PRECOMPLETE NEGATION AND UNIVERSAL QUANTIFICATION

by

Paul J. Voda

Technical Report 86-9

April 1986

Precomplete Negation And Universal Quantification.

Paul J. Voda

Department of Computer Science, The University of British Columbia,
Vancouver, B.C. V6T 1W5, Canada.

ABSTRACT

This paper is concerned with negation in logic programs. We propose to extend negation as failure by a stronger form of negation called precomplete negation. In contrast to negation as failure, precomplete negation has a simple semantic charaterization given in terms of computational theories which deliberately abandon the law of the excluded middle (and thus classical negation) in order to attain computational efficiency. The computation with precomplete negation proceeds with the direct computation of negated formulas even in the presence of free variables. Negated formulas are computed in a mode which is dual to the standard positive mode of logic computations. With negation as failure the formulas with free variables must be delayed until the latter obtain values. Consequently, in situations where delayed formulas are never sufficiently instantiated, precomplete negation can find solutions unattainable with negation as failure. As a consequence of delaying, negation as failure cannot compute unbounded universal quantifiers whereas precomplete negation can. Instead of concentrating on the model-theoretical side of precomplete negation this paper deals with questions of complete computations and efficient implementations.

April 1986

Precomplete Negation And Universal Quantification.

Paul J. Voda

Department of Computer Science, The University of British Columbia,
Vancouver, B.C. V6T 1W5, Canada.

1. Introduction.

Logic programming languages based on Prolog experience significant difficulties with negation. There are many Prolog implementations with unsound negation where the computation succeeds with wrong results. Some Prologs, most notably IC-Prolog and MU-Prolog [3,12], have sound negation. The basic strategy of computing the negation $\neg \mathbf{A}$ is to compute the formula $\mathbf{A}$. When the computation fails the formula $\neg \mathbf{A}$ succeeds, and vice versa. This is the well known negation as failure [2]. Such a negation is implemented in the sound Prologs by delaying the negated formula until the values of free variables in it are sufficiently known. Then the formula is reduced to truth or falsehood.

Quite often we want to compute a negated formula with some variables existentially or universally bound in it. Here Prolog's lack of expressive power comes to light. Although, by introducing auxiliary predicates, one can express any form of quantification [9], in the absence of a preprocessor, the additional predicates and the double negation certainly do not increase readability. Quite a few researchers, the present author among them [18,19] proposed computations with explicit quantifiers. The latest attempt within the framework of Prolog is in the form of NU-Prolog [14]. With negation as failure it is impossible to compute general universal quantifiers. The formula $\forall \mathbf{x} \mathbf{A}$ is computed as $\neg \exists \mathbf{x} \neg \mathbf{A}$, where the inner negation cannot possibly get instantiated unless the quantifier is bounded, i.e. there are only finitely many $\mathbf{x}$ satisfying $\mathbf{A}$.

The restrictions of negation as failure spawned quite a vivid research in this area. If a logic programming language is embedded within a full theorem prover, such as TABLOG [10] or F-Prolog [17], then classical negation can be attained. We have argued in a previous paper [20] that classical negation is very costly in terms of implementation efficiency when doing logic *computations* rather than full theorem proving. We understand logic computations as a restricted form of theorem proving. This means in practice that we abandon the law of the excluded middle for formulas of the form $\mathbf{P}(\mathbf{c}) \vee \neg \mathbf{P}(\mathbf{c})$. We restrict the excluded middle only to certain formulas which we can resolve without costly computations.

We can illustrate the cost of classical negation as follows. Suppose we are trying to prove a formula in Herbrand normal form (all free variables are implicitly existentially quantified)

$$\mathbf{A}_1 \vee \mathbf{A}_2 \vee \cdots \vee \mathbf{A}_n. \tag{1}$$

We have to look for all possible pairs of formulas of the form $\mathbf{A}_i \equiv \mathbf{P}(\mathbf{a}) \,\&\, \mathbf{C}$ and $\mathbf{A}_j \equiv \neg \mathbf{P}(\mathbf{b}) \,\&\, \mathbf{D}$. When we succeed in unifying the term $\mathbf{a}$ with the term $\mathbf{b}$ into the term $\mathbf{c}$ we can employ the tautology

$$\mathbf{C}' \,\&\, \mathbf{D}' \rightarrow \mathbf{P}(\mathbf{c}) \,\&\, \mathbf{C}' \vee \neg \mathbf{P}(\mathbf{c}) \,\&\, \mathbf{D}'$$

which is dual to the cut and follows from the excluded middle $\mathbf{P}(\mathbf{c}) \vee \neg \mathbf{P}(\mathbf{c})$. Using this tautology we try to prove the formula

$$\mathbf{A}_1 \vee \mathbf{A}_2 \vee \cdots \vee \mathbf{A}_n \vee (\mathbf{C}' \,\&\, \mathbf{D}').$$

If we succeed in proving the last formula we have proven (1). Readers used to proofs by refutation will note that the process just described is dual to resolution. The price of classical negation lies in the enormous cost of trying out all candidates for resolution. The reason for the efficiency of Prolog (without negation) is that the Horn clause based proofs restrict the candidates of resolution only to to the heads of clauses.

By abolishing the law of the excluded middle we have to expand both negative and positive calls of predicates until we obtain a formula we can decide directly. In [20] we have formulated a logic theory which we can call a *computational* logic. The theory is called CTP (Computational Theory of Pairs) and although it pertains to pairs (S-expressions of LISP) the principles can be applied to any other universe. We have expressed our conviction that this is probably the most one can reasonably expect to compute efficiently. We did not present any computational algorithm beyond observing that the computation of negations must proceed even in the presence of free variables by the employment of *dual* rules. This means that conjunctions are computed as disjunctions, identities $\mathbf{a} = \mathbf{b}$ as inequalities $\mathbf{a} \neq \mathbf{b}$, and so on. Consider the predicate

$$M(x) \leftrightarrow x \neq 3 \;\&\; x \neq 2. \tag{2}$$

Negation as failure will not be able to compute the formula $\neg M(y) \;\&\; y \neq 3$ because the variable y will never get instantiated. We, however, shall compute the negative call $\neg M(y)$ and end up with the solution $y = 2$ (see section 5).

This paper presents an algorithm for the computation with such a negation. We propose to call the extended negation *Precomplete Negation*, i.e. almost complete negation, as it is more than negation by failure and *deliberately* less than the complete classical negation. With precomplete negation we expand negated formulas even in the presence of free variables. As a consequence we have universal quantification over infinite domains, something impossible with negation as failure.

The algorithm is essentially an extension of our algorithms computing negation as failure within the full first order language [18,19]. These algorithms deal with a theory of pairs, TP. They were extended by Andrews in his thesis [1] to the full precomplete negation over pairs. We present precomplete negation in a general framework applicable to many concrete theories. As an illustration we shall use examples from a simple theory of natural numbers. The soundness of our computations is immediately seen from the form of computational rules.

We have the feeling that the readers of our previous papers have perceived our computational methods which rely on the rewriting of formulas within a first order language as a transformation of trees and therefore not readily implementable. We want to correct this mistaken impression. Actually, we have an implementation employing choice points on the stack. This is the standard method of implementing Prolog as invented by Warren [21]. As we proceed with the presentation of our method we comment on the mapping of it into the framework of the Warren model. As it happens there is no additional price in terms of efficiency for programs without negation. Actually even negations can be implemented efficiently. Only universal quantifiers require extra computations. Note that this price will be paid for the computation of negated predicates containing unbounded existential quantifiers. This is because the last turn into universal quantifiers under negation.

The completeness of precomplete negation with respect to computational logic is, however, achieved only if disjunctions in the positive context (conjunctions in the negative context) are computed in the breadth first manner not achievable with the method of choice points. The method of choice points treats the connectives sequentially.

Finally, to illustrate the increased power of precomplete negation as opposed to negation as failure we devote section (5) to the computation of examples not computable with the latter method.

The inspiring discussions of the author with Jamie Andrews, Karl Abrahamson, and Peter Ludemann which greatly stimulated this research are gladly acknowledged.

2. Computational Theories.

The language of formulas we shall compute is the ordinary first order language with identity (see for instance [16]). Formulas are composed from atomic predicates by negations, disjunctions, conjunctions, universal and existential quantifiers. Certain predicate symbols are designated as *basic*. The predicate symbol of identity $=$ as well as the nullary predicate symbols $\mathbf{T}$ and $\mathbf{F}$ denoting truth and falsehood are always basic. Formulas built (by the employment of connectives and quantifiers) from only basic atomic predicates are called *basic* formulas. Whether there are additional basic predicates than the above three depends on the concrete computational theory but we always require that basic formulas are decidable. The existence of the decision method for the basic formulas obviates the need for an explicit axiomatization of properties of basic predicates.

All non-basic predicate symbols, called *defined* predicates, must have a single axiom of the form

$$\mathbf{P}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \leftrightarrow \mathbf{A}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \tag{3}$$

where $\mathbf{A}$ is an arbitrary formula containing at most the variables $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_n$ free. Although Prolog predicates are not given by equivalences their *completion*, which is necessary for the negative computation in Prolog, has the form (3). We are not concerned here with interpretations but we require that the axioms are consistent. This means that there is a classical model satisfying the properties of basic predicates (as expressed by the decision algorithm). The model must also satisfy the axioms (3) for all defined predicates.

We could have alternatively characterized the true formulas of Computational Logic by provability in a Gentzen-like sequent calculus. The theory CTP is presented in such a style. The rules of introduction (both on the left and right) of connectives ($\neg$, $\&$, $\vee$) as well as of quantifiers ($\forall$, $\exists$) are adopted in the standard form of Gentzen's calculus. For each predicate $\mathbf{P}$ defined by the formula (3) we add two new rules for predicate introduction left and right.

$$\frac{\mathbf{A}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n), \Gamma \rightarrow \Delta}{\mathbf{P}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n), \Gamma \rightarrow \Delta}\ (\mathbf{P}\text{-L}) \qquad \frac{\Gamma \rightarrow \Delta, \mathbf{A}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)}{\Gamma \rightarrow \Delta, \mathbf{P}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)}\ (\mathbf{P}\text{-R}).$$

The predicate rules allow both positive and negative expansion of predicate calls by their replacement with the bodies after the substitutions of terms $\mathbf{a}_i$ for the corresponding variables $\mathbf{x}_i$.

Gentzen's axioms $\mathbf{A}, \Gamma \rightarrow \Delta, \mathbf{A}$ express the excluded middle and thus will not be retained. We adopt instead as axioms all sequents $\Gamma \rightarrow \Delta$ such that when we remove from it all non-basic formulas the remaining basic formulas $\Gamma' \rightarrow \Delta'$ form a true sequent. For instance the sequent

$$P(6), z = 3 \rightarrow R(x, y) \,\&\, x = y, z = 1, z \neq 1$$

is an axiom because it becomes a true sequent without the non-basic formulas

$$z = 3 \rightarrow z = 1, z \neq 1.$$

The existence of a decision procedure for the basic formulas allows us to recognize a true sequent composed entirely of basic formulas. Consequently, we can always recognize an axiom of computational logic.

Sequents unprovable within this setup are considered not true. If we characterize the computational logic in this way we obtain truth value gaps, i.e. certain formulas may fail to obtain a truth value. It is always possible to construct a model (albeit not a classical one) characterizing as true formulas exactly those which are provable, and as false exactly those which are refutable. Such models have been developed by Kripke [8]. For an application of Kripke's models to logic programming see [5]. Another application of Kripke's models relevant to computer science which goes beyond the first order language we employ here is the natural deduction set theory of Gilmore [6]. The reader interested in details of how to build semantics to reflect a proof theory and vice versa is referred to this paper.

It is not the goal of the present paper to investigate the model theory of computational logic but rather to present an efficient algorithm for computing its formulas. The reason why we investigate the question of efficient computability before the general meta-theoretic questions is very simple. Computational logic is theoretically interesting only if it can be shown to be practically usable. Without the concern for usability one can invent thousands of non-standard logics. For the sake of simplicity we henceforth assume that our models can be constructed classically, i.e. that the predicate axioms (3) are consistent. Under this assumption we show that our computations are sound, i.e. that whatever is computed as a solution to a formula classically satisfies the formula. We are preparing a paper showing completeness, i.e. that whatever is true in the non-classically formulated computational logic can be computed by our method of precomplete negation.

Throughout this paper we shall use as an example natural numbers. Apart from $=$, **T**, and **F** we do not require any additional basic predicates. The terms of our language are either variables or numerals such as 2, 5, etc. All other predicate symbols must be defined by the axioms of the form (3) subject to the restriction that they have an interpretation in the domain of natural numbers. Although the theories thus obtained are very weak they will be sufficient to illustrate the computations. The reason for the weakness is that we do not have any function symbols (such as the successor) so we cannot build the arithmetic even though our axioms for defined predicates permit recursion.

We could have easily used the successor function and the constant 0 as the starting function symbols. We could have also included $<$ as the basic predicate and we would still have a decision method [see for instance 11]. Now some of arithmetic can be developed by the definition of predicates of addition and multiplication. We have decided against examples in the stronger arithmetic because the description of the decision method would be longer. The logic computations, as described below, are essentially independent of the concrete form of the decidable subpart.

3. Environments.

Consider the binary predicate K over natural numbers:

$$K(x,\, y) \leftrightarrow x = 1 \;\&\; (y = 2 \vee y = 3) \vee x \neq 1 \;\&\; (y = 1 \vee y = 3). \tag{4}$$

The formula $\mathbf{a} \neq \mathbf{b}$ is an abbreviation for $\neg\, \mathbf{a} = \mathbf{b}$. The formula defining the predicate K is a basic formula. The formula $\forall z\, K(z,\, v)$ is not. A computation of this formula might replace the application of the predicate $K(z,v)$ by its definition (substituting for the variables x and y). The new formula would be decidable, and equivalent to $v = 3$, yielding the result of the computation.

This is essentially how we shall compute. We shall repeatedly replace applications of defined predicates by their definitions until we obtain a decidable formula. We shall, however, not employ substitution as it is costly to implement. We shall instead employ special basic formulas called *environments* which record the computed information about the values of variables. In the above example, we construct the environment $x = z \;\&\; y = v$ and compute the formula on the right-hand-side of (4) directly without any changes to it. Computer scientists would say that this environment expresses the *bindings* for the variables x and y. We prefer to say that environments record *constraints* on the variables. Bindings are given only by identities, constraints can be given by any basic predicates in both positive and negated form.

The state of a computation can be visualized by the formula $\mathbf{E} \;\&\; \mathbf{A}$ where $\mathbf{A}$ is the formula to be computed and $\mathbf{E}$ is the environment constraining the free variables of $\mathbf{A}$. The computation of $\mathbf{A}$ starts in the state $\mathbf{T} \;\&\; \mathbf{A}$ where the environment $\mathbf{T}$ does not constrain the free variables of $\mathbf{A}$ in any way. Environments are shipped around the computed formula and in the process they are transformed and amended. At the end of the computation environments give solutions. The computation proceeds by a transformation of the formula into a disjunction of the form

$$\mathbf{E}_1 \vee \mathbf{E}_2 \vee \;\cdots\; \vee \mathbf{E}_n \vee \mathbf{A}' \tag{5}$$

where $\mathbf{A}'$ is a not yet converted formula. Those environments $\mathbf{E}_i$ which can be satisfied give solutions to the formula $\mathbf{A}$. Environments used with the disjunctive form are called *disjunctive* environments.

As the computation reaches a negation it switches into the construction of a conjunctive normal form. The environments used for this are called *conjunctive* environments and are negations of disjunctive environments. Nested negations continue to switch between disjunctive and conjunctive environments. Whenever a computation reaches an existential quantifier we switch into the disjunctive mode (unless we are already in this mode). Within universal quantifiers we switch into the conjunctive mode.

Our method works for any form of environments subject to the restriction that they are basic formulas. As will be seen below, our ability to deal with the quantifiers rests on their *eliminability*. This means that for any environment $\mathbf{E}$ and any variable $\mathbf{x}$ we can always find an environment $\mathbf{E}'$

such that $\exists x\mathbf{E} \leftrightarrow \mathbf{E}'$ (or $\forall x\mathbf{E} \leftrightarrow \mathbf{E}'$ for universal quantifiers). The eliminability of quantifiers comes automatically with the decision procedure for basic predicates.

In the description of the algorithm for precomplete negation we do not use such a general form of environments. We put additional constraints on their form. The additional constraints guarantee a straightforward implementation of environments on the computation stack. If the reader finds the additional constraints too prohibitive he is welcome to rephrase the algorithm with his own environments. As it happens, only the rules of basic predicate absorption (21-22) and of quantifier elimination (37-46) depend on the concrete form of environments.

The additional constraint on the form of environments is that the **conjunctive** environments be of the form

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n \bullet\!\!\rightarrow \mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_m \tag{6}$$

where the formulas $\mathbf{A}_k$, $\mathbf{B}_k$ are atomic applications of basic predicates. The operator $\bullet\!\!\rightarrow$ can be viewed either as implication or as Gentzen's sequent symbol. Indeed, the environment (6) is just an abbreviation for the formula

$$\neg\,\mathbf{A}_1 \vee \neg\,\mathbf{A}_2 \vee \cdots \vee \neg\,\mathbf{A}_n \vee \mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_m \tag{7}$$

which can be also written as

$$\mathbf{A}_1 \,\&\, \mathbf{A}_2 \,\&\, \cdots \,\&\, \mathbf{A}_n \rightarrow \mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_m.$$

We adopt the usual sequent conventions for the environments. We shall use Greek letters to abbreviate finite sequences of basic atomic predicates: $\Gamma\bullet\!\!\rightarrow\Delta$. We allow Γ, Δ, or both to be empty. Thus $\bullet\!\!\rightarrow\Delta$ stands for $\mathbf{T}\bullet\!\!\rightarrow\Delta$, $\Gamma\bullet\!\!\rightarrow$ stands for $\Gamma\bullet\!\!\rightarrow\mathbf{F}$, and $\bullet\!\!\rightarrow$ for $\mathbf{F}$. The sequent operator $\bullet\!\!\rightarrow$ binds stronger than $\&$ and $\&$ binds stronger than $\vee$.

For the **disjunctive** mode of computation we use the negations of (6) in the form $\neg\,(\Gamma\bullet\!\!\rightarrow\Delta)$. This will be abbreviated as $\Gamma\bullet\!\!\not\rightarrow\Delta$. The reader will note that a disjunctive environment is composed entirely of conjunctions whereas the conjunctive one is composed of disjunctions.

It is advantageous to keep the environments in a *normalized* form. As will be seen below, the normalized form guarantees a fast execution of the rules for atomic absorption and quantifier elimination (21-22,37-46). For instance, the environments, i.e. the systems of equations and inequalities, used in Prolog2 [4] have a special *reduced* form.

The normalized form of the environments for our theory of natural numbers satisfies two conditions.

> **No Circularity Condition**: A normalized environment does not contain the possibly commuted identities
>
> $$\mathbf{a}_1 = \mathbf{a}_2, \mathbf{a}_2 = \mathbf{a}_3, \ldots, \mathbf{a}_n = \mathbf{a}_1$$
>
> for $n \geq 1$ at the same time. Note that this prohibits also $\mathbf{a} = \mathbf{a}$.
>
> **Single Value Condition**: If an normalized environment contains the identity $\mathbf{x} = \mathbf{a}$ in the antecedent then there is no additional identity $\mathbf{x} = \mathbf{b}$ in the environment.

For example, the environment

$$x = 3, y = z, z = 5 \bullet\!\!\rightarrow x = 5$$

violates the second condition. It is, however, equivalent to the normalized environment

$$x = 3, y = z, z = 5 \bullet\!\!\rightarrow.$$

The negative unnormalized environment

$$x = y, y = z \bullet\!\!\not\rightarrow x = z$$

is equivalent to another unnormalized environment

$$x = y, y = z \bullet\!\!\not\rightarrow y = z$$

which is again equivalent to the environment violating the first condition

$$z = y, y = z \bullet\!\!\not\rightarrow z = z$$

which is finally equivalent to the normalized environment $\bullet\!\!\rightarrow$. This last example illustrates how the decision procedure takes care of the transitivity of identity.

There is a very simple implementation of normalized disjunctive environments $\Gamma \bullet\!\!\nrightarrow \Delta$. Each variable is allocated on the stack. The identities of the form $\mathbf{x} = \mathbf{a}$ in the antecedent Γ give bindings to the variables. The single value condition prohibits an additional constraint on $\mathbf{x}$. The variables for which there is no binding of the form $\mathbf{x} = \mathbf{a}$ in the antecedent are constrained to be different from $\mathbf{b}$ for every occurrence of the identity $\mathbf{x} = \mathbf{b}$ in the consequent. The circularity condition is implemented by an ordering of all variables and permitting the identity $\mathbf{x} = \mathbf{y}$ in an environment only if the variable $\mathbf{x}$ is after the variable $\mathbf{y}$. The variables closer to the top of the stack are considered to come after the variables deeper in the stack. So we allow only pointers pointing from the top of the stack towards the bottom. Conjunctive environments have exactly the same representation as the disjunctive ones, we simply maintain a flag recording the fact that we are in the conjunctive mode. Actually, it is possible to compile away the conjunctive mode by the employment of de Morgan laws by a compiler.

The computation of a formula $\mathbf{A}$ which does not contain any environments is commenced in the disjunctive mode by putting the formula in the context of the most inclusive environment $\bullet\!\!\nrightarrow \,\&\, \mathbf{A}$. Since $\mathbf{A} \leftrightarrow \bullet\!\!\nrightarrow \,\&\, \mathbf{A}$ we do not change the solutions of $\mathbf{A}$. At every subsequent stage of the computation we apply a rule of computations to the current form $\mathbf{A}_k$ of the computed formula. The rules of computation have always the form $\mathbf{B} \leftrightarrow \mathbf{C}$ with at least one environment in the formula $\mathbf{B}$. The next formula $\mathbf{A}_{k+1}$ is then obtained by the replacement of one occurrence of the formula $\mathbf{B}$ by the formula $\mathbf{C}$ in the formula $\mathbf{A}_k$. The environments serve as markers indicating positions in the computed formula which can be replaced. As all rules of computation are valid in the model we do not change the set of solutions. We leave it to the reader to check the almost obvious validity of our rules of computation. The soundness of our computations is obvious since we never change the meaning of the computed formula.

The disjunctive computation terminates when either the formula $\bullet\!\!\rightarrow$ or $\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}'$ with $\Gamma \bullet\!\!\nrightarrow \Delta$ satisfiable is reached. In the former case the original formula is false and does not have any solutions. In the latter case the disjunctive environment $\Gamma \bullet\!\!\nrightarrow \Delta$, if satisfiable, gives one set of solutions. The formula $\mathbf{A}'$ can be viewed as the *backtrack* formula (it will necessarily contain environments) and its reduction may supply additional solutions.

There is always at most one rule applicable for each occurrence of an environment in the computed formula. It is, however, possible to apply different rules to the different occurrences of environments. If the computing agent always selects the leftmost applicable rule we have a *sequential* computation which can be efficiently implemented by the method of choice points.

The sequential computation, however, does not guarantee completeness with respect to the computational logic. In order to achieve completeness we have to try all applicable rules either in parallel or in a breadth first manner. The presentation of rules of computation in the next section is biased towards sequential computation. We assume that the left operand of a binary connective is reduced first. We could have corrected the situation by adding new rules taking into the account the situation where the right operand of a disjunction or conjunction reduces first. The interested reader is referred to [18] where we present a parallel method of computation of binary connectives. Alternatively, since with parallelism we already have nondeterministic computation, we can simply add the rules $\mathbf{A} \,\&\, \mathbf{B} \leftrightarrow \mathbf{B} \,\&\, \mathbf{A}$ as well as $\mathbf{A} \vee \mathbf{B} \leftrightarrow \mathbf{B} \vee \mathbf{A}$ which will bring the computed formula into the desired form.

4. Computation Rules.

The reader will observe that the disjunctive environment $\Gamma \bullet\!\!\nrightarrow \Delta$ always travels down the computed formula in the context $\Gamma \bullet\!\!\nrightarrow \Delta \,\&\, \mathbf{A}$. When the computation switches to the conjunctive mode, the environment $\Gamma \bullet\!\!\rightarrow \Delta$ travels downwards in the context $\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{A}$. This continues until the basic atomic formula $\mathbf{A}$ is reached. This formula is then absorbed into the environment (see rules 21,22). The environment starts to *retract* after the absorption, either in the form of truth value propagation or in the form of backtracking.

The goal of the disjunctive mode is to obtain the context $\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}$, the goal of the conjunctive mode is to attain the context $\Gamma \bullet\!\!\rightarrow \Delta \;\&\; \mathbf{A}$. The reader will note that the computation modes are completely dual. He is advised to try to understand the disjunctive mode (the one with $\Gamma \bullet\!\!\nrightarrow \Delta$) first. This is because he may be more familiar with it since it is the only mode used in the computations with negation as failure.

For each mode there is one downward rule applicable depending on the outermost operator of the computed formula. This formula never contains an environment. We leave it to the reader to check that the rules are valid in first order logic with identity.

Conjunction Rules. The formulas **A** and **B** do not contain environments.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; (\mathbf{A} \;\&\; \mathbf{B}) \leftrightarrow (\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{A}) \;\&\; \mathbf{B} \tag{8}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee (\mathbf{A} \;\&\; \mathbf{B}) \leftrightarrow (\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{A}) \;\&\; (\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{B}). \tag{9}$$

Disjunction Rules. The formulas **A** and **B** do not contain environments.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; (\mathbf{A} \vee \mathbf{B}) \leftrightarrow (\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{A}) \vee (\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{B}) \tag{10}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee (\mathbf{A} \vee \mathbf{B}) \leftrightarrow (\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{A}) \vee \mathbf{B}. \tag{11}$$

The rule (8) (and dually the rule (11)) does not require any special implementation. The rule (10) (and dually the rule (9)) can be efficiently implemented by laying down a choice point without any copying of the environment. All subsequent changes to the environment must be, however, recorded (trailed) so the same environment can be restored upon failure.

Negation Rules. The formula **A** does not contain environments.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \neg\, \mathbf{A} \leftrightarrow \neg\, (\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{A}) \tag{12}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \neg\, \mathbf{A} \leftrightarrow \neg\, (\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{A}). \tag{13}$$

The computation mode is simply switched without any change in the environment.

Existential Quantifier Rules. The formula **A** does not contain environments.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \exists \mathbf{x} \mathbf{A} \leftrightarrow \exists \mathbf{x}'(\Gamma' \bullet\!\!\nrightarrow \Delta' \;\&\; \mathbf{A}') \tag{14}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \exists \mathbf{x} \mathbf{A} \leftrightarrow \Gamma \bullet\!\!\rightarrow \Delta \vee \exists \mathbf{x}'(\Gamma' \bullet\!\!\nrightarrow \Delta' \;\&\; \mathbf{A}'). \tag{15}$$

Universal Quantifier Rules. The formula **A** does not contain environments.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \forall \mathbf{x} \mathbf{A} \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \forall \mathbf{x}'(\Gamma' \bullet\!\!\rightarrow \Delta' \vee \mathbf{A}') \tag{16}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \forall \mathbf{x} \mathbf{A} \leftrightarrow \forall \mathbf{x}'(\Gamma' \bullet\!\!\rightarrow \Delta' \vee \mathbf{A}'). \tag{17}$$

When an environment enters the scope of a quantifier some of its free variables may become bound. The renaming of the bound variable **x** to **x'** prevents this problem. The renaming is necessary only within the logical formalism. In an implementation a new bound variable **x** without any constraints on it is created on the top of the stack. A disjunctive environment entering the existential quantifier (14), and dually a conjunctive environment entering the universal quantifier (17) poses no problem. The rules (15) and (16) require a change of the computation mode. This is necessary so the quantifier elimination works smoothly (the rules 37-46). The change of the disjunctive to the conjunctive mode (16) relies on the tautology

$$\mathbf{B} \;\&\; \mathbf{C} \leftrightarrow \mathbf{B} \;\&\; (\neg\, \mathbf{B} \vee \mathbf{C}).$$

The rule (15) is dual. The environment entering the quantifier is retained outside of the quantifier. An implementation does not have to copy the environment provided it will be able to recover it from the trailing information upon quantifier elimination.

There are two different actions when a descending environment reaches an atomic formula: either a defined predicate is called or a basic predicate is absorbed into the environment. The calls of defined predicates rely on the following properties

$$\begin{aligned} &\exists \mathbf{x}(x = \mathbf{a} \;\&\; \mathbf{A}(\mathbf{x})) \leftrightarrow \mathbf{A}(\mathbf{a}) \\ &\forall \mathbf{x}(\mathbf{x} = \mathbf{a} \rightarrow \mathbf{A}(\mathbf{x})) \leftrightarrow \mathbf{A}(\mathbf{a}). \end{aligned} \tag{18}$$

The tautologies hold only if the term **a** does not contain the variable **x**. Assuming that the predicate $\mathbf{P}$ is defined by the defining axiom (3) the following rules apply.

Predicate Call Rules. The predicate $\mathbf{P}$ is defined by (3) and the formula $\mathbf{A}$ does not contain environments.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{P}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n) \leftrightarrow$$
$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \exists \mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_n(\mathbf{x}'_1 = \mathbf{a}_1 \;\&\; \mathbf{x}'_2 = \mathbf{a}_2 \;\&\; \cdots \;\&\; \mathbf{x}'_n = \mathbf{a}_n \;\&\; \mathbf{A}') \tag{19}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{P}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n) \leftrightarrow$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \forall \mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_n(\mathbf{x}'_1 \neq \mathbf{a}_1 \vee \mathbf{x}'_2 \neq \mathbf{a}_2 \vee \cdots \vee \mathbf{x}'_n \neq \mathbf{a}_n \vee \mathbf{A}'). \tag{20}$$

It may happen that the bound variables $\mathbf{x}_i$ must be renamed to $\mathbf{x}_i'$ to prevent the binding of free variables in the terms $\mathbf{a}_i$. Implementationally this is not necessary. As the quantifiers are entered a new *frame* is pushed on the top of the stack with the bindings of formals $\mathbf{x}_i$ to the actuals $\mathbf{a}_i$. We should note here that an implementation will combine the predicate call rule with the subsequent n quantifier rules and n basic predicate absorptions into a single action. After the formals have been bound to the actuals the body $\mathbf{A}$ of the predicate $\mathbf{P}$ will be computed without any changes in it. Similarly on the exit from the body, the n-fold quantifier elimination will be done in a single step.

Basic Predicate Absorption Rules. The atomic formula $\mathbf{A}$ is an application of a basic predicate.

$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{A} \leftrightarrow \mathbf{A}, \Gamma \bullet\!\!\nrightarrow \Delta \tag{21}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{A} \leftrightarrow \Gamma \bullet\!\!\rightarrow \Delta, \mathbf{A}. \tag{22}$$

These rules construct a new environment by the absorption of the formula $\mathbf{A}$. If the computations is done with normalized environments the environments on the right hand side should be now normalized. We note again that the normalization is an implementation optimization step and from the logical point of view is not necessary. If, however, the computation is sequential it is important that an unsatisfiable environment $\Gamma \bullet\!\!\nrightarrow \Delta$ is now replaced by $\bullet\!\!\rightarrow$, and dually an unfalsifiable environment $\Gamma \bullet\!\!\rightarrow \Delta$ is replaced by $\bullet\!\!\nrightarrow$. The first case corresponds to a failed test, the second to its dual. We namely have to enable the **F**–**T** propagation rules (23-26) culminating in the truth value elimination (27-28) so a backtrack computation can be started. With parallel computation other alternatives (or dually conjuncts) contribute their solutions anyway.

The normalization procedure for our natural numbers consists of about twenty cases. The cases depend on the form of sequences Γ, Δ as well as on the form of $\mathbf{A}$. We just sketch some of the typical cases here and invite the interested reader to fill in the rest.

In the following we shall denote by $\mathbf{n}$, and $\mathbf{m}$ two different numerals. If $\mathbf{A} \equiv \mathbf{n} = \mathbf{n}$ the rule (21) yields $\Gamma \bullet\!\!\nrightarrow \Delta$ and the rule (22) yields $\bullet\!\!\nrightarrow$. If $\mathbf{A} \equiv \mathbf{n} = \mathbf{m}$ then the first rule yields $\bullet\!\!\rightarrow$ and the second $\Gamma \bullet\!\!\rightarrow \Delta$. If $\mathbf{A} \equiv \mathbf{x} = \mathbf{n}$ then if there is an identity $\mathbf{x} = \mathbf{a}$ in Γ the following holds.

$$\mathbf{x} = \mathbf{n}, \Gamma \bullet\!\!\nrightarrow \Delta \leftrightarrow \mathbf{x} = \mathbf{n}, \mathbf{x} = \mathbf{a}, \Gamma' \bullet\!\!\nrightarrow \Delta \leftrightarrow \mathbf{x} = \mathbf{n}, \Gamma' \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{n} = \mathbf{a}.$$

We recursively normalize the last environment observing that the term $\mathbf{a}$ is either a numeral or a variable before $\mathbf{x}$, so the recursion cannot go forever. The situation with $\mathbf{x} = \mathbf{a}$ in Δ of the second rule is dual. With the same $\mathbf{A} \equiv \mathbf{x} = \mathbf{n}$, if there is no binding $\mathbf{x} = \mathbf{a}$ in the Γ of (21), we reorder Δ to show all constraints $\mathbf{x} = \mathbf{a}_i$ in it.

$$\mathbf{x} = \mathbf{n}, \Gamma \bullet\!\!\nrightarrow \Delta \leftrightarrow \mathbf{x} = \mathbf{n}, \Gamma \bullet\!\!\nrightarrow \Delta', \mathbf{x} = \mathbf{a}_1, \mathbf{x} = \mathbf{a}_2, \ldots, \mathbf{x} = \mathbf{a}_n \leftrightarrow$$
$$\mathbf{x} = \mathbf{n}, \Gamma \bullet\!\!\nrightarrow \Delta' \;\&\; \mathbf{n} \neq \mathbf{a}_1 \;\&\; \mathbf{n} \neq \mathbf{a}_2, \ldots, \mathbf{n} \neq \mathbf{a}_n.$$

The normalization goes into an n-fold recursion with "lesser" formulas. Although the above formula looks as if we had to employ the negation rules n-times, an implementation will only perform an equivalent action without actually doing the negations. The reader familiar with delaying in some Prologs will note that the inequalities $\mathbf{x} \neq \mathbf{a}_i$ were delayed within the environment $\Gamma \bullet\!\!\nrightarrow \Delta$ because there was no binding for the variable $\mathbf{x}$. Once the binding arrives the inequalities are rescheduled for execution.

Once an environment absorbs an basic atomic predicate the environment will start a backward propagation through enclosing formulas until the context $\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \mathbf{A}$ or $\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{A}$ is restored for a renewed downward travel. Generally the disjunctive environment $\Gamma \bullet\!\!\nrightarrow \Delta$ travels backwards through the enclosing operators in the context $\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}$ and a conjuctive environment travels backward in the context $\Gamma \bullet\!\!\rightarrow \Delta \,\&\, \mathbf{A}$. If the environments are kept normalized the backward movement does not cost anything (see below for details) in most of the cases. The reader is advised to read the backward rules as an explanation of what goes on at the logical level.

The following group of rules is important only with the sequential computation. With parallel computation the rules can be viewed as only an optimization.

T and F Propagation Rules:

$$\bullet\!\!\nrightarrow \vee \mathbf{A} \leftrightarrow \bullet\!\!\nrightarrow \qquad (23)$$
$$\bullet\!\!\rightarrow \,\&\, \mathbf{A} \leftrightarrow \bullet\!\!\rightarrow \qquad (24)$$
$$\exists \mathbf{x}(\bullet\!\!\rightarrow) \leftrightarrow \bullet\!\!\rightarrow \qquad (25)$$
$$\forall \mathbf{x}(\bullet\!\!\nrightarrow) \leftrightarrow \bullet\!\!\nrightarrow. \qquad (26)$$

T and F Elimination Rules: The formula **A** contains environments.

$$\bullet\!\!\nrightarrow \,\&\, \mathbf{A} \leftrightarrow \mathbf{A} \qquad (27)$$
$$\bullet\!\!\rightarrow \vee \mathbf{A} \leftrightarrow \mathbf{A}. \qquad (28)$$

The rule (24) corresponds to the fail situation in Prolog. We had a disjunctive environment which has changed into $\bullet\!\!\rightarrow$ after absorption, i.e. the test in the absorption rule (21) failed. The failure (and dually the success in the rule (23)) is propagated backwards until the situation from the rule (28) (dually the situation from (27)) is reached. Then the computation continues with the alternative. Inspection of the rules will reveal that in such situations the formula **A** will always contain an environment. In an implementation the failure (dually the success) immediately continues from the last choice point with the environment restored from the trailed record of changes.

The rule (25) propagates the failure through an existential quantifier. The rule applies only when the quantifier was entered without the change of the mode via the rule (14). Nothing happens in the implementation. The case when the existential quantifier was entered with the mode change requires some implementation action and is covered with the quantifier elimination rules (37-46). The success propagation in (26) is completely dual.

Disjunction Propagation Rules:

$$(\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}) \vee \mathbf{B} \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta \vee (\mathbf{A} \vee \mathbf{B}) \qquad (29)$$
$$(\Gamma \bullet\!\!\rightarrow \Delta \,\&\, \mathbf{A}) \vee \mathbf{B} \leftrightarrow (\Gamma \bullet\!\!\rightarrow \Delta \vee \mathbf{B}) \,\&\, (\mathbf{A} \vee \mathbf{B}). \qquad (30)$$

Conjunction Propagation Rules:

$$(\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}) \,\&\, \mathbf{B} \leftrightarrow (\Gamma \bullet\!\!\nrightarrow \Delta \,\&\, \mathbf{B}) \vee (\mathbf{A} \,\&\, \mathbf{B}) \qquad (31)$$
$$(\Gamma \bullet\!\!\rightarrow \Delta \,\&\, \mathbf{A}) \,\&\, \mathbf{B} \leftrightarrow \Gamma \bullet\!\!\rightarrow \Delta \,\&\, (\mathbf{A} \,\&\, \mathbf{B}). \qquad (32)$$

A retracting disjunctive environment travelling in the context $\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}$ moves backwards through enclosing disjunctions (29) by employing associativity and through enclosing conjunctions (31) by employing distributivity. The two other situations are dual. This is what happens at the logical level. At the level of the implementation all the choice points have been already laid when the downward travelling disjunctive environment entered a disjunction and the computation proceeds to the formula **B** in the rule (31) directly without any copying whatsover. Similarly with the conjunctive environment in the dual situation.

Negation Elimination and Propagation Rules:

$$\neg\,(\Gamma \bullet\!\!\nrightarrow \Delta) \leftrightarrow \Gamma \bullet\!\!\rightarrow \Delta \qquad (33)$$
$$\neg\,(\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}) \leftrightarrow \Gamma \bullet\!\!\rightarrow \Delta \,\&\, \neg\,\mathbf{A} \qquad (34)$$
$$\neg\,(\Gamma \bullet\!\!\rightarrow \Delta) \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta \qquad (35)$$
$$\neg\,(\Gamma \bullet\!\!\rightarrow \Delta \,\&\, \mathbf{A}) \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta \vee \neg\,\mathbf{A}. \qquad (36)$$

An environment retracting either by itself or with a backtracking companion simply switches from the disjunctive to the conjunctive mode and vice versa.

Existential Quantifier Elimination Rules:

$$\exists \mathbf{x}(\Gamma \bullet\!\!\nrightarrow \Delta) \leftrightarrow \mathbf{E} \tag{37}$$
$$\exists \mathbf{x}(\Gamma \bullet\!\!\nrightarrow \Delta \vee \mathbf{A}) \leftrightarrow \mathbf{E} \vee \exists \mathbf{x}\mathbf{A} \tag{38}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \exists \mathbf{x}(\Gamma' \bullet\!\!\nrightarrow \Delta') \leftrightarrow \mathbf{E} \tag{39}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \exists \mathbf{x}(\Gamma' \bullet\!\!\nrightarrow \Delta' \vee \mathbf{A}) \leftrightarrow \mathbf{E} \vee \exists \mathbf{x}\mathbf{A} \tag{40}$$
$$\Gamma \bullet\!\!\rightarrow \Delta \vee \exists \mathbf{x}(\bullet\!\!\rightarrow) \leftrightarrow \Gamma \bullet\!\!\rightarrow \Delta. \tag{41}$$

Universal Quantifier Elimination Rules:

$$\forall \mathbf{x}(\Gamma \bullet\!\!\rightarrow \Delta) \leftrightarrow \mathbf{E} \tag{42}$$
$$\forall \mathbf{x}(\Gamma \bullet\!\!\rightarrow \Delta \;\&\; \mathbf{A}) \leftrightarrow \mathbf{E} \;\&\; \forall \mathbf{x}\mathbf{A} \tag{43}$$
$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \forall \mathbf{x}(\Gamma' \bullet\!\!\rightarrow \Delta') \leftrightarrow \mathbf{E} \tag{44}$$
$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \forall \mathbf{x}(\Gamma' \bullet\!\!\rightarrow \Delta' \;\&\; \mathbf{A}) \leftrightarrow \mathbf{E} \;\&\; \forall \mathbf{x}\mathbf{A} \tag{45}$$
$$\Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \forall \mathbf{x}(\bullet\!\!\nrightarrow) \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta. \tag{46}$$

We explain here only the disjunctive case, the conjunctive one is dual. The rules (37-38) apply when the existential quantifier has been entered without a mode change by the rule (14). The environment $\mathbf{E}$ is obtained by the existential quantifier elimination to satisfy

$$\mathbf{E} \leftrightarrow \exists \mathbf{x}(\Gamma \bullet\!\!\nrightarrow \Delta).$$

This is always possible but unless the environment $\Gamma \bullet\!\!\nrightarrow \Delta$ is kept normalized it can be time consuming. The rule (38) explains the reason why the formula within an existential quantifier has to be computed disjunctively. We simply need a disjunction so we are able to split the quantifier by the application of the tautology

$$\exists \mathbf{x}(\mathbf{B} \vee \mathbf{C}) \leftrightarrow \exists \mathbf{x}\mathbf{B} \vee \exists \mathbf{x}\mathbf{C}.$$

The quantifier is then eliminated from the split environment. This is the only way to bring out the results from within an existential quantifier.

In our theory of natural numbers the environment $\Gamma \bullet\!\!\nrightarrow \Delta$ is normalized and since the scopes of quantifiers are completely nested the bound variable $\mathbf{x}$ is on the top of the stack and is simply popped. By popping the stack we delete all identities containing the variable $\mathbf{x}$ from the environment. The justification for this can be seen from the following.

Since $\mathbf{x}$ is the variable occurring after all other variables in the environment all its occurrences in the environment $\Gamma \bullet\!\!\nrightarrow \Delta$ must be of the form $\mathbf{x} = \mathbf{a}$ where the term $\mathbf{a}$ cannot be $\mathbf{x}$. When the identity $\mathbf{x} = \mathbf{a}$ occurs in the antecedent then by the single value constraint on normalized environment there are no other occurrences of $\mathbf{x}$ in the environment and one simply uses the tautology (18) to obtain the environment $\mathbf{E}$

$$\exists \mathbf{x}(\Gamma \bullet\!\!\nrightarrow \Delta) \leftrightarrow \exists \mathbf{x}(\mathbf{x} = \mathbf{a}, \Gamma' \bullet\!\!\nrightarrow \Delta) \leftrightarrow \Gamma' \bullet\!\!\nrightarrow \Delta \leftrightarrow \mathbf{E}.$$

When $\mathbf{x}$ does not occur in the antecedent of $\Gamma \bullet\!\!\nrightarrow \Delta$ it may occur possibly many times in the consequent

$$\exists \mathbf{x}(\Gamma \bullet\!\!\nrightarrow \Delta) \leftrightarrow \exists \mathbf{x}(\Gamma \bullet\!\!\nrightarrow \Delta', \mathbf{x} = \mathbf{a}_1, \mathbf{x} = \mathbf{a}_2, \ldots, \mathbf{x} = \mathbf{a}_n) \leftrightarrow$$
$$\Gamma \bullet\!\!\nrightarrow \Delta' \;\&\; \exists \mathbf{x}(\mathbf{x} \neq \mathbf{a}_1 \;\&\; \mathbf{x} \neq \mathbf{a}_2 \;\&\; \cdots \;\&\; \mathbf{x} \neq \mathbf{a}_n) \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta' \leftrightarrow \mathbf{E}.$$

When an universal quantifier has been entered from a disjunctive mode by the rule (16) the rules (44-46) apply for the quantifier elimination. The universal quantifier can be split (45) similarly as in (38). This is why we compute in the conjunctive mode within universal quantifiers. In both cases the environment $\mathbf{E}$ is obtained by the quantifier elimination and the collapse of two basic formulas

$$\mathbf{E} \leftrightarrow \Gamma \bullet\!\!\nrightarrow \Delta \;\&\; \forall \mathbf{x}(\Gamma' \bullet\!\!\rightarrow \Delta'). \tag{47}$$

In our theory of natural numbers the environment $\Gamma' \bullet\!\!\rightarrow \Delta'$ is normalized and $\mathbf{x}$ comes after all other variables. The universal quantifier is eliminated by simply dropping all identities containing the variable $\mathbf{x}$. This is simply implemented by popping the variable $\mathbf{x}$ off the stack. The justification is dual to the justification of the existential quantifier elimination. This is not expensive. The expensive part comes when this new environment is conjuncted with the environment

$\Gamma \bullet\!\!\nrightarrow \Delta$ stored upon the change of mode. This is obviously always possible but not necessarily cheap. We do not suggest any technique here beyond observing that the methods of implementing the all-solutions meta-predicate in Prolog (see for instance [13]) should be applicable here. An example of environment operations (47) is given in section (5) where we compute the formula $\forall z\, K(z, v)$. The reader will note there that the environment **E** actually becomes a disjunction of disjunctive predicates.

The rule (46) is applied when a success is being propagated in the conjunctive mode and it encounters a change of mode connected with the universal quantifier. The universally quantified formula acted as a succeeding test, so we simply continue with the stored environment.

5. Negation as Failure versus Precomplete Negation.

Negation as failure delays negated formulas with free variables. If these variables are not subsequently sufficiently instantiated the computation does not lead anywhere. Precomplete negation simply starts to compute dually. Consider the predicate M defined by (2). The formula

$$\neg\, M(y) \;\&\; y \neq 3$$

will never be computed by delaying. We present the computation with precomplete negation in a shortened way by omitting the application of straightforward computation rules. We remind the reader that & binds stronger than $\vee$.

$$\begin{aligned}
&\bullet\!\!\nrightarrow \;\&\; (\neg\, M(y) \;\&\; y \neq 3) \leftrightarrow \neg\, (\bullet\!\!\rightarrow \vee\, M(y)) \;\&\; y \neq 3 \leftrightarrow \\
&\neg\, \forall x(\bullet\!\!\rightarrow \vee\, x \neq y \vee x \neq 3 \;\&\; x \neq 2) \;\&\; y \neq 3 \leftrightarrow \\
&\neg\, \forall x(x = y \bullet\!\!\rightarrow \vee\, x \neq 3 \;\&\; (x = y \bullet\!\!\rightarrow \vee\, x \neq 2)) \;\&\; y \neq 3 \leftrightarrow \\
&\neg\, \forall x(x = 3,\, y = 3 \bullet\!\!\rightarrow \;\&\; (x = y \bullet\!\!\rightarrow \vee\, x \neq 2)) \;\&\; y \neq 3 \leftrightarrow \\
&\neg\, (y = 3 \bullet\!\!\rightarrow \;\&\; \forall x(x = y \bullet\!\!\rightarrow \vee\, x \neq 2)) \;\&\; y \neq 3 \leftrightarrow \\
&(y = 3 \bullet\!\!\nrightarrow \vee\, \neg\, \forall x(x = y \bullet\!\!\rightarrow \vee\, x \neq 2)) \;\&\; y \neq 3 \leftrightarrow \\
&(y = 3 \bullet\!\!\nrightarrow \;\&\; y \neq 3) \vee \neg\, \forall x(x = y \bullet\!\!\rightarrow \vee\, x \neq 2) \;\&\; y \neq 3 \leftrightarrow \\
&\bullet\!\!\rightarrow \vee\, (\neg\, \forall x(x = y \bullet\!\!\rightarrow \vee\, x \neq 2) \;\&\; y \neq 3) \leftrightarrow \\
&\neg\, \forall x(x = y \bullet\!\!\rightarrow \vee\, x \neq 2) \;\&\; y \neq 3 \leftrightarrow y = 2 \bullet\!\!\nrightarrow \;\&\; y \neq 3 \leftrightarrow y = 2 \bullet\!\!\nrightarrow
\end{aligned}$$

There is a subtle problem with negation as failure in Prolog. The problem is not well-known because the quantifiers are not explicit. Nevertheless, it is an obvious shortcoming of negation as failure. The problem stems from the fact that quantifiers are not eliminated properly by delaying. Consider the Prolog predicate P defined by the clause

$$P(x) \leftarrow x = 2,\, x \neq y.$$

The formula $P(2)$ is obviously true, yet it will not be computed by delaying because nothing instantiates the variable y. The precomplete negation starts with the definition

$$P(x) \leftrightarrow \exists y(x = 2 \;\&\; x \neq y)$$

and computes the formula as

$$\begin{aligned}
&\bullet\!\!\nrightarrow \;\&\; P(2) \leftrightarrow \exists x(x = 2 \bullet\!\!\nrightarrow \;\&\; \exists y(x = 2 \;\&\; x \neq y)) \leftrightarrow \\
&\exists x, y(x = 2 \bullet\!\!\nrightarrow \;\&\; x \neq y) \leftrightarrow \exists x, y\, \neg\, (x = 2 \bullet\!\!\rightarrow \vee\, x = y) \leftrightarrow \\
&\exists x, y\, \neg\, (x = 2 \bullet\!\!\rightarrow y = x) \leftrightarrow \exists x, y(x = 2 \bullet\!\!\nrightarrow y = x) \leftrightarrow \qquad (48) \\
&\exists x(x = 2 \bullet\!\!\nrightarrow) \leftrightarrow \bullet\!\!\nrightarrow.
\end{aligned}$$

The crucial step is the proper application of quantifier elimination ($\exists y$) in the line (48).

Although both of the above examples are artificially simple one should not jump to the conclusion that they do not address a real problem with the negation as failure. The problem is that not only is negation as failure almost impossible to characterize semantically in the presence of free variables, but also that its treatment by delaying is insufficient.

As the last example we compute the formula $\forall z\, K(z, v)$ with (4) as the definition of the predicate K. After translation into Prolog by a double negation and an auxiliary predicate we obtain

$$\begin{aligned}
&\leftarrow \neg\, R(v) \\
&R(v) \leftarrow \neg\, K(z, v).
\end{aligned}$$

Negation as failure will not be able to compute the solution $v = 3$ because of delaying. Since the computation of this formula with precomplete negation is quite long we use the following abbreviations to improve the readablity:

$$\mathbf{A} \equiv x = z,\ y = v \bullet\!\!\rightarrow \vee\ y = 2 \vee y = 3$$
$$\mathbf{B} \equiv x \neq 1\ \&\ (y = 1 \vee y = 3)$$
$$\mathbf{C} \equiv v = 1 \bullet\!\!\nrightarrow \vee\ v = 3 \bullet\!\!\nrightarrow$$
$$\mathbf{D} \equiv y = v,\ x = z \bullet\!\!\rightarrow v = 2,\ v = 3 \vee y = 1 \vee y = 3$$

The environment operations (47) associated with the universal quantifier occur in lines (49,50,51). As mentioned in section (3) the single basic formula **E** of (47) actually becomes a disjunction of disjunctive predicates. We give the three applications of (47) in the order of their appearance in formulas (49,50,51):

$$v = 1 \bullet\!\!\nrightarrow \vee\ v = 3 \bullet\!\!\nrightarrow\ \leftrightarrow\ \bullet\!\!\nrightarrow\ \&\ \forall z, x, y(x = z,\ y = v \bullet\!\!\rightarrow z = 1,\ v = 1,\ v = 3)$$
$$v = 3 \bullet\!\!\nrightarrow\ \leftrightarrow \mathbf{C}\ \&\ \forall z, x, y(y = v,\ x = 1,\ z = 1 \bullet\!\!\rightarrow v = 2,\ v = 3)$$
$$v = 3 \bullet\!\!\nrightarrow\ \leftrightarrow v = 3 \bullet\!\!\nrightarrow\ \&\ \forall z, x, y(y = v,\ x = z \bullet\!\!\rightarrow v = 2,\ v = 3,\ v = 1).$$

The computation proceeds as follows:

$$\bullet\!\!\nrightarrow\ \&\ \forall z\, K(z, v) \leftrightarrow\ \bullet\!\!\nrightarrow\ \&\ \forall z(\bullet\!\!\rightarrow \vee\ K(z, v)) \leftrightarrow \forall z(\bullet\!\!\rightarrow \vee\ K(z, v)) \leftrightarrow$$
$$\forall z, x, y(x = z,\ y = v \bullet\!\!\rightarrow \vee\ x = 1\ \&\ (y = 2 \vee y = 3) \vee \mathbf{B}) \leftrightarrow$$
$$\forall z, x, y(x = z,\ y = v \bullet\!\!\rightarrow z = 1\ \&\ \mathbf{A} \vee \mathbf{B}) \leftrightarrow$$
$$\forall z, x, y((x = z,\ y = v \bullet\!\!\rightarrow z = 1 \vee \mathbf{B})\ \&\ (\mathbf{A} \vee \mathbf{B})) \leftrightarrow$$
$$\forall z, x, y(\bullet\!\!\nrightarrow\ \&\ (x = z,\ y = v \bullet\!\!\rightarrow z = 1 \vee y = 1 \vee y = 3)\ \&\ (\mathbf{A} \vee \mathbf{B})) \leftrightarrow$$
$$\forall z, x, y((x = z,\ y = v \bullet\!\!\rightarrow z = 1 \vee y = 1 \vee y = 3)\ \&\ (\mathbf{A} \vee \mathbf{B})) \leftrightarrow$$
$$\forall z, x, y(x = z,\ y = v \bullet\!\!\rightarrow z = 1,\ v = 1,\ v = 3\ \&\ (\mathbf{A} \vee \mathbf{B})) \leftrightarrow \tag{49}$$
$$(v = 1 \bullet\!\!\nrightarrow \vee\ v = 3 \bullet\!\!\nrightarrow)\ \&\ \forall z, x, y(\mathbf{A} \vee \mathbf{B}) \leftrightarrow$$
$$\mathbf{C}\ \&\ \forall z, x, y(y = v,\ x = z \bullet\!\!\rightarrow v = 2,\ v = 3 \vee \mathbf{B}) \leftrightarrow$$
$$\mathbf{C}\ \&\ \forall z, x, y(y = v,\ x = 1,\ z = 1 \bullet\!\!\rightarrow v = 2,\ v = 3\ \&\ \mathbf{D}) \leftrightarrow \tag{50}$$
$$v = 3 \bullet\!\!\nrightarrow\ \&\ \forall z, x, y\, \mathbf{D} \leftrightarrow$$
$$v = 3 \bullet\!\!\nrightarrow\ \&\ \forall z, x, y(y = v,\ x = z \bullet\!\!\rightarrow v = 2,\ v = 3,\ v = 1) \leftrightarrow \tag{51}$$
$$v = 3 \bullet\!\!\nrightarrow.$$

6. Conclusions.

We hope that we have sufficiently demonstrated the weaknesses of negation as failure. The strengthening of negation as failure to precomplete negation allows us to compute both positive and negative formulas dually without a concern for the free variables within negations.

We consider precomplete negation as a natural barrier on the way to the classical negation. Classical negation is certainly computationally possible but unfortunately it is very expensive. On the other hand, the implementation of precomplete negation is a straightforward extension of known efficient techniques. The only computationally expensive part is concerned with the rules (39-40,44-45) when we generate values of free variables accross a quantifier entered with a change of mode. If the stored environment contains bindings for the free variables then the quantified formula acts as a mere test. Otherwise quite complex environment operations are necessary. We think that these operations can be efficiently handled by the methods of all-solutions meta-predicates in Prolog. The reader will note that the relatively high cost of the environment operations is not to be associated with our particular method but rather with the inherent complexity of computing universal quantifiers disjunctively (existential quantifiers conjunctively).

Although we have demonstrated the techniques of environment manipulation with a very simple theory of natural numbers the method is completely general and applies to any decidable theories. We are currently implementing the precomplete negation in our logic language R-Maple where we decide also the relation $<$ over integers and allow terms of the form **x+n** with **n** an integer numeral. We are seriously contemplating adding the full treatment of addition by allowing arbitrary terms **a+b**. This requires the implementation of the decision procedure for Pressburger arithmetic [7] where the multiplication terms must have the form **n.a**. The decision procedure for

Pressburger arithmetic is in general extremely costly [15] but may be worthwhile for small programs. Obviously, there is no decision procedure for the general terms **a**.**b**. Outside of the domain of integers we decide equations and inequalities among pairs (S-expressions). R-Maple is a typed language where the types have a Pascal-like form allowing recursion in types. Associated with each type T is a predicate $T(\mathbf{a})$ satisfied iff the object denoted by **a** is of the type T. We impose certain restrictions on the form of types and as a consequence we can decide arbitrary formulas involving the predicates T. In other words, we count the type predicates as basic predicates.

We are currently preparing a paper where we prove the completness of precomplete negation with respect to the computational logic characterized by the Gentzen-like axioms in section (2). Computational logic restricts the law of the excluded middle. The excluded middle applies only to decidable formulas. The proof of completness requires that we show that everything provable in a computational theory is computable with the precomplete negation. The converse merely states the obvious fact that precomplete negation is sound. However, completness is achieved only when we apply the rules of computations to all environments in parallel. The sequential case is very difficult to characterize semantically. We shall discuss this in the prepared paper.

References:

[1] Andrews J., An Environment Theory with Precomplete Negation over Pairs, Msc. thesis UBC, Vancouver 1986.

[2] Clark K., Negation as Failure, in: Logic and Databases, Gallaire and Minker (eds.), Plenum Press, 1978.

[3] Clark K., McCabe F., micro-Prolog: Programming in Logic, Prentice Hall, 1984.

[4] Colmerauer A., Kanoui H., van Caneghem M., Prolog, Theoretical Principle and Current Trends, Tech, Sci. Inf. 2:4, 1983.

[5] Fitting M., A Kripke-Kleene Semantics for Logic Programs, J. Logic Programming 4:295-312, 1985.

[6] Gilmore P., Natural deduction Based Set Theories: A new Resolution of the Old Paradoxes, J. Symbolic Logic 51:2, 1986.

[7] Hilbert D., Bernays P., Grundlagen der Mathematik, Springer 1968.

[8] Kripke S., Outline of a Theory of Truth, J. Philosophy 72:690-716, 1975.

[9] Lloyd J., Topor R., Making Prolog more Expressive, J. Logic Programming 3:225-240, 1984.

[10] Malachi Y., Manna Z., Waldinger R., TABLOG: The Deductive-Tableau Programming Language, STAN-CS-84, Standford 1984.

[11] Monk J., Mathematical Logic, Springer 1976.

[12] Naish L., MU-Prolog 3.1db Reference Manual, University of Melbourne 1984.

[13] Naish L., All Solutions Predicates in Prolog, IEEE Symposium on Logic Programming, Boston 1985.

[14] Naish L., Negation and Quantifiers in NU-Prolog, Proceedings of 3rd International Conference on Logic Programming, London, Springer 1986.

[15] Rabin M., Fischer M., Superexponential Complexity of Pressburger's Arithmetic, SIAM-AMS Proceedings 7:27-41, 1974.

[16] Shoenfield J., Mathematical Logic, Addison-Wesley 1967.

[17] Umrigar Z., Pitchumani V., An Experiment in Programming with Full First-order Logic, IEEE Symposium on Logic Programming, Boston 1985.

[18] Voda P., A View of Programming Languages as Symbiosis of Meaning and Computations, New generation Computing 3:71-100, Springer 1985.

[19] Voda P., Computation of Full Logic Programs Using One-Variable Environments, New generation Computing 4:2, Springer 1986.

[20] Voda P., The Choices in, and Limitations of, Logic Programming, Proceedings of 3rd International Conference on Logic Programming, London, Springer 1986.

[21] Warren D., An Abstract Prolog Instruction Set. SRI Note 309, Standford Research Institute 1983.