Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

A FUNCTIONAL PLUS PREDICATE LOGIC PROGRAMMING LANGUAGE

M. Bellia, P. Degano, G. Levi

Nota interna B80-16

Giugno 1980

Marco Bellia(+), Pierpaolo Degano(*), Giorgio Levi(+)

1. Introduction.

In the last few years, languages based on first order logic became very popular, due to several reasons that make them good candidates not only as specification languages, but also as practical programming languages.

The main features of such languages are:

i) They have a clear mathematical basis, which allows to define a straightforward formal semantics and which provides a natural environment for proving properties of programs;

ii) They are nice examples of applicative languages, whose semantics is not based on state transitions, and they lead to a hierarchically structured non von Neumann programming style /1/;

iii) Last but not least, today's technology allows to design efficient implementations /2-6/.

Predicate logic programming languages can be classified according to the kind of procedures they define. In the first class (relational languages) procedures are defined as relations. The first example of a relational language is PLANNER /7/. Kowalski's language /8/ is a milestone within this family, because of the formal definition of procedures as sets of Horn clauses, and its clean mathematical semantics /9/. On Kowalski's footsteps, PROLOG /2-6, 10-11/ and other similar languages /12-14/ have been proposed. In the second class of languages (functional languages) procedures are defined by sets of functional equations. Languages within such a class have been motivated by several different problems, namely proving program properties in formal systems /15-19/, and abstract data type specification /20-23/.

There are no definite arguments in favour of one class against the other, yet each class has its own appealing features. Namely, a uniform evaluation rule can more easily be defined for functional languages, while relational languages lead to non-deterministic interpreters. Properties of programs (i.e. lemmas and theorems to be used in symbolic simplifications) are more expressively defined within the functional approach. On the other hand, relational languages are exactly what is needed to describe procedures with more than one output.

The language described in this paper is based on an attempt to combine relational and functional languages in a unified environment, which provides the best features of both approaches.

Our goal was to design a first order logic language, which allows to define both functions and procedures. Our language is a proper extension of functional languages enriched with somewhat constrained Horn clauses. The constraints are concerned with distinguishing between input and output parameters and sequencing of literals. In the resulting language, predicates play the role of standard programming language procedures. Moreover, it is possible to define an efficient deterministic interpreter.

(+) Istituto di Elaborazione dell'Informazione - C.N.R., Via S.Maria, 46 - I56100 Pisa (ITALY)

(*) Istituto di Scienze dell'Informazione - Università di Pisa - I56100 Pisa (ITALY)

2. The Syntax of FPL.

The Functional plus Predicate Logic (FPL) programming language is a strongly typed first order language, whose programs are equations defined according to first order logic over the alphabet $A=\{S,C,D,V,F,R\}$, where:

S is a set of identifiers. Given S, we define a sort s which is:

i) simple if $s \in S$, ii) functional if $s \in S^* \rightarrow S$, iii) relational if $s \in S^* \rightarrow S^*$.

C is a family of sets of constant symbols indexed by simple sorts.

D is a family of sets of data constructor symbols indexed by functional sorts.

V is a family of denumerable sets of variable symbols indexed by simple sorts.

F is a family of sets of function symbols indexed by functional sorts.

R is a family of sets of predicate symbols indexed by relational sorts.

Families are defined in the language by declarations, which assign a specific simple or functional or relational sort to each object.

Examples are:

0:→NAT; succ: NAT→NAT

nil:→NLIST; cons: NAT x NLIST→NLIST

+: NAT x NAT→NAT

eqn: NAT x NAT→BOOL

ndiv: NAT x NAT→NAT x NAT.

A FPL program is a set of declarations and equations. Each symbol occurring in an equation must be declared.

The syntax of equations is based on the standard concepts of term and atomic formula.

A term is either a data term or a functional term.

A data term of sort s ($s \in S$) is

i) a constant symbol of sort s,

ii) a variable symbol of sort s,

iii) a data constructor application $d(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ are data terms of sort $s_1,\ldots,s_n$ and $d \in D$ has sort $s_1 \times \ldots \times s_n \rightarrow s$.

A functional term of sort s ($s \in S$) is a function application $f(t_1,\ldots,t_n)$, such that $t_1,\ldots,t_n$ are data terms of sorts $s_1,\ldots,s_n$ and $f \in F$ has sort $s_1 \times \ldots \times s_n \rightarrow s$.

An atomic formula is either

i) a functional atomic formula of the form t=d, where d is a data term of sort s and t is a term of the same sort, or

ii) a relational atomic formula of the form $r(\underline{in}:t_1,\ldots,t_m ; \underline{out}:t_{m+1},\ldots,t_n)$, such that $t_1,\ldots,t_m,t_{m+1},\ldots,t_n$ are data terms of sorts $s_1,\ldots,s_m,s_{m+1},\ldots,s_n$ and $r \in R$ has sort $s_1 \times \ldots \times s_m \rightarrow s_{m+1} \times \ldots \times s_n$.

A constraint is either

i) an atomic formula, or

ii) a formula of the form c_1,c_2 such that c_1 is an atomic formula and c_2 is a constraint.

Constraints are used to combine functional terms (function calls) and atomic formulas (procedure calls) in a program. Constraints define a local environment which is shared by (and allows the interaction among) its components. Constraints can be used within function and procedure definitions, according to the following syntax of equations.

Equations are formulas of the following form $l \leftarrow r$, where l is the left part and r is the right part, such that its left part l is an atomic formula possibly followed by a constraint and its right part r is either empty or a constraint.

The equation is functional or relational, according to the type of its atomic formula.

Example:

1. true:→BOOL
2. false:→BOOL
3. 0: → NAT
4. s: NAT→NAT
5. minus: NATxNAT→NAT
6. lt: NATxNAT→BOOL
7. ndiv: NATxNAT→NATxNAT

e8. minus(x,0)=x ←
e9. minus(s(x),s(y))=z←minus(x,y)=z
e10. lt(0,s(x))=true ←
e11. lt(x,0)=false ←
e12. lt(s(x),s(y))=z←lt(x,y)=z
e13. ndiv(in:x,y:out:0,x),lt(x,y)=true←
e14. ndiv(in:x,y;out:s(q),r),lt(x,y)=false←ndiv(in:z,y;out:q,r),minus(x,y)=z
e15. isfact(x,y)=false,ndiv(in:y,x;out:z,s(r))←
e16. isfact(x,y)=true,ndiv(in:y,x;out:z,0)←

Declarations 1-3, 4, 5-6, 7 are constant, data constructor, function and relation declarations, respectively. The example is completed with the functional equations e8-e12, e15-e16 and the relational equations e13-e14.

The above definition of equation is inadequate, since context-dependent conditions on variable occurrences are needed to guarantee proper nesting of constraints and binding of local variables. Some more definitions are needed to introduce the conditions. In order to give some insight into the meaning of the conditions, we will informally use operational arguments.

A definition contains atomic formulas of the form $r(\underline{in}:x_1,\ldots,x_n;\underline{out}:y_1,\ldots,y_m)$, or $f(x_1,\ldots,x_n)=y$. Let us define, for each atomic formula a the multisets of input and output variable occurrences. Namely,

$M_{in}(a)$ is the multiset of the variable occurrences in terms $x_1,\ldots,x_n$, while
$M_{out}(a)$ is the multiset of the variable occurrences in terms $y_1,\ldots,y_m$ or y.

Each definition has a <u>header</u>, consisting of the leftmost atomic formula, and a set of <u>invocations</u>, whose element are the other atomic formulas. Let H and $I=\{I_i\}$ be the header and the set of invocations of an equation e.

<u>Condition 1</u>. The multisets $M_{in}(H)$ and $M_{out}(I)=\bigcup_i M_{out}(I_i)$ must be sets.

The absence of multiple occurrences of a variable in the header corresponds to the left-linearity, while the absence of multiple output occurrences of a variable in the set of invocations rules aliasing out.

Examples of equations not satisfying condition 1 are:

eq(x,x)=true ← (since it would impose a specific relation on input values),

r(<u>in</u>:x;<u>out</u>:y,z) ← g(w)=y,f(x)=w,q(<u>in</u>:x;<u>out</u>:w,z) (since variable w is (output) constrained (i.e. could be computed) by two different constraints).

<u>Condition 2</u>. $M_{in}(H) \cap M_{out}(I)=\phi$.

Disjointness of sets of header input variables and invocations output variables in an equation is connected with the non invertibility of programs. As an example, the equation p(<u>in</u>:x,y;<u>out</u>:z) ← r(<u>in</u>:y,z;<u>out</u>:x),f(y)=z,is ruled out, because it imposes a constraint on the variable x (i.e. it may invert with respect to x).

<u>Condition 3</u>.

<u>3.1</u>. All variable symbols occurring in $M_{out}(H)$ and $M_{in}(I_i)$, must belong either to $M_{in}(H)$ or to $M_{out}(I_k)$, where I_k is an inner invocation (the innermost invocations possibly being in the left part constraint).

<u>3.2</u>. For each invocation I_k in a right part constraint, $M_{out}(I_k)$ must contain at least one variable symbol belonging either to $M_{out}(H)$ or to $M_{in}(I_i)$, where I_k in an inner invocation.

Example of equations which do not satisfy condition 3 are:
r(in:x,y;out:z,w),f(x,y)=t←h(t)=w (since the output z cannot be computed),
p(in:x,y;out:z)←g(t,w)=z,f(x,y)=w (since intermediate variable t cannot be computed),
f(x,y)=z,k(x,y,t)=z←h(x)=t (since the left part constraint could not be computed before the right part constraint),
h(x)=t←g(x,z)=t,f(x,t)=z (since there exists a circular precedence relation between invocations),
r(in:s(x),y;out:s(z))←r(in:x,y;out:z),f(x,y)=w (since the invocation f(x,y)=w never needs to be computed).
r(in:x,y;out:z)←h(x)=z,f(x,y)=false (since false is a constant symbol occurring as output of an invocation which will never be computed).

Thus far we have defined well-formed equations. A set of equations should denote sets of procedures. Since our aim is to restrict sets of equations so as to define (deterministic) procedures by disjunct cases, we are forced to introduce more definitions and conditions.

Conditions on a set of equations are concerned with the non superposition property on the equations left parts and relies on (first order) unification.

An equation left part consists of a header and a (possibly empty) set of invocations. Let c be any header or invocation,

i) $n(c)$ be the function or relation symbol in c,
ii) $D_{in}(c)$ the n-tuple of input data terms in c,
iii) $D_{out}(c)$ the n-tuple of output data terms in c.

Given a set of equations $E=\{e_i\}$, the set has the non superposition property if for any pair of equations $l_i \leftarrow r_i$, $l_j \leftarrow r_j$, the left parts l_i and l_j are non overlapping.

Condition 4. Two left parts l_i and l_j are non overlapping if one of the following properties holds:

1) $n(h_i) \neq n(h_j)$, where h_i and h_j are the header of l_i and l_j.
2) $D_{in}(h_i)$ and $D_{in}(h_j)$ are non-unifiable.
3) $D_{in}(h_i)$ and $D_{in}(h_j)$ are unifiable with most general unifier λ, l_i and l_j have constraints k_i and k_j, and $[k_i]_\lambda$, $[k_j]_\lambda$ are syntactically disjoint.

Condition 5. Two constraints k_i and k_j are syntactically disjoint if one of the following properties holds.

1) k_i and k_j are invocations, $n(k_i)=n(k_j)$, $D_{in}(k_i)=D_{in}(k_j)$ and $D_{out}(k_i)$, $D_{out}(k_j)$ are non-unifiable.
2) k_i and k_j have the form c_{i1}, k_{i2} and c_{j1}, k_{j2} respectively, and either
   2.1 c_{i1} and c_{j1} are syntactically disjoint, or
   2.2 $n(c_{i1})=n(c_{j1})$, $D_{in}(c_{i1})=D_{in}(c_{j1})$, $D_{out}(c_{i1})$ and $D_{out}(c_{j1})$ are unifiable with most general unifier λ, and $[k_{i2}]_\lambda$, $[k_{j2}]_\lambda$ are syntactically disjoint. The following are sets of overlapping equations:

{+(x,0)=x←,
+(0,x)=x←},

{plus(in:x,y;out:y),eq(x,0)=true←,
plus(in:x,y;out:z)←plus(in:y,x;out:z),

plus(in:x,y;out:x),eq(y,0)=true←,
plus(in:x,y;out:0),+(x,y)=0←,
plus(in:x,y;out:z),+(x,y)=z←}

Let us finally introduce the syntactic construct program. A program has the same form of an equation right part, namely it is a constraint. Hence a program consists of a set of invocations $I=\{I_i\}$, whose variables must obey the following conditions.

Condition 6. $M_{out}(I)=\bigcup_i M_{out}(I_i)$ must be a set.

Condition 7.

7.1. For each I_i in a program, each variable belonging to $M_{in}(I_i)$ must belong to $M_{out}(I_k)$, where I_k is a inner invocation.

7.2. For each I_k in a program, $M_{out}(I_k)$ must contain at least one variable symbol which belongs to $M_{in}(I_i)$, where I_k is an inner invocation.

Conditions 6 and 7 ensure that a program is closed.

In section 3 we will introduce FPL operational semantics, which allows to define a computation from given program and set of equations. It is worth noting that our lengthy and tedious definition of the FPL syntax (typically, the conditions for well-formedness of equations, sets of equations and programs), was mainly concerned with semantic properties, which can be incorporated into the syntax and statically checked. The possibility of defining a deterministic FPL interpreter relies exactly on such conditions.

Let us finally note that the syntax we have defined does not allow function composition. However, our syntax has to be seen as the abstract FPL syntax. The concrete syntax will allow to use standard function composition. Namely, a general term obtained by function composition can replace a functional term every where in an equation, but in an equation header.

The functional and relational aspects of FPL can be distinguished leading to two different subsets of the language.

The language obtained ruling out relational atomic formulas and left part constraints, is a subset of the functional language TEL /15/, since it does not allow to express properties.

Ruling out functional atomic formulas and left part constraints, we obtain a specific class of Horn clauses, characterized by input-output separation and ordering of the right part atomic formulas. The above constraint forbids program invertibility, yet leads to a deterministic interpreter.

FPL can be extended by releasing some of the above conditions in order to allow to express properties of programs as well. Such an extension, however, is outside the scope of this paper.

3. Operational Semantics.

The operational semantics will be defined by describing the FPL interpreter. The interpreter consists of a set of mutually recursive procedure (EVAL,MATCH,UNIFY) which operate on abstract representations of programs and constraints (closure structures), that will be defined in the following.

A set of invocations $I=\{I_i\}$ can be represented as a closure set, which contains a closure for each invocation I_i. The closure corresponding to invocation I_i is the pair $c=<I_i,env(I_i)>$, where $env(I_i)$ is a set of bindings for all the input variables of I_i (which are also input variables of closure c).

A binding possibly associates an input variable v to the closures which correspond to those invocations in I which have v among their output variables.

A closure structure is a set of closures $C=\{c_i\}$, such that:

i) For each closure c_i in the set and for each input variable v in c_i, v is bound to exactly one closure in C.

ii) The multiset of output variables of all the closures of C is a set.

Let Γ be a closure structure. If we associate a labeled node to each closure in Γ and a directed arc from node labeled c_i to node labeled c_j, if some input variable of c_i is bound to c_j. Then, a closure structure is a directed graph.

Let Γ be a closure structure and c_i be a closure in Γ. The substructure of Γ rooted at c_i is the closure structure Γ/c_i defined as follows;

i) $c_i \in \Gamma/c_i$

ii) If closure c_k belongs to Γ/c_i, then Γ/c_i contains all the closures of Γ whose output variables are input variables of c_k.

A substitution is a closure structure λ, such that for each closure $c \in \lambda$ and for each output variable v in c, there exists no closure belonging to the substructure λ/c which has v among its input variables.

A root is any closure c_i of λ, such that there exists no closure in λ having an input variable bound to c_i. Hence a substitution is a directed acyclic graph. Note that each substructure of a substitutions is itself a substitution.

The composition $\lambda.\mu$ of a substitution λ with a substitution μ is the closure structure containing the following closures.

i) All the closures of μ.

ii) Only those closures of λ whose output variables are different from the output variables of closures of μ.

The closure structure $\lambda.\mu$ is itself a substitution, because it is acyclic. In fact, the presence of a cycle would require the existence of a closure c_i, such that $c_i \in \mu$ and $c_i \in \lambda.\mu$, which has as input variable a variable v which is bound to some closure c_j, such that $c_j \in \lambda$ and $c_j \in \lambda.\mu$. Even if such a c_j belonging to λ may exist, c_j cannot belong to $\lambda.\mu$ by definition of composition, since variable v must also be an output variable of μ.

A set of closures $C=\{c_i\}$ can be appended to a substitution λ, only if:

i) For each closure c_i and for each input variable v of c_i, v is an output variable of some closure in λ.

ii) The multiset of output variables of C is a set.

iii) The sets of output variables of C and λ are disjoint.

The result $C \| \lambda$, of appending a legal set of closures C to a substitution λ is a substitution.

A FPL program, as defined in Section 2, is a single-rooted substitution (i.e. a directed single-rooted acyclic graph). A program is a closure structure, because

i) Each input variable in an invocation is bound to at least one invocation (condition 7.1) and such an invocation happens to be unique (condition 6).

ii) The multiset of output variables of its invocations is a set (condition 6).

Moreover, a program is a substitution, i.e. it is acyclic, because each invocation input variable is bound to an inner invocation (condition 7.1). Finally, it is single-rooted because condition 7.2 ensures that there exists only one invocation which does not occur in any binding.

The interpreter procedure EVAL will operate on a program, giving a new program as output. In order to allow single-rootness to be preserved by EVAL, the substitution corresponding to a program will be "topped" with a virtual closure (which models the external environment) which contains an empty invocation and has as input variables all the output variables of the program.

It is worth noting that each substructure of a program is a program.

A set of closures $C=\{c_i\}$ is a schematic closure structure if

i) For each closure c_i and for each variable v in c_i, either v is bound to a unique closure of C, or v is free.

ii) The multiset of all the output variables of closures in C is a set.

Hence, a schematic closure structure is different from a closure structure only because some input variables can be free. Schematic substructures and schematic substitutions can easily be defined following the definitions given for the closure structure case. In particular, a schematic substitution G is an acyclic schematic closure structure.

Let free(G) the set of free input variables in G. A schematic substitution G can be instantiated by a substitution λ, if

i) For each variable v in free(G), there exists a closure in λ having v among its outputs.

ii) The sets of output variables of G and λ are disjoint.

The instantiation $[G]_\lambda$ contains all the closures of G and only those closures of λ which belong to a λ/c, such that c has some variable in free(G) among its outputs. $[G]_\lambda$ is a substitution, because all its inputs are bound, all its outputs are different, and there are no cycles since each input of a closure of λ cannot be an output of a closure of G.

A FPL equation e is a triple $<H(e),G_l(e),G_r(e)>$, where:

i) H(e) is the header.

ii) $G_l(e)$ is the left part constraint.

iii) $G_r(e)$ is the right part constraint.

It is possible to prove that both $G_l(e)$ and $G_r(e)$ are schematic substitutions. In fact, for each closure c corresponding to an invocation of either $G_l(e)$ or $G_r(e)$, and for each variable v in c, v is either free, or bound to at least one closure (condition 3.1), which is unique (condition 1). Moreover, the multiset of output variables is a set (condition 1), and there are no cycles, since v can only be bound to an inner constraint (condition 3.1).

We are now able to describe the interpreter procedures.

UNIFY (X:n-tuple of terms,D:n-tuple of terms, λ:substitution);
returns <failure/success, μ:substitution>

X is a n-tuple of data terms $(x_1,\ldots,x_n)$, which contain free variables not occuring in any closure of λ, with no multiple occurrences of the same variable.

D is a n-tuple of data terms $(d_1,\ldots,d_n)$, whose only variables are bound to some closure of λ.

UNIFY is basically first order unification, which returns failure or, in case of success, a set of associations of the form t=v, where v is a variable and t is a data term. In our framework, each association is a closure, having the association as the invocation, variable v as output, and all the variables occurring in t as inputs. As soon as a new association is generated, the corresponding closure is inserted in the (initially empty) set of closures MGU.

Unification proceeds like standard first order unification comparing terms of X to terms of D (possibly) associating variables occurring in X to terms occurring in D. The difference has to do with bound variables occurring in D, which cannot be instantiated, just because they are bound. If unification reaches the point where a bound variable b_i is matched against a non-variable data term t_k (which occurs in X), the following actions are taken.

Step 1. If b_i is bound to closure c whose invocation has the form $t_j=b_i$ and t_j is a data term, then unification proceeds with b_i replaced by t_j.

Step 2. Otherwise, standard unification is suspended and a call is made to EVAL, passing the closure c (to which b_i is bound) and the substitution λ, as parameters. If EVAL returns failure, UNIFY returns failure. Otherwise, EVAL returns a new substitution λ', such that the closure of λ' which has b_i among its outputs is different from c. Step 1 is taken one more time, possibly leading to a further evaluation.

Eventually, unless some EVAL process does not terminate, unification will end up with failure or with a set of closures MGU and a substitution λ^*.

REMARK. The output variables of MGU are exactly the variables occurring in the n-tuple X, while its input variables are all bound to some closure of λ^*. From the conditions imposed on variables occurring in X (which also prevent circularity in most general unifiers), it follows that the set of closures MGU can be appended to the substitution λ^*.

UNIFY returns the substitution μ=MGU $\parallel \lambda^*$.

MATCH (e:equation,a:atomic formula, λ:substitution);

returns <failure/success, μ:substitution>

a is an atomic formula, whose only variables are bound to closures of λ.

e is an equation, with header H(e) and (possibly empty) left part constraint $G_1(e)$.

Step 1. If the function (or predicate) symbols occurring in a and H(e) are different, returns failure.

Step 2. Otherwise, let X be the n-tuple of input data terms in H(e) and let D be the n-tuple of input data terms in a.

REMARK. When MATCH is called, e is a renaming of a FPL equation. Hence all the variables in X do not occur in any closure of λ. Moreover, because of condition 1, no variable can have multiple occurrences in X. Finally, all the input variables of D are bound to some closure of λ. Therefore, UNIFY can be applied to parameters X, D and λ.

Call UNIFY(X,D, λ). If UNIFY returns failure, return failure. Otherwise, let λ' be the substitution returned by UNIFY. If G_1 is empty, return $\mu = \lambda'$.

Step 3. Otherwise, let $\lambda'' = [G_1(e)]_{\lambda'}$ be the instantiation of the schematic substitution $G_1(e)$ by the substitution λ'.

REMARK. Each variable in free($G_1(e)$) is bound to some closure in λ', because a free input variable in the left part constraint can only be an input variable of H(e) (condition 3.1), and because all the input variables of H(e) are output variables of λ' (by definition of UNIFY). Hence λ' can be used to instantiate $G_1(e)$.

Let $C=\{c_i\}$, $1 \leqslant i \leqslant k$, be the k-tuple of closures, such that each c_i is a root of λ''. Set i=1 and $\lambda_1 = \lambda''$.

Step 4. Call EVAL(λ_i,c_i). If EVAL returns failure, return failure. Otherwise, if i=k return the substitution $\mu = \lambda' . \lambda_{i+1}$, which is the composition of the output substitution of UNIFY and the output substitution λ_{i+1} of the last EVAL.

Step 5. If $i \neq k$, increase i by 1, and iterate Step 4.

REMARK. If eventually, MATCH returns success, its output substitution μ has among its output variables all the input variables of H(e) and all the output variables of $G_1(e)$ (if any).

EVAL (λ:substitution,c:closure);

returns <failure/success, μ:substitution>, c is any closure of λ.

Step 1. Let I be the invocation associated with the closure c. According to the form of I, one of the following actions is taken.

1.1 If I is empty (top closure of a program), let $C=\{c_i\}$, $1 \leqslant i \leqslant k$, the k-tuple

of closures to which the input variables of I are bound. Set i=1 and $\lambda_i = \lambda/c_i$.

1.1.1 Call EVAL(λ_i,c_i). If EVAL returns failure return failure. Otherwise let λ'_i be the substitution returned by EVAL. If i=k, let $\lambda' = \lambda'_i$ and go to step 2, otherwise increase i by 1, set $\lambda_{i+1} = \lambda/c_{i+1} . \lambda'_i$ and iterate step 1.1.1.

1.2 If I has the form d=v, where d is a data term and v is a variable, then

1.2.1 If d is not a variable, then return λ.

1.2.2 If d is an (input) variable, let c' be the (unique) closure in λ to which d is bound. Call EVAL(λ/c',c'). If EVAL returns failure, return failure. Otherwise, let λ' be the output substitution of EVAL and go to step 2.

1.3 If I is an atomic formula, for each equation e_i in the global set of equations E, a nondeterministic call to MATCH is performed, MATCH(e_i,I, λ^+), where e_i is a new consistent renaming of equation e_i, and λ^+ is the substructure of λ rooted at c, c non included.

1.3.1 If no MATCH succeeds, return failure. Otherwise, let e'_k and λ_k be one successful equation and the output substitution of the corresponding MATCH. If $G_r(e'_i)$ is empty, set $\nu' = \lambda_k$ and go to 1.3.3.

REMARK. Because of the non superposition condition (conditions 4 and 5) on sets of equations, a unique MATCH can terminate successfully. However, we are not allowed to handle the different equations sequentially since MATCH could be nonterminating.

1.3.2 Let ν be the instantiation $[G_r(e'_k)]_{\lambda_k}$, of the schematic substitution, associated to the right part constraint of the successful equation by the output substitution of the successful MATCH, and $\nu' = \lambda_k . \nu$.

REMARK. λ_k can be used to instantiate $G_r(e'_k)$, because each variable in free($G_r(e'_k)$) is either an input variable of $H(e'_k)$ or an output variable of $G_l(e'_k)$ (because of condition 3.1), and λ_k has all such variables as output variables (see the last remark to MATCH). Moreover, for each output variable v of $G_r(e'_k)$, v cannot be an output variable of λ_k. In fact, because of equation renaming, for v to be an output variable of λ_k, v must be either an input variable of $H(e'_k)$ (contradictory because of condition 2) or an output variable of $G_l(e'_k)$ (contradictory because of condition 1).

1.3.3 Let X be the n-tuple of output data terms of closure c, and D be the n-tuple of the output data terms of $H(e'_k)$.

REMARK. We want to show that X, D and ν' are legal parameters for UNIFY. We must prove that

i) There are no multiple occurrences of a variable in X (by condition 6, i.e. absence of aliasing in a procedure call).

ii) All the variables in D are bound to some closure in ν'. Each variable in D is an output variable of $H(e'_k)$. By condition 3.1 it must also be either an input variable of $H(e'_k)$ or an output variable of $G_l(e'_k)$ or $G_r(e'_k)$. On the other hand, all the output variables of $G_r(e'_k)$ are outputs of ν, while all the input variables of $H(e'_k)$ and the output variables of $G_l(e'_k)$ are output variables of λ_k. Hence, they are all output variables of $\nu' = \lambda_k . \nu$.

iii) Each variable v in X is not an output variable of any closure in ν'. Initially, c is the only closure in λ having v as output variable. It is rather easy to prove that the only new output variables directly generated by MATCH and UNIFY are variables coming from renamed equations (and therefore different from v). We only need to show that each recursive call to EVAL (via MATCH and UNIFY) has the following property.

EVAL property. Let μ be the output of EVAL(λ,c); for each closure c' such that:
a) $c' \varepsilon \mu$ b) $c' \notin \lambda$ c) c' has an output variable which is also an output variable of some closure c" in λ,
the closure c" belongs to λ/c.

We will assume here the property to hold.

1.3.4 Call UNIFY(X,D, ν'). If UNIFY fails, returns failure. Otherwise, let λ^* be the output substitution of UNIFY.

REMARK. λ^* has all the output variables of c as outputs.

Let λ' be the structure which contains only those closure of λ^* which belong to substructures of λ^* rooted at closures which have as output variable an output variable of c.

REMARK. λ' is a substitution.

Step 2. Return $\mu = \lambda . \lambda'$.

REMARK. The EVAL property follows directly from the above construction.

A FPL program Π is evaluated by calling EVAL with substitution Π and with the unique root of Π as closure.

EVAL is clearly based on an external rule. Since our language has no builtin data types, and since "constructors are not evaluated", the FPL rule is a call-by-need, whose behaviour can be summarized as follows. "An atomic formula is evaluated so much as it is needed to allow unification".

The above defined abstract interpreter suggests an efficient implementation, which does not require equation renaming, and which modifies programs and substitutions by side effects, through the use of structure sharing. The same technique was successfully used in several predicate logic language implementations and theorem provers. In fact, with structure sharing, different instances of the same atomic formula are identified, thus avoiding multiple evaluations of atomic formulas which typically arise in call-by-name interpreters.

Even if the language is deterministic, the above described interpreter is nondeterministic. The EVAL Step, in which a program is nondeterministically MATCHED against all the equations left parts, could be implemented by backtracking, provided that the following property holds.

Backtracking property. Let I be an invocation whose input variables are bound in a substitution λ, and let $E=\{e_1,\ldots,e_n\}$ be a set of equations, such that for each equation e_i in E, $H(e_i)$ contains the function or predicate symbols occurring in I. If MATCH(e_k,I, λ), for some $e_k \varepsilon$ E, diverges, then MATCH(e_i,I, λ) diverges for all e_j belonging to E.

The above property holds if one more simple condition is imposed on sets equations. For the sake of brevity, the condition will not be described here. Let us only remark that if we take equations satisfying such a condition (which, roughly speaking, are simply good recursive definitions), the call-by-need and structure sharing implementation is in a sense "optimal", because all the evaluations which could have been performed within a failing MATCH are transmitted to the next MATCH, and would have been, in any case, performed by the successful MATCH, if any.

We will now give an example showing our use of the left part constraint, which allows recursive by cases definitions (without built-in conditional), with cases being defined by general atomic formulas. The example shows the evaluation of the program isfact(s(s(0)),s(0)) with equations e8,...,e16 in Section 2 (i(c) denotes the invocation of closure c).

4. Fixed-point Semantics.

In this Section, we will describe the fixed-point semantics of a set of equations $E=\{e_i\}$. For the fixed-point semantics, each equation e can be seen as a pair $<H(e_i), G(e_i)>$; such that $G(e_i)$ is the set of all the invocations occurring both in the left part and in the right part of e_i. It is worth noting that generally two equations

1. P , Q ← R , S 2. P ← R , S , Q

which differ only because one invocation occurs in the left part and in the right part, are different both from the operational and the mathematical viewpoint. The difference is only operational if invocation Q satisfies condition 3.2 (i.e. it has at least one output variable, which is an input to S or to R or an output of P). If this is the case equation 2 is a legal equation. The operational difference is concerned with nondeterminism. With equation 1, as soon as a MATCH succeeds, the other nondeterministic attempts in EVAL can be killed (since we are guaranteed from conditions 4 and 5 that any other MATCH would fail). Failing in the evaluation of Q, within MATCH, would just kill the current attempt. With equation 2, a failure in the evaluation of Q could only be detected after the successful MATCH. This would require to backtrack to a choice-point which had already succeeded (nonrecursive backtracking). This situation corresponds to the fact that equation 2 could possibly have a superposition with other equations. In such a case we are not guaranteed that when a match is successful, no other successful MATCHing is possible.

On the other hand, if 2 does not satisfy condition 3.2, equation 2 is not a legal equation. As a matter of fact, equations 1 and 2 would have a completely different semantics if the evaluation of Q diverges or fails. In fact, in such a case, Q would not be evaluated by equation 2.

The fixed-point semantics gives to equation e_i a semantics which is equivalent to the operational semantics of e_i, only if all the invocations of $G(e_i)$ which do not satisfy condition 3.2 occur in the left part of e_i (i.e. if e_i is a legal equation). The fixed-point semantics of E is a model of E, obtained as the fixed-point of a transformation φ_E on interpretations. Our fixed-point semantics is very close to the semantics defined in /9/. Our semantics however, is a call-by-name semantics. Therefore our domain will contain an undefined object ω_s, for each simple sort s.

Interpretations are defined on an abstract domain A, which is a family of sets A_s, each set beeing indexed by a sort s occurring in E. Each A_s is defined as follows:

i) ω_s belongs to A_s;

ii) All the constant symbols of sort s, occurring in E, are in A_s;

iii) For each data constructor symbol d of sort $s_1 \times \ldots \times s_n \rightarrow s$, A_s contains all the terms $d(t_1,\ldots,t_n)$, such that $t_1,\ldots,t_n$ belong to $A_{s1},\ldots,A_{sn}$, respectively. A term belonging to a family A is undefined if it contains ω_s, for some sort s which indexes a set A_s in A.

An interpretation ξ_i is any subset of the interpretation base B. The interpretation base B is a set of atomic formulas defined as follows:

i) For each function symbol f (occurring in E) of sort $s_1 \times \ldots \times s_n \rightarrow s$, B contains all the formulas $f(t_1,\ldots,t_n)=t$ such that $t_1,\ldots,t_n$ and t have sorts $s_1,\ldots,s_n,s$ respectively, and term t is not undefined.

ii) For each predicate symbol P(occurring in E) of sort $s_1 \times \ldots \times s_m \times s_{m+1} \times \ldots \times s_n$, B contains all the formulas $P(\underline{in}:t_1,\ldots,t_m;\underline{out}:t_{m+1},\ldots,t_n)$, such that $t_1,\ldots,t_m,t_{m+1},\ldots,t_n$ have sorts $s_1,s_m,s_{m+1},\ldots s_n$ respectively, and terms $t_{m+1},\ldots,t_n$ are not undefined.

Roughly speaking, an interpretation assigns output values to applications of functions and relations to ground input values. All the other applications have some undefined output. An interpretation ξ_i is "more defined" than interpretation ξ_j if $\xi_j \sqsubseteq \xi_i$, where $\sqsubseteq$ is set inclusion. Note that the partial ordering relation $\sqsubseteq$ on interpretations corresponds to an intuitive notion of better approximation. In fact, if $\xi_j \sqsubseteq \xi_i$, ξ_i assigns output values to some applications that in ξ_j had some undefined output.

Transformation φ_E maps interpretations on interpretations and is defined as follows.

Let ξ_i be any interpretation and $e_k = \langle H(e_k), G(e_k) \rangle$ be an equation of E. Equation e_k defines a transformation φ^k which maps ξ_i onto the interpretation $\xi_i^\kappa = \varphi^k(\xi_i)$, such that

1)) All the atomic formulas of ξ_i are in ξ_i^κ.

2) For each instantiation λ of variables to terms such that, for each invocation I_j in $G(e_k)$ either

2.1) $[I_j]_\lambda$ is in ξ_i, or

2.2) An output variable v of I_j, which is not an output variable of $H(e_k)$, is instantiated to an undefined term by λ,

the formula $[H(e_k)]_\lambda$ is in ξ_i^κ.

Note that λ must instantiate a variable v of sort s to a term belonging to A_s, and that if $G(e_k)$ is empty, condition 2.2 is satisfied for any instantation λ.

The transformation φ_E is the transformation defined by all the equations of E according to the above definition, i.e. $\varphi_E(\xi_i) = \bigcup_{e_k \in E} \varphi^k(\xi_i)$.

It can be proved that transformation φ_E on the set of interpretations partially ordered by set inclusion is monotonic and continuous. Hence, there exists the least fixed-point interpretation ξ^* such that $\xi^* = \varphi_E(\xi^*)$, which can be obtained by iteratively applying φ_E, starting with the empty subset of B, which is the bottom element of the partially ordered set of interpretations.

5. Conclusion

We have described a new first order logic language, which combines the functional and the relational approach. We have defined the fixed-point semantics and we have shown an interesting operational model which is both formal and close to efficient implementations. Both the fixed-point and the operational semantics are based on theorems that have been either assumed or informally proved in this paper. A complete formalization was certainly outside of the scope of the present paper.

We have some nice examples of FPL programs, that would be innatural and awkward, in a predicate language without left part constraints or in a functional language. The improved expressive power of the language is due to the presence of both the function and the procedure constructs and to the left part constraints which provide the full power of a built-in conditional, while saving the first order logic axiomatic flavour. One more interesting feature of FPL is its ability to describe non-strict functions and relations. Non strict functions, as the if-then-else, can easily and naturally be defined in FPL, just because of its call by need evaluation rule.

We have almost completed an experimental FPL interpreter, whose architecture is strictly related to the operational model of Section 3. The interpreter (written in LISP) is based on structure sharing and relies on LISP garbage collector.

Future work on FPL will include its extension to allow the definition of theorems and parallel programs. Our final goal is creating an FPL environment providing tools for program proving also.

REFERENCES

1. Backus,J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. C.ACM 21,8 (1978), 613-641.
2. Warren, D. Implementing PROLOG - compiling predicate logic programs. Report 39, Dept. of AI, Edinburgh, 1977.
3. Roussel,P. PROLOG: Manuel de Réference et d'utilisation, Groupe d'Intelligence Artificielle. Université d'Aix-Marseille.
4. McCabe,F.G. Programmer's guide to IC-PROLOG. Dept. of Computation and Control, Imperial College, London, 1978.
5. Roberts,G.M. An implementation of PROLOG. M.Sc. TH., Dept. of Computer Science, Univ. of Waterloo, 1977.
6. Szeredi,P. PROLOG- a very high level language based on predicate logic. 2nd Hungarian Computer Science Conf., Budapest, 1977.
7. Hewitt,C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. AI Memo 231, MIT Project MAC, 1972.
8. Kowalski,R.A. Predicate logic as a programming language. Information Processing 74, North Holland (1974), 556-574.
9. vanHemden,M.H., and Kowalski,R.A. The semantics of predicate logic as a programming language. J.ACM 23,4 (1976), 733-742.
10. Warren,D., Pereira,L.M., and Pereira,F. PROLOG - the language and its implementation compared with LISP. Proc. ACM Symp. on AI and PL, Rochester, 1977.
11. Colmerauer,A. Le grammaires de metamorphose. Group d'Intelligence Artificielle. Université d'Aix Marseille, 1975.
12. Tärnlund,S-Å. Horn clause computability. BIT 17 (1977), 215-226.
13. Hänsonn,Å. and Tärnlund,S-Å. A natural programming calculus. Proc. 6th IJCAI, Tokyo 1979.
14. Stickel,S. Invertibility of logic programs. Proc. 4th Workshop on Automated Deduction, Austin, 1979, 103-109.
15. Levi,G. and Sirovich,F. Proving program properties, symbolic evaluation and logical procedural semantics. Proc. MFCS '75. Lecture notes in Computer Science, Springer Verlag (1975), 294-301.
16. Burstall,R.M. Recursive programs: Proof, transformation and synthesis. Rivista di Informatica 7, 2 (1976), 25-42.
17. Aubin,R. Strategies for mechanizing structural induction. Proc. 5th IJCAI,Cambridge 1977, 363-369.
18. Boyer,R.S. and Moore,J S. A lemma driven automatic theorem prover for recursive function theory. Proc. 5th IJCAI, Cambridge, 1977, 511-519.
19. Cartwright,R. and McCarthy,J. First order programming logic. Proc. of 6th POPL, San Antonio, 1979, 68-80.
20. Burstall,R.M. and Goguen,J.A. Putting theories toghether to make specifications. Proc. 5th IJCAI, Cambridge, 1977, 1045-1058.
21. Goguen,J.A. and Tardo,J. OBJ-0 Preliminary user manual. Semantics and theory of computation Report, UCLA, 1977.
22. Guttag,J.V., Horowitz,E. and Musser,D.P. Abstract data types and software validation. C.ACM 21, 12 (1978), 1048-1063.
23. Musser,D.R. Abstract data type specification in the AFFIRM system. Proc. Specification of Reliable Software Conf., Boston, 1979.