A Denotational Semantics for First-Order Logic

Krzysztof R. Apt[1,2]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, the Netherlands
[2] University of Amsterdam, the Netherlands
http://www.cwi.nl/~apt

Abstract. In Apt and Bezem [AB99] we provided a computational interpretation of first-order formulas over arbitrary interpretations. Here we complement this work by introducing a denotational semantics for first-order logic. Additionally, by allowing an assignment of a non-ground term to a variable we introduce in this framework logical variables.
The semantics combines a number of well-known ideas from the areas of semantics of imperative programming languages and logic programming. In the resulting computational view conjunction corresponds to sequential composition, disjunction to "don't know" nondeterminism, existential quantification to declaration of a local variable, and negation to the "negation as finite failure" rule. The soundness result shows correctness of the semantics with respect to the notion of truth. The proof resembles in some aspects the proof of the soundness of the SLDNF-resolution.

1 Introduction

Background

To explain properly the motivation for the work here discussed we need to go back to the roots of logic programming and constraint logic programming. *Logic programming* grew out of the seminal work of Robinson [Rob65] on the *resolution method* and the *unification method*. First, Kowalski and Kuehner [KK71] introduced a limited form of resolution, called linear resolution. Then Kowalski [Kow74] proposed what we now call *SLD-resolution*. The SLD-resolution is both a restriction and an extension of the resolution method. Namely, the clauses are restricted to Horn clauses. However, in the course of the resolution process a substitution is generated that can be viewed as a result of a computation. Right from the outset the SLD-resolution became then a crucial example of the *computation as deduction* paradigm according to which the computation process is identified with a constructive proof of a formula (a query) from a set of axioms (a program) with the computation process yielding the witness (a substitution).

This lineage of logic programming explains two of its relevant characteristics:

1. the queries and clause bodies are limited to the conjunctions of atoms,
2. the computation takes place (implicitly) over the domain of all ground terms of a given first-order language.

The restriction in item 1. was gradually lifted and through the works of Clark [Cla78] and Lloyd and Topor [LT84] one eventually arrived at the possibility of using as queries and clause bodies arbitrary first-order formulas. This general syntax is for example available in the language Gödel of Lloyd and Hill [HL94].

A way to overcome the restriction in item 2. was proposed in 1987 by Jaffar and Lassez in their influential CLP(X) scheme that led to *constraint logic programming*. In this proposal the computation takes place over an arbitrary interpretation and the queries and clause bodies can contain constraints, i.e., atomic formulas interpreted over the chosen interpretation. The unification mechanism is replaced by a more general process of constraint solving and the outcome of a computation is a sequence of constraints to which the original query reduces.

This powerful idea was embodied since then in many constraint logic programming languages, starting with the CLP($\mathcal{R}$) language of Jaffar, Michaylov, Stuckey, and Yap [JMSY92] in which linear constraints over reals were allowed, and the CHIP language of Dincbas et al. [DVS$^+$88] in which linear constraints over finite domains, combined with constraint propagation, were introduced. A theoretical framework for CHIP was provided in van Hentenryck [Van89].

This transition from logic programming to constraint logic programming introduced a new element. In the CLP(X) scheme the test for satisfiability of a sequence of constraints was needed, while a proper account of the CHIP computing process required an introduction of constraint propagation into the framework. On some interpretations these procedures can be undecidable (the satisfiability test) or computationally expensive (the "ideal" constraint propagation). This explains why in the realized implementations some approximation of the former or limited instances of the latter were chosen for.

So in both approaches the computation (i.e., the deduction) process needs to be parametrized by external procedures that for each specific interpretation have to be provided and implemented separately. In short, in both cases the computation process, while parametrized by the considered interpretation, also depends on the external procedures used. In conclusion: constraint logic programming did not provide a satisfactory answer to the question of how to lift the computation process of logic programming from the domain of all ground terms to an arbitrary interpretation without losing the property that this process is effective.

Arbitrary interpretations are important since they represent a declarative counterpart of data types. In practical situations the selected interpretations would admit sorts that would correspond to the data types chosen by the user for the application at hand, say terms, integers, reals and/or lists, each with the usual operations available. It is useful to contrast this view with the one taken in typed versions of logic programming languages. For example, in the case of the Gödel language (polymorphic) types are provided and are modeled by (polymorphic) sorts in the underlying theoretic model. However, in this model the computation still implicitly takes place over one fixed domain, that of all ground terms partitioned into sorts. This domain properly captures the built-in types but does not provide an account of user defined types. Moreover, in this approach different (i.e., not uniform) interpretation of equality for different types is needed, a feature present in the language but not accounted for in the theoretical model.

Formulas as Programs

The above considerations motivated our work on a computational interpretation of first-order formulas over arbitrary interpretations reported in Apt and Bezem [AB99]. This allowed us to view first-order formulas as executable programs. That is why we called this approach *formulas as programs*. In our approach the computation process is a search of a satisfying valuation for the formula in question. Because the problem of finding such a valuation is in general undecidable, we had to introduce the possibility of partial answers, modeled by an existence of run-time errors.

This ability to compute over arbitrary interpretations allowed us to extend the computation as deduction paradigm to arbitrary interpretations. We noted already that the SLD-resolution is both a restriction and an extension of the resolution method. In turn, the formulas as programs approach is both a restriction and an extension of the logic programming. Namely, the unification process is limited to an extremely simple form of matching involving variables and ground terms only. However, the computation process now takes place over an arbitrary structure and full-first order syntax is adopted.

The formulas as programs approach to programming has been realized in the programming language Alma-0 [ABPS98] that extends imperative programming by features that support declarative programming. In fact, the work reported in Apt and Bezem [AB99] provided logical underpinnings for a fragment of Alma-0 that does not include destructive assignment or recursive procedures and allowed us to reason about non-trivial programs written in this fragment.

Rationale for This Paper

The computational interpretation provided in Apt and Bezem [AB99] can be viewed as an operational semantics of first-order logic. The history of semantics of programming languages has taught us that to better understand the underlying principles it is beneficial to abstract from the details of the operational semantics. This view was put forward by Scott and Strachey [SS71] in their proposal of *denotational semantics* of programming languages according to which, given a programming language, the meaning of each program is a mathematical function of the meanings of its direct constituents.

The aim of this paper is to complement the work of [AB99] by providing a denotational semantics of first-order formulas. This semantics combines a number of ideas realized in the areas of (nondeterministic) imperative programming languages and the field of logic programming. It formalizes a view according to which conjunction can be seen as sequential composition, disjunction as "don't know" nondeterminism, existential quantification as declaration of a local variable, and it relates negation to the "negation as finite failure" rule.

The main result is that the denotational semantics is sound with respect to the truth definition. The proof is reminiscent in some aspects of the proof of the soundness of the SLDNF-resolution of Clarke [Cla78]. The semantics of equations allows matching involving variables and non-ground terms, a feature not present in [AB99] and in Alma-0. This facility introduces logical variables in this framework but also creates a number of difficulties in the soundness proof because bindings to local variables can now be created.

First-order logic is obviously a too limited formalism for programming. In [AB99] we discussed a number of extensions that are convenient for programming purposes, to wit sorts (i.e., types), arrays, bounded quantification and non-recursive procedures. This leads to a very expressive and easy to program in subset of Alma-0. We do not envisage any problems in incorporating these features into the denotational semantics here provided. A major problem is how to deal with recursion.

The plan of the paper is as follows. In the next section we discuss the difficulties encountered when solving arbitrary equations over algebras. Then, in Section 3 we provide a semantics of equations and in Section 4 we extend it to the case of first-order formulas interpreted over an arbitrary interpretation. The resulting semantics is denotational in style. In Section 5 we relate this semantics to the notion of truth by establishing a soundness result. In Section 6 we draw conclusions and suggest some directions for future work.

2 Solving Equations over Algebras

Consider some fixed, but arbitrary, *language of terms* L and a fixed, but arbitrary *algebra* $\mathcal{J}$ for it (sometimes called a *pre-interpretation*). A typical example is the language defining arithmetic expressions and its standard interpretation over the domain of integers.

We are interested in solving equations of the form $s = t$ over an algebra, that is, we seek an instantiation of the variables occurring in s and t that makes this equation true when interpreted over $\mathcal{J}$. By varying L and $\mathcal{J}$ we obtain a whole array of specific decision problems that sometimes can be solved efficiently, like the unification problem or the problem of solving linear equations over reals, and sometimes are undecidable, like the problem of solving Diophantine equations.

Our intention is to use equations as a means to assign values to variables. Consequently, we wish to find a natural, general, situation for which the problem of determining whether an equation $s = t$ has a solution in a given algebra is decidable, and to exhibit a "most general solution", if one exists. By using most general solutions we do not lose any specific solution.

This problem cannot be properly dealt with in full generality. Take for example the polynomial equations over integers. Then the equation $x^2 - 3x + 2 = 0$ has two solutions, $\{x/1\}$ and $\{x/2\}$, and none is "more general" than the other under any reasonable definition of a solution being more general than another.

In fact, given an arbitrary interpretation, the only case that seems to be of any use is that of comparing a variable and an arbitrary term. This brings us to equations of the form $x = t$, where x does not occur in t. Such an equation has obviously a most general solution, namely the instantiation $\{x/t\}$.

A dual problem is that of finding when an equation $s = t$ has no solution in a given algebra. Of course, non-unifiability is not a rescue here: just consider the already mentioned equation $x^2 - 3x + 2 = 0$ the sides of which do not unify.

Again, the only realistic situation seems to be when both terms are ground and their values in the considered algebra are different. This brings us to equations $s = t$ both sides of which are ground terms.

3 Semantics of Equations

After these preliminary considerations we introduce specific "hybrid" objects in which we mix the syntax and semantics.

Definition 1. *Consider a language of terms L and an algebra $\mathcal{J}$ for it. Given a function symbol f we denote by $f_{\mathcal{J}}$ the interpretation of f in $\mathcal{J}$.*

- *Consider a term of L in which we replace some of the variables by the elements of the domain D. We call the resulting object a* generalized term.
- *Given a generalized term t we define its $\mathcal{J}$-*evaluation *as follows:*
  - *replace each constant occuring in t by its value in $\mathcal{J}$,*
  - *repeatedly replace each sub-object of the form $f(d_1, \ldots, d_n)$ where f is a function symbol and $d_1, \ldots, d_n$ are the elements of the domain D by the element $f_{\mathcal{J}}(d_1, \ldots, d_n)$ of D.*

  *We call the resulting generalized term a $\mathcal{J}$-*term *and denote it by $[\![t]\!]_{\mathcal{J}}$. Note that if t is ground, then $[\![t]\!]_{\mathcal{J}}$ is an element of the domain of $\mathcal{J}$.*
- *By a $\mathcal{J}$-*substitution *we mean a finite mapping from variables to $\mathcal{J}$-terms which assigns to each variable x in its domain a $\mathcal{J}$-term different from x. We write it as $\{x_1/h_1, \ldots, x_n/h_n\}$.* □

The $\mathcal{J}$-substitutions generalize both the usual substitutions and the valuations, which assign domain values to variables. By adding to the language L constants for each domain element and for each ground term we can reduce the $\mathcal{J}$-substitutions to the substitutions. We preferred not to do this to keep the notation simple.

In what follows we denote the empty $\mathcal{J}$-substitution by ε and arbitrary $\mathcal{J}$-substitutions by θ, η, γ with possible subscripts.

A more intuitive way of introducing $\mathcal{J}$-terms is as follows. Each ground term of s of L evaluates to a unique value in $\mathcal{J}$. Given a generalized term t replace each maximal ground subterm of t by its value in $\mathcal{J}$. The outcome is the $\mathcal{J}$-term $[\![t]\!]_{\mathcal{J}}$.

We define the notion of an application of a $\mathcal{J}$-substitution θ to a generalized term t in the standard way and denote it by $t\theta$. If t is a term, then $t\theta$ does not have to be a term, though it is a generalized term.

Definition 2.

– *A* composition of two $\mathcal{J}$-substitutions θ and η, *written as* $\theta\eta$, *is defined as the unique* $\mathcal{J}$*-substitution* γ *such that for each variable* x

$$x\gamma = [\![(x\theta)\eta]\!]_{\mathcal{J}}.$$

□

Let us illustrate the introduced concepts by means of two examples.

Example 1. Take an arbitrary language of terms L. The *Herbrand algebra* Her for L is defined as follows:

– its domain is the set HU_L of all ground terms of L (usually called the *Herbrand universe*),
– if f is an n-ary function symbol in L, then its interpretation is the mapping from $(HU_L)^n$ to HU_L which maps the sequence $t_1, \ldots, t_n$ of ground terms to the ground term $f(t_1, \ldots, t_n)$.

Consider now a term s. Then $[\![s]\!]_{Her}$ equals s because in Her every ground term evaluates to itself. So the notions of a term, a generalized term and a Her-term coincide. Consequently, the notions of substitutions and Her-substitutions coincide. □

Example 2. Take as the language of terms the language AE of *arithmetic expressions*. Its binary function symbols are the usual $\cdot$ ("times"), $+$ ("plus") and $-$ ("minus"), and its unique binary symbol is $-$ ("unary minus"). Further, for each integer $\mathbf{k}$ there is a constant k.

As the algebra for AE we choose the standard algebra Int that consists of the set of integers with the function symbols interpreted in the standard way. In what follows we write the binary function symbols in the usual infix notation.

Consider the term $s \equiv x + (((3+2)\cdot 4) - y)$. Then $[\![s]\!]_{AE}$ equals $x + (\mathbf{20} - y)$. Further, given the AE-substitution $\theta := \{x/\mathbf{6} - z,\ y/\mathbf{3}\}$ we have $s\theta \equiv (\mathbf{6} - z) + (((3+2)\cdot 4) - \mathbf{3})$ and consequently, $[\![s\theta]\!]_{AE} = (\mathbf{6} - z) + \mathbf{17}$. Further, given $\eta := \{z/\mathbf{4}\}$, we have $\theta\eta = \{x/\mathbf{2},\ y/\mathbf{3},\ z/\mathbf{4}\}$. □

To define the meaning of an equation over an algebra $\mathcal{J}$ we view $\mathcal{J}$-substitutions as states and use a special state

– *error*, to indicate that it is not possible to determine effectively whether a solution to the equation $s\theta = t\theta$ in $\mathcal{J}$ exists.

We now define the semantics $[\![\cdot]\!]$ of an equation between two generalized terms as follows:

$$[\![s = t]\!](\theta) := \begin{cases} \{\theta\{s\theta/[\![t\theta]\!]_{\mathcal{J}}\}\} & \text{if } s\theta \text{ is a variable that does not occur in } t\theta, \\ \{\theta\{t\theta/[\![s\theta]\!]_{\mathcal{J}}\}\} & \text{if } t\theta \text{ is a variable that does not occur in } s\theta \\ & \text{and } s\theta \text{ is not a variable,} \\ \{\theta\} & \text{if } [\![s\theta]\!]_{\mathcal{J}} \text{ and } [\![t\theta]\!]_{\mathcal{J}} \text{ are identical,} \\ \emptyset & \text{if } s\theta \text{ and } t\theta \text{ are ground and } [\![s\theta]\!]_{\mathcal{J}} \neq [\![t\theta]\!]_{\mathcal{J}}, \\ \{error\} & \text{otherwise.} \end{cases}$$

It will become clear in the next section why we collect here the unique outcome into a set and why we "carry" θ in the answers.

Note that according to the above definition we have $[\![s = t]\!](\theta) = \{error\}$ for the non-ground generalized terms $s\theta$ and $t\theta$ such that the $\mathcal{J}$-terms $[\![s\theta]\!]_{\mathcal{J}}$ and $[\![t\theta]\!]_{\mathcal{J}}$ are different. In some situations we could safely assert then that $[\![s = t]\!](\theta) = \{\theta\}$ or that $[\![s = t]\!](\theta) = \emptyset$. For example, for the standard algebra Int for the language of arithmetic expressions we could safely assert that $[\![x + x = 2 \cdot x]\!](\theta) = \{\theta\}$ and $[\![x + 1 = x]\!](\theta) = \emptyset$ for any AE-substitution θ.

The reason we did not do this was that we wanted to ensure that the semantics is uniform and decidable so that it can be implemented.

4 A Denotational Semantics for First-Order Logic

Consider now a first-order language with equality $\mathcal{L}$. In this section we extend the semantics $[\![\cdot]\!]$ to arbitrary first-order formulas from $\mathcal{L}$ interpreted over an arbitrary interpretation. $[\![\cdot]\!]$ depends on the considered interpretation but to keep the notation simple we do not indicate this dependence. This semantics is denotational in the sense that meaning of each formula is a mathematical function of the meanings of its direct constituents.

Fix an interpretation $\mathcal{I}$. $\mathcal{I}$ is based on some algebra $\mathcal{J}$. We define the notion of an application of a $\mathcal{J}$-substitution θ to a formula ϕ of $\mathcal{L}$, written as $\phi\theta$, in the usual way.

Consider an atomic formula $p(t_1, \ldots, t_n)$ and a $\mathcal{J}$-substitution θ. We denote by $p_{\mathcal{I}}$ the interpretation of p in $\mathcal{I}$.

We say that

- $p(t_1, \ldots, t_n)\theta$ is *true* if $p(t_1, \ldots, t_n)\theta$ is ground and $([\![t_1\theta]\!]_{\mathcal{J}}, \ldots, [\![t_n\theta]\!]_{\mathcal{J}}) \in p_{\mathcal{I}}$,
- $p(t_1, \ldots, t_n)\theta$ is *false* if $p(t_1, \ldots, t_n)\theta$ is ground and $([\![t_1\theta]\!]_{\mathcal{J}}, \ldots, [\![t_n\theta]\!]_{\mathcal{J}}) \notin p_{\mathcal{I}}$.

In what follows we denote by $Subs$ the set of $\mathcal{J}$-substitutions and by $\mathcal{P}(A)$, for a set A, the set of all subsets of A.

For a given formula ϕ its semantics $[\![\phi]\!]$ is a mapping

$$[\![\phi]\!] : Subs \to \mathcal{P}(Subs \cup \{error\}).$$

The fact that the outcome of $[\![\phi]\!](\theta)$ is a set reflects the possibility of a nondeterminism here modeled by the disjunction.

To simplify the definition we extend $[\![\cdot]\!]$ to deal with subsets of $Subs \cup \{error\}$ by putting

$$[\![\phi]\!](error) := \{error\},$$

and for a set $X \subseteq Subs \cup \{error\}$

$$[\![\phi]\!](X) := \bigcup_{e \in X} [\![\phi]\!](e).$$

Further, to deal with the existential quantifier, we introduce an operation $DROP_x$, where x is a variable. First we define $DROP_x$ on the elements of $Subs \cup \{error\}$ by putting for a $\mathcal{J}$-substitution θ

$$DROP_x(\theta) := \begin{cases} \theta \text{ if } x \text{ is not in the domain of } \theta, \\ \eta \text{ if } \theta \text{ is of the form } \eta \uplus \{x/s\}, \end{cases}$$

and

$$DROP_x(error) := error.$$

Then we extend it element-wise to subsets of $Subs \cup \{error\}$, that is, by putting for a set $X \subseteq Subs \cup \{error\}$

$$DROP_x(X) := \{DROP_x(e) \mid e \in X\}.$$

$[\![\cdot]\!]$ is defined by structural induction as follows, where A is an atomic formula different from $s = t$:

- $[\![A]\!](\theta) := \begin{cases} \{\theta\} & \text{if } A\theta \text{ is true,} \\ \emptyset & \text{if } A\theta \text{ is false,} \\ \{error\} & \text{otherwise, that is if } A\theta \text{ is not ground,} \end{cases}$
- $[\![\phi_1 \wedge \phi_2]\!](\theta) := [\![\phi_2]\!]([\![\phi_1]\!](\theta))$,
- $[\![\phi_1 \vee \phi_2]\!](\theta) := [\![\phi_1]\!](\theta) \cup [\![\phi_2]\!](\theta)$,
- $[\![\neg\phi]\!](\theta) := \begin{cases} \{\theta\} & \text{if } [\![\phi]\!](\theta) = \emptyset, \\ \emptyset & \text{if } \theta \in [\![\phi]\!](\theta), \\ \{error\} & \text{otherwise,} \end{cases}$
- $[\![\exists x\, \phi]\!](\theta) := DROP_y([\![\phi\{x/y\}]\!](\theta))$, where y is a fresh variable.

To better understand this definition let us consider some simple examples that refer to the algebras discussed in Examples 1 and 2.

Example 3. Take an interpretation $\mathcal{I}$ based on the Herbrand algebra Her. Then

$$[\![f(x) = z \wedge g(z) = g(f(x))]\!](\{x/g(y)\}) = [\![g(z) = g(f(x))]\!](\theta) = \{\theta\},$$

where $\theta := \{x/g(y), z/f(g(y))\}$. On the other hand

$$[\![g(f(x)) = g(z)]\!](\{x/g(y)\}) = \{error\}.$$

□

Example 4. Take an interpretation $\mathcal{I}$ based on the standard algebra AE for the language of arithmetic expressions. Then

$$[\![y = z - 1 \wedge z = x + 2]\!](\{x/\mathbf{1}\}) = [\![z = x + 2]\!](\{x/\mathbf{1}, y/z - \mathbf{1}\}) = \{x/\mathbf{1}, y/\mathbf{2}, z/\mathbf{3}\}.$$

Further,

$$[\![y + 1 = z - 1]\!](\{y/\mathbf{1}, z/\mathbf{3}\}) = \{y/\mathbf{1}, z/\mathbf{3}\}$$

and even

$$[\![x \cdot (y+1) = (v+1) \cdot (z-1)]\!](\{x/v+\mathbf{1},\ y/\mathbf{1},\ z/\mathbf{3}\}) = \{x/v+\mathbf{1},\ y/\mathbf{1},\ z/\mathbf{3}\}.$$

On the other hand

$$[\![y-1 = z-1]\!](\varepsilon) = \{error\}.$$

□

The first example shows that the semantics given here is weaker than the one provided by the logic programming. In turn, the second example shows that our treatment of arithmetic expressions is more general than the one provided by Prolog.

This definition of denotational semantics of first-order formulas combines a number of ideas put forward in the area of semantics of imperative programming languages and the field of logic programming.

First, for an atomic formula A, when $A\theta$ is ground, its meaning coincides with the meaning of a Boolean expression given in de Bakker [dB80, page 270]. In turn, the meaning of the conjunction and of the disjunction follows [dB80, page 270] in the sense that the conjunction corresponds to the sequential composition operation ";" and the disjunction corresponds to the "don't know" nondeterministic choice, denoted there by $\cup$.

Next, the meaning of the negation is inspired by its treatment in logic programming. To be more precise we need the following observations the proofs of which easily follow by structural induction.

Note 1.

(i) If $\eta \in [\![\phi]\!](\theta)$, then $\eta = \theta\gamma$ for some $\mathcal{J}$-substitution γ.
(ii) If $\phi\theta$ is ground, then $[\![\phi]\!](\theta) \subseteq \{\theta\}$. □

First, we interpret $[\![\phi]\!](\theta) \cap Subs \neq \emptyset$ as the statement "the query $\phi\theta$ succeeds". More specifically, if $\eta \in [\![\phi]\!](\theta)$, then by Note 1(i) for some γ we have $\eta = \theta\gamma$.

In general, γ is of course not unique: take for example $\theta := \{x/0\}$ and $\eta = \theta$. Then both $\eta = \theta\varepsilon$ and $\eta = \theta\theta$. However, it is easy to show that if η is less general than θ, then in the set $\{\gamma \mid \eta = \theta\gamma\}$ the $\mathcal{J}$-substitution with the smallest domain is uniquely defined. In what follows given $\mathcal{J}$-substitutions η and θ such that η is less general than θ, when writing $\eta = \theta\gamma$ we always refer to this uniquely defined γ.

Now we interpret $\theta\gamma \in [\![\phi]\!](\theta)$ as the statement "γ is the computed answer substitution for the query $\phi\theta$". In turn, we interpret $[\![\phi]\!](\theta) = \emptyset$ as the statement "the query $\phi\theta$ finitely fails".

Suppose now that $[\![\phi]\!](\theta) \cap Subs \neq \emptyset$, which means that the query $\phi\theta$ succeeds. Assume additionally that $\phi\theta$ is ground. Then by Note 1(ii) $\theta \in [\![\phi]\!](\theta)$ and consequently by the definition of the meaning of negation $[\![\neg\phi]\!](\theta) = \emptyset$, which means that the query $\neg\phi\theta$ finitely fails.

In turn, suppose that $[\![\phi]\!](\theta) = \emptyset$, which means that the query $\phi\theta$ finitely fails. By the definition of the meaning of negation $[\![\neg\phi]\!](\theta) = \{\theta\}$, which means that the query $\neg\phi\theta$ succeeds with the empty computed answer substitution.

This explains the relation with the "negation as finite failure" rule according to which for a ground query Q:

- if Q succeeds, then $\neg Q$ finitely fails,
- if Q finitely fails, then $\neg Q$ succeeds with the empty computed answer substitution.

In fact, our definition of the meaning of negation corresponds to a generalization of the negation as finite failure rule already mentioned in Clark [Cla78], according to which the requirement that Q is ground is dropped and the first item is replaced by:

- if Q succeeds with the empty computed answer substitution, then $\neg Q$ finitely fails.

Finally, the meaning of the existential quantification corresponds to the meaning of the block statement in imperative languages, see, e.g., de Bakker [dB80, page 226], with the important difference that the local variable is not initialized. From this viewpoint the existential quantifier $\exists x$ corresponds to the declaration of the local variable x. The $DROP_x$ operation was introduced in Clarke [Cla79] to deal with the declarations of local variables.

We do not want to make the meaning of the formula $\exists x\ \phi$ dependent on the choice of y. Therefore we postulate that for *any* fresh variable y the set $DROP_y([\![\phi\{x/y\}]\!](\theta))$ is a meaning of $\exists x\ \phi$ given a $\mathcal{J}$-substitution θ. Consequently, the semantics of $\exists x\ \phi$ has many outcomes, one for each choice of y. This "multiplicity" of meanings then extends to all formulas containing the existential quantifier. So for example for any variable y different from x and z the $\mathcal{J}$-substitution $\{z/f(y)\}$ is the meaning of $\exists x\ (z = f(x))$ given the empty $\mathcal{J}$-substitution ε.

5 Soundness

To relate the introduced semantics to the notion of truth we first formalize the latter using the notion of a $\mathcal{J}$-substitution instead of the customary notion of a valuation.

Consider a first-order language $\mathcal{L}$ with equality and an interpretation $\mathcal{I}$ for it based on some algebra $\mathcal{J}$. Let θ be a $\mathcal{J}$-substitution. We define the relation $\mathcal{I} \models_\theta \phi$ for a formula ϕ by structural induction. First we assume that θ is defined on all free variables of ϕ and put

- $\mathcal{I} \models_\theta s = t$ iff $[\![s\theta]\!]_{\mathcal{J}}$ and $[\![t\theta]\!]_{\mathcal{J}}$ coincide,
- $\mathcal{I} \models_\theta p(t_1, \ldots, t_n)$ iff $p(t_1, \ldots, t_n)\theta$ is ground and $([\![t_1\theta]\!]_{\mathcal{J}}, \ldots, [\![t_n\theta]\!]_{\mathcal{J}}) \in p_{\mathcal{I}}$.

In other words, $\mathcal{I} \models_\theta p(t_1, \ldots, t_n)$ iff $p(t_1, \ldots, t_n)\theta$ is true. The definition extends to non-atomic formulas in the standard way.

Now assume that θ is not defined on all free variables of ϕ. We put

– $\mathcal{I} \models_\theta \phi$ iff $\mathcal{I} \models_\theta \forall x_1, \ldots, \forall x_n \phi$ where $x_1, \ldots, x_n$ is the list of the free variables of ϕ that do not occur in the domain of θ.

Finally,

– $\mathcal{I} \models \phi$ iff $\mathcal{I} \models_\theta \phi$ for all $\mathcal{J}$-substitutions θ.

To prove the main theorem we need the following notation. Given a $\mathcal{J}$-substitution $\eta := \{x_1/h_1, \ldots, x_n/h_n\}$ we define $\langle\eta\rangle := x_1 = h_1 \wedge \ldots \wedge x_n = h_n$.

In the discussion that follows the following simple observation will be useful.

Note 2. For all $\mathcal{J}$-substitutions θ and formulas ϕ

$$\mathcal{I} \models_\theta \phi \text{ iff } \mathcal{I} \models \langle\theta\rangle \rightarrow \phi.$$

□

The following theorem now shows correctness of the introduced semantics with respect to the notion of truth.

Theorem 1 (Soundness).

Consider a first-order language $\mathcal{L}$ with equality and an interpretation $\mathcal{I}$ for it based on some algebra $\mathcal{J}$. Let ϕ be a formula of $\mathcal{L}$ and θ a $\mathcal{J}$-substitution.

(i) For each $\mathcal{J}$-substitution $\eta \in [\![\phi]\!](\theta)$

$$\mathcal{I} \models_\eta \phi.$$

(ii) If error $\notin [\![\phi]\!](\theta)$, then

$$\mathcal{I} \models \phi\theta \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{y}_i \langle\eta_i\rangle,$$

where $[\![\phi]\!](\theta) = \{\theta\eta_1, \ldots, \theta\eta_k\}$, and for $i \in [1..k]$ $\mathbf{y}_i$ is a sequence of variables that appear in the range of η_i.

Note that by (ii) if $[\![\phi]\!](\theta) = \emptyset$, then

$$\mathcal{I} \models_\theta \neg\phi.$$

In particular, if $[\![\phi]\!](\varepsilon) = \emptyset$, then

$$\mathcal{I} \models \neg\phi.$$

Proof. The proof proceeds by simultaneous induction on the structure of the formulas.

ϕ is $s = t$.

If $\eta \in [\![\phi]\!](\theta)$, then three possibilities arise.

1. $s\theta$ is a variable that does not occur in $t\theta$.

Then $[\![s = t]\!](\theta) = \{\theta\{s\theta/[\![t\theta]\!]_{\mathcal{J}}\}\}$ and consequently $\eta = \theta\{s\theta/[\![t\theta]\!]_{\mathcal{J}}\}$. So $\mathcal{I} \models_\eta (s = t)$ holds since $s\eta = [\![t\theta]\!]_{\mathcal{J}}$ and $t\eta = t\theta$.

2. $t\theta$ is a variable that does not occur in $s\theta$ and $s\theta$ is not a variable.

Then $[\![s = t]\!](\theta) = \{\theta\{t\theta/[\![s\theta]\!]_{\mathcal{J}}\}\}$. This case is symmetric to 1.

3. $[\![s\theta]\!]_{\mathcal{J}}$ and $[\![t\theta]\!]_{\mathcal{J}}$ are identical.

Then $\eta = \theta$, so $\mathcal{I} \models_\eta (s = t)$ holds.

If $error \notin [\![\phi]\!](\theta)$, then four possibilities arise.

1. $s\theta$ is a variable that does not occur in $t\theta$.

Then $[\![s = t]\!](\theta) = \{\theta\{s\theta/[\![t\theta]\!]_{\mathcal{J}}\}\}$. We have $\mathcal{I} \models (s = t)\theta \leftrightarrow s\theta = [\![t\theta]\!]_{\mathcal{J}}$.

2. $t\theta$ is a variable that does not occur in $s\theta$ and $s\theta$ is not a variable.

Then $[\![s = t]\!](\theta) = \{\theta\{t\theta/[\![s\theta]\!]_{\mathcal{J}}\}\}$. This case is symmetric to 1.

3. $[\![s\theta]\!]_{\mathcal{J}}$ and $[\![t\theta]\!]_{\mathcal{J}}$ are identical.

Then $[\![s = t]\!](\theta) = \{\theta\}$. We have $[\![s = t]\!](\theta) = \{\theta\varepsilon\}$ and $\mathcal{I} \models_\theta s = t$, so $\mathcal{I} \models (s = t)\theta \leftrightarrow \langle\varepsilon\rangle$, since $\langle\varepsilon\rangle$ is vacuously true.

4. $s\theta$ and $t\theta$ are ground $\mathcal{J}$-terms and $[\![s\theta]\!]_{\mathcal{J}} \neq [\![t\theta]\!]_{\mathcal{J}}$.

Then $[\![s = t]\!](\theta) = \emptyset$ and $\mathcal{I} \models_\theta \neg(s = t)$, so $\mathcal{I} \models (s = t)\theta \leftrightarrow falsum$, where $falsum$ denotes the empty disjunction.

ϕ is an atomic formula different from $s = t$.

If $\eta \in [\![\phi]\!](\theta)$, then $\eta = \theta$ and $\phi\theta$ is true. So $\mathcal{I} \models_\theta \phi$, i.e., $\mathcal{I} \models_\eta \phi$.

If $error \notin [\![\phi]\!](\theta)$, then either $[\![\phi]\!](\theta) = \{\theta\}$ or $[\![\phi]\!](\theta) = \emptyset$. In both cases the argument is the same as in case 3. and 4. for the equality $s = t$.

Note that in both cases we established a stronger form of (ii) in which each list $\mathbf{y}_i$ is empty, i.e., no quantification over the variables in $\mathbf{y}_i$ appears.

ϕ is $\phi_1 \wedge \phi_2$. This is the most elaborate case.

If $\eta \in [\![\phi]\!](\theta)$, then for some $\mathcal{J}$-substitution γ both $\gamma \in [\![\phi_1]\!](\theta)$ and $\eta \in [\![\phi_2]\!](\gamma)$. By induction hypothesis both $\mathcal{I} \models_\gamma \phi_1$ and $\mathcal{I} \models_\eta \phi_2$. But by Note 1(i) η is less general than γ, so $\mathcal{I} \models_\eta \phi_1$ and consequently $\mathcal{I} \models_\eta \phi_1 \wedge \phi_2$.

If $error \notin [\![\phi]\!](\theta)$, then for some $X \subseteq Subs$ both $[\![\phi_1]\!](\theta) = X$ and $error \notin [\![\phi_2]\!](\eta)$ for all $\eta \in X$.

By induction hypothesis

$$\mathcal{I} \models \phi_1\theta \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{y}_i \langle \eta_i \rangle,$$

where $X = \{\theta\eta_1, \ldots, \theta\eta_k\}$ and for $i \in [1..k]$ $\mathbf{y}_i$ is a sequence of variables that appear in the range of η_i. Hence

$$\mathcal{I} \models (\phi_1 \wedge \phi_2)\theta \leftrightarrow \bigvee_{i=1}^{k} (\exists \mathbf{y}_i \langle \eta_i \rangle \wedge \phi_2\theta),$$

so by appropriate renaming of the variables in the sequences $\mathbf{y}_i$

$$\mathcal{I} \models (\phi_1 \wedge \phi_2)\theta \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{y}_i (\langle \eta_i \rangle \wedge \phi_2\theta).$$

But for any $\mathcal{J}$-substitution δ and a formula ψ

$$\mathcal{I} \models \langle \delta \rangle \wedge \psi \leftrightarrow \langle \delta \rangle \wedge \psi\delta,$$

so

$$\mathcal{I} \models (\phi_1 \wedge \phi_2)\theta \leftrightarrow (\bigvee_{i=1}^{k} \exists \mathbf{y}_i (\langle \eta_i \rangle \wedge \phi_2\theta\eta_i). \tag{1}$$

Further, we have for $i \in [1..k]$

$$[\![\phi_2]\!](\theta\eta_i) = \{\theta\eta_i\gamma_{i,j} \mid j \in [1..\ell_i]\}$$

for some $\mathcal{J}$-substitutions $\gamma_{i,1}, \ldots, \gamma_{i,\ell_i}$. So

$$[\![\phi_1 \wedge \phi_2]\!](\theta) = \{\theta\eta_i\gamma_{i,j} \mid i \in [1..k], j \in [1..\ell_i]\}.$$

By induction hypothesis we have for $i \in [1..k]$

$$\mathcal{I} \models \phi_2\theta\eta_i \leftrightarrow \bigvee_{j=1}^{\ell_i} \exists \mathbf{v}_{i,j} \langle \gamma_{i,j} \rangle,$$

where for $i \in [1..k]$ and $j \in [1..\ell_i]$ $\mathbf{v}_{i,j}$ is a sequence of variables that appear in the range of $\gamma_{i,j}$.

Using (1) by appropriate renaming of the variables in the sequences $\mathbf{v}_{i,j}$ we now conclude that

$$\mathcal{I} \models (\phi_1 \wedge \phi_2)\theta \leftrightarrow \bigvee_{i=1}^{k} \bigvee_{j=1}^{\ell_i} \exists \mathbf{y}_i \exists \mathbf{v}_{i,j} (\langle \eta_i \rangle \wedge \langle \gamma_{i,j} \rangle),$$

so

$$\mathcal{I} \models (\phi_1 \wedge \phi_2)\theta \leftrightarrow \bigvee_{i=1}^{k} \bigvee_{j=1}^{\ell_i} \exists \mathbf{y}_i \exists \mathbf{v}_{i,j} \langle \eta_i\gamma_{i,j} \rangle,$$

since the domains of η_i and $\gamma_{i,j}$ are disjoint and for any $\mathcal{J}$-substitutions γ and δ with disjoint domains we have

$$\mathcal{I} \models \langle \gamma \rangle \wedge \langle \delta \rangle \leftrightarrow \langle \gamma\delta \rangle.$$

ϕ is $\phi_1 \lor \phi_2$.

If $\eta \in [\![\phi]\!](\theta)$, then either $\eta \in [\![\phi_1]\!](\theta)$ or $\eta \in [\![\phi_2]\!](\theta)$, so by induction hypothesis either $\mathcal{I} \models_\eta \phi_1$ or $\mathcal{I} \models_\eta \phi_2$. In both cases $\mathcal{I} \models_\eta \phi_1 \lor \phi_2$ holds.

If $error \notin [\![\phi]\!](\theta)$, then for some $\mathcal{J}$-substitutions $\eta_1, \ldots, \eta_k$

$$[\![\phi_1]\!](\theta) = \{\theta\eta_1, \ldots, \theta\eta_k\},$$

where $k \geq 0$, for some $\mathcal{J}$-substitutions $\eta_{k+1}, \ldots, \eta_{k+\ell}$,

$$[\![\phi_2]\!](\theta) = \{\theta\eta_{k+1}, \ldots, \theta\eta_{k+\ell}\},$$

where $\ell \geq 0$, and

$$[\![\phi_1 \lor \phi_2]\!](\theta) = \{\theta\eta_1, \ldots, \theta\eta_{k+\ell}\}.$$

By induction hypothesis both

$$\mathcal{I} \models \phi_1\theta \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{y}_i \langle \eta_i \rangle$$

and

$$\mathcal{I} \models \phi_2\theta \leftrightarrow \bigvee_{i=k+1}^{k+\ell} \exists \mathbf{y}_i \langle \eta_i \rangle$$

for appropriate sequences of variables $\mathbf{y}_i$. So

$$\mathcal{I} \models (\phi_1 \lor \phi_2)\theta \leftrightarrow \bigvee_{i=1}^{k+\ell} \exists \mathbf{y}_i \langle \eta_i \rangle.$$

ϕ is $\neg\phi_1$.

If $\eta \in [\![\phi]\!](\theta)$, then $\eta = \theta$ and $[\![\phi_1]\!](\theta) = \emptyset$. By induction hypothesis $\mathcal{I} \models_\theta \neg\phi_1$, i.e., $\mathcal{I} \models_\eta \neg\phi_1$.

If $error \notin [\![\phi]\!](\theta)$, then either $[\![\phi]\!](\theta) = \{\theta\}$ or $[\![\phi]\!](\theta) = \emptyset$. In the former case $[\![\phi]\!](\theta) = \{\theta\varepsilon\}$, so $[\![\phi_1]\!](\theta) = \emptyset$. By induction hypothesis $\mathcal{I} \models_\theta \neg\phi_1$, i.e., $\mathcal{I} \models (\neg\phi_1)\theta \leftrightarrow \langle\varepsilon\rangle$, since $\langle\varepsilon\rangle$ is vacuously true. In the latter case $\theta \in [\![\phi_1]\!](\theta)$, so by induction hypothesis $\mathcal{I} \models_\theta \phi_1$, i.e., $\mathcal{I} \models (\neg\phi_1)\theta \leftrightarrow falsum$.

ϕ is $\exists x\ \phi_1$.

If $\eta \in [\![\phi]\!](\theta)$, then $\eta \in DROP_y([\![\phi_1\{x/y\}]\!](\theta))$ for some fresh variable y. So either (if y is not in the domain of η) $\eta \in [\![\phi_1\{x/y\}]\!](\theta)$ or for some $\mathcal{J}$-term s we have $\eta \uplus \{y/s\} \in [\![\phi_1\{x/y\}]\!](\theta)$. By induction hypothesis in the former case $\mathcal{I} \models_\eta \phi_1\{x/y\}$ and in the latter case $\mathcal{I} \models_{\eta\uplus\{y/s\}} \phi_1\{x/y\}$. In both cases $\mathcal{I} \models \exists y\ (\phi_1\{x/y\}\eta)$, so, since y is fresh, $\mathcal{I} \models (\exists y\ \phi_1\{x/y\})\eta$ and consequently $\mathcal{I} \models (\exists x\ \phi_1)\eta$, i.e., $\mathcal{I} \models_\eta \exists x\ \phi_1$.

If $error \notin [\![\phi]\!](\theta)$, then $error \notin [\![\phi_1\{x/y\}]\!](\theta)$, as well, where y is a fresh variable. By induction hypothesis

$$\mathcal{I} \models \phi_1\{x/y\}\theta \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{y}_i \langle \eta_i \rangle, \tag{2}$$

where

$$[\![\phi_1\{x/y\}]\!](\theta) = \{\theta\eta_1, \ldots, \theta\eta_k\} \tag{3}$$

and for $i \in [1..k]$ $\mathbf{y}_i$ is a sequence of variables that appear in the range of η_i.

Since y is fresh, we have $\mathcal{I} \models \exists y\ (\phi_1\{x/y\}\theta) \leftrightarrow (\exists y\ \phi_1\{x/y\})\theta$ and $\mathcal{I} \models (\exists y\ \phi_1\{x/y\})\theta \leftrightarrow (\exists x\ \phi_1)\theta$. So (2) implies

$$\mathcal{I} \models (\exists x\ \phi_1)\theta \leftrightarrow \bigvee_{i=1}^{k} \exists y \exists \mathbf{y}_i \langle \eta_i \rangle.$$

But for $i \in [1..k]$

$$\mathcal{I} \models \exists y \langle \eta_i \rangle \leftrightarrow \exists y \langle DROP_y(\eta_i) \rangle,$$

since if $y/s \in \eta_i$, then the variable y does not appear in s. So

$$\mathcal{I} \models (\exists x\ \phi_1)\theta \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{y}_i \exists y \langle DROP_y(\eta_i) \rangle. \tag{4}$$

Now, by (3)

$$[\![\exists x\ \phi_1]\!](\theta) = \{DROP_y(\theta\eta_1), \ldots, DROP_y(\theta\eta_k)\}.$$

But y does not occur in θ, so we have for $i \in [1..k]$

$$DROP_y(\theta\eta_i) = \theta DROP_y(\eta_i)$$

and consequently

$$[\![\exists x\ \phi_1]\!](\theta) = \{\theta DROP_y(\eta_1), \ldots, \theta DROP_y(\eta_k)\}.$$

This by virtue of (4) concludes the proof. □

Informally, (i) states that every computed answer substitution of $\phi\theta$ validates it. It is useful to point out that (ii) is a counterpart of Theorem 3 in Clark [Cla78]. Intuitively, it states that a query is equivalent to the disjunction of its computed answer substitutions written out in an equational form (using the $\langle \eta \rangle$ notation). In our case this property holds only if *error* is not a possible outcome. Indeed, if $[\![s = t]\!](\theta) = \{error\}$, then nothing can be stated about the status of the statement $\mathcal{I} \models (s = t)\theta$.

Note that in case $error \notin [\![\phi]\!](\theta)$, (ii) implies (i) by virtue of Note 2. On the other hand, if $error \in [\![\phi]\!](\theta)$, then (i) can still be applicable while (ii) not.

Additionally existential quantifiers have to be used in an appropriate way. The formulas of the form $\exists \mathbf{y} \langle \eta \rangle$ also appear in Maher [Mah88] in connection with a study of the decision procedures for the algebras of trees. In fact, there are some interesting connections between this paper and ours that could be investigated in a closer detail.

6 Conclusions and Future Work

In this paper we provided a denotational semantics to first-order logic formulas. This semantics is a counterpart of the operational semantics introduced in Apt and Bezem [AB99]. The important difference is that we provide here a more general treatment of equality according to which a non-ground term can be assigned to a variable. This realizes logical variables in the framework of Apt and Bezem [AB99]. This feature led to a number of complications in the proof of the Soundness Theorem 1.

One of the advantages of this theorem is that it allows us to reason about the considered program simply by comparing it to the formula representing its specification. In the case of operational semantics this was exemplified in Apt and Bezem [AB99] by showing how to verify non-trivial Alma-0 programs that do not include destructive assignment.

Note that it is straightforward to extend the semantics here provided to other well-known programming constructs, such as destructive assignment, **while** construct and recursion. However, as soon as a destructive assignment is introduced, the relation with the definition of truth in the sense of Soundness Theorem 1 is lost and the just mentioned approach to program verification cannot be anymore applied. In fact, the right approach to the verification of the resulting programs is an appropriately designed Hoare's logic or the weakest precondition semantics.

The work here reported can be extended in several directions. First of all, it would be useful to prove equivalence between the operational and denotational semantics. Also, it would interesting to specialize the introduced semantics to specific interpretations for which the semantics could generate less often an error. Examples are Herbrand interpretations for an arbitrary first-order language in which the meaning of equalities could be rendered using most general unifiers, and the standard interpretation over reals for the language defining linear equations; these equations can be handled by means of the usual elimination procedure. In both cases the equality could be dealt with without introducing the *error* state at all.

Other possible research directions were already mentioned in Apt and Bezem [AB99]. These involved addition of recursive procedures, of constraints, and provision of a support for automated verification of programs written in Alma-0. The last item there mentioned, relation to dynamic predicate logic, was in the meantime extensively studied in the work of van Eijck [vE98] who, starting with Apt and Bezem [AB99], defined a number of semantics for dynamic predicate logic in which the existential quantifier has a different, dynamic scope. This work was motivated by applications in natural language processing.

Acknowledgments

Many thanks to Marc Bezem for helpful discussions on the subject of this paper.

References

[AB99] K. R. Apt and M. A. Bezem. Formulas as programs. In K.R. Apt, V.W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: A 25 Year Perspective*, pages 75–107, 1999. Available via `http://xxx.lanl.gov/archive/cs/`.

[ABPS98] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM Toplas*, 20(5):1014–1066, 1998.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[Cla79] E. M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. of the ACM*, 26(1):129–147, January 1979.

[dB80] J. W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall International, Englewood Cliffs, N.J., 1980.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *FGCS-88: Proceedings International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, December 1988. ICOT.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[JMSY92] J. Jaffar, S. Michayov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[KK71] R.A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP'74*, pages 569–574. North-Holland, 1974.

[LT84] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1:225–240, 1984.

[Mah88] M.J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 348–357. The MIT Press, 1988.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[SS71] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG–6, Programming Research Group, University of Oxford, 1971.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[vE98] J. van Eijck. Programming with dynamic predicate logic. Technical Report INS-R9810, CWI, Amsterdam, 1998.