Formulas as Programs

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dept. of Mathematics, Computer Science, Physics & Astronomy
University of Amsterdam, The Netherlands
`http://www.cwi.nl/~apt`

Marc Bezem
CWI
and
Dept. of Philosophy, Utrecht University
P.O. Box 80126, 3508 TC Utrecht, The Netherlands
`bezem@phil.uu.nl`

ABSTRACT

We provide here a computational interpretation of first-order logic based on a constructive interpretation of satisfiability w.r.t. a fixed but arbitrary interpretation. In this approach the *formulas* themselves are *programs*. This contrasts with the so-called *formulas as types* approach in which the proofs of the formulas are typed terms that can be taken as programs. This view of computing is inspired by logic programming and constraint logic programming but differs from them in a number of crucial aspects.

Formulas as programs is argued to yield a realistic approach to programming that has been realized in the implemented programming language Alma-0 Apt, Brunekreef, Partington & Schaerf (1998) that combines the advantages of imperative and logic programming. The work here reported can also be used to reason about the correctness of non-recursive Alma-0 programs that do not include destructive assignment.

1991 Mathematics Subject Classification: 68N05, 68N17, 68Q60
1991 Computing Reviews Classification System: F.3.1, F.4.1
Keywords and Phrases: declarative programming, first-order logic, logic programming.
Note: This paper will appear in *The Logic Programming Paradigm: a 25 Years Perspective*, K.R. Apt, V. Marek, M. Truszczynski and D.S. Warren (eds) Springer-Verlag, *Artificial Intelligence Series.* Work carried out under project PNA1.2, CIP.

1. Introduction

1.1 Logic Programming and Program Verification

The logic programming paradigm in its original form (see Kowalski (1974)) is based on a computational interpretation of a subset of first-order logic that consists of Horn clauses. The proof theory and semantics for this subset has been well understood for some time already (see, e.g. Lloyd (1987)).

However, the practice has quickly shown that this subset is too limited for the programming purposes, so it was extended in a number of ways, notably by allowing negation. This led to a long and still inconclusive quest for extending the appropriate soundness and completeness results to logic programs that allow negation (see, e.g. Apt & Bol (1994)). To complicate the matters further, Prolog extends logic programming with negation by several features that are very operational in nature.

Constraint logic programming (see, e.g. Jaffar & Lassez (1987)) overcomes some of Prolog's deficiencies, notably its clumsy handling of arithmetic, by extending the computing process from the (implicit) domain of terms to arbitrary structures.

Logic programming and constraint logic programming are two instances of declarative programming. According to declarative programming a program has a dual reading as a formula in a logic with a simple semantics.

One of the important advantages of declarative programming is that, thanks to the semantic interpretation, programs are easier to understand, modify and verify. In fact, the dual reading of a declarative program as a formula allows us to reason about its correctness by restricting our attention to a logical analysis of the corresponding formula. For each logical formalism such an analysis essentially boils down to the question whether the formula corresponding to the program is in an appropriate sense equivalent to the specification.[1]

However, in our opinion, we do not have at our disposal *simple* and *intuitive* methods that could be used to verify in a rigorous way realistic "pure" Prolog programs (i.e. those that are also logic programs) or constraint logic programs.

We believe that one of the reasons for this state of affairs is recursion, on which both logic programming and constraint logic programming rely. In fact, recursion is often less natural than iteration, which is a more basic concept. Further, recursion in combination with negation can naturally lead to programs that are not easily amenable to a formal analysis. Finally, recursion always introduces a possibility of divergence which explains why the study of termination is such an important topic in the case of logic programming (see, e.g., De Schreye & Decorte (1994)).

1.2 First-order Logic as a Computing Mechanism

Obviously, without recursion logic programming and constraint logic programming are hopelessly inexpressive. However, as we show in this paper, it is still possible to construct a simple and realistic approach to declarative programming that draws on the ideas of these two formalisms and in which recursion is absent. This is done by providing a constructive interpretation of satisfiability of first-order formulas w.r.t. to a fixed but arbitrary interpretation. Iteration is realized by means of bounded quantification that is guaranteed to terminate.

More precisely, assuming a first-order language L, we introduce a computation mechanism that, given an interpretation I for L and a formula $\phi(\bar{x})$ of L, computes a witness $\bar{a}$ (that is, a vector of elements of the domain of I such that $\phi(\bar{a})$ holds in I) if $\phi(\bar{x})$ is satisfiable in I, and otherwise it reports a failure.

From the technical point of view our approach, called *formulas as programs*, is obtained by isolating a number of concepts and ideas present (often implicitly) in the logic programming and constraint logic programming framework, and reusing them in a simple and self-contained way. In fact, the proposed computation mechanism and a rigorous account of its formal properties rely only on the basics of first-order logic. This contrasts with the expositions of logic programming and constraint logic programming which require introduction of several concepts and auxiliary results (see for the latter e.g. Jaffar, Maher, Marriott & Stuckey (1998)).

[1] This can be made precise in the following way. Let $\vec{x}$ be the free variables of the specification ϕ_s, and $\vec{y}$ some auxiliary variables used in the program ϕ_p. Now correctness of the program with respect to the specification can be expressed by the sentence $\forall \vec{x}\ ((\exists \vec{y}\ \phi_p(\vec{x}, \vec{y})) \rightarrow \phi_s(\vec{x}))$, to be valid under the fixed interpretation. This sentence ensures that all solutions found by the program indeed satisfy the specification. Note that, under this definition, a program corresponding to a false formula is vacuously "correct", because there are no solutions found. Therefore the stronger notion of correctness and completeness obtained by requiring also the converse implication above, and loosely phrased as "equivalence in an appropriate sense", is the more adequate one.

1.3 Computing Mechanism

Let us explain now the proposed computation mechanism by means of an example. Consider the formula

$$(x = 2 \lor x = 3) \land (y = x + 1 \lor 2 = y) \land (2 * x = 3 * y) \tag{1.1}$$

interpreted over the standard structure of natural numbers. Is it satisfiable? The answer is "yes": indeed, it suffices to assign 3 to x and 2 to y.

In fact, we can compute this valuation systematicaly by initially assigning 2 to x and first trying the assignment of the value of $x + 1$, so 3, to y. As for this choice of value for y the equality $2 * x = 3 * y$ does not hold, we are led to the second possibility, assignment of 2 to y. With this choice $2 * x = 3 * y$ does not hold either. So we need to assign 3 to x and, eventually, 2 to y.

The above informal argument can be extended to a systematic procedure that attempts to find a satisfying valuation for a large class of formulas.

1.4 Plan and Rationale of the Paper

This paper is organized as follows.

In Section 2 we provide a formal account of the proposed computation mechanism. Admittedly, this computation mechanism is incomplete in the sense that in some circumstances the computation abnormally terminates in an error. This is unavoidable because for many first-order languages and interpretations, for example the language of Peano arithmetic and its standard interpretation, the set of true closed formulas is undecidable.

In Section 3 we show that this approach is both correct (sound) and, in the absence of errors, complete. In the Appendix, Subsection 9.3, 9.4, we investigate ways of limiting the occurrence of errors for the case of negation and implication.

For programming purposes first-order logic has limited expressiveness, so we extend it in Section 4 by a number of features that are useful for programming. This involves sorts (i.e., types), use of arrays and bounded quantifiers. The resulting fragment is surprisingly expressive and the underlying computation mechanism allows us to interpret many formulas as highly non-trivial programs.

As already mentioned above, formulas as programs approach to computing here discussed is inspired by logic programming and constraint logic programming but differs from them in a number of ways.

For example, formula (1.1) cannot be interpreted as a logic programming query or run as a Prolog query. The reason is that the equality symbol in logic programming and Prolog stands for "is unifiable with" and the term $2 * x$ does not unify with $3 * y$. In case of Prolog a possible remedy is to replace in (1.1) specific occurrences of the equality symbol by Prolog's arithmetic equality "`=:=`" or by the Prolog evaluator operator `is`. The correct Prolog query that corresponds to formula (1.1) is then

```
(X = 2 ; X = 3), (Y is X+1 ; 2 = Y), 2*X =:= 3*Y.
```

(Recall that "`;`" stands in Prolog for disjunction and "`,`" for conjunction.) This is clearly much less readable than (1.1) as three different kinds of equality-like relations are used here.

A more detailed comparison with (constraint) logic programming and Prolog requires knowledge of the details of our approach and is postponed to Section 5. In principle, the formulas as programs approach is a variant of constraint logic programming in which both recursion and constraint handling procedures are absent, but the full first-order syntax is used. We also compare in Section 5 our formulas as programs approach with the formulas as types approach, also called the Curry-Howard-De Bruijn interpretation.

The formulas as programs approach to programming has been realized in the programming language Alma-0 Apt et al. (1998) that extends imperative programming by features that support declarative programming. This shows that this approach, in contrast to logic programming and constraint logic programming, can easily be combined with imperative programming. So the introduced restrictions, such as lack of a constraint store, can be beneficial in practice. In Section 6 we summarize the main features of Alma-0.

The work reported here can be used to provide logical underpinnings for a fragment of Alma-0 that does not include destructive assignment or recursive procedures, and to reason about programs written in this fragment. We substantiate the latter claim by presenting in Section 7 the correctness proof of a purely declarative Alma-0 solution to the well-known non-trivial combinatorial problem of partitioning a rectangle into a given set of squares.

In conclusion, we provided here a realistic framework for declarative programming based on first-order logic and the traditional Tarskian semantics, which can be combined in a straightforward way with imperative programming.

2. Computation Mechanism

Consider an arbitrary first-order language with equality and an interpretation for it. We assume in particular a domain of discourse, and a fixed signature with a corresponding interpretation of its elements in the domain.

Definition 2.1 [valuation, assignment] A *valuation* is a finite mapping from variables to domain elements. Valuations will be denoted as single-valued sets of pairs x/d, where x is a variable and d a domain element. We use $\alpha, \alpha', \beta, \ldots$ for arbitrary valuations and call α' an *extension* of α when $\alpha \subseteq \alpha'$, that is, every assigment to a variable by α also occurs in α'. Further, ε denotes the empty valuation.

Let α be a valuation. A term t is *α-closed* if all variables of t get a value in α. In that case t^α denotes the *evaluation* of t under α in the domain. More generally, for any expression E the result of the replacement of each α-closed term t by t^α is denoted by E^α.

An α-assignment is an equation $s = t$ one side of which, say s, is a variable that is not α-closed and the other side, t, is an α-closed term. □

In our setting, the only way to assign values to variables will be by evaluating an α-assignment as above. Given such an α-assignment, say $x = t$, we evaluate it by assigning to x the value t^α.

Definition 2.2 [formulas] In order to accomodate the definition of the operational semantics, the set of fomulas has an inductive definition which may look a bit peculiar. First, universal quantification is absent since we have no operational interpretation for it. Second, every formula is taken to be a conjunction, with every conjunct (if any) either an atomic formula (in short: an *atom*), or a disjunction, conjunction or implication of formulas, a negation of a formula or an existentially quantified formula. The latter two unary constructors are assumed to bind stronger then the previous binary ones. The atoms include equations of the form $s = t$, with s and t terms.

For maximal clarity we give here an inductive definition of the set of formulas. In the operational semantics all conjunctions are taken to be right associative.

1. The empty conjunction $\Box$ is a formula.
2. If ψ is a formula and A is an atom, then $A \wedge \psi$ is a formula.
3. If ψ, ϕ_1, ϕ_2 are formulas, then $(\phi_1 \vee \phi_2) \wedge \psi$ is a formula.
4. If ψ, ϕ_1, ϕ_2 are formulas, then $(\phi_1 \wedge \phi_2) \wedge \psi$ is a formula.
5. If ψ, ϕ_1, ϕ_2 are formulas, then $(\phi_1 \rightarrow \phi_2) \wedge \psi$ is a formula.
6. If ϕ, ψ are formulas, then $\neg\phi \wedge \psi$ is a formula.
7. If ϕ, ψ are formulas, then $\exists x\ \phi \wedge \psi$ is a formula. □

Definition 2.3 [operational semantics] The operational semantics of a formula will be defined in terms of a tree $[\![\phi]\!]_\alpha$ depending on the formula ϕ and the (initial) valuation α. The root of $[\![\phi]\!]_\alpha$ is labelled with the pair ϕ, α. All internal nodes of the tree $[\![\phi]\!]_\alpha$ are labelled with pairs consisting of a formula and a valuation. The leaves of the tree $[\![\phi]\!]_\alpha$ are labelled with either

- *error* (representing the occurrence of an error in this branch of the computation), or
- *fail* (representing logical failure of the computation), or
- a valuation (representing logical success of the computation and yielding values for the free variables of the formula that make the formula true). □

It will be shown that valuations labelling success leaves are always extensions of the initial valuation. For a fixed formula, the operational semantics can be viewed as a function relating the initial valuation to the valuations labelling success leaves.

We can now define the computation tree $[\![\phi]\!]_\alpha$. The reader may consult first Figure 1 to see such a tree for formula (1.1) and the empty valuation ε.

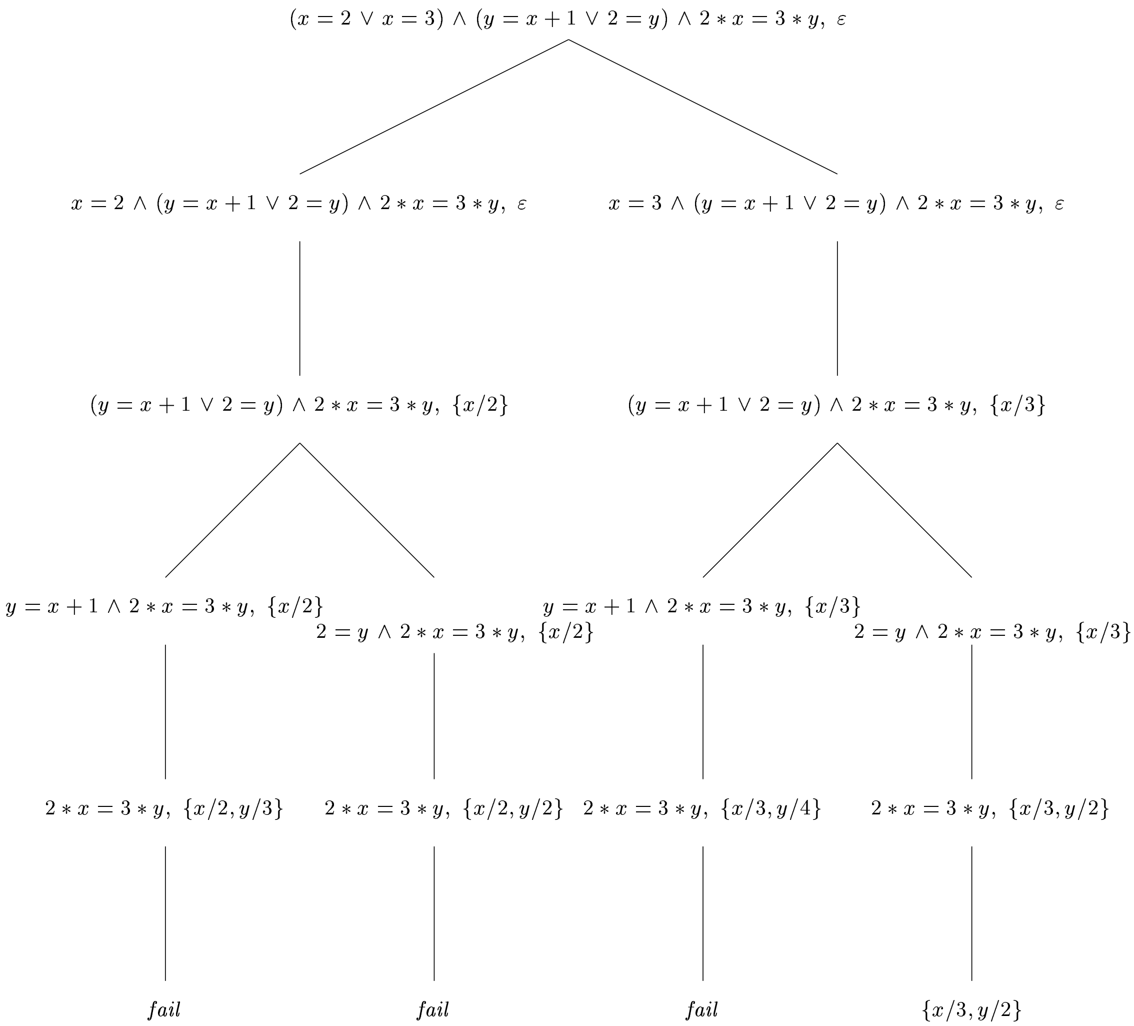

Figure 1: The computation tree for formula (1) and valuation ε.

Definition 2.4 [computation tree] The (computation) tree $[\![\phi]\!]_\alpha$ is defined by lexicographic induction on the pairs consisting of the *size* of the formula ϕ, and of the *size* of the formula ϕ_1 for which ϕ is of the form $\phi_1 \wedge \psi$, following the structure given by Definition 2.2.

1. For the empty conjunction we define $[\![\Box]\!]_\alpha$ to be the tree with the root that has a success leaf α as its son:

$$\begin{tikzcd} \Box,\ \alpha \arrow[d, dash] \\ \alpha \end{tikzcd}$$

2. If ψ is a formula and A is an atom, then we distinguish four cases depending on the form of A. In all four cases $[\![A \wedge \psi]\!]_\alpha$ is a tree with a root of degree one.

   - Atom A is α-closed and true. Then the root of $[\![A \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_\alpha$ as its subtree:

$$\begin{tikzcd} A \wedge \psi,\ \alpha \arrow[d, dash] \\ {[\![\psi]\!]_\alpha} \end{tikzcd}$$

   - Atom A is α-closed and false. Then the root of $[\![A \wedge \psi]\!]_\alpha$ has the failure leaf *fail* as its son:

$$\begin{tikzcd} A \wedge \psi,\ \alpha \arrow[d, dash] \\ \mathit{fail} \end{tikzcd}$$

   - Atom A is not α-closed, but is not an α-assignment. Then the root of $[\![A \wedge \psi]\!]_\alpha$ has the *error* leaf as its son:

$$\begin{tikzcd} A \wedge \psi,\ \alpha \arrow[d, dash] \\ \mathit{error} \end{tikzcd}$$

   - Atom A is an α-assignment $s = t$. Then either s or t is a variable which is not α-closed, say $s \equiv x$ with x not α-closed and t α-closed. Then the root of $[\![A \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_{\alpha'}$ as its subtree, where α' extends α with the pair x/t^α:

$$\begin{tikzcd} A \wedge \psi,\ \alpha \arrow[d, dash] \\ {[\![\psi]\!]_{\alpha'}} \end{tikzcd}$$

   The symmetrical case is analogous.

3. If ψ, ϕ_1, ϕ_2 are formulas, then we put $[\![(\phi_1 \vee \phi_2) \wedge \psi]\!]_\alpha$ to be the tree with a root of degree two and with left and right subtrees $[\![\phi_1 \wedge \psi]\!]_\alpha$ and $[\![\phi_2 \wedge \psi]\!]_\alpha$, respectively:

$$\begin{tikzcd}[column sep=small] & (\phi_1 \vee \phi_2) \wedge \psi,\ \alpha \arrow[dl, dash] \arrow[dr, dash] & \\ [\![\phi_1 \wedge \psi]\!]_\alpha & & [\![\phi_1 \wedge \psi]\!]_\alpha \end{tikzcd}$$

Observe that $\phi_1 \wedge \psi$ and $\phi_2 \wedge \psi$ are smaller formulas than $(\phi_1 \vee \phi_2) \wedge \psi$ in the adopted lexicographic ordering.

4. If ψ, ϕ_1, ϕ_2 are formulas, then we put $[\![(\phi_1 \wedge \phi_2) \wedge \psi]\!]_\alpha$ to be the tree with a root of degree one and the tree $[\![\phi_1 \wedge (\phi_2 \wedge \psi)]\!]_\alpha$ as its subtree:

$$\begin{tikzcd} (\phi_1 \wedge \phi_2) \wedge \psi,\ \alpha \arrow[d, dash] \\ [\![\phi_1 \wedge (\phi_2 \wedge \psi)]\!]_\alpha \end{tikzcd}$$

This substantiates the association of conjunctions to the right as mentioned in Definition 2.2. Note that, again, the definition refers to lexicographically smaller formulas.

5. If ψ, ϕ_1, ϕ_2 are formulas, then we put $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$ to be a tree with a root of degree one. We distinguish three cases.

   - Formula ϕ_1 is α-closed and $[\![\phi_1]\!]_\alpha$ contains only failure leaves. Then the root of $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_\alpha$ as its subtree:

$$\begin{tikzcd} (\phi_1 \rightarrow \phi_2) \wedge \psi,\ \alpha \arrow[d, dash] \\ [\![\psi]\!]_\alpha \end{tikzcd}$$

   - Formula ϕ_1 is α-closed and $[\![\phi_1]\!]_\alpha$ contains at least one success leaf. Then the root of $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$ has $[\![\phi_2 \wedge \psi]\!]_\alpha$ as its subtree:

$$\begin{tikzcd} (\phi_1 \rightarrow \phi_2) \wedge \psi,\ \alpha \arrow[d, dash] \\ [\![\phi_2 \wedge \psi]\!]_\alpha \end{tikzcd}$$

   - In all other cases the root of $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$ has the error leaf *error* as its son:

$$\begin{tikzcd} (\phi_1 \rightarrow \phi_2) \wedge \psi,\ \alpha \arrow[d, dash] \\ \mathit{error} \end{tikzcd}$$

The above definition relies on the logical equivalence of $\phi_1 \rightarrow \phi_2$ and $\neg\phi_1 \vee \phi_1$, but avoids unnecessary branching in the computation tree that would be introduced by the disjunction. In the Appendix, Subsection 9.3, we explain how in the first case the condition that ϕ_1 is α-closed can be relaxed.

6. If ϕ, ψ are formulas, then to define $[\![\neg\phi \wedge \psi]\!]_\alpha$ we distinguish three cases with respect to ϕ. In all of them $[\![\neg\phi \wedge \psi]\!]_\alpha$ is a tree with a root of degree one.

   - Formula ϕ is α-closed and $[\![\phi]\!]_\alpha$ contains only failure leaves. Then the root of $[\![\neg\phi \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_\alpha$ as its subtree:

$$\begin{array}{c} \neg\phi \wedge \psi,\ \alpha \\ | \\ [\![\psi]\!]_\alpha \end{array}$$

   - Formula ϕ is α-closed and $[\![\phi]\!]_\alpha$ contains at least one success leaf. Then the root of $[\![\neg\phi \wedge \psi]\!]_\alpha$ has the failure leaf *fail* as its son:

$$\begin{array}{c} \neg\phi \wedge \psi,\ \alpha \\ | \\ \mathit{fail} \end{array}$$

   - In all other cases the root of $[\![\neg\phi \wedge \psi]\!]_\alpha$ has the error leaf *error* as its son:

$$\begin{array}{c} \neg\phi \wedge \psi,\ \alpha \\ | \\ \mathit{error} \end{array}$$

   There are basically two classes of formulas ϕ in this contingency: those that are not α-closed and those for which $[\![\phi]\!]_\alpha$ contains no success leaf and at least one error leaf. In Subsection 9.3 we give some examples of formulas in the first class and show how in some special cases their negation can still be evaluated in a sound way.

7. The case of $\exists x\ \phi \wedge \psi$ requires the usual care with bound variables to avoid name clashes. Let α be a valuation. First, we require that the variable x does not occur in the domain of α. Second, we require that the variable x does not occur in ψ. Both requirements are summarized by phrasing that x is *fresh* with respect to α and ψ. They can be met by appropriately renaming the bound variable x.

   With x fresh as above we define $[\![\exists x\ \phi \wedge \psi]\!]_\alpha$ to be the tree with a root of degree one and $[\![\phi \wedge \psi]\!]_\alpha$ as its subtree:

$$(\exists x\phi) \land \psi,\ \alpha$$

$$\Big|$$

$$[\![\phi \land \psi]\!]_\alpha$$

Thus the operational semantics of $\exists x\ \phi \land \psi$ is, apart from the root of degree one, identical to that of $\phi \land \psi$. This should not come as a surprise, as $\exists x\ \phi \land \psi$ is logically equivalent to $\exists x\ (\phi \land \psi)$ when x does not occur in ψ.

Observe that success leaves of $[\![\phi \land \psi]\!]_\alpha$, and hence of $[\![\exists x\ \phi \land \psi]\!]_\alpha$, may or may not contain an assignment for x. For example, $\exists x\ x = 3 \land \psi$ yields an assignment for x, but $\exists x\ 3 = 3 \land \psi$ does not. In any case the assignment for x is not relevant for the formula as a whole, as the bound variable x is assumed to be fresh. In an alternative approach, the possible assignment for x could be deleted. □

3. Soundness and Completeness

The computation mechanism defined in the previous section attempts to find a valuation that makes the original formula true if this formula is satisfiable, and otherwise it reports a failure. The lexicographic ordering used in Definition 2.3 guarantees that for any formula the computation tree is finite. In this section we prove correctness and completeness of this mechanism.

We start with an easy lemma which is helpful to keep track of valuations, followed by a definition.

Lemma 3.1 *For every formula ϕ and valuation α, $[\![\phi]\!]_\alpha$ contains only valuations extending α with pairs x/d, where x occurs free in ϕ or appears existentially quantified in ϕ. Moreover, if ϕ is α-closed then $[\![\phi]\!]_\alpha$ contains only valuations extending α with variables that appear existentially quantified in ϕ.*

Proof. By induction on the lexicographic ordering of formulas as given in Definition 2.4. □

Definition 3.2 [status of computation tree] A computation tree is

- *successful* if it contains a success leaf,
- *failed* if it contains only failure leaves,
- *determined* if it is either successful or failed, that is, it either contains a success leaf or contains only failure leaves. □

Note that according to this definition a successful tree can contain the *error* leaves. This means that the *error* leaves differ from Prolog's run-time errors. In fact, in a top-down implementation of the proposed computation mechanism the depth-first search traversal of a computation tree should *not* abort but rather backtrack upon encounter of such a leaf and continue, if possible, in a search for a successful leaf.

We can now state the desired correctness result.

Theorem 3.3 (Soundness) *Let ϕ be a formula and α a valuation.*

(i) If $[\![\phi]\!]_\alpha$ contains a success leaf labelled with α', then α' extends α and $\forall(\phi^{\alpha'})$ is true. (In particular $\exists(\phi^\alpha)$ is true in this case.)

(ii) If $[\![\phi]\!]_\alpha$ is failed, then $\exists(\phi^\alpha)$ is false.

Proof. See Appendix, Subsection 9.1. □

The computation mechanism defined in Section 3 is obviously incomplete due to the possibility of errors. The following results states that, in the absence of errors, this mechanism is complete.

Theorem 3.4 (Completeness) *Let ϕ be a formula and α a valuation such that $[\![\phi]\!]_\alpha$ is determined.*

(i) Suppose that $\exists(\phi^\alpha)$ is true. Then the tree $[\![\phi]\!]_\alpha$ is successful.

(ii) Suppose that $\exists(\phi^\alpha)$ is false. Then the tree $[\![\phi]\!]_\alpha$ is failed.

Proof. See Appendix, Subsection 9.2. □

It is useful to point out that the computation mechanism of Section 2 used in the above theorems is by no means a simple counterpart of the provability relation of the first-order logic.

For the sake of further discussion let us say that two formulas ϕ and ψ are *equivalent* if

- the computation tree $[\![\phi]\!]_\varepsilon$ is successful iff the computation tree $[\![\psi]\!]_\varepsilon$ is successful and in that case both computation trees have the same set of successful leaves,
- $[\![\phi]\!]_\varepsilon$ is failed iff $[\![\psi]\!]_\varepsilon$ is failed.

Then $\phi \wedge \psi$ is not equivalent to $\psi \wedge \phi$ (consider $x = 0 \wedge x < 1$ and $x < 1 \wedge x = 0$) and $\neg(\phi \wedge \psi)$ is not equivalent to $\neg\phi \vee \neg\psi$ (consider $\neg(x = 0 \wedge x = 1)$ and $\neg(x = 0) \vee \neg(x = 1)$. In contrast, $\phi \vee \psi$ *is* equivalent to $\psi \vee \phi$.

We can summarize this treatment of the connectives by saying that we use a sequential conjunction and a parallel disjunction. The above notion of equivalence deviates from the usual one, for example de Morgan's Law is not valid.

A complete axiomatization of the equivalence relation induced by the computation mechanism of Section 2 is an interesting research topic.

4. Extensions

The language defined up to now is clearly too limited as a formalism for programming. Therefore we discuss a number of extensions of it that are convenient for programming purposes. These are: non-recursive procedures, sorts (i.e., types), arrays and bounded quantification.

4.1 Non-recursive Procedures

We consider here non-recursive procedures. These can easily be introduced in our framework using the well-known *extension by definition* mechanism (see, e.g., Shoenfield (1967)[pages 57-58]).

More specifically, consider a first-order formula ψ with the free variables $x_1, \ldots, x_n$. Let p be a *new* n-ary relation symbol. Consider now the formula

$$p(x_1, \ldots, x_n) \leftrightarrow \psi$$

that we call the *definition* of p.

Suppose that, by iterating the above procedure, we have a collection P of definitions of relation symbols. We assume furthermore that the fixed but arbitrary interpretation has been extended with interpretations of the new relation symbols in such a way that all definitions in P become true. There is only one such extension for every initial interpretation.

Let ϕ be a formula in the extended first-order language, that is, with atoms $p(t_1, \ldots, t_n)$ from P included. We extend the computation mechanism $[\![\phi]\!]_\alpha$ of Section 2, by adding at the beginning of Clause 2 in Definition 2.4 the following item for handling atoms $p(t_1, \ldots, t_n)$ from P.

- Atom A is of the form $p(t_1, \ldots, t_n)$, where p is a defined relation symbol with the definition

  $$p(x_1, \ldots, x_n) \leftrightarrow \psi_p.$$

  Then the root of $[\![A \wedge \psi]\!]_\alpha$ has $[\![\psi_p\{x_1/t_1, \ldots, x_n/t_n\} \wedge \psi]\!]_\alpha$ as its subtree:

  $$\begin{array}{c} A \wedge \psi,\ \alpha \\ \Big| \\ [\![\psi_p\{x_1/t_1, \ldots, x_n/t_n\} \wedge \psi]\!]_\alpha \end{array}$$

  Here $\psi_p\{x_1/t_1, \ldots, x_n/t_n\}$ stands for the result of substituting in ψ_p the free occurrences of the variables $x_1, \ldots, x_n$ by $t_1, \ldots, t_n$, respectively.

The proof of the termination of this extension of the computation mechanism introduced in Section 2 relies on a refinement of the lexicographic ordering used in Definition 2.4, taking into account the new atoms.

The above way of handling defined relation symbols obviously corresponds to the usual treatment of procedure calls in programming languages.

The soundness and completeness results can easily be extended to the case of declared relation symbols. In this version truth and falsity refer to the extended interpretation.

So far for *non-recursive* procedures.

4.2 Sorts

In this subsection we introduce sorts (i.e., types). The extension of one-sorted to many-sorted first-order logic is standard. It requires a refinement of the notion of signature: arities are no longer just numbers, but have to specify the sorts of the arguments of the function and predicate symbols, as well as the sorts of the function values. Terms and atoms are well-formed if the sorts of the arguments comply with the signature. In quantifying a variable, its sort should be made explicit (or should at least be clear from the context).

Interpretations for many-sorted first-order languages are obtained by assigning to each sort a non-empty domain and by assigning to each function symbol and each predicate symbol respectively an appropriate function and relation on these sorts.

Sorts can be used to model various basic data types occurring in programming practice: integers, booleans, characters, but also compound data types such as arrays.

4.3 Arrays

Arrays can be modelled as vectors or matrices, using projection functions that are given a *standard interpretation*. Given a sort for the indices (typically, a segment of integers or a product of segments) and a sort for the elements of the array, we add a sort for arrays of the corresponding type to the signature. We also add to the language *array variables*, or *arrays* for short, to be interpreted as arrays in the standard interpretation.

We use the letters a, b, c to denote arrays and to distinguish arrays from objects of other sorts. We write $a[t_1, \ldots, t_n]$ to denote the projection of the array a on the index $[t_1, \ldots, t_n]$, akin to the use of subscripted variables in programming languages. The standard interpretation of each projection function maps a given array and a given index to the correct element. Thus subscripted variables are simply terms. These terms are handled by means of an extension of the computation mechanism of Section 2.

A typical example of the use of such a term is the formula $a[0, 0] = 1$, which should be matched with the formula $x = 1$ in the sense that the evaluation of each equality can result in an assignment of the value 1 to a variable, either $a[0,0]$ or x. So we view $a[0,0]$ as a variable and not as a compound term.

To this end we extend a number of notions introduced in the previous section.

Definition 4.1 An *array valuation* is a finite mapping whose elements are of the form $a[d_1,\ldots,d_n]/d$, where a is an n-ary array symbol and $d_1,\ldots,d_n,d$ are domain elements. An *extended valuation* is a finite mapping that is a union of a valuation and an array valuation. □

The idea is that an element $a[d_1,\ldots,d_n]/d$ of an array valuation asssigns the value d to the (interpretation of) array a applied to the arguments $d_1,\ldots,d_n$. Then, if the terms $t_1,\ldots,t_n$ evaluate to the domain elements $d_1,\ldots,d_n$ respectively, the term $a[t_1,\ldots,t_n]$ evaluates to d. This simple inductive clause yields an extension of the notion of evaluation t^α, where α is an extended valuation, to terms t in the presence of arrays. The notions of an α-closed term and an α-assignment are now somewhat more complicated to define.

Definition 4.2 Consider an extended valuation α.

- A variable x is *α-closed* if for some d the pair x/d is an element of α.
- A term $f(t_1,\ldots,t_n)$, with f a function symbol, is *α-closed* if each term t_i is α-closed.
- A term $a[t_1,\ldots,t_n]$ is *α-closed* if each term t_i is α-closed and evaluates to a domain element d_i such that for some d the pair $a[d_1,\ldots,d_n]/d$ is an element of α.

An equation $s = t$ is an *α-assignment* if either

- one side of it, say s, is a variable that is not α-closed and the other, t, is an α-closed term, or
- one side of it, say s, is of the form $a[t_1,\ldots,t_n]$, where each t_i is α-closed but $a[t_1,\ldots,t_n]$ is not α-closed, and the other, t, is an α-closed term. □

The idea is that an array a can be assigned a value at a selected position by evaluating an α-assignment $a[t_1,\ldots,t_n] = t$. Assuming the terms $t_1,\ldots,t_n,t$ are α-closed and evaluate respectively to $d_1,\ldots,d_n,d$, the evaluation of $a[t_1,\ldots,t_n] = t$ results in assigning the value d to the array a at the position $d_1,\ldots,d_n$.

With this extension of the notions of valuation and α-assignment we can now apply the computation mechanism of Section 2 to first-order formulas with arrays. The corresponding extensions of the soundness and completeness theorems of Section 3 remain valid.

4.4 Bounded quantification

In this subsection we show how to extend the language with a form of bounded quantification that essentially amounts to the generalized conjunction and disjunction. We treat bounded quantification with respect to the integer numbers, but the approach can easily be generalized to data types with the same discrete and ordered structure as the integers.

Definition 4.3 [bounded quantification] Let α be a valuation and let $\phi(x)$ be a formula with x not occurring in the domain of α. Furthermore, let s,t be terms of integer type. We assume the set of formulas to be extended in such a way that also $\exists x \in [s..t]\ \phi(x)$ and $\forall x \in [s..t]\ \phi(x)$ are formulas. The computation trees of these formulas have a root of degree one and depend on s and t in the following way.

- If s or t is not α-closed, then the roots of both $[\![\exists x \in [s..t]\ \phi(x)]\!]_\alpha$ and $[\![\forall x \in [s..t]\ \phi(x)]\!]_\alpha$ have the error leaf *error* as its son.
- If s and t are α-closed and $s^\alpha > t^\alpha$, then the root of $[\![\exists x \in [s..t]\ \phi(x)]\!]_\alpha$ has the failure leaf *fail* as its son and the root of $[\![\forall x \in [s..t]\ \phi(x)]\!]_\alpha$ has a success leaf α as its son.

- If s and t are α-closed and $s^\alpha \leq t^\alpha$, then
  - the root of $[\![\exists x \in [s..t]\ \phi(x)]\!]_\alpha$ has $[\![\phi(x) \vee \exists y \in [s{+}1..t]\ \phi(y)]\!]_{\alpha \cup \{x/s^\alpha\}}$ as its subtree,
  - the root of $[\![\forall x \in [s..t]\ \phi(x)]\!]_\alpha$ has $[\![\phi(x) \wedge \forall y \in [s{+}1..t]\ \phi(y)]\!]_{\alpha \cup \{x/s^\alpha\}}$ as its subtree.

  In both cases y should be a fresh variable with respect to $\alpha, \phi(x)$ in order to avoid name clashes.

The soundness and completeness results can easily be extended to include bounded quantification. □

5. Relation to Other Approaches

The work here discussed is related in many interesting ways to a number of seminal papers on logic, logic programming and constraint logic programming.

5.1 Definition of Truth compared to Formulas as Programs

First, it is instructive to compare our approach to the inductive definition of truth given in Tarski (1933). This definition can be interpreted as an algorithm that, given a first-order language L, takes as input an interpretation I of L and a formula ϕ of L, and yields as output the answer to the question whether the universal closure of ϕ is true in I. This algorithm is not effective because of the way quantifiers are dealt with. This is unavoidable since truth is undecidable for many languages and interpretations, for instance Peano arithmetic and its standard model.

In the formulas as programs approach the initial problem is modified in that one asks for a constructive answer to the question whether a formula is satisfiable in an interpretation. The algorithm proposed here is effective at the cost of occasionally terminating abnormally in an error.

5.2 Relation to Logic Programming

Some forty years later, in his seminal paper Kowalski (1974) proposed to use first-order logic as a computation formalism. This led to logic programming. However, in spite of the paper's title, only a subset of first-order logic is used in his proposal, namely the one consisting of Horn clauses. This restriction was essential since what is now called SLD-resolution was used as the computation mechanism.

In the discussion we first concentrate on the syntax matters and then focus on the computation mechanism.

The restriction of logic programs and goals to Horn clauses was gradually lifted in Clark (1978), by allowing negative literals in the goals and in clause bodies, in Lloyd & Topor (1984), by allowing arbitrary first-order formulas as goals and clause bodies, and in Lobo, Minker & Rajasekar (1992) by allowing disjunctions in the clause heads. In each case the computation mechanism of SLD-resolution was suitably extended, either by introducing the negation as failure rule, or by means of transformation rules, or by generalizing so-called linear resolution.

From the syntactic point of view our approach is related to that of Lloyd & Topor (1984). Appropriate transformation rules are used there to get rid of quantifiers, disjunctions and the applications of negation to non-atomic formulas. So these features of first-order logic are interpreted in an indirect way. It is useful to point out that the approach of Lloyd & Topor (1984) was implemented in the programming language Gödel of Hill & Lloyd (1994).

Further, it should be noted that bounded quantifiers and arrays were also studied in logic programming. In particular, they are used in the specification language Spill of Kluźniak & Miłkowska (1997) that allows us to write executable, typed, specifications in the logic programming style. Other related references are Voronkov (1992), Barklund & Bevemyr (1993) and Apt (1996).

So from the syntactic point of view our approach does not seem to differ from logic programming in an essential way. The difference becomes more apparent when we analyze in more detail the underlying computation mechanism.

To this end it is useful to recall that in logic programming the computing process takes place implicitly over the free algebra of all terms and the values are assigned to variables by means of unification. The first aspect can be modelled in the formulas as programs approach by choosing a *term interpretation*, so an interpretation the domain D of which consists of all terms and such that each n-ary function symbol f is mapped to a function f_D that assigns to elements (so terms) $t_1, \ldots, t_n$ of D the term $f(t_1, \ldots, t_n)$. With this choice our use of α-assignment boils down to an instance of matching which in turn is a special case of unification.

Unification in logic programming can be more clearly related to equality by means of the so-called homogenization process the purpose of which is to remove non-variable terms from the clauses heads. For instance,

```
append(x1, ys, z1) <- x1 = [x | xs], z1 = [x | zs], append(xs, ys, zs)
```

is a homogenized form of the more compact clause

```
append([x | xs], ys, [x | zs]) <- append(xs, ys, zs).
```

To intepret the equality in the right way the single clause

```
x = x <-
```

should then be added. This enforces the "is unifiable with" intepretation of equality. So the homogenization process reveals that logic programming relies on a more general interpretation of equality than the formulas as programs approach. It allows one to avoid generation of errors for all equality atoms.

In conclusion, from the computational point of view, the logic programming approach is at the same time a restriction of the formulas as programs approach to the term interpretations and a generalization of this approach in which all equality atoms can be safely evaluated.

5.3 Relation to Pure Prolog

By pure Prolog we mean here a subset of Prolog formed by the programs and goals that are Horn clauses.

Programming in Prolog and in its pure subset relies heavily on lists and recursion. As a result termination is one of the crucial issues. This led to an extensive study of methods that allow us to prove termination of logic and Prolog programs (see De Schreye & Decorte (1994) for a survey of various approaches).

In contrast, our approach to programming is based on arrays and iteration that is realized by means of bounded quantification. These constructs are guaranteed to terminate. In fact, it is striking how far one can go in programming in this style without using recursion. If the reader is not convinced by the example given of Section 7 below, he/she is invited to consult other examples in Voronkov (1992) and Apt et al. (1998).

In the formulas as programs approach the absence of recursion makes it possible to analyze queries without explicit presence of procedures, by systematically replacing procedures by their bodies. This allows us to represent each program as a single query and then rely on the well-understood Tarskian semantics of first-order logic.

In the standard logic programming setting very few interesting programs can be represented in this way. In fact, as soon as recursion is used, a query has to be studied in the context of a program that defines the recursive procedures. As soon as negation is also present, a plethora of different semantics arises — see e.g. Apt & Bol (1994). Finally, in the presence of recursion it is difficult to account for Prolog's selection rule in purely semantic terms.

5.4 Relation to Pure Prolog with Arithmetic

By pure Prolog with arithmetic we mean here an extension of pure Prolog by features that support arithmetic, so Prolog's arithmetic relations such as "=:=" and the Prolog evaluator operator `is`.

These features allow us to compute in the presence of arithmetic but in a clumsy way as witnessed by the example of formula (1.1) of Subsection 1.3 and its elaborated representation in Prolog in Subsection 1.4.

Additionally, a possibility of abnormal termination in an error arises. Indeed, both arithmetic relations and the `is` operator introduce a possibility of run-time errors, a phenomenon absent in pure

Prolog. For instance, the query `X is Y` yields an error and so does `X =:= Y`.

In contrast, in the formulas as programs approach arithmetic can be simply modelled by adding the sorts of integers and of reals. The α-assignment then deals correctly with arithmetic expressions because it relies on automatic evaluation of terms. This yields a simpler and more uniform approach to arithmetic in which no new special relation symbols are needed.

5.5 Relation to Constraint Logic Programming

The abovementioned deficiencies of pure Prolog with arithmetic have been overcome in constraint logic programming, an approach to computing that generalizes logic programming. In what follows we concentrate on a specific approach, the generic scheme CLP(X) of Jaffar & Lassez (1987) that generalizes pure Prolog by allowing constraints. In this scheme atoms are divided into those defined by means of clauses and those interpreted in a direct way. The latter ones are called constraints.

In CLP(X), as in our case, the computation is carried out over an arbitrary interpretation. At each step (instead of the unification test of logic programming and its application if it succeeds) satisfiability of the so far encountered constraints is tested. A computation is successful if the last query consists of constraints only.

There are two differences between the formulas as programs approach and the CLP(X) scheme. The first one has to do with the fact that in our approach full first-order logic is allowed, while in the latter — as in logic programming and pure Prolog — Horn clauses are used.

The second one concerns the way values are assigned. In our case the only way to assign values to variables is by means of an α-assignment, while in the CLP(X) scheme satisfiability of constraints guides the computation and output is identified with a set of constraints (that still have to be solved or normalized).

The CLP(X) approach to computing has been realized in a number of constraint logic programming languages, notably in the CLP($\mathcal{R}$) system of Jaffar, Michayov, Stuckey & Yap (1992) that is an instance of the CLP(X) scheme with a two-sorted structure that consists of reals and terms. In this system formula (1.1) of Subsection 1.3 can be directly run as a query.

Once negation is added to the CLP(X) scheme (it is in fact present in CLP($\mathcal{R}$)), the extension of the CLP(X) syntax to full first-order logic could be achieved by using the approach Lloyd & Topor (1984) or by extending the computation mechanism along the lines of Section 2.

So, ignoring the use of the first-order logic syntax in the formulas as programs approach and the absence of (recursive) procedures that could be added to it, the main difference between this approach and the CLP(X) scheme has to do with the fact that in the former only very limited constraints are admitted, namely ground atoms and α-assignments. In fact, these are the only constraints that can be resolved directly.

So from this point of view the formulas as programs approach is less general than constraint logic programming, as embodied in the CLP(X) scheme. However, this more limited approach does not rely on the satisfiability procedure for constraints (i.e., selected atomic formulas), or any of its approximations used in specific implementations. In fact, the formulas as programs approach attempts to clarify how far constraint logic programming approach can be used without any reliance on external procedures that deal with constraint solving or satisfiability.

5.6 Formulas as Programs versus Formulas as Types

In the so-called *formulas as types* approach, also called the Curry-Howard-De Bruijn interpretation (see e.g. Troelstra & van Dalen (1988)) (constructive) proofs of a formula are terms whose type is the formula in question. The type corresponding to a formula can thus be viewed as the (possibly empty) set of all proofs of the formula. Here 'proof' refers to an operational notion of proof, in which

- a proof of $\phi \vee \psi$ is either *left*(p) with p a proof of ϕ, or *right*(p) with p a proof of ψ;
- a proof of $\phi \wedge \psi$ is a pair $\langle p, q\rangle$ consisting of a proof p of ϕ and and a proof q of ψ;

- a proof of an implication $\phi \rightarrow \psi$ is a function that maps proofs of ϕ to proofs of ψ;
- a proof of $\forall x\ \phi(x)$ is a function that maps domain elements d to proofs of $\phi(d)$;
- a proof of $\exists x\ \phi(x)$ is of the form $ex(d, p)$ with domain element d a witness for the existential statement, and p a proof of $\phi(d)$.

Such proofs can be taken as programs. For example, a constructive proof of $\forall x\ \exists y\ \phi(x, y)$ is a function that maps d to an expression of the form $ex(e_d, p_d)$ with p_d a proof of $\phi(d, e_d)$. After extraction of the witness e_d the proof yields a program computing e_d from d.

The main difference between formulas as types and formulas as programs is that in the latter approach not the proofs of the formulas, but the formulas themselves have an operational interpretation. To illustrate this difference, consider the computation tree of formula (1.1) in Figure 1 with its proof:

$$ex(3, ex(2, \langle right(p_{3=3}), \langle right(p_{2=2}), p_{2*3=3*2}\rangle\rangle))$$

Here p_A is a proof of A, for each true closed atom A.

Observe that in the above proof the witnesses 3 and 2 for x and y, respectively, *have to be given beforehand*, whereas in our approach they are computed. In the formulas as programs approach the proofs are constructed in the successful branches of the computation tree and the computation is guided by the search for such a proof. Apart from differences in syntax, the reader will recognize the above proof in the successful branch of Figure 1.

Given the undecidability of the first-order logic, there is a price to be paid for formulas programs. It consists of the possibility of abnormal termination in an error.

6. Alma-0

We hope to have convinced the reader that the formulas as programs approach, though closely related to logic programmming, differs from it in a number of crucial aspects.

This approach to programming has been realized in the implemented programming language Alma-0 (Apt et al. 1998). A similar approach to programming has been taken in the 2LP language of McAloon & Tretkoff (1995). 2LP (which stands for “logic programming and linear programming”) uses C syntax and has been designed for constraint programming in the area of optimization.

Alma-0 is an extension of a subset of Modula-2 that includes nine new features inspired by the logic programming paradigm. We briefly recall those that are used in the sequel and refer to Apt et al. (1998) for a detailed presentation.

- Boolean expressions can be used as statements and vice versa. A boolean expression that is used as a statement and evaluates to `FALSE` is identified with a *failure*.
- *Choice points* can be created by the non-deterministic statements `ORELSE` and `SOME`. The former is a dual of the statement composition and the latter is a dual of the `FOR` statement. Upon failure the control returns to the most recent choice point, possibly within a procedure body, and the computation resumes with the next branch in the state in which the previous branch was entered.
- The notion of *initialized* variable is introduced and the equality test is generalized to an assignment statement in case one side is an uninitialized variable and the other side an expression with known value.
- A new parameter passing mechanism, *call by mixed form*, denoted by the keyword `MIX`, is introduced for variables of simple type. It works as follows: If the actual parameter is a variable, then it is passed by variable. If the actual parameter is an expression that is not a variable, its value is computed and assigned to a new variable v (generated by the compiler): it is v that is then passed by variable. So in this case the call by mixed form boils down to call by value.

Using this parameter mechanism we can pass both expressions with known values and uninitialized variables as actual parameters. This makes it possible to use a single procedure both for testing and computing.

For efficiency reasons the Alma-0 implementation does not realize faithfully the computation mechanism of Section 2 as far as the errors are concerned. First, an evaluation of an atom that is not α-closed and is not an α-assignment yields a run-time error. On the other hand, in the other two cases when the evaluation ends with the *error* leaf, in the statements `NOT S` and `IF S THEN T END`, the computation process of Alma-0 simply proceeds.

The rationale for this decision is that the use of insufficiently unstantiated atoms in Alma-0 programs is to be discouraged whereas the catching of other two cases for errors would be computationally prohibitive. In this respect the implementation of Alma-0 follows the same compromise as the implementations of Prolog.

We now associate with each first-order formula ϕ an Alma-0 statement $\mathcal{T}(\phi)$. This is done by induction on the structure of the formula ϕ. The translation process is given in Table 1.

Formula	Alma-0 construct
A (atom)	A
$\phi_1 \vee \phi_2$	`EITHER` $\mathcal{T}(\phi_1)$ `ORELSE` $\mathcal{T}(\phi_2)$ `END`
$\phi_1 \wedge \phi_2$	$\mathcal{T}(\phi_1); \mathcal{T}(\phi_2)$
$\phi \rightarrow \psi$	`IF` $\mathcal{T}(\phi)$ `THEN` $\mathcal{T}(\psi)$ `END`
$\neg\phi$	`NOT` $\mathcal{T}(\phi)$

$\exists x \phi(x, \bar{y})$	`p`$(\bar{y})$, where the procedure `p` is defined by
	`PROCEDURE p(MIX` $\bar{y}$: $\bar{\texttt{T}}$);
	`VAR` x : `T;`
	`BEGIN`
	  $\mathcal{T}(\phi(x, \bar{y}))$
	`END;`
	where `T` is the type (sort) of the variable x and
	$\bar{\texttt{T}}$ is the sequence of types of the variables in $\bar{y}$.

$\exists x \in [s..t]\phi$	`SOME` $x := s$ `TO` t `DO` $\mathcal{T}(\phi)$ `END`
$\forall x \in [s..t]\phi$	`FOR` $x := s$ `TO` t `DO` $\mathcal{T}(\phi)$ `END`

Table 1: Translation of formulas into Alma-0 statements.

This translation allows us to use in the sequel Alma-0 syntax to present specific formulas.

7. Example: Partitioning a Rectangle into Squares

To illustrate the Alma-0 programming style and the use of formulas as programs approach for program verification, we consider now the following variant of a problem from Honsberger (1970, pages 46-60).

Squares in the rectangle. Partition an integer sized $nx \times ny$ rectangle into given squares $S_1, \ldots, S_m$ of integer sizes $s_1, \ldots, s_m$.

We develop a solution that, in contrast to the one given in Apt et al. (1998), is purely declarative. To solve this problem we use a backtracking algorithm that fills in all the cells of the rectangle one by one, starting with the left upper cell and proceeding downward in the leftmost column, then the next column, and so on. The algorithm checks for each cell whether it is already covered by some square used to cover a previous cell. Given the order in which the cells are visited, it suffices to inspect the left neighbour cell and the upper neighbour cell (if these neighbours exist). This is done by the test

$$((1 < \texttt{i}) \texttt{ AND } (\texttt{i} < \texttt{RightEdge}[\texttt{i}-1,\texttt{j}])) \texttt{ OR } ((1 < \texttt{j}) \texttt{ AND } (\texttt{j} < \texttt{LowerEdge}[\texttt{i},\texttt{j}-1])). \quad (7.1)$$

Here `[i,j]` is the index of the cell under consideration, and `RightEdge[i-1,j]` is the right edge of the square covering the left neighbour (`[i-1,j]`, provided `i > 1`), and `LowerEdge[i, j-1]` is the lower edge of the square covering the upper neighbour (`[i,j-1]`, provided `j > 1`). The cell under consideration is already covered if and only if the test succeeds. If it is not covered, then the algorithm looks for a square not yet used, which is placed with its top-left corner at `[i,j]` provided the square fits within the rectangle. The algorithm backtracks when none of the available squares can cover the cell under consideration without sticking out of the rectangle. See Figure 2.

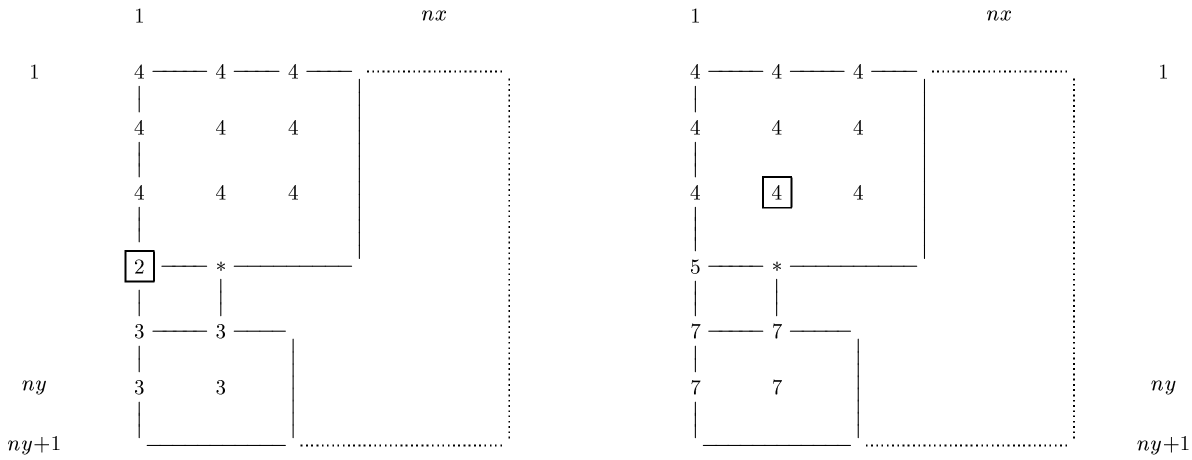

Figure 2: Example of values of `RightEdge` (left diagram) and `LowerEdge` (right diagram), respectively. Entry $*$ is indexed by `[2,4]`. It is not covered already since neither $2 < \texttt{RightEdge}[1,4] = 2$ nor $4 < \texttt{LowerEdge}[2,3] = 4$.

In test (7.1) we used the `AND` and `OR` connectives instead of the ";" and `ORELSE` constructs for the following reason. In case all variables occurring in a test are instantiated, some optimizations are in order. For example, it is not necessary to backtrack within the test, disjunctions do not have to create choice points, and so on. The use of `AND` and `OR` enables the compiler to apply these optimizations.

Backtracking is implemented by a `SOME` statement that checks for each square whether it can be put to cover a given cell. The solution is returned via two arrays `posX` and `posY` such that for square S_k (of size `Sizes[k]`) `posX[k]`, `posY[k]` are the coordinates of its top-left corner.

The two equations `posX[k] = i` and `posY[k] = j` are used both to construct the solution and to prevent using an already placed square again at a different place.

The declaration of the variables `posX` and `posY` as `MIX` parameters allows us to use the program both to check a given solution or to complete a partial solution.

```
TYPE SquaresVector = ARRAY [1..M] OF INTEGER;

PROCEDURE Squares(Sizes:SquaresVector, MIX posX, posY:SquaresVector);

VAR RightEdge,LowerEdge: ARRAY [1..NX],[1..NY] OF INTEGER;
    i,i1, j,j1, k: INTEGER;

BEGIN
```

```
  FOR i := 1 TO NX DO
    FOR j := 1 TO NY DO
      IF NOT
         (((1 < i) AND (i < RightEdge[i-1,j])) OR
         ((1 < j) AND (j < LowerEdge[i, j-1])))    (* cell [i,j] already covered? *)
      THEN
        SOME k := 1 TO M DO
          PosX[k] = i;
          PosY[k] = j;                             (* square k already used? *)
          Sizes[k] + i <= NX + 1;
          Sizes[k] + j <= NY + 1;                  (* square k fits? *)
          FOR i1 := 1 TO Sizes[k] DO
            FOR j1 := 1 TO Sizes[k] DO
              RightEdge[i+i1-1,j+j1-1] = i+Sizes[k];
              LowerEdge[i+i1-1,j+j1-1] = j+Sizes[k]
            END                                   (* complete administration *)
          END
        END
      END
    END
  END
END Squares;
```

This program is declarative and consequently has a dual reading as the formula

$$\begin{aligned}&\forall i \in [1..nx]\ \forall j \in [1..ny]\\ &\quad \neg(1 < i < \mathtt{RightEdge}[i{-}1, j] \vee 1 < j < \mathtt{LowerEdge}[i, j{-}1]) \rightarrow \exists k \in [1..m]\ \phi(i,j,k),\end{aligned}$$

where $\phi(i,j,k)$ is the formula

$$\mathtt{PosX}[k] = i \wedge \mathtt{PosY}[k] = j \wedge \mathtt{Sizes}[k]{+}i \leq nx{+}1 \wedge \mathtt{Sizes}[k]{+}j \leq ny{+}1 \wedge \psi(i,j,k)$$

and $\psi(i,j,k)$ is the formula

$$\begin{aligned}&\forall i' \in [1..\mathtt{Sizes}(k)]\ \forall j' \in [1..\mathtt{Sizes}(k)]\\ &\quad \mathtt{RightEdge}[i{+}i'{-}1, j{+}j'{-}1] = i{+}\mathtt{Sizes}[k] \wedge \mathtt{LowerEdge}[i{+}i'{-}1, j{+}j'{-}1] = j{+}\mathtt{Sizes}[k]\end{aligned}$$

This dual reading of the program entails over the standard interpretation the formula

$$\begin{aligned}&\forall i \in [1..nx]\ \forall j \in [1..ny]\ \exists k \in [1..m]\\ &\quad \mathtt{PosX}[k] \leq i < \mathtt{PosX}[k]{+}\mathtt{Sizes}[k] \leq nx{+}1 \wedge \mathtt{PosY}[k] \leq j < \mathtt{PosY}[k]{+}\mathtt{Sizes}[k] \leq ny{+}1 \qquad (7.2)\end{aligned}$$

expressing that every cell is covered by a square. The entailment is not trivial, but can be made completely rigorous. The proof uses arithmetic, in particular induction on lexicographically ordered pairs (i,j). This entailment actually means that the program satisfies its specification, that is, if the computation is successful, then a partition is found (and can be read off from $\mathtt{PosX}[k]$ and $\mathtt{PosY}[k]$). The latter fact relies on the Soundness Theorem 3.3.

Conversely, assuming that the surfaces of the squares sum up exactly to the surface of the rectangle, the specification (7.2) entails the formula corresponding to the program, with suitable values for $\mathtt{RightEdge}, \mathtt{LowerEdge}$. Furthermore, the absence of *error*s can be established by lexicographic induction. This ensures that the computation tree is always determined. By the Completeness Theorem 3.4, one always gets an answer. If this answer is negative, that is, if the computation tree is failed, then by the Soundness Theorem 3.3 the formula corresponding to the program cannot be satisfied, and hence (7.2) cannot be satisfied.

8. Current and Future Work

The work here presented can be pursued in a number of directions. We listed here the ones that seem to us most natural.

Recursive procedures The extension of the treatment of non-recursive procedures in Subsection 4.1 to the case of recursive procedures is far from obvious. It requires an extension of the computation mechanism to one with possible non-terminating behaviour. This could be done along the lines of Apt & Doets (1994) where the SLDNF-resolution of logic programs with negation is presented in a top down, non-circular way.

Also, on the semantic level several choices arise, much like in the case of logic programming, and the corresponding soundness and completeness results that provide a match between the computation mechanism and semantics need to be reconsidered from scratch.

Constraints As already said in Subsection 5.5, the formulas as programs approach can be seen as a special case of constraint logic programming, though with a full first-order syntax. It is natural to extend our approach by allowing constraints, so arbitrary atoms that have no definition in the sense of Subsection 4.1. The addition of constraints will require on the computation mechanism level use of a constraint store and special built-in procedures that approximate the satisfiability test for conjunctions of constraints.

Automated Verification The correctness proof presented in Section 7 was carried out manually. It boils down to a proof of validity of an implication between two formulas, This proof is based on an lexicographic ordering os it should be possible to mechanize this proof. This would lead a fully mechanized correctness proof of the Alma-0 program considered there.

Relation to Dynamic Predicate Logic In Groenendijk & Stokhof (1991) an alternative "input-output" semantics of first-order logic is provided. In this semantics both the connectives and the quantifiers obtain a different, dynamic, interpretation that better suits their use for natural language analysis. This semantic is highly nondeterministic due to its treatment of existential quantifiers and it does not take into account a possibility of errors.

It is natural to investigate the precise connection between this semantics and our formulas as programs approach. A colleague of us, Jan van Eijck, has recently undertook this study. Also, it would be useful to clarify to what extent our approach can be of use for linguistic analysis, both as a computation mechanism and as a means for capturing errors in discourse analysis.

Acknowledgements

We would like to thank David Scott Warren for a number of helpful suggestions.

References

Apt, K. R. (1996), 'Arrays, bounded quantification and iteration in logic and constraint logic programming', *Science of Computer Programming* **26**(1-3), 133–148.

Apt, K. R. & Bol, R. (1994), 'Logic programming and negation: a survey', *Journal of Logic Programming* **19-20**, 9–71.

Apt, K. R. & Doets, H. C. (1994), 'A new definition of SLDNF-resolution', *Journal of Logic Programming* **18**(2), 177–190.

Apt, K. R., Brunekreef, J., Partington, V. & Schaerf, A. (1998), 'Alma-0: An imperative language that supports declarative programming', *ACM Toplas*. In press. Available via `http://www.cwi.nl/~apt`.

Barklund, J. & Bevemyr, J. (1993), Prolog with arrays and bounded quantifications, *in* A. Voronkov, ed., 'Logic Programming and Automated Reasoning—Proc. 4th Intl. Conf.', LNCS 698, Springer-Verlag, Berlin, pp. 28–39.

Clark, K. L. (1978), Negation as failure, *in* H. Gallaire & J. Minker, eds, 'Logic and Databases', Plenum Press, New York, pp. 293–322.

De Schreye, D. & Decorte, S. (1994), 'Termination of logic programs: the never-ending story', *Journal of Logic Programming* **19-20**, 199–260.

Groenendijk, J. & Stokhof, M. (1991), 'Dynamic predicate logic', *Linguistics and philosophy* **14**(2), 39–101.

Hill, P. M. & Lloyd, J. W. (1994), *The Gödel Programming Language*, The MIT Press.

Honsberger, R. (1970), *Ingenuity in Mathematics*, Random House, Inc., New York.

Jaffar, J. & Lassez, J.-L. (1987), Constraint Logic Programming, *in* 'POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages', ACM, pp. 111–119.

Jaffar, J., Maher, M., Marriott, K. & Stuckey, P. (1998), 'The semantics of constraint logic programs', *Journal of Logic Programming* **37**(1-3), 1–46.

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

Kluźniak, F. & Miłkowska, M. (1997), 'Spill: A logic language for writing testable requirements specifications', *Science of Computer Programming* **28**(2 & 3), 193–223.

Kowalski, R. (1974), Predicate logic as a programming language, *in* 'Proceedings IFIP'74', North-Holland, pp. 569–574.

Lloyd, J. W. (1987), *Foundations of Logic Programming*, second edn, Springer-Verlag, Berlin.

Lloyd, J. W. & Topor, R. W. (1984), 'Making Prolog more expressive', *Journal of Logic Programming* **1**, 225–240.

Lobo, J., Minker, J. & Rajasekar, A. (1992), *Foundations of Disjunctive Logic Programming*, The MIT Press.

McAloon, K. & Tretkoff, C. (1995), 2LP: Linear programming and logic programming, *in* P. Van Hentenryck & V. Saraswat, eds, 'Principles and Practice of Constraint Programming', MIT Press, pp. 101–116.

Shoenfield, J. R. (1967), *Mathematical Logic*, Addison-Wesley, Reading, Massachusetts.

Tarski, A. (1933), *Pojęcie prawdy w językach nauk dedukcyjnych*, Towarzystwo Naukowe Warszawskie, Warszawa. In Polish. English version appeared in A. Tarski, *Logic, semantics, metamathematics: papers from 1923 to 1938*, Oxford, Clarendon, 1956.

Troelstra, A. S. & van Dalen, D. (1988), *Constructivism in Mathematics*, Studies in Logic and the Foundations of Mathematics, North-Holland Publ. Co., Amsterdam. Two vols.

Voronkov, A. (1992), Logic programming with bounded quantifiers, *in* A. Voronkov, ed., 'Logic Programming and Automated Reasoning—Proc. 2nd Russian Conference on Logic Programming', LNCS 592, Springer-Verlag, Berlin, pp. 486–514.

9. Appendix

9.1 Proof of the Soundness Theorem 3.3

The proof proceeds by induction on the lexicographic ordering on formulas which is defined in Definition 2.4. We carefully go through all inductive cases.

1. The case of the empty conjunction is trivial.

2. The first three of the four cases concerning atom A are obvious. It remains to deal with the last case, where atom A is an α-assignment $s = t$. Then either s or t is a variable which is not α-closed, say $s \equiv x$ with x not α-closed and t α-closed. The symmetrical case is analogous. The tree $[\![x = t \wedge \psi]\!]_\alpha$ is, apart from the root of degree one, identical to $[\![\psi]\!]_{\alpha \cup \{x/t^\alpha\}}$.

   If $[\![\psi]\!]_{\alpha \cup \{x/t^\alpha\}}$ contains a success leaf labelled by β, then by the induction hypothesis $\forall(\psi^\beta)$ is true. Since t is α-closed and β extends $\alpha \cup \{x/t^\alpha\}$, we have $(x = t)^\beta \equiv (x^\beta = t^\beta) \equiv (t^\alpha = t^\alpha)$. The last formula is true, so also $\forall((x = t \wedge \psi)^\beta)$ is true.

   If $[\![\psi]\!]_{\alpha \cup \{x/t^\alpha\}}$ is failed, then by the induction hypothesis $\exists(\psi^{\alpha \cup \{x/t^\alpha\}})$ is false. Note again that t is α-closed and let $x, x_1, \ldots, x_n$ be all the free variables of ψ that are not in the domain of α. (If $n = 0$ or if x does not occur in ψ, then the argument is even simpler.) Then we have $\exists((x = t \wedge \psi)^\alpha) \equiv \exists x, x_1, \ldots, x_n\ (x = t^\alpha \wedge \psi^\alpha(x, x_1, \ldots, x_n))$, which is logically equivalent to $\exists x_1, \ldots, x_n\ \psi^\alpha(t^\alpha, x_1, \ldots, x_n)) \equiv \exists(\psi^{\alpha \cup \{x/t^\alpha\}})$. It follows that $\exists((x = t \wedge \psi)^\alpha)$ is also false.

3. The case of $(\phi_1 \vee \phi_2) \wedge \psi$ uses the distributive law and the induction hypothesis applied to the the lexicographically smaller formulas $\phi_1 \wedge \psi$ and $\phi_2 \wedge \psi$.

4. The case of $(\phi_1 \wedge \phi_2) \wedge \psi$ uses the associativity of conjunction and the induction hypothesis applied to the the lexicographically smaller formulas ϕ_1 and $\phi_2 \wedge \psi$.

5. The case of $(\phi_1 \rightarrow \phi_2) \wedge \psi$ uses the logical equivalence of $\phi_1 \rightarrow \phi_2$ and $\neg\phi_1 \vee \phi_2$. If formula ϕ_1 is α-closed and $[\![\phi]\!]_\alpha$ is failed, then the argument is similar to the corresponding case of $\neg\phi \wedge \psi$ in the next case. The other case can be dealt with by applying the induction hypothesis to $\phi_1 \wedge (\phi_2 \wedge \psi)$.

6. For $\neg\phi \wedge \psi$ we distinguish three cases with respect to ϕ.

   - Formula ϕ is α-closed and $[\![\phi]\!]_\alpha$ is failed. Then, by the induction hypothesis, $\exists(\phi^\alpha)$ is false, so $\forall(\neg\phi^\alpha)$ is true. Since $[\![\neg\phi \wedge \psi]\!]_\alpha$ is, apart from the root of degree one, identical to $[\![\psi]\!]_\alpha$, we apply the induction hypothesis to ψ. If $[\![\psi]\!]_\alpha$ is failed, then $\exists(\psi^\alpha)$ is false, and hence $\exists((\neg\phi \wedge \psi)^\alpha)$ is false. If $[\![\psi]\!]_\alpha$ contains a success leaf β then β extends α and $\forall(\psi^\beta)$ is true. Note that $\forall(\neg\phi^\alpha)$ implies $\forall(\neg\phi^\gamma)$, for any γ extending α, *even if ϕ is not α-closed*. It follows that $\forall((\neg\phi \wedge \psi)^\beta)$ is true. Observe that we did not use the fact that ϕ is α-closed. So the proof remains valid under the first relaxation described in Subsection 9.3.
   - Formula ϕ is α-closed and $[\![\phi]\!]_\alpha$ contains at least one success leaf, labelled by an extension β of α. The tree $[\![\neg\phi \wedge \psi]\!]_\alpha$ consists of a root and a failure leaf in this case, so we have to show that $\exists((\neg\phi \wedge \psi)^\alpha)$ is false. By the induction hypothesis, $\forall(\phi^\beta)$ is true, and hence $\forall(\phi^\alpha)$ is true, as β is an extension of α and ϕ is α-closed. *This implication also holds if ϕ is not α-closed, provided that β does not contain any pair x/d where x is free in ϕ^α.* Consequently, $\exists(\neg\phi^\alpha)$ is false and hence also $\exists((\neg\phi \wedge \psi)^\alpha)$ is false. Observe that the proof remains valid under the second relaxation described in Subsection 9.3.
   - In all other cases there is nothing to prove as $[\![\neg\phi \wedge \psi]\!]_\alpha$ has then only *error* leaves.

7. For the case $\exists x\ \phi \wedge \psi$, assume that x is fresh with respect to ψ and some valuation α. It is convenient to make the possible occurrence of x in ϕ explicit by writing $\phi(x)$ for ϕ. Recall that apart form the root of degree one $[\![\exists x\ \phi(x) \wedge \psi]\!]_\alpha$ is identical to $[\![\phi(x) \wedge \psi]\!]_\alpha$.

   Assume $[\![\phi(x) \wedge \psi]\!]_\alpha$ contains a success leaf labelled by β. By applying the induction hypothesis to the lexicographically smaller formula $\phi(x) \wedge \psi$ we get that $\forall((\phi(x) \wedge \psi)^\beta)$ is true. It follows that $\forall((\exists x\ \phi(x) \wedge \psi)^\beta)$ is true. Some minor technicalities have been left to the reader here: the case in which x does not occur in the domain of β has to be settled by applying $(\forall x\ \phi(x)) \rightarrow (\exists x\ \phi(x))$ and not by inferring $\exists x\ \phi(x)$ from $\phi(x^\beta)$.

   Assume $[\![\phi(x) \wedge \psi]\!]_\alpha$ is failed. Then, again by the induction hypothesis, $\exists((\phi(x) \wedge \psi)^\alpha)$ is false. Since x does occur neither in α, nor in ψ, it follows that $\exists((\exists x\ \phi(x) \wedge \psi)^\alpha)$ is false. $\square$

9.2 Proof of the Completeness Theorem 3.4
(i) Suppose by contradiction that $[\![\phi]\!]_\alpha$ is not successful. Since this tree is determined, it is failed. By the Soundness Theorem 3.3 $\exists(\phi^\alpha)$ is false which is a contradiction.

(ii) Suppose by contradiction that $[\![\phi]\!]_\alpha$ is not failed. Since this tree is determined, it is successful. By the Soundness Theorem 3.3 for some β that extends α we have that $\forall(\phi^\beta)$ is true. This is a contradiction since the falsity of $\exists(\phi^\alpha)$ is equivalent to the truth $\forall(\neg\phi^\alpha)$ that implies the truth of $\forall(\neg\phi^\beta)$. $\square$

9.3 More liberal negation
In this subsection we show how in Definition 2.4 the restriction "ϕ is α-closed" in the case of the tree $[\![\neg\phi \wedge \psi]\!]_\alpha$ can be relaxed without losing soundness. There are basically two such relaxations.

First, observe by means of example that $\neg(0 = 1 \wedge x = y)$ is true, independent of the values of x and y. This observation can be generalized as follows. If $[\![\phi]\!]_\alpha$ is failed, then $[\![\neg\phi \wedge \psi]\!]_\alpha$ can be defined as the tree with a root of degree one and $[\![\psi]\!]_\alpha$ as its subtree, *even if ϕ is not α-closed.* In the proof of the Soundness Theorem we already accommodated for this relaxation, see Subsection 9.1.

Second, observe that the dual phenomenon also exists: $\neg(0 = 0 \vee x = y)$ is false, independent of the values of x and y. More generally, if $[\![\phi]\!]_\alpha$ contains a success leaf β not containing any pair x/d with x free in ϕ^α, then $[\![\neg\phi \wedge \psi]\!]_\alpha$ can be defined as the tree with a root of degree one and a failure leaf as its son, *even if ϕ is not α-closed.*

9.4 More liberal implication
The first and the second relaxation above can be both applied to the computation tree $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$, the first to the case in which the tree $[\![\phi_1]\!]_\alpha$ is failed, and the second to the case in which the tree $[\![\phi_1]\!]_\alpha$ contains a success leaf not containing any pair x/d with x free in ϕ_1^α.

There are several other ways to liberalize implication. The aim is to be more complete, that is, to yield more determined computation trees (without losing soundness, of course).

As a first example, consider the computation tree $[\![(0 = 1 \rightarrow x = 0) \wedge x < 1]\!]_\varepsilon$:

$$\begin{tikzcd}
(0 = 1 \rightarrow x = 0) \wedge x < 1,\ \varepsilon \arrow[d, dash] \\
x < 1,\ \varepsilon \arrow[d, dash] \\
\mathit{error}
\end{tikzcd}$$

This computation tree is not determined. In contrast, using the equivalence of $\phi_1 \rightarrow \phi_2$ and $\neg\phi_1 \vee \phi_2$, we get the computation tree $[\![(\neg(0 = 1) \vee x = 0) \wedge x < 1]\!]_\varepsilon$ is determined, as it contains a success leaf $\{x/0\}$:

$$
\begin{array}{c}
(\neg(0=1)\ \vee\ x=0)\ \wedge\ x<1,\ \varepsilon \\
\swarrow \qquad\qquad \searrow \\
\begin{array}{cc}
\neg(0=1)\ \wedge\ x<1,\ \varepsilon & x=0\ \wedge\ x<1,\ \varepsilon \\
| & | \\
x<1,\ \varepsilon & x<1,\ \{x/0\} \\
| & | \\
\mathit{error} & \{x/0\}
\end{array}
\end{array}
$$

The above example shows that $\neg\phi_1 \vee \phi_2$ can be "more complete" than $\phi_1 \to \phi_2$, although in some cases the disjunction involves unnecessary branching in the computation tree. As an example of the latter phenomenon, compare the computation trees for $(0=1 \to 0=0) \wedge \psi$ and $(\neg(0=1) \vee 0=0) \wedge \psi$:

$$
\begin{array}{c}
(0=1 \to 0=0)\ \wedge\ \psi,\ \alpha \\
| \\
[\![\psi]\!]_\alpha
\end{array}
\qquad\qquad
\begin{array}{c}
(\neg(0=1)\ \vee\ 0=0)\ \wedge\ \psi,\ \alpha \\
\swarrow \qquad\qquad \searrow \\
\begin{array}{cc}
\neg(0=1)\ \wedge\ \psi,\ \alpha & 0=0\ \wedge\ \psi,\ \alpha \\
| & | \\
[\![\psi]\!]_\alpha & [\![\psi]\!]_\alpha
\end{array}
\end{array}
$$

As a second example, consider the computation tree $[\![(x=0 \to x<1]\!]_\varepsilon$, which is not determined since $x=0$ is not ε-closed:

$$
\begin{array}{c}
x=0 \to x<1,\ \varepsilon \\
| \\
\mathit{error}
\end{array}
$$

One would like to have this tree succeed with $\{x/0\}$. (The fact that $\{x/1\}$ is also a solution is beyond the scope of our method, since $[\![\neg(x=0)]\!]_\varepsilon$ is not determined.) For this the equivalence of $\phi_1 \to \phi_2$ and $\neg\phi_1 \vee \phi_2$ does not help, as the computation tree $[\![\neg(x=0) \vee x<1]\!]_\varepsilon$ is not determined either:

$$
\begin{array}{c}
\neg(x=0)\ \vee\ x<1,\ \varepsilon \\
\swarrow \qquad\qquad \searrow \\
\begin{array}{cc}
\neg(x=0),\ \varepsilon & x<1,\ \varepsilon \\
| & | \\
\mathit{error} & \mathit{error}
\end{array}
\end{array}
$$

Note that the left subtree ends with *error* since $[\![x=0]\!]$ succeeds with $\{x/0\}$. Liberal negation does not help us any further here.

In order to have $[\![(x=0 \rightarrow x<1]\!]_\varepsilon$ succeed it is necessary to transfer the valuation of the success leaf of the antecedent, i.e. $\{x/0\}$, to the consequent. Thus we are tempted to consider $\neg\phi_1 \vee (\phi_1 \wedge \phi_2)$ as a more useful logical equivalent of $\phi_1 \rightarrow \phi_2$ than $\neg\phi_1 \vee \phi_2$. The conjunction $\phi_1 \wedge \phi_2$ has the desired effect on the transfer of valuations. Indeed the following computation tree is successful:

$$\begin{tikzcd}
& \neg(x=0) \vee (x=0 \wedge x<1),\ \varepsilon \arrow[dl, dash] \arrow[dr, dash] & \\
\neg(x=0),\ \varepsilon \arrow[d, dash] & & x=0 \wedge x<1,\ \varepsilon \arrow[d, dash] \\
\mathit{error} & & x<1,\ \{x/0\} \arrow[d, dash] \\
& & \{x/0\}
\end{tikzcd}$$

In combination with liberal negation the computation tree $[\![((x=0 \wedge x=1) \rightarrow 0=1) \wedge \psi]\!]_\varepsilon$ has now the following form:

$$\begin{tikzcd}
& (\neg(x=0 \wedge x=1) \vee (x=0 \wedge x=1 \wedge 0=1)) \wedge \psi,\ \varepsilon \arrow[dl, dash] \arrow[dr, dash] & \\
(\neg(x=0) \wedge x=1) \wedge \psi,\ \varepsilon \arrow[d, dash] & & x=0 \wedge x=1 \wedge 0=1 \wedge \psi,\ \varepsilon \arrow[d, dash] \\
[\![\psi]\!]_\varepsilon & & x=1 \wedge 0=1 \wedge \psi,\ \{x/0\} \arrow[d, dash] \\
& & \mathit{fail}
\end{tikzcd}$$

Note that, if $[\![\psi]\!]_\varepsilon$ is failed, the "guard" $x=0 \wedge x=1$ prevents $[\![\psi]\!]_\varepsilon$ to be computed twice, even if we replace $0=1$ by $0=0$. On the other hand, this guard also prevents successful computations, such as in $[\![((x=0 \wedge x=1) \rightarrow x=0) \wedge x<1]\!]_\varepsilon$, where the solution $\{x/0\}$ is missed when $\neg\phi_1 \vee (\phi_1 \wedge \phi_2)$ is used instead of $\neg\phi_1 \vee \phi_2$ for $\phi_1 \rightarrow \phi_2$.

The above example shows that $\neg\phi_1 \vee (\phi_1 \wedge \phi_2)$ is not always "more complete" than $\neg\phi_1 \vee \phi_2$. Thus we are led to consider $\neg\phi_1 \vee \phi_2 \vee (\phi_1 \wedge \phi_2)$ as a third logical equivalent of $\phi_1 \rightarrow \phi_2$, in an attempt to collect all the successes of $\neg\phi_1 \vee \phi_2$ and $\neg\phi_1 \vee (\phi_1 \wedge \phi_2)$. Indeed this works for the successes, but not for the failures, as the following delicate example shows.

Consider $[\![\phi_1 \rightarrow \phi_2]\!]_\varepsilon$ with $0=0 \vee x<1$ for ϕ_1 and $0=1$ for ϕ_2. Then both $[\![\neg\phi_1]\!]_\varepsilon$ and $[\![\phi_2]\!]_\varepsilon$ are failed, but $[\![\phi_1 \wedge \phi_2]\!]_\varepsilon$ has an *error* leaf due to the disjunct $x<1$. This means that $[\![\neg\phi_1 \vee \phi_2]\!]_\varepsilon$ is failed, whereas neither $[\![\neg\phi_1 \vee (\phi_1 \wedge \phi_2)]\!]_\varepsilon$ nor $[\![\neg\phi_1 \vee \phi_2 \vee (\phi_1 \wedge \phi_2)]\!]_\varepsilon$ is determined.

From the above we can draw the following conclusions:

- Liberal negation is always an improvement for implication;

- For finding successes, use $\phi_1 \to \phi_2 \equiv \neg\phi_1 \vee \phi_2 \vee (\phi_1 \wedge \phi_2)$;
- For failures, use $\phi_1 \to \phi_2 \equiv \neg\phi_1 \vee \phi_2$.