In Praise of Impredicativity: A Contribution to the Formalisation of Meta-Programming

François Bry
Institute for Informatics, Ludwig-Maximilian University of Munich, Germany

January 2018

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 formalisations of meta-programming do not explicitly address its impredicativity and are cumbersome. This article aims at overcoming this unsatisfactory situation by discussing the relevance of impredicativity to meta-programming and by revisiting the syntax and model theory of Ambivalent Logic, an impredicative logic the language and model theory of which are conservative extensions of the language and model theory of first-order logic.

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 what releases 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 "Amalgamation" in [9]:

```
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

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

includes a clause:

```
itIsWet ← itRains
```

This fact violates the syntaxes of most logics that require that a fact be formed from a predicate, like `believes`, and a list of terms like `ann` but unlike the clause `itIsWet ← itRains`. Indeed, in most logics a clause is a formula, not a term.

Examples referring to beliefs and trust are given in this article because they are intuitive. This article does not address how to specify belief and trust systems but instead how to formalise 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 formalisations of meta-programming pay a tribute to this dictate of classical logic: They require to code a clause like

```
itIsWet ← itRains
```

as a term like

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

when it occurs within a fact, expressing the second clause above as:

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

Such a contortion is not necessary. Atomic and compound formulas can be treated as terms, as Ambivalent Logic has shown [47,48]. 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 Ambivalent 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 [57] as a benchmark for testing the expressive power and naturalness of knowledge representation formalisms can be expressed in Ambivalent Logic exactly as it is expressed in [49].

Ambivalent Logic also allows expressions such as

```
(loves ∧ trusts)(ann, bill)
```

that can be defined by

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

```
(∀ 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 [35–37] by first assuming that `t(ann, bill)` 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 `t(ann, bill)`. 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 because they reflect in simple forms a widespread practice of meta-programming in Prolog [18, 43, 44, 63, 75].

This article corrects a deficiency of the model theory of Ambivalent Logic [47, 48]: The model theory proposed below in Section 8 ensures that two expressions like

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

(with the intended meaning that Bill believes that Ann believes everything) that differ only in the variables occurring in object-level expressions are identically interpreted. As pointed out in [47,48], in a same Ambivalent Logic interpretation, the one of these two expressions can be true and the other false.

Furthermore, the syntax proposed below differs from the original syntax of Amibivalent Logic by adopting the paradigm "quantification makes variables". Thanks to this paradigm, one can construct from an expression like `p(a, b)` in which `a` and `b` are constants, the expression `∀ a p(a, b)` in which `a` is a variable and `b` is a constant. The paradigm "quantification makes variables" makes it easy to generate from an expression `t(ann, bill)` a quantified expression `∀ t t(ann, bill)` what, as observed above, is needed in proving implications in the manner of Natural Deduction [35–37]. The paradigm "quantification makes variables" eases meta-programming as the following example shows. The formula

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

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

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

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

```
∀ charly ∃ itRains
 ((believes(charly, itRains) ∧ believes(charly, ¬ itRains))
    ⇒ ∀ X believes(charly, 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. Arguably, this is a low price to pay since explicit universal quantifications are beneficial to program readability and amount to variable declarations that, since ALGOL 58 [4, 65] 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 the logic has no open formulas. This is, however, not a restriction, since in classical logic 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.

In the following, the name "Ambivalent Logic" is kept for the revised syntax and model theory proposed in this article in recognition of the seminal nature the original proposal [47, 48].

This article is structured as follows: Section 1 is this introduction. Section 2 reports on related work. Section 3 is a comparison of meta-programming and higher-order logics. Section 4 recalls why predicativity has been sought for and under which conditions impredicativity is nowadays considered acceptable. Section 5 defines an Ambivalent Logic syntax allowing impredicative atoms under the paradigm "quantification makes variables". Section 6 discusses expressing the Barber and Russell's Paradoxes in Ambivalent Logic. Section 7 gives an approach for detecting variant expressions based on rectification, or standardisation apart, of Ambivalent logic expressions. Section 8 defines a model theory for Ambivalent Logic that corrects a deficiency of the model theory given in [47, 48]. Section 9 shows that Ambivalent Logic is a conservative extension of first-order logic. Section 10 is the conclusion.

The novel contributions of this article are as follows:

1. A discussion how meta-programming relates to higher-order logics and impredicativity.

2. A simplification of the syntax and model theory of Ambivalent Logic.

3. A correction of the model theory of Ambivalent Logic.

4. An explanation why a logic with impredicative atoms is not only acceptable but also desirable as a formalisation of meta-programming and for knowledge representation.

2 Related Work

Meta-programming has been considered since the early days of logic programming. It is discussed in [9, 50]. Meta-programming in Prolog is addressed among others in [18, 63, 75]. However, the standard formalisation of logic programming [55] does not cover meta-programming. Formalisations of meta-programming of three kinds have been proposed so far:

1. Formalisations relying on higher-order logics commonly called “higher-order logic programming”.
2. Formalisations representing formulas by terms.
3. Ambivalent Logic, a logic lifting the distinction between terms and formulas and allowing impredicative atoms.

Higher-order logic programming languages, most prominently λ-Prolog [59, 60], Elf [66] and Twelf [67], are based on higher-order logics. They have been used for specifying deductive systems and program-processing programs like parsers, compilers, interpreters, and static type checkers [66, 68].

Being based on higher-order logics, higher-order logic programming languages require each expression, term or formula, to belong to a single “order”, or type, and strictly distinguish between terms and formulas. Thus, higher-order logic programming languages preclude expressions like

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

that have formulas in places where first and higher-order logics allow only terms. As a consequence, higher-order logic programming languages do not support the flavour of meta-programming called “Amalgamation” [9]. In particular, the higher-orderdness of higher-order logic programming languages should prevent them from representing object variables by meta-variables.

Higher-order logic programming languages have been designed conforming to the Edinburgh Logical Framework LF [41], 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 [62] and on the typed lambda calculus [5]. As a consequence, higher-order logic programming languages have no model theories. Interestingly, LF represents object variables by meta-variables [41], a form of Amalgamation [9] actually alien to higher-order logics.

Meta-programming and higher-order logics are discussed in detail below in Section 3.

Formalisations representing formulas by terms. The formalisations of meta-programming Metalog [10, 11], the language proposed by Barklund in [6], Reflexive Prolog [20], R-Prolog* [76, 77], ’LOG (spoken “quotelog”) [15], HiLog [17], Gödel [43] and the language proposed by Higgins in [42] rely on representing, or mirroring, in different yet conceptually similar manners a formula or a program by one or two first-order terms thus keeping with the imperative of first and higher-order logics that formulas and terms are distinct categories.

Metalog uses "metalevel names" for representing object programs as first-order terms. Reflexive Prolog distinguishes meta-variables from object variables and represents object programs using first-order "name constants". R-Prolog* [76,77] uses a quote notation à la Lisp [56, 74] for representing object programs as first-order terms. Barklund proposes in [6] a "naming scheme" for representing object programs using terms built from reserved first-order function symbols and first-order constants. 'LOG relies on a "naming scheme" associating "with every syntactic object of the language, from characters to programs", a "constant name" and a structured ground term called the object's "structural representation". HiLog [17] 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 $n \in \mathbb{N}$.

Gödel [43, 44] is equipped with a variable typing distinguishing object variables from meta-variables and "ground representations" of programs as first-order terms. Higgins proposes in [42] a language relying on a "naming scheme" similar to that of 'LOG associating with every syntactic object a first-order "primitive name" and a first-order "structured name".

The articles [52,73] give meta-programming foundations in the form of a Herbrand model theory that also rely on mirroring formulas or programs as first-order terms in the meta-language of a "demo" or "solve" meta-predicate [9,50].

Ambivalent Logic [47,48] was the first formalisation of meta-programming giving up the distinction between terms and formulas thus, as a consequence, allowing impredicative atoms (see below Section 4) and making superfluous representations of formulas by terms the afore-mentioned languages and formalisms rely on.

It is surprising that the logic programming community payed little attention to this seminal and elegant proposal. The article [2] titled "Meta-Variables in Logic Programming, or in Praise of Ambivalent Syntax" is a notable exception. Relying on Ambivalent Logic's treatment of formulas as terms, and therefore of meta-variables as first-order variables, it extends SLD resolution [55] to logic programs with meta-variables and establishes the soundness and completeness of that extension.

The data modelling language RDF [21] provides a form of meta-programming by treating formulas of the form "subject property object", so-called triples, as terms: A triple can be for example the subject of another triple. However, RDF imposes syntactical restrictions on the use of properties in meta-level expressions: They cannot occur as triples' subjects.

Xcerpt [12, 13, 72], a declarative XML query language inspired from logic programming, is related to the present work insofar as it treats uniformly expressions amounting to formulas and terms of a logic language. Xcerpt however, is a query language and therefore neither aimed at full programming nor at meta-programming.

More on meta-programming and on meta-reasoning can be found in the survey [19].

3 Meta-Programming and Higher-Order Logics

Prolog-style meta-programming [18, 63, 75] called "Amalgamation" in [9] allows

1. variables to range over predicates and formulas,

2. predicates the arguments of which are predicates or formulas, and

3. reflection [24] in the sense that every predicate can have any predicate, including itself, and formulas including any predicate as argument.

Prolog's extremely permissive approach to meta-programming goes back to a fruitful disregard by Alain Colmerauer, Prolog's designer, of the relationship between meta-programming, higher-order logic, and impredicativity and to a time at which the undecidability of unification in second-order logic [53] and third-order logic [46] as well as Damas-Hindley-Milner type systems [22,23,45,61] were unknown. 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 both static and dynamic type checking.

- Reflection in the sense of a predicate occurring in an argument of itself can be used for applying an optimisation to the very predicate specifying this optimisation.

- Formulas occurring in place where first and higher-order logics expect terms are useful for expressing believes and trust as shown in the articles [9,49,50] and outlined in the introduction.

Can meta-programming be formalised in higher-order logics? In order to answer this question, let us recall higher-order logics main traits [27, 51]:

- First-order logic has (first-order) variables that are (first-order) terms and (first-order) predicates the arguments of which are (first-order) terms.
- Second-order logic extends first-order logic with second-order variables that are first-order predicates.
- Third-order extends second-order logic with (second-order) predicates the arguments of which are first-order predicates.
- Fourth-order logic extends third-order logic with fourth-order variables that are second-order predicates.
- Etc.

Disregarding function symbols, the model theories of higher-order logics are as follows:

- A first-order term is interpreted as an individual, that is, as an element of the universe of discourse.
- A first-order n-ary predicate is interpreted as a set of n-tuples of individuals.
- A second-order n-ary predicate is interpreted as a set of sets of n-tuples of individuals.
- A third-order n-ary predicate is interpreted as a set of sets of sets of n-tuples of individuals.
- Etc.

Prolog-style meta-programming resembles higher-order logics since it has variables for predicates or formulas and predicates the arguments of which are predicates or formulas. However, in contrast to Prolog-style meta-programming, higher-order logics allow no confusion of orders like

- A unary predicate ranging over all unary predicates including itself.
- A predicate being, or occurring in, an argument of itself.

Such confusions are widespread in Prolog-style meta-programming [9, 18, 50, 63, 75]. Thus, Prolog-style meta-programming cannot be fully formalised in higher-order logics.

4 Predicativity and Impredicativity

Predicativity [28] is an essential trait of classical logic atoms. Meta-programming in general, and its Amalgamation [9] flavour in particular build on impredicative atoms. This section is devoted to clarifying these issues.

Consider a property P on the nodes of an undirected graph G defined as follows: A node n of G has property P if its immediate neighbours all 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 Betrand Russell have proposed the Vicious Circle Principle [69, 71] 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, "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 [1] 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 [3, 79]. (The definition of a property P is hereditary if it states that whenever a natural number n has property P, so does $n+1$. A definite logic program is a program the clauses of which contain no negative literals.)

The Vicious Circle Principle also rejects the definition sketched above 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 fulfills the base and induction cases of its definition [1]. 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 reject definitions like the following that are widely accepted even though they are not inductive and they do not provide with 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.
- y is the greatest lower bound of an ordered set S if and only if for all elements x of S, y is less than or equal to x, and any z less than or equal to all elements of S is less than or equal to y.
- Eigenvector definitions like that of PageRank [64].
- The definition of the stable models of logic programs [34].

Such definitions have in common that they quantify over domains the definitions of which refer to the entities being defined.

Examples like the afore-mentioned led some mathematicians, most notably Kurt Gödel, to object that impredicative definitions are acceptable provided the entities they refer to are clearly apprehensible [39]. Nowadays, most logicians and mathematicians follow Gödel and accept impredicative definitions of the following kinds [1, 28]:

- Inductive definitions.
- Impredicative definitions that characterise 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 Gottlob Frege, whom Russell convinced, 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 [29–31], the precursor of first-order logic. (A brief introduction into Frege's logic is given in an appendix.) A predicate that "applies of itself" is the core of Russell's Paradox [54]: A S set of all sets that are not elements of themselves. (The paradox is that such a set cannot exist because if it existed and were an element of itself, then by definition it would not be an element of itself, and if it existed and were not an element of itself, then by definition it would be an element of itself.) The Vicious Circle Principle and paradoxes, among other the paradox bearing his name, motivated Russell to develop the Ramified Theory of Types [70], that is, logics of various, first and higher, orders. By considering a strict hierarchy of types, or orders, as recalled in Section 3, expressions with predicates "applying to themselves" are precluded.

Because classical logic adheres to Russell's Ramified Theory of Types, classical logic rejects expressions like the following that are at the core of meta-programming and of its flavour Amalgamation [9]:

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

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

So as 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 standard 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", or "reflexive sets", like the non-standard set of beliefs in the above example, is the essence of Ambivalent Logic's model theory [47, 48] and of its revision proposed below. The resulting impredicative definitions (in the example given above, the definition of beliefs) fulfills Gödel's condition to be interpreted in reference to clearly apprehensible entities (in the example given above, a standard set of closed atoms).

If, as pointed out above, impredicative definitions are (under conditions) acceptable in mathematics and in functional and logic programming, why should they be inacceptable as logical formulas and in logic programs? How could meta-programming, the purpose of which is the coding of novel program semantics, deploy its full potential if it is restricted to predicative definitions that, as pointed out above, do not suffice to formalise logic programming's semantics? If the rejection of impredicativity has been overcome in mathematics, why should it not be overcome as well in logic?

The next sections are devoted to Ambivalent Logic, a conservative extension of first-order logic in which impredicative atoms like `belief(belief(itRains))` can be expressed.

5 Ambivalent Logic's Syntax Revisited

This section introduces "expressions" that amount to both the terms and the formulas of classical logic languages. Except for the use of the paradigm "quantification makes variables" and a careful parenthesising ensuring that expressions are non-ambiguous, the syntax given below is that of Ambivalent Logic [47,48].

Definition 5.1 (Symbols and Expressions) *An Ambivalent 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 an Ambivalent 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. An non-logical or atomic expression, or atom, is an expression the outermost constructor of which is neither a connective nor a quantifier.

A countable infinite set of non-logical symbols could be considered in the above definition but this is not necessary in programming since programs are finite and a denumerable supply of vartiables, needed for building proofs, is assumed below in Section 7.

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

More parentheses are required by Definition 5.1 than in classical logic. 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 5.1, that is, the syntax given above is a conservative extension of the syntax of first-order logic. This issue is addressed in more details below in Section 9.

The expressions of an Ambivalent Logic language can be proven non-ambiguous similarly as classical logic formulas are proven non-ambiguous. The subexpressions of an expression of an Ambivalent Logic language can be similarly defined as the subformulas of a classical logic formula. In the following, an expression is considered a sub-expression of itself. The scope of a quantified variable in an expression is defined as usual: The scope of E_1 in $(\forall E_1\ E_2)$ and in $(\exists E_1\ E_2)$ is E_2 except those subexpressions of E_2 that are of the form $(\forall E_1\ F)$ or $(\exists E_1\ F)$.

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 expresssions. 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 definition above 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 an Ambivalent Logic language have all arities is a convenience, not a necessity. The definition above can be refined to a less permissive definition as of non-logical symbols' arities.

In contrast to the syntax of Ambivalent Logic given in [47, 48], 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 5.1 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 generalised 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 generalised 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 Ambivalent 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. This section argues that even though the Barber and Russell's paradoxes can be expressed in Ambivalent Logic, Ambivalent Logic is not paradoxical. Indeed, consistent and usefull specifications can be expressed in Ambivalent Logic.

Since Ambivalent 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 Ambivalent Logic:

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

where, extending Definition 5.1, $(E_1 \Leftrightarrow E_2)$ 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 [47, 48] and 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 [29–31] and Ambivalent Logic (a brief introduction into Frege's logic is given in an 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 do not make paradoxical 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 $(p \wedge \neg p)$ which is inconsitent 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 – see the appendix for a brief introduction into Frege's logic and an explanation of its inconsistency. While it is understandable that Russell, Frege and their contemporaries shared the (unexpressed) conception that every expression must define a set, this conception can, and even must, be given up. Giving up this conception provides for a logic perfectly formalising meta-programming and its flavour Amalgamation [9]. Giving up this conception allows for impredicative atoms interpreted as "reflexive sets" like that of beliefs mentioned above in Section 4, that is, non-standard sets that have some of their subsets, possibly themselves, as elements [28].

7 Variant Expressions and Expression Rectification

In the next section, a model theory is given for Ambivalent Logic such that two syntactically distinct atomic expressions that are variant 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 in the introduction, this is not the case with the model theory initially proposed for Ambivalent Logic [47,48].

Intuitively, two expressions are variants of each other if they mean the same. Formally, 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 σ of the variables occurring in E_1 into the variables occurring in E_2 such that applying σ to E_1 yieds 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 variant 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 Ambivalent 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 variant of each other, the first-order formulas

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

are variant 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 "rectification", as it is called in mathematical logic, or "standardisation apart", as it is called in automated reasoning [16,26]. Rectifying an expression consist 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 no variables are overridden in the resulting expression. Thus, rectifying the expression

```
(∀ X (p(X) ∧ ∃ X q(X)))
```

results in the expression

$$(\forall\ v_2\ \ (\mathtt{p}(v_2)\ \wedge\ \exists\ v_1\ \ \mathtt{q}(v_1)))$$

if the "fresh" variables considered are $v_1, v_2, \ldots$

It is assumed in the following that there is a denumerable supply $v_1, v_2, \ldots, v_i$, … of variables such that each v_i is distinct from every logical and every non-logical symbol of the Ambivalent 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.)

Rectification is performed by an inside-out variable renaming that is conveniently specified as a predicate *rect* recursively defined on an expression's structure. 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 7.1 (Rectification) *The 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 $E = E_1(E_2,\dots,E_n)$ with $n \geq 2$:*
  $rect(E,i,R_1(R_2,\dots,R_n),i_n)$ 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 $rect(E_n,i_{n-1},i_n)$
- *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 is obtained from R_2 by simultaneously replacing all occurrences of E_1 in R_2 by v_j.

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

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

The algorithm specified in Definition 7.1 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. 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}. Two distinct formulas of a rectified set of expressions do have variables in common.

In contrast to first-order formulas' rectification, the algorithm of Definition 7.1 considers the variables that might occur in the constructors of Ambivalent Logic atoms. The Ambivalent Logic atom

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

is for example rectified by the algorithm of Definition 7.1 into:

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

Definition 7.2 (Variant Expressions) *Two expressions E_1 and E_2 are variants of each other, noted $E_1 \sim E_2$, if their rectified are identical.*

The relation $\sim$ on the expressions of an Ambivalent Logic language is an equivalence relation.

8 Ambivalent Logic's Model Theory Revisited

Atoms in Ambivalent 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 Ambivalent Logic expressions, including all atoms, are closed expresssions thanks to the paradigm "quantification makes variables".
2. First-order logic atoms cannot have anything but terms as arguments, whereas atoms in Ambivalent 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 (like `likes`), whereas in Ambivalent Logic it can be any expression (like `(∀ T trust(T) ⇒ T)` and `likes`).

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 satisfies

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

regardless of whether none, only one, or both of

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

are satisfied in that interpretation [78]. 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 behaviours, 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, Ambivalent 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 an Ambivalent 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 what she herself believes) of an Ambivalent Logic language corresponds to both ground atoms and ground terms of first-order logic languages, the set of all expressions of an Ambivalent Logic language corresponds to both the Herbrand universe [16, 79] and the Herbrand base [16, 79] of a first-order logic language.

Definition 8.1 (Herbrand Universe) *Let $\mathscr{A}$ be the set of atoms of an Ambivalent 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, rectification (or standardisation 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 rectification 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 Ambivalent 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) what 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, after rectification, the expressions no longer contain 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 Ambivalent Logic language considered, incorrect instantiations like in the former example are impossible.

Definition 8.2 (Notations) *If A is an atom of an Ambivalent 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 and if A_1 and A_2 are atoms, then $R[A_2/A_1]$ denote the expression obtained from R by simultaneously replacing in R all occurrences of A_1 by A_2.

In defining the notation $R[A_2/A_1]$, there is no need for caring about overridden variables because that notation wil only apply to rectified expressions R and because, as observed in the previous section 7, in rectified expressions no variables are overridden.

Definition 8.3 (Interpretations and Models) *An Herbrand interpretation $I(S)$ of an Ambivalent 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 an 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.*

$I(S) \models A$	*iff*	$class(A) \in S$
$I(S) \models (\neg R)$	*iff*	$I(S) \not\models R$
$I(S) \models (R_1 \wedge R_2)$	*iff*	$I(S) \models R_1$ *and* $I(S) \models R_2$
$I(S) \models (R_1 \vee R_2)$	*iff*	$I(S) \models R_1$ *or* $I(S) \models R_2$
$I(S) \models (R_1 \Rightarrow R_2)$	*iff*	*if* $I(S) \models R_1$, *then* $I(S) \models R_2$
$I(S) \models (\exists v_i\, R)$	*iff*	$I(S) \models R[A/v_i]$ *for some A*
$I(S) \models (\forall v_i\, R)$	*iff*	$I(S) \models R[A/v_i]$ *for all A*

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 S, respectively) if it satisfies E (every expression in S, respectively).

Satisfaction of logical expressions (that is, 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 (that is, 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 a first-order logic expressions, Definitions 8.1 and 8.3 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 8.3 is a conservative extension of first-order logic's notions of Herbrand interpretations and models. This observation is formally developped in the next section 9.

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 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 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 [47,48] is more stringent than Definition 8.3: Instead of relying on the variant relationship, it requires syntactical identity. As a consequence, the set S of atoms given above does not specify a model in the sense of [47,48] of the set P 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 an Ambivalent 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 characterise elements of that universe even though they cannot be interpreted as standard sets.

9 Ambivalent Logic as a Conservative Extension of First-Order Logic

Ambivalent 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 Ambivalent Logic as follows:

Definition 9.1 (First-Order Terms) *Let $\mathscr{L}$ be an Ambivalent 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 (that is, 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* $t_1, \dots, t_n$ *are terms, then* $f(t_1, \dots, t_n)$ *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 and is often dispensed with in computer science.

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

Definition 9.2 (First-Order Formulas) *Let* $\mathscr{L}$ *be an Ambivalent 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* $t_1, \dots, t_n$ *are terms, then* $p(t_1, \dots, t_n)$ *is a formula.*
- *If* F *is a formula, then* $(\neg F)$ *is a formula.*
- *If* F_1 *and* F_2 *are formulas, then* $(F_1 \wedge F_2)$*,* $(F_1 \vee F_2)$*,* $(F_1 \Rightarrow F_2)$ *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* $(\forall x E)$ *or* $(\exists x E)$*. 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 $P \cap T = \emptyset$. These requirements are not necessary and are often dispensed with in computer science.

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

Definitions 9.1 and 9.2 use and constrain rules of Definition 5.1. As a consequence, terms and formulas of a first-order logic fragment of an Ambivalent Logic language are expressions of that language.

Definition 9.3 (First-Order Herbrand Interpretations) *Let $\mathscr{F}$ be a first-order fragment of an Ambivalent 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 8.3. 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 an Ambivalent 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 8.3. 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 9.1 (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*
- *$\mathscr{F}_{AL}$ the Ambivalent Logic language with set of non-logical symbols $V \cup T \cup P$*
- *I a Herbrand intrerpretation of $\mathscr{F}$*
- *F a formula of $\mathscr{F}$*

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

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

where $\models_{FOL}$ and $\models_{AL}$ denote satisfiability in first-order and Ambivalent 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.

10 Conclusion

This article has proposed a simplification of Ambivalent Logic's syntax and corrected a deficiency of Ambivalent Logic's model theory [47, 48]. Furthermore, it has stressed that Ambivalent Logic's impredicativity not only is necessary for a proper formalisation of meta-programming, but also makes as much sense in logic as in other areas of mathematics.

With the exception of Ambivalent Logic, all formalisations of meta-programming proposed so far remain in the realm of predicativity. In the case of higher order logic programming, the adherence to predicativity results in formalisations rejecting Amalgamation, an essential aspect of meta-programming. In the case of the formalisations representing formulas by terms, it results in complicated formalisations.

Ambivalent Logic's impredicativity has been shown in this article to make celebrated paradoxes expressible as inconsistent expressions, a perfectly appropriate manner to express paradoxes. Ambivalent Logic's impredicativity has been shown to be impeccable for reasons stressed in the past among others by Gödel and nowadays widely accepted.

Impredicativity frees from constraints and therefore holds the promise of novel and fruitful approaches to declarative programming and knowledge representation. This article's author believes that impredicativity is essential for meta-programming to deploy its full potential. Adopting an impredicative formalisation of meta-programming is promising for another reason: It provides a complement to functional programming's predicativity stressed by Damas-Hindley-Milner type systems [22, 23, 45, 61].

This article is only a first step. Much remains to be done, among others:

- Generalising the model theory of this article to universes of all kinds, possibly including universes that are "reflexive sets" of the type suggested in this article.

- Giving this article's logic a unification and a proof calculus, preferably a resolution calculus, and investigating their decidability and completeness. First investigations point to an homomorphism between this article's logic and first-order logic. Inspired by the article "Meta-Variables in Logic Programming, or in Praise of Ambivalent Syntax" [2], the author conjectures that a unification and a resolution calculus for this article's Ambivalent Logic can be similar to those of first-order logic.

- Specifying a program syntax and an immediate consequence operator based on this article's Ambivalent Logic.
- Investigating how constructs such as modules and embedded implications [58] can be expressed in programs built on this article's Ambivalent Logic.
- Defining impredicative types for logic programming amenable to static type checking.

Acknowledgments

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 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.

A 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, nor as a presentation of Frege's terminology and notations that have become outdated. This brief introduction owes to both Franz von Kutschera [80] and John P. Burgess [14] 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" [29] translated in English as "A Concept Notation: A formula language of pure thought, modelled upon that of arithmetic" [33]
- "Grundgesetze der Arithmetik" volumes I and II [30, 31] partly translated in English as "The Basic Laws of Arithmetic" [32].

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 70es of the 20th century, structured programs [7, 25] 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 meta-programming and Amalgamation [9], Frege's logic language includes notations for assumptions, theorems, logical equivalence (noted nowadays $\models\!\dashv$), and the extension of a 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 the truth values of its two arguments, that are sentences, are equal, what depending on the context is expressed nowadays using $\models\!\dashv$ or $\Leftrightarrow$
- $\vdash$ used for introducing an assumption
- $|\!\vdash$ 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 Φ

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." [30, 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 [38,40].

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 [30, § 20 p. 35] states:

$$\models (\text{ext}\ \varepsilon\ (f(\varepsilon)) = \text{ext}\ \alpha\ (g(\alpha))) = \forall \mathfrak{a}\ (f(\mathfrak{a}) = g(\mathfrak{a}))$$

what means that the courses of values of two concepts, ε and α, 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 $\models$ 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 $\text{ext}(c)$ 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:

$(\star\star)\quad \models\ \forall x\ \ e(x) = \neg x(x)$

[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."

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

In contrast to the afore-mentioned 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 [70] 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 immediatly 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 all kinds of 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 [69, 71] 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 none the less puzzling that the obvious fix of Frege's logic consisiting 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 formalised [8].

It is tempting to think that following Frege's inspiration, a logic with impredicative atoms similar to Ambivalent 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 "reflexive sets" Frege's logic and Ambivalent Logic, because of their impredicative atoms, give rise to define. Such "reflexive sets", like the set 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.

References

[1] Peter Aczel. *Handbook of Mathematical Logic*, chapter An introduction to inductive definitions, pages 739–782. North-Holland, 1977.

[2] Krzysztof R Apt and Rachel Ben-Eliyahu. Meta-variables in logic programming, or in praise of ambivalent syntax. *Fundamenta Informaticae*, 28(1):23–36, 1996.

[3] Krzysztof R. Apt and Maarten van Emden. Contributions to the theory of logic programming. *Journal of the Association for Computating Machinery*, 29:841–862, 1982.

[4] John W. Backus. The syntax and semantics of the proposed international algebraic language of Zürich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. UNESCO, 1959.

[5] Henk Barendregt. *Handbook of Logic in Computer Science*, volume 2. Background: Computational Structures, chapter Lambda Calculi with Types, pages 117—309. Oxford University Press, 1992.

[6] Jonas Barklund. *Meta-Programming in Logic Programming*, chapter What is a meta-variable in Prolog?, pages 383–398. MIT Press, 1989.

[7] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, pages 366—371, May 1966.

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

[9] Kenneth Bowen and Robert Kowalski. *Logic Programming*, chapter Amalgamating Language and Metalanguage in Logic Programming, pages 153–173. Academic Press, 1982.

[10] Kenneth A. Bowen. Meta-level programming and knowledge representation. *New Generation Computing*, 3:359–383, 1985.

[11] Kenneth A. Bowen and Tobias Weinberg. A meta-level extension of Prolog. In *Proceeding of the IEEE Symposium on Logic Programming*, pages 669–675. IEEE, 1985.

[12] François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proceedings International Conference on Logic Programming*. Springer-Verlag, 2002.

[13] François Bry and Sebastian Schaffert. An entailment relation for reasoning on the web. In *Proceedings Rules and Rule Markup Languages for the Semantic Web*, 2003.

[14] John P. Burgess. *Fixing Frege*. Princeton University Press, Princeton, New Jersey, 2005.

[15] Iliano Cervesato and Gianfranco Rossi. Logic meta-programming facilities in 'LOG. Research Showcase @ CMU 6-1992, Carnegie Mellon University, School of Computer Science, Computer Science Department, 1992.

[16] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Academic Press, 1997.

[17] Weidong Chen, Michael Kifer, and David Scott Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[18] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[19] Stefania Costantini. *Computational Logic: Logic Programming and Beyond (Festschrift in honour of Robert Kowalski)*, volume 2408 of *LNCS*, chapter Meta-reasoning: A Survey, pages 254–288. Springer-Verlag, 2002.

[20] Stefania Costantini and Gaetano Aurelio Lanzarone. A metalogic programming language. In *Proceedings of the International Conference on Logic Programming*, pages 218–233, 1989.

[21] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF concepts and abstract syntax. Recommendation, W3C, 2014.

[22] Luis Damas. Type assignment in programming languages. PhD thesis, report number CST-33-85, University of Edinburgh, 1985.

[23] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th Symposium on Principles of Programming Languages (POPL)*, pages 207–212. ACM, 1982.

[24] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: A short comparative study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI.*, pages 29–38, 1995.

[25] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[26] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2001. Second edition.

[27] Herbert B. Enderton. Second-order and higher-order logic. http://plato.stanford.edu/, 2015.

[28] Solomon Feferman. *The Oxford Handbook of Philosophy of Mathematics and Logic*, chapter Predicativity, pages 590–624. Oxford University Press, 2005.

[29] Gottlob Frege. *Begriffsschrift – Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Louis Nebert, Halle an der Saale, 1879.

[30] Gottlob Frege. *Grundgesetze der Arithmetik, Band I*. Verlag Herman Pohle, Jena, 1893.

[31] Gottlob Frege. *Grundgesetze der Arithmetik, Band II*. Verlag Herman Pohle, Jena, 1903.

[32] Gottlob Frege. *The Basic Laws of Arithmetic*. University of California Press, Berkeley, California, 1964. Pasrtial English translation by Montgomery Furth.

[33] Gottlob Frege. *From Frege to Gödel: A Sourcebook in Mathematical Logic, 1879-1931*, chapter A Concept Notation: A Formula Language of Pure

Thought, Modelled Upon That of Arithmetic, pages 1–82. Harvard University Press, 1967, 2002.

[34] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Kenneth A. Bowen Robert A. Kowalski, editor, *Proceedings of the Fifth International Conference on Logic Programming (ICLP)*, pages 1070–1080. MIT Press, 1988.

[35] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(2):176–210, 1934.

[36] Gerhard Gentzen. Investigations into logical deduction. *American Philosophical Quarterly*, 1(4):288–306, October 1964.

[37] Gerhard Gentzen. *The Collected Works of Gerhard Gentzen*, chapter Investigations into Logical Deduction, pages 68–131. Studies in logic and the foundations of mathematics. North-Holland, 1969.

[38] Kurt Gödel. Über die Vollständigkeit des Logikkalküls. Doctoral thesis, University of Vienna, Austria, 1930.

[39] Kurt Gödel. *The Philosophy of Bertrand Russell*, chapter Russell's Mathematical Logic. Tudor, New York, 1944.

[40] Kurt Gödel. *From Frege to Gödel: A Sourcebook in Mathematical Logic, 1879-1931*, chapter The Completeness of the Axioms of the Functional Calculus of Logic, pages 582–591. Harvard University Press, 1967, 2002.

[41] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[42] Christopher P. Higgins. On the declarative and procedural semantics of definite metalogic programs. *Journal of Logic and Computation*, 6(3):363–407, 1996.

[43] Patricia Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[44] Patricia M. Hill and John W. Lloyd. Analysis of meta-programs. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming, Workshop on Meta-Programming in Logic (META-88)*, pages 23–51. MIT Press, 1988.

[45] Roger J. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

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

[47] Y. Jiang. Ambivalent logic as the semantic basis of logic programming. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 387–401. MIT Press, 1994.

[48] M. B. Kalsbeek and Y. Jiang. *Meta-Logics and Logic Programming*, chapter A Vademecum of Ambivalent Logic, pages 27–56. MIT Press, 1995.

[49] Robert Kowalski and Jin-Sang Kim. *Artificial and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, chapter A Metalogic Approach to Multi-Agent Knowledge and Belief, pages 231–246. Academic Press, 1991.

[50] Robert Kowalsky. *Logic For Problem Solving*. North Holland, 1979.

[51] Daniel Leivant. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2: Deduction Methodologies of *Oxford Science Publications*, chapter Higher Order Logic, pages 229–321. Clarendon Press, 1994.

[52] Giorgio Levi and Davide Ramundo. A formalization of meta-programming for real. In David Scott Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 354–373. MIT Press, 1993.

[53] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Information and Computation*, 159(1-2):125–150, 2000.

[54] Godehard Link, editor. *One Hundred Years of Russell's Paradox: Mathematics, Logic, Philosophy*. de Gruyter, 2004.

[55] John W. Lloyd. *Foundation of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[56] John McCarthy and Michael I. Levin. Lisp 1.5 programmer's manual. Technical report, Computation Center, Massachusetts Institute of Technology, USA, 1965.

[57] John McCarthy, Masahiko Sato, Takeshi Hayashi, and Shigeru Igrashi. On the model theory of knowledge. Technical Report AIM-312, Stanford University, 1978.

[58] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, 1989.

[59] Dale Miller and Gopalan Nadathur. An overview of Lambda-Prolog. In *Proceedings Fifth International Conference and Symposium on Logic Programming*, pages 810–827. MIT Press, 1988.

[60] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[61] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–374, 1978.

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

[63] Richard O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[64] Lawrence Page, Sergey Brin, Rajeev Motwany, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Computer Science Department, Stanford University, 1999.

[65] Alan Jay Perlis and Klaus Samelson. Preliminary report: International algebraic language. *Communications of the ACM*, 1(12):8–22, 1958.

[66] Frank Pfenning. Logic programming in the LF logical framework. Research report, School of Computer Science, Carnegie Mellon University, 1991.

[67] Frank Pfenning and Carsten Schürmann. System description: Twelf – A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206. Springer-Verlag, 1999.

[68] Brigitte Pientka. Tabled higher-order logic programming. Technical Report CMU-CS-03-185, School of Computer Science, Carnegie Mellon University, 2003.

[69] Bertrand Russell. On some difficulties in the theory of transfinite numbers and order types. *Proceedings of the London Mathematical Society*, s2-4(1):29–53, 1907.

[70] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, July 1908.

[71] Bertrand Russell. *The Collected Papers of Bertrand Russell. Volume 6: Logical and Philosophical Papers 1909-13*. George Allen & Unwin, 1986.

[72] Sebastian Schaffert and François Bry. Querying the web reconsidered: A practical introduction to Xcerpt. In *Proceedings Extreme Markup Languages*, 2004.

[73] Danny De Schreye and Bern Martens. A sensible least Herbrand semantics for untyped vanilla meta-programming and its extension to a limited form of amalgamation. In A. Peterossi, editor, *Meta-Programming in Logic, Proceedings of the 3rd International Workshop on Meta-Programming (META)*, LNCS, pages 192–204. Springer-Verlag, 1992.

[74] Laurent Siklóssy. *Let's Talk Lisp*. Prentice-Hall, 1976.

[75] Leon S. Sterling and Ehud Y. Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.

[76] Hiroyasu Sugano. Reflective computation in logic language and its semantics. Technical report, International Institute for Avanced Study of Social and Information Science, Fujitsu Limited, 1989.

[77] Hiroyasu Sugano. Meta and reflective computation in logic programming and its semantics. In Maurice Bruynooghe, editor, *Proceedings of the 2nd Workshop on Meta-Programming in Logic Programming (META)*, pages 19–34, 1990.

[78] Donal Trump. "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`, Sun Nov 12 00:48:01 +0000 2017.

[79] Maarten van Emden and Robert Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computating Machinery*, 23:733–742, 1976.

[80] Franz von Kutschera. *Gottlob Frege: Eine Einführung in sein Werk*. de Gruyter, Berlin, 1989.