*TPLP* **20** (1): 99–146, 2020. 

doi:10.1017/S1471068419000024 First published online 25 February 2019

# *In Praise of Impredicativity: A Contribution to the Formalization of Meta-Programming*[*]

FRANÇOIS BRY
*Institute for Informatics, Ludwig-Maximilian University of Munich, Germany*
(*e-mail:* bry@lmu.de)

*submitted 31 January 2018; accepted 13 January 2019*

**Abstract**

Processing programs as data is one of the successes of functional and logic programming. Higher-order functions, as program-processing programs are called in functional programming, and meta-programs, as they are called in logic programming, are widespread declarative programming techniques. In logic programming, there is a gap between the meta-programming practice and its theory: The formalizations of meta-programming do not explicitly address its impredicativity and are not fully adequate. This article aims at overcoming this unsatisfactory situation by discussing the relevance of impredicativity to meta-programming, by revisiting former formalizations of meta-programming, and by defining Reflective Predicate Logic, a conservative extension of first-order logic, which provides a simple formalization of meta-programming.

*KEYWORDS*: logic programming, meta-programming, model theory, Barber Paradox, Russell's Paradox, reflection

## 1 Introduction

Processing programs as data is one of the successes of functional and logic programming. Indeed, in most functional and logic languages, programs are standard data structures which release programmers writing program-processing programs from explicitly coding or importing parsers. The following program, in which upper case characters are variables, specifies beliefs of Ann and Bill using the programming style called meta-programming:

```
believes(ann, itRains)
believes(ann, itIsWet ← itRains)
believes(bill, X) ← believes(ann, X)
```

This program's intended meaning is that Ann believes that it rains, Ann believes that it is wet when it rains, and Bill believes everything Ann believes. This program is a meta-program because its second fact

[*] The author is thankful to Norbert Eisinger, Bob Kowalski, and Antonius Weinzierl for fruitful discussions on the subject of this article. The author acknowledges useful hints from the journal area editor Michael J. Maher, from the anonymous reviewers, and from the audience of the 21st International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2017) during which part of the work reported about in this article has been presented in a talk without associated publication. The author is thankful to Norbert Eisinger and Elke Kroiß for their help in correcting typos and stylistic lapses in drafts of this article.

```
believes(ann, itIsWet ← itRains)
```

includes a clause:

```
itIsWet ← itRains
```

This fact violates the syntax of classical predicate logic that requires that a fact is formed from a predicate, like `believes`, and a list of terms like `ann` but unlike the clause `itIsWet ← itRains`. Indeed, in classical predicate logic a clause is a formula, not a term.

Examples referring to beliefs and trust are given in this article because they are intuitive. However, this article does not address how to specify belief and trust systems but instead how to formalize meta-programming, a technique using which such systems can be specified.

While most logics, especially classical predicate logic, prescribe a strict distinction between terms and formulas, meta-programming is based upon disregarding this distinction. Both Prolog and most formalizations of meta-programming pay a tribute to this dictate of classical logic: They require to code a clause like

```
itIsWet ← itRains
```

as a compound term like

```
cl(itIsWet, itRains)
```

or as an atomic term commonly denoted (using a so-called "quotation" ⌜ . ⌝) as follows:

```
⌜itIsWet ← itRains⌝
```

when it occurs within a fact, expressing the second clause above in one of the following forms:

```
believes(ann, cl(itIsWet, itRains))
believes(ann, ⌜itIsWet ← itRains⌝))
```

Such encodings or quotations are not necessary. Atomic and compound formulas can be treated as terms, as HiLog (Chen *et al.* 1993) and Ambivalent Logic (Jiang 1994; Kalsbeek and Jiang 1995) have shown. An expression such as

```
likes(ann, bill)
```

(with the intended meaning that Ann likes Bill) is built up from the three symbols `likes`, `ann`, and `bill` that all three can be used for forming nested HiLog, Ambivalent Logic, and Reflective Predicate Logic expressions such as

```
likes(ann, likes(bill, ann))
```

(with the intended meaning that Ann likes that Bill likes her). As a consequence, the Wise Man Puzzle suggested in the article McCarthy *et al.* (1978) as a benchmark for testing the expressive power and naturalness of knowledge representation formalisms can be expressed in HiLog, Ambivalent Logic, and Reflective Predicate Logic exactly as it is expressed in the article Kowalski and Kim (1991).

Like Ambivalent Logic (Jiang 1994; Kalsbeek and Jiang 1995), but unlike HiLog (Chen *et al.* 1993), Reflective Predicate Logic also allows expressions such as

```
(loves ∧ trusts)(ann, bill)
```

that can be defined by

```
(loves ∧ trusts)(X, Y) ← loves(X, Y) ∧ trusts(X, Y)
```

or more generally by

```
(P1 ∧ P2)(X, Y) ← P1(X, Y) ∧ P2(X, Y)
```

and expressions such as

```
likes(ann, (bill ∧ charlie))
```

that can be defined by

```
P(X, (Y ∧ Z)) ← P(X, Y) ∧ P(X, Z)
```

Even more general expressions like the following are possible in Ambivalent Logic and Reflective Predicate Logic:

```
(∀ T trust(T) ⇒ T)(ann, bill)
```

or, in a program syntax with implicit universal quantification

```
(T ← trust(T))(ann, bill)
```

with the intended meaning that Ann trusts Bill, expressed as `T(ann, bill)`, in all forms of trust specified by the meta-predicate `trust`. If there are finitely many forms of trust, that is, if `trust(T)` holds for finitely many values of `T`, then this intended meaning can be expressed by the following rule that relies on negation as failure:

```
(T ← trust(T))(X, Y) ← not (trust(T) ∧ not T(X, Y))
```

The expression `(T ← trust(T))(ann, bill)` can also be proven in the manner of Gerhard Gentzen's Natural Deduction (Gentzen 1934; Gentzen 1969; Gentzen 1964) by first assuming that `trust(t)` holds for some surrogate `t` form of trust that does not occur anywhere in the program, then proving `t(ann, bill)` and finally discarding (or, as it is called "discharging") the assumption `trust(t)`. This second approach to proving

```
(T ← trust(T))(ann, bill)
```

is, in contrast to the first approach mentioned above, applicable if there are infinitely many forms of trust.

Even though Prolog's syntax does not allow compound predicate expressions such as

```
(loves ∧ trusts)
(T ← trust(T))
```

such expressions make sense.

Reflective Predicate Logic has, in contrast to HiLog and Ambivalent Logic, an unconventional representation of variables. Its syntax adopts the paradigm "quantification makes variables." Thanks to this paradigm, one can construct from the expression `p(a, b)` in which `a` and `b` do not serve as variables, the expression `∀ a p(a, b)` in which `a` serves as a variable and `b` does not serve as a variable. The paradigm "quantification makes variables" makes it easy to generate from an expression `t(ann, bill)` a

quantified expression ∀ `t t(ann, bill)` which, as observed above, is needed in proving implications in the manner of Natural Deduction (Gentzen 1934; Gentzen 1969; Gentzen 1964). The paradigm "quantification makes variables" eases meta-programming as the following example shows. The formula

```
(believes(charlie,itRains) ∧ believes(charlie,¬itRains))
```

(with the intended meaning that Charlie believes both, that it rains, and that it does not rain) can easily be used in generating the (arguably reasonable) assertion

```
(∃ itRains
 (believes(charlie,itRains)∧believes(charlie,¬itRains)))
 ⇒ ∀ X believes(charlie, X)
```

(with the intended meaning that if Charlie believes something and its negation, then Charlie believes everything) and also

```
∀ charlie (∃ itRains
 ((believes(charlie,itRains)∧believes(charlie,¬itRains)))
 ⇒ ∀ X believes(charlie, X)
```

(with the intended meaning that everyone believing something and its negation believes everything).

A price to pay for the paradigm "quantification makes variables" is that, in contrast to the widespread logic programming practice, universal quantifications can no longer be kept implicit. This is arguably a low price to pay since explicit universal quantifications are beneficial to program readability and amount to variable declarations that, since ALGOL 58 (Perlis and Samelson 1958; Backus 1959), are considered a highly desirable feature of programming languages. Furthermore, explicit quantifications make system predicates like Prolog's `var/1` that do not have a declarative semantics replaceable by declarative syntax checks because the presence of explicit universal quantifications distinguishes non-instantiated from instantiated variables. Another consequence of the paradigm "quantification makes variables" is that Reflective Predicate Logic has no open formulas. This is, however, not a restriction, since open formulas have no expressivity in their own and serve only as components of closed formulas. It is even an advantage: Without open formulas, models are simpler to define.

Reflective Predicate Logic can be seen as a late realization, or rehabilitation, of Frege's logic (Frege 1879; Frege 1893; Frege 1903). Except for the representation of variables, the syntax of Reflective Predicate Logic is a systematization of the syntax of Frege's logic (see Section 3). Therefore, the model theory given below can be seen as a model theory for Frege's logic. The name "Frege's logic" would have been given to the logic of this article if not for Frege's anti-democratic and anti-Semitic views.

This article is structured as follows: Section 1 is the introduction. Section 2 considers a few meta-programs that motivate requirements to formalizations of meta-programming. Section 3 reports on related work. Section 4 recalls why predicativity has been sought for and why impredicative atoms are acceptable. Section 5 defines the syntax of Reflective Predicate Logic that allows impredicative atoms under the paradigm "quantification makes variables." Section 6 discusses the expression of the Barber and Russell's Paradoxes in Reflective Predicate Logic. Section 7 gives a variant test for Reflective Predicate Logic

expressions that is needed for the model theory of Reflective Predicate Logic. Section 8 defines a model theory for Reflective Predicate Logic. Section 9 paves the way to the following section by recalling that symbols can be overloaded (as it is called in programming) in classical predicate logic languages. Section 10 shows that Reflective Predicate Logic is a conservative extension of first-order logic. Section 11 concludes the article by discussing its contributions and giving perspectives for further work. A brief introduction into Frege's logic is given in the Appendix.

The main contributions of this article are as follows:

1. A discussion of how Prolog-style meta-programming relates to Frege's logic, type theory, impredicativity, and Russell's Paradox of self-reflection.
2. A formalization of meta-programming which is simple and a conservative extension of first-order logic.
3. A model theory realizing Frege's initial intuition that impredicative, or reflective, predicates can be accommodated in a predicate logic.

## 2 Requirements to formalizations of meta-programming

This section introduces two Prolog meta-programs so as to stress some aspects of meta-programming. The first meta-program is the well-known program `maplist` (Sterling and Shapiro 1994; O'Keefe 1990):

```
maplist(_, [], []).
maplist(P, [X|Xs], [Y|Ys]) :-
   call(P, X, Y),
   maplist(P, Xs, Ys).
```

The third argument of `maplist` is the list obtained by applying the first argument of `maplist`, a binary predicate `P`, to each element of the second argument of `maplist` which is expected to be a list. If `twice/2` is defined as

```
twice(X, Y) :- Y is 2 * X,
```

then `maplist(twice, [0,1,2], [0,2,4])` holds. `maplist` is a meta-program because of the Prolog expression `call(P, X, Y)` which builds from bindings of the variables `P` and `X` like `P = twice` and `X = 1` a fact like `twice(1, Y)` and evaluates it. If the program `maplist` would be seen as a set of classical predicate logic clauses, then `call(P, X, Y)` would be expressed as `P(X, Y)`, the variable `P` would be a second-order variable because it ranges over predicate symbols (like `twice`) and the other variables would be first-order variables because they range over first-order terms (like integers or lists of integers). The semantics of the meta-program `maplist` can be conveyed by an infinite set of ground atoms such as:

```
maplist(twice, [0,1,2], [0,2,4])
maplist(length, [[a], [b,c]], [1,2])
maplist(reverse, [[a,b], [c,d,e]], [[b,a], [e,d,c]])
```

The semantics of many, but not all, meta-programs can be similarly conveyed by ground atoms.

The second example is a meta-program the semantics of which is not appropriately conveyed by ground atoms:

```
studyProgram(mathematics).
studyProgram(computing).

syllabus(mathematics, logic).
syllabus(computing,   logic).
syllabus(computing,   compilers).

enrolled(student(anna), mathematics).
enrolled(student(ben),  computing).

attends(anna, logic).
attends(ben,  compilers).

student(X) :- enrolled(student(X), _)

course(C) :- syllabus(_, C).

forall(R, F) :- not (R, not F).
```

The facts specify two study programs, their syllabi, the enrolments of students in study programs, and the courses' attendance. The clauses with heads `student(S)` and `course(C)` extract the students' names and courses' titles, respectively, from the enrolments and syllabi. The clause with head `forall(R, F)` serves to check properties like whether all mathematics students attend the course on logic:

```
forall(enrolled(student(S), mathematics),
               attends(S, logic)
)
```

or whether all students attend all the courses listed in their study programs' syllabi:

```
forall(enrolled(student(S), P),
               forall(syllabus(P, C), attends(S, C)
               )
)
```

`forall` is a meta-program because during its evaluation its arguments `R` and `F` are themselves evaluated. In the clause defining `forall`, the left occurrences of `R` and `F` correspond to classical logic terms, while the right occurrences of the same variables correspond to classical logic formulas. `forall` is a well-known Prolog meta-program which implements the failure-driven loop already mentioned in the introduction. It is representative of reflection in meta-programming, that is, the expression and the processing of formulas in meta-programs. The evaluation of, for example,

```
forall(p(X), q(X))
```

consists in a search for an instance of `p(X)` without corresponding instance of `q(X)`. If the search fails, then the evaluation succeeds. Thus, the evaluation of

```
forall(p(X), q(X))
```

can be conveyed in classical logic by the formula:

```
¬∃X (p(X) ∧ ¬q(X))
```

which, in classical logic, is logically equivalent to:

```
∀X (p(X) ⇒ q(X))
```

As a consequence, the instance of the clause defining the predicate `forall`

```
forall(p(X), q(X)) :- not (p(X), not q(X))
```

corresponds to the classical logic formula

```
(∀X (p(X) ⇒ q(X)) ⇒ ∀Y forall(p(Y), q(Y)))
```

(in which, `forall` is, like in the Prolog program, a predicate and `X` and `Y` are distinct variables) but does not correspond to the universal closure of the aforementioned clause instance:

```
∀X (p(X) ⇒ q(X)) ⇒ forall(p(X), q(X)))
```

(in which only one variable occurs). As a consequence, ground instances of the meta-program

```
forall(R, F) :- not (R, not F)
```

like

```
forall(p(a), q(a))
```

do not convey that meta-progam's semantics. Non-ground instances like

```
forall(p(X), q(X))
```

are necessary to properly convey the semantics of `forall`.

In this, `forall` is not a rare exception. Another example is the meta-program `hasSingVar(C)` that checks whether a clause `C` contains at least one singleton variable. Its semantics cannot be described by ground instances like:

```
hasSingVar(p(a) :- q(a, b))
```

Indeed, such a ground instance does not make sense. In contrast, a non-ground expression

```
hasSingVar(p(X) :- q(X, Y))
```

does convey the semantics of the meta-program `hasSingVar(C)`. More generally, the semantics of many reflective meta-programs, of many meta-programs performing program analyses, and of meta-programs generating improved program versions cannot be properly expressed by the ground expressions of standard Herbrand interpretations.

A non-ground expression like the aforementioned one used for conveying a meta-program's semantics does not stand for the set of its ground instances. Thus, an explicit quantification like in

```
hasSingVar(∀ X ∀ Y p(X) :- q(X, Y))
```

better conveys the semantics of meta-programs. Conveying meta-programs' semantics with such non-ground expressions generalizes Herbrand interpretations.

Summing up, three aspects of meta-programming have been stressed in this section:

1. *Self-reflective predicates*: Some meta-predicates are self-reflective in the sense that they can occur within their own arguments like `forall` in:

```
forall(enrolled(student(S), P),
                forall(syllabus(P, C), attends(S, C)
                )
)
```

2. *Confounding of object and meta-variables*: Some meta-programs contain occurrences of a same variable where classical logic expects a term and where classical logic expects a formula or a predicate like in:

```
forall(R, F) :- not (R, not F)
```

3. *Need for generalized Herbrand interpretations*: The semantics of some meta-programs like `forall` and `hasSingVar(C)` is not properly conveyed by the ground atoms of standard Herbrand interpretations. It is appropriately conveyed by generalized Herbrand interpretations specified by non-ground and quantified expressions.

In the following section, the adequacy of formalizations of meta-programming is assessed by referring to the aforementioned three aspects of meta-programming.

## 3 Related work

This article relates to the many formalizations of meta-programming that have been proposed. These formalizations are of three kinds:

- Formalizations interpreting meta-programs as higher-order theories
- Formalizations interpreting meta-programs as first-order theories
- Formalizations interpreting meta-programs as theories in non-classical logics

The formalizations of meta-programming all relate to type theory. Therefore, type theory is discussed below before the formalizations of meta-programming. As usual, the phrase "type theory" is used to refer either to the research field devoted to type theories or to a given type theory. This article also relates to reflection in computing, knowledge representation and logic, and, because of examples it mentions, logics of knowledge and belief.

*Meta-programming* has been considered since the early days of logic programming. It is discussed in the articles Kowalski (1979) and Bowen and Kowalski (1982). Meta-programming in Prolog is addressed among others in the works Clocksin and Mellish (1981), Sterling and Shapiro (1994), and O'Keefe (1990). However, the standard formalization of logic programming (Lloyd 1987) does not cover meta-programming. The articles Barklund (1995) and Costantini (2002) survey meta-programming.

Prolog's extremely permissive approach to meta-programming goes back to a fruitful disregard by Alain Colmerauer and Philippe Roussel, Prolog's designers, of the relationship between meta-programming, higher-order logic, impredicativity, and type theory and to a time at which the undecidability of unification in second-order logic (Levy and Veanes 2000) and third-order logic (Huet 1976) as well as Damas–Hindley–Milner type systems (Hindley 1969; Milner 1978; Damas and Milner 1982; Damas 1985) were unknown or not widely known. Prolog's permissive approach to meta-programming is very useful in practice, as the following examples demonstrate:

- A unary predicate ranging over all unary predicates (including itself) can be used for (static or dynamic) type checking.
- A predicate occurring in an argument of itself can be used for applying an optimization to the very predicate specifying this optimization.
- Formulas occurring in places where classical predicate logic expect terms are useful as it is shown in the articles Kowalski (1979) and Kowalski and Kim (1991), in the introduction, and in the previous section.

*Type theory.* The first type theory, or theory of types, was developed by Bertrand Russell as a correction of Gottlob Frege's Logic (Frege 1879; Frege 1893; Frege 1903), the archetype of classical predicate logic. Frege's logic is discussed below in Sections 4 and 6 and its salient aspects are recalled in the Appendix. Frege's logic is reflective in the sense that a predicate can apply to a formula or to a predicate, including itself. In Frege's logic, a unary predicate `r` holding of all unary predicates that apply to themselves can be defined as follows:

```
∀x (r(x) ⇔ x(x))
```

As a consequence, Russell's Paradox (Link 2004), which is discussed in more details below in Section 6, can be expressed in Frege's logic. In Frege's logic, some predicates cannot be interpreted as sets as the following set of atomic formulas illustrates in which `a` and `b` are individual constants and `p` and `q` are predicates:

```
{p(a), p(b), q(a), q(q)}
```

If the constants `a` and `b` are interpreted as 1 and 2, respectively, then the predicate `p` can be interpreted as the set $\{1, 2\}$ but the predicate `q` cannot be interpreted as a set because a set cannot be one of its own elements. Russell developed a type theory so as to avoid "impredicative atoms" like `q(q)` and to preclude paradoxes of self-reflexivity like the paradox bearing his name.

Russell successively developed various type theories before publishing with Alfred North Whitehead in Principia Mathematica (Whitehead and Russell 1913) the "Ramified Theory of Types." Leon Chwistek and Frank P. Ramsey later simplified the Ramified Theory of Types yielding the theory now known as the "Simple Type Theory" or "Theory of Simple Types" (Chwistek 1921; Ramsey 1926). Aiming at avoiding paradoxes of the untyped lambda calculus (Church 1932), Alonzo Church re-expressed the Simple Type Theory as a theory which is now commonly called "Church's Simply Typed Lambda Calculus," or $\lambda^{\rightarrow}$, (Church 1940; Barendregt 1993), a typed variation (with a single type constructor, $\rightarrow$, for function types) of the untyped lambda calculus.

Every type theory requires that every symbol, among others every variable, and every compound expression have a single type like first-order term, first-order predicate, first-order formula, second-order predicate, second-order formula, etc. In other words, a type theory imposes a strong typing. This strong typing is Russell's treatment of impredicative atoms like `q(q)` and of paradoxes of self-reflexivity among others, Russell's Paradox: It precludes them. Endowed with, and modified by, the strong typing of a type theory, Frege's logic became classical predicate logic.

Referring to the typing policy of classical predicate logic as a "strong typing" is an anachronism. Indeed that denomination has been introduced only in 1974 in the article Liskov and Zilles (1974) where it is defined as follows: "Whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function." This definition perfectly describes the requirement of a type theory if "object" is understood as "argument" and "function" as "predicate or function," which justifies the anachronism.

A strong typing in the manner of, but different from, the type theory of classical predicate logic has been shown later to be useful for preventing programming errors (Cardelli and Wegner 1985). This has resulted in "type systems" that assign properties to program constructs (Pierce 2002; Cardelli 2004). These properties and type systems depart from the "orders" or "types" of the aforementioned type theories.

Remarkable achievements in type theory (among others the Curry–Howard isomorphism, inductive types, and dependent types) have given type theories an aura of indispensability. This article shows that type theories can be dispensed with: It gives a model theory to a systematization of Frege's logic, Reflective Predicate Logic, a logic without type theory, yielding a simple and intuitive formalization of Prolog-style meta-programming.

*Formalizations interpreting meta-programs as higher-order theories.* Some logic programming languages, most prominently λProlog (Miller and Nadathur 1988; Miller and Nadathur 2012), Elf (Pfenning 1991), and Twelf (Pfenning and Schürmann 1999), are formalized in classical higher-order predicate logics, syntactical restrictions ensuring necessary properties like the decidability of unification.

λProlog is based on the Simply Typed Lambda Calculus (Church 1940; Barendregt 1993). Elf (Pfenning 1991) and Twelf (Pfenning and Schürmann 1999) have been designed conforming to the Edinburgh Logical Framework LF (Harper *et al.* 1993), a predicative language for a uniform representation of the syntax, the inference rules, and the proofs of predicative logics. LF is based on intuitionistic logic (Moschovakis 2015) and on the Simply Typed Lambda Calculus (Church 1940; Barendregt 1993). Surprisingly, LF represents object variables by meta-variables (Harper *et al.* 1993, p. 145):

> "[. . .] substitution in the logical system is encoded as substitution in LF (which relies on the identification of object-logic variables with the variables of LF)."

This identification, or confounding, of object and meta-variables is, in fact, precluded by the Simply Typed Lambda Calculus on which LF is based (as it is precluded by any other type theory). This article can be seen as a justification for LF's confounding of object and meta-variables.

William W. Wadge has proposed in the article Wadge (1991) a fragment of higher-order Horn logic called "definitional programs" as a meta-programming language "based

on Church's Simple Theory of Types" (Wadge 1991), that is, the Simply Typed Lambda Calculus or $\lambda^{\rightarrow}$ (Church 1940; Barendregt 1993). The particularity of Wadge's language is that its semantics fulfils a condition called "extensionality" that makes predicates with a same extension interchangeable. Extensionality is justified in the article Wadge (1991) with the following example:

```
p(a).
q(a).
phi(p).
```

The predicates `p` and `q` are interpreted in the articles Wadge (1991) and Charalambidis *et al.* (2013) as first-order predicates. `phi` is a meta-program interpreted in the articles Wadge (1991) and Charalambidis *et al.* (2013) as a second-order predicate which applies to the predicate `p`. The argument given in the article Wadge (1991) for justifying extensionality is that `phi(q)` should hold like `phi(p)` because `p` and `q` have the same extension. Marc Bezem has given in the article Bezem (1999) under the name of "good programs," a first decidable sufficient condition for extensionality as an improvement over Wadge's "definitional programs," and under the name of "hoapata programs" (Bezem 2001), a second decidable sufficient condition for extensionality as an improvement of the "good programs."

Extensionality is a questionable requirement because it abstracts out the algorithmic aspects of programs as the following example shows. Replace in the aforementioned example `p` by the tail-recursive program `reverse` and `q` be the non-tail-recursive program `naivereverse` (Sterling and Shapiro 1994; O'Keefe 1990). Recall that `reverse` and `naivereverse` have the same extension. Assume that the meta-predicate `phi` is a versioning predicate `final` distinguishing final implementations from less efficient preliminary versions. Since `final(reverse)` holds, `final(naivereverse)` must, by virtue of extensionality, also hold. Clearly, this is not desirable.

All formalizations of meta-programs as higher-order theories adhere to the strong typing of classical predicate logic: They strictly distinguish between terms and formulas and therefore cannot express meta-programs like

```
forall(R, F) :- not (R, not F)
r(X) :- X(X)
```

in which some variables stand for both a term and a formula or a predicate. All formalizations interpreting meta-programs as higher-order theories preclude self-reflective predicates and the confounding of object and meta-variables of Prolog-style meta-programming. The formalizations interpreting meta-programs as higher-order theories do not provide generalized Herbrand interpretations specified by non-ground and quantified expressions. Thus, the formalizations of meta-programs as higher-order theories do not fulfil the requirements stressed at the end of Section 2.

*Formalizations interpreting meta-programs as first-order theories* are based on reifying first-order formulas by encoding them as first-order terms. The advantage of the approach over formalizations of meta-programming in higher-order logic is that it makes possible reflective, including self-reflective, formulas. Its drawbacks are the encodings that make programs complicated and less intuitive than their Prolog-style counterparts. The encodings upon which formalizations of meta-programming are based are called "naming

relations" or "naming schemes." A formalization of meta-programming in first-order logic is largely, if not fully, characterized by its naming relation.

A naming relation refers to two first-order languages, an object language $\mathscr{L}_O$ and a meta-language $\mathscr{L}_M$, for encoding terms, especially variables, and formulas of $\mathscr{L}_O$ as terms of $\mathscr{L}_M$. Some naming relations encode all terms and all formulas of $\mathscr{L}_O$; others encode only some of the terms and formulas of $\mathscr{L}_O$. Naming relations typically encode variables of $\mathscr{L}_O$ as variable-free, or ground, terms of $\mathscr{L}_M$ so as to avoid the confounding of object and meta-variables of Prolog-style meta-programming which, as it is recalled above, is precluded by the strong typing of classical predicate logic. Three kinds of naming relations have been considered:

- A *structure-hiding naming relation* encodes the structure of an object formula as a meta-term the structure of which does not reflect that of the formula.
- A *quotation* encodes an object formula $F$ without conveying its structure, typically as a meta-language constant usually denoted $\ulcorner F \urcorner$.
- A *structure-preserving naming relation* encodes a formula as a meta-term the structure of which reflects that of the formula.

Kurt Gödel and Alfred Tarski first used naming relations for encoding self-reflective formulas in first-order logic that, because of classical predicate logic's strong typing, cannot be directly expressed in that logic. Gödel used the following naming relation (Gödel 1931): If $F$ is the object language formula represented as $s_1 s_2 \ldots s_n$ where the $s_i$ are non-negative integers representing the object language symbols, then $F$ is encoded as the non-negative integer $G(F) = p_1^{s_1} \times p_2^{s_2} \ldots \times p_n^{s_n}$, the Gödel number of $F$, where $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$ is the ordered sequence of the prime numbers. Because of the Unique-Prime-Factorization theorem, a formula $F$ can be reconstructed from its Gödel number $G(F)$, that is, the naming relation is structure hiding. Shortly later, Tarski used quotations for expressing his "Schema $T$" (Tarski 1935), that is, the requirement that every theory of truth has a truth predicate $T$ fulfilling $(T(\ulcorner F \urcorner) \Leftrightarrow F)$ for all formulas $F$.

The first reference to a naming relation in logic programming is probably the article Bowen and Kowalski (1982). In that article, provability of an object language $L$ is expressed in a meta-language $M$ with a predicate *Demo* implementing SLD resolution (Kowalski 1973). *Demo* is inspired from Gödel's predicate *Bew* (Gödel 1931), short for Beweis, that is, proof in German.

Under the naming relation of Bowen and Kowalski (1982), the formulas of the object language $L$ are encoded as terms of the meta-language $M$. The atom `p(X, bill)` is, for example, encoded as the term `atom(pred(1), [var(1), constant(212)])` in which `pred(1)`, `var(1)`, and `constant(212)` are variable-free, or ground, terms of $M$. Thus, the naming relation considered in the article Bowen and Kowalski (1982) is structure preserving. Following a widespread logic programming practice, in both $L$ and $M$ commas denote conjunctions, universal quantifications are implicit and existential quantifications are not used. As a consequence, the article Bowen and Kowalski (1982) can avoid to address that a naming relation in fact requires to encode in the meta-language the logical connectives and quantifiers of the object language.

In the article Bowen and Kowalski (1982), the amalgamation of an object language $L$ and a meta-language $M$ is defined as "$L$ and $M$," meaning $L \cup M$, equipped with:

- a naming relation which associates with every expression of $L$ a variable-free term of $M$,
- a representation of $\vdash_L$ in $M$ by means of a predicate *Demo* specified in M as a theory $Pr$,
- the "linking rules"

$$\frac{Pr \vdash_M Demo(A', B')}{A \vdash_L B} \qquad \frac{A \vdash_L B}{Pr \vdash_M Demo(A', B')}$$

in which $F'$ denotes the encoding of $F$ under the naming relation.

The linking, or reflection or attachment, rules have been proposed in the article Weyhrauch (1980), a formalization of meta-level reasoning. They correspond to the correctness and completeness of Gödel's predicate *Bew* (Gödel 1931) with respect to provability. The article Bowen and Kowalski (1982) states that "the amalgamation of $L$ and $M$ is a conservative extension in the sense that no sentence is provable in the amalgamation that is not in either $L$ or $M$." This statement disregards sentences of the amalgamation that may contain symbols of $L$ but not of $M$ as well as symbols of $M$ but not of $L$ and therefore that are sentences of neither $L$ nor $M$.

The article Bowen and Kowalski (1982) further states that "the amalgamation allows to have $L = M$, where the two languages are identical" meaning that a language may contain an encoding of its own formulas as terms.

The article Barklund *et al.* (1995a) formalizes naming relations as rewrite systems and investigates the expressivity of various naming relations. The article van Harmelen (1992) argues that naming relations can encode, together with an object formula, pragmatic and semantic information resulting in a more efficient (meta-language) version of the original formula. Naming relations have indeed been defined for achieving such "compilations" what explains their large number and, as a consequence, the large number of formalizations of meta-programming in first-order logic: metaProlog (Bowen 1985; Bowen and Weinberg 1985), MOL (Eshghi 1986), the language proposed by Barklund in the article Barklund (1989), Reflective Prolog (Costantini and Lanzarone 1989; Costantini and Lanzarone 1994), R-Prolog* (Sugano 1989; Sugano 1990), 'LOG (spoken "quotelog") (Cervesato and Rossi 1992), Gödel (Hill and Lloyd 1994), the language proposed by Higgins in the article Higgins (1996), and the generalization of Reflective Prolog proposed in the article Barklund *et al.* (2000).

Most formalizations of meta-programming in first-order logic make use of naming relations that are structure preserving because they are "compositional," that is, recursively defined on the expressions' structures, yielding names (or encodings) satisfying

$$\overline{f(g(a), b)} \approx \overline{f}(\overline{g}(\overline{a}), \overline{b}),$$

where $\overline{e}$ denotes the encoding of an expression $e$. The approximation $\approx$ cannot always be replaced by an equality because of cases like the aforementioned one (Bowen and Kowalski 1982): $\overline{p(t_1, t_2)} = \mathrm{atom}(\overline{p}, [\overline{t_1}, \overline{t_2}]) \neq \overline{p}(\overline{t_1}, \overline{t_2})$. Some formalizations of meta-programming in first-order logic make use not only of structure-preserving naming relations but also of quotations giving compound expressions short "names" that, in general, are individual constants of the meta-language. Some other formalizations of meta-programming in first-order logic make use only of quotations. Note that the denomination "quotation" is often

used in the meta-programming literature in the sense of "structure-preserving naming relation" instead of the aforementioned sense of encodings of formulas that hide the formulas' structures. Most formalizations of meta-programming in first-order logic are amalgamations of object and meta-languages in the sense of Bowen and Kowalski (1982) recalled above.

metaProlog (Bowen 1985; Bowen and Weinberg 1985) has two naming relations such that "constants act as names of themselves. For non-constant items, metaProlog provides structural or non-structural names (and sometimes both), where the former are compound terms whose structure reflects the syntactic structure of the syntactic item they name." metaProlog has "metalevel names" that are terms used for representing object programs. Object variables are represented in metaProlog as constants of the meta-language. metaProlog treats theories, or programs, as named entities that, thanks to the aforementioned naming relation, can be referred to in a program. This makes possible that, in metaProlog, goals are proven in reference to a named theory and updates are formalized in logic. metaProlog has an explicit quantification that avoids problems in updating theories similar to those stressed above in Section 2 while discussing the semantics of the meta-program `forall`. metaProlog provides "methods for moving between a name and the thing it names [...] analogous to univ (`=..`) of ordinary Prolog." Thus, in metaProlog, an object language expression can be obtained from its encoding as a meta-term.

MOL (Eshghi 1986) has a structure-preserving naming relation and an involved treatment of reflection through "inheritance and scoping axioms": An inheritance axiom can be used to express that an object-level program $P$ contains the object-level program $Q$; a scoping axiom is used to express that if a ground assertion can be proved from the meta-theory $M$, then this ground assertion is part of a "description" of an object-level program $P$, that is, that $M$ is a meta-theory for $P$. In MOL, an object language expression can be obtained from its encoding as a meta-term. MOL supports self-reflection called "self-reference" in the thesis Eshghi (1986). This thesis shows how reflection and self-reflection can be used in meta-programming and for expressing "non-floundering negation as failure" and a "declarative control [of program execution] without jeopardizing the soundness of the interpreter."

The language proposed by Barklund in the article Barklund (1989) has a naming relation for "a naming of Prolog formulas and terms as Prolog terms" built from reserved function symbols and constants "to create a practical and logically appealing language for reasoning about terms, programs." In this language, an object language expression can be obtained from its encoding as a meta-term.

Reflective Prolog (Costantini and Lanzarone 1989; Costantini and Lanzarone 1994) has a naming relation, called "quotation" in the article, that, for example, encodes, or "names," the term `f(a)` as `function(functor({f}), arity(1), args(["a"]))`. The article mentions that "it is possible to build names of names of names, and so on" but does not explain the use of this feature. Reflection is well supported by Reflective Prolog through an "unquotation mechanism" realized by "a distinguished truth predicate which relates names to what is named." Reflective Prolog is based on an amalgamation of object and meta-languages that are disjoint in the sense that they have no symbols in common. As a consequence, "language and metalanguage are amalgamated in a non-conservative extension": "Statements are provable in the amalgamated language, that

are provable neither in the language nor in the metalanguage alone." Reflective Prolog has an "extended resolution procedure which automatically switches the context between levels" which "relieves the programmer from having to explicitly deal with control aspects of the inference process." Reflective Prolog's "extended resolution is proved sound and complete with respect to the least reflective Herbrand model."

R-Prolog* (Sugano 1989; 1990) has a naming relation called "quotation" and denoted à la Lisp (McCarthy and Levin 1965; Siklóssy 1976) with a single quote: If `t = f(a, b)`, then `'t = 'f('a, 'b)`. R-Prolog* furthermore has predicates ↑ (spoken "up") and ↓ (spoken "down") such that ↑`t` = `'t` and ↓`'t` = `t`. Thus, in R-Prolog*, an object language expression can be obtained from its encoding as a meta-term. R-Prolog* reifies not only object terms and formulas but also substitutions so as to express much of the language's runtime system in its own meta-language. The semantics of R-Prolog* is based on an "extended notion of interpretations and models" that departs from the usual semantics of logic programs based on a fixpoint of the immediate consequence operator or Herbrand interpretations: "Because computational reflection is a procedural notion, we cannot adopt the usual declarative semantics."

'LOG (Cervesato and Rossi 1992) has two naming relations associating "two different but related meta-representations with every syntactic object of the language, from characters to programs," "a constant name and a structured ground term, called the structural representation." This double naming relation generalizes those of Martelli and Rossi (1988) and Rossi (1989; 1992) that apply only to programs. 'LOG has an operator `<=>` that relates the name and the structural representation of each syntactic object. 'LOG has no "mechanism which would allow a meta-representation to be obtained from the object it denotes or vice versa." The article stresses that "this differentiates (both in aims and in nature) our proposal from others, such as Reflective Prolog and R-Prolog*, that, on the contrary, assume a reflection mechanism to be available, though not visible at the user level."

Gödel (Hill and Lloyd 1988; Hill and Lloyd 1994) is equipped with a strong typing system distinguishing object language from meta-language expressions and a naming relation encoding object language expressions as constants of the meta-language called "ground representations." Gödel's naming relation is thus a quotation. Gödel makes the encoding of object language expressions by the naming relation, "explicitly available to the programmer." Gödel expresses much of the language's deduction system in its meta-language: It reifies among other the object language's provability. Gödel has no mechanism for obtaining an object expression from its "ground representation," that is, its encoding in the meta-language. Gödel is a rare case of a formalization of meta-programming in first-order logic which is not an amalgamation of the object and meta-language in the sense of the article Bowen and Kowalski (1982) recalled above.

Higgins has proposed in the article Higgins (1996) a language relying on two naming relations similar to those of 'LOG associating with every syntactic object a first-order "primitive name" and a first-order "structured name": "We have names of symbols, terms, clauses, and sets of clauses." The procedural semantics of Higgins' language is "a resolution rule and a meta-level to object-level reflection rule." In other words, the language's procedural semantics obtains an object language expression from its encoding as a meta-term.

The articles Barklund *et al.* (1995a), Barklund *et al.* (1995b), and Barklund *et al.* (2000) generalize Reflective Prolog (Costantini and Lanzarone 1989; 1994) into a meta-language similar in spirit to LF (Harper *et al.* 1993), Elf (Pfenning 1991), and Twelf (Pfenning and Schürmann 1999) which offers a sublanguage for expressing various kinds of naming relations (or encodings), primitives for handling naming relations, and a meta-language of Horn clauses for expressing object-level inference rules. The article Barklund *et al.* (1995a) considers eight previously proposed naming relations (or encodings) that are compositional in the sense that the name of a compound expression is built from its components' names. The article submits that naming relations should be compositional and shows that compositional naming relations can be expressed as rewrite systems provided the systems preserve truth (and falsity), are confluent, and are terminating. The article Barklund *et al.* (1995b) extends the generalization of Reflective Prolog to SLD Resolution. The article Barklund *et al.* (2000) further specifies a fixpoint semantics for definite programs in the proposed meta-language, establishes the soundness and completeness of SLD resolution with respect to that fixpoint semantics, and gives three examples of meta-programs one of which is a re-implementation of Reflective Prolog (Costantini and Lanzarone 1989; Costantini and Lanzarone 1994).

All formalizations interpreting meta-programs as first-order theories adhere to the strong typing of classical predicate logic: They strictly distinguish between terms and formulas. As a consequence, they cannot directly express meta-programs in which some variables stand for both a term and a formula or a predicate and self-reflective predicates or formulas like:

```
forall(R, F) :- not (R, not F).
r(X) :- X(X).
```

Whether such (encoded) examples can be efficiently processed depends among other things on whether the encoding is invertible, that is, whether a decoding primitive is available that returns from a code the object language expression its encodes and whether such a decoding primitive can be efficiently implemented. If the encoding is efficiently invertible, then adjustments to the deduction methods suffice to implement a language that handles meta-programs by relying on naming relations. metaProlog (Bowen 1985; Bowen and Weinberg 1985), MOL (Eshghi 1986), the language proposed by Barklund in the article Barklund (1989), Reflective Prolog (Costantini and Lanzarone 1989; 1994), R-Prolog* (Sugano 1989; 1990), and the language proposed by Higgins in the article Higgins (1996) have efficiently invertible naming relations. 'LOG (Cervesato and Rossi 1992) and Gödel (Hill and Lloyd 1994) do not have invertible naming relations.

However, as many authors have observed, with a strong typing preventing the confounding of object and meta-variables, much of the object language reasoning, among others unification, must be re-implemented in the meta-language (De Schreye and Martens 1992; Kalsbeek 1993; Levi and Ramundo 1993; Barklund *et al.* 1994; Apt and Ben-Eliyahu 1996). Even if such re-implementations are provided with a language, they are a burden to the programmers. Furthermore, such re-implementations result in significant losses in efficiency. Finally, such re-implementations are redundant since the object language's reasoning is a special case of the meta-language's reasoning. These observations have triggered a debate dubbed "ground versus non-ground representations" that revolves around the following question: If the theory requires that object and meta-languages

expressions, especially variables, be distinguished, then why are well-working deduction systems possible that confound object and meta-variables? So far, the debate was not settled. This article shows that the logic in which meta-programming is formalized can be adapted to Prolog-style meta-programming giving a formal justification to the practice of confounding object and meta-variables in deduction systems.

Summing up, the formalizations of meta-programs as first-order theories do fulfil the first of the three requirements mentioned in Section 2: Through encodings, they can express self-reflective predicates. However, they do not fulfil the second requirement of Section 2: They cannot confound object and meta-variables. Some of these formalizations do not, other partly fulfil the third requirement of Section 2: Those formalizations relying on non-ground encodings do not have generalized Herbrand interpretations specified by non-ground atoms while those formalizations relying on ground encodings can express (ground encodings of) such atoms. Generalized Herbrand interpretations with atoms including quantified formulas have not been considered by the authors of formalizations of meta-programs as first-order theories.

*Formalizations interpreting meta-programs as theories in non-classical logics* adapt first-order logic to fit Prolog-style meta-programming by giving up type theory, which makes reflective, including self-reflective, formulas, and the confounding of object and meta-variables possible. A predicate logic without type theory has already been defined at the end of the 19th century: Frege's logic (Frege 1879; Frege 1893; Frege 1903), the precursor of classical predicate logic. A brief presentation of Frege's logic is given in the Appendix. The following formula (see Section 2) that confounds object and meta-variables can be expressed in Frege's logic:

```
∀ x (r(x) ⇔ x(x))
```

Modified by the addition of a type theory, Frege's logic became classical predicate logic. Unsurprisingly, the formalizations of meta-programming in non-classical logics are closely related to Frege's logic.

The syntax of Frege's logic has been defined before inductive definitions (and grammars as formalisms easing the expression of inductive definitions) were established (see Section 4). As a consequence, Frege did not fully formalize expressions resulting from replacing in an expression a subexpression by its definition according to his logic's Basic Law V (see the Appendix which introduces into Frege's logic). Consider, for example, the following definition of `p`:

```
(p ⇔ (q ∧ r))
```

Replacing `p` by its definition in `p(a)` results in the following expression with compound predicate:

```
(q ∧ r)(a)
```

If `q` is defined by

```
(q ⇔ r(b, c))
```

then replacing `q` by its definition in `q(a)` results in the following "multiple application":

```
(r(b, c))(a)
```

"Multiple applications" are known from currying. Currying, common in Functional Programming but rare in predicate logic, is used in automated deduction among other for implementing efficient term indexes (Graf 1996). The aforementioned substitutions are not possible in first-order logic because of its strong typing that precludes the predicate definitions they result from. Indeed, since `p`, `q`, and `r` are unary predicates, instead of, for example `(p ⇔ (q ∧ r))`, first-order logic would require:

```
∀ x (p(x) ⇔ (q(x) ∧ r(x)))
```

Frege's logic has a proof calculus but no model theory. It was only in 1930 that Gödel would introduce in his doctoral thesis (Gödel 1930) the concepts of interpretation and model in establishing the completeness of Frege's proof calculus for a fragment of Frege's logic, first-order logic. Section 5 gives Frege's logic the systematized syntax discussed above (under the paradigm "quantifications make variables") and Section 8 a Herbrand-style model theory. This results in a simple and intuitive formalization of Prolog-style meta-programming.

HiLog (Chen *et al.* 1993) was the first formalization of meta-programming in a non-classical logic. At first, HiLog seems to be a formalization in first-order logic because the article Chen *et al.* (1993) refers to a type theory and a naming relation. However, HiLog treats every symbol (except connectives and quantifiers) as a predicate and maps it to both

- an "infinite tuple of functions" over the universe, one function of arity $n$ for each $n \in \mathbb{N}$ and
- an "infinite tuple of relations" over the universe, one relation of arity $n$ for each $n \in \mathbb{N}$.

As a consequence, every HiLog expression built without connectives and quantifiers is, in the sense of first-order logic, both a term and an atomic formula. Since every expression being of all types amounts to no expressions being of any type and since a (standard) type theory assigns a single type to an expression, HiLog can be seen as a logic without (standard) type theory. In the article Chen *et al.* (1993), a naming relation is used for encoding HiLog in first-order logic but not for encoding formulas as terms in HiLog: Reflection in HiLog is achieved without naming relation. As a consequence, HiLog can express meta-programs like the following (see Section 2) without resorting to a naming relation:

```
forall(R, F) :- not (R, not F).
r(X) :- X(X).
```

HiLog's syntax is the Horn fragment with implicit universal quantifications of the aforementioned systematized syntax of Frege's logic. Thus, though reflection is possible in HiLog, it is limited to that fragment. The following statement (with the intended meaning that Ann believes that it rains and the sun shines) cannot be expressed in HiLog because HiLog does not allow connectives to occur within an atom:

```
believes(ann, (itRains ∧ theSunShines)).
```

However, the article Chen *et al.* (1993) rightly claims that allowing connectives within atoms is a minimal extension. The first of the following statements cannot be expressed

in HiLog because HiLog does not allow quantifiers within atoms while the second and the third of the following statements can be expressed in HiLog through skolemization:

```
believes(ann, ∃ x (logician(x) ∧ loves(x, ann))).
∃ x (logician(x) ∧ believes(ann, loves(x, ann))).
∃ x believes(ann, (logician(x) ∧ loves(x, ann))).
```

Note the difference between the three aforementioned statements: The first expresses that Ann believes to be loved by a logician, the second expresses the existence of a logician Ann believes to be loved by, the third expresses the existence of an entity Ann believes to be a logician who loves her. In a world without logicians, the first statement can be fulfilled, the second not. In a world without enough entities, the number of which might be limited by the axioms considered, the first statement could be fulfilled and the third not. Such differences are significant in meta-programming, especially in meta-programs performing program analyses. The requirement in a module $A$ for another module $B$ should, for example, not be misinterpreted as stating the existence of a module $B$. The article Chen *et al.* (1993) states that "encoding formulas with quantified variables would require introduction of lambda-abstraction which can be done but is out of the scope of this paper." It is correct that this can be done, though not through lambda-abstraction. It has been done in the articles Jiang (1994) and Kalsbeek and Jiang (1995) in a manner, however, which is not satisfying. An appropriate treatment of quantifiers within atoms is given below in Section 8.

HiLog has a Herbrand model theory that specifies the semantics of the meta-program `maplist` (see Section 2) as a set of ground facts like

```
maplist(twice, [0,1,2], [0,2,4]).
```

However, HiLog's model theory is not satisfying for non-ground atoms like the following (see Section 2):

```
forall(enrolled(student(S), P),
                forall(syllabus(P, C), attends(S, C)
                )
).
```

Indeed, HiLog's model theory interprets such a non-ground atom as a set of ground atoms. Recall the need for generalized Herbrand models specified by non-ground and quantified expressions stressed at the end of Section 2. The article Chen *et al.* (1993) claims that "under 'reasonable' assumptions" a HiLog language can be given a "classical semantics." Clearly, this is only possible by equipping the language with a type theory that would keep apart terms from formulas. This would considerably restrict the language's meta-programming capability and, in fact, ruin HiLog's objectives. HiLog has a specific treatment of equality based on paramodulation (Robinson and Wos 1968; Robinson and Wos 1969; Robinson and Voronkov 2001). Finally, the article Chen *et al.* (1993) neither mentions that Russell's Paradox (see Section 6) is expressible in HiLog nor refers to Frege's logic.

Ambivalent Logic (Jiang 1994; Kalsbeek and Jiang 1995) is a second formalization of meta-programming in a non-classical logic. It is explicitly defined as a non-classical predicate logic that does not distinguish between terms and formulas. Thus, even though

the articles Jiang (1994) and Kalsbeek and Jiang (1995) do not mention type theory, they nonetheless specify a logic without type theory. The syntax of Ambivalent Logic is the aforementioned systematized syntax of Frege's logic (a few additional parentheses, necessary for disambiguation, are missing in its definition). Ambivalent Logic has a Herbrand-style model theory in which the meta-program `maplist` (see Section 2) is expressed as a set ground facts like:

```
maplist(twice, [0,1,2], [0,2,4]).
```

Ambivalent Logic can express meta-programs like the following (see Section 2) without resorting to a naming relation:

```
forall(R, F) :- not (R, not F).
r(X) :- X(X).
```

The treatment in Ambivalent Logic's model theory of existentially quantified formulas, in contrast to that of HiLog, is not based on skolemization. As a consequence, as stated in the article Kalsbeek and Jiang (1995, p. 55):

"[...] unlike in Hilog, both $\exists p \forall x(p(x) \leftrightarrow \neg q(x))$ and $\exists p \forall x(p(x) \leftrightarrow \exists u(q(u)(x))$ are valid in AL [Ambivalent Logic]."

In contrast to HiLog, Ambivalent Logic can express as follows that Ann believes to be loved by a logician:

```
believes(ann, ∃ x (logician(x) ∧ loves(x, ann))).
```

The treatment in Ambivalent Logic's model theory of non-ground atoms like:

```
forall(p(X), q(X))
forall(p(Y), q(Y))
```

or

```
believes(bill, ∀ X believes(ann, X))
believes(bill, ∀ Y believes(ann, Y))
```

is not satisfying. As pointed out in the articles Jiang (1994) and Kalsbeek and Jiang (1995), in a same Ambivalent Logic interpretation, the one variant expression of each example can be true and the other false:

"[...] It should be noted that 'similar' expressions like, for example, $\forall x.f(x)$ and $\forall y.f(y)$ constitute different, and unrelated, objects in the domains of models. That is, the truth values of the closed expressions $t(\forall x.f(x))$ and $t(\forall y.f(y))$ need not be the same." (Kalsbeek and Jiang 1995, p. 38)

In contrast to HiLog, Ambivalent Logic has no specific treatment of equality. Thus, an Ambivalent Logic theory can, like a classical predicate logic theory, have normal interpretations that interpret the equality predicate $=$ as the equality relation as well as other interpretations that interpret it as an equivalence relation which might be useful in logic programming or for knowledge representation. The article Kalsbeek and Jiang (1995, p. 55) discusses a property called "opaqueness" and states:

"[...] the schema $\forall x \forall y.(x = y \rightarrow \phi(x) \leftrightarrow \phi(y))$, and also $\forall x \forall y \forall z.(x = y \rightarrow x(z) \leftrightarrow y(z))$ [hold in HiLog]. In contrast, in the context of AL [Ambivalent Logic] we have a choice between validating the above schema or not, by either taking all of ET [the equality axioms] as the

equality theory, or restricting the equality to ET(I,II) [the equality axioms I and II]. Thus, unlike AL [Ambivalent Logic], HiLog is not appropriate for intensional logics, where opaqueness is usually desirable."

Finally, the articles Jiang (1994) and Kalsbeek and Jiang (1995) do not mention that Russell's Paradox (see below Section 6) is expressible in Ambivalent Logic and do not refer to Frege's logic.

Like Ambivalent Logic, Reflective Predicate Logic has a more expressive syntax than HiLog that allows quantifiers and connectives to appear within atoms. The syntaxes of Ambivalent Logic and Reflective Predicate Logic are systematizations of the syntax of Frege's logic that differ only in their representations of variables. The Herbrand-style model theory of Reflective Predicate Logic is similar to those of HiLog and Ambivalent Logic. It is more general than that of HiLog and corrects a serious deficiency of that of Ambivalent Logic by ensuring that in an interpretation, variant expressions are identically interpreted. Like the model theories of classical predicate logic and Ambivalent Logic, and unlike the model theory of HiLog, the model theory of Reflective Predicate Logic is not constrained to a specific treatment of equality.

HiLog, Ambivalent Logic, and Reflective Predicate Logic fulfil the first two requirements to formalizations of meta-programming mentioned in Section 2: They can express self-reflective predicates and they confound object and meta-variables. HiLog does not fulfil the third requirement of Section 2: It has no generalized Herbrand interpretations specified by non-ground and quantified expressions. Ambivalent Logic does fulfil this third requirement although in an unsatisfying manner. Reflective Predicate Logic corrects this deficiency of Ambivalent Logic.

*Reflection in computing, knowledge representation, and logic.* A language is reflective if statements can be expressed in this language that refer to themselves or other statements of the same language. Reflection is ubiquitous in computing: The program-as-data paradigm is a form of reflection, the von Neumann architecture is reflective, some programming languages are reflective, reflection is used for proving undecidability results, reflection is often needed in knowledge representation because introspective capabilities are often required from intelligent software and robots, etc.

Formalizations of meta-programming in first-order logic are closely related to reflection in knowledge representation (Weyhrauch 1980; Perlis 1985; Aiello *et al.* 1986; Perlis 1988a; Perlis 1988b; van Harmelen 1989; Benjamin 1990).

*Logics of knowledge and belief.* The examples referring to the beliefs of agents given in this article could be expressed in a logic of knowledge and belief (Hintikka 1964; Halpern and Moses 1985; Lismont and Mongin 1994; van Ditmarsch *et al.* 2015).

## 4 Predicativity and impredicativity

Predicativity (Feferman 2005) is an essential trait of classical logic atoms. Meta-programming makes use of impredicative atoms. This section recalls one of the reasons why impredicative atoms have been banned from classical predicate logic.

Consider a property $P$ on the nodes of an undirected graph $G$ defined as follows: A node $n$ of $G$ has property $P$ if all its immediate neighbours have property $P$. This definition is

not acceptable because it is ambiguous: It applies among others to the property holding of no nodes and to the property holding of all nodes.

At the beginning of the 20th century, Henri Poincarré and Bertrand Russell have proposed the Vicious Circle Principle (Russell 1907; Russell 1986) that forbids circular definitions, that is, definitions referring to the very concept they define like the above definition of property $P$. Russell called "predicative" definitions that adhere to the Vicious Circle Principle and "impredicative" definitions that violate it. Thus, the Vicious Circle Principle is the adhesion to predicativity and the rejection of impredicativity.

The Vicious Circle Principle, however, has a drawback: It forbids hereditary and, more generally, inductive definitions (Aczel 1977) like the definition of the formulas of a logic, or of the programs of a programming language, or of the fixpoint of the immediate consequence operator of a definite logic program (van Emden and Kowalski 1976; Apt and van Emden 1982). (Recall that the definition of a property $P$ is hereditary if it states that whenever a natural number $n$ has property $P$, so does $n+1$. Recall that a definite logic program is a program the clauses of which contain no negative literals.) The Vicious Circle Principle also rejects the definition sketched at the beginning of this section even though this definition makes sense as an inductive definition:

- Base cases: A (possibly empty) set of nodes of $G$ is specified that have the property $P$.
- Induction case: If a node has the property $P$, then all its immediate neighbours have the property $P$.

An inductively defined property (or set) is the smallest property (or set) that fulfils the base and induction cases of its definition (Aczel 1977). Thus, understood as an inductive definition, the definition sketched at the beginning of this section is that of the empty relation, that is, of the relation that holds of no nodes.

Since the semantics of recursive functions and predicates is defined in terms of inductive definitions, the Vicious Circle Principle also implies the rejection of recursive functions and predicates. Clearly, such a rejection is not compatible with programming. The Vicious Circle Principle further rejects definitions like the following that are widely accepted even though they are not inductive and they do not provide constructions of the entities they define:

- $y$ is the smallest element of an ordered set $S$ if and only if for all elements $x$ of $S$, $y$ is less than or equal to $x$, and $y$ is in $S$.
- The definition of the stable models of logic programs (Gelfond and Lifschitz 1988).

Such definitions have in common that they quantify over domains the definitions of which refer to the entities being defined. Examples like the aforementioned led some mathematicians, most notably Gödel, to object that impredicative definitions are acceptable provided the entities they refer to are clearly apprehensible (Gödel 1944). Nowadays, most logicians and mathematicians follow Gödel and accept impredicative definitions of the following kinds (Aczel 1977; Feferman 2005):

- Inductive definitions.
- Impredicative definitions that characterize elements (like the smallest number in a set) of clearly apprehensible sets (including inductively defined sets).

The Vicious Circle Principle was the reason for Russell and Frege, whom Russell persuaded, to reject impredicative atoms built up from predicates that "apply to themselves", like a predicate expressing a set of all sets. Such predicates are expressible in Frege's logic (Frege 1879; Frege 1893; Frege 1903), the precursor of first-order logic. A predicate that "applies to itself" is the core of Russell's Paradox (Link 2004) (discussed below in Section 6). The Vicious Circle Principle and paradoxes, among other the paradox bearing his name, motivated Russell to develop the Ramified Theory of Types (Russell 1908) that precludes predicates "applying to themselves".

Because classical logic adheres to a type theory, classical logic rejects reflective expressions like the following that are at the core of meta-programming:

```
believes(ann, itRains)
believes(ann, believes(bill, itRains))
```

where `itRains` might be true or false, that is, amounts to a formula, not a term.

In order to simplify the following argument, let us consider a unary predicate "belief" derived from the above definitions by disregarding who is holding a belief:

```
belief(itRains)
belief(belief(itRains))
```

On the one hand, the unary predicate `belief` cannot be interpreted by a set $B$ because $B$ would have as elements, the subset of all beliefs like `itRains` that are believed to be believed. On the other hand, a set of closed atoms like the above two expressions perfectly gives a semantics to the unary predicate `belief`.

Relying in such a manner on a standard set of closed atoms for defining non-standard "sets", say "collections", like the collection of beliefs in the above example, is the essence of the model theory proposed below. The resulting impredicative definitions (in the example given above, the definition of beliefs) fulfil Gödel's condition to be interpreted in reference to clearly apprehensible entities (in the example given above, a standard set of closed atoms).

## 5 Syntax of Reflective Predicate Logic

This section introduces "expressions" that amount to both the terms and the formulas of classical predicate logic languages. Except for the use of the paradigm "quantification makes variables" and a more careful parenthesizing ensuring that expressions are non-ambiguous, the syntax given below is that of Ambivalent Logic (Jiang 1994; Kalsbeek and Jiang 1995).

*Definition 1* (*Symbols and Expressions*)
A Reflective Predicate Logic language $\mathscr{L}$ is defined by

- the logical symbols consisting of
  - — the connectives $\wedge$, $\vee$, $\Rightarrow$, and $\neg$,
  - — the quantifiers $\forall$ and $\exists$,
  - — the parentheses ) and ( and the comma , .
- at least one and at most finitely many non-logical symbols each of which is distinct from every logical symbol.

The expressions of a Reflective Predicate Logic language $\mathscr{L}$ and their outermost constructors are inductively defined as follows:

- A non-logical symbol $s$ is an expression the outermost constructor of which is $s$ itself.
- If $E$ and $E_1, \ldots, E_n$ $(n \geq 1)$ are expressions, then $E(E_1, \ldots, E_n)$ is an expression the outermost constructor of which is $E$.
- If $E$ is an expression, then $(\neg E)$ is an expression the outermost constructor of which is $\neg$.
- If $E_1$ and $E_2$ are expressions, then $(E_1 \wedge E_2)$, $(E_1 \vee E_2)$, $(E_1 \Rightarrow E_2)$ are expressions the outermost constructors of which are $\wedge$, $\vee$, and $\Rightarrow$, respectively.
- If $E_1$ and $E_2$ are expressions, then $(\forall E_1\ E_2)$ and $(\exists E_1\ E_2)$ are expressions the outermost constructors of which are $\forall$ and $\exists$ respectively.

A logical or non-atomic expression is an expression the outermost constructor of which is a connective or a quantifier. A non-logical or atomic expression, or atom, is an expression the outermost constructor of which is neither a connective nor a quantifier.

The set of expressions of a Reflective Predicate Logic language is not empty since, by definition, the language has at least one non-logical symbol.

More parentheses are required by Definition 1 than in classical predicate logic and than stated in the article Kalsbeek and Jiang (1995). This is necessary for distinguishing (well-formed) expressions such as $(\neg a)(b)$ and $(\neg a(b))$ or $(\forall x\ p(x))(a)$ and $(\forall x\ p(x)(a))$. Provided a few additional parentheses are added, first-order logic formulas are expressions in the sense of Definition 1, that is, the syntax given above is a conservative extension of the syntax of first-order logic. This issue is addressed in more detail below in Section 10.

The expressions of a Reflective Logic language can be proven non-ambiguous similarly as classical logic formulas are proven non-ambiguous. The sub-expressions and proper sub-expressions of an expression of a Reflective Predicate Logic language can be similarly defined as the sub-formulas and proper sub-formulas of a classical logic formula. Thus, an expression is a sub-expression but not a proper sub-expression of itself.

The definition of the scope of a quantified variable in an expression generalizes that of classical predicate logic: The scope of $E_1$ in $(\forall E_1\ F)$ and in $(\exists E_1\ F)$ is $F$ except those subexpressions of $F$ of the form $(\forall E_2\ G)$ or $(\exists E_2\ G)$ such that $E_2$ is a subexpression of $E_1$. Thus, in each of the following expressions, the inner quantified expression is not within the scope of the outer quantified expression:

```
( ∀ x     ( ∀ x     p(f(x), x)   ) )
( ∀ x     ( ∀ x     p(x(f), x)   ) )
( ∀ f(a)  ( ∀ f     p(f(a), f)   ) )
```

In each of the following expressions that can be obtained from each other by a variable renaming, the inner quantified expression is within the scope of the outer quantified expression.

```
( ∀ f  ( ∀ x     p(x,     f) ) )
( ∀ f  ( ∀ f(a)  p(f(a), f) ) )
```

Recall that a logical, or non-atomic, expression is an expression the outermost constructor of which is a negation, a connective, or a quantifier. Thus,

```
(believes(ann, itRains) ∧ believes(ann, itIsWet))
(∃ X believes(ann, X))
(¬(∃ Y (believes(bill, Y))))
```

are logical expressions. Logical expressions correspond to first-order logic compound (that is, non-atomic) or quantified formulas.

Recall that an atomic expression, or atom, is an expression the outermost constructor of which is neither a connective nor a quantifier. Thus,

```
believes(ann, (itRains ∧ itIsWet))
believes(ann, (∀ X believes(bill, X)))
(believes ∧ trusts)(ann, bill)
(∀ T (trust(T) ⇒ T))(ann, bill)
```

are atoms while

```
(believes(ann, itRains) ∧ believes(ann, itIsWet))
(∀ X believes(bill, X))
```

are non-atomic, or logical, expressions. Note that if $A_1$ and $A_2$ are atoms, then the (well-formed) expressions $(\forall A_1 A_2)$ and $(\exists A_1 A_2)$ are not atoms. Note also that an expression is either atomic (or non-logical) or non-atomic (or logical).

Skolemization can be specified as usual by adding additional non-logical symbols to the language.

For the sake of simplicity, the above definition assumes that every non-logical symbol has all arities. This reflects a widespread logic programming practice: Using `p/2`, that is, `p` with arity 2, in a Prolog program, for example, does not preclude using `p/3`, that is, `p` with arity 3, in the same program. Assuming that non-logical symbols of a Reflective Predicate Logic language have all arities is a convenience, not a necessity. The above definition can be refined to a less permissive definition as of non-logical symbols' arities.

In contrast to the syntax of Ambivalent Logic given in the articles Jiang (1994) and Kalsbeek and Jiang (1995), the above definition does not distinguish between variables and constants. According to the above definition, quantifications make variables:

- `likes(ann, bill)` contains no variables. In this expression, `ann` and `bill` serve as constants.
- `(∃ ann likes(ann, bill))` means that there is someone who likes Bill. In this expression, `ann` serves as a variable and `bill` as a constant.
- `(∀ bill (∃ ann likes(ann, bill)))` means that everyone is liked by someone. In this expression, `ann` and `bill` serve as a variables.

A first advantage of the paradigm "quantifications make variables" is that every expression is closed. Indeed, a symbol which is not quantified such as `x` in `likes(x, bill)` is not a variable. This is not a restriction, since in logics with open formulas, open formulas serve only as components of closed formulas. The paradigm "quantifications make variables" corresponds to the declarations of programming languages. The paradigm "quantification makes variables" is akin to lambda-abstraction. We give it an expressive denomination for avoiding referring to the lambda calculus our proposal does not build upon.

Explicit quantifications as introduced in Definition 1 are not usual in logic programming. Combined with the paradigm "quantifications makes variables", they are useful for meta-programming because they make it easy to transform expressions. The expression `likes(ann, bill)` for example can be abstracted into

```
(∃ likes likes(ann, bill))
```

(meaning that Ann and Bill are in some relationship) and generalized as

```
(∀ likes likes(ann, bill))
```

(meaning that Ann and Bill are in all possible relationships). Similarly, the expression

```
(((∃ x p(x)) ∧ (¬ (∃ x p(x)))) ⇒ (∀ G G))
```

(meaning that every expression follows from `(∃ x p(x))` and its negation) can easily be generalized into

```
(∀ (∃x p(x)) (((∃x p(x)) ∧ (¬(∃x p(x)))) ⇒ (∀G G)))
```

that is, after renaming `F`, the expression `(∃ x p(x))` serving as variable,

```
(∀ F ((F ∧ (¬ F)) ⇒ (∀ G G)))
```

(meaning that every expression follows from an expression and its negation).

## 6 The Barber and Russell's Paradoxes in Reflective Predicate Logic

One of the reasons for the Vicious Circle Principle, that is, the rejection of impredicative definitions, was Russell's Paradox, a second-order variation of the first-order Barber Paradox. Both the Barber and Russell's Paradoxes can be expressed in Reflective Predicate Logic. This section explains why this is not a problem.

Since Reflective Predicate Logic's syntax is a conservative extension of the syntax of first-order logic, a formulation of the Barber Paradox in first-order logic like the following is also a formulation of that paradox in Reflective Predicate Logic:

```
man(barber)
(∀y (man(y) ⇒ (shaves(barber,y) ⇔ (¬shaves(y,y)))))
```

where, extending Definition 1, $(E_1 \Leftrightarrow E_2)$ is defined as a shorthand notation for $((E_1 \wedge E_2) \vee ((\neg E_1) \wedge (\neg E_2)))$. (This extension is a common manner to define the semantics of $\Leftrightarrow$ in classical logic.) The above expressions convey that the barber is a man shaving all men who do not shave themselves. The Barber Paradox is a mere inconsistency: The barber cannot exist because he would have both to shave himself and not to shave himself. The self-contradictory formula

```
(shaves(barber,barber) ⇔ (¬shaves(barber,barber)))
```

follows in first-order logic from the above specification of the Barber Paradox. A formula expressing the Barber Paradox is inconsistent with respect to the model theory defined in the section after next as it is in first-order logic.

The syntax of Section 5 that does not distinguish between formulas and terms gives rise to (well-formed) expressions that are not expressible in first-order logic, and that,

like the Barber Paradox, are inconsistent. One such expression is the following that expresses Russell's Paradox in both Frege's logic (Frege 1879; Frege 1893; Frege 1903) and in Reflective Predicate Logic (a brief introduction into Frege's logic is given in the Appendix):

```
(⋆) (∀x (e(x) ⇔ (¬x(x))))
```

Instantiating x with e in (⋆) yields the self-contradictory expression

```
(e(e) ⇔ (¬ e(e)))
```

Expression (⋆) is inconsistent for the model theory given in the next section as it must be in every well-specified model theory because it is self-contradictory. Thus, expression (⋆) is, like the above specification of the Barber Paradox a mere inconsistency.

While it is paradoxical to think of concepts that cannot exist, inconsistent expressions are no reasons to reject the language in which they are expressed. After all, nobody considers the language of propositional logic as paradoxical, notwithstanding the fact that it can express the formula `(p ∧ (¬ p))` which is inconsistent for requiring a proposition `p` to be both true and false.

The rejection of logics in which Russell's Paradox can be expressed stems from the conception that every expression must define a set. While it is understandable that Russell, Frege, and their contemporaries shared this conception, this conception can be given up. Giving up this conception provides for a simple logic perfectly formalizing Prolog-style meta-programming. Giving up this conception allows for impredicative atoms that are interpreted as collections like the collection of beliefs mentioned above in Section 4.

## 7 Variant expressions and expression rectification

In the next section, a model theory is given for Reflective Predicate Logic such that two syntactically distinct atomic expressions that are variants of each other like

```
believes(ann, (∀ X believes(bill, X)))
believes(ann, (∀ Y believes(bill, Y)))
```

(with the intended meaning that Ann believes that Bill believes everything) are identically interpreted. As already mentioned, this is not the case with the model theory of Ambivalent Logic (Jiang 1994; Kalsbeek and Jiang 1995).

Two first-order logic atoms or terms $E_1$ and $E_2$ are variants of each other if there is a one-to-one mapping $\sigma$ of the variables occurring in $E_1$ into the variables occurring in $E_2$ such that applying $\sigma$ to $E_1$ yields $E_2$, noted $E_1\sigma = E_2$. Thus, the first-order formulas

```
p(X, Y)
p(Y, Z)
```

(in which `X`, `Y` and `Z` are first-order variables) are variants of each other but the first-order formulas

```
p(X, Y)
p(Z, Z)
```

(in which `X`, `Y` and `Z` are first-order variables) are not.

Variance is more complex to formalize for first-order logic formulas and Reflective Predicate Logic expressions because of the overriding (or variable shadowing, or shadowing, for short) that might take place with quantification. While the first-order formulas

```
(p(X) ∧ q(Y))
(p(X) ∧ q(X))
```

(in which `X` and `Y` are first-order variables) are not variants of each other, the first-order formulas

```
(∀ X (p(X) ∧ ∃ Y q(Y)))
(∀ X (p(X) ∧ ∃ X q(X)))
```

are variants of each other because, in the second formula, the second quantification of the variable `X` overrides the first.

Variant expressions can easily be defined by relying on a rectification. Rectifying an expression consists in renaming its variables from a predefined pool of "fresh" variables, that is, variables not occurring in the expressions considered, in such a manner that the variables of two distinct quantifications are distinct. Thus, if the "fresh" variables considered are $v_1, v_2, \ldots$, then rectifying the expression

```
(∀ X (p(X) ∧ ∃ X q(X)))
```

might result in

```
(∀ v1 (p(v1) ∧ ∃ v2 q(v2)))
```

and rectifying the expression

```
((∀ X p(X)) ∧ (∃ X q(X)))
```

might result in

```
((∀ v3 p(v3)) ∧ (∃ v1 q(v1)))
```

It is assumed in the following that there is a denumerable supply $v_1, v_2, \ldots, v_i, \ldots$ of variables such that each $v_i$ is distinct from every logical and every non-logical symbol of the Reflective Predicate Logic language considered. (An infinite supply of variables is necessary to ensure that proofs are not bounded in length. This infinity does not threaten computability because the variables needed in a proof can be created on demand while computing proofs.)

A rectification is performed by a variable renaming which is conveniently specified as a predicate *rect* recursively defined on an expression's structure. It implements a left-to-right outside-in traversal of expressions of all kinds except quantified expressions that are traversed inside-out so as to reflect the quantified variables' scopes: Note, in the last case of the algorithm, the recursive call $rect(E_2, i, R_2, j)$ instead of $rect(E_1, i, R_1, j)$. Logic programming pseudo-code is used in the following definition because it expresses the sideway passing of variable indices between recursive calls in a more readable manner than functional pseudo-code.

*Definition 2* (*Rectification*)
A rectified $R$ of an expression $E$ is specified by the predicate $rect(E, i, R, j)$ where:

- $i \geq 1$, the "initial variable index", denotes the first variable $v_i$ that might be used in rectifying $E$

- $j = i$ if $E$ and its rectified contain no variables
- $j > i$, the "final variable index", is $k + 1$ if $k$ is the highest index of a variable occurring in the rectified of $E$

The predicate *rect* is recursively defined on an expression's structure:

- if $E$ is a symbol:
  $rect(E, i, E, i)$
- if $E = E_1(E_2, \ldots, E_n)$ with $n \geq 2$:
  $rect(E, i, R_1(R_2, \ldots, R_n), i_n)$ where the $R_k$ and $i_n$ are defined by
  $rect(E_1, i, R_1, i_1)$, $rect(E_2, i_1, R_2, i_2)$, ..., and $rect(E_n, i_{n-1}, R_n, i_n)$
- if $E = (\neg E_1)$:
  $rect(E, i, (\neg R_1), j)$ is defined by $rect(E_1, i, R_1, j)$
- if $E = (E_1 \theta E_2)$ with $\theta \in \{\wedge, \vee, \Rightarrow\}$:
  $rect(E, i, (R_1 \theta R_2), i_2)$ where $R_1$, $R_2$, and $i_2$ are defined by
  $rect(E_1, i, R_1, i_1)$ and $rect(E_2, i_1, R_2, i_2)$
- if $E = (\theta\ E_1\ E_2)$ with $\theta \in \{\forall, \exists\}$:
  $rect(E, i, (\theta\ v_j\ R), k)$ where $R$, $j$, and $k$ are defined by
  $rect(E_2, i, R_2, j)$, $k = j+1$, and $R$ are obtained from $R_2$ by simultaneously replacing all occurrences of $E_1$ in $R_2$ by $v_j$.

A rectified of a finite set $\{E_1, \ldots, E_n\}$ of expressions is the set $\{R_1, \ldots, R_n\}$ where $(R_1 \wedge (\ldots \wedge R_n) \ldots)$ is the rectified of the expression$(E_1 \wedge (\ldots \wedge E_n) \ldots)$.

An expression (set of expressions, respectively) is said to be rectified if it is a rectified of some expression (set of expressions, respectively).

Because of the inside–out traversal of quantified expressions, the predicate *rect* is not tail recursive. An efficient, tail recursive, implementation of *rect* would require an accumulator for storing embedding quantified expressions the variable renaming of which is delayed.

The algorithm specified in Definition 2 terminates because every recursive call refers to a strict sub-expression. $rect(E, i, R, j)$ is functional in the sense that there is exactly one pair $(R, j)$ for each pair $(E, i)$ because the cases of the above definition are mutually exclusive. The call pattern of *rect* is $rect(+E, +i, ?R, ?n)$ meaning that in a call to *rect* $E$ and $i$ must be specified and that each of $R$ and $n$ can, but do not have to, be specified. Therefore, it is also possible to define rectification functionally with a binary mapping $rect : expr \times \mathbb{N} \to expr \times \mathbb{N}$ where *expr* is the set of all expressions of a Reflective Predicate Logic language. No variables are overridden in the rectified of an expression because of the sideway passing of variable indices between recursive calls: Each recursive call to *rect* uses as initial variable index the final variable index of the previous recursive call to *rect*. Let $R_i$ denote the rectified of an expression $E$ specified by $rect(E, i, R_i, n)$. For all $k \geq 1$, $R_{i+k}$ can be obtained from $R_i$ by replacing in $R_i$ every variable $v_j$ by $v_{j+k}$.

In general, a finite set of expressions has more than one rectified, each resulting from an ordering of the set's expressions. Note that the expressions in a rectified set of expressions are standardized-apart, that is, two distinct expressions have no variables in common.

In contrast to first-order formulas' rectification, the algorithm of Definition 2 considers the variables that might occur in the constructors of Reflective Predicate Logic atoms. The Reflective Predicate Logic expression

```
((∀ X p(X)) ∧ (∃ X q(X)))
```

is for example rectified by the algorithm of Definition 2 into:

```
((∀ v2 p(v2)) ∧ (∃ v1 q(v1)))
```

while the Reflective Predicate Logic atom

```
(∀ T (trust(T) ⇒ T))(ann, bill)
```

is rectified by the same algorithm into:

```
(∀ v1 (trust(v1) ⇒ v1))(ann, bill)
```

*Definition 3* (*Variant Expressions*)
Two expressions $E_1$ and $E_2$ are variants of each other, noted $E_1 \sim E_2$, if their rectified after Definition 2 and computed with the same initial variable index are identical.

The relation $\sim$ on the expressions of a Reflective Predicate Logic language is an equivalence relation.

## 8 A Herbrand-style model theory for Reflective Predicate Logic

Atoms in Reflective Predicate Logic (that is, expressions the outermost constructors of which are neither connectives nor quantifiers) like

```
likes(ann, likes(bill, ann))
likes(ann, (∀ x likes(bill, x)))
(likes ∧ trusts)(ann, bill)
(∀ T trust(T) ⇒ T)(ann, bill)
```

(meaning that Ann likes that Bill likes her, that Ann likes that Bill likes everyone and everything, that Ann likes and trusts Bill, and that Ann trusts Bill in all specified forms of trust) differ from atoms in a first-order logic language in three respects:

1. First-order atoms may be open or closed, whereas all Reflective Predicate Logic expressions, including all atoms, are closed expressions thanks to the paradigm "quantification makes variables".
2. First-order logic atoms cannot have anything but terms as arguments, whereas atoms in Reflective Predicate Logic may have as arguments non-atomic (or logical) expressions such as `(∀ x likes(bill, x))` that amount to first-order formulas, not terms.
3. In first-order logic the outermost constructor of an atom is a symbol such as `likes`, whereas in Reflective Predicate Logic it can be any expression such as `likes` and `(∀ T trust(T) ⇒ T)`.

How the model theory should treat atoms that contain non-atomic, or logical, expressions can be seen on the following example the meaning of which is that calling someone "A and B" implies calling him "A" and calling him "B", that claiming not to call someone "A" is in fact calling him "A" and that calling someone "fat" is offending.

```
(∀ x (∀ y (∀ z
   (says(x, (y ∧ z)) ⇒ (says(x, y) ∧ says(x, z))))))
(∀ x (∀ y (says(x, (¬ says(x, y))) ⇒ says(x, y))))
(∀ x (∀ y (says(x, is(y, fat)) ⇒ offends(x, y))))
```

An interpretation satisfying the three above expressions as well as the additional atom

```
says(donald, (¬says(donald, is(kim, (short∧fat))))
```

should also satisfy

```
offends(donald, kim)
```

regardless of whether none, only one, or both of

```
is(kim, (short ∧ fat))
is(kim, fat)
```

are satisfied in that interpretation (Trump 2017). This requirement is essential among others for static program analyses (like static type checking) to be expressible as meta-programs. Indeed, a static program analysis is independent of the analysed programs' run-time behaviors, that is, a static analysis of a logic program is independent of which program parts evaluate to true. The model theory specified below is tuned to ensure this requirement.

In contrast to first-order logic ground atoms, Reflective Predicate Logic atoms can have variants, like

```
believes(ann, (∀ y     (believes(ann, y    )
                            ⇒ believes(bill, y    ))))
believes(ann, (∀ t(a)  (believes(ann, t(a))
                            ⇒ believes(bill, t(a)))))
```

that should be given the same meaning even though they syntactically differ from each other. Thus, the Herbrand base of a Reflective Predicate Logic language must be defined as the set of equivalence classes of the language's atoms with respect to the variant relation $\sim$.

Since atoms like

```
believes(ann, (∀x (believes(ann,x)⇒believes(bill,x))))
```

(with the intended meaning that Ann believes that Bill believes all that she herself believes) of a Reflective Predicate Logic language correspond to both ground atoms and ground terms of first-order logic languages, the set of all expressions of a Reflective Predicate Logic language corresponds to both the Herbrand universe (van Emden and Kowalski 1976; Chang and Lee 1997) and the Herbrand base (van Emden and Kowalski 1976; Chang and Lee 1997) of a first-order logic language.

*Definition 4* (*Herbrand Universe*)
Let $\mathscr{A}$ be the set of atoms of a Reflective Predicate Logic language $\mathscr{L}$ and $\sim$ the variant relation of $\mathscr{L}$. The Herbrand universe of $\mathscr{L}$ is $\mathscr{A}/\sim$, that is, the set of equivalence classes of $\sim$.

As the following example illustrates, standardization-apart is needed in proving so as to properly reflect variable scopes. In this example, upper case characters denote variables. From the clauses

$[\neg p(X), \neg q(Y), r(X, Y)]$
$[p(Z)]$
$[q(Z)]$

the clause

$$[r(Z_1, Z_2)]$$

can be derived by resolution. If, however, no standardization-apart of the set of clauses was performed during resolution, then the more specific clause "$[r(Z, Z)]$" (or a variant of that clause) would be wrongly derived instead of "$[r(Z_1, Z_2)]$" (or a variant of that clause).

Under the paradigm "quantification makes variables", something similar might happen while instantiating variables. Consider once again the Reflective Predicate Logic expression

(†) (∀ `bill` (∃ `ann` `likes(ann, bill)`))

(meaning that everyone is liked by someone) in which the symbols `ann` and `bill` serve as variables. Prematurely instantiating `bill` with `ann` yields

(∃ `ann` `likes(ann, ann)`)

(meaning that someone likes herself) which is not a logical consequence of (†). Such incorrect instantiations are avoided by rectifying the expressions under consideration using the infinite supply of variables $v_1, v_2, \ldots$ that has been assumed in the previous Section 7. Indeed, rectified expressions do not contain non-variable symbols serving as variables. Since each variable $v_i$ is distinct from every non-logical symbol as well as from every logical symbol of the Reflective Predicate Logic language considered, incorrect instantiations like in the former example are impossible.

*Definition 5* (*Notations*)
If $A$ is an atom of a Reflective Predicate Logic language $\mathscr{L}$, then class($A$) denotes the variant class of $A$, that is, the equivalence class of $A$ in the Herbrand universe $\mathscr{A}/\sim$ of $\mathscr{L}$.

If $R$ is a rectified expression, if $v_i$ is a variable, and if $A$ is an atom, then $R[A/v_i]$ denotes the expression obtained from standardized-apart rectified variants $R^v$ and $A^v$ of $R$ and $A$, respectively by simultaneously replacing in $R^v$ all occurrences of $v_i$ by $A^v$.

In defining the notation $R[A/v_i]$, there is no need for caring about overridden variables because that notation will only apply to rectified expressions $R$ and because, as observed in the previous Section 7, in rectified expressions no variables are overridden. Since $R^v$ and $A^v$ are rectified and have no variables in common (they are standardized-apart), $R[A/v_i]$ is rectified.

*Definition 6* (*Interpretations and Models*)
A Herbrand interpretation $I(S)$ of a Reflective Predicate Logic language $\mathscr{L}$ is specified as a subset $S$ of the universe $\mathscr{A}/\sim$ of $\mathscr{L}$.

An expression $E$ is satisfied in a Herbrand interpretation $I(S)$ of $\mathscr{L}$, denoted $I(S) \models E$, if a rectified $R$ of $E$ is satisfied in $I(S)$, denoted $I(S) \models R$, in the following sense, where:

- $R, R_1$, and $R_2$ denote *rectified* expressions.
- $A$ denotes a *rectified* atom.

$$\begin{array}{lll} I(S) \models A & \text{iff} & \text{class}(A) \in S \\ I(S) \models (\neg R) & \text{iff} & I(S) \not\models R \end{array}$$

$$
\begin{array}{lll}
I(S) \models (R_1 \wedge R_2) & \text{iff} & I(S) \models R_1 \text{ and } I(S) \models R_2 \\
I(S) \models (R_1 \vee R_2) & \text{iff} & I(S) \models R_1 \text{ or } I(S) \models R_2 \\
I(S) \models (R_1 \Rightarrow R_2) & \text{iff} & \text{if } I(S) \models R_1, \text{ then } I(S) \models R_2 \\
I(S) \models (\exists v_i\ R) & \text{iff} & I(S) \models R[A/v_i] \text{ for some } A \\
I(S) \models (\forall v_i\ R) & \text{iff} & I(S) \models R[A/v_i] \text{ for all } A
\end{array}
$$

A set $T$ of expressions is satisfied in $I(S)$, denoted $I(S) \models T$, if every expression in $T$ is satisfied in $I(S)$.

An interpretation is called a model of an expression $E$ (a set of expressions $P$, respectively) if it satisfies $E$ (every expression in $P$, respectively).

Satisfaction of logical expressions (i.e., expressions the outermost symbols of which are logical symbols ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$)) is defined like in first-order logic. Satisfaction of atoms (i.e., expressions the outermost expressions of which are non-logical symbols) is not defined like in first-order logic: It is based on variance instead of syntactical identity. However, if an atom does not contain variables, then it is the single element of its variant class and, as a consequence, its satisfaction is defined like in first-order logic.

Applied to first-order logic expressions, Definitions 4 and 6 amount to the definitions of Herbrand universes, interpretations and models of first-order logic. Indeed, a ground atom $A$ of a first-order logic language is the only element of its equivalence class for the variant relation. Thus, Definition 6 is a conservative extension of first-order logic's notions of Herbrand interpretations and models. This observation is formally developed in the next section.

An interpretation as defined above can be seen as a set $S$ of atoms. An atom is satisfied in the interpretation specified by $S$ if and only if it is a variant of an element of $S$.

Consider the following set $P_1$ of expressions, a simple meta-program (with explicit quantifications) on the beliefs of Ann and Bill:

```
believes(ann, itRains)
believes(ann, (itRains ⇒ itIsWet))
believes(ann,(∀x (believes(ann,x) ⇒ believes(bill,x))))
(∀ x (believes(ann, x) ⇒ believes(bill, x)))
```

The following set $S_1$ of atoms specifies a model of $P_1$ (consisting of the atoms' equivalence classes for $\sim$):

```
believes(ann, itRains)
believes(ann, (itRains ⇒ itIsWet))
believes(ann,(∀y (believes(ann,y) ⇒ believes(bill,y))))
believes(bill,itRains)
believes(bill,(itRains ⇒ itIsWet))
believes(bill,(∀z (believes(ann,z) ⇒ believes(bill,z))))
```

Consider the following set $P_2$ of expressions in which `(∀ T (trust(T) ⇒ T))` is an atom constructor:

```
trust(t1)
trust(t2)
t1(ann, bill)
t2(ann, bill)
```

```
(∀ X (∀ Y
     ( (∀ T (trust(T) ⇒ T(X, Y))) ⇒
       (∀ T (trust(T) ⇒ T))(X, Y)
     )
   )
)
```

The following set $S_2$ of atoms, in which `(∀ X (trust(X) ⇒ X))` is an atom constructor, specifies a model of $P_2$:

```
trust(t1)
trust(t2)
t1(ann, bill)
t2(ann, bill)
(∀ X (trust(X) ⇒ X))(ann, bill)
```

A proof theory convenient for expressions like the last of $P_2$ is out of the scope of this article.

The definition of an interpretation given in the articles Jiang (1994) and Kalsbeek and Jiang (1995) is more stringent than Definition 6: Instead of relying on the variant relationship, it requires syntactical identity. As a consequence, the set $S_1$ of atoms given above does not specify a model in the sense of Jiang (1994) and Kalsbeek and Jiang (1995) of the set $P_1$ of expressions given above. This is undesirable because meta-programming requires to interpret identically expressions like the following that are variants of each other:

```
believes(ann,(∀x (believes(ann,x) ⇒ believes(bill,x))))
believes(ann,(∀y (believes(ann,y) ⇒ believes(bill,y))))
```

Even though a Reflective Predicate Logic language gives rise to inconsistent expressions (like the definition ($\star$) of Russell's paradoxical set given in Section 6), the model theory given above is adequate for the reasons mentioned at the end of Section 4. Indeed, it refers to an inductively defined universe, the (standard) set of all expressions, and impredicative atoms like

```
believes(ann,(∀x (believes(ann,x)⇒believes(bill,x))))
belief(belief(itRains))
```

perfectly characterize elements of that universe even though they cannot be interpreted as standard sets.

## 9 Symbol overloading in classical predicate logic languages

The clause

```
student(X) :- enrolled(student(X), _)
```

can be seen as a first-order logic clause even though under this view the symbol `student` occurs in that clause both as a predicate symbol (left) and as a function symbol (right). Indeed, even though first-order logic languages are conventionally defined such that a same (predicate or function) symbol cannot be used with different arities and a same symbol cannot be used both as a function and as a predicate symbol, these conventions

are not necessary for two reasons: First, arities can be seen as part of a symbol which allows to disambiguate occurrences of a same symbol with different arities; second, the syntactic context in a formula allows to disambiguate occurrences of a same symbol denoting a function symbol and a predicate symbol.

The following example illustrates this observation. Consider

- a function symbol `e` of artity 1 meant to express a function "twice", that is, `e(1)` stands for $2 \times 1$.
- a function symbol `e` of artity 2 meant to express the addition of integers, that is, `e(1,1)` stands for $1 + 1$.
- a predicate symbol `e` of arity 1 meant to express "even", that is, `e(1)` stands for `isEven(1)`.
- a predicate symbol `e` of arity 2 meant to express equality, that is, `e(1,1)` stands for $1 = 1$.

If `e(1)` is a first-order term, then it can only stand for $2 \times 1$. If `e(1,1)` is a first-order term, then it can only stand for $1 + 1$. If `e(e(1,1),e(1))` is a first-order formula, then the outer occurrence of `e` must be a predicate symbol, the inner occurrences of `e` must be function symbols and the formula can only stand for $1 + 1 = 2 \times 1$. Similarly, if `e(e(e(1,1)))` is a first-order formula, then it can only stand for `isEven(2 × (1 + 1))`.

The above observation is common knowledge in logic and in programming. In logic, Church's re-formulation of the Simple Type Theory (Chwistek 1921; Ramsey 1926), the Simply Typed Lambda Calculus (Church 1940; Barendregt 1993), exploits it. The above observation has been made for classical predicate logic in the article Chen *et al.* (1993). In that article, a predicate logic language overloading symbols in the aforementioned sense is called "contextual." In programming, the use of a same symbol with different meanings in different contexts is called overloading. In functional and Prolog programs, a same symbol is commonly used with different arities. In functional programs type selectors are commonly named like the corresponding type constructors and in Haskell compound types are named using their own type constructor (`[Int]` for example denotes the type of the lists of integers).

Thus, interpreting in first-order logic a clause like

```
student(X) :- enrolled(student(X), _)
```

in which some symbols are interpreted both as function and predicate symbols requires no further formalization and the aforementioned symbol overloading in first-order languages accounts for the aforementioned Prolog facts:

```
call(twice, 1, 2)

forall(enrolled(student(S), mathematics),
               attends(S, logic)
)

forall(enrolled(student(S), P),
               forall(syllabus(P, C), attends(S, C)
               )
)
```

However, the aforementioned symbol overloading in first-order languages accounts neither for the representation in classical predicate logic of `call(P, X, Y)`, that is, `P(X, Y)`, nor for the following Prolog meta-program that defines the `forall` predicate:

```
forall(R, F) :- not (R, not F)
```

Indeed, the aforementioned symbol overloading in first-order languages does not lift the strong typing of classical predicate logic discussed in the next section. In classical predicate logic, a variable cannot range over both terms and formulas. The following example is an another case of confounding of object and meta-variables that cannot be accounted for in classical predicate logic by the aforementioned symbol overloading:

```
r(X) :- X(X)
```

For the same reason, Russell's Paradox ($\star$) is not expressible in a first-order logic language overloading symbols.

## 10 Reflective Predicate Logic is a conservative extension of first-order logic

Reflective Predicate Logic's syntax differs from that of first-order logic in having only one category of expressions, whereas first-order logic distinguishes between terms and formulas. First-order logic terms can be expressed in Reflective Predicate Logic as follows:

*Definition 7* (*First-Order Terms*)
Let $\mathscr{L}$ be a Reflective Predicate Logic language the set of non-logical symbols of which is $S$. A first-order term fragment $\mathscr{F}_t$ of $\mathscr{L}$ is specified by:

- A set $V \subseteq S$ of variables and a set $T \subseteq S$ of term symbols such that $V \cap T = \emptyset$
- The assignment to each element of $T$ of at least one arity (i.e., non-negative integer)

The terms of $\mathscr{F}_t$ are inductively defined as follows:

- A variable is a term.
- A term symbol with arity 0 is a term.
- If $f$ is a term symbol with arity $n \geq 1$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

Term symbols with arity 0 are commonly called "constants", term symbols with arity $n \geq 1$ "function symbols". It is commonly required in mathematical logic that each term symbol has exactly one arity. This requirement is not necessary as it is recalled above in Section 9.

Neglecting that Reflective Predicate Logic requires parentheses around quantified and negated expressions that are superfluous in first-order logic, first-order logic formulas can be expressed in Reflective Predicate Logic as follows:

*Definition 8* (*First-Order Formulas*)
Let $\mathscr{L}$ be a Reflective Predicate Logic language, the set of non-logical symbols of which is $S$. A first-order fragment $\mathscr{F}$ of $\mathscr{L}$ is specified by a first-order term fragment of $\mathscr{L}$ with set of variables $V$ and set of term symbols $T$ and by

- A set $P \subseteq S$ of predicate symbols such that $V \cap P = \emptyset$
- The assignment to each element of $P$ of at least one arity (that is, non-negative integer)

The first-order formulas of $\mathscr{F}$ are inductively defined as follows:

- A predicate symbol with arity 0 is a formula.
- If $p$ is a predicate symbol with arity $n \geq 1$ and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula.
- If $F$ is a formula, then $(\neg F)$ is a formula.
- If $F_1$ and $F_2$ are formulas, then $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, and $(F_1 \Rightarrow F_2)$ are formulas.
- If $x$ a variable and $F$ is a formula, then $(\forall x\ F)$ and $(\exists x\ F)$ are formulas.

A formula $F$ is open if a variable $x$ occurring in a subexpression of $F$ is not in the scope of a quantification of the form $(\forall xE)$ or $(\exists xE)$. A formula is closed, or a sentence, if it is not open.

It is commonly required in mathematical logic that each predicate symbol has exactly one arity and that $P \cap T = \emptyset$. As it is recalled above in Section 9, these requirements are not necessary.

A closed formula which is an atomic expression, or atom, is commonly called a "ground atom".

Definitions 7 and 8 use and constrain rules of Definition 1. As a consequence, terms and formulas of a first-order logic fragment of a Reflective Predicate Logic language are expressions of that language.

*Definition 9* (*First-Order Herbrand Interpretations*)
Let $\mathscr{F}$ be a first-order fragment of a Reflective Predicate Logic language $\mathscr{L}$. A first-order Herbrand interpretation of $\mathscr{F}$ is specified as a set of ground atoms of $\mathscr{F}$. Satisfiability in a first-order Herbrand interpretation of $\mathscr{F}$ is defined as in Definition 6. If $I$ is a Herbrand interpretation of $\mathscr{L}$, the restriction of $I$ to the first-order fragment $\mathscr{F}$ of $\mathscr{L}$, noted $I_{\mathscr{F}}$, is the subset of $I$ consisting of the variant classes of atoms of $\mathscr{F}$.

Observe that if $I$ is a Herbrand interpretation of a Reflective Predicate Logic language $\mathscr{L}$ and if $\mathscr{F}$ is a first-order fragment of $\mathscr{L}$, then the restriction of $I$ to $\mathscr{F}$, $I_{\mathscr{F}}$, is a Herbrand interpretation in the sense of Definition 6. Herbrand interpretations of first-order logic languages are usually specified as sets of ground atoms, not sets of ground atoms' variant classes. However, since a ground atom is the single element of its variant class, Herbrand interpretations of first-order logic languages can be seen as sets of variant classes.

*Proposition 1* (*Conservative Extension*)
Consider

- $\mathscr{F}$ a first-order language with set of variables $V$, set of term symbols $T$, and set of predicate symbols $P$
- $\mathscr{F}_{RPL}$ the Reflective Predicate Logic language with set of non-logical symbols $V \cup T \cup P$
- $I$ a Herbrand interpretation of $\mathscr{F}$
- $F$ a formula of $\mathscr{F}$

Reflective Predicate Logic is a conservative extension of first-order logic, that is:

- $F$ is an expression of $\mathscr{F}_{RPL}$.

- $I \models_{FOL} F$ if and only if for all Herbrand interpretations $J$ of $\mathscr{F}_{RPL}$ such that $J_{\mathscr{F}} = I$, $J \models_{RPL} F$.

where $\models_{FOL}$ and $\models_{RPL}$ denote satisfiability in first-order and Reflective Predicate Logic interpretations, respectively.

*Proof.* The first point has been already observed above. The second point follows from the fact that satisfiability of a formula (or expression) in an interpretation is defined recursively on the formula's (or expression's) structure. The satisfiability of $F$ in $J$ therefore depends only on expressions built from $\mathscr{F}$'s vocabulary, that is, depends only on the subset $J_{\mathscr{F}} = I$ of $J$.

## 11 Conclusion

This article has given Prolog-style meta-programming, which is characterized by a confounding of terms and formulas and of object and meta-variables, a simple formalization, arguably the simplest, the most complete and the closest to the programming practice so far proposed. This formalization consists in a systematization of the syntax of Frege's logic, the precursor of classical predicate logic, and in a generalization of Herbrand model theory. The resulting logic, Reflective Predicate Logic, has been shown to be a conservative extension of first-order logic.

The aforementioned syntax systematization is simple: It consists in drawing the consequences of replacing expressions by their definitions and in an unconventional representation of variables easing meta-programming. This syntax systematization had been initiated in the article Chen *et al.* (1993) and completed in the articles Jiang (1994) and Kalsbeek and Jiang (1995).

The aforementioned generalization of Herbrand model theory is simple, too. It consists in considering reflective atoms, that is, atoms that might contain connectives, quantifiers, and variables, instead of ground atoms. This, too, had been initiated in the article Chen *et al.* (1993) and furthered, though in an unsatisfying manner, in the articles Jiang (1994) and Kalsbeek and Jiang (1995). A simple change, the identical interpretation of variant expressions, was enough to yield a satisfying Herbrand-style model theory.

The resulting logic, Reflective Predicate Logic, is simple. It differs from first-order logic in only three rather simple aspects. The syntax of Reflective Predicate Logic renounces type theory, which makes it simpler than the syntax of first-order logic. The syntax of Reflective Predicate Logic has an unconventional representation of variables. The model theory of Reflective Predicate Logic is based on a generalization of Herbrand model theory which gives rise to specify collections in Reflective Predicate Logic, generalizations of sets which perfectly interpret reflective expressions.

In spite of its simplicity, Reflective Predicate Logic is significant to logic programming, knowledge representation, and mathematical logic. Reflective Predicate Logic is significant to logic programming because it accommodates Prolog-style meta-programming without the restrictions required by the formalizations of meta-programming in higher-order logic, without the restrictions and without the encodings (or naming relations) required by the formalizations of meta-programming in first-order logic, and without the inadequacies of HiLog and Ambivalent Logic (discussed in Section 3). Reflective Predicate

Logic is also significant to logic programming because it provides a justification to the confounding of object and meta-variables, the so-called "non-ground representations" (mentioned in Section 3) of efficient deduction systems. Reflective Predicate Logic is significant to knowledge representation because it is natively reflective thus considerably simpler than the standard approaches to reflection that are based on reification. Reflective Predicate Logic is significant to mathematical logic because it is an alternative to type theory and a rehabilitation of Frege's logic. The simplicity of Reflective Predicate Logic contributes to its significance. Indeed, in science, simplicity is not a drawback but instead an advantage.

A further contribution of this article is its handling of Russell's Paradox of self-reflectivity by proposing a logic in which the paradox is expressible. The widespread common wisdom is instead that a well-defined logic should preclude paradoxes. A paradox is a counter-intuitive inconsistency. A thesis of this article is that there is nothing problematic with a logic in which inconsistencies, be they intuitive or counter-intuitive, can be expressed.

This article is a first step. Further work should be devoted to:

- generalizing the model theory of this article to universes of all kinds, possibly including universes that are collections in the sense of Section 4 instead of sets,
- giving Reflective Predicate Logic a unification and a resolution calculus,
- specifying a logic programming syntax, preferably with a type system, based on Reflective Predicate Logic,
- investigating how structuring constructs such as modules and embedded implications (Chen 1987; Miller 1989; Chen *et al.* 1993; Giordano and Olivetti 1994; Giordano *et al.* 1994; Haemmerlé and Fages 2006) can be formalized in Reflective Predicate Logic,
- investigating how self-reflection, especially as used in proving Gödel's incompleteness theorems, can be expressed in Reflective Predicate Logic.

## References

Aczel, P. 1977. An introduction to inductive definitions. In *Handbook of Mathematical Logic*, Jon Barwise, Ed. North-Holland, 739–782.

Aiello, L. C., Cecchi, C. and Sartini, D. 1986. Representation and use of meta-knowledge. *Proceedings of the IEEE 74,* 10, 1304–1321.

Apt, K. R. and Ben-Eliyahu, R. 1996. Meta-variables in logic programming, or in praise of ambivalent syntax. *Fundamenta Informaticae 28,* 1, 23–36.

Apt, K. R. and van Emden, M. 1982. Contributions to the theory of logic programming. *Journal of the Association for Computating Machinery 29*, 841–862.

Backus, J. W. 1959. The syntax and semantics of the proposed international algebraic language of Zürich ACM-GAMM conference. In *Proc. of the International Conference on Information Processing*, Jun. 1959, V. Veronese, Ed. Paris, France, UNESCO (Paris, France), R. Oldenbourg (München, Germany) and Butterworths (London, Great Britain), 125–132.

Barendregt, H. 1993. Lambda Calculi with types. In *Handbook of Logic in Computer Science.* Vol. 2, Background: Computational Structures. Oxford University Press, 117–309.

Barklund, J. 1989. What is a meta-variable in Prolog? In *Meta-Programming in Logic Programming*, Harvey Abramson and M. H. Rogers, Eds. MIT Press, 383–398.

Barklund, J. 1995. Metaprogramming in logic. *Encyclopedia of Computer Science and Technology 33*, 205–227. Supplement 18: Case-Based Reasoning to User Interface Software Tools.

Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A. 1994. SLD-resolution with reflection. In *Logic Programming, Proceedings of the 1994 International Symposium*, M. Bruynooghe, Ed. Melbourne, Australia, 554–568.

Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A. 1995a. Semantical properties of encodings in logic programming. In *Logic Programming – The 1995 International Symposium*, J. W. Lloyd, Ed. MIT Press, Portland, Oregon, USA, 288–302.

Barklund, J., Dell'Acqua, P., Costantini, S. and Lanzarone, G. A. 1995b. Semantical properties of SLD-resolution with reflection. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Tokyo, Japan, 830.

Barklund, J., Dell'Acqua, P., Costantini, S. and Lanzarone, G. A. 2000. Reflection principles in computational logic. *Journal of Logic and Computation 10*, 743–786.

Benjamin, P. 1990. A meta level manifesto. In *Machine Learning, Meta-reasoning and Logics*, P. B. Brazdil and K. Konolige, Eds. Kluwer Academic Publishers, 3–17.

Bezem, M. 1999. Extensionality of simply typed logic programs. In *Proc. of the 16th International Conference on Logic Programming (ICLP)*, D. de Schreye, Ed. MIT Press, 395–410.

Bezem, M. 2001. An improved extensionality criterion for higher-order logic programs. In *Computer Science Logic, 15th International Workshop, CSL*. Vol. 2142, LNCS. L. Fribourg, Ed. Springer-Verlag, Paris, France, 5, 203–216.

Böhm, C. and Jacopini, G. 1966. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, **9**, 366–371.

Boole, G. 1854. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. Macmillan. Reprinted by Dover Publications, New York, 1958, and by Cambridge University Press, 2009.

Bowen, K. and Kowalski, R. 1982. Amalgamating language and metalanguage in logic programming. In *Logic Programming*, K. Clark and S. A. Tärnlund, Eds. Academic Press, 153–173.

Bowen, K. A. 1985. Meta-level programming and knowledge representation. *New Generation Computing 3*, 359–383.

Bowen, K. A. and Weinberg, T. 1985. A meta-level extension of Prolog. In *Proc. of the IEEE Symposium on Logic Programming*, J. Cohen and J. Conery, Eds. IEEE, Boston, Massachusetts, USA, 669–675.

Burgess, J. P. 2005. *Fixing Frege*. Princeton University Press, Princeton, NJ, USA.

Cardelli, L. 2004. Type systems. In *CRC Handbook of Computer Science and Engineering*, A. B. Tucker, Ed. CRC Press, 2208–2236.

Cardelli, L. and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17,* 4 (December), 471–522.

Cervesato, I. and Rossi, G. 1992. Logic meta-programming facilities in 'LOG. Research Showcase @ CMU 6-1992, Carnegie Mellon University, School of Computer Science, Computer Science Department.

Chang, C.-L. and Lee, R. C.-T. 1997. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press.

Charalambidis, A., Handjopoulos, K., Rondogiannis, P. and william L.W̃adge. 2013. Extensional higher-order logic programming. *ACM Transactions on Computational Logic 14,* 3, 91–103.

Chen, W. 1987. A theory of modules based on second-order logic. In *Proc. IEEE Symposium on Logic Programming*, IEEE Computer Society, San Francisco, California, USA, 24–33.

Chen, W., Kifer, M. and Warren, D. S. 1993. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming 15,* 3, 187–230.

Church, A. 1932. A set of postulates for the foundation of logic. *Annals of Mathematics – Series 2 33,* 2, 346–366.

Church, A. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

Chwistek, L. 1921. Antynomje logikiformalnej. *Przeglad Filozoficzny 24*, 164–171. In Polish, English translation: Chwistek (1967).

Chwistek, L. 1967. Antinomies of formal logic. In *Polish Logic: 1920–1939*, S. McCall, Ed. Clarendon Press, 338–345. English translation by Z. Jordan of Chwistek (1921).

Clocksin, W. F. and Mellish, C. S. 1981. *Programming in Prolog*. Springer-Verlag.

Costantini, S. 2002. Meta-Reasoning: A survey. In *Computational Logic: Logic Programming and Beyond (Festschrift in honour of Robert Kowalski)*, A. C. Kakas and F. Sadri, Eds. Vol. 2408, LNCS. Springer-Verlag, 254–288.

Costantini, S. and Lanzarone, G. A. 1989. A metalogic programming language. In *Proc. of the International Conference on Logic Programming*, Jun. 1989, G. Levi and M. Martelli, Ed. Lisbon, Portugal, 218–233.

Costantini, S. and Lanzarone, G. A. 1994. A metalogic programming approach: Language, semantics and applications. *Journal of Experimental & Theoretical Artificial Intelligence 6,* 3, 239–287.

Damas, L. 1985. *Type Assignment in Programming Languages*. Ph.D. thesis, Report No. CST-33-85, University of Edinburgh.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *Proc. of the 9th Symposium on Principles of Programming Languages (POPL)*, R. DeMillo, Ed. ACM, Albuquerque, New Mexico, USA, 207–212.

De Schreye D. and Martens, B. 1992. A sensible least Herbrand semantics for untyped vanilla meta-programming and its extension to a limited form of amalgamation. In *Meta-Programming in Logic, Proc. of the 3rd International Workshop on Meta-Programming (META)*, Jun. 1992, A. Peterossi, Ed. LNCS. Springer-Verlag, Uppsala, Sweden, 192–204.

Dijkstra, E. W. 1968. Letters to the editor: Go to statement considered harmful. *Communications of the ACM 11,* 3, 147–148.

Eshghi, K. 1986. *Meta-language in Logic Programming*. Ph.D. thesis, Department of Computing, Imperial College of Science and Technnology, University of London, UK.

Feferman, S. 2005. Predicativity. In *The Oxford Handbook of Philosophy of Mathematics and Logic*, S. Shapiro, Ed. Oxford University Press, 590–624.

Frege, G. 1879. *Begriffsschrift – Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Louis Nebert, Halle an der Saale. In German, English translation: Frege (1967, 2002).

Frege, G. 1893. *Grundgesetze der Arithmetik, Band I*. Verlag Herman Pohle, Jena. In German, partial English translation: Frege (1964).

Frege, G. 1903. *Grundgesetze der Arithmetik, Band II*. Verlag Herman Pohle, Jena. In German, partial English translation: Frege (1964).

Frege, G. 1964. *The Basic Laws of Arithmetic*. University of California Press, Berkeley, CA, USA. Partial English translation of Frege (1893, 1903) by Montgomery Furth.

Frege, G. 1967, 2002. A concept notation: A formula language of pure thought, modelled upon that of arithmetic. In *From Frege to Gödel: A Sourcebook in Mathematical Logic, 1879-1931*, J. van Heijenoort, Ed. Harvard University Press, 1–82, English translation of Frege (1879).

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the Fifth International Conference on Logic Programming (ICLP)*, Aug. 1988, K. A. Bowen and R. A. Kowalski, Eds. MIT Press, Seattle, Washington, 1070–1080.

Gentzen, G. 1934. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift 39,* 2, 176–210. In German. English translations: Gentzen (1964), Gentzen (1969).

Gentzen, G. 1964. Investigations into logical deduction. *American Philosophical Quarterly 1,* 4 (October), 288–306. English translation of Gentzen (1934).

Gentzen, G. 1969. Investigations into logical deduction. In *The Collected Works of Gerhard Gentzen*, M. E. Szabo, Ed. Studies in logic and the foundations of mathematics. North-Holland, 68–131. English translation of Gentzen (1934).

Giordano, L., Martelli, A. and Rossi, G. 1994. Structured Prolog: A language for structured logic programming. *Software-Concepts and Tools 15*, 125–145.

Giordano, L. and Olivetti, N. 1994. Combining negation as failure and embedded implications in logic programs. *Journal of Logic Programming 19,* 20, 1–679.

Gödel, K. 1930. *Über die Vollständigkeit des Logikkalküls.* Doctoral thesis, University of Vienna, Austria. In German, English translation: Gödel (2002).

Gödel, K. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsheft für Mathematik und Physik 38*, 173–198. In German, English translation: Gödel (1986).

Gödel, K. 1944. Russell's mathematical logic. In *The Philosophy of Bertrand Russell*, P. A. Schilpp, Ed. Northwestern University, Chicago, IL, USA, 123–153.

Gödel, K. 1967, 2002. The completeness of the Axioms of the functional calculus of logic. In *From Frege to Gödel: A Sourcebook in Mathematical Logic, 1879-1931.* J. van Heijenoort, Ed. Harvard University Press, 582–591.

Gödel, K. 1986. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems I.* Vol. I Publications 1929–1936. Oxford University Press. English translation of Gödel (1930).

Graf, P. 1996. *Term Indexing.* Vol. 1053, LNCS. Springer-Verlag. Doctoral thesis, Saarland University, Germany, 1995.

Haemmerlé, R. and Fages, F. 2006. Modules for Prolog revisited. In *Logic Programming – International Conference on Logic Programming (ICLP)*, S. Etalle and M. Truszczynski, Ed. Springer-Verlag, Seattle, Washington, USA. 41–55.

Halpern, J. Y. and Moses, Y. 1985. A guide to the modal logics of knowledge and belief. In *Proc. 9th International Joint Conference on Artificial Intelligence (IJCAI)*, A. Joshi, Ed. Vol. 1, Morgan Kaufmann, Los Angeles, California, USA, 480–490.

Harper, R., Honsell, F. and Plotkin, G. 1993. A framework for defining logics. *Journal of the ACM (JACM) 40,* 1, 143–184.

Higgins, C. P. 1996. On the declarative and procedural semantics of definite metalogic programs. *Journal of Logic and Computation 6,* 3, 363–407.

Hill, P. and Lloyd, J. W. 1994. *The Gödel Programming Language.* MIT Press.

Hill, P. M. and Lloyd, J. W. 1988. Analysis of meta-programs. In *Meta-Programming in Logic Programming, Workshop on Meta-Programming in Logic (META-88)*, Jun. 1988, H. Abramson and M. H. Rogers, Eds. MIT Press, Bristol, Great Britain, 23–51.

Hindley, R. J. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146*, Bristol, Great Britain, 29–60.

Hintikka, J. 1964. *Knowledge and Belief – An Introduction to the Logic of the Two Notions*, 2nd ed. Cornell University Press.

Huet, G. P. 1976. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. Doctoral thesis, Mathématiques, Université Paris VII.

Jiang, Y. 1994. Ambivalent Logic as the semantic basis of logic programming. In *Proc. International Conference on Logic Programming*, P. V. Hentenryck, Ed. MIT Press, Santa Marherita Ligure, Italy, 387–401.

Kalsbeek, M. B. 1993. *The Vanilla Metainterpreter for Definite Logic Programs and Ambivalent Syntax*. Tech. Rep. CT 93-01, Institute for Logic Language and Computation, University of Amsterdam.

Kalsbeek, M. B. and Jiang, Y. 1995. A vademecum of ambivalent logic. In *Meta-Logics and Logic Programming*, K. R. Apt and F. Turini, Ed. MIT Press, 27–56.

Kowalski, R. 1973. Predicate logic as a programming language. Memo 70, Department of Artificial Intelligence, Edinburgh University. Also in: *Proc. IFIP Congress*, Stockholm, North Holland, 1974, 569–574.

Kowalski, R. 1979. *Logic For Problem Solving*. North Holland.

Kowalski, R. and Kim, J.-S. 1991. A metalogic approach to multi-agent knowledge and belief. In *Artificial and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lipschitz, Ed. Academic Press, 231–246.

Levi, G. and Ramundo, D. 1993. A formalization of meta-programming for real. In *Proc. of the 10th International Conference on Logic Programming*, Jun. 1993, D. S. Warren, Ed. MIT Press, Budapest, Hungary, 354–373.

Levy, J. and Veanes, M. 2000. On the undecidability of second-order unification. *Information and Computation 159,* 1–2, 125–150.

Link, G., Ed. 2004. *One Hundred Years of Russell's Paradox: Mathematics, Logic, Philosophy*. de Gruyter.

Liskov, B. and Zilles, S. 1974. Programming with abstract data types. *ACM SIGPLAN Notices 9,* 4, 50–59.

Lismont, L. and Mongin, P. 1994. On the logic of common belief and common knowledge. *Theory and Decision 37*, 75–106.

Lloyd, J. W. 1987. *Foundation of Logic Programming*, 2nd ed. Springer-Verlag.

Martelli, A. and Rossi, G. 1988. Enhancing Prolog to support Prolog programming environments. In *ESOP'88: 2nd European Symposium on Programming*, Mar. 1988, Vol. 300, LNCS. H. Ganzinger, Ed. Springer-Verlag, Nancy, France, 317–327.

McCarthy, J. and Levin, M. I. 1965. *Lisp 1.5 Programmer's Manual*. Tech. Rep., Computation Center, Massachusetts Institute of Technology, USA.

McCarthy, J., Sato, M., Hayashi, T. and Igrashi, S. 1978. *On the Model Theory of Knowledge*. Tech. Rep., AIM-312, Stanford University.

Miller, D. 1989. A logical analysis of modules in logic programming. *Journal of Logic Programming 6,* 1–2, 79–108.

Miller, D. and Nadathur, G. 1988. An overview of Lambda-Prolog. In *Proc. Fifth International Conference and Symposium on Logic Programming*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Seattle, Washington, USA, 810–827.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic*. Cambridge University Press.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Science 17*, 348–374.

Moschovakis, J. 1999, 2015. *Intuitionistic Logic*. The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Stanford, CA 94305-4115. https://plato.stanford.edu/entries/logic-intuitionistic/.

O'Keefe, R. 1990. *The Craft of Prolog*. MIT Press.

Perlis, A. J. and Samelson, K. 1958. Preliminary report: International algebraic language. *Communications of the ACM 1,* 12, 8–22.

Perlis, D. 1985. Languages with self-reference i: Foundations. *Artificial Intelligence 25,* 3, 307–322.

Perlis, D. 1988a. Languages with self-reference ii: Knowledge, belief, and modality. *Artificial Intelligence 34,* 2, 179–212.

Perlis, D. 1988b. Meta in logic. In *Meta-level Architectures and Reflection*, P. Maes and D. Nardi, Ed. North-Holland, 37–49.

Pfenning, F. 1991. *Logic Programming in the LF Logical Framework*. Research Rep., School of Computer Science, Carnegie Mellon University.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf – A meta-logical framework for deductive systems. In *Proc. of the 16th International Conference on Automated Deduction (CADE)*. Vol. 1632, LNAI. H. Ganzinger, Ed. Springer-Verlag, Trento, Italy, 202–206.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

Ramsey, F. P. 1926. The foundations of mathematics. *Proceedings of the London Mathematical Society, Series 2 25*, 338–384.

Robinson, A. and Voronkov, A., Eds. 2001. *Handbook of Automated Reasoning*. Vol. 1. ScienceDirect.

Robinson, G. and Wos, L. 1968. Completeness of paramodulation. In *Proc. of the Spring 1968 Meeting of the Association for Symbolic Logic, Journal of Symbolic Logic,* Vol. 34, (1969), 102–103.

Robinson, G. and Wos, L. 1969. Paramodulation and theorem proving in first order theories with equality. In *Machine Intelligence 4*. Edinburgh University Press, 135–150. *Proc. of the Fourth Annual Machine Intelligence Workshop*. Edinburgh, 1968. Reprinted in Siekmann and Wrightson (1983, pp. 98–3134).

Rossi, G. 1989. Meta-programming facilities in an extended Prolog. In *Proc. of the Fifth International Conference on Artificial Intelligence and Information-Control Systems of Robots – 89*, I. Plander, Ed. World Scientific, distributed by North Holland, Štrbské Pleso, Czechoslovakia

Rossi, G. 1992. Programs as data in an extended Prolog. *The Computer Journal 36,* 3, 217–226.

Russell, B. 1907. On some difficulties in the theory of transfinite numbers and order types. *Proceedings of the London Mathematical Society s2-4,* 1, 29–53.

Russell, B. 1908. Mathematical logic as based on the theory of types. *American Journal of Mathematics 30*, 222–262.

Russell, B. 1986. *The Collected Papers of Bertrand Russell*. Vol. 6, Logical and Philosophical Papers 1909–13. George Allen & Unwin.

Siekmann, J. H. and Wrightson, G., Eds. 1983. *Automation of Reasoning*. Symbolic Computation (Artificial Intelligence). Vol. 1, Classical Papers on Computational Logic 1957–1966. Springer-Verlag.

Siklóssy, L. 1976. *Let's Talk Lisp*. Prentice-Hall.

Sterling, L. S. and Shapiro, E. Y. 1994. *The Art of Prolog*, 2nd ed. MIT Press.

Sugano, H. 1989. *Reflective Computation in Logic Language and Its Semantics*. Tech. Rep., International Institute for Advanced Study of Social and Information Science, Fujitsu Limited.

Sugano, H. 1990. Meta and reflective computation in logic programming and its semantics. In *Proc. of the 2nd Workshop on Meta-Programming in Logic Programming (META)*, M. Bruynooghe, Ed. Springer-Verlag, Leuven, Belgium, 19–34.

Tarski, A. 1935. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica : commentarii Societatis Philosophicae Polonorum 146*, 261–405. In German, English translation: Tarski (1935).

Tarski, A. 1935. The concept of truth in formalized languages. In *Logic, Semantics, Metamathematics, Papers from 1923 to 1938*, 2nd ed. J. Corcoran, Eds. Hackett Publishing Company, Inc., 152– 278.

Trump, D. 2017. "Why would Kim Jong-un insult me by calling me 'old', when I would NEVER call him 'short and fat'? Oh well, I try so hard to be his friend – And maybe someday

that will happen!". Twitter Web Client https://twitter.com/realdonaldtrump/status/929511061954297857.

van Ditmarsch, H., Halpern, J. Y., van der Hoek, W. and Kooi, B., Eds. 2015. *Handbook of Epistemic Logic*. College Publications.

van Emden, M. and Kowalski, R. 1976. The semantics of predicate logic as a programming language. *Journal of the Association for Computating Machinery 23*, 733–742.

van Harmelen, F. 1989. A classification of metalevel architectures. In *Logic-based Knowledge Representation*. P. M. Jackson, J. Reichgelt, and F. van Harmelen, Eds. MIT Press, 1–35.

van Harmelen, F. 1992. Definable naming relations in meta-level systems. In *Proc. of the Third Workshop on Meta-Programming in Logic (META)*, Uppsala, Sweden, Vol. 649, LNCS, A. Pettorossi, Ed. Springer-Verlag, 89–104.

von Kutschera, F. 1989. *Gottlob Frege: Eine Einführung in sein Werk*, de Gruyter, Berlin. In German.

Wadge, W. W. 1991. Higher-order horn-logic programming. In *Logic Programming: Proceedings of the 1991 International Symposium*, V. A. Saraswat and K. Ueda, Eds. MIT Press, 289–303.

Weyhrauch, R. W. 1980. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence 13,* 1–2, 133–170.

Whitehead, A. N. and Russell, B. 1910, 1912, 1913. *Principia Mathematica*. Vols. 1–3. Cambridge University Press.

## Appendix: A brief introduction into Frege's logic

This brief introduction into Frege's logic aims at providing the material necessary for understanding the references to that logic given in the article. It is neither intended as a presentation of Frege's often subtle thoughts, nor as a presentation of the number theories for which Frege developed his logic and nor as a presentation of Frege's terminology and notations that have become outdated. This brief introduction owes to both von Kutschera (1989) and Burgess (2005) even though it slightly differs from the presentations of Frege's logic by these authors.

Frege's logic is defined in three books:

- "Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens" (Frege 1879) translated in English as "A Concept Notation: A formula language of pure thought, modelled upon that of arithmetic" (Frege 1967, 2002).
- "Grundgesetze der Arithmetik" volumes I and II (Frege 1893; 1903) partly translated in English as "The Basic Laws of Arithmetic" (Frege 1964).

Frege's logic is the archetype of predicate logic as it is known today. However, it departs from predicate logic in several aspects of various significance.

A first salient but insignificant aspect of Frege's logic is its two-dimensional syntax that, even though decried by logicians, makes much sense to a computer scientist because it reminds of how, since the 70s of the 20th century, structured programs (Böhm and Jacopini 1966; Dijkstra 1968) are rendered or "pretty printed" for better readability.

A second salient and important aspect of Frege's logic is that its syntax covers both what are called today the logical and (some of) the meta-logical language. In a manner that reminds of Prolog meta-programming, Frege's logic language includes notations for assumptions, theorems, logical equivalence (noted nowadays $\models\models$), and the extension of a predicate (in the sense of the assignment of truth values to atoms), and the truth values

predicate (in the sense of the assignment of truth values to atoms), and the truth values "true" and "false". Frege's logic includes the following symbols that, nowadays, are seen as meta-logical:

- = used for expressing that its two arguments (sentences) have the same truth value (depending on the context, this is expressed nowadays using $\models\models$ or $\Leftrightarrow$);
- |– used for introducing an assumption;
- ||– used for introducing a theorem;
- ext used as in ext $\alpha$ $\Phi(\alpha)$ for denoting the "course of values" or, as the notation suggests, the extension, that is, the graph of the function $\Phi$.

Frege's logic also includes the symbol – as a mark for the beginning of a (two-dimensional) sentence.

A third salient aspect of Frege's logic is that the denotation, or literal meaning, of each of its sentences (or closed formulas) is a truth value "true" or "false". Consistently with this, predicates called "concepts" in Frege's logic are functions mapping their defining sentences to truth values. For avoiding confusions, the name "concept" is used in the following for referring to the functional predicates of Frege's logic.

Frege's logic has first-order terms examples of which are the numbers 1 and 3 and the compound term $1 + 3$ built using the function symbol $+$. Frege's logic has atomic sentences built from concepts of arity 1 or 2. Interestingly, Frege's logic has no concepts of arity 0 that would correspond to propositional variables. Frege did not make use of concepts of arities greater than 2 because his number theories are conveniently expressed without such concepts. In Frege's logic, compound sentences are built very much like in nowadays' predicate logic from the connectives $\neg$ and $\Rightarrow$ and the universal quantifier $\forall$ that can be applied to terms as well as concepts. As Frege points out, concepts can be first-order (if they predicate only of terms) or second-order (if they predicate of concepts):

> "We call first-order functions functions the arguments of which are objects, second-order functions functions the arguments of which are first-order functions." (Frege 1893, p. 36).[1]

Frege points out how conjunction ($\wedge$), disjunction ($\vee$), and equivalence ($\Leftrightarrow$) can be expressed using negation ($\neg$) and implication ($\Rightarrow$) and how existential quantification ($\exists$) can be expressed using negation ($\neg$) and universal quantification ($\forall$).

Frege's logic has a proof calculus but no concept of interpretation, that is, no model theory. Interpretations would be introduced later and used in 1930 by Gödel for proving the completeness of the proof calculus of Frege's logic (Gödel 1930).

In Frege's logic, concept symbols but no formulas can occur as argument of second-order concepts. Let $S[a]$ denote a sentence in which the constant $a$ might occur and let $S[x/a]$ denote the expression obtained by replacing in $S$ each occurrence of $a$ by a variable $x$. Basic Law V of Frege's logic is an axiom making it possible to define a concept $c$ from a sentence $S$ by an expression amounting to $\forall x\ c(x) = S[x/a]$. Basic Law V (Frege 1893, § 20 p. 35) states:

$$\text{|– } (\text{ext } \epsilon\ (f(\epsilon)) = \text{ext } \alpha\ (g(\alpha))) = \forall \mathfrak{a}\ (f(\mathfrak{a}) = g(\mathfrak{a}))$$

[1] "Wir nennen nun die Functionen, deren Argumente Gegenstände sind, Functionen erster Stufe; die Functionen dagegen, deren Argumente Functionen erster Stufe sind, mögen Functionen zweiter Stufe heissen."

which means that the courses of values of two concepts, $\epsilon$ and $\alpha$, are identical if and only if the open formulas defining these concepts, $f$ and $g$ respectively, have the same truth values for all the values of their variables. (Note the use in Basic Law V of the symbol $\vdash$ for "assumption" or "axiom", and the second and third occurrences of $=$ expressing that two formulas denote the same truth value, that is, are logically equivalent.)

Basic Law V makes it possible to define what Frege calls a "first-order concept" $c$ that holds of all natural numbers (that is, non-negative integers) that are both even and odd. Since $c$'s defining expression, natural numbers being both even and odd, is inconsistent, the course of values $\mathrm{ext}(c)$ of $c$ is, in nowadays notation, $\{(n, false) \mid n \in \mathbb{N}\}$ and $c$ specifies an empty set.

Basic Law V also makes it possible to define what Frege calls a "second-order concept" $e$ as follows:

$$(\star\star) \quad \vdash \forall x \;\; e(x) = \neg x(x)$$

that is, apart from the use of symbol $\vdash$ for expressing an assumption and of $=$ instead of $\Leftrightarrow$, is exactly the definition of Russell's Paradox in Reflective Predicate Logic ($\star$) given in Section 6.

In contrast to the aforementioned concept $c$ that holds of all natural numbers that are both even and odd, $e$'s defining expression $\neg x(x)$, a concept not holding of itself, is not inconsistent. It is the whole sentence $\forall x \; e(x) = \neg x(x)$ that is inconsistent. Indeed, instantiating the variable $x$ with $e$ in that sentence yields $e(e) = \neg e(e)$, that is, in nowadays syntax $(e(e) \Leftrightarrow \neg e(e))$. Since its tentative definition is an inconsistent sentence, $e$ does not exist, hence its course of values does not exist either, that is, $e$ does not specify anything at all, not even an empty set.

Russell's Paradox reminds of a propositional variable $p$ defined by the sentence $p = \neg p$ in Frege logic's syntax or $(p \Leftrightarrow \neg p)$ in nowadays syntax. Since its tentative definition is an inconsistent sentence, $p$ does not exist, hence its course of values does not exist either, that is, $p$ does not specify anything at all.

Thus, the inconsistency of Frege's logic does not result from Basic Law V in itself. It results from both, Frege logic's impredicative atoms and Basic Law V, that together make possible inconsistent concept definitions like Russell's paradox ($\star\star$).

Russell devised his Ramified Theory of Types (Russell 1908) so as to make impossible inconsistent sentences like that of the paradox bearing his name. Russell's Ramified Theory of Types requires that a higher-order concept applies only of concepts of the immediately preceding order. In other words, Russell's Ramified Theory of Types precludes impredicative atoms like $e(e)$ or `belief(belief(itRains))`. Thus, Russell's fix of Frege's logic ensures the consistency of axioms resulting from Basic Law V by precluding not only inconsistent axioms but also some expressions including consistent sentences.

Interestingly, the Ramified Theory of Types does not preclude that a propositional variable $p$ is defined by the inconsistent sentence $(p \Leftrightarrow \neg p)$. Russell's position was not flawed, though. Since the Vicious Circle Principle (Russell 1907; 1986) he advocated for (see Section4) forbids to define something by referring to that same thing, that principle also forbids to define a propositional variable $p$ by $(p \Leftrightarrow \neg p)$. Thus, Russell had no reasons to preclude such inconsistent definitions of propositional variables with his Ramified Theory of Types.

It is nonetheless puzzling that the obvious fix of Frege's logic consisting in requiring that concept definitions are consistent would have been immediately apparent if Frege had included propositional variables in his logic that, three decades earlier, George Boole had proposed and formalized (Boole 1854).

It is tempting to think that following Frege's inspiration, a logic with impredicative atoms similar to Reflective Predicate Logic could have been proposed and accepted much earlier. This, however, is doubtful. Indeed, one essential step towards Frege's goal of specifying number theories was to provide a set theory. Russell's objection to Frege's logic was rooted at the kind of "collections" Frege's logic and Reflective Predicate Logic, because of their impredicative atoms, give rise to define. Such "collections", like the collection of beliefs mentioned in Section 4, make much sense in knowledge representation and in formalizing meta-programming. For specifying number theories, however, they are more complicated than necessary.