A VIEW OF PROGRAMMING LANGUAGES AS SYMBIOSIS OF MEANING AND CONTROL

by

Paul J. Voda

Technical Report 84-9

May 1984

A View of Programming Languages as Symbiosis of Meaning and Control.

Paul J. Voda

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

May 1984

A View of Programming Languages as Symbiosis of Meaning and Control.

Paul J. Voda

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

1. Introduction.

A computation is a production of a sequence of symbols from another sequence using *rewrite rules*. Turing machines, systems of Post, algorithms of Markov, and sequence of states in a modern electronic computer are examples of the use of rewrite rules. Such generally conceived rewriting rules usually have no obvious connection to functions being computed. Rewriting systems of computations operating on terms in some formal logical framework are on a higher level of organization. Examples are lambda calculus, Herbrand-Gödel systems of equations of recursive functions [7], and computation rules of traditional programming languages like Pascal [6]. A computation uses rewriting rules as formal identities.

Uninterpreted use of rewriting rules is called *operational semantics* of a programming language. The school of *denotational semantics* rightly criticises this approach as a purely formal one not shedding much light on what is exactly being computed.

The basic idea of denotational semantics [11] is to interpret programs into some model as functions. A legal program is given a precise meaning as a function from its inputs to its outputs. Moreover, the operational rewriting rules are shown to be identities preserving meaning. The problem of using models of denotational semantics (almost invariably models of lambda calculus) with practical programs is that it is quite difficult to prove even simple properties of programs. The reason is that the denotational semantics stresses the model-theoretical part and underplays the importance of derivations within a formal theory.

We propose to explain the semantics of a programming language by means of two formal theories. One formal theory gives *denotations* to programs while the other one, which is actually a subtheory of the first theory, specifies the *operations* of the computing machine executing the programs in that the computations correspond exactly to proofs in the weaker theory. We are proposing to use formal theories rather than models because we have in mind a practical use of the theories for computer-assisted verification and transformation of programs. The importance of semi-automatic transformations can be seen from the intensive research in this area [see for instance 1].

One of the difficulties with formal proofs of programs lies with the needless complexity of programming languages with assignments to variables. Fortunately the growing acceptance of *declarative* programming languages demonstrates that side effects are not essential to a good and practical programming language. We shall deal in this paper only with declarative languages by discussing both functional and logic programming languages.

With the decision to concentrate on declarative programming languages we can use a weaker formal theory than any formalized lambda calculus. We shall use a *theory of pairs*, TP, developed by the author for precisely this purpose [15]. TP is a first order theory with the power equivalent to the Peano Arithmetic. It is more appropriate for the needs of programming languages because it is based on an equivalent of S-expressions of Lisp. Pairs permit a natural encoding of data structures and programs alike by using equivalents of Lisp forms rather than somewhat cumbersome Gödel numbers of Peano Arithmetic.

The domain of the intended interpretation of TP consists of the individual 0 and is closed under the operation of pairing. Two pairs are equal if the corresponding components are equal, no pair is equal to 0. The individuals in TP are denoted by *literals*. The term 0 denotes the individual 0. If the terms $\overline{n}$ and $\overline{m}$ denote individuals n and m respectively, then $[\overline{n},\overline{m}]$ denotes the ordered pair composed of n and m. Note that such an use of the term *literal* corresponds to the use of the term *numeral* in Peano Arithmetic. We shall write [a,b,c] as the abbreviation for [a,[b,c]]. A Lisp-like list composed from elements $a_1, a_2, \ldots, a_n$ is written as

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

in TP. Note that the literals of TP correspond to S-expressions of Lisp where the atoms consist only of 0, i.e. of *nil*. We shall use a standard notation of logic [see for instance 10]. Boldfaced letters act as syntactic meta-variables ranging over variables, terms and formulas.

We do not present the axioms of TP here. It is enough to say that they include the *induction axioms* over pairs. The paper on TP [15] proves a very general meta-theorem which permits the introduction into TP by means of conservative extensions functions defined by recursive equations. Such functions can be introduced into TP provided one proves that the arguments of the recursive invocations descend in a well-founded relation.

TP, being a consistent formal theory, has an intended model satisfied in the domain of pairs. The model gives precise meaning to all functions and predicates introduced into TP. We shall define functional programming languages as *subclasses* of terms of TP. Thus a function of a functional programming language can be introduced as a conservative extension of TP by a new function symbol satisfying its (recursive) definition. Similarly, a predicate of a logic programming language, will be introduced into TP by a predicate symbol. The standard interpretation of TP then gives the *denotation* to the function or predicate. TP can be used for formal, possibly computer assisted, proofs of properties of programs.

TP gives *meaning* to declarative programming languages. What about the *control*? How does the operational semantics come in? We shall compute by rewriting rules which will be *invariant* under the meaning of programs being computed. In other words, we shall use only such rewriting rules which can be proven theorems in TP. A computation in a programming language will be understood as a formal proof of certain theorems of TP. Since there will be a one-to-one correspondence between a computation sequence and a formal proof, we cannot expect to use the full deductive power of TP in the computational proofs. We show on five examples of simple programming languages how one can set-up a *subtheory* of TP whose proofs correspond exactly to computations.

The idea that computations are proofs is certainly not a new one. It was used for the first time by Kleene in his formal systems of general recursive (Herbrand-Gödel) equations [7]. Computations in Prolog [2] are by refutations (indirect proofs). Algebraic language OBJ [3] computes by systems of formal equations.

The novelty of our approach comes from the consistent use of two theories. TP (or a similar theory) gives *meaning* to programs, whereas a deductively restricted subtheory S of TP specifies *operational* semantics in that the proofs in S correspond exactly to computations. Thus the questions of program termination, complexity, and absence of deadlocks are reduced to the questions of existence of proofs in S. The questions of operational semantics of programming languages become proof-theoretical questions.

Here lies the difference between our approach and the standard denotational semantics. The meaning and control in denotational semantics are inseparable. The program computing a formula

$$P(6) \vee 5{=}5$$

with sequential disjunction will denote the undefined element if the computation of the predicate $P(6)$ does not terminate. This will happen if the predicate were, for instance, defined by the recursive equation $P(x) \leftrightarrow P(x)$. In our approach the meaning is uncoupled from the control. The above program denotes a true sentence since the formula is a theorem of TP. The

computation will, however, fail to terminate.

Our approach is reminiscent of Hoare's axiomatic method [5] where one proves only *partial* correctness with the need for a separate proof of termination. This separation of concerns has profound practical consequences. By trying first to prove a program only partially correct, we simplify the proof by temporarily ignoring the termination. If the program cannot be proven correct, there is no need to go into the generally more difficult questions of termination. In practice the termination is not as important as the complexity of computation. How does a never terminating program differ from a program which cannot be computed in a million years? By reducing the problems of termination and complexity to the proof theoretical questions of the existence of proofs and of their length we are in better position than denotational semantics which lacks such tools.

There is an apparent paradox in our strict requirement that the recursive functions are properly introduced into TP and our acceptance of non-terminating computations. As pointed above, there is no practical difference between a non-terminating and a terribly inefficient program. On the other side, if one wants to *prove* something at all about a program, one has to make sure that TP is not rendered inconsistent by permitting "definitions" like $P(x) \leftrightarrow \neg P(x)$. Everything is provable in an inconsistent theory.

The subtheory S of TP specifying a programming language will have as terms (formulas) subsets of terms (formulas) of a recursive extension of TP. All axioms of S will be required to be theorems of TP. The rules of inference of S have to be shown to lead from theorems of TP to other theorems. Thus everything which can be proven in S will have to be a theorem of TP. A computation is invariant to the meaning.

The axioms and rules of inference of S will be severely restricted in order to facilitate an efficient implementation of the programming language S on a computer.

For a functional programming language S, the closed term **a** of S is said to be *reducible* if $\vdash_S \mathbf{a}=\overline{n}$ for some literal $\overline{n}$. In the view of $\vdash_{TP} \mathbf{a}=\overline{n}$, the literal $\overline{n}$ is indeed the result of computing the program **a**. For a logic programming language the formula **A** is reducible if $\vdash_S \mathbf{A}$.

In this paper we present five subtheories of TP. Three of these define simple functional programming languages. The programming language F serves just as the introduction to the treatment of control. Programming language N has the applicative order of evaluation, whereas the language L computes by lazy evaluations. The last two languages P and R are logic programming languages. The language P is a Prolog-like programming language whereas R is based on the author's language R-Maple intended to overcome some weak points of Prolog and at the same time give the user the control over the sequential and parallel execution. The discussion of R outlines our view of the formal treatment of concurrent programs.

In order to simplify the discussion the five languages are stripped down to essential parts. A practical programming language will have to add more constructs into the language but the general idea remains unchanged. We hope that the five languages sufficiently cover the different *computational models* used with declarative programming languages.

2. Two Simple Functional Programing Languages.

Let us start with two simple functional programming languages. The language F uses unrestricted reductions of functions. The language N uses the *normal*, or *applicative*, order of evaluations. Arguments are evaluated from left to right before functions are invoked. F is not intended as a basis for a practical programming language, we shall use it as a simple introduction into the techniques of restricting proofs. On the other hand, the language N can be easily extended to a practical programming language.

Since TP is based on pairs we can treat all functions as one-argument only. This greatly simplifies the treatment of recursive functions within TP. We shall write $f(\mathbf{a},\mathbf{b})$ as the abbreviation for $f([\mathbf{a},\mathbf{b}])$.

Let us assume that functions *hd*, *tl*, and if-then-else were introduced into TP with the obvious properties:

$\vdash_{TP}\ hd\ (\mathbf{a},\mathbf{b}) = \mathbf{a}$ (1)
$\vdash_{TP}\ tl\ (\mathbf{a},\mathbf{b}) = \mathbf{b}$ (2)
$\vdash_{TP}\ hd\ (0) = tl\ (0) = 0$ (3)
$\vdash_{TP}\ (\mathbf{if}\ 0\ \mathbf{then}\ \mathbf{c}\ \mathbf{else}\ \mathbf{d}) = \mathbf{c}$ (4)
$\vdash_{TP}\ (\mathbf{if}\ [\mathbf{a},\mathbf{b}]\ \mathbf{then}\ \mathbf{c}\ \mathbf{else}\ \mathbf{d}) = \mathbf{d}$ (5)

Let us abbreviate $hd\ (\mathbf{a})$ to $\mathbf{a}.h$. Similarly $tl\ (\mathbf{a})$ will be abbreviated to $\mathbf{a}.t$.

The programs of F are a subclass of terms, called F-terms, of a recursive extension of TP. F-terms are composed from 0, individual variables, invocations of functions introduced into TP $f\ (\mathbf{a})$, pairing $[\mathbf{a},\mathbf{b}]$, projection functions $\mathbf{a}.h$, $\mathbf{a}.t$ and from the terms of the form **if a then b else c**.

All F-formulas are of the form $\mathbf{a} = \mathbf{b}$ where **a** and **b** are F-terms. For each function f in F we require an identity $\vdash_{TP}\ f\ (x) = \mathbf{b}$ where **b** is a F-term with at most the variable x free.

In the presentation of rules of inference we shall use the notation $\cdots\mathbf{a}\cdots$ to stand for a term (formula) with one distinguished occurrence of the term or formula **a**. A repeated use of dots in the same rule of inference $\cdots\mathbf{b}\cdots$ stands for the same term (formula) but with the single distinguished occurrence of **a** replaced by **b**.

The functional programming languages F has the following axioms and rules of inference.

a) **Initial Axioms** are formulas of the form $\mathbf{a} = \mathbf{a}$ where **a** is a closed F-term.

b) **Function Axioms** are of the form $f\ (\mathbf{a}) = \mathbf{b}$ where **a** and **b** are closed F-terms. We require that we have $\vdash_{TP}\ f\ (\mathbf{a}) = \mathbf{b}$ for each Function Axiom.

c) **Computation Axioms** are the formulas (1), (2), (4), and (5) for any F-term **a**, **b**, **c**, and **d**.

d) **Replacement Rule**.

$$\frac{\mathbf{a} = \cdots\mathbf{b}\cdots,\quad \mathbf{b} = \mathbf{c}}{\mathbf{a} = \cdots\mathbf{c}\cdots}$$

where $\mathbf{b} = \mathbf{c}$ is a Computation or a Function axiom.

It is easy to see that all axioms of F are theorems of TP. We have as a meta-theorem of TP the following one.

If $\vdash_{TP}\ \mathbf{a} = \cdots\mathbf{b}\cdots$ and $\vdash_{TP}\ \mathbf{b} = \mathbf{c}$ then $\vdash_{TP}\ \mathbf{a} = \cdots\mathbf{c}\cdots$.

Thus the Replacement rule does not lead outside of theorems of TP. As a consequence, F is a deductively restricted subtheory of TP. The computation of a closed F-term **a** is an attempt to derive from $\vdash_F\ \mathbf{a}=\mathbf{a}$ the theorem $\vdash_F\ \mathbf{a}=\overline{n}$ for a literal $\overline{n}$.

One can introduce into TP the function *len* to satisfy the recursive equation

$\vdash_{TP}\ len\ (x) = \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ (x.t)]$ (6)

This is possible because the recursion descends in pairs. A possible computation of $len\ (a,0)$ for some constant a is given by the following proof. Each line contains both premises to the Replacement Rule. The first one is either an Initial Axiom or the conclusion of the rule applied to the previous line. The second formula is either a Computation or a Function axiom.

$len\ (a,0) = len\ (a,0)$ $\quad len\ (a,0) = \mathbf{if}\ [a,0]\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ ([a,0].t)]$
$len\ (a,0) = \mathbf{if}\ [a,0]\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ ([a,0].t)]$
$\quad\quad \mathbf{if}\ [a,0]\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ ([a,0].t)] = [0, len\ ([a,0].t)]$
$len\ (a,0) = [0, len\ ([a,0].t)]$ $\quad [a,0].t = 0$
$len\ (a,0) = [0, len\ (0)]$ $\quad len\ (0) = \mathbf{if}\ 0\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ (0.t)]$
$len\ (a,0) = [0, \mathbf{if}\ 0\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ (0.t)]]$ $\quad \mathbf{if}\ 0\ \mathbf{then}\ 0\ \mathbf{else}\ [0, len\ (0.t)] = 0$
$len\ (a,0) = [0,0]$

Note that we did not include the theorems (3) among the Computation axioms of F. This means that a term containing $0.h$ cannot be reduced to a literal simply because there are no rules of inference available. We call such a situation a *deadlock*. A computing machine should abort the

computation in the case of deadlocks. Note that there is nothing wrong with the inclusion of (3) among Computation axioms. Actually, it makes the theory of proofs in F simpler. One can then prove the following theorem.

Theorem: Provided that for each function f of F+(3) the term $f\ (\overline{n}\ )$ is reducible for any literal $\overline{n}$ then any closed F-term is reducible.

The requirement of reducibility of f is a crucial one because we do not make any assumptions about the form of Functional Axioms other than that they are theorems of TP. The theorem (7) is an obvious consequence of (6).

$$\vdash_{TP}\ len\ (x\ ) = len\ (0,x\ ).t \tag{7}$$

Had we decided to use (7) instead of (6) as the Function Axiom for the function *len* we would never be able to reduce invocations of *len* because of infinite recursion.

As a practical programming language F has some serious drawbacks. Some meaningful computations may fail to terminate not only because we admit Function axioms of the form (7). Even with (6) as the Function axiom for *len* the computing machine could stubbornly decide to replace the recursive calls to *len* before reducing if-then-elses. The computing machine can easily tread into a blind alley. In this respect F is *underspecified* as a programming language. There is a great freedom in the order of reductions. Any sensible interpreter of F would have to impose some order on the reductions. This happens in practice all the time. The situation is clearly unacceptable because we are forced to define the operational behaviour of F by its concrete interpreter. The definition of a programming language by its compiler is traditionally considered to be the gravest sin the designer of a language can commit.

On the other hand, the under-specification of F is not as harmful as in those languages with side-effects. The worst thing which can happen, is that an execution may fail to terminate when it could have. If the execution terminates with a literal on the right hand side, then the correct result must have been computed. One can say that the questions concerning meaning of programs are satisfactorily settled by F. The proof-theoretical questions of complexity of computations are not settled at all. There is no way to give an upper bound to the length of computations in F.

The second drawback of F as a practical programming language comes from the need for substitution. For obvious reasons one cannot store in a machine all instances of Function axioms. Function axioms will have to be stored as $f\ (x\ ) = \mathbf{a}$. When the interpreter decides to replace the invocation $f\ (\mathbf{b}\ )$ it will have to perform the substitution $\mathbf{a}\{x:=\mathbf{b}\}$ of the actual argument $\mathbf{b}$ in the body $\mathbf{a}$ of f. The implementation of substitution on the present day computing machinery is very inefficient.

The drawbacks of F are remedied in the language N. We choose the axioms and rules of inference so that the executing machine will have at most one derivation rule available at any given moment. N is a sequential programming language. The impact of concurrent computations will be studied later on the language R in section (5). The control of N is the *applicative* order of execution. Pairs are evaluated from left to right and the argument of a function invocation is reduced before the function body is evaluated. Invocations of functions are not by substitution but by *environments* which are carried around by computations.

We introduce into TP three auxiliary functions called *markers*.

$$do\ (\mathbf{a},\mathbf{b}\ ) = \mathbf{a}\{x:=\mathbf{b}\} \tag{8}$$
$$keep\ (\mathbf{a},y\ ) = do\ (\mathbf{a},y\ ) \tag{9}$$
$$done\ (x\ ) = x \tag{10}$$

Note that (8) and (9) are actually definition schemas defining variable binding operators implicitly binding the distinguished variable x. The variable x is bound in the terms $\mathbf{a}$ of $do\ (\mathbf{a},\mathbf{b}\ )$ and $keep\ (\mathbf{a},\mathbf{b}\ )$. The function *keep* is the same as *do* and the function *done* does not seem to do anything. These functions play an important role in the computations of N. They mark the place where the computating machine attempts a simplification. $do\ (\mathbf{a},\mathbf{b}\ )$ can be viewed as a program counter. The term $\mathbf{a}$ is just about to be reduced, the value for the variable x occurring in $\mathbf{a}$ is $\mathbf{b}$.

The term *done* (**a**) carries the results of computations back.

N-terms are just F-terms with the functions *do*, *keep*, and *done* included. Markers are needed solely for derivations in N. Programmers in N are not permitted to use the markers in their programs. N-formulas are of the form **a** = **b** with **a** and **b** being N-terms.

The axioms and the rules of inference of N are as follows.

a) **Initial Axioms** are the formulas **a** = *do* (**a**,0) for each closed N-term **a** not containing the auxiliary markers.

b) **Function Axioms** are the formulas $f\ (x)$=**c** for each function f and a suitable N-term **c** without markers containing at most the variable x free. We require that we have $\vdash_{TP} f\ (x)$=**c** for each Function axiom.

c) **Computation Axioms** are the following ones for each N-terms without markers **a**, **a'**, **b**, **c**, and **e**.

Pairing Group.

do ([**a**,**b**],**e**) = [*do* (**a**,**e**), *keep* (**b**,**e**)]
[*done* (**a**), *keep* (**b**,**e**)] = [**a**, *do* (**b**,**e**)]
[**a**, *done* (**b**)] = *done* (**a**,**b**)

If Group.

do (**if a then b else c**, **e**) = **if** *do* (**a**,**e**) **then** *keep* (**b**,**e**) **else** *keep* (**c**,**e**)
if *done* (0) **then** *keep* (**b**,**e**) **else** *keep* (**c**,**e**) = *do* (**b**,**e**)
if *done* (**a**,**a'**) **then** *keep* (**b**,**e**) **else** *keep* (**c**,**e**) = *do* (**c**,**e**)

Projection Group.

do (**a**.*h*,**e**) = *do* (**a**,**e**).*h*
done (**a**,**b**).*h* = *done* (**a**)
do (**a**.*t*,**e**) = *do* (**a**,**e**).*t*
done (**a**,**b**).*t* = *done* (**b**)

Turn - around Axioms.

do (0,**e**) = *done* (0)
do (x,**e**) = *done* (**e**)

Argument Axioms. For each function f

do (f (**a**),**e**) = f (*do* (**a**,**e**))

d) **Replacement Rule.**

$$\frac{\mathbf{a} = \cdots \mathbf{b} \cdots, \quad \mathbf{b} = \mathbf{c}}{\mathbf{a} = \cdots \mathbf{c} \cdots}$$

where **b** = **c** is a Computation axiom.

e) **Function Call Rule.**

$$\frac{\mathbf{a} = \cdots f\ (\mathit{done}\ (\mathbf{b})) \cdots, \quad f\ (x) = \mathbf{c}}{\mathbf{a} = \cdots \mathit{do}\ (\mathbf{c},\mathbf{b}) \cdots}$$

where $f\ (x)$ = **c** is a Function Axiom.

f) **Termination Rule.**

$$\frac{\mathbf{a} = \mathit{done}\ (\mathbf{b})}{\mathbf{a} = \mathbf{b}}$$

We leave it to the reader to check that each axiom is a theorem of TP and that an application of a derivation rule leads from theorems of TP to a theorem of TP. Thus N is a subtheory of TP.

A closed N-term **a** not containing markers is reducible if from $\vdash_N$ **a** = *do* (**a**,0) one can derive $\vdash_N$ **a**=$\bar{n}$ for some literal $\bar{n}$. Steps in the execution of **a** in the applicative order correspond exactly to the proof of reducibility in N.

We shall compute the term *len* (*a* ,0) again. This time let *a* be an N-term which reduces to *b*. N-proofs are longer than F-proofs because of the "one-track" property of N-proofs. We shall ommit some obvious lines in the proof. Each line corresponds again to the application of either the Replacement or the Function call rule. Instead of giving the second premise explicitly we only indicate the rule and the group of axioms used.

len (*a* ,0) = *do* (*len* (*a* ,0), 0) Argument axiom and Repl. rule
len (*a* ,0) = *len* (*do* ([*a* ,0], 0)) Pairing axiom and Repl. rule
len (*a* ,0) = *len* (*do* (*a* ,0), *keep* (0,0)) · · ·
· · ·
len (*a* ,0) = *len* (*done* (*b*), *keep* (0,0)) Pairing axiom and Repl. rule
len (*a* ,0) = *len* (*b* ,*do* (0,0)) Turn-around axiom and Repl. rule
len (*a* ,0) = *len* (*b* ,*done* (0)) Pairing axiom and Repl. rule
len (*a* ,0) = *len* (*done* (*b* ,0)) Function axiom and Func. rule
len (*a* ,0) = *do* ((**if** *x* **then** 0 **else** [0, *len* (*x.t*)]), [*b* ,0]) · · ·
· · ·
len (*a* ,0) = [0, *do* (*len* (*x.t*), [*b* ,0]) · · ·
· · ·
len (*a* ,0) = [0, *done* (0)] Pairing axiom and Repl. rule
len (*a* ,0) = *done* (0,0) Termination rule
len (*a* ,0) = [0,0]

One can show by the induction on the proofs in N that whenever *done* (**a**) and *do* (**a**,**e**) occur in an N-proof both terms **a** and **e** are literals. This fact enables an efficient implementation of N using a stack of reduced literals which are elements of pairs and arguments to function invocations.

3. Lazy Evaluation.

Applicative order of evaluation can be wasteful if an argument of a function is not needed in its computation. Applicative order can also waste the storage. The function call *len* (*append* (*a* ,*b*)) evaluated in the applicative order will require, in addition to the storage for the lists *a* and *b*, also the storage for the concatenated list although the last will be replaced by the result.

Lazy evaluation is an order of computation which reduces the body of a function before the arguments. Arguments are reduced only to the depth needed in the body of the function. There is no straight-forward adaptation of N to the language L supporting lazy evaluation. This is because the environment **e** in *do* (**a**,**e**) must be updated once a function call occurring in it is reduced. The updated environment is shared during the subsequent computations rather than being distributed by copying it during the evaluation of pairs and if-then-elses as in N. Actually, an implementation of N will not copy the environments because they will be shared by a pointer. Lazy evaluation, formulated as the subtheory L, requires the sharing of parts of a term to be visible at the symbolic level. Here we need a tree with shared nodes. Such a structure is called a *dag* (directed acyclic graph). Dags can be introduced into TP with the help of the variable binding operator **let** introduced as follows.

(let v:=a in b) = b{v:=a}

The variable **v** is bound in the term **b** but free in **a**. Note that the operator **do** of N is similar but the bound variable is explicit in **let** and implied in *do*.

The term **let** *x* :=*f* (6) **in** [*x* ,7,*x*] which is equal to [*f* (6),7,*f* (6)] permits a single evaluation of the function *f* (6) which is then used twice. **Lets** will be used in groups. We abbreviate **let** *x* :=**a** **in** **let** *y* :=**b** **in** **c** to **let** *x* :=**a**, *y* :=**b** **in** **c**. Similarly, for longer sequences of *bindings* **v**:=**a**. A (possibly empty) sequence of bindings will be abbreviated by a greek letter as in **let** α **in** **a**. For the empty sequence of bindings we set **let** **in** **a** to stand for **a**.

A bound variable in the sequence of bindings α corresponds to the pointer leading to a shared object. Although it is possible to have the same variable bound twice or more times we shall *always* assume that all variables bound in α are different. This can be assured at the symbolic

level by the systematic renaming of bound variables. In practice a different binding uses a different pointer anyway.

A term of L will be simplified in such a way that the bindings will be always kept on the outermost level. We shall use the marker *do* (**a**) to mark the term just being reduced. The marker *wait* (**v**) will be used to transfer the control to the evaluation of a binding when the control reaches a variable.

$$\begin{array}{l}\textbf{let}\ \alpha,\ x := f\ (6),\ \beta\ \textbf{in}\ \cdots\ do\ (x)\cdots \Rightarrow \\ \quad \textbf{let}\ \alpha,\ x := do\ (f\ (6)),\ \beta\ \textbf{in}\ \cdots\ wait\ (x)\cdots \Rightarrow \\ \quad \cdots \\ \quad\quad \textbf{let}\ \alpha,\ x := do\ (8),\ \beta\ \textbf{in}\ \cdots\ wait\ (x)\cdots \Rightarrow \\ \quad\quad\quad \textbf{let}\ \alpha,\ x := 8,\ \beta\ \textbf{in}\ \cdots\ do\ (8)\cdots\end{array}$$

This sequence is typical for lazy evaluation. From now on, all references to the variable x will use the reduced value 8 rather than have to reevaluate the call $f\ (6)$.

Lazy evaluation proceeds by a constant oscillation between the evaluation of bodies and bindings. Although this oscillation adds to the cost of housekeeping in the interpreter one achieves in return the economy of space not possible with applicative order. As soon as the argument of a function is partially reduced the function can proceed

$$len\ (append\ (a,b)) \Rightarrow len\ (a_1, append\ (a_2, b)) \Rightarrow [0,\ len\ (append\ (a_2, b))]$$

Lazy evaluation is *shallow* in the sense that when the control reaches a pair *do* (**a**,**b**) it does not evaluate the pair but it reverses. This is not desirable on the outermost level where the evaluation would stop after one shallow reduction.

$$do\ (append\ ([\mathbf{a},\mathbf{b}],\mathbf{c})) \Rightarrow \cdots \Rightarrow do\ (\mathbf{a}, append\ (\mathbf{b},\mathbf{c}))$$

The markers *down* and *up* are used to force the full evaluation on the top level. All four markers are introduced into TP by trivial definitions to satisfy

$$do\ (x) = wait\ (x) = down(x) = up(x) = x$$

The terms and formulas of L are exactly the same as in F with the addition of markers **let** , *do* , *wait* , *down*, and *up*.

The axioms and inference rules of L are the following ones.

a) **Initial Axioms** are the formulas $\mathbf{a} = down(do\ (\mathbf{a}))$ for each closed L-term **a** not containing markers.

b) **Function Axioms** are the formulas $f\ (x) = \mathbf{a}$ where the L-term **a** is without markers and contains at most the variable x free. We require that $\vdash_{TP} f\ (x) = \mathbf{a}$.

c) **Computation Axioms** are the following formulas for any L-terms without markers **a**, **a'**, **b**, and **c**.

Forcing Group.

$$\begin{array}{l} down(do\ (0)) = up(0) \\ down(do\ (\mathbf{a},\mathbf{b})) = [down(do\ (\mathbf{a})),\mathbf{b}] \\ [up(\mathbf{a}),\mathbf{b}] = [\mathbf{a}, down(do\ (\mathbf{b}))] \\ [\mathbf{a}, up(\mathbf{b})] = up(\mathbf{a},\mathbf{b}) \end{array}$$

If Group.

$$\begin{array}{l} do\ (\textbf{if a then b else c}) = \textbf{if}\ do\ (\mathbf{a})\ \textbf{then b else c} \\ \textbf{if}\ do\ (0)\ \textbf{then b else c} = do\ (\mathbf{b}) \\ \textbf{if}\ do\ (\mathbf{a},\mathbf{a}')\ \textbf{then b else c} = do\ (\mathbf{c}) \end{array}$$

Projection Group.

$$\begin{array}{l} do\ (\mathbf{a}.h) = do\ (\mathbf{a}).h \\ do\ (\mathbf{a}.t) = do\ (\mathbf{a}).t \\ do\ (\mathbf{a},\mathbf{b}).h = do\ (\mathbf{a}) \\ do\ (\mathbf{a},\mathbf{b}).t = do\ (\mathbf{b}) \end{array}$$

d) **Replacement Rule.**

$$\frac{\mathbf{a} = \cdots\mathbf{b}\cdots,\quad \mathbf{b} = \mathbf{c}}{\mathbf{a} = \cdots\mathbf{c}\cdots}$$

where $\mathbf{b} = \mathbf{c}$ is a Computation axiom.

e) **Function Call Rule.**

$$\frac{\mathbf{a} = \textbf{let } \alpha, \beta \textbf{ in } \cdots do\ (f\ (\mathbf{b}))\cdots,\quad f\ (x) = \mathbf{c}}{\mathbf{a} = \textbf{let } \alpha, x{:=}\mathbf{b}, \beta \textbf{ in } \cdots do\ (\mathbf{c})\cdots}$$

where $f\ (x) = \mathbf{c}$ is a Function Axiom and all free variables of **b** have binding occurrences in α. In order to maintain the uniqueness of bindings we may be forced to rename the variable x to a new variable $\overline{x}$ in the binding $\overline{x}{:=}\mathbf{b}$ and in the term $\mathbf{c}\{x{:=}\overline{x}\}$. In order to assure the uniqueness of derivations we stipulate that the sequence of bindings α is the shortest sequence binding all free variables of the term **b**.

f) **Wait Introduction Rule.**

$$\frac{\mathbf{a} = \textbf{let } \alpha, \mathbf{u}{:=}\mathbf{b} \textbf{ in } \cdots do\ (\mathbf{u})\cdots}{\mathbf{a} = \textbf{let } \alpha, \mathbf{u}{:=}do\ (\mathbf{b}) \textbf{ in } \cdots wait\ (\mathbf{u})\cdots}$$

g) **Wait Elimination Rules.**

$$\frac{\mathbf{a} = \textbf{let } \alpha, \mathbf{u}{:=}do\ (0) \textbf{ in } \cdots wait\ (\mathbf{u})\cdots}{\mathbf{a} = \textbf{let } \alpha, \mathbf{u}{:=}0 \textbf{ in } \cdots do\ (0)\cdots}$$

$$\frac{\mathbf{a} = \textbf{let } \alpha, \mathbf{u}{:=}do\ (\mathbf{a},\mathbf{b}) \textbf{ in } \cdots wait\ (\mathbf{u})\cdots}{\mathbf{a} = \textbf{let } \alpha, \mathbf{v}{:=}\mathbf{a}, \mathbf{w}{:=}\mathbf{b}, \mathbf{u}{:=}[\mathbf{v},\mathbf{w}] \textbf{ in } \cdots do\ (\mathbf{v},\mathbf{w})\cdots} \qquad (1)$$

where in (1) the introduced variables **v** and **w** do not occur in the premiss.

h) **Termination Rule.**

$$\frac{\mathbf{a} = \textbf{let } \alpha \textbf{ in } up\,(\mathbf{b})}{\mathbf{a} = \mathbf{b}}$$

where the L-term **b** does not contain free variables.

We leave it to the reader to convince himself that the axioms and the rules of inference do not lead outside of the theorems of TP. It can be proven by the induction on the proofs of L that the term $up\,(\mathbf{a})$ occurring in a proof is always a literal.

A note on so called lazy lists which are currently very fashionable. Lazy evaluation is often recommended because the argument of a function call can be an infinite, or lazy list, given by a function term. An infinite list is used-up only partially. We do not have infinite lists in TP. Thus it is impossible to create a lazy list of, say, all numbers by writing $f\ (0)$ where $f\ (x) = [x, f\ (x+1)]$. Such a function can be introduced into TP only at the cost of inconsistency. Lazy lists are one of those features of programming languages which are easy to use operationally and hard to explain semantically.

As mentioned before, it is possible to work with a stronger theory which admits infinite lists, but in the process we will probably lose the basic tool, i.e. the induction over pairs, with which one proves properties of functions. It is the author's contention that lazy lists are quite esoteric objects and they can be justified only when one *insists* on working with a stronger theory than TP. The possible loss of induction seems to be a little too high a price to pay. All problems which can be formulated with lazy lists can be programmed with infinite generators in a logic programming environment. In the above case one can easily introduce a predicate $Nat\ (x)$ which generates natural numbers in a sequence as it is backtracked into. Infinite generators are readily definable in TP (see section 5).

Another argument in favor of infinite lists goes as follows. Parallel programs synchronize on lists of messages. An operating system is a function which goes into endless recursion, ergo one needs infinite lists for endless communications. Our answer to this argument is that very rarely, if at all, an operating system executes longer than, say, 10 years. If the operating system does not crash by then, the machine will certainly become obsolete. Infinite lists are nice to have when

uncoditionally needed, but an operating system can be also explained as a function recurring on a list of commands issued by the operator. There is nothing wrong with interpreting the empty list as a command shutting down the system by terminating the recursion. One can certainly define in TP a list of operator commands which will not be used up in million years.

4. Logic Programming

Logic programming [see for instance 8], as originally conceived, strives for a programming language based on a first order theory such as TP without any special controlling mechanism. In our view the control is given by a subtheory S. An ideal logic programming then requires that S is the same as TP. Computations of logic programs are performed by the full deductive apparatus of a first order theory.

It is obvious that such conceived logic programming must remain an unattainable ideal because an unrestricted theorem prover can quickly enter a blind alley. A full theorem prover with backtracking will be intolerably inefficient. As we have seen with the functional programming languages, the idea of control lies in severely restricting the subtheory S so the proofs can be performed in an efficient way.

Prolog is an example of a language computing predicates which severely restricts the computations. We present a language P with Prolog-like computations. The basic theory for the meaning of P remains TP. We shall reduce predicates instead of terms. Similarly as with functions, all predicates can be defined with one argument only. We shall write $P\ (\mathbf{a},\mathbf{b})$ for $P\ ([\mathbf{a},\mathbf{b}])$.

Let us define the class of P-formulas as a subset of formulas of TP. For that we need to define the class of P-terms. P-terms are composed from variables, introduced constants and functions by pairing. Prolog uses constants and functions as data structures only. Functions are never computed. This means that any function $f\ (x)$ can be introduced by a defining axiom.

$$f\ (x)=[c_f, x]$$

where c_f is a constant unique for the function f.

Atomic P-formulas are of the form $P\ (\mathbf{a})$ where P is an introduced predicate symbol and $\mathbf{a}$ is a P-term. Conjunctive P-formulas are conjunctions of atomic P-formulas. If $\mathbf{A}$ and $\mathbf{B}$ are conjunctive P-formulas then $\mathbf{A} \rightarrow \mathbf{B}$ is a P-formula.

A *clause* is a P-Formula of the form $\mathbf{A} \rightarrow P\ (\mathbf{a})$ or $P\ (\mathbf{a})$ where $\mathbf{A}$ is a conjunctive P-Formula and $\mathbf{a}$ is a P-term.

We shall abbreviate the simultaneous substitution in a term or formula $\mathbf{A}$

$$\mathbf{A}\{\mathbf{v}_1:=\mathbf{a}_1, \mathbf{v}_2:=\mathbf{a}_2, \ldots, \mathbf{v}_n:=\mathbf{a}_n\}$$

by $\mathbf{A}\ \alpha$. A substitution α is said to *unify* the P-terms $\mathbf{a}$ and $\mathbf{b}$ when $\mathbf{a}\ \alpha \equiv \mathbf{b}\ \alpha$. It is easy to see that if there is a substitution α unifying two P-terms $\mathbf{a}$ and $\mathbf{b}$ not sharing common variables, then there is a minimal unifying substitution β such that $\mathbf{a}\ \beta \equiv \mathbf{b}\ \beta$. At the same time there is a substitution γ such that $\mathbf{a}\ \beta\ \gamma \equiv \mathbf{a}\ \alpha$.

Note that it is decidable whether a substitution is a minimal unifying substitution. This permits the formulation of P as a subtheory of TP because we shall be always able to recognize an axiom or a rule of inference.

The axioms and rules of inference of P are the following ones.

a) **Initial Axiom** is a P-formula $\mathbf{A} \rightarrow \mathbf{A}$ for a conjunctive P-formula $\mathbf{A}$.

b) **Computation Axioms** are clauses of the form $\mathbf{A} \rightarrow P\ (\mathbf{a})$ or $P\ (\mathbf{a})$. All Computation axioms must be theorems of TP.

c) **Structural Inference Rule**.

$$\frac{(\mathbf{A}\ \&\ \mathbf{B})\ \&\ \mathbf{C} \rightarrow \mathbf{D}}{\mathbf{A}\ \&\ (\mathbf{B}\ \&\ \mathbf{C}) \rightarrow \mathbf{D}}$$

d) **Clause Rules**.

$$\frac{P\,(\mathbf{a})\ \&\ \mathbf{A}\rightarrow\mathbf{B},\quad \mathbf{C}\rightarrow P\,(\mathbf{b})}{(\mathbf{C}\ \&\ \mathbf{A}\rightarrow\mathbf{B})\,\alpha}\qquad\qquad \frac{P\,(\mathbf{a})\ \&\ \mathbf{A}\rightarrow\mathbf{B},\quad P\,(\mathbf{b})}{(\mathbf{A}\rightarrow\mathbf{B})\,\alpha}$$

where $\mathbf{C}\rightarrow P\,(\mathbf{b})$ or $P\,(\mathbf{b})$ are Computation axioms and α is the minimal unifying substitution such that $\mathbf{a}\,\alpha\equiv\mathbf{b}\,\alpha$.

e) **Termination Rule.**

$$\frac{P\,(\mathbf{a})\rightarrow\mathbf{B},\quad P\,(\mathbf{b})}{\mathbf{B}\,\alpha}$$

where $P\,(\mathbf{b})$ is a Computation axiom and α is the minimal unifying substitution such that $\mathbf{a}\,\alpha\equiv\mathbf{b}\,\alpha$.

A conjunctive P-formula **A** is said to be *reducible* if one can deduce from $\vdash_P\ \mathbf{A}\rightarrow\mathbf{A}$ the formula $\vdash_P\ \mathbf{A}\,\alpha$ for some substitution α. Using the fact that substitutions can be composed, it is not difficult to prove by the induction on the proofs of P that when a computation of **A** stops by an application of the Termination rule the substitution α has been effectively found.

It is easy to verify that all axioms are theorems of TP and that the rules of inference lead from theorems to theorems. The programming language P is only Prolog-like because its interpreter is underspecified. It computes by replacing atomic P-formulas in the antecedent from left to right. In that it agrees with Prolog computations. There are, however, no restrictions on the order in which Computation axioms are applied. Without the backtracking a computation may lead to a blind alley. It is not very difficult to remedy this by admitting as Computation axioms disjunctions of clauses

$$\{\mathbf{A}_1\rightarrow P\,(\mathbf{a}_1)\}\vee\{\mathbf{A}_2\rightarrow P\,(\mathbf{a}_2)\}\vee\ \cdots\{\mathbf{A}_n\rightarrow P\,(\mathbf{a}_n)\}$$

We would have one disjunctive Computation axiom for each predicate symbol P. The derivation rules would have to be modified in such a way that the alternative clauses would be tried from left to right. We shall not do it with P because such an ordering of alternatives plays a crucial role in the computations of the logic programming language R-Maple. R-Maple generalizes the class of logic computations. It will be explained in the next section.

As an example of a P-computation let us introduce the predicates T and M by the following defining axioms.

$$\vdash_{TP}\ T\,(x)\leftrightarrow x=b \tag{1}$$
$$\vdash_{TP}\ M\,(x,y)\leftrightarrow x\neq 0\ \&\ (x.h=y\ \vee\ M\,(x.t,y)) \tag{2}$$

Here b is a constant. A definition of a two place predicate $P\,(x,y)\leftrightarrow\ \cdots x\cdots y\cdots$ is just an abbreviation for

$$P\,(w)\leftrightarrow w\neq 0\ \&\ (\cdots w.h\cdots w.t\cdots)$$

The recursive definition of the list membership predicate M is admissible because the recursion descends in pairs. We have the obvious theorems

$$\vdash_{TP}\ T\,(b) \tag{3}$$
$$\vdash_{TP}\ M\,([x,y],x) \tag{4}$$
$$\vdash_{TP}\ M\,(y,z)\rightarrow M\,([x,y],z) \tag{5}$$

These theorems will be used as P-clauses in the following P-computation where we attempt to find a value w satisfying the formula $M\,([a,b,c,0],w)\ \&\ T\,(w)$ with a and c being some P-terms. We start with the Initial axiom.

$$M\,([a,b,c,0],w)\ \&\ T\,(w)\rightarrow M\,([a,b,c,0],w)\ \&\ T\,(w)$$

We use the substitution $\alpha\equiv\{x:=a,y:=[b,c,0],z:=w\}$ and the P-clause (5) to obtain

$$M\,([b,c,0],w)\ \&\ T\,(w)\rightarrow M\,([a,b,c,0],w)\ \&\ T\,(w)$$

Next we use the substitution $\alpha\equiv\{x:=b,y:=[c,0],w:=b\}$ and the P-clause (4) to obtain

$$T\,(b)\rightarrow M\,([a,b,c,0],b)\ \&\ T\,(b)$$

The use of the clause (3) in the Termination rule with the identical substitution $\alpha\equiv\{\}$ leads to

$M([a,b,c,0],b) \;\&\; T(b)$.

5. Logic Programming Language R.

R-Maple permits logical connectives, quantifiers, and if-then-else constructs. It also permits sequential and parallel computations. As a result we have a purely declarative programming language with no cuts. R-Maple pushes the computational subtheory R as close as possible to TP.

We shall find it advantageous to rephrase TP as a *term* logic. A term logic [see for instance 4] does not make a distinction between a term and a formula. Logical connectives and quantifieres are just functions. We shall need term logic because R will have parallel connectives of conjunction and disjunction which must be first introduced. Logical connectives are given in a traditional logic and there is no provision for defining new ones. The second advantage of term logic is that there is no need to distinguish between a predicate and its characteristic function.

In term logic we can write

if $a=6$ **then** b **else** c

instead of

if $eq\,(a,6)$ **then** b **else** c

where eq is the characteristic function of identity $=$.

$$eq\,(w)=z \leftrightarrow w=0 \;\&\; z=1 \vee \exists x,y\,\{w=[x,y] \;\&\; (x=y \;\&\; z=0 \vee x\neq y \;\&\; z=1)\}$$

We do not give here the axiom system for TP rephrased as a term logic. The basic constructs are just informally explained here. A true formula is a term **a** equal to 0. Thus 0 stands for truth. A non-zero term $[\mathbf{a},\mathbf{b}]$ stands for falsehood. In practice we shall use $1 \equiv [0,0]$ to denote falsehood. Logical connectives of conjunction, disjunction, and negations are just functions satisfying the obvious identities.

$0 \vee 0 = [\mathbf{a},\mathbf{b}] \vee 0 = 0 \vee [\mathbf{a},\mathbf{b}] = 0$
$[\mathbf{a},\mathbf{b}] \vee [\mathbf{c},\mathbf{d}] = 1$
$0 \;\&\; 0 = 0$
$[\mathbf{a},\mathbf{b}] \;\&\; 0 = 0 \;\&\; [\mathbf{a},\mathbf{b}] = [\mathbf{a},\mathbf{b}] \;\&\; [\mathbf{c},\mathbf{d}] = 1$
$\neg\, 0 = 1$
$\neg\, [\mathbf{a},\mathbf{b}] = 0$

The identity relation $\mathbf{a}=\mathbf{b}$ is a function yielding 0 when **a** and **b** are identical and 1 otherwise. For any term **a** the term $\exists x\,\mathbf{a}$ denotes a function of all free variables in **a** but x. We have $\exists x\,\mathbf{a} = 0$ if there is a literal $\bar{n}$ such that $\mathbf{a}\{x:=\bar{n}\} = 0$ and $\exists x\,\mathbf{a}=1$ otherwise. We call a term **a** a *predicate* if it corresponds to a formula of the classically formulated TP. Predicates are equal to either 0 or 1.

We shall now proceed to formulate the subtheory R of TP. For that we define the following functions.

$do\,(x) = x$
$done\,(x) = x$
$x \;||\; y = x \;\&\; y$
$x \;\mathbf{orp}\; y = x \vee y$
$x:=y \;=\; x=y$
$(\mathbf{if}\; x \;\mathbf{then}\; y \;\mathbf{else}\; (\mathbf{v},\mathbf{w})\,\mathbf{a})=z \;=\; x=0 \;\&\; z=y \vee \exists \mathbf{v},\mathbf{w}\,(x=[\mathbf{v},\mathbf{w}] \;\&\; z=\mathbf{a})$

Functions *do* and *done* are used in a similar way as in N. Next two functions are connectives of *parallel* conjunction and disjunction. Function $:=$ is called an *assignment*. We trust that the reader realizes by now that these functions will play only a control role in the deductions of R, semantically they are superfluous.

The function if-then-else is defined as a variable binding operator binding the variables **v** and **w** in the term **a**. Its reduction properties are as follows.

$\vdash_{TP}$ **if** 0 **then** x **else** $(\mathbf{v},\mathbf{w})\,\mathbf{a} \;=\; x$

$\vdash_{TP}$ **if** [**b**,**c**] **then** x **else** (**v**,**w**) **a** = **a**{**v**:=**b**, **w**:=**c**}

If-then-else as a variable binding operator disposes with the need of projection functions *hd* and *tl*.

R-terms are composed from 0 and variables by pairing. Atomic R-predicates are obtained from R-terms by the identity =, assignment :=, and invocations of predicates P (**a**). R-predicates are composed of atomic predicates by conjunctions (both & and | |), disjunctions (both $\vee$ and **orp**), negation, existential quantifiers, markers *do* and *done*, and if-then-else. All predicates $P(x)$ must be introduced into the theory TP as one place predicates in such a way that $\vdash_{TP} P(x)$=**a**. As usual, the markers *do* and *done* may not be used in R-programs, they are introduced during computations.

The markers *do* and *done* are called *processes*. They mark the positions where the computation takes place. The markers can be replaced in any order, possibly in parallel. The marker *do* goes down in a R-term in the usual way *do* (**A** $\vee$ **B**) => *do* (**A**) $\vee$ **B**. The backward marker *done* brings the computed value back. The computed values are either truth values or assignments. The computation with truth values is straight-forward *done* (0) $\vee$ **B** => *done* (0) or *done* (1) $\vee$ **B** => *do* (**B**). Parallel connectives *fork* the forward marker *do* creating two processes *do* (**A** | | **B**) => *do* (**A**) | | *do* (**B**).

The interesting part of R is the backward shipment of assignments *do* (x:=**a**) => *done* (x:=**a**). The general idea is to move this assignment back through enclosing disjunctions and conjunctions by employing the associativity and distributivity of the connectives until the assignment reaches its existential quantifier.

{(*done* (x:=**a**) & **A**) $\vee$ **B**} $\vee$ **C** => {*done* (x:=**a**) & **A**} $\vee$ (**B** $\vee$ **C**)
{(*done* (x:=**a**) & **A**) $\vee$ **B**} & **C** => {*done* (x:=**a**) & (**A** & **C**)} $\vee$ (**B** & **C**)

When the assignment reaches its quantifier, the quantifier is split for a possible backtrack and discharged in one side of disjunction.

$\exists x$ {(*done* (x:=**a**) & **A**) $\vee$ **B**} =>
$\exists x$ (*done* (x:=**a**) & **A**) $\vee$ $\exists x$ **B** =>
do (**A**{x:=**a**}) $\vee$ $\exists x$ **B**

The assigned value **a** is used in the formula **A** and the formula $\exists x$ **B** is kept as a backtrack in the case the first formula fails. Synchronization of parallel processes is achieved by the use of a shared variable, say x, in two concurrent processes. One process assigns a value to the variable x thus awaking the other process which probably will be blocked on the predicate *do* (**if** x **then A else** (y,z) **B**) for which there is no Computation axiom.

R does not provide any rules for moving an assignment back through a negation. Such a program is considered to be incorrect and the attempted execution deadlocks. The full rules for the moving of assignments are slightly more complicated because we have to deal with all combinations of sequential and parallel connectives. An assignment can be moved back through a quantifier on a different variable. Indeed this movement of assignments through quantifiers permits the powerful technique of cooperation of concurrent processes by partially instantiated streams as employed for the first time in Concurrent Prolog [9].

As an example of an introduced predicate we present the predicate *Append* concatenating two lists. We have

$\vdash_{TP}$ *Append* ([f,s],r) =
if f **then** r:=s **else** (h,t) $\exists r'$ {r:=[h,r'] & *Append* ([t,s],r')} (1)

It does not harm to repeat that the above theorem is not a definition of the predicate *Append*. Function *Append* can be introduced into TP to satisfy the theorem (1) only because the recursion descends in the first list. In the case of more complicated recursions it is the programmer's responsibility to make sure that a recursive "definition" is derivable in TP. As an example of what we mean by this consider the predicate, or as it is called in R-Maple, a generator *Gennum* (y,z) generating all numbers z not less than y. We assume that the natural numbers have been

introduced into TP.

$$\vdash_{TP} \ Gennum\ (y,z) \ = \ z{:=}y \ \vee \ \exists x\ (Add\ (y,1,x) \ \&\ Gennum\ (x,z)) \qquad (2)$$

Gennum can be used in R in this form. Since the recursion will never terminate, (2) cannot serve as the defining equation. The predicate can be introduced by an explicit definition

$$Gennum\ (y,z) \ = \ y \leq z$$

from which (2) is provable. Incidentally, the predicate *Gennum* is the counterpart of infinite, or lazy lists (see the discussion in section (3)). TP cannot handle infinite lists but there is no problem with infinite predicates. The typical use of *Gennum* is as follows.

$$\exists w\ \{Gennum\ (12,w) \ \&\ P\ (w)\}$$

The generator will be repeatedly entered to generate the numbers 12, 13,... until the predicate $P\ (w)$ is satisfied.

In order to simplify the presentation of R we compute invocations of predicates by substitutions rather than by environments. We abbreviate a sequence of existential quantifiers into the form $\exists\alpha\mathbf{A}$ where α is a sequence of variables. We do not exclude the empty sequence by setting $\exists\mathbf{A}$ to stand for **A**. We shall write **and** to stand for either sequential or parallel conjunction, similarly **or**.

The axioms and inference rules of R are as follows.

a) **Initial Axioms** are of the form $\mathbf{A} = do\ (\mathbf{A})$ for every R-term **A** not containing markers.

b) **Predicate Axioms** are of the form $P\ (x) = \mathbf{A}$ where **A** is a R-term not containg markers and having at most the variable x free. We require that $\vdash_{TP} P\ (x) = \mathbf{A}$.

c) **Computation Axioms** are the following ones for each R-terms **A**, **B**, **C**, **a**, **b**, **c**, and **d**. The variable **v** does not occur in the term **a**.

Turn-around Group.

$$do\ (0) \ = \ done\ (0)$$
$$do\ (1) \ = \ done\ (1)$$
$$do\ (\mathbf{v}{:=}\mathbf{a}) \ = \ (done\ (\mathbf{v}{:=}\mathbf{a}) \ \&\ 0) \vee 1 \qquad (3)$$

In (3) the variable **v** does not occur in **a**.

Identity Group.

$$do\ (0{=}[\mathbf{a},\mathbf{b}]) \ = \ done\ (1)$$
$$do\ ([\mathbf{a},\mathbf{b}]{=}0) \ = \ done\ (1)$$
$$do\ (0{=}0) \ = \ done\ (0)$$
$$do\ ([\mathbf{a},\mathbf{b}]{=}[\mathbf{c},\mathbf{d}]) \ = \ do\ (\mathbf{a}{=}\mathbf{c}) \ \&\ \mathbf{b}{=}\mathbf{d}$$

Negation Group.

$$do\ (\neg\, \mathbf{A}) \ = \ \neg\, do\ (\mathbf{A})$$
$$\neg\, done\ (0) \ = \ done\ (1)$$
$$\neg\, done\ (1) \ = \ done\ (0)$$

Disjunction Group.

$$do\ (\mathbf{A} \vee \mathbf{B}) \ = \ do\ (\mathbf{A}) \vee \mathbf{B}$$
$$do\ (\mathbf{A}\ \mathbf{orp}\ \mathbf{B}) \ = \ do\ (\mathbf{A})\ \mathbf{orp}\ do\ (\mathbf{B})$$
$$done\ (0)\ \mathbf{or}\ \mathbf{B} \ = \ done\ (0)$$
$$done\ (1) \vee \mathbf{B} \ = \ do\ (\mathbf{B})$$
$$done\ (1)\ \mathbf{orp}\ \mathbf{B} \ = \ \mathbf{B}$$
$$\{\exists\alpha\ (done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A})\ \mathbf{or}\ \mathbf{B}\}\ \mathbf{or}_1\ \mathbf{C} \ =$$
$$\exists\alpha\{done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A}\}\ \overline{\mathbf{or}}\ (\overline{\mathbf{B}}\ \mathbf{or}_1\ \mathbf{C}) \qquad (4)$$
$$\mathbf{A}\ \mathbf{orp}\ \mathbf{B} \ = \ \mathbf{B}\ \mathbf{orp}\ \mathbf{A} \qquad (5)$$

In (4) $\overline{\mathbf{or}} \equiv \mathbf{or}$ and $\overline{\mathbf{B}} \equiv \mathbf{B}$ unless $\mathbf{or}_1 \equiv \mathbf{orp}$. In the latter case $\overline{\mathbf{or}} \equiv \mathbf{orp}$ and if $\mathbf{or} \equiv \vee$ then $\overline{\mathbf{B}} \equiv do\ (\mathbf{B})$.

In (5) **B** has one of the forms $done\ (0)$, $done\ (1)$, or $\exists\alpha\{done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{C}\}\ \mathbf{or}\ \mathbf{D}$ and **A**

is not of any of these forms.

Conjunction Group.

$do\ (\mathbf{A}\ \&\ \mathbf{B}) = do\ (\mathbf{A})\ \&\ \mathbf{B}$
$do\ (\mathbf{A}\ ||\ \mathbf{B}) = do\ (\mathbf{A})\ \&\ do\ (\mathbf{B})$
$done\ (1)\ \mathbf{and}\ \mathbf{B} = done\ (1)$
$done\ (0)\ \&\ \mathbf{B} = do\ (\mathbf{B})$
$done\ (0)\ ||\ \mathbf{B} = \mathbf{B}$
$\{\exists\alpha\,(done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A})\ \mathbf{or}\ \mathbf{B}\}\ \mathbf{and}_1\ \mathbf{C} =$
$\quad \exists\bar{\alpha}\{done\ (\bar{\mathbf{v}}{:=}\bar{\mathbf{a}})\ \overline{\mathbf{and}}\ (\bar{\mathbf{A}}\ \mathbf{and}_1\ \mathbf{C})\}\ \overline{\mathbf{or}}\ (\bar{\mathbf{B}}\ \mathbf{and}_1\ \mathbf{C})$ (6)
$\mathbf{A}\ ||\ \mathbf{B} = \mathbf{B}\ ||\ \mathbf{A}$ (7)

where in (6) some of the bound variables in α must be renamed yielding $\bar{\alpha}$, $\bar{\mathbf{v}}$, $\bar{\mathbf{a}}$, and $\bar{\mathbf{A}}$ should a free variable of **C** become bound inside $\exists\alpha$. Also $\overline{\mathbf{and}} \equiv \mathbf{and}$, $\overline{\mathbf{or}} \equiv \mathbf{or}$, $\bar{\mathbf{A}} \equiv \mathbf{A}$ and $\bar{\mathbf{B}} \equiv \mathbf{B}$ unless $\mathbf{and}_1 \equiv\ ||$. In the latter case $\overline{\mathbf{and}} \equiv\ ||$ and $\overline{\mathbf{or}} \equiv \mathbf{orp}$. If also $\mathbf{and} \equiv\ \&$ then $\bar{\mathbf{A}} \equiv do\ (\mathbf{A})$. Similarly, if $\mathbf{or} \equiv \vee$ then $\bar{\mathbf{B}} \equiv do\ (\mathbf{B})$.
In (7) **B** has one of the forms $done\ (0)$, $done\ (1)$, or $\exists\alpha\{done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{C}\}\ \mathbf{or}\ \mathbf{D}$ and **A** is not of any of these forms.

Quantifier Group.

$do\ (\exists\mathbf{v}\mathbf{A}) = \exists\mathbf{v}\ do\ (\mathbf{A})$
$\exists\mathbf{v}\ done\ (0) = done\ (0)$
$\exists\mathbf{v}\ done\ (1) = done\ (1)$
$\exists\mathbf{v}\{\exists\alpha\,(done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A})\ \mathbf{or}\ \mathbf{B}\} = \exists\alpha\bar{\mathbf{A}}\{\mathbf{v}{:=}\mathbf{a}\}\ \mathbf{or}\ \exists\mathbf{v}\mathbf{B}$ (8)
$\exists\mathbf{w}\{\exists\alpha\,(done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A})\ \mathbf{or}\ \mathbf{B}\} = \exists\alpha\{done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \exists\mathbf{w}\mathbf{A}\}\ \mathbf{or}\ \exists\mathbf{w}\mathbf{B}$ (9)
$\exists\mathbf{w}\{\exists\alpha\,(done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A})\ \mathbf{or}\ \mathbf{B}\} = \exists\mathbf{w},\alpha\{done\ (\mathbf{v}{:=}\mathbf{a})\ \mathbf{and}\ \mathbf{A}\}\ \mathbf{or}\ \exists\mathbf{w}\mathbf{B}$ (10)

In (8) $\bar{\mathbf{A}} \equiv \mathbf{A}$ unless $\mathbf{and} \equiv\ \&$. In the latter case $\bar{\mathbf{A}} \equiv do\ (\mathbf{A})$. In (9) and (10) the variable **w** is different from **v**. In (9) the variable **w** does not occur in **a**, whereas in (10) it does.

If Group.

$do\ (\mathbf{if}\ 0\ \mathbf{then}\ \mathbf{A}\ \mathbf{else}\ (v,w)\ \mathbf{B}) = do\ (\mathbf{A})$
$do\ (\mathbf{if}\ [\mathbf{a},\mathbf{b}]\ \mathbf{then}\ \mathbf{A}\ \mathbf{else}\ (\mathbf{v},\mathbf{w})\ \mathbf{B}) = do\ (\mathbf{B}\{\mathbf{v}{:=}\mathbf{a},\ \mathbf{w}{:=}\mathbf{b}\})$

d) **Replacement Rule.**

$$\frac{\mathbf{A} = \cdots\mathbf{B}\cdots,\quad \mathbf{B} = \mathbf{C}}{\mathbf{A} = \cdots\mathbf{C}\cdots}$$

where $\mathbf{B} = \mathbf{C}$ is a Computation axiom.

e) **Predicate Call Rule.**

$$\frac{\mathbf{A} = \cdots do\ (P\ (\mathbf{a}))\cdots,\quad P\ (x) = \mathbf{B}}{\mathbf{A} = \cdots do\ (\mathbf{B}\{x{:=}\mathbf{a}\})\cdots}$$

where $P\ (x) = \mathbf{B}$ is a Predicate axiom.

f) **Termination Rules.**

$$\frac{\mathbf{A} = done\ (0)}{\mathbf{A}} \qquad \frac{\mathbf{A} = done\ (1)}{\neg\,\mathbf{A}}$$

It is straight-forward, though perhaps not so trivial as before, to demonstrate that the axioms and inference rules of R do not lead outside of TP.

A note on the parallelism of R. Programming languages F and P are underspecified. The executing machine, by adopting a wrong proof strategy, may fail to reduce the program although the program is reducible. The languages N and L permit at most one reduction. Thus whatever can be reduced, will be reduced. R is underspecified. The presence of the explicit parallelism allows multiple markers *do* and *done* in programs to be reduced in different order. Each marker corresponds to a process. A reduction of one process generally kills other processes. For instance the satisfaction of one operand of a parallel *or* kills all processes in the other operand. This underspecification is intentional. A correct concurrent program should be written in such a way

that, no matter in what order the reduction goes, a deadlock or an infinite computation does not occur. This correctness condition can be expressed formally and any proof of correctness of a parallel program must contain the demonstration that the program will be reduced no matter which way the computation goes.

The difference between the *unintentional* non-determinism of underspecified languages F and P and the *intentional* non-determinism of underspecified R is that correct R-programs reduce regardless of the reduction strategy. Programs of R intentionally distribute the *do* and *done* processes by the judicious use of sequential and parallel connectives and quantifiers. R-programs are intended not only to protect themselves from the non-determinism, but actually to utilize it by employing a concurrent hardware. Programs in the underspecified languages F and P must rely on the *benevolence* of the executing machine. A benevolent machine supposedly finds a proof if there is one. A correct F-program may still fail to reduce on a *diabolical* F-machine. On the other hand, the programs in R do not make this extra-linguistic assumption on the character of its computing machine. They are willing to live with a *malevolent* executing machine provided the machine does not violate the basic rules of the game, i.e. the machine generates correct proofs of R.

6. Conclusions.

We have mentioned in section (2) that the proof-theoretical questions are easier to settle if there are as few deadlock situations as possible. This means that a *typeless* programming language (such as all of the languages treated in this paper) should be allowed to proceed even if it tries to perform $0.h$ or a similar operation. The situation in R is more complex and possibilities of deadlocks arise from an incorrectly designed scheme of synchronization of parallel processes via assignments to shared variables (two simultaneous assignments to a shared variable, or none at all). Deadlocks of the first type can be prevented by imposing a system of *decidable* types on the programs. This amounts to imposing *syntactic* restrictions on the legal S-terms and S-formulas. The author is presently working on such a scheme. It is actually an adaptation of Pascal-like typing within the framework of TP. It is the author's belief that the deadlocks occurring from parallel processes can be also eliminated by syntactic restrictions on legal terms and formulas of S. What kind of types are sufficient to prevent parallel deadlocks remains a very interesting and practical research problem.

The problems of termination of programs are *harder* than the problems of meanings of programs. A semantically correct program may still fail to terminate. By a semantically correct program we mean a function or predicate which is introduced as a conservative extension of TP or some other underlying semantic theory. A programmer willing to demonstrate semantical correctness (also called *partial* correctness) must explicitly produce the evidence that his function or predicate can be introduced into the theory. If he wants to prove the termination he has to demonstrate the existence of a proof of the program. If the program is a concurrent one, the programmer must show that any sequence of formulas legal in a proof can be extended to a proof. A sequence of formulas is legal in a proof in S when each formula in the sequence is either an axiom or the conclusion of a rule of inference applied to the preceding formulas. This will guarantee that no matter which way the computing machine goes, it will not miss a proof if there is one.

We think that the proof-theoretical questions of complexity of computations, absence of deadlocks, and termination should be formalized either in a suitable meta-theory or, preferrably, in TP itself. This is to be done by the well-known method of *arithmetization* of programs and their proofs. TP is a suitable theory because the Gödel numbers are straight-forward. For instance the term **if** x **then** 1 **else** $[x,0]$ can be assigned the Gödel number

$$[if\,c\,,[var,0],[const,1],[pair\,,[var,0],[const,0]]]$$

where *if c*, *var*, *const*, and *pair* are defined as different constants of TP. The reader will recognize that such Gödel numbers are similar to forms of Lisp.

Let us denote by $\lceil \mathbf{a} \rceil$ the Gödel number of the term or formula **a** of a programming language S. It is possible to introduce the predicate $Red_S\,(x,n,p\,)$ into TP. $Red_S\,(\lceil \mathbf{a} \rceil, \overline{n}, p\,)$ means that p is

the Gödel number of the proof in S of the S-formula $\mathbf{a}=\bar{n}$. Thus the program **a** in the deterministic language S terminates iff $\exists p,n\ Red_S(\lceil \mathbf{a} \rceil, n, p)$. In the case of a language with parallelism one needs a stronger predicate $Ared_S(\lceil \mathbf{A} \rceil, t)$ satisfied when t is the Gödel number of a tree of *all* possible derivations of the formula **A**. If $\exists t\ Ared_S(\lceil \mathbf{A} \rceil, t)$ then the formula **A** will be proven no matter how the execution goes.

In the further development of the formal theory of termination of S one can proceed to prove sufficient conditions under which the above predicates are satisfied. Because of the incompleteness of TP we cannot always hope to have $\vdash_{TP} \exists p,n\ Red_S(\lceil \mathbf{a} \rceil, n, p)$ for a sufficiently strong S. Since the theorems of S are always theorems of TP, S must be consistent and all the formulas

$$Red_S(\lceil \mathbf{a} \rceil, \bar{n}, p) \rightarrow \mathbf{a}=\bar{n} \qquad (1)$$

are true but generally unprovable in TP. We can strengthen TP by adding (1) as a new axiom schema. This will enable the development of the full theory of termination of programs in S within TP.

The class of functions which can be introduced into TP by recursive equations is called *provably* recursive functions. In the view of incompleteness of any sufficiently strong theory, no theory can admit all recursive functions. For the sake of simplicity we have dealt in this paper only with functions over pairs. One can introduce higher order functions, i.e. functionals, into TP. The functionals which can be introduced are so called *type recursive* or functionals of *finite types* [see for instance 12]. The class of functions and functionals definable in TP is the class of functions definable by transfinite induction up to any ordinal α such that $\alpha<\epsilon_0$ [12,13]. This class encompasses a huge number of functions. We believe that it includes all the functions and functionals which are practical to compute. But even if there were practical functions which need recursion of higher type than ϵ_0, TP can be strengthened by addition of so called *reflexion* axioms (similar to (1)) which progressively strengthen the power of TP to any degree required by practical applications.

We hope that we have convinced the reader that our method of two theories, one for meaning the other for control, is an eminently workable approach to the problems of the definition of programming languages.

[1] Bauer F. et al. Description of the Wide Spectrum Language CIP-L; Research Report, Technical University Munich, May 1983.

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

[3] Goguen, Tardo, An Introduction to OBJ: A Language for Writing and Testing Software Specifications, In: Specification of Reliable Software, 1979.

[4] Hermes H., Term Logic with Choice Operator, Lecture Notes in Mathematics, no. 6, Springer Berlin 1970.

[5] Hoare C. A. R., An Axiomatic Basis for Computer Programming, CACM, vol. 12. Oct. 1969.

[6] Jensen K., Wirth N., PASCAL User Manual and Report, Springer Berlin 1974.

[7] Kleene S., Introduction to Metamathematics; North-Holland, Amsterdam 1971.

[8] Kowalski R., Logic for Problem Solving; North Holland, Amsterdam 1979.

[9] Shapiro E., A Subset of Concurrent Prolog and its Interpreter, TR3 Institute for New Generation Computer Technology, Jan 1983.

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

[11] Stoy J., Denotational Semantics: The Scott-Strachey Approach to the Mathematical Semantics. MIT Press, Cambridge 1977.

[12] Schuette K., Proof Theory, Springer Berlin, 1977.

[13] Schwichtenberg H., Elimination of Higher Type Levels in Definitions of Primitive Recursive Functionals by Means of Transfinite Recursion, In: Logic Colloquium '73, Rose and

Shepherdson (eds.), North Holland, Amsterdam 1973.

[14] Voda P. J., R-Maple: A Concurrent Programming Language Based on Predicate Logic, Part I: Syntax and Computation; Technical Report of Dept. Comp. Science UBC, Vancouver August 1983.

[15] Voda P. J., A First Order Theory of Pairs, Technical Report of Dept. Comp. Science UBC, Vancouver May 1984.