional semantics for first-order predicate logic

The conventional semantics for first-order predicate logic focuses on the conditions under which a sentence, that is, a variable-free formula, is satisfied by an interpretation for constants, function symbols, and relation symbols. Yet terms with variables and formulas with free variables are potentially definitions of new functions and relations defined in terms of existing ones.

**Definition 3.** Let $t$ be a term with set $V$ of variables, and let $M$ be an interpretation for the theory in which the term occurs. $M(t)$ is a function of type $(V \rightarrow D) \rightarrow D$ that maps $A \in (V \rightarrow D)$ to $M_A(t)$.

**Example** $t = x^2 + 2y^2 + 3z^2$, $V = \{x, y, z\}$, $D = \mathbb{R}$, $M(t)$ is the function with map 

$$A \in (V \rightarrow \{x, y, z\}) \mapsto (M_A(t) \in \mathbb{R}).$$

e.g. $M(t)(\frac{x^2 + y^2 + z^2}{3 + 1}) = 3^2 + 2 \cdot 2^2 + 3 \cdot 1^2 = 20$.

Following Cartwright [2]:

**Definition 4.** Let $V$ be the set of the free variables in the formula $p(t_0, \ldots, t_{n-1})$. We extend $M$ to atomic formulas containing variables by defining $M(p(t_0, \ldots, t_{n-1}))$ to be 

$$\{A \in V \rightarrow D \mid (M_A(t_0), \ldots, M_A(t_{n-1})) \in M(p)\}.$$
According to this definition, a closed formula denotes a relation consisting of
tuples of length 0. As there is only one such tuple, there are only two such
relations, each of which is identified with one of the two truth values. With
this understanding, the following definition generalizes the conventional one for
logical implication.

**Definition 5.** Let formulas $\varphi_0$ and $\varphi_1$ have the same set of free variables and
admit of the same interpretations. We define $\varphi_0 \models \varphi_1$ to mean that $M(\varphi_0) \subseteq M(\varphi_1)$ for all interpretations $M$.

In the previous section we used the conventional semantics, which determines
under what conditions a sentence is satisfied by an interpretation for the relation
symbols and constants, to define a semantics that extends the meaning function
$M$ from from relation symbols to atomic formulas with free variables. According
to this extended semantics every atomic formula with set $V$ of free variables
denotes a relation of type $V \to D$.

For a semantics to be compositional it is necessary that the meaning of a
composite formula is a composition of the relations that are the meanings of its
constituent formulas. Accordingly we define in this section the compositional se-
mantics of conjunctions, negations and existentially quantified for mulas in terms
of the relations denoted by their constituent formulas. And althou gh we have
already given, in Definition 4 a relational semantics for an atomic for mula that
may have free variables, this semantics is not compositional. For this it is nec-
essary that we specify what operation on $M(p)$ gives the $M(p(t_0, \ldots, t_{n-1}))$ of
Definition 4.

**Theorem 1.** If $t_0, \ldots, t_{n-1}$ are variables, then we have $M(p(t_0, \ldots, t_{n-1})) = M(p)/\langle t_0, \ldots, t_{n-1} \rangle$.

*Proof.* Let $A$ be such that $a = A\downarrow V$.

$a$ in the left-hand side $\Leftrightarrow$ (Definition 4)

$p(t_0, \ldots, t_{k-1})$ is satisfied by $M$ with $A \Leftrightarrow$ (use satisfaction)

$(a(t_0), \ldots, a(t_{k-1})) \in M(p) \Leftrightarrow$ (use $f = a \circ t$)

$(f_0, \ldots, f_{k-1}) \in M(p) \Leftrightarrow$ (use definition of $/$ (quotient))

$a \in M(p)/\langle t_0, \ldots, t_{k-1} \rangle$.

**Example**

$M(sum(x, x, y)) = \{ s \in x, y \to \mathcal{R} \mid 2s_x = s_y \}$

$= \{ t \in \{0, 1, 2\} \to \mathcal{R} \mid t_0 + t_1 = t_2 \}/(x, x, y)$

$= M(sum)/(x, x, y)$

The first equality arises by Definition 4. The second equality arises by Definition 2. The third equality arises by the meaning of $\text{sum}$ in the assumed interpre-
tation of the relation symbol.

**Theorem 2.** For any formulas $\varphi_0, \ldots, \varphi_{k-1}$ we have

$M(\varphi_0 \land \cdots \land \varphi_{k-1}) = M(\varphi_0) \Join \cdots \Join M(\varphi_{k-1})$
Theorem 3. Let \( \varphi \) be a formula with \( V \) as its set of free variables, and \( W = \{ w_0, \ldots, w_{k-1} \} \) a be subset of \( V \). Then we have
\[
M(\exists w_0 \ldots w_{k-1}. \varphi) = \pi_{V \setminus W}(M(\varphi))
\]

Theorem 4. Let \( \varphi \) be a formula with \( V \) as set of free variables. Then \( M(\neg \varphi) \) is the complement in \( V \to D \) of \( M(\varphi) \).

4 Extensions of theories

Most subroutines call subroutines. This means that much of a programmer’s activity consists of defining a new subroutine in terms of existing ones. What a programmer should look for in logic is the possibility of defining new functions and relations in terms of existing ones. In this section we use the compositional semantics developed earlier as the basis of such a definition mechanism.

Definition of functions

Let \( t \) be a term with \( V \) as set of variables. It can be a complex term, deeply nested, with many occurrences of function symbols. Its subterms can share variables in intricate patterns. This richness of expression makes it attractive for a programmer to encapsulate such a complex term by making its denotation the interpretation of a new function symbol \( f \). A candidate for the interpretation of such an \( f \) is \( M(t) \).

But suppose that we interpret \( f \) by \( M(t) \), how do we then interpret the term \( f(t_0, \ldots, t_{n-1}) \) when the interpretation gives \( a_0, \ldots, a_{n-1} \in D \) as values for \( t_0, \ldots, t_{n-1} \)? How do the \( n \) individuals \( a_0, \ldots, a_{n-1} \) find their way to the corresponding \( n \) variables in \( t_0, \ldots, t_{n-1} \)? The difficulty here is that \( M(t) \) is a function of type \( (V \to D) \to D \), whereas \( f \) needs to be interpreted by a function of type \( (\{0, \ldots, n-1\} \to D) \to D \). The difficulty is resolved in the following definition.

Definition 6. Let \( T \) be a theory of first-order predicate logic not containing a function symbol \( f \). Let \( M \) be an interpretation for \( T \) and let \( t \) with set \( V \) of variables be a term of \( T \). The extension of \( T \) by \( f \) is a theory \( T' \) with function symbol \( f \) and otherwise identical to \( T \). The extension of \( M \) by \( f \) and \( t \) is an interpretation \( M' \) that is identical to \( M \) except that \( M' \) assigns to \( f \) the function of type \( (\{0, \ldots, n-1\} \to D) \to D \) with map \( (a_0, \ldots, a_{n-1}) \mapsto M_A(t) \) (footnote 1) where
\[
A = (a_0, \ldots, a_{n-1}) \circ (x_0, \ldots, x_{n-1})^{-1} = \begin{vmatrix}
  a_0 & \cdots & a_{n-1} \\
  x_0 & \cdots & x_{n-1}
\end{vmatrix}
\]
and \( (x_0, \ldots, x_{n-1}) \) (see footnote 2) is some enumeration of \( V \).

1 See Definition 3 for the meaning of \( M_A(t) \).
2 This tuple is a function of type \( \{0, \ldots, n-1\} \to V \). The inverse \( (x_0, \ldots, x_{n-1})^{-1} \) exists because all variables in the tuple are different.
Example: $t = x^2 + 2y^2 + 3z^2$, $V = \{x, y, z\}$, $D = R$.

With $A = (3, 2, 1) \circ (x, y, z)^{-1} = \frac{x^2y^2z^2}{3^22^21^2} \in \{x, y, z\} \rightarrow R$ we get $f(3, 2, 1) = M'_A(t) = 3^2 \cdot 2^2 + 3 \cdot 1^2 = 20$.

With a different enumeration of the variables we get a different function. E.g.,

$A = (3, 2, 1) \circ (y, z, x)^{-1} = \frac{y^2z^2x^2}{1^23^22^2} \in \{x, y, z\} \rightarrow R$ we get $f(3, 2, 1) = M'_A(t) = 1^2 \cdot 3^2 + 2 \cdot 3 \cdot 1^2 = 31$.

Definition 6 is a semantic one, so has no commitment to any particular syntax. Syntax for the definition is only of secondary concern in this paper. However, we do want to remark here that specifying the extensions to $T$ and $M$ by writing

$$f \text{ def } \lambda(x_0, \ldots, x_{n-1}), t$$

(1)
carries the necessary information. The use of lambda suggests the intent of the definition that to evaluate $f(t_0, \ldots, t_{n-1})$ one has to pair the $x_i$ in $t$ with the values of the $t_i$. However, this has no formal connection to lambda calculus: the meaning of (1) is determined by the interpretation for $f$ specified in Definition 6.

Iterated extensions of interpretations for new function symbols When an interpretation has been extended by new function symbols, one can repeat a similar process where new function symbols are defined in terms of the function symbols that have already been introduced. This is natural from a programming point of view: a function subroutine often contains calls to subroutines defined in the same program.

We therefore define the iterated extension of order $n$ for $n = 0, 1, 2, \ldots$

– The iterated extension of order 0 is the extension according to Definition 6.

– The iterated extension of order $n > 0$ is the extension according to Definition 6 when the interpretation $M$ has incorporated all iterated extensions of orders $0, \ldots, n - 1$.

Definition of relations Let $F$ be a formula with $V$ as set of free variables. Then $M(F)$ is a relation of type $V \rightarrow D$. $F$ can be a formula with quantifications nested arbitrarily deeply, with many relation and function symbols. Such complexity, together with intricate patterns of shared variables within the same scope give the programmer a powerfully expressive tool for the definition of new relations out of existing ones.

However, a new relation symbol $p$ needs to be interpreted by a relation of type $\{0, \ldots, n-1\} \rightarrow D$, where $n$ is the number of free variables in $F$. Theorem 1 suggests how to resolve the type mismatch.

**Definition 7.** Let $T$ be a theory of first-order predicate logic not containing the relation symbol $p$. Let $M$ be an interpretation for $T$ and let $F$ be a formula of $T$. The extension of $T$ by $p$ is a theory $T'$ with relation symbol $p$ and otherwise identical to $T$. The extension of $M$ by $p$ and $F$ is an interpretation $M'$ that...
is identical to $M$ except that $M'$ assigns to $p$ the relation $M(F)/(x_0, \ldots, x_{n-1})$ where $(x_0, \ldots, x_{n-1})$ is some enumeration of the variables in $F$.

Definition 7 is a semantic one, so has no commitment to any particular syntax. Syntax for the definition is only of secondary concern in this paper. However, we do want to remark here that specifying the extensions to $T$ and $M$ by writing

$$p \text{ def } \lambda(x_0, \ldots, x_{n-1}). F$$

(2)
carries the necessary information. The use of lambda suggests the intent of the definition that to determine the truth value of $p(y_0, \ldots, y_{n-1})$ one has to pair the $x_i$ with the $M(y_i)$. However, this has no formal connection to lambda calculus: the meaning of (2) is determined by $M(F)/(x_0, \ldots, x_{n-1})$ being the interpretation for $p$.

Example

Let $M(\text{sum}) = \{ h \in \{0, 1, 2\} \to \mathcal{R} \mid h_0 + h_1 = h_2 \}, V = \{x, y\}$, and $F = \text{sum}(x, x, y)$. With $p \text{ def } \lambda(x, y). F$ and $q \text{ def } \lambda(y, x). F$ we get e.g.

$p(6, 3) \iff M(\text{sum}(x, x, y)/(x, y))(6, 3) \iff M(\text{sum}(6, 6, 3)) \iff \text{false}$

$q(6, 3) \iff M(\text{sum}(x, x, y)/(y, x))(6, 3) \iff M(\text{sum}(3, 3, 6)) \iff \text{true}$

5 Recursively defined extensions

So far we have assumed that the introduced function or relation symbol does not occur in the defining term or formula. If we allow the introduced function symbol to occur in the defining term we allow the possibility of the resulting function to be partial. As we stay within classical first-order predicate logic, where functions are total, we impose the restriction that the definiendum cannot occur in the definiens.

However, in first-order logic, relation symbols are interpreted by relations. As the interpretation of $p$ with $n$ arguments can be any subset of $\{0, \ldots, n-1\} \to D$, including the empty subset, no such obstacle exists for the definition of new relation symbols. In this section we consider the case where the definition is recursive in the sense of the defining formulas containing new relation symbols.

Because of the absence of recursive definitions of new functions we can suppose all new function symbols introduced by extensions of all orders to have been replaced by their definition before considering the semantics of the recursive definitions of the relations. This is only necessary for theoretical purposes; in practice one leaves the function definitions in place to have the advantage of a compact theory.

Definition 8. Among interpretation extensions with fixed function interpretations, an interpretation $M$ is included in $M'$ iff $M(p) \subseteq M'(p)$ for all new relation symbols $p$. A mapping $T$ from the set of interpretations to itself is said to be monotonic iff $M$ is included in $M'$ implies that $T(M)$ is included in $T(M')$, where $M$ and $M'$ have the same interpretation for their function symbols.

Thus we find that the desire to stay within first-order predicate logic suggests unrestricted definitions of new relations based on a fixed repertoire of given
relations and given functions. The use of logic proposed here has not proposed any inference system. Yet we find something in common with logic programming, where only relations are defined by the program and the function symbols have fixed interpretations. A difference is that logic programming also fixes the universe of discourse to be the Herbrand universe. Here the fixed interpretations for function symbols are freely chosen functions over arbitrary universes of discourse.

We consider simultaneous definitions \( p_i \) def \( \lambda(x_i, 0, \ldots, x_i, m_i - 1).F_i \). How to extend a given theory with these relation symbols? According to Definition \( 7 \) the extended interpretation assigns to the new relation symbols \( p_0, \ldots, p_{n - 1} \) the relations \( M(p_0), \ldots, M(p_{n - 1}) \) that satisfy the equations

\[
\begin{align*}
M(p_0) &= M(F_0)/(x_0, 0, \ldots, x_{0, m_0 - 1}) \\
\cdots &= \cdots \\
M(p_{n - 1}) &= M(F_{n - 1})/(x_{n - 1, 0}, \ldots, x_{n - 1, m_{n - 1} - 1})
\end{align*}
\]

The variables are local to each of the right-hand sides separately. One can see this by observing that a systematic renaming of the variables in a right-hand side does not change the meaning of that expression.

In general we cannot say anything about existence and uniqueness of solutions. Let us consider one example of a condition on \( F_0, \ldots, F_{n - 1} \) that ensures a unique solution: that these formulas are existentially quantified conjunctions of atomic formulas with the right sets of free variables. When such formulas are translated to clausal form they are right-hand sides of Horn clauses. Let us call such formulas “Horn formulas”, even though they do not represent Horn clauses in their most general form.

**Theorem 5.** The equations in (3) have a unique least solution if \( F_0, \ldots, F_{n - 1} \) are Horn formulas.

**Proof.** Because the formulas in Equation 3 are Horn formulas, the right-hand sides in 3 constitute a monotonic mapping on the set of interpretations with fixed function interpretations. The monotonicity implies that 3 has a unique least solution.

**Example: Euclid’s algorithm in a Euclidean domain**

The logical theory has constants zero and unit, binary function symbols + and *, and binary relation symbol <. We extend the theory by the definition \( \text{gcd} \) def \( \lambda(x, y, z).F_0 \) where \( F_0 \) is the Horn formula

\[
\begin{align*}
\text{gcd}(x, y, z) &\leftarrow x < y \land \text{gcd}(x, y - x, z) \\
\text{gcd}(x, y, z) &\leftarrow y < x \land \text{gcd}(x - y, y, z) \\
\text{gcd}(x, y, z) &\leftarrow y = x \land z = x
\end{align*}
\]

where we write \( A \leftarrow B \) for \( A \lor \neg B \). The meaning of \( \text{gcd} \) is given as the least solution of 3 where \( n = 1, p_0 \) is \( \text{gcd} \) and \( \{x_0, 0, \ldots, x_{0, m_0 - 1}\} \) is \( \{x, y, z\} \).
6 Implementation of interpretations

So far all we have done is to evaluate logic on its merits as a programming language. This would be futile without a way to use a computer to obtain results that are verified by a logic theory. In this section we describe a method for this purpose.

Our starting point is the way a logical theory is used to define an abstract mathematical concept. Take for example the concept of group. Whether a structure is a group is determined by the group axioms, which are formalized as a theory of logic. The criterion is whether the structure, regarded as an interpretation, makes the theory a true sentence. Many different structures are groups according to this criterion. Many computer applications can be analyzed in terms of mathematical structures: numbers of various kinds, strings, n-ary relations, vectors, matrices, graphs, partially-ordered sets, ... Axiomatizations of these structures have been expressed as logical theories or are candidates for such treatment. The values to be computed appear as values of functions or as arguments to relations. These functions and relations occur in a logical theory or, more likely, as extensions defined in the way described in this paper.

One way of combining logic and a computer application is to arrange the operations of the computer in such a way that they can be interpreted as inferences from an axiomatic theory. This is what is done in logic programming, where resolution is the inference rule. In this paper we propose a different way. We do not use an inference system. We use the fact that a given theory can be satisfied by two different interpretations, say, $A$ and $B$. $A$ is the familiar mathematical structure. $B$ a program in a conventional programming language that is compiled and executed in the conventional way. If $B$ is also sufficiently similar to an interpretation of the theory and if this interpretation satisfies the theory, then we can say that $A$ is a specification of $B$ and that $B$ is verified with respect to $A$. If $B$ is written in a language like C++, then there is the possibility that it makes optimal use of the computer’s hardware.

Consider the mathematical concept of a Euclidean domain. The structure is axiomatized as having as functions commutative addition, subtraction and commutative multiplication with multiplication distributing over addition. It contains 0 as the neutral element for addition and 1 as the neutral element for multiplication. The integers are an example of a structure that satisfies the Euclidean domain axioms. As another example of such a structure consider the set of bit patterns stored in computer memory and operations on them implemented by hardware instructions or software programs.

*In so far as the resulting structure satisfies the Euclidean domain axioms, these instructions and programs are verified as being a correct implementation of a Euclidean domain.*

This fact is the basis of the method of using logic for programming that we propose in this paper. It remains to find a convenient way of tying together subroutines and a type that can be regarded as an interpretation that can be examined whether it satisfies the intended theory. The class mechanism of C++ offers a
reasonably convenient way of assembling types and subroutines to be regarded as an interpretation for a theory of logic.

We present the class listed below as an interpretation for the axioms for a Euclidean domain extended with the definition of the three-argument relation \( \gcd \). Ideally one should be able to translate the Horn formula (4) to the code

```c
bool gcd(x, y, z){
    if (x<y && gcd(x, y-x, z)) return true;
    if (y<x && gcd(x-y, y, z)) return true;
    z = x; return true;
}
```

In actual fact we needed to clutter up this definition as shown in the listing below. The listing implements one specific Euclidean domain: that of the natural numbers modulo 65521. It has the property that the class is an exact interpretation of the axioms: no approximations are made, nor is the correctness vitiated by the possibility of overflow. The bit patterns in the computer (little-endian two’s complement integers) are one of the many universes of discourse for interpretations satisfying the axioms for Euclidean domains.

class ED{ // ED: Euclidean Domain
    int val; const static int mod = 65521;
    //class invariant: 0 <= val < mod
public:
    ED(): val(0) {}
    ED(int val): val(val) {
        if (val < 0) this -> val = mod - (-val)%mod;
        else this -> val %= mod; }
    static ED zero() { return ED(0); }
    static ED unit() { return ED(1); }
    friend ED operator+(const ED& x, const ED& y)
        { return ED(x.val + y.val); }
    friend ED operator-(const ED& x, const ED& y)
        { return ED(x.val - y.val); }
    friend ED operator*(const ED& x, const ED& y)
        { return ED(x.val * y.val); }
    friend bool operator<(const ED& x, const ED& y)
        { return x.val < y.val; }
    static bool gcd(const ED& x, const ED& y, ED& z){
        if (x<y && gcd(x, y-x, z)) return true;
        if (y<x && gcd(x-y, y, z)) return true;
        z = x; return true;
    }
};
int main() { ED c; ED::gcd(ED(48), ED(36), c); }
7 Conclusions

In this paper we addressed the question of what first-order predicate logic, in its pristine form before there were computers, can do for programming. On the positive side we see terms that range over a universe of discourse (corresponding to the values that program variables assume), function symbols that have functions as interpretation (corresponding to side-effect free function subroutines), and relation symbols that have relations as interpretation (corresponding to side-effect free procedures). On the negative side are (1) no mechanism for defining new functions and relations on the basis of existing ones and (2) it is not clear how to get a computer to evaluate a term or determine the truth value of a formula. Both of these shortcomings have been met in this paper.

As for problem (1), our analysis is that it is caused by the absence of compositional semantics for logic. We corrected this deficiency by introducing a mechanism for extending an existing theory with new function and relation symbols and its interpretation with the corresponding functions and relations.

Our function definitions are not allowed to be recursive. This restriction is forced by the fact that function symbols are interpreted by total functions. Lifting this restriction has been the subject of much research, a sample of which is found in [2]. For us this is not a high priority, as the restriction is no obstacle to making definitions of new relation symbols recursive.

This leaves us with a language in which the programmer can define new relations in mutual recursion on the basis of existing relations and a repertoire of total functions (some coming from the axiomatic theory, some programmer-defined according to the mechanism described in this paper) that is fixed in the context of the relational definitions. Logic programming is more restricted: the universe of discourse is the Herbrand universe, therefore entirely determined by the function symbols of the theory, and the same holds for their interpretations. In our approach the universe of discourse can be any data types that are representable in a computer memory; the functions can be any total functions definable as first-order terms.

The meaning of our recursive definitions of relation symbols is determined by a set of equations. We restrict ourselves to a simple special case in which these equations are known to have a unique least solution. We call the formulas of this special case “Horn formulas” as they correspond to a subset of the Horn clauses if translated to clausal form.

Let us now consider problem (2), how to connect the logical theory and its extensions to a computer in a way that optimally uses its hardware, including its arithmetic. In logic programming, relations are defined by Horn clauses. The computer is used to carry out resolution inference to obtain a logical consequence of the definitions. In our method we use no inference. Instead we use the fact that a theory of logic is agnostic about its interpretations. According to our method we write a program that can be regarded as an interpretation for this same theory. Our counterpart of the elements of the Euclidean domain are two’s complement little-endian bit patterns that behave according to some Intel manual. The fact that the program is also an interpretation of a theory that is
satisfied by Euclidean domains verifies the program as a correct implementation of this abstract algebraic structure.

To our knowledge no programming language exists that allows one to specify an interpretation for a theory in first-order predicate logic. Our method is interesting because one can approach this ideal by writing in C++ a class with functions and relations that correspond closely enough to those of a Euclidean domain. We extend our theory by a three-place relation for \textit{gcd}.

The correspondence between the C++ definition of the algorithm and the logical formula extending the theory is far from perfect, but significant. That anything like this is possible at all is a marvel, considering that C++ started out as “C With Classes” \cite{stroustrup1994} and has remained constrained by compatibility with C during its formative years. We rejected Java as a language for interpretations because of its reliance on heap storage allocation. Our hunch is that there are plenty of interesting algorithms that only need stack storage allocation, which is what we see exclusively in the listing in Section \ref{sec:listing}.

The results in this paper suggest research both in logic and in programming languages. First-order predicate logic, as we have inherited it from the early twentieth century, is only suited for the formalization of small axiom systems. It works fine for a group. But it fails already for something as mundane as a Euclidean domain (“a ring with cancelation law and a valuation, where a ring is a commutative additive group as well as a multiplicative monoid” \cite{birkhoff1970}). The mechanisms developed for programming languages may be helpful here. On the programming-language side C++ is an encouraging example. In spite of its having evolved under the constraint of compatibility with C, it seems the best existing vehicle for implementing interpretations of logic theories that run efficiently. A language for implementing interpretations of logic theories that is similar to C++, but released from the constraint of compatibility with a primitive language, may be an advance in programming languages not seen since the main paradigms, exemplified by Fortran, Lisp, Algol, Simula, Prolog, Smalltalk, and ML, were all in place.

\section*{Acknowledgments}

Thanks to Paul McJones and the referees for help in many ways. This research benefited from facilities provided by the University of Victoria and by the Natural Science and Engineering Research Council of Canada.

\section*{References}

1. Garrett Birkhoff and Thomas Bartee. \textit{Applied Modern Algebra}. McGraw-Hill, 1970.
2. Robert Cartwright. Recursive programs as definitions in first order logic. \textit{SIAM Journal of Computing}, vol. 13 (1984), pages 374 – 408.
3. Alexander Stepanov and Paul McJones. \textit{Elements of Programming}. Addison-Wesley, 2009.
4. Bjarne Stroustrup. \textit{The Design and Evolution of C++}. Addison-Wesley, 1994.