Computation of Full Logic Programs Using One-Variable Environments

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

Technical Report 85-8

Computation of Full Logic Programs Using One-Variable Environments.

Paul J. Voda

Department of Computer Science, The University of British Columbia, 6356 Agricultural Road, Vancouver, B.C. Canada V6T 1W5.

ABSTRACT

Computation of formulas of the full first order predicate calculus is performed by first converting the multi-variable formulas into a single variable presentation of the Theory of Pairs (TP). Pairs are symbolic expressions of LISP with only one atom 0. Single variable calculus is suitable for both computations and mechanical theorem proving because the problems of multiple variable names and clashes between free and bound variables are eliminated. We present a logic programming language R^+–Maple which computes by solving equations instead of unifications and refutations. Pairs permit a single one-variable equation called *environment* equation to hold the values of all variables. Traditionally environments are implementation tools used to carry bindings of variables during the computation. With the help of environment equations for the single variable of our calculus we make the environments visible within the framework of a first order theory. This allows a straightforward demonstration of soundness of our computations. Moreover, the explicit form of environments allows to experiment with different forms of computational rules directly within the logic. The soundness of new rules can be thus readily proven formally. This has the advantage over the traditional method which buries the rules, as they are closely connected with environments, deeply within the code of an interpreter.

Keywords: Formal Semantics, Logic Programming, Environments, Formal Theories.

July 1985

Computation of Full Logic Programs Using One-Variable Environments.

Paul J. Voda

Department of Computer Science, The University of British Columbia,
6356 Agricultural Road, Vancouver, B.C. Canada V6T 1W5.

1. Introduction.

Formulas of predicate logic with multiple variables are easy to read but hard to compute. Values have to be correlated to the corresponding variables either by substitutions or by bindings. Substitutions are costly to implement, bindings require names of variables and systems of equations. We propose to compute within the framework of a Theory of Pairs (TP) [10,11] which provides for the operation of pairing. This will allow the reduction of multiple variables into a single one and the collapse a system of equations (bindings) into one equation giving the binding to the only variable.

Bindings, whether multiple or single, are maintained during computation of formulas (procedures) in a stack-like data structure called an *environment*. Environments are implementation tools not directly related to predicate logic. As a consequence it is rather hard to demonstrate an interpretation strategy correct (sound). We reformulate TP as a one-variable first-order theory and show how to express environments within the theory. As a consequence the computation rules are directly visible in the logic and an interpreter is immediately seen sound. In such a framework we can easily investigate alternative interpretation strategies. Furthermore, by having to deal with only one variable, we can make the process of interpretation more efficient.

The use of environments allows the computation of logic programs by the solution of equations rather than by unification and proofs by refutations of most logic programming languages. There has been recently an interest in the computation of logic programs without unification. To name just two examples: Prolog-II of Colmerauer [2,3] and R-Maple by the present author [9,11]. Prolog-II solves systems of equations (i.e. environments), whereas R-Maple works with annotated (tests and assignments) identities. The annotated identities allow computation without unification as well as synchronisation of parallel processes. We illustrate the computation with environments with a sequential subset of R-Maple called R^+–Maple. Without the sychronisation there is no need to distinguish between tests and generators and we regain the symmetry of Prolog predicates. We show how to map Prolog [see for instance 1] into R^+–Maple. Thus everything which can be computed in Prolog can be computed with a better expressivness in R^+–Maple. This is not only because R^+–Maple allows ordinary connectives and quantifieres of predicate logic but also because we permit the use of different computation rules for positive and negative predicates. The safe method of dealing with negations [5,6,7] in Prolog requires a restricted form of negative predicates obtained via completion.

The paper is divided into following sections. Section (2) gives an overview of the Theory of Pairs. Section (3) illustrates the reduction to one variable informally. Section (4) develops the one-variable Theory of Pairs formally. Section (5) discusses the environments. These two sections establish the definitions and theorems required in the sections (6) and (7) to prove the soundness of the computation in R^+–Maple. Section (6) defines computations in R^+–Maple. Section (7) gives the rewriting rules used in the computations.

2. Overview of TP.

The basic theory into which programs of R^+-Maple are interpreted is the Theory of Pairs (TP) developed by the author as a formalization of S-expressions of Lisp. Pairs are S-expressions with only one atom 0. At this point the reader may ask why we do not adopt the approach of Prolog where a program is viewed as a set of axioms. The only rules of inference are the computational rules. Prolog sets up a new formal theory for each program. The basic question one has to ask about a new formal theory is the question of consistency. Pure Prolog computes only positive calculus of implications so there is a model for each set of axioms. The situation is dramatically different as soon as one computes with a Prolog implementing a sound form of negation, for instance with MU-Prolog [7]. Now one should - in principle - prove the consistency of each program before trusting the results of computations.

The approach we have taken in a previous paper [11] is based on defining a programming language as a subset of a sufficiently rich theory which is *extended* with each new predicate. If one takes suitable precautions with the form of extensions by using only *conservative extensions* [see for instance 8] then the extended theory contains the computational rules as theorems rather than as new axioms. With a Prolog like approach one has to prove that a program has a model. With the extension approach one introduces the predicates of a new program in a restricted way for which the consistency is guaranteed. The computational axioms which are usually recursive have to be then proven as theorems from non-recursive definitions. Thus instead of constructing models we are proving theorems. In this respect our approach is only slightly superior to the model theoretic one. Models are usually non-constructive, provability is semi-decidable. The real advantage of embedding programs into a basic theory is that one can use the full deductive apparatus of the theory to reason about programs. One can use the full quantifier rules and a suitable induction principle.

Theory of Pairs is described elsewhere [10,11]. In this section we only recapitulate the basic principles of TP. The domain of the intended interpretation of TP contains the individual 0 and is closed under the operation of pairing. Two pairs are equal iff the corresponding components are equal. No pair is equal to 0. TP is a first order formal theory (see for instance [11]) with the primitive constant symbol 0 and the binary function symbol $[\_,\_]$ of pairing. In addition to the identity, there is only one binary predicate symbol, $\in$, of *list* membership.

A term composed only by pairing from 0 is called a *literal*. For instance $[[0,0],[0,0]]$ is a literal. Literals correspond to the numerals of Peano Arithmetic. We shall assume the operation of pairing to associate to the right. Thus $[\mathbf{a},\mathbf{b},\mathbf{c}]$ abbreviates $[\mathbf{a},[\mathbf{b},\mathbf{c}]]$. We shall use bold faced letters as syntactic variables ranging over variables $(\mathbf{x},\mathbf{y},\cdots)$, terms $(\mathbf{a},\mathbf{b},\cdots)$ or formulas $(\mathbf{A},\mathbf{B},\cdots)$ in the standard fashion of logic. To maintain close ties to Prolog we shall also use the reverse implication $\mathbf{A}\leftarrow\mathbf{B}$ as an abbreviation for $\mathbf{B}\rightarrow\mathbf{A}$. The precedence of connectives and quantifiers is given in the ascending order as follows: $(\rightarrow,\ \leftarrow,\ \leftrightarrow)$, $(\vee)$ $(\&)$ $(\neg,$ $\forall,\exists)$. We denote by $\mathbf{a}\{\mathbf{x}:=\mathbf{b}\}$, resp. $\mathbf{A}\{\mathbf{x}:=\mathbf{b}\}$ the term, resp. formula, obtained from the term $\mathbf{a}$ (formula $\mathbf{A}$) by the *substitution* of the term $\mathbf{b}$ for all free occurrences of the variable $\mathbf{x}$. We use the meta-theoretic symbol of syntactic identity $\mathbf{a}\equiv\mathbf{b}$ ($\mathbf{A}\equiv\mathbf{B}$) to express the fact that the terms $\mathbf{a}$ and $\mathbf{b}$ (formulas $\mathbf{A}$ and $\mathbf{B}$) are exactly the same sequences of symbols.

The axioms of TP - besides the logical axioms for connectives, quantifiers and identity - are the following ones:

$$[x,y]=[x',y']\rightarrow x=x'\ \&\ y=y' \qquad \text{(Un)}$$
$$x\notin 0 \qquad \text{(Mem1)}$$
$$x\in[y,z]\leftrightarrow x=y\vee x\in z \qquad \text{(Mem2)}$$
$$\mathbf{A}\{\mathbf{x}:=0\}\ \&\ \forall\mathbf{x}\forall\mathbf{y}\,(\mathbf{A}\ \&\ \mathbf{A}\{\mathbf{x}:=\mathbf{y}\}\rightarrow\mathbf{A}\{\mathbf{x}:=[\mathbf{x},\mathbf{y}]\})\rightarrow\forall\mathbf{x}\mathbf{A}. \qquad \text{(Ind)}$$

The axiom (Un) asserts the uniqueness of pairing (the reverse implication is a logical equality axiom). The schema of axioms of induction (Ind) in effect says that all individuals are composed from 0 by pairing. Thus all individuals are denoted by literals. The membership axioms define properties of $\in$.

The domain of pairs corresponds to S-expressions of LISP with 0 being the only atom. Note that instead of $(\mathbf{a}.\mathbf{b})$ we write $[\mathbf{a}, \mathbf{b}]$ where $\mathbf{a}$ and $\mathbf{b}$ are literals. The principle of induction, together with membership axioms guarantees that every non zero individual can be uniquely written in the form $[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n, 0]$. This means that every individual of TP is a list. We do not introduce the list notation of LISP because the list $(\mathbf{a}\ \mathbf{b}\ \mathbf{c})$ can be written as $[\mathbf{a}, \mathbf{b}, \mathbf{c}, 0]$ in TP. Note also that there is no need for the operation $cons(\mathbf{a}, \mathbf{b})$ of LISP. One simply writes $[\mathbf{a}, \mathbf{b}]$ to denote exactly the same element.

The standard interpretation of the predicate symbol $\in$ makes

$$x \in [a_1, a_2, \ldots, a_n, 0]$$

true iff $x = a_i$ for some i such that $1 \leq i \leq n$.

One can introduce into TP new predicate and function symbols by conservative extensions. We briefly sketch the method here, for a detailed treatment see for instance [11]. Predicates are introduced by explicit definitions. For instance the predicate $Large(x)$ satisfied only by lists of at least two elements has the following defining axiom:

$$Large(x) \leftrightarrow \exists w \exists y \exists z\ x = [w, y, z].$$

Functions are introduced by contextual or explicit definitions. Function $prefix(x)$ prefixing a list by two zeros can be defined by an explicit definition

$$prefix(x) = [0, 0, x].$$

More complicated functions are introduced by contextual definitions. Assume for instance that the formula $\mathbf{A}(x, y, z)$ contains at most the variables x, y, and z free. Suppose further that we are able to prove the existence and uniqueness conditions

$$\vdash \exists z\, \mathbf{A}(x, y, z) \tag{1}$$
$$\vdash \mathbf{A}(x, y, z)\ \&\ \mathbf{A}(x, y, z') \rightarrow z = z'. \tag{2}$$

We can now introduce a new binary function symbol f with the help of the defining axiom

$$\mathbf{A}(x, y, f(x, y)). \tag{3}$$

The explicit definition of predicates guarantees their elimination in any context. Each predicate is simply replaced by its right hand side. The same is true of explicit definitions of functions. Functions introduced by contextual definitions can be always eliminated in a context of formulas. For instance $\mathbf{B}(f(\mathbf{a}, \mathbf{b}), \mathbf{c})$ is equivalent to the formula

$$\exists w (f(\mathbf{a}, \mathbf{b}) = w\ \&\ \mathbf{B}(w, \mathbf{c}))$$

assuming that the variable w does not occur in the above terms and formulas. In the view of the uniqueness condition (2), this is again equivalent to

$$\exists w (\mathbf{A}(\mathbf{a}, \mathbf{b}, w)\ \&\ \mathbf{B}(w, \mathbf{c})).$$

As a consequence we can eliminate all occurrences of defined predicates and functions by replacing them with equivalent terms or formulas. We cannot possibly prove anything new with introduced symbols which we could not have proved without them. In particular we cannot prove a contradictory formula if there was no contradiction before. Thus if we make sure that TP is extended only by the above definitions we improve the readability of TP formulas without compromising the consistency of TP.

Let us introduce two unary *projection* functions $\_.\mathbf{h}$, $\_.\mathbf{t}$ taking the head and tail of an element respectively by the following contextual defining axioms

$$(x = 0\ \&\ x.\mathbf{h} = 0) \lor \exists y\ x = [x.\mathbf{h}, y]$$
$$(x = 0\ \&\ x.\mathbf{t} = 0) \lor \exists y\ x = [y, x.\mathbf{t}].$$

We leave it to the reader to prove the existence and uniqueness conditions. As the consequence of the defining axioms we can easily prove the expected properties (4) through (7)

$$\vdash [x, y].\mathbf{h} = x \tag{4}$$
$$\vdash [x, y].\mathbf{t} = y \tag{5}$$
$$\vdash 0.\mathbf{h} = 0 \tag{6}$$

$\vdash \ 0.\mathbf{t} = 0.$ (7)

Natural numbers can be introduced into TP as lists containing only 0 as elements. We can abbreviate $[0, 0]$ as 1, $[0, 1]$ as 2, etc. The predicate of being a natural number has the explicit definition

$$Nat(x) \leftrightarrow \forall y(y \in x \rightarrow y = 0).$$

3. Reduction of Variables.

Before we start the discussion of environments we informally illustrate how to reduce formulas of TP into equivalent one-variable formulas. The reduction of variables is made possible by the pairing function used together with projection functions _.**h** and _.**t**. As a result the variable-free effect of combinatory logic is achieved. To keep the formulas readable we shall use multiple variables but the computations will be done on formulas containing only one variable w. A two-place predicate

$$P(x, y) \leftrightarrow \ \cdots x \cdots y \cdots$$

can be viewed as an abbreviation for the one-place predicate

$$P(w) \leftrightarrow w \neq 0 \ \& \ (\cdots w.\mathbf{h} \cdots w.\mathbf{t} \cdots)$$

or equivalently as

$$P(w) \leftrightarrow \exists x \exists y(w = [x, y] \ \& \ (\cdots x \cdots y \cdots)).$$

Now that all predicates are one place only, we can write $P(\mathbf{a}, \mathbf{b})$, or $R(\mathbf{a}, \mathbf{b}, \mathbf{c})$, etc. as abbreviations for $P([\mathbf{a}, \mathbf{b}])$ and $R([\mathbf{a}, \mathbf{b}, \mathbf{c}])$ respectively. The two-place predicate $Len(x, y)$ satisfied when y is the length of the list x is defined in Prolog with the help of clauses

$$Len(0, 0)$$
$$Len([x, y], [0, y']) \leftarrow Len(y, y')$$

R$^+$-Maple requires all predicate definitions to be in one or or more of the following forms.

$$\vdash P(w) \leftarrow \mathbf{A}$$
$$\vdash P(w) \rightarrow \mathbf{B}$$
$$\vdash P(w) \leftrightarrow \mathbf{C}.$$

Here **A**, **B**, and **C** are formulas of a suitable extension of TP with only one variable w. The predicate P will have either a *general* definition given by the third formula or one or two *special* definitions given by the first two formulas. The first formula is used to compute $P(\mathbf{a})$ in *positive* contexts (in the scope of even number of negations), the second formula computes predicate calls in *negative* contexts. The general formula computes in any contexts. Notice that we require the clauses to be theorems of TP rather than axioms. In the absence of negations, all Prolog computations occur in the positive context.

Preparatory to the conversion of the predicate *Len* into an R$^+$-Maple form, we reduce the terms in heads of clauses to one variable by introducing identities. We temporarily retain multiple variables in the identities.

$$Len(w) \leftarrow w = [0, 0]$$
$$Len(w) \leftarrow w = [[x, y], [0, y']] \ \& \ Len(y, y').$$

The reader can convince himself that these two forms of *Len* are equivalent in TP. The last two clauses are again equivalent to the single formula

$$Len(w) \leftarrow w = [0, 0] \lor w = [[x, y], [0, y']] \ \& \ Len(y, y'). \quad (1)$$

The formula (1) can be, after the elimination of variables x, y, and y', used only in positive contexts.

The next step consists of the elimination of variables other than w from the bodies. The idea of elimination is illustrated first with a few examples. The identity $w = 0$ is satisfied only by w having the value 0. The identity $w = [x, y]$ is satisfied by w being any pair (i.e. a non-zero value).

This fact can be expressed by an identity formulated solely in terms of the variable w as $w = [w.\mathbf{h},\ w.\mathbf{t}]$. The theorem

$$\vdash\ w = [w.\mathbf{h},\ w.\mathbf{t}] \leftrightarrow \exists x \exists y\ w = [x,\ y]$$

makes clear that only pairs as values of w satisfy the left-hand side. The implication $(\rightarrow)$ is obvious. To show $(\leftarrow)$ we observe that from $w = [x,\ y]$ we have $w.\mathbf{h} = x$ and $w.\mathbf{t} = y$, so we obtain $w = [w.\mathbf{h},\ w.\mathbf{t}]$ by the properties of identity.

Using the projections of the variable w we can eliminate all other variables from identities. This will be demonstrated in section (6) formally. We can even express dependencies among parts of w. The identity $w = [x,\ x]$ is satisfied only by such pairs as values for w that have the head and tail parts identical. The identity $w = [0,\ x]$ is satisfied by any pair with the head part equal to 0. The elimination of the variable x yields the equivalent identities $w = [w.\mathbf{t},\ w.\mathbf{t}]$ and $w = [0,\ w.\mathbf{t}]$. These identities can be depicted as trees with pointers as in the figure 1. We can see that the right occurrence of the free variable x in the identities $w = [x,\ x]$ and $w = [0,\ x]$ corresponding to the projection $w.\mathbf{t}$ is a self-pointer, whereas the left occurrence of x in the first identity corresponds to a pointer. As we shall shortly see, self-pointers can assume any values, but the other pointers have their values determined by the structure they point to. We are now in the position to eliminate the variables x and y from the predicate *Len*.

$$Len(w) \leftarrow w = [0,\ 0] \lor w = [[w.\mathbf{h}.\mathbf{h},\ w.\mathbf{h}.\mathbf{t}],\ [0,\ w.\mathbf{t}.\mathbf{t}]]\ \&\ Len(w.\mathbf{h}.\mathbf{t},\ w.\mathbf{t}.\mathbf{t}). \qquad (2)$$

The reader can easily convince himself that the universal closures of formulas (1) and (2) are equivalent in TP. To allow the computation of *Len* in both positive and negative contexts the predicate can be introduced into TP to satisfy (2) with a biconditional ($\leftrightarrow$) instead of implication.

Formulas like (2) are single variable formulas. Such, essentially, variable-free formulas will have the identities represented during the computation by pointers. The figure 2 shows the graph representation of both identities of *Len*, where the self-pointers are shown as asterisks. Just as in combinatory logic, variable-free formulas are hard to read and comprehend. A practical programming language will have to introduce abbreviations in order to simplify the presentation. We can use a slight generalisation of the connective **case** of R-Maple [11]. The formula (2) (with the biconditional) becomes

$$\begin{aligned}\vdash\ Len(w) \leftrightarrow\ & \mathbf{case}\ w\ \mathbf{of} \\ & [0,\ 0]: \\ & [[*,\ x],\ [0,\ y]] : Len(x,\ y).\end{aligned}$$

Bound (local) variables x and y are used as abbreviations for $w.\mathbf{h}.\mathbf{t}$ and $w.\mathbf{t}.\mathbf{t}$ respectively. Note that the self-pointer $w.\mathbf{h}.\mathbf{h}$ which is not needed is abbreviated by an asterisk.

Variables not occurring in heads of Horn-clauses are called by Prolog programmers *local* variables. Consider the Prolog clause defining the predicate *Long* satisfied only by lists having more than six elements:

$$Long(w) \leftarrow Len(w,\ x)\ \&\ x{>}6. \qquad (3)$$

The variable x is a local variable. Logically speaking, local variables correspond to existential quantifiers. The existential quantifier is not necessary in clauses like (3) since it can be always added by $\exists$-introduction. The same predicate introduced into TP by an explicit definition with a biconditional requires the quantifier:

$$\vdash\ Long(w) \leftrightarrow \exists x(Len(x,\ w)\ \&\ x{>}6). \qquad (4)$$

This is because we cannot have an implicit existential quantifier for the $(\rightarrow)$ part of the equivalence which can be used to compute within negations.

The reader will notice that in the head of the formula (4) we have the predicate $Long(w)$ in the standard one-variable form, yet we cannot use the trick of eliminating the variable x by a projection of w. The reduction to one variable in this case will be made possible by an introduction of a *one-variable existential quantifier* $\exists\mathbf{A}$ in the next section. The formula $\mathbf{A}$ will contain only the variable w. The quantifier $\exists$ will bind the projection $w.\mathbf{h}$ leaving the projection $w.\mathbf{t}$ free in $\exists\mathbf{A}$.

This corresponds to the standard implementation of environments with stacks. Roughly speaking, in order to compute the formula $\exists\mathbf{A}(w.\mathbf{h},\ w.\mathbf{t})$ in the environment $\mathbf{a}(w)$ giving the binding for the only variable w, we start a computation of the formula $\mathbf{A}(w.\mathbf{h},\ w.\mathbf{t})$ in the *extended* environment $[w.\mathbf{h},\ \mathbf{a}(w.\mathbf{t})]$. A programmer would say that a new stack frame for an, as of yet unassigned, variable $w.\mathbf{h}$ is created on top of the stack, pushing the previous environment $\mathbf{a}(w)$ further down the stack.

Before we present the one-variable form of TP in the next section we show how to transform Prolog terms containing constants and function symbols into TP. Functions of Prolog are never computed. they are used only to generate values freely. This amounts to functions being just constructors of data structures. If Prolog is used with negation then the completion procedure [X] requires that all constants and terms generated from constants by function symbols denote different objects. This can be guaranteed in TP in a straight-forward way. For instance, the predicate $In(e,\ t)$ looking for a value e in the binary tree t has the following definition in Prolog

$$\begin{aligned}&In(x,\ leaf(x))\\&In(x,\ node(y,\ z)) \leftarrow In(x,\ y)\\&In(x,\ node(y,\ z)) \leftarrow In(x,\ z).\end{aligned}$$

The functions *leaf* and *node* can be introduced into TP by the explicit definitions

$$\begin{aligned}&\vdash\ leaf(x) = [0,\ x]\\&\vdash\ node(x,\ y) = [1,\ x,\ y].\end{aligned}$$

This definition guarantees that

$$\begin{aligned}&\vdash\ leaf(x) \neq node(y,\ z)\\&\vdash\ node(x,\ y) = node(x',\ y') \leftrightarrow x = x'\ \&\ y = y'.\end{aligned}$$

The predicate In can be introduced to TP to satisfy

$$\begin{aligned}\vdash\ In(w) \leftrightarrow \textbf{case}\ w\ \textbf{of}\ &\\ &[x,\ leaf(x)] :\\ &[x,\ node(y,\ z)] : In(x,\ y) \vee In(x,\ z).\end{aligned}$$

4. One-Variable Terms and Formulas.

We now define a class of terms and formulas with only one variable w. These classes will be called *w-terms* and *w-formulas* respectively. R^+–Maple predicate definitions and programs are constructed from w-formulas. Environments, defined in the next section, are subsets of w-terms. We do not intend to compute implications, equivalences and universal quantifiers explicitly. Consequently, we restrict w-formulas only to existential quantifiers, negations, disjunctions and conjunctions. Obviously, universal quantifiers and implications can be defined with the help of the remaining connectives. We start with a few auxiliary definitions.

A *projection* α is a possibly empty sequence of symbols $.\mathbf{h}$ and $.\mathbf{t}$. We shall use Greek letters as meta-variables ranging over projections. The empty (unit) projection will be denoted by ϵ. Projections can be concatenated in the standard way. Thus, for instance, $\epsilon\alpha.\mathbf{h}\beta$ denotes the projection obtained by appending $.\mathbf{h}$ after the (possibly empty) projection α and then appending the projection β at the end.

If α is a projection then $w\alpha$ is a *pointer*. We have $w\epsilon \equiv w$. W-terms are defined as the least class satisfying the following clauses.

i. The constant 0 and all pointers are w-terms.

ii If $\mathbf{a}$ and $\mathbf{b}$ are w-terms then $[\mathbf{a},\ \mathbf{b}]$ is a w-term.

W-formulas form the least class satisfying the following clause.

i. If $\mathbf{a}$ and $\mathbf{b}$ are w-terms then $\mathbf{a} = \mathbf{b}$ is a (atomic) w-formula.

ii. If $\mathbf{a}$ is a w-term and $\mathbf{P}$ is a one-place predicate symbol introduced into TP, then $\mathbf{P}(\mathbf{a})$ is a (atomic) w-formula.

iii. If **A** is a w-formula, so are $\exists$**A** and $\neg$ **A**.

iv. If **A** and **B** are w-formulas, so are $(\mathbf{A}\ \&\ \mathbf{B})$ and $(\mathbf{A} \vee \mathbf{B})$.

An occurrence of a subformula of the w-formula **A** is said to occur in a *positive* (*negative*) context if it occurs within the scope of even, including 0, (odd) number of negation signs.

We do not intend to give special semantics to w-terms and w-formulas as we want them to be subsets of terms and formulas of an extension of TP. R$^+$–Maple programs will then inherit the meaning from the standard interpretation of TP. W-terms are already a subset of terms of TP. W-formulas can contain the one-variable existential quantifiers which must introduced into TP as variable binding operators. The schema of defining axioms introducing the operator $\exists$ is as follows:

$$\exists(w)\mathbf{A} \leftrightarrow \exists \mathbf{x}\ \mathbf{A}\{w:=[\mathbf{x},\ w]\}. \tag{1}$$

where **A** is any formula of TP and $\mathbf{x} \not\equiv w$ is the first variable in the standard sequence not occurring in **A**. We shall abbreviate $\exists(w)\mathbf{A}$ just to $\exists\mathbf{A}$.

The variables free in $\exists(w)\mathbf{A}$ are w and all other variables free in **A**. At the same time the variable w is bound in **A**. In order to guarantee that the schema of axioms (1) is a conservative extension of TP we have to define the meta-theoretic operation of substitution for the free variables of $\exists(w)\mathbf{A}$ in such a way that the corresponding substitution in the right-hand-side of (1) leads to an equivalent formula. For that it is enough to set

$$\begin{aligned}
&(\exists(w)\mathbf{A})\{w:=\mathbf{a}\} \equiv \exists(\mathbf{a})\mathbf{A} \\
&(\exists(w)\mathbf{A})\{\mathbf{x}:=\mathbf{a}\} \equiv \exists(w)\mathbf{A} \quad \textit{where } \mathbf{x} \not\equiv w \textit{ is first variable not in } \mathbf{A} \\
&(\exists(w)\mathbf{A})\{\mathbf{x}:=\mathbf{a}\} \equiv \exists(w)\mathbf{A}\{\mathbf{x}:=\mathbf{a}\} \quad \textit{otherwise.}
\end{aligned}$$

The last identity is subject to the standard restriction of not binding free variables of **a**. Ordinary substitution leads outside of the class of w-terms and w-formulas as can be seen by the following examples.

$$\begin{aligned}
&w.\mathbf{h}\{w:=0\} \equiv 0.\mathbf{h} \\
&(\exists(w)\mathbf{A})\{w:=0\} \equiv \exists(0)\mathbf{A}
\end{aligned}$$

The expressions on the right-hand-side are not w-expressions although all left-hand-side expressions are. Since we intend to perform computations of w-formulas by doing proofs containing only w-formulas we have to define the substitution in w-expressions to be closed in w-expressions. Although we do not intend to reason directly within a calculus of w-formulas a computer-assisted theorem prover can be simplified by reasoning entirely within the w-calculus. The human interface to the user will then either introduce standard variables by means of abbreviations or perform the two-way translations between the standard TP and w-calculus which we can call w-TP.

We have decided here to embed w-calculus within TP. We are preparing a detailed paper on w-TP discussing its semantics and proof-theory as well as abbreviations for multiple variables. To give the reader the flavor of quantifier rules of w-TP we prove them here within TP.

The w-expression obtained by the substitution of the w-term **b** for the variable w in the w-term **a** or w-formula **A** is denoted by $\mathbf{a}(\mathbf{b})$ or $\mathbf{A}(\mathbf{b})$. We adopt the standard predicate call conventions and, for instance, write $\mathbf{A}([\mathbf{a},\mathbf{b}])$ as $\mathbf{A}(\mathbf{a},\mathbf{b})$. The substitution is defined by induction on the construction of w-expressions.

$$\begin{aligned}
&0(\mathbf{b}) \equiv 0 \\
&[\mathbf{a},\mathbf{b}](\mathbf{c}) \equiv [\mathbf{a}(\mathbf{c}),\mathbf{b}(\mathbf{c})] \\
&w\alpha(w\beta) \equiv w\beta\alpha \\
&w\alpha(0) \equiv 0 \\
&w(\mathbf{a},\mathbf{b}) \equiv [\mathbf{a},\mathbf{b}] \\
&w.\mathbf{h}\alpha(\mathbf{a},\mathbf{b}) \equiv w\alpha(\mathbf{a}) \\
&w.\mathbf{t}\alpha(\mathbf{a},\mathbf{b}) \equiv w\alpha(\mathbf{b}) \\
&(\mathbf{a} = \mathbf{b})(\mathbf{c}) \equiv \mathbf{a}(\mathbf{c}) = \mathbf{b}(\mathbf{c}) \\
&P(\mathbf{a})(\mathbf{b}) \equiv P(\mathbf{a}(\mathbf{b})) \\
&(\neg\ \mathbf{A})(\mathbf{a}) \equiv \neg\ \mathbf{A}(\mathbf{a})
\end{aligned}$$

$$(\mathbf{A} \,\&\, \mathbf{B})(\mathbf{a}) \equiv \mathbf{A}(\mathbf{a}) \,\&\, \mathbf{B}(\mathbf{a})$$
$$(\mathbf{A} \vee \mathbf{B})(\mathbf{a}) \equiv \mathbf{A}(\mathbf{a}) \vee \mathbf{B}(\mathbf{a})$$
$$(\exists\mathbf{A})(\mathbf{a}) \equiv \exists\mathbf{A}(w.\mathbf{h},\, \mathbf{a}(w.\mathbf{t})).$$

We have the following theorem.

Substitution Theorem (ST): If **a**, **b**, and **c** are w-terms and **A** is a w-formula then

$$\mathbf{a}(\mathbf{b})(\mathbf{c}) \equiv \mathbf{a}(\mathbf{b}(\mathbf{c})) \tag{2}$$
$$\mathbf{A}(\mathbf{b})(\mathbf{c}) \equiv \mathbf{A}(\mathbf{b}(\mathbf{c})) \tag{3}$$
$$\vdash \mathbf{a}(\mathbf{b}) = \mathbf{a}\{w{:=}\mathbf{b}\} \tag{4}$$
$$\vdash \mathbf{A}(\mathbf{b}) \leftrightarrow \mathbf{A}\{w{:=}\mathbf{b}\} \tag{5}$$

The theorem is proven by a straight-forward induction on the construction of w-terms and formulas, although there are quite a few cases to be considered. Just for the illustration we show one case of (3) and one case of (5). A case of (3):

$$\begin{aligned}&(\exists\mathbf{A})(\mathbf{b})(\mathbf{c}) \equiv (\exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(w.\mathbf{t}))(\mathbf{c}) \equiv \exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(w.\mathbf{t}))(w.\mathbf{h},\, \mathbf{c}(w.\mathbf{t})) \equiv\\ &\exists\mathbf{A}([\,w.\mathbf{h},\, \mathbf{b}(w.\mathbf{t})](w.\mathbf{h},\, \mathbf{c}(w.\mathbf{t}))) \equiv \exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(w.\mathbf{t})(w.\mathbf{h},\, \mathbf{c}(w.\mathbf{t}))) \equiv\\ &\exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(w.\mathbf{t}(w.\mathbf{h},\, \mathbf{c}(w.\mathbf{t})))) \equiv \exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(\mathbf{c}(w.\mathbf{t}))) \equiv\\ &\exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(\mathbf{c})(w.\mathbf{t})) \equiv (\exists\mathbf{A})(\mathbf{b}(\mathbf{c})).\end{aligned}$$

We have arranged the chain of symbolic identities ($\equiv$) in such a way that the first formula on a new line has been obtained from the last formula on the previous line by the use of (2) or by the use of induction hypothesis.

The interesting case of (5) is the following one.

$$\begin{aligned}&(\exists\mathbf{A})(\mathbf{b}) \equiv \exists\mathbf{A}(w.\mathbf{h},\, \mathbf{b}(w.\mathbf{t})) \leftrightarrow\\ &\exists\mathbf{A}\{w{:=}[\,w.\mathbf{h},\, \mathbf{b}\{w{:=}w.\mathbf{t}\}]\} \leftrightarrow \exists\mathbf{x}\, \mathbf{A}\{w{:=}[\,w.\mathbf{h},\, \mathbf{b}\{w{:=}w.\mathbf{t}\}]\}\{w{:=}[\mathbf{x},\, w]\} \leftrightarrow\\ &\exists\mathbf{x}\, \mathbf{A}\{w{:=}[\mathbf{x},\, \mathbf{b}]\} \leftrightarrow \exists(\mathbf{b})\mathbf{A} \equiv (\exists\mathbf{A})\{w{:=}\mathbf{b}\}\end{aligned}$$

As an example consider the formula

$$\mathbf{A} \equiv x = y \vee \exists z(x \in z \,\&\, \exists y(z \in y \,\&\, x = y))$$

where $x \in z$ is an abbreviation for $\in(x,\, z)$. The w-equivalent of **A** is the w-formula **B**

$$\mathbf{B} \equiv w.\mathbf{h} \in w.\mathbf{t} \vee \exists(w.\mathbf{t}.\mathbf{h} \in w.\mathbf{h} \,\&\, \exists(w.\mathbf{t}.\mathbf{h} \in w.\mathbf{h} \,\&\, w.\mathbf{t}.\mathbf{t}.\mathbf{h} = w.\mathbf{h})).$$

Note that the same free variable $w.\mathbf{h}$ becomes $w.\mathbf{t}.\mathbf{h}$ within the scope of one existential quantifier and $w.\mathbf{t}.\mathbf{t}.\mathbf{h}$ within two quantifiers. The variable $w.\mathbf{h}$ occurring within the first quantifier corresponds to the bound variable z of **A**. The same variable becomes $w.\mathbf{t}.\mathbf{h}$ within two quantifiers. We can say that existential quantifiers *push* all free variables down the list of variables.

The substitution $\mathbf{A}\{x{:=}y\}$ can be performed only after the bound variable y has been renamed

$$\mathbf{A}\{x{:=}y\} \equiv y = y \vee \exists z(y \in z \,\&\, \exists v(z \in v \,\&\, y = v)).$$

There is no problem with the corresponding substitution in **B**

$$\mathbf{B}(w.\mathbf{t},\, w.\mathbf{t}) \equiv w.\mathbf{t} \in w.\mathbf{t} \vee \exists(w.\mathbf{t}.\mathbf{t} \in w.\mathbf{h} \,\&\, \exists(w.\mathbf{t}.\mathbf{h} \in w.\mathbf{h} \,\&\, w.\mathbf{t}.\mathbf{t}.\mathbf{t} = w.\mathbf{h})).$$

The counterpart of Substitution axioms of predicate logic is the following schema of theorems.

Theorem: If **a** is a w-term and **A** a w-formula then

$$\vdash \mathbf{A}(\mathbf{a},\, w) \rightarrow \exists\mathbf{A}.$$

Proof: We start with an instance of the substitution axiom of TP

$$\vdash \mathbf{A}\{w{:=}[\mathbf{x},\, w]\}\{\mathbf{x}{:=}\mathbf{a}\} \rightarrow \exists\mathbf{x}\, \mathbf{A}\{w{:=}[\mathbf{x},\, w]\}.$$

Simplifying the substitution and employing the definition of $\exists$ yields

$$\vdash \mathbf{A}\{w{:=}[\mathbf{a},\, w]\} \rightarrow \exists\mathbf{A}.$$

The theorem follows by an application of (5).

The counterpart of $\exists$-introduction of TP is given by the following derived rule of inference.

Variable-free $\exists$-Introduction: If $\mathbf{A}$, and $\mathbf{B}$ are w-formulas and $\vdash \mathbf{A}(w.\mathbf{h},\ w.\mathbf{t}) \rightarrow \mathbf{B}(w.\mathbf{t})$ then also $\vdash \exists\mathbf{A} \rightarrow \mathbf{B}$.

Proof: Under the assumption $w = [x,\ y]$ we have $w.\mathbf{t} = y$ and $[w.\mathbf{h},\ w.\mathbf{t}] = w = [x,\ y]$. Thus also

$$\mathbf{B}(w.\mathbf{t}) \leftrightarrow \mathbf{B}\{w{:=}w.\mathbf{t}\} \leftrightarrow \mathbf{B}\{w{:=}y\}$$

and

$$\mathbf{A}(w.\mathbf{h},\ w.\mathbf{t}) \leftrightarrow \mathbf{A}\{w{:=}[w.\mathbf{h},\ w.\mathbf{t}]\} \leftrightarrow \mathbf{A}\{w{:=}[x,\ y]\}.$$

Using the assumption of the theorem we obtain

$$\vdash\ w = [x,\ y] \rightarrow (\mathbf{A}(w.\mathbf{h},\ w.\mathbf{t}) \rightarrow \mathbf{B}(w.\mathbf{t})).$$

From this we have

$$\vdash\ w = [x,\ y] \rightarrow (\mathbf{A}\{w{:=}[x,\ y]\} \rightarrow \mathbf{B}\{w{:=}y\}).$$

We observe that the variable w does not occur free in the consequent. We substitute $w{:=}[x,\ y]$ and use modus ponens to obtain $\exists w$-introduction and modus-ponens to obtain

$$\vdash\ \mathbf{A}\{w{:=}[x,\ y]\} \rightarrow \mathbf{B}\{w{:=}y\}.$$

We observe again that the variable x does not occur in the consequent. We use $\exists x$-introduction and replace y by w.

$$\vdash\ \exists x\, \mathbf{A}\{w{:=}[x,\ w]\} \rightarrow \mathbf{B}$$

The conclusion of the theorem follows from the definition of $\exists$ by a suitable renaming of x.

The reader will note that w-TP has a great potential for the proof-theory as well as for the automatic theorem proving as there are no clashes of free and bound variables and no eigen-variable restrictions on $\exists$-introductions.

5. Environments and Environment Equations.

Computations in R-Maple were defined with the help of substitutions. To compute the predicate call $P(\mathbf{a})$ we used its definition $P(x) \leftrightarrow \mathbf{A}$ and computed the formula $\mathbf{A}\{x{:=}\mathbf{a}\}$ obtained from the body of predicate by a substitution. As every programmer knows, substitutions are fine in logic, but they are expensive to implement on a computer. Since the Algol-60 times one has computed a *pure* program, i.e. never changing formula, $\mathbf{A}$ in the environment $\mathbf{a}$ *binding* the variable x. This environment is carried around the formula $\mathbf{A}$ during the computation. It is modified as new bindings for variables are computed. Environments are extended upon entrance to the scope of an existential quantifier or when executing a predicate call. Environment contraction occurs when existential variable has been found or the predicate call has been completed.

Environments of R^{+}–Maple form a certain subset of w-terms. If $\mathbf{a}$ is an environment then the w-formula $w = \mathbf{a}$ is called an *environment equation*. This *binds* the variable w to the w-term $\mathbf{a}$. Environment equation $w = \mathbf{a}$ specifies a class of pairs as values for w such that the equation $w = \mathbf{a}$ is satisfied. Before we can specify environments we need some auxiliary definitions.

A projection α is an *ancestor* of a projection β iff there is a projection $\beta' \not\equiv \epsilon$ such that $\beta \equiv \alpha\beta'$. Two projections are in the *ancestor relation* iff one is an ancestor of the other. A projection β is to the *right* of a projection α iff there are projections γ, β' and α' such that $\alpha \equiv \gamma.\mathbf{h}\alpha'$ and $\beta \equiv \gamma.\mathbf{t}\beta'$.

A term $\mathbf{a}$ is called a *semi-w-term* iff it is composed from 0, pointers, and ordinary variables by pairing. Semi-w-terms are like w-terms but they may also contain variables other than w but no projections of these variables.

Let us define α-parts of semi-w-terms. The α-part of the semi-w-term $\mathbf{a}$, if it exists, will be denoted by $\mathbf{a}/\alpha$. Parts are defined by the induction on the construction of projections:

i. $\mathbf{a}/\epsilon \equiv \mathbf{a}$,

ii. if $\mathbf{a}/\alpha \equiv [\mathbf{b}, \mathbf{c}]$ then $\mathbf{a}/\alpha.\mathbf{h} \equiv \mathbf{b}$ and $\mathbf{a}/\alpha.\mathbf{t} \equiv \mathbf{c}$,

iii. there are no parts of the term **a** other then those defined by i. and ii.

Note that if $\mathbf{a}/\alpha$ is a part of **a**, all ancestors β of α are also parts of **a**, i.e. $\mathbf{a}/\beta$ is defined. By the induction on the projections one can easily prove the following theorem.

Part Theorem: If for a semi-w-term **a** the part $\mathbf{a}/\alpha$ exists then

$$\vdash\ w = \mathbf{a} \rightarrow w\alpha = \mathbf{a}/\alpha.$$

Informally, the Part theorem says that for all values w satisfying the environment equation $w = \mathbf{a}$ the pointer $w\alpha$ selects the part $\mathbf{a}/\alpha$.

The pointer $w\alpha$ is a *self-pointer* in the semi-w-term **a** iff $\mathbf{a}/\alpha \equiv w\alpha$. The pointer $w\beta$ occurring in the w-term **a** at the position α, i.e. $\mathbf{a}/\alpha \equiv w\beta$ *points to the right* iff β is to the right of α. A pointer $w\alpha$ is *admissible* in the w-term **a** iff there is a number n such that $w\alpha$ is admissible in **a** with n jumps. The admissibility with jumps is defined as follows.

i. If α is a part of **a**, i.e. $\mathbf{a}/\alpha$ exists then $w\alpha$ is admissible in **a** with 0 jumps.

ii. If $\mathbf{a}/\alpha \equiv w\beta$ and the pointer $w\beta\beta'$ such that $\beta' \not\equiv \epsilon$ is admissible in **a** with n jumps then the pointer $w\alpha\beta'$ is admissible in **a** with n+1 jumps.

A pointer not admissible in a w-term is a *dangling* pointer. Figure 3 shows the graph representation of the following w-terms

a) $[[w.\mathbf{h}.\mathbf{t},\ w.\mathbf{h}.\mathbf{t}.\mathbf{h},\ w.\mathbf{t}],\ 0,\ w.\mathbf{t}.\mathbf{t}]$,

b) $[w, 0]$,

c) $[[[w.\mathbf{h},\ w.\mathbf{h}.\mathbf{h}.\mathbf{h}],\ w.\mathbf{t}.\mathbf{h}],\ w.\mathbf{t}]$.

The pointer $w.\mathbf{h}.\mathbf{t}$ in the term a) points to the right, $w.\mathbf{h}.\mathbf{t}.\mathbf{h}$ is a self-pointer, and $w.\mathbf{h}.\mathbf{h}.\mathbf{t}.\mathbf{h}$ is an admissible pointer with two jumps selecting 0. The pointer w in the head of b) and the pointer $w.\mathbf{h}$ in the part $.\mathbf{h}.\mathbf{h}.\mathbf{h}$ of c) point to their ancestors. Terms b) and c) thus contain *loops*. Pointers $w.\mathbf{h}.\mathbf{h}.\mathbf{h}.\mathbf{t}$ and $w.\mathbf{t}.\mathbf{h}$ both point to the same non-existent part of the term c). So they are both dangling.

The w-term $[w, 0]$ is the *empty* environment. A w-term **a** is a *proper* environment iff each pointer occurring in **a** is admissible and it is either a self-pointer or it points to the right. An *environment* is either empty or proper. The w-term a) in the figure 3 is a proper environment, the term b) is empty and the term c) is not an environment because it contains loops, pointers to the left, and also dangling pointers.

The environment equation for the empty environment $w = [w, 0]$ will be abbreviated as **fail**. The equation **fail** can be never satisfied as we have

$$\vdash\ \neg\ \mathbf{fail}. \tag{1}$$

The equation **fail** can be understood as *falsehood*. The proper environment w, containing only a self-pointer, is the most inclusive environment. Its equation $w = w$ is satisfied by any value w, i.e.

$$\vdash\ w = w$$

Thus $w = w$ can be understood as *truth*. Proper environments can be always satisfied , i.e.

$$\vdash\ \exists w\ w = \mathbf{a}.$$

This is an immediate consequence of the Satisfiability theorem (5) proven below.

Although it is true that environments with cyclical pointers are empty, environment equations for w-terms with left-pointers can still be satisfied. The w-term $[w.\mathbf{h}, w.\mathbf{h}]$ is such an example. It is equivalent to the environment $[w.\mathbf{t}, w.\mathbf{t}]$. The reason for excluding pointers to the left will become obvious once the computation rules for environment contraction are given in section (7). A w-term $[\mathbf{a}, \mathbf{b}]$ with a pointer in **b** leading to **a** could be then contracted to **b** by popping the current frame **a** leaving a dangling pointer. This phenomenon is well-known from the implementation of Algol-like languages with pointers. Dangling pointers are not dangerous just because they seemingly lead to nowhere. The pointer $w.\mathbf{h}$ is still defined, eventhough $w = 0$. Both equations $w = w.\mathbf{h}$ and $w = [w.\mathbf{t}.\mathbf{t},\ w.\mathbf{t}]$ can be still satisfied by $w = 0$ and, say, $w = [0, 0]$ respectively. Dangling pointers are excluded because they destroy the validity of the Environment theorem on which the computation of R$^+$–Maple is based. The Environment theorem is concerned with the solution of a system of equations.

> **Environment Theorem**: If $\mathbf{a}$ is an environment and $\mathbf{b}$, $\mathbf{c}$ are w-terms such that all pointers occurring in them are admissible in $\mathbf{a}$ provided $\mathbf{a}$ is proper then we can effectively find an environment $\mathbf{d}$ such that
>
> $\vdash\ w = \mathbf{a}\ \&\ \mathbf{b} = \mathbf{c} \leftrightarrow w = \mathbf{d}$.
>
> Moreover, if $\mathbf{d}$ is a proper environment then all pointers admissible in $\mathbf{a}$ are admissible in $\mathbf{d}$.

The Environment theorem says that a proper solution $\mathbf{d}$ is a *refinement* of $\mathbf{a}$. The proof of the Environment theorem includes a presentation of an algorithm for finding the environment $\mathbf{d}$ together with the proof of termination and of correctness. The proof is quite lengthy and technical. We shall just present the algorithm below and refer the interested reader to the paper [12] under preparation.

Had we allowed w-terms with dangling pointers as environments, we would not be able to find a single w-term $\mathbf{d}$ solving the system

$\vdash\ w = [w.\mathbf{t}.\mathbf{h},\ w.\mathbf{t}]\ \&\ w = [w.\mathbf{t}.\mathbf{t},\ w.\mathbf{t}] \leftrightarrow w = \mathbf{d}$.

Any solution with a pair as a tail part will have to be of the form $\mathbf{d} \equiv [w.\mathbf{t}.\mathbf{t},\ w.\mathbf{t}.\mathbf{t},\ w.\mathbf{t}.\mathbf{t}]$. This w-term excludes the value $[0, 0]$ satisfying the left-hand-side. The only way to include $[0, 0]$ in a single w-term is to have a self-pointer $\mathbf{d} \equiv [\mathbf{a},\ w.\mathbf{t}]$ with $\mathbf{a}$ being either 0 or a self-pointer. This would include the value $[0, 0, 0, 0]$ which fails the left-hand-side.

Before we can give the Environment solution algorithm we have to define the notion of replacement of parts of semi-w-terms. Let us denote by $\mathbf{a}{<}\alpha{:=}\mathbf{b}{>}$ the term obtained by the *replacement* of the part α of the semi-w-term $\mathbf{a}$ by the term $\mathbf{b}$. The replacement is defined as follows.

i. $\mathbf{a}{<}\epsilon{:=}\mathbf{b}{>} \equiv \mathbf{b}$,

ii. If $\mathbf{a}/\alpha$ is a pair then

$$\mathbf{a}{<}\alpha.\mathbf{h}{:=}\mathbf{b}{>} \equiv \mathbf{a}{<}\alpha{:=}[\mathbf{b},\ \mathbf{a}/\alpha.\mathbf{t}]{>}$$
$$\mathbf{a}{<}\alpha.\mathbf{t}{:=}\mathbf{b}{>} \equiv \mathbf{a}{<}\alpha{:=}[\mathbf{b}/\alpha.\mathbf{h},\ \mathbf{b}]{>}$$

We have the following theorem.

> **Replacement Theorem**: For any semi-w-term $\mathbf{a}$ with an α part
>
> $\vdash\ \mathbf{a}/\alpha = \mathbf{b} \rightarrow \mathbf{a} = \mathbf{a}{<}\alpha{:=}\mathbf{b}{>}$. (2)

The Replacement theorem is proven by the induction on α.

> **Environment Solution Algorithm**: To solve the system
>
> $\vdash\ w = \mathbf{a}\ \&\ \mathbf{b} = \mathbf{c} \leftrightarrow w = \mathbf{d}$
>
> consider the following cases.

i. The environment $\mathbf{a}$ is empty, or one of the terms $\mathbf{b}$ and $\mathbf{c}$ is 0 and the other a pair, or the terms $\mathbf{b}$ and $\mathbf{c}$ are pointers in the ancestor relation. Set $\mathbf{d} \equiv [w, 0]$.

ii. The equation $\mathbf{b} = \mathbf{c}$ is of the form $0 = 0$ or $w\alpha = w\alpha$. Set $\mathbf{d} \equiv \mathbf{a}$.

iii. The equation $\mathbf{b} = \mathbf{c}$ is of the form $[\mathbf{b}_1, \mathbf{b}_2] = [\mathbf{c}_1, \mathbf{c}_2]$. Solve the system of equations

$\vdash\ w = \mathbf{a}\ \&\ \mathbf{b}_2 = \mathbf{c}_2 \leftrightarrow w = \mathbf{d}'$
$\vdash\ w = \mathbf{d}'\ \&\ \mathbf{b}_1 = \mathbf{c}_1 \leftrightarrow w = \mathbf{d}$.

iv. The equation $\mathbf{b} = \mathbf{c}$ is of the form $w\alpha = \mathbf{c}$. If $\mathbf{c} \equiv w\beta$ we can assume after possibly reversing the equation that β is to the right of α. There are three subcases all yielding an auxiliary environment $\mathbf{d}'$.

a) The pointer $w\alpha$ contains a jump or $\mathbf{a}/\alpha$ points to the right, i.e. $\alpha \equiv \alpha_1\alpha_2$ and $\mathbf{a}/\alpha_1 \equiv w\gamma$ for some projections α_1, α_2, $\gamma \not\equiv \alpha$. Solve the system

$$\vdash\ w = \mathbf{a}\ \&\ w\gamma\alpha_2 = \mathbf{c} \leftrightarrow w = \mathbf{d}'.$$

b) The pointer $w\alpha$ is a self-pointer, i.e. $\mathbf{a}/\alpha \equiv w\alpha$. If $\mathbf{c}$ is a pair $[\mathbf{c}_1, \mathbf{c}_2]$ then solve the system

$$\vdash\ w = \mathbf{a}{<}\alpha{:=}[w\alpha.\mathbf{h},\ w\alpha.\mathbf{t}]{>}\ \&\ [w\alpha.\mathbf{h},\ w\alpha.\mathbf{t}] = [\mathbf{c}_1, \mathbf{c}_2] \leftrightarrow w = \mathbf{d}'.$$

Otherwise, if $\mathbf{c} \equiv 0$ set $\mathbf{d}' \equiv \mathbf{a}{<}\alpha{:=}0{>}$ else $\mathbf{c}$ is a pointer and set $\mathbf{d}' \equiv \mathbf{a}$.

c) $\mathbf{a}/\alpha$ must be 0 or a pair. Solve

$$\vdash\ w = \mathbf{a}\ \&\ \mathbf{a}/\alpha = \mathbf{c} \leftrightarrow w = \mathbf{d}'.$$

If $\mathbf{d}'$ is a proper environment and $\mathbf{c} \equiv w\beta$ set $\mathbf{d} \equiv \mathbf{d}'{<}\alpha{:=}w\beta{>}$. Set $\mathbf{d} \equiv \mathbf{d}'$ otherwise.

Consider the following system:

$$\vdash\ w = [[[0,\ w.\mathbf{h}.\mathbf{h}.\mathbf{t}],\ 0,\ w.\mathbf{t}],\ w.\mathbf{t}]\ \&\ w.\mathbf{h}.\mathbf{h} = w.\mathbf{t} \leftrightarrow w = \mathbf{d}.$$

The solution obtained by the environment algorithm is

$$\mathbf{d} \equiv [[w.\mathbf{t},\ 0,\ w.\mathbf{t}],\ 0,\ w.\mathbf{t}.\mathbf{t}].$$

The two environments are shown in the figure 4.

We now prove some technical theorems concerned with environments. The theorems are necessary for the proofs of computation rules of R$^+$-Maple. Readers interested only in the computation of R$^+$-Maple and not in its soundness can skip the rest of the section.

Copy Theorem: If $\mathbf{a}$ is a proper environment such that $\mathbf{a}/\alpha \equiv w\beta$, $\alpha \not\equiv \beta$ and $\mathbf{a}/\beta$ exists then $\mathbf{b} \equiv \mathbf{a}{<}\alpha{:=}\mathbf{a}/\beta{>}$ is a proper environment and

$$\vdash\ w = \mathbf{b} \leftrightarrow w = \mathbf{a}. \tag{3}$$

Proof: We can show by induction on the number of jumps that every pointer admissible in $\mathbf{a}$ is admissible in $\mathbf{b}$ (with one less jump if the pointer jumps through α). All pointers occurring in $\mathbf{b}$ occur in $\mathbf{a}$ so they are admissible in $\mathbf{a}$ and also in $\mathbf{b}$. All pointers in the part $\mathbf{b}/\alpha \equiv \mathbf{a}/\beta$ point to the right, the other pointers in $\mathbf{b}$ occur in $\mathbf{a}$ so they are either self-pointers or point to the right. Thus $\mathbf{b}$ is a proper environment.

Now we prove the formula (3). ($\rightarrow$) Assume $w = \mathbf{b}$. By the Part theorem we have

$$\mathbf{a}/\alpha \equiv w\beta = \mathbf{b}/\beta \equiv \mathbf{a}/\beta.$$

The Replacement theorem applies and we have $\mathbf{a} = \mathbf{b}$ and thus also $w = \mathbf{a}$. ($\leftarrow$) Assume $w = \mathbf{a}$. We have

$$\mathbf{a}/\alpha \equiv w\beta = \mathbf{a}/\beta.$$

So again by the Replacement theorem we have $\mathbf{a} = \mathbf{b}$ and $w = \mathbf{b}$.

Self-pointer Theorem: If $\mathbf{x}$ is a variable not occurring in the semi-w-term $\mathbf{a}$, $w\alpha$ is a self-pointer, i.e. $\mathbf{a}/\alpha \equiv w\alpha$, and if $\mathbf{b}$ is the semi-w-term obtained from $\mathbf{a}$ by replacing all pointers $w\alpha$ in $\mathbf{a}$ by $\mathbf{x}$ then

$$\vdash\ \exists\mathbf{x}\ w = \mathbf{b} \leftrightarrow w = \mathbf{a}. \tag{4}$$

Proof: ($\rightarrow$) Assume $w = \mathbf{b}$. By the Part theorem we have $w\alpha = \mathbf{x}$. By the Equality theorem of the predicate calculus $w = \mathbf{a}$. ($\leftarrow$) Because $\mathbf{a} \equiv \mathbf{b}\{\mathbf{x}{:=}w\alpha\}$ this implication is a Substitution axiom of predicate calculus.

Satisfiability Theorem: For a proper environment $\mathbf{a}$ we can find a semi-w-term $\mathbf{b}$ with no pointers and containing variables x_1, x_2, $\cdots$, x_n ($n \geq 0$) such that

$$\vdash\ \exists x_1 \exists x_2 \cdots \exists x_n\ w = \mathbf{b} \leftrightarrow w = \mathbf{a}. \tag{5}$$

Note that we have

$\vdash\ w = \mathbf{b} \rightarrow w = \mathbf{a}$.

If we substitute **b** for w we obtain

$\vdash\ \mathbf{b} = \mathbf{b}\{w{:=}\mathbf{b}\} \rightarrow \mathbf{b} = \mathbf{a}\{w{:=}\mathbf{b}\}$.

And since **b** does not contain the variable w we have

$\vdash\ \mathbf{b} = \mathbf{a}\{w{:=}\mathbf{b}\}$

by modus ponens. Thus the term **b** solves the environment equation $w = \mathbf{a}$. The ($\leftarrow$) part of the theorem (5) says that any solution of the equation will have to be obtained by a substitution for the variables in **b**. In this sense the term **b** is the most general term solving the equation $w = \mathbf{a}$.

The proof of the Satisfiability theorem is by induction on the number of occurrences of pointers in **a** which do not occur also as self-pointers. If all pointers in **a** directly point to self-pointers then we obtain the term **b** by repeatedly applying the Self-pointer theorem, replacing the self-pointers by the variables x_1, x_2, etc. If there is a pointer in **a** which does not occur also as a self-pointer then we select the rightmost occurrence $\mathbf{a}/\alpha \equiv w\beta$. The part $\mathbf{a}/\beta$ must exist, and it cannot contain pointers which are not also self-pointers, so $\mathbf{a}' \equiv \mathbf{a}{<}\alpha{:=}\mathbf{a}/\beta{>}$ is by the Copy theorem a proper environment such that

$$\vdash\ w = \mathbf{a} \leftrightarrow w = \mathbf{a}' \tag{6}$$

and **a**' contains one less occurrence of a pointer which is not also a self-pointer. Induction hypothesis applies and because of (6) the term **b** also satisfies (5).

The most general solution of the environment equation

$w = [\,[\,w.\mathbf{t},\ 0,\ w.\mathbf{t}\,],\ 0,\ w.\mathbf{t}.\mathbf{t}\,]$

for the second environment from the figure 4 is

$w = [\,[\,[0,\ x_1],\ 0,\ 0,\ x_1],\ 0,\ x_1].$

It is obtained by a two-fold application of the Copy theorem to remove the two pointers to the right in **c** and by a single use of the Self-pointer theorem to replace the only self-pointer of **c** by x_1.

Admissibility Lemma: If **b** is a w-term and **a** is a proper environment then every pointer $w\alpha$ is admissible in **a** with n jumps iff $w.\mathbf{t}.\alpha$ is admissible in the w-term $[\mathbf{b},\ \mathbf{a}(w.\mathbf{t})]$ with n jumps iff $w.\mathbf{h}\alpha$ is admissible in $[\mathbf{a}(w.\mathbf{h}),\ \mathbf{b}]$ with n jumps.

The lemma has a straightforward, although tedious proof, by the induction on the number of jumps.

Extension Lemma: If **a** is a proper environment and **b** a w-term with all its pointers admissible in **a** then both terms $\mathbf{c} \equiv [\,w.\mathbf{h},\ \mathbf{a}(w.\mathbf{t})]$ and $\mathbf{d} \equiv [\mathbf{b}(w.\mathbf{t}),\ \mathbf{a}(w.\mathbf{t})]$ are proper environments. If **b** is a proper environment then the term $\mathbf{e} \equiv [\mathbf{b}(w.\mathbf{h}),\ \mathbf{a}(w.\mathbf{t})]$ is a proper environment.

Proof: By the admissibility lemma all pointers occurring in the tails of **c**, **d**, and **e** are admissible. Moreover, all of them have the form $w.\mathbf{t}\beta$. If $\mathbf{c}/.\mathbf{t}\alpha \equiv w.\mathbf{t}\beta$ then $\mathbf{a}/\alpha \equiv w\beta$. Consequently, either $\alpha \equiv \beta$ or β is to the right of α. Thus also all pointers in the tail of **c** are either self-pointers or they point to the right. The same holds of tails of **d**, and **e**. Similar argument can be used with the head of **e**. Thus **e** is a proper environment if **b** is. The only pointer in the head of **c** is a self-pointer which is admissible with 0 jumps. Thus **c** is a proper environment. All pointers in the head of **d** have the form $\mathbf{d}/.\mathbf{h}\alpha \equiv w.\mathbf{t}\beta$ so they point to the right. We then also have $\mathbf{b}/\alpha \equiv w\beta$ with $w\beta$ admissible in **a** so by the Admissibility lemma, $w.\mathbf{t}\beta$ is admissible in **d**. Thus also **d** is a proper environment.

Contraction Lemma: If $\mathbf{c} \equiv [\mathbf{b},\ \mathbf{a}']$ is a proper environment then there is a proper environment **a** such that $\mathbf{a}' \equiv \mathbf{a}(w.\mathbf{t})$ and

$$\vdash\ w = \mathbf{a} \rightarrow \exists x\ x = \mathbf{b}\{w{:=}[x,\ w]\}. \tag{7}$$

Proof: Every pointer occurring in $\mathbf{a}'$ must be of the form $w.\mathbf{t}\alpha$ so the w-term $\mathbf{a}$ is obtained from $\mathbf{a}'$ by replacing all pointers $w.\mathbf{t}\alpha$ by $w\alpha$. By the Admissibility lemma all pointers occurring in $\mathbf{a}$ are admissible. By a similar reasoning as in the proof of the Extension lemma we show that they either point to the right or are self-pointers. Thus $\mathbf{a}$ is a proper environment.

The formula (7) is proven as the Satisfiability theorem was. We first eliminate from $[\mathbf{b}, \mathbf{a}(w.\mathbf{t})]$ all pointers of the form $w.\mathbf{h}\alpha$ which do not occur also as self-pointers by a repeated application of Copy theorem. We obtain a new term $\mathbf{b}'$ such that

$$\vdash\ w = [\mathbf{b}', \mathbf{a}(w.\mathbf{t})] \leftrightarrow w = [\mathbf{b}, \mathbf{a}(w.\mathbf{t})]$$

holds. Next we apply the Self-pointer theorem repeatedly to remove all self-pointers of the form $w.\mathbf{h}\alpha$ from the term $\mathbf{b}'$. We obtain a new semi-w-term $\mathbf{b}''$ containing variables $x_1, \cdots, x_n$ with all pointers only of the form $w.\mathbf{t}\alpha$. Thus $\mathbf{b}'' \equiv \mathbf{d}\{w{:=}w.\mathbf{t}\}$ for a certain semi-w-term $\mathbf{d}$. We have from the Self-pointer theorem

$$\vdash\ w = [\mathbf{d}\{w{:=}w.\mathbf{t}\}, \mathbf{a}(w.\mathbf{t})] \rightarrow w = [\mathbf{b}, \mathbf{a}(w.\mathbf{t})].$$

This can be weakenned to

$$\vdash\ w = [\mathbf{d}\{w{:=}w.\mathbf{t}\}, \mathbf{a}(w.\mathbf{t})] \rightarrow w.\mathbf{h} = \mathbf{b}.$$

Let us substitute $w{:=}[\mathbf{d}, w]$ in the last formula. After the simplifications $[\mathbf{d}, w].\mathbf{h} = \mathbf{d}$ and $[\mathbf{d}, w].\mathbf{t} = w$ we obtain

$$\vdash\ [\mathbf{d}, w] = [\mathbf{d}, \mathbf{a}] \rightarrow \mathbf{d} = \mathbf{b}\{w{:=}[\mathbf{d}, w]\}.$$

The theorem (7) follows after the simplification of the antecedent and an application of $\exists x$-introduction right.

6. Computation of Formulas of TP.

Before a formula of TP can be computed in R$^+$-Maple it has to be converted into a suitable w-formula. In this section we show how to convert formulas and predicate definitions of TP into the format of R$^+$-Maple. The rewriting rules presented in the next section assume that the computed w-formulas have a certain format. We also require predicate definitions to be in a special form. The formulas of R-Maple are w-formulas with all identities of the form of environment equations $w = \mathbf{a}$ for an environment $\mathbf{a}$. We present below an algorithm for the conversion of TP formulas into w-formulas which replaces all identities by equivalent environment equations. This will be done by a repeated use of the Environment theorem (section 5).

In section (3) we said that we require the predicates of R$^+$-Maple to be in the form $\vdash P(w) \leftrightarrow \mathbf{A}$ or in an implicational form. These forms are not directly suitable for computations in R$^+$-Maple. The reader is invited to inspect the computation rule for predicate calls (7.5) given in the next section and note that we would be forced to perform the substitution $\mathbf{A}(w.\mathbf{h})$. There are two objections to the substitution $\mathbf{A}(w.\mathbf{h})$. One of the reasons for the introduction of environments is to eliminate substitutions which are costly to implement. Here the substitution would occur in the pure code for the body of P. The second objection is that the substitution $\mathbf{A}(w.\mathbf{h})$ would destroy the environment equations as the only form of identities allowed in the formulas of R$^+$-Maple. These difficulties are overcome by requiring that the predicate definitions are of the form

$$\vdash\ w \neq 0 \,\&\, P(w.\mathbf{h}) \leftrightarrow \mathbf{A}$$
$$\vdash\ w \neq 0 \,\&\, P(w.\mathbf{h}) \leftarrow \mathbf{A}$$
$$\vdash\ w \neq 0 \,\&\, P(w.\mathbf{h}) \rightarrow \mathbf{A}$$

where the formula $\mathbf{A}$ is a formula of R$^+$-Maple with an additional restriction that the pointer $w.\mathbf{t}$ never occurs in arguments to predicate calls and occurs only as a self-pointer in environment equations. This restriction is necessary to guarantee that the body of a predicate accesses only its arguments. The conversion algorithm given below takes care of this. The predicate Len presented in section (3) will have the following form:

$$\vdash\ w \neq 0 \,\&\, Len(w.\mathbf{h}) \leftrightarrow w = [[0, 0], w.\mathbf{t}] \lor$$
$$w = [[[w.\mathbf{h}.\mathbf{h}.\mathbf{h}, w.\mathbf{h}.\mathbf{h}.\mathbf{t}], [0, w.\mathbf{h}.\mathbf{t}.\mathbf{t}]], w.\mathbf{t}] \,\&\, Len(w.\mathbf{h}.\mathbf{h}.\mathbf{t}, w.\mathbf{h}.\mathbf{t}.\mathbf{t}).$$

The same formula can be abbreviated as

$$\vdash\ w \neq 0 \;\&\; Len(w.\mathbf{h}) \leftrightarrow \textbf{case } w.\mathbf{h} \textbf{ of} \quad [0, 0] : \quad [[*, x], [0, y]] : Len(x, y).$$

We now describe the conversion of TP formulas into the formulas of R$^+$–Maple. Let us assume that we have eliminated all implications, equivalences, universal quantifiers, one-variable existential quantifiers as well as all introduced function symbols (including **.h** and **.t**) from the formula **A** we want to convert into a w-formula. We assume that the formula **A** does not contain the variable w. We first construct a semi-w-term $\mathbf{a} \equiv [\mathbf{a}', \mathbf{z}]$ where $\mathbf{z}$ is a variable not occurring in **A**. If **A** is closed we set $\mathbf{a}' \equiv \mathbf{z}$ otherwise we form the term $\mathbf{a}'$ from the complete list $\mathbf{x}_1 \mathbf{x}_2, \cdots, \mathbf{x}_n$ $(n>0)$ of all free variables of **A**. The term $\mathbf{a}'$ is constructed in such a way that it contains all variables $\mathbf{x}_i$ exactly once and nothing else. The shape of $\mathbf{a}'$ is not important. For a single variable x the term $\mathbf{a}'$ will have to be x. For three variables x, y, z we can group them either $[x, y, z]$ or $[[x, y], z]$, or even $[z, x, y]$. The important thing is that every variable $\mathbf{x}_i$ has a uniquely determined location α_i in $\mathbf{a}$, i.e. $\mathbf{a}/\alpha_i \equiv \mathbf{x}_i$. The conversion algorithm replaces first all variables $\mathbf{x}_i$ by their corresponding pointers $w\alpha_i$. Then it replaces existential quantifiers by one-variable quantifiers and finally it brings all identities into the form of environment equations.

We define a sequence of formulas $\mathbf{A}_i$, $0 \leq i \leq m$ such that $\mathbf{A}_0 \equiv \mathbf{A}$, $\mathbf{A}_m$ is a formula of R$^+$–Maple and for all i the following holds:

$$\vdash\ w = \mathbf{a} \;\&\; \mathbf{A} \leftrightarrow w = \mathbf{a} \;\&\; \mathbf{A}_i. \tag{1}$$

Note that (1) holds for $i = 0$. For i such that $0 \leq i < n$ we set

$$\mathbf{A}_{i+1} \equiv \mathbf{A}_i\{\mathbf{x}_i := w.\alpha_i\}.$$

Since $\vdash\ w = \mathbf{a} \rightarrow w.\alpha_i = \mathbf{x}_i$ we have by the Equality theorem of predicate calculus

$$\vdash\ w = \mathbf{a} \;\&\; \mathbf{A}_i \leftrightarrow w = \mathbf{a} \;\&\; \mathbf{A}_{i+1}.$$

This implies (1) for all $i \leq n$. The formula $\mathbf{A}_n$ does not contain variables other than w free.

For $i \geq n$ we obtain $\mathbf{A}_{i+1}$ from $\mathbf{A}_i$ by the elimination of one existential quantifier $\exists\mathbf{x}$. If $\mathbf{A}_i$ does not contain existential quantifiers then it is a w-formula and we set $j = i$. Otherwise, the formula $\mathbf{A}_{i+1}$ is obtained from $\mathbf{A}_i$ by a replacement of its subformula $\exists\mathbf{x}\mathbf{C}$ by the equivalent formula in the right-hand-side of

$$\vdash\ \exists\mathbf{x}\mathbf{C} \leftrightarrow \exists\mathbf{C}\{w := w.\mathbf{t}\}\{\mathbf{x} := w.\mathbf{h}\}.$$

As the consequence we have $\vdash\ \mathbf{A}_{i+1} \leftrightarrow \mathbf{A}_i$, this is enough to satisfy (1) for $n < i \leq j$. We select the subformula $\exists\mathbf{x}\mathbf{C}$ in such a way that it is not contained in another existential quantifier $\exists\mathbf{y}$. This restriction is necessary to prevent a later substitution

$$(\exists\mathbf{D})\{w := w.\mathbf{t}\} \equiv \exists(w.\mathbf{t})\mathbf{D}$$

when $\exists\mathbf{y}$ will be eliminated. The substitution obviously leads outside of w-formulas.

For $i \geq j$ we obtain $\mathbf{A}_{i+1}$ from $\mathbf{A}_i$ by transforming one identity into an environment equation. If $\mathbf{A}_i$ contains only environment equations then it is a formula of R$^+$–Maple and we set $m = i$. Otherwise, the formula $\mathbf{A}_{i+1}$ is obtained from the formula $\mathbf{A}_i$ by choosing one occurrence of an identity $\mathbf{c} = \mathbf{d}$ which is not an environment equation and replacing it by the environment equation $w = \mathbf{e}$. If the occurrence of identity $\mathbf{c} = \mathbf{d}$ is within the scope of p existential quantifiers $(p \geq 0)$ the environment $\mathbf{e}$ is found as the solution of the environment system

$$w = \mathbf{b}_p \;\&\; \mathbf{c} = \mathbf{d} \leftrightarrow w = \mathbf{e}$$

where the proper environment $\mathbf{b}_p$ is called the *main* environment for the scope of depth p.

Main environments are defined as follows. The environment for the scope 0: $\mathbf{b} \equiv \mathbf{b}_0$ is obtained from the semi w-term $\mathbf{a}$ by the replacement of all variables $\mathbf{x}_i$ by $w\alpha_i$ and $\mathbf{z}$ by $w.\mathbf{t}$. Since the only location of $\mathbf{x}_i$ is $\mathbf{a}/\alpha_i$ the term $\mathbf{b}$ is a proper environment. Note that $\mathbf{b}$ contains only self-pointers if **A** is open and is $[w.\mathbf{t}, w.\mathbf{t}]$ otherwise. We set

$$\mathbf{b}_{p+1} \equiv [w.\mathbf{h},\ \mathbf{b}_p(w.\mathbf{t})].$$

By the Extension lemma (section 5) all environments $\mathbf{b}_p$ are proper. It is an easy proof by induction on p to ascertain that all pointers in $\mathbf{A}_j$ occurring at the depth p are admissible in $\mathbf{b}_p$ in 0 jumps. Consequently, all pointers in the identity $\mathbf{c} = \mathbf{d}$ are also admissible and the Environment theorem (section 5) applies. Thus the environment $\mathbf{e}$ can be effectively found. To show that (1) is satisfied for $i+1$ we have to demonstrate

$$\vdash\ w = \mathbf{a}\ \&\ \mathbf{A}_{i+1} \leftrightarrow w = \mathbf{a}\ \&\ \mathbf{A}_i.$$

This is done by induction on depth p of the identity $\mathbf{c} = \mathbf{d}$. If the identity occurs at the depth 0 we observe that $\vdash\ w = \mathbf{a} \rightarrow w = \mathbf{b}_0$. Thus under the assumption $w = \mathbf{a}$ we have

$$\mathbf{c} = \mathbf{d} \leftrightarrow w = \mathbf{b}_0\ \&\ \mathbf{c} = \mathbf{d} \leftrightarrow w = \mathbf{e}$$

so we can use the Equality theorem of predicate calculus. If the identity $\mathbf{c} = \mathbf{d}$ occurs within $p+1$ existential quantifiers in the subformula $\exists\mathbf{C}$ of $\mathbf{A}_i$ then we use the theorem (7.5) proven in the next section to obtain

$$\vdash\ w = \mathbf{b}_p\ \&\ \exists\mathbf{C} \leftrightarrow \exists(w = \mathbf{b}_{p+1}\ \&\ \mathbf{C}).$$

It can be readily seen that under the assumption $w = \mathbf{a}$ we are allowed to perform the replacement

$$\mathbf{c} = \mathbf{d} \leftrightarrow w = \mathbf{b}_{p+1}\ \&\ \mathbf{c} = \mathbf{d} \leftrightarrow w = \mathbf{e}.$$

From (1) we have for the *w-convert* $\mathbf{A}_m$ of the formula $\mathbf{A}$

$$\vdash\ w = \mathbf{a}\ \&\ \mathbf{A}_m \rightarrow \mathbf{A}. \tag{2}$$

Let us now turn to predicate definitions. Every predicate P we want to use in R$^+$–Maple has to be brought into the required form as given at the beginning of the section. We show here just the general form (with the equivalence). The special forms (with implications) are converted similarly. We first bring the predicate P into the form

$$\vdash\ P(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n) \leftrightarrow \mathbf{A}$$

for $n>0$ with all variables $\mathbf{x}_i$ distinct and occurring in $\mathbf{A}$. Next we use the above procedure to convert the formula $\mathbf{A}$ into the R$^+$–Maple form. We start with the semi-w-term $\mathbf{a} \equiv [\mathbf{a}', \mathbf{z}]$ where $\mathbf{z}$ is a variable not occurring in $\mathbf{A}$ and $\mathbf{a}' \equiv \mathbf{x}_1$ for $n = 1$ or $\mathbf{a}' \equiv [\mathbf{x}_1, \cdots, \mathbf{x}_n]$ otherwise. From (1) we have

$$\vdash\ w = [\mathbf{a}', \mathbf{z}]\ \&\ P(\mathbf{a}') \leftrightarrow w = [\mathbf{a}', \mathbf{z}]\ \&\ \mathbf{A}_m.$$

For the main environment $\mathbf{b}$ we have

$$\vdash\ w = [\mathbf{a}', \mathbf{z}] \rightarrow \mathbf{b} = [\mathbf{a}', \mathbf{z}]\ \&\ w.\mathbf{h} = \mathbf{a}'\ \&\ w.\mathbf{t} = \mathbf{z}.$$

Using this we obtain

$$\vdash\ w = [w.\mathbf{h},\ w.\mathbf{t}]\ \&\ P(w.\mathbf{h}) \leftrightarrow w = \mathbf{b}\ \&\ \mathbf{A}_m.$$

Finally we observe that $\vdash\ w \neq 0 \leftrightarrow w = [w.\mathbf{h},\ w.\mathbf{t}]$ so we have the predicate P in the desired computational form

$$\vdash\ w \neq 0\ \&\ P(w.\mathbf{h}) \leftrightarrow w = \mathbf{b}\ \&\ \mathbf{A}_m.$$

To start the computation of the formula $\mathbf{A}$ we compute $w = \mathbf{b}\ \&\ \mathbf{A}_m$ by repeatedly applying *rewriting rules* of R$^+$–Maple to it (see section 7 for the rules).

Rewriting rules are always theorems of an extension of TP and have the following forms:

$$\vdash\ \mathbf{B} \leftarrow \mathbf{C} \tag{3}$$
$$\vdash\ \mathbf{B} \rightarrow \mathbf{C} \tag{4}$$
$$\vdash\ \mathbf{B} \leftrightarrow \mathbf{C}. \tag{5}$$

where $\mathbf{B}$ and $\mathbf{C}$ are w-formulas. Suppose that a subformula $\mathbf{B}$ of the w-formula $\mathbf{D}$ is the left-hand-side of a rewriting rule. We say that the rewriting rule is *applicable* to that occurrence of $\mathbf{B}$ if it is of the form (5), or it is of the form (3) and the occurrence of $\mathbf{B}$ is in the positive context, or the rewriting rule is of the form (4) and the occurrence of **B** is in the negative context. A w-formula **E** is obtained from the w-formula **D** by one *computation step* if the *leftmost* occurrence in **D** of a subformula with an applicable rule is replaced by the right-hand-side of the rule yielding **E**. The rewriting rules of R$^+$–Maple are chosen in such a way that at most one rule applies to a given subformula. From the definition of applicability we have

$$\vdash \mathbf{E} \rightarrow \mathbf{D}.$$

A *computation sequence* for a w-formula $\mathbf{D}_0$ is a finite sequence of w-formulas

$$\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \cdots, \mathbf{D}_k$$

such that for each i ($0 \leq i < k$) the formula $\mathbf{D}_{i+1}$ is obtained from the formula $\mathbf{D}_i$ by one computation step, and the formula $\mathbf{D}_k$ is the first formula in the sequence such that $\mathbf{D}_k \equiv w = \mathbf{c}$ for an environment **c** or $\mathbf{D}_k \equiv w = \mathbf{c} \vee \mathbf{D}'$ for a *proper* environment **c** and a w-formula **D**'. If there is no computation sequence for $\mathbf{D}_0$ it is because there is either no rewriting rule applicable to a formula $\mathbf{D}_i$ or the sequence can be arbitrarily extended without ever encountering a formula of the form suitable for $\mathbf{D}_k$. In the first case we say that the computation *blocks*, in the latter case the computation is said to be *non-terminating*.

We always have $\vdash \mathbf{D}_{i+1} \rightarrow \mathbf{D}_i$ for $0 \leq i < k$. Consequently we have $\vdash \mathbf{D}_k \rightarrow \mathbf{D}_0$ and also

$$\vdash w = \mathbf{c} \rightarrow \mathbf{D}_0. \tag{6}$$

If $\mathbf{D}_k \equiv w = \mathbf{c}$ then we say that the computation sequence is *deterministic*. Otherwise the formula $\mathbf{D}_k$ contains the *backtrack* formula **D**'. Any value satisfying the environment equation $w = \mathbf{c}$ satisfies the initial formula $\mathbf{D}_0$. If the computation sequence is deterministic and **c** is empty we say that the computation *failed*. Unless the rewriting rules are complete a failed computation means only that we did not succeed in finding a value w satisfying the formula $\mathbf{D}_0$. In R$^+$–Maple we basically use the negation as failure and a depth-first search so it can very well happen, that R$^+$–Maple, just as Prolog, can miss a solution.

If a computation (both deterministic or with a backtrack formula) terminates with the proper environment **c** then by (6) any solution of the environment equation $w = \mathbf{c}$ satisfies the starting formula. Additional solutions can be obtained in the non-deterministic case by realising that $\vdash \mathbf{D}' \rightarrow \mathbf{D}_0$ and by using the backtrack formula **D**' as the starting formula of a new computation sequence.

Let us now relate the computations to the w-convert $\mathbf{A}_m$ of a formula **A**. We start the computation with $\mathbf{D}_0 \equiv w = \mathbf{b} \mathbin{\&} \mathbf{A}_m$. When we terminate with a proper environment $w = \mathbf{c}$ we have

$$\vdash w = \mathbf{c} \rightarrow w = \mathbf{b} \mathbin{\&} \mathbf{A}_m.$$

It is a relatively straight-forward proof by the induction on the length of the computation sequence (utilizing the rewriting rules as given in the next section) to show that the environment **c** is a refinement of the main environment **b**. Thus every pointer admissible in **b** is admissible in **c**. We use the Satisfiability theorem (5.5) to find a semi-w-term **d** without pointers such that

$$\vdash w = \mathbf{d} \rightarrow w = \mathbf{c}.$$

Since the variables free in **d** do not occur in **c** we can always rename the variables in **d** so they are disjoint with the variables $\mathbf{x}_i$ and **z**. We have

$$\vdash w = \mathbf{d} \rightarrow w = \mathbf{b} \mathbin{\&} \mathbf{A}_m.$$

Observing that **d** is a refinement of **b** and that

$$\vdash w = \mathbf{d} \rightarrow w\alpha_i = \mathbf{d}/\alpha_i \mathbin{\&} w.\mathbf{t} = \mathbf{d}/.\mathbf{t}$$

we replace every pointer $w\alpha$ in **b** by $\mathbf{d}/\alpha$ yielding a new semi-w-term **e** such that

$$\vdash w = \mathbf{d} \rightarrow w = \mathbf{e} \mathbin{\&} \mathbf{A}_m. \tag{7}$$

Next we observe that

$$\mathbf{e} \equiv \mathbf{a}\{\mathbf{x}_1{:=}\mathbf{d}/\alpha_1, \cdots, \mathbf{x}_n{:=}\mathbf{d}/\alpha_n, \mathbf{z}{:=}\mathbf{d}/.\mathbf{t}\}.$$

Performing the above substitution in the formula (2) yields

$$\vdash\ w = \mathbf{e}\ \&\ \mathbf{A}_m \rightarrow \mathbf{A}\{\mathbf{x}_1{:=}\mathbf{d}/\alpha_1,\ \cdots,\mathbf{x}_n{:=}\mathbf{d}/\alpha_n\}. \tag{8}$$

Note that $\mathbf{z}$ does not occur in **A**. Combining (7) and (8) together yields

$$\vdash\ w = \mathbf{d} \rightarrow \mathbf{A}\{\mathbf{x}_1{:=}\mathbf{d}/\alpha_1,\ \cdots,\mathbf{x}_n{:=}\mathbf{d}/\alpha_n\}. \tag{9}$$

Since neither **d** nor **A** contain the variable w we substitute **d** for w in (9) and use the modus ponens to obtain

$$\vdash\ \mathbf{A}\{\mathbf{x}_1{:=}\mathbf{d}/\alpha_1,\ \cdots,\mathbf{x}_n{:=}\mathbf{d}/\alpha_n\}. \tag{10}$$

We can say that we have found an *answer* to the query **A**. The substitution for the variables $\mathbf{x}_i$ is called the *answer substitution* in [6].

7. Rewriting Rules of R$^+$-Maple.

We now present the rewriting rules of R$^+$-Maple. For simplicity's sake we are concerned purely with sequential computations in the style of Prolog. Our computations with negations are basically computations with negation as finite failure.

A formula of TP, after the conversion into an equivalent w-formula, is of the form $w = \mathbf{a}\ \&\ \mathbf{C}$ where **a** is a proper environment and **C** a w-formula. We have seen that the computation agent always selects the leftmost application of a rewriting rule. As a consequence, the environment will traverse the formula in the depth-first left-to-right order. As the computation goes on, the initial environment is modified, extended, and contracted. The rules are chosen in such a way that the resulting w-terms will be always environments. The rewriting rules will, however, distinguish between empty and proper environments.

The first group of rewriting rules will carry the proper environment $w = \mathbf{a}$ down the computed formula. There is one rule for each possible form of the w-formula **C**.

The cases when **C** is a disjunction or conjunction are straightforward and the reader can easily convince himself that they are indeed theorems of TP.

Rewriting Rules for Conjunctions and Disjunctions: If **a** is a proper environment and **A** and **B** w-formulas then

$$\vdash\ w = \mathbf{a}\ \&\ (\mathbf{A}\ \&\ \mathbf{B}) \leftrightarrow (w = \mathbf{a}\ \&\ \mathbf{A})\ \&\ \mathbf{B} \tag{1}$$
$$\vdash\ w = \mathbf{a}\ \&\ (\mathbf{A} \vee \mathbf{B}) \leftrightarrow (w = \mathbf{a}\ \&\ \mathbf{A}) \vee (w = \mathbf{a}\ \&\ \mathbf{B}). \tag{2}$$

When a proper environment reaches a negation it will go inside the negation unchanged. The rule for negation relies on the tautology $\vdash\ \mathbf{B}\ \&\ \neg\ \mathbf{A} \leftrightarrow \neg\ (\mathbf{B}\ \&\ \mathbf{A})\ \&\ \mathbf{B}$.

Rewriting Rules for Negations: If **a** is a proper environment and **A** a w-formula then

$$\vdash\ w = \mathbf{a}\ \&\ \neg\ \mathbf{A} \leftrightarrow \neg\ (w = \mathbf{a}\ \&\ \mathbf{A})\ \&\ w = \mathbf{a}. \tag{3}$$

When the computation reaches an existential quantifier then the environment $w = \mathbf{a}$ will be extended to $w = [\,w.\mathbf{h},\ \mathbf{a}(w.\mathbf{t})]$. The old environment **a** is pushed further down in the stack. This means that every pointer $w\alpha$ in **a** will have to be renamed to $w.\mathbf{t}.\alpha$. This is achieved by the substitution $\mathbf{a}(w.\mathbf{t})$. Although substitutions are expensive to implement on computers, this substitution comes for free when environments are implemented by pointers. Figure 5 illustrates the extension with environment graphs.

Rewriting Rules for Existential Quantifiers: If **a** is a proper environment and **A** a w-formula then $\mathbf{b} \equiv [\,w.\mathbf{h},\ \mathbf{a}(w.\mathbf{t})]$ is a proper environment and

$$\vdash\ w = \mathbf{a}\ \&\ \exists\mathbf{A} \leftrightarrow \exists(w = \mathbf{b}\ \&\ \mathbf{A}). \tag{4}$$

Proof: By the Extension lemma (section (5)) the term **b** is a proper environment. Preparatory to the demonstration of (4) we observe that

$$(w = \mathbf{b})\{w{:=}[\mathbf{x},\ w]\} \leftrightarrow (w = [\,w.\mathbf{h},\ \mathbf{a}(w.\mathbf{t})])\{w{:=}[\mathbf{x},\ w]\} \leftrightarrow$$
$$[\mathbf{x},\ w] = [\mathbf{x},\ \mathbf{a}\{w{:=}w.\mathbf{t}\}\{w{:=}[\mathbf{x},\ w]\}] \leftrightarrow w = \mathbf{a}\{w{:=}w\} \leftrightarrow w = \mathbf{a}.$$

We have

$$w = \mathbf{a} \mathbin{\&} \exists \mathbf{A} \leftrightarrow w = \mathbf{a} \mathbin{\&} \exists \mathbf{x}\, \mathbf{A}\{w{:=}[\mathbf{x}, w]\} \leftrightarrow \exists \mathbf{x}(w = \mathbf{a} \mathbin{\&} \mathbf{A}\{w{:=}[\mathbf{x}, w]\}) \leftrightarrow$$
$$\exists \mathbf{x}((w = \mathbf{b})\{w{:=}[\mathbf{x}, w]\} \mathbin{\&} \mathbf{A}\{w{:=}[\mathbf{x}, w]\}) \leftrightarrow \exists(w = \mathbf{b} \mathbin{\&} \mathbf{A}).$$

This terminates the proof of (4). The last two cases are concerned with atomic formulas.

When the environment $w = \mathbf{a}$ reaches the predicate call $P(\mathbf{b})$ we construct from the pure code for $P(\mathbf{b})$ the w-term $\mathbf{b}(w.\mathbf{t})$. The environment $\mathbf{a}$ is extended by pushing the term $\mathbf{b}(w.\mathbf{t})$ on the top of stack yielding the environment $[\mathbf{b}(w.\mathbf{t}), \mathbf{a}(w.\mathbf{t})]$. The situation is shown in the figure 6.

Rewriting Rules for Predicate Calls: If $\mathbf{a}$ is a proper environment, $\mathbf{b}$ a w-term with all pointers admissible in $\mathbf{a}$ and the predicate P has the definition $\vdash\ w \neq 0 \mathbin{\&} P(w.\mathbf{h}) \leftrightarrow \mathbf{A}$ where $\mathbf{A}$ is a w-formula then the w-term $\mathbf{c} \equiv [\mathbf{b}(w.\mathbf{t}), \mathbf{a}(w.\mathbf{t})]$ is a proper environment and

$$\vdash\ w = \mathbf{a} \mathbin{\&} P(\mathbf{b}) \leftrightarrow \exists(w = \mathbf{c} \mathbin{\&} \mathbf{A}). \tag{5}$$

If the predicate P is defined by $\vdash\ w \neq 0 \mathbin{\&} P(w.\mathbf{h}) \leftarrow \mathbf{A}$ or $\vdash\ w \neq 0 \mathbin{\&} P(w.\mathbf{h}) \rightarrow \mathbf{A}$ then we use as rewriting rules the theorems obtained from (5) by replacing $\leftrightarrow$ by $\leftarrow$ or $\rightarrow$.

Proof: The Extension lemma (section 5) shows that $\mathbf{c}$ is a proper environment. Preparatory to the demonstration of (5) we observe that

$$(w = \mathbf{c})\{w{:=}[\mathbf{x}, w]\} \leftrightarrow (w = [\mathbf{b}(w.\mathbf{t}), \mathbf{a}(w.\mathbf{t})])\{w{:=}[\mathbf{x}, w]\} \leftrightarrow [\mathbf{x}, w] = [\mathbf{b}, \mathbf{a}]$$
$$P(w.\mathbf{h})\{w{:=}[\mathbf{x}, w]\} \leftrightarrow P(\mathbf{x}).$$

We have

$$w = \mathbf{a} \mathbin{\&} P(\mathbf{b}) \leftrightarrow w = \mathbf{a} \mathbin{\&} \exists \mathbf{x}(\mathbf{x} = \mathbf{b} \mathbin{\&} P(\mathbf{x})) \leftrightarrow$$
$$\exists \mathbf{x}(\mathbf{x} = \mathbf{b} \mathbin{\&} w = \mathbf{a} \mathbin{\&} P(\mathbf{x})) \leftrightarrow \exists \mathbf{x}([\mathbf{x}, w] = [\mathbf{b}, \mathbf{a}] \mathbin{\&} P(\mathbf{x})) \leftrightarrow$$
$$\exists \mathbf{x}((w = \mathbf{c})\{w{:=}[\mathbf{x}, w]\} \mathbin{\&} P(w.\mathbf{h})\{w{:=}[\mathbf{x}, w]\}) \leftrightarrow \exists(w = \mathbf{c} \mathbin{\&} P(w.\mathbf{h})) \leftrightarrow$$
$$\exists(w = \mathbf{c} \mathbin{\&} \mathbf{A}).$$

The last step is justified by observing that $\mathbf{c} \neq 0$. The counterparts of (5) for the cases when the predicate P is not defined by an equivalence are proven just like (5).

All identities occurring in a computed formula are environment equations. Thus it is enough to consider the following case.

Rewriting Rules for Identities: If $\mathbf{a}$ is a proper environment and $\mathbf{b}$ is an environment then we can effectively find an environment $\mathbf{c}$ such that

$$\vdash\ w = \mathbf{a} \mathbin{\&} w = \mathbf{b} \leftrightarrow w = \mathbf{c}. \tag{6}$$

Moreover, if $\mathbf{b}$ and $\mathbf{c}$ are proper then every pointer admissible in $\mathbf{a}$ and $\mathbf{b}$ is admissible in $\mathbf{c}$.

Proof: If $\mathbf{b}$ is empty then it suffices to set $\mathbf{c} \equiv [w, 0]$. If $\mathbf{b}$ is proper then by the Extension lemma (section 5) the w-term $[\mathbf{b}(w.\mathbf{h}), \mathbf{a}(w.\mathbf{t})]$ is a proper environment with pointers $w.\mathbf{h}$ and $w.\mathbf{t}$ admissible with 0 jumps. The Environment theorem (section 5) applies and we can apply the environment solution algorithm to solve the system

$$\vdash\ w = [\mathbf{b}(w.\mathbf{h}), \mathbf{a}(w.\mathbf{t})] \mathbin{\&} w.\mathbf{h} = w.\mathbf{t} \leftrightarrow w = \mathbf{d}.$$

Let us call this formula $\mathbf{A}$. By the substitution theorem of the predicate calculus we have $\vdash\ \mathbf{A}\{w{:=}[w, w]\}$ and thus also $\vdash\ \mathbf{A}(w, w)$. This last formula is

$$\vdash\ [w, w] = [\mathbf{b}, \mathbf{a}] \mathbin{\&} w = w \leftrightarrow [w, w] = \mathbf{d}(w, w).$$

After some simplification we have

$$\vdash\ w = \mathbf{a} \mathbin{\&} w = \mathbf{b} \leftrightarrow [w, w] = \mathbf{d}(w, w).$$

If $\mathbf{d}$ is empty then

$$[w, w] = \mathbf{d}(w, w) \equiv [w, w] = [[w, w], 0] \leftrightarrow w = [w, 0].$$

Thus it suffices to take $\mathbf{c}$ empty. Otherwise, $\mathbf{d}$ must have pointers $w.\mathbf{h}$ and $w.\mathbf{t}$ admissible, so it is of the form $\mathbf{d} \equiv [\mathbf{d}_1, \mathbf{d}_2]$ where $\mathbf{d}_1 \equiv w.\mathbf{t}$ and $\mathbf{d}_2 \equiv \mathbf{c}(w.\mathbf{t})$ for a proper environment $\mathbf{c}$. We have

$$[w, w] = \mathbf{d}(w, w) \equiv [w, w] = [w, \mathbf{c}] \leftrightarrow w = \mathbf{c}.$$

We see that $\mathbf{c}$ solves the equation (6). The method for finding the environment $\mathbf{c}$ will have to be implemented in the software or hardware interpreter of R^+–Maple.

We have seen how environments travel down in the computed formula until they reach an environment equation. The new environment **d** does not have to be necessarily proper, it can be also empty. Moreover, it does not have to be the case that the new environment equation $w = \mathbf{d}$ is in the context $w = \mathbf{d} \,\&\, \mathbf{D}$ which would allow further downward computation. Alternatively, the environment equation can be in the context $w = \mathbf{d} \,\&\, \mathbf{D}$, but **d** may be empty, i.e. we have the situation **fail** & **D**. When the computation cannot proceed further down, it starts to ascend in the computed formula. Ascent starts with the empty environment equation **fail** occurring in any context, a proper equation $w = \mathbf{a}$ occurring in the context of negations and existential quantifiers, or a proper equation in the context $w = \mathbf{a} \vee \mathbf{C}$. The formula **C** in the last case is a backtracking formula. The second case, when a proper equation travels backwards by itself is the deterministic case when a single term has been computed.

We shall first give rules for the backward movement of failure **fail** through enclosing connectives and quantifiers. The reader can convince himself that the rewriting rules are theorems of TP.

Rewriting Rules for Failure: If **A** is a w-formula and **a** a proper environment then

$$\vdash \ \mathbf{fail} \,\&\, \mathbf{A} \leftrightarrow \mathbf{fail} \tag{7}$$
$$\vdash \ \mathbf{fail} \vee \mathbf{A} \leftrightarrow \mathbf{A} \tag{8}$$
$$\vdash \ \neg\, \mathbf{fail} \,\&\, w = \mathbf{a} \leftrightarrow w = \mathbf{a} \tag{9}$$
$$\vdash \ \exists\, \mathbf{fail} \leftrightarrow \mathbf{fail} \tag{10}$$

The reader will note that the rule for negation (3) guarantees that there will be a proper environment equation after $\neg$. So the rule (9) is sufficient.

The proper environment equation $w = \mathbf{a}$ will travel backwards in the deterministic computation through enclosing negations and existential quantifiers. As soon as it reaches a conjunction it will start a downward movement in the conjunct. When it reaches a disjunction, it changes into a backtrack computation.

Rewriting Rules for Deterministic Computation: If **a** and $[\mathbf{b}, \mathbf{a}(w.\mathbf{t})]$ are proper environments then

$$\vdash \ \neg\, w = \mathbf{a} \,\&\, w = \mathbf{a} \leftrightarrow \mathbf{fail} \tag{11}$$
$$\vdash \ \exists\, w = [\mathbf{b}, \mathbf{a}(w.\mathbf{t})] \leftrightarrow w = \mathbf{a}. \tag{12}$$

Proof: The theorem (11) is obvious. By the Contraction lemma every proper environment $[\mathbf{b}, \mathbf{a}']$ has the form $[\mathbf{b}.\mathbf{a}(w.\mathbf{t})]$ where **a** is a proper environment. We have

$$\exists\, w = [\mathbf{b}, \mathbf{a}(w.\mathbf{t})] \leftrightarrow \exists \mathbf{x}(w = [\mathbf{b}, \mathbf{a}(w.\mathbf{t})]\{w{:=}[\mathbf{x}, w]\} \leftrightarrow$$
$$\exists \mathbf{x}\, [\mathbf{x}, w] = [\mathbf{b}\{w{:=}[\mathbf{x}, w]\}, \mathbf{a}] \leftrightarrow w = \mathbf{a} \,\&\, \exists \mathbf{x}\, \mathbf{x} = \mathbf{b}\{w{:=}[\mathbf{x}, w]\} \leftrightarrow w = \mathbf{a}.$$

The last step is justified by the Contraction lemma.

The rewriting rules (9) and (11) cater to the case when the formula in the negation either fails or succeeds without changing the environment. We do not supply here any rules for the case when the environment is changed within a negation $\neg\, w = \mathbf{a} \,\&\, w = \mathbf{b}$. This is consistent with the negation as failure as implemented in R-Maple or with a *correctly* implemented negation of some Prolog interpreters such as MU-Prolog.

A proper environment **a** in the context $w = \mathbf{a} \vee \mathbf{C}$ will travel backwards through the enclosing environments carrying around the backtracking formula **C**. The backtracking formula will be amended in the course of its backward travel. There is one case when backtracking is not necessary. This happens in the following computation.

$$w = \mathbf{a} \,\&\, (\mathbf{A} \vee \mathbf{B}) \leftrightarrow (w = \mathbf{a} \,\&\, \mathbf{A}) \vee (w = \mathbf{a} \,\&\, \mathbf{B}) \leftrightarrow \cdots$$
$$w = \mathbf{a} \vee (w = \mathbf{a} \,\&\, \mathbf{B})$$

We see that the computation of the formula **A** in the environment **a** did not change the environment, i.e. **A** is a *test* which succeeded. This situation is captured by the following straight-forward rewriting rule which discards the backtracking formula.

Rewriting Rules for Tests: If $\mathbf{A}$ is a w-formula and $\mathbf{a}$ a proper environment then

$$\vdash\ w = \mathbf{a} \lor (w = \mathbf{a}\ \&\ \mathbf{A}) \leftrightarrow w = \mathbf{a}. \tag{13}$$

The other cases are captured by the following rule.

Rewriting Rules for Backtracking: If $\mathbf{a}$ and $[\mathbf{b}, \mathbf{a}(w.\mathbf{t})]$ are proper environments and $\mathbf{C}$ is a w-formula for which (13) does not apply then

$$\vdash\ (w = \mathbf{a} \lor \mathbf{C})\ \&\ \mathbf{A} \leftrightarrow (w = \mathbf{a}\ \&\ \mathbf{A}) \lor (\mathbf{C}\ \&\ \mathbf{A}) \tag{14}$$
$$\vdash\ (w = \mathbf{a} \lor \mathbf{C}) \lor \mathbf{A} \leftrightarrow w = \mathbf{a} \lor (\mathbf{C} \lor \mathbf{A}) \tag{15}$$
$$\vdash\ \neg\ (w = \mathbf{a} \lor \mathbf{C})\ \&\ w = \mathbf{a} \leftrightarrow \mathbf{fail} \tag{16}$$
$$\vdash\ \exists(w = [\mathbf{b}, \mathbf{a}(w.\mathbf{t})] \lor \mathbf{C}) \leftrightarrow w = \mathbf{a} \lor \exists\mathbf{C}. \tag{17}$$

The theorems (14), (15), and (16) are obvious. (17) relies on the easily proven theorem

$$\vdash\ \exists(\mathbf{A} \lor \mathbf{B}) \leftrightarrow \exists\mathbf{A} \lor \exists\mathbf{B}$$

and on the theorem (12).

Consider the situation when an environment reaches a predicate call in the context of an existential quantifier

$$\exists(w = [\mathbf{b}, \mathbf{a}]\ \&\ P(\mathbf{c})).$$

The standard rewriting rules for predicate calls will extend the environment as follows:

$$\exists\,\exists(w = [\mathbf{c}(w.\mathbf{t}), \mathbf{b}(w.\mathbf{t}), \mathbf{a}(w.\mathbf{t})]\ \&\ \mathbf{A}).$$

Here $\mathbf{A}$ is the body of the predicate P. This situation is called *tail recursion* and when the computation of $\mathbf{A}$ results in a new call to a predicate (probably P itself) we will have to extend the environment again. To prevent such explosion of environments by a nesting of existential quantifiers we can coalesce the terms $\mathbf{b}$ and $\mathbf{c}$ into one. This is called tail recursion optimization.

Rewriting Rules for Tail Recursion: If the predicate P is defined by $\vdash\ w \neq 0\ \&\ P(w.\mathbf{h}) \leftrightarrow \mathbf{A}$ where $\mathbf{A}$ is a w-formula, and if $[\mathbf{b}, \mathbf{a}]$ is a proper environment and $\mathbf{c}$ is a w-term with all pointers admissible in $[\mathbf{b}, \mathbf{a}]$ then

$$\vdash\ \exists(w = [\mathbf{b}, \mathbf{a}]\ \&\ P(\mathbf{c})) \leftrightarrow \exists(w = [\mathbf{c}', \mathbf{a}']\ \&\ \mathbf{A}) \tag{18}$$

where the new proper environment $[\mathbf{c}', \mathbf{a}']$ is obtained by the solution of the system

$$\vdash\ w = [\mathbf{b}(w.\mathbf{h}, w.\mathbf{t}.\mathbf{t}), w.\mathbf{t}.\mathbf{h}, \mathbf{a}(w.\mathbf{t})]\ \&\ \mathbf{c}(w.\mathbf{h}, w.\mathbf{t}.\mathbf{t}) = w.\mathbf{t}.\mathbf{h} \leftrightarrow w = [\mathbf{b}', \mathbf{c}', \mathbf{a}']$$

for some w-term $\mathbf{b}'$. The equivalence $\leftrightarrow$ in (18) should be replaced by implications if the predicate P is defined with implications.

The trick of the rule is to substitute into $\mathbf{c}$ parts of the term $\mathbf{b}$ yielding the argument of the new call $\mathbf{c}'$. However, the substitution must be done carefully since the w-term $\mathbf{b}$ can contain dependencies, i.e. pointers to itself of the form $w.\mathbf{h}\alpha$. These dependencies must be incorporated into $\mathbf{c}'$. We leave the proof of this optimization rule to the reader.

8. Conclusions and Future Work.

We have formulated in this paper a one-variable version of Theory of Pairs. This version can be used by itself as the basis for an automated theorem prover because it simplifies the handling of variables. We have presented a method of collapsing of a system of equations with many variables into a single one-variable equation. Moreover, this equation can be represented efficiently within the computer. The single equation can be used as an environment in computing full predicate logic programs. We have presented and proven sound the rules of computations dealing with the sequential computation and with the negation as failure. The language R^+–Maple has basically the computation power of a Prolog with a good negation (and occurs check) although it has a greater expressive power. Moreover, we do not require the full completion for the computation of negated predicates.

We are presently preparing a paper proving the Environment theorem and dealing also with the solution of inequalities. Environments amended by inequalities will be then incorporated into a set of rewriting rules which will have a stronger form of negation than the presently allowed one. We intend also to present connectives of the **case** type to improve the readability of logic programs by *syntax sugaring*. We also plan to present computation rules for lazy evaluation. This will allow the computation with functions other than data structures within the R$^+$-Maple framework.

We are working on the use of environments within a complete proof system having the classical negation. This system will be equivalent to TP without induction but with $w = 0 \vee \exists x \exists y\, w = [x, y]$ as a new axiom. Such a system is complete in the sense that every formula valid in every (including non-standard) model is provable. On the other hand, it is equivalent to the elementary arithmetic of Abraham Robinson [see for instance 4], and so it is incomplete in the sense of Gödel, i.e. there are unprovable formulas true in the standard interpretation. Obviously, such unprovable formulas cannot be valid in some non-standard models. The environments will take care of the axioms for identity as well as the axioms for pairs. Although this system could be a good basis for an automated theorem prover, we have a good reason to believe that a classicaly complete system cannot be efficiently used in a computational way where we have to execute deep chains of recursive predicate calls. The best we can probably hope for computationally is a TP system with an intuitionistic negation.

The author would like to thank Karl Abrahamson and Jamie Andrews for long series of discussions and suggestions about environments.

[1] Clocksin W., Mellish C., Programming in Prolog, Springer Berlin 1981.

[2] Colmerauer A., Prolog and Infinite Trees; in Logic Programming (eds Clark, Tarnlund), Academic Press 1982.

[3] van Emden, M., Lloyd J., A Logical Reconstruction of Prolog II; 2nd International Logic Programming Conference 1984, Uppsala Sweden.

[4] Kleene S., Introduction to Meta-Mathematics; North Holland 1952.

[5] LLoyd J., Foundations of Logic Programming; Springer Berlin 1984.

[6] LLoyd J., Topor R., Making Prolog More Expressive; Journal of Logic Programming 1984 no. 3.

[7] Naish L., An Introduction to MU-Prolog, Tech. Report Dept. of Comp. Science University of Melbourne, 1982.

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

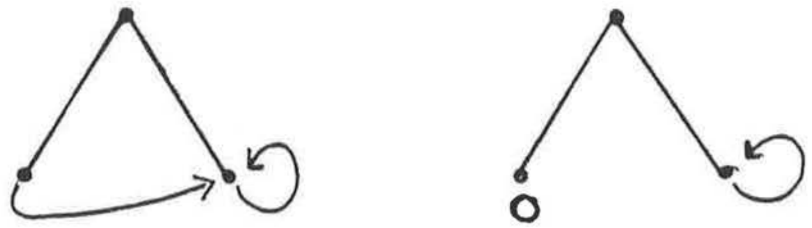
[9] Voda P., Yu B., RF-Maple: A logic Programming Language with Functions, Types and Concurrency, Proceedings of the International Conference on Fifth Generation Computer Systems 1984, Tokyo November 1984.

[10] Voda P., Theory of Pairs, Part I: Provably Recursive Functions; Technical Report of Dept. Comp. Science UBC, Vancouver December 1984.

[11] Voda P., A View of Programming Languages as Symbiosis of Meaning and Computations; New Generation Computing, Tokyo, February 1985.

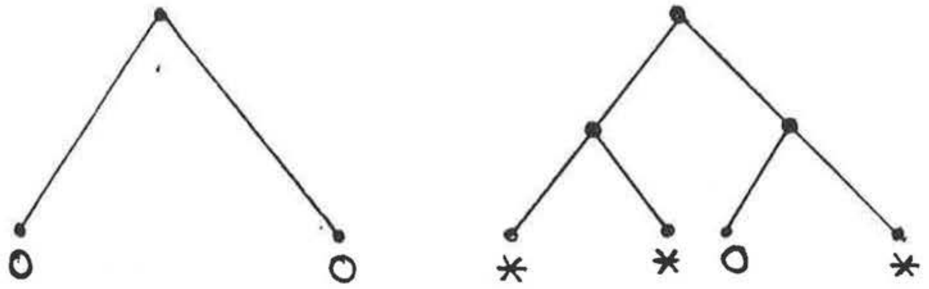
[12] Voda P., Solving Multiple Equations and Inequalities with Environments. Under preparation.

Figure 1



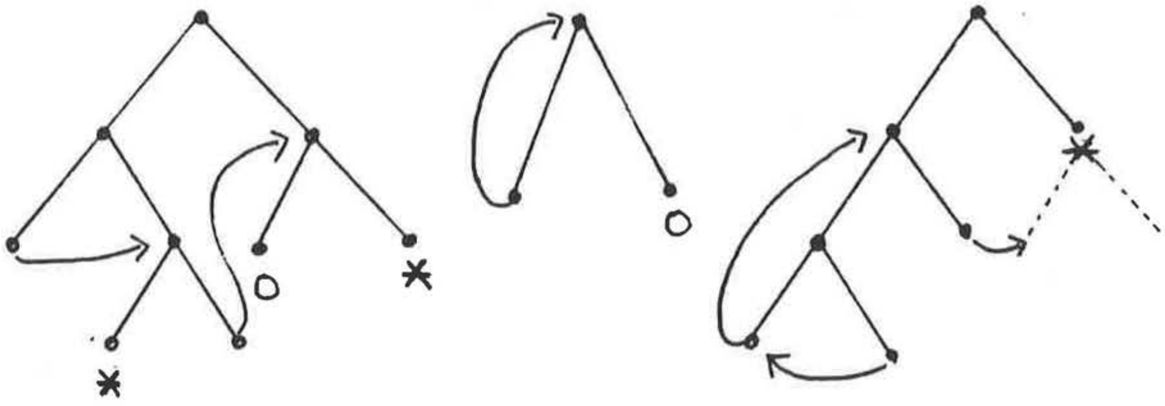
$w = [\,w.\mathbf{t},\ w.\mathbf{t}\,]$ $\qquad$ $w = [0,\ w.\mathbf{t}]$

Figure 2



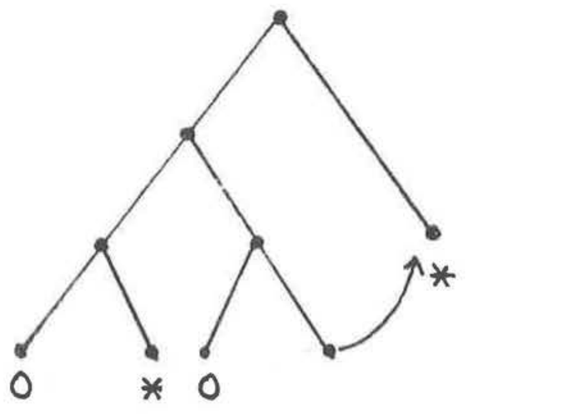
$w = [0,\ 0]$ $\qquad$ $w = [[\,w.\mathbf{h}.\mathbf{h},\ w.\mathbf{h}.\mathbf{t}],\ 0,\ w.\mathbf{t}.\mathbf{t}]$

Figure 3

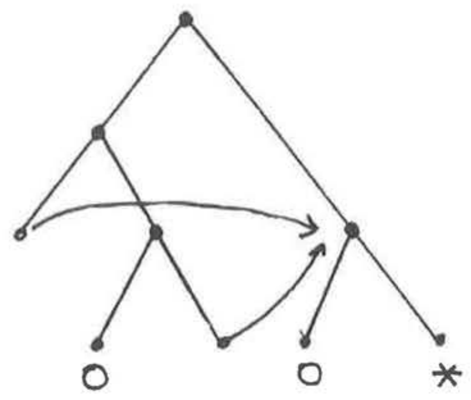


$w = [[\,w.\mathbf{h}.\mathbf{t},\ w.\mathbf{h}.\mathbf{t}.\mathbf{h},\ w.\mathbf{t}],\ 0,\ w.\mathbf{t}.\mathbf{t}]$ $\quad$ $w = [\,w,\ 0]$ $\quad$ $w = [[[\,w.\mathbf{h},\ w.\mathbf{h}.\mathbf{h}.\mathbf{h}],\ w.\mathbf{t}.\mathbf{h}],\ w.\mathbf{t}]$

Figure 4

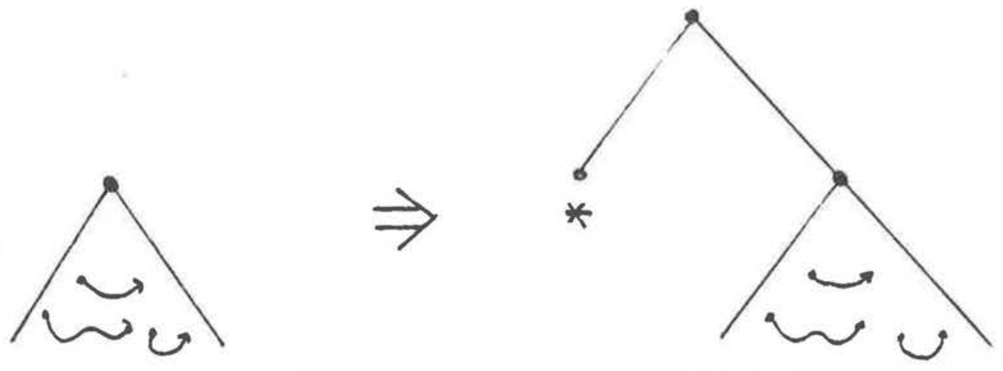


$w = [[[0,\ w.\mathbf{h}.\mathbf{h}.\mathbf{t}],\ 0,\ w.\mathbf{t}],\ w.\mathbf{t}]$



$w = [[w.\mathbf{t},\ 0,\ w.\mathbf{t}],\ 0, w.\mathbf{t}.\mathbf{t}]$

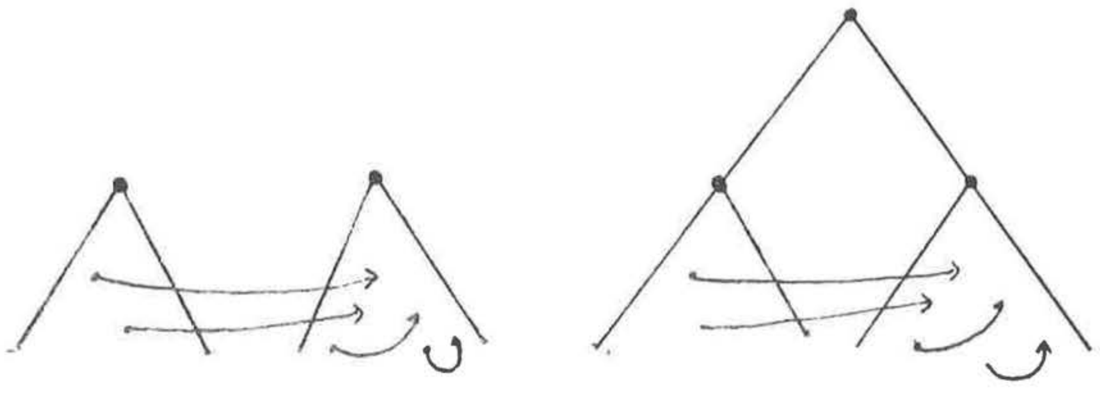
Figure 5



$w = \mathbf{a}$

$w = [w.\mathbf{h},\ \mathbf{a}(w.\mathbf{t})]$

Figure 6



$\mathbf{b}$ $w = \mathbf{a}$

$w = [\mathbf{b}(w.\mathbf{t}),\ \mathbf{a}(w.\mathbf{t})]$