THE RELATION BETWEEN LOGIC AND FUNCTIONAL LANGUAGES: A SURVEY*

MARCO BELLIA* AND GIORGIO LEVI

▷ The paper considers different methods of integrating the functional and logic programming paradigms, starting with the identification of their semantic differences. The main methods to extend functional programs with logic features (i.e. unification) are then considered. These include narrowing, completion, SLD-resolution of equational formulas, and set abstraction. The different techniques are analyzed from several viewpoints, including the ability to support both paradigms, lazy evaluation, and concurrency. ◁

1. WHY INTEGRATE LOGIC AND FUNCTIONAL LANGUAGES?

In the last two years several efforts have been devoted to the problem of integrating the most promising classes of declarative languages, i.e. the logic and the functional languages. Several different approaches have been proposed. The most relevant will be surveyed in the following. Some differences can also be found in the aims. We will then start by listing some of the arguments that have been used to support the combination of the two paradigms.

(1) *Notation.* Logic languages (based on definite Horn clauses) allow the definition of *relations*. Some problems can more naturally be described in terms of *functions*. A (syntactically) powerful language should then allow the definition (and the composition) of relations and functions.

(2) *Control.* Functional languages must certainly be considered declarative languages in contrast with standard imperative languages. However, a functional

Address correspondence to Giorgio Levi, Dipartimento di Informatica, Università di Pisa, I-56100 Pisa, Corso Italia 40.

*This work was partially supported by the European Community under ESPRIT Project 415.

*At same address as Giorgio Levi.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/86/$03.50

program contains much more control knowledge than the "corresponding" logic program (essentially related to function application). Such knowledge can be exploited by very efficient (deterministic) implementations. A functional language is then adequate to define *algorithms*. The same goal can only be achieved in logic programs by means of a complex (and sometimes semantically unclear) combination of declarative and control (or metalevel) knowledge.

(3) *Implementation techniques.* For some functional languages there exist powerful *compilation* techniques (e.g. translation to combinatorial logic) and *implementation* techniques (e.g. reduction machines). It is interesting to investigate the possibility of applying these techniques to logic languages or to functional languages extended with typical logic-language features.

(4) *Language features.* Functional languages offer a variety of powerful programming concepts (*higher-order functions*, *lazy evaluation and streams*, *types and polymorphism*), which are difficult to cast in the standard logic-programming framework, yet could be easier to understand for a functional-logic language.

(5) *Existing functional environments.* Some functional languages (e.g. LISP) have excellent *programming environments* that could be inherited by a logic component, implemented in the functional language, and somehow interfaced to it.

So far we have only considered some of the advantages of functional programming. Of course, logic-programming languages are much more powerful than standard functional languages. The most relevant aspect is their being *problem-solving languages*, halfway between theorem provers and standard programming languages (including the functional ones). This is why they are adequate to cope with artificial-intelligence and data-base applications and they are good candidates as programming languages for non-expert end users. In the next section we will consider the most relevant distinguishing features of logic languages. This will eventually allow us to better understand the difference between logic and functional languages.

2. THE UNIQUE FEATURES OF LOGIC LANGUAGES

A logic program is a set of definite Horn clauses. Theorems that can be proved contain existentially quantified variables only (i.e., they are queries). As a consequence there exists a complete refutation procedure (SLD resolution) which is essentially a (*nondeterministic*) *rewriting process*, and which computes answers by composing the most general unifiers. This is why, operationally, the theorem prover is essentially an interpreter: rewriting is the typical operational semantics of programming languages. The corresponding declarative semantics property is the existence of a standard model (the minimal Herbrand model), which can also be obtained as a least fixpoint [19].

Even if logic programs are similar to regular programs, there are some relevant differences. If we consider functional programs, we can mention the following

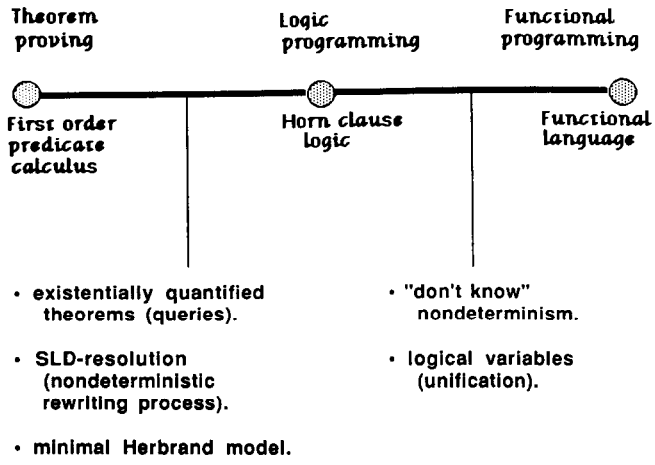

FIGURE 1. The distinguishing features of logic programming.

aspects:

(1) *Notation.* Relations versus functions.

(2) *Style.* Definition by separate clauses versus conditional expressions.

(3) *Control.* Flat "AND" composition versus application-induced partial ordering.

(4) *Determinism.* Search-based computation ("don't know" nondeterminism) versus deterministic computation.

(5) *Logical variables.* Unification versus parameter passing and value return.

(1) is essentially syntactic. The same is true for (2): several modern functional languages allow the same definition style and are syntactically very similar to logic languages. In the following, when discussing functional languages, we will mainly refer to such languages, which have also been called *pattern-matching languages*, and include first-order languages (such as TEL [41, 42], OBJ [25], and HOPE [12]), term-rewriting systems [34, 30], and some higher-order languages (such as SASL [57] and ML [28, 46]).

(3) is a semantic issue, related to the semantic difference between functions and relations. The real relevant aspects, however, as shown in Figure 1, are (4) and (5). Most of the technical solutions we will describe are based on one of the following ideas:

Add logical variables (and sometimes search) to a functional language to obtain a logic language with functional notation.

Constrain the behavior of logical variables (and sometimes of the search rule) of a logic language to obtain a "more functional" language.

In the following section we will consider some semantic properties mainly related to logical variables.

3. UNIFICATION AND LOGICAL VARIABLES

Unification is basically different from pattern matching in its being symmetric: variables occurring in the "procedure call" and variables occurring in the "procedure declaration" are handled exactly in the same way. This is the basis of some typical logic-programming features, such as invertibility of procedures and partially determined data structures. In the process interpretation of logic programs, where a goal is represented by a set of processes (the atoms) connected by channels (the variables), logical variables cause channels to be undirected. There exists no static producer-consumer relation among processes. In the case of pattern-matching functional languages, on the contrary, channels are directed: a functional process can only consume data on its input channels and produce data on its output channel. There exist other intermediate cases between pure unification (logical variables) and pure pattern matching (no logical variables), which typically have been found in the framework of concurrent logic languages.

3.1. From Logic Languages to Functional Languages

One problem in concurrent logic languages is process synchronization. One approach to synchronization consists in the introduction of *variable annotations* and/or in modifications of the basic unification algorithm. The proposed solutions can be grouped in two classes:

(1) *Static input-output mode declarations* and conditions which guarantee that each channel has one producer only. Examples are:

LCA [4–7],

the relational parallel language in [14],

the language in [20].

(2) *Dynamic variable annotations.* Examples are the AND-parallel component of PARLOG [15], Concurrent Prolog [54], and Guarded Horn Clauses [58], where

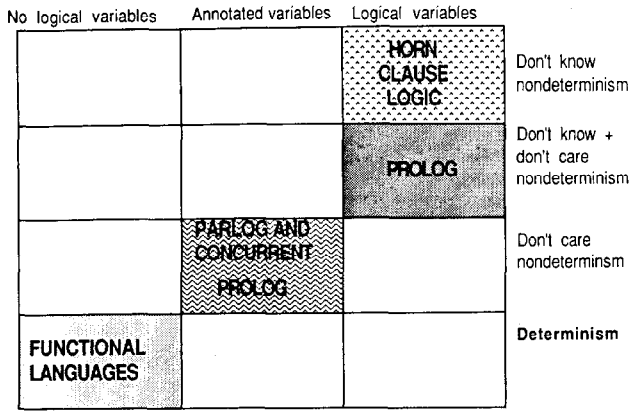

FIGURE 2. From logic to functional languages.

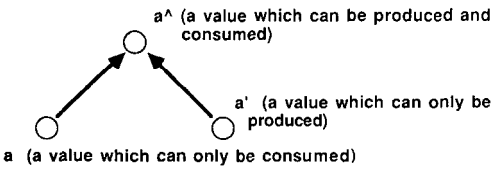

FIGURE 3. Annotated values.

all the variables in a process are sometimes handled by unification as if they were implicitly read-only.

Our interest here is mainly in the languages of class (1). The constraints on the logical variables cause the resulting language to be essentially a functional language. In fact, processes cannot write on input channels, and no multiple producers are allowed. The only differences from standard functional languages are that the language allows tuple-valued functions and (possibly) some sort of nondeterminism (don't care). Languages in the second class lie between pure logic programs and functional programs, as shown in Figure 2, which considers nondeterminism as well.

The essential difference between logical variables and standard or annotated variables should be recognizable in the model-theoretic (or declarative) semantics too. A promising direction was found in an attempt to define a declarative semantics for logic programs with read-only variables [43]. The idea is to define a Herbrand universe which contains annotated values. Values annotated by ^ are values which can both be produced and consumed (i.e., they are standard logic-program values). Values without annotation can only be consumed (and are related to read-only variables or input-mode declarations). Finally, values annotated by ' are values which can only be produced (and are related to output mode declarations or to read-only variables occurring in the clause head). Of course, the new Herbrand universe has a partial-ordering relation which is derived from the relation shown in Figure 3.

A functional program can then be naturally characterized on the minimal Herbrand model. For each atom in the denotation of predicate symbol P, each tuple of terms must contain either consumed values (i.e. `cons(a,cons(b,nil))`) or produced values (i.e. `cons'(a',cons'(b',nil'))`).

In the next section we will look into the symmetric problem, i.e. how to transform a functional language into a logic language, by introducing logical variables and unification.

3.2. Extending the Power of Functional Languages

There are a number of proposals for adding unification to pattern-matching functional languages, with the aim of partially achieving some typical logic-programming features: HASL [1], Uniform [37], and FGL + LV [44] are worth mentioning. Other languages were designed with the aim of defining logic languages with a functional syntax. The technical solutions will be considered in Sections 6 and 7, and we will now only provide an informal justification.

Let us consider a first-order equational language, whose programs are equations of the following form:

(1) $+(0, x) = x$,

(2) $+(s(x), y) = s(+(x, y))$,

(3) $*(0, x) = 0$,

(4) $*(s(x), y) = +(*(x, y), y)$.

The language operational semantics (rewriting, reduction) allows us to reduce a closed term (i.e. a term containing no variables) to its normal form. If some syntactic conditions are satisfied (e.g. if the left parts are not unifiable), term reduction is deterministic.

Adding logical variables essentially means that variables may occur in the term to be evaluated. Such variables correspond to existentially quantified variables. As a consequence, in the operational semantics (essentially a symbolic evaluation semantics), unification replaces pattern matching. Unification can bind variables occurring in the term, so that term reduction becomes nondeterministic, with different variable bindings corresponding to different paths.

In order to transform the symbolic interpreter into a logic-programming theorem prover, we only need to change the top-level task from term symbolic reduction to proving reducibility (or unifiability) of two terms, possibly containing (existentially quantified) variables.

Let us consider an example, using equations (1) and (2), where the "goal" to be proved is

$+(x,0) = s(0)$ (find an x such that...).

1. [By equation (1)] $x = 0$ and the new goal is $0 = s(0)$ (failure).
2. [By equation (2)] $x = s(x_1)$ and the new goal is $s(+(x_1,0)) = s(0)$, which, by unification becomes $+(x_1,0) = 0$.
   - 2.1. [By equation (1)] $x_1 = 0$ and the new goal is $0 = 0$ [successful termination with answer $x = s(0)$].
   - 2.2. [By equation (2)] $x_1 = s(x_2)$ and the new goal is $s(+(x_2,0)) = 0$ (failure).

The simple example shows that one relevant logic-programming feature related to logical variables, i.e. invertibility, can be achieved naturally. The same is true for partially determined data structures. The resulting language is then a logic language based on a functional notation. In Section 6 we will describe three different but equivalent techniques that are essentially based on the idea sketched above. A rather different approach will be considered in Section 7.

Before going into the technical aspects of logic languages based on functional notation, we will first consider some proposals for interfacing a logic-programming language to a separate functional language.

4. INTEGRATING SEPARATE LANGUAGES

Several systems have been built in the last few years with the aim of making available in a single environment a functional language and a logic language. Most of the systems are based on LISP or LISP dialects, the main motivations being the

existence of excellent programming environments and widespread programming expertise. Let us mention LOGLISP [51, 52] and QLOG [39], based on LISP; POPLOG [45], based on POP2; and the system in [55], based on SCHEME. The systems are quite similar. We will consider LOGLISP as a typical example of this class.

The logic language (Horn-clause logic) is implemented in LISP. Specific LISP functions are provided for program data-base updating and querying. A query may occur within any LISP expression. An example of a LISP expression which queries the logic program is the following:

```
(ALL(x1...xt)C1...Cn),
```

where `C1...Cn` is the LISP representation of the query, and `x1...xt` are the "answer" variables (occurring in the query). The function `ALL` returns the list of all solution tuples (which are lists, in turn). The result of a logic-program invocation is then a LISP data structure, which can then be processed by other LISP functions. It is also possible to call LISP programs from a logic program: Any LISP expression can occur in a query or in a clause body.

The resulting language can be viewed as a combination of language and metalanguage [10], where Horn-clause logic is the language and LISP is its metalanguage. A LISP function like `ALL` is similar to the `DEMO` predicate proposed in [10]. The combined language is therefore very powerful, since it allows one to extend (through suitable metalanguage functions) the inference rules of the language. However, moving from one language to the other requires rather complex communication interfaces. Moreover, LISP (i.e. the metalanguage) data structures are the only data which can be communicated between the two languages. This forces unnecessary constraints on logic data structures or complex data conversions, which would not be required if the two languages were closer from the viewpoint of data structures. This is the case of the language LEAF [2, 3], where a logic language is interfaced to the functional language LCA [4–7], which is essentially a functional language obtained by removing logical variables and nondeterminism from the logic language (see Section 3.1). The two language components have then exactly the same data structures (logical terms). A similar situation can be found in the interface between the two language components (And-parallel and Or-parallel) of PARLOG [15].

The main drawback of this approach is that the two language components are necessarily separated. This makes the semantics of the combination rather complex. The approaches that will be considered in the next sections aim at the definition of a single functional-logic language.

5. LOGIC PROGRAMMING WITH EQUALITY

There has always been a lot of interest in extending resolution theorem proving to theories with equality. One example of an inference system is paramodulation. Equality has been added to some logic-programming languages as well (see, for example, [40] and a nice semantic characterization of complete logic programs with equality in [36]). Of course, through the equality relation, the functional notation comes in. Function symbols are no longer restricted to denote data constructors (for which equality is the same as identity). The real problem with equality is the complexity of the refutation procedure. It seems that some constraints on equality are needed to achieve the computational properties of logic programs. This can

better be done in the framework of equational theories [34, 30], which, under suitable conditions, are real functional-programming languages. In the next section we will consider equational theories as logic languages with functional syntax. In Section 8 we will look into the problem of how they can be combined with logic languages with the relational syntax.

6. LOGIC LANGUAGES WITH FUNCTIONAL SYNTAX (EQUATIONAL THEORIES)

In this section we will consider the technical foundations of the solution sketched in Section 3.2, namely, how to transform a functional equational language into a logic language.

The operational semantics of equational languages, like the semantics of any functional language, can be defined in terms of reductions (or rewritings). If we allow existentially quantified variables and unification, we have a corresponding operational semantics, which is based on *narrowing*.

6.1. Narrowing

Narrowing [21, 35] a functional expression is applying to it the minimum substitution such that the resulting expression is reducible, and then reducing it. The substitution is found by unifying the expression with the left-hand sides of equations. In general there will be several narrowings for an expression, one for each equation whose left-hand side is unifiable with the expression. One specific narrowing does not preserve the semantics of an expression. However, the set of all narrowings does.

Any expression (or an equation) containing existentially quantified variables (logical variables) can be narrowed, possibly yielding a set of narrowing substitutions (answers). This is essentially the same situation of query evaluation in logic programs. The simple example of Section 3.2 was exactly an example of narrowing.

It is worth noting that narrowing can be efficiently implemented as a proper extension of reduction. Namely, if we apply narrowing to a closed expression, containing no existentially quantified variables, we obtain exactly the same sequence of rewritings that we would obtain through reduction.

Narrowing is complete in the case of equational theories which have the finite termination property and are confluent. If we consider a specific class of equational theories, i.e. equational theories with constructors, narrowing [23] is complete even if the finite termination property does not hold [48]. Note that equational theories with constructors, where one explicitly distinguishes between data constructors and functions, can more naturally be viewed as programming languages and have a minimal Herbrand-model semantics [42], similar to that of logic languages [19].

Recently, there have been several proposals to use equational languages with narrowing as logic programming languages. Let us mention:

EQLOG [26, 27], which combines a relational Horn-clause language with confluent and terminating equational theories (without constructors).

FUNLOG [56], whose semantic unification is essentially narrowing, even if no completeness proof is given.

The language proposed by Reddy [48,49], based on equational theories with constructors.

The language proposed in [18], which uses conditional equations. Other similar proposals are Qute [53] and the language in [50].

There exist other inference methods which are essentially equivalent to narrowing. We will consider these methods in the next two subsections.

6.2. Completion

The *Knuth-Bendix completion algorithm* [38] was originally introduced as a procedure to derive canonical term-rewriting systems from equational theories. In the late 70s, several authors found that the algorithm could be used as a technique for proving the validity of equations in the inductive theories defined by a wide class of equational theories. The technique is complete (and sound) for the class of sort-separable equational theories [34], i.e. theories made complete by considering different those ground terms which cannot be proved to be equal in the theory. Under this assumption, the validity of an equation in a theory is equivalent to the consistency of the theory augmented with such an equation. The technique has widely been used to prove inductive theorems, without explicitly using the induction inference rule. In the general case, the completion procedure is computationally rather complex and expensive (the complexity measure is the number of generated critical pairs). If the equational theory can be represented by a canonical term-rewriting system, the critical pairs depend upon the superpositions of the theory equations with the equation to be proved. In such a case, the complexity can be dramatically reduced.

Recently, completion has been applied to predicate-calculus theorem proving [31]. The first step was showing that there exists a canonical term-rewriting system for the propositional calculus. The validity of a propositional-calculus formula can then be proved by reducing the corresponding term. The second step was to show that predicate calculus can be represented as a (sort-separable) equational theory. The completion algorithm can then be used as a first-order-theories theorem prover. There exist some simplifications of the method (N-completion), which allow one to reduce the number of critical pairs. It is worthwhile to look at one example, where the theory is a set of definite Horn clauses and the theorem to be proved is a typical logic-programming query.

The set of Horn clauses

(1) `Times(0,x,0).`

(2) `Times(s(x),y,z):-Times(x,y,u), Plus(u,y,z)`

(3) `Plus(0,x,x).`

(4) `Plus(s(x),y,s(z)):-Plus(x,y,z)`

is represented by the following set of rewrite rules:

1. `Times(0,x,0)+1-->∅`
2. `Times(s(x),y,z)*Times(x,y,u)*Plus(u,y,z)+`
   `Times(x,y,u)*Plus(u,y,z)-->∅`

3. `Plus(0,x,x)+1-->∅`
4. `Plus(s(x),y,s(z))*Plus(x,y,z)+Plus(x,y,z)-->∅.`

(The operators +, *, ∅, and 1 are introduced for representation purposes.) If we now consider the theorem to be proved

5. `Times(s(0),s(0),w)-->∅`

(corresponding to the query `:-Times(s(0),s(0),w)`), the proof is the following:

5. `Times(s(0),s(0),w)-->∅`
   N-completion on 2 (first term) and 5
   Most general term
   `Times(s(0,s(0),w)*Times(0,s(0),u)*Plus(u,s(0),w)+`
   `Times(0,s(0),u)*Plus(u,s(0),w).`

by 2: `∅*Times(0,s(0),u)*Plus(u,s(0),w)+`
`Times(0,s(0),u)*Plus(u,s(0),w)`
by 5: `∅`
`<∅*Times(0,s(0),u)*Plus(u,s(0),w)+`
`Times(0,s(0),u)*Plus(u,s(0),w),∅>`
by simplification

6. `Times(0,s(0),u)*Plus(u,s(0),w)-->∅`
   N-completion on 1 and 6
   Most general term
   `(Times(0,s(0),0)+1)*Plus(0,s(0),w).`

by 1 [rewritten as 1'. `(Times(0,s(0),0)+1)*v-->∅`]: `∅.`
by 6: [properties of + and *]
`∅+1*Plus(0,s(0),w)`
`<∅+1*Plus(0,s(0),w),∅>`
by simplification

7. `Plus(0,s(0),w)-->∅`
   N-completion on 3 and 7
   Most general term
   `Plus(0,s(0),w)+1`

by 3: `∅.`
by 7: `∅+1`
`<∅+1,∅>`
by simplification

8. `1-->∅` (contradiction).

It is worth noting that the number of inferences is the same we would have with SLD-resolution of Horn clauses. However, the unification is rather more complex, because the operators + and * are both associative and commutative. Moreover, from a programming viewpoint, there is a need of further complications to obtain the answers.

A variant of the method has been proposed by Paul [47], which shows that rewrite methods on confluent rewriting systems can be applied to first-order-logic

theorem proving, not only as a refutational proof technique but for satisfiable theories as well.

The method has been refined and specialized to logic programming in [17], where theories are restricted to definite Horn clauses. The resulting completion algorithm is quite efficient and does not require extensions to the unification algorithm, even if there is still the need for a special predicate to collect the answers.

The completion approach is based on representing Horn clauses by means of equational theories. Therefore it is not a method for introducing logic-programming features in an equational language. Completion, however, has a strong relation to narrowing, which is very similar to linear completion. Informally, narrowing is related to completion as linear resolution is related to general resolution. Any form of completion more general than narrowing, therefore, even if interesting from a theorem-proving viewpoint, seems to be not adequate to logic programming.

Another proposal related to the completion approach is SLOG [22, 24], which is a Horn-clause language with the only predicate symbol "=" (essentially similar to the canonical representation of functional equations in LEAF [3] which will be described in the next section). The other predicates are viewed as boolean functions. Clausal superposition instead of resolution is used as the inference rule. Clausal superposition is an extension of the rule used in the Knuth-Bendix completion procedure. It is complete for first-order logic with equality, but it is rather time and space consuming compared to resolution. A specific efficient strategy (called inner superposition) [24] has been proposed, which is complete for a wide class of programs. The resulting inference system is once again very similar to narrowing.

6.3. *Transformation to Clause Form and* SLD*-resolution*

Equational theories with constructors, as we have already mentioned, have some interesting properties, which we will now briefly summarize.

(1) The distinction between functions and data constructors allows us to characterize theories which essentially define *functional programs*, ruling out equations that can be viewed as *program properties*. For example, if `nil` and `cons` are considered data constructors, the equations

```
append(nil,x)=x
append(cons(x,y),z)=cons(x,append(y,z)),
```

such that the left parts contain only one (outermost) function symbol, are function definitions, while an equation like

```
append(append(x,y),z)=append(x,append(y,z))
```

would be considered a program property and would not be allowed as program component.

(2) With the above assumption, programs have a *minimal Herbrand model semantics* [42] similar to that of definite Horn clauses (where the Herbrand universe is defined by constants and data constructors only). If the Church-Rosser property is guaranteed by suitable syntactic constraints, the minimal model defines a set of (deterministic) functions.

(3) *Narrowing is complete* as the operational semantics of a logic language based on equations.

Narrowing, however, is not the only inference rule which can be used for logic languages based on equational theories with constructors. A different approach, suggested in the language LEAF [3], is based on the transformation of the equations into Horn clauses and on an SLD-resolution inference rule. This approach, as we will see in Section 8, is relevant to the definition of logic languages which allow one to define both relations and functions.

A set of equations is first *transformed to canonical form*, which is a "flat" clause form where function composition is eliminated and replaced by the logical operator AND. For example, the canonical form of the equation

```
*(s(x),y)=+(*(x,y),y)
```

is the clause

```
*(s(x),y)=z:-*(x,y)=w,+(w,y)=z.
```

The same transformation applies to the equation to be proved, which is transformed into a goal clause, consisting of a conjunction of function composition free atoms, all containing the predicate symbol =. For example, the "goal" equation

```
+(*(s(0),x),0)=s(s(0))
```

is transformed into the goal clause

```
:-*(s(0),x)=y,+(y,0)=s(s(0)).
```

Equations in canonical form are then similar to SLOG clauses [22, 24]. In LEAF, however, they are handled using the standard Horn-clause logic inference rule, i.e. SLD-resolution. = is considered as a predicate symbol without any specific property, i.e., an atom in the goal (whose predicate symbol is =) is unified (using standard unification) with a clause head. SLD-resolution has been shown to be complete [42] for proving existentially quantified equations in equational theories with constructors. Note that the equational theory is required to have the Church-Rosser property and is not required to have the finite-termination property. In conclusion, SLD-resolution on the canonical form is semantically equivalent to narrowing in equational theories with constructors [48].

As we mentioned in Section 1, a functional program contains much more control information than the equivalent relational program. The partial ordering corresponding to nested function applications is lost in a "flat" relational representation. Such information is explicitly used by narrowing, which is therefore more efficient than blind SLD-resolution. The same information can usefully be exploited also with the relational representation if the original partial ordering is represented as control information and if this information can be used to drive the SLD-resolution interpreter. The solution proposed in LEAF is based on a producer-consumer variable annotation in clause bodies and goal clauses. For example, the annotated clause corresponding to the equation

```
*(s(x),y)=+(*(x,y),y)
```

is

```
*(s(x),y)=z:-*(x,y)=w,+(?w,y)=z,
```

while the annotated goal clause corresponding to the equation

```
+(*(s(0),x),0)=+(x,s(s(0)))
```

is

```
:-*(s(0),x)=y,+(?y,0)=z,+(x,s(s(0)))=z.
```

A variable annotation (denoted by the prefix functor ?) in a atom means that the atom should preferably read a value for that variable, produced by other atoms. If the variable is not bound and the unification requires that variable to be instantiated, the rewriting of the atom is suspended. This annotation, similar to Concurrent Prolog read-only variables [54], was shown to define a complete evaluation rule for SLD-resolution, provided that the annotation is ignored in some cases (when there are no other atoms which can compute that variable, or when there is a loop of suspended atoms in a deadlock-like situation) [43]. A similar complete producer-consumer annotation was first proposed in the language in [29], which is a superset of Horn-clause logic (with equality) whose inference system is based on natural deduction techniques.

SLD-resolution driven by annotated canonical clauses is fully equivalent to narrowing even from the operational viewpoint. As is the case for narrowing, when the functional language is used for reducing terms to their normal form (and not to prove existentially quantified equations), SLD-resolution is essentially standard reduction. Namely, the refutation does not require any searching (i.e., it is deterministic), and unification reduces to pattern matching. The standard functional language is then a special case of the logic language (with functional syntax), also from the implementation viewpoint.

7. SET ABSTRACTION AND UNIFICATION

In this section we will consider a completely different approach to the problem of defining a logic language based on a functional language. The idea, originally proposed in SUPERLOGLISP [8] and later advocated in [16], is that relational logic programming can be achieved by using a functional language augmented with set abstraction and unification. Let us first consider Robinson's approach [8].

The starting point has to be found in the LOGLISP [51,52] experience. In that system, the logic component is viewed through a function returning all the answers to the query (similar to the metalevel "all solutions" predicates in PROLOG). Conceptually, a predicate can then be viewed as a (deterministic) *set-valued function*, returning the set of all the relation tuples. For example, the Horn clause

```
append(cons(x,y),z,cons(x,w)):-append(y,z,w)
```

can be represented by the equation

```
append(x1,x2,x3)={x1,x2,x3|x1=cons(x,y) and x2=z and
                  x3=cons(x,w) and append(y,z,w)}.
```

It is worth noting that variables `x`, `y`, `z`, and `w` are *existentially quantified* (logical variables). The right-hand side of the equation must be read "the set of all triples `<x1,x2,x3>` such that there exist `x,y,z,w` which make the conjunction hold."

The second idea is that of grouping all the different definitions of the relation into a single clause. This requires the use of the **or** operator (*union* on sets). In the example, if the only other clause for `append` is

```
append(nil,x,x).,
```

the complete function definition is

```
append(x1,x2,x3)={x1,x2,x3|(x1=nil and x2=x and x3=x) or
                           (x1=cons(x,y) and x2=z and
                            x3=cons(x,w) and append(y,z,w))}.
```

The functional definition is equivalent to the set of clauses only under the closed-world assumption [13], which allows one to express the two clauses by the single non-Horn clause

```
append(x1,x2,x3) if and only if (x1=nil and x2=x and x3=x) or
                                (x1=cons(x,y) and x2=z and
                                 x3=cons(x,w) and append(y,z,w)).
```

The functional expression corresponding to the goal

```
:-append(x,y,cons(a,cons(b,nil)))
```

is the set expression

```
{x,y|append(x,y,cons(a,cons(b,nil)))}.
```

Two aspects are worth mentioning:

(1) Nondeterminism is replaced by set union.

(2) The evaluation requires the solution of conjunctive conditions, which include equations involving existentially quantified variables. The equations are the explicit representation of the unification in the original clauses.

The aim of the proposal is to use an efficient reduction machine to execute logic programs. Of course, new reduction rules are needed to cope with existentially quantified variables and unification. An interesting proposal is *epsilon-reduction* [9], which consists of two reduction rules, decomposition and specification. The specification rule eliminates an existentially quantified variable by substituting for it an expression to which it has been found equal. The decomposition rule corresponds to term unification and may produce new equations between variables and terms. A simulation of epsilon-reduction has been implemented in the RED1 machine. Some problems, mainly related to the `or`-connective, have no satisfactory solution yet.

Darlington's approach [16] can be viewed as a combination of narrowing and set abstraction. Standard functions (defined by equations) and set-valued functions can be combined. Deterministic set-valued functions are used, as in SuperLOGLISP, to represent nondeterministic relations. Let us consider for example the function `append` defined by the standard equations:

```
append(nil,x)=x
append(cons(x,y),z)=cons(x,append(y,z)).
```

If we want to use `append` as a relation, to find all ways of decomposing a given list into two lists, we define a suitable set-valued function as follows:

```
split(x)={y,z|append(y,z)=x}.
```

For evaluating an application of the function `split(a)`, the interpreter must be able to solve equations (such as `append(y,z)=a`), involving existentially quantified variables. Note that in SuperLOGLISP the conjuncts in a set expression are

either (recursive) function calls or equations between variables and terms (corresponding to the unification). Darlington's equations are more complex. The technical solution is essentially narrowing, with nondeterminism replaced (by the interpreter) with set union.

One aspect that is worth mentioning is that there must exist a specific function for each possible way of calling a relation, while in the case of SUPERLOGLISP this is done by evaluating a set expression. This can be rather heavy from the programmer viewpoint, but allows program optimization. In fact, since the program is always activated by a function call, the input-output modes of the variables occurring in the equations to be solved are statically known.

Both languages are based on a *higher-order functional language*. The relevance of higher-order features in logic languages will be discussed in Section 10. It is one of the most attractive features of some functional languages and should be preserved when the language is extended with logic-programming features. There are, however, technical problems that have still to be solved, e.g. higher-order unification and higher-order narrowing.

8. INTEGRATING RELATIONAL AND FUNCTIONAL LOGIC LANGUAGES

All the languages mentioned in Sections 6 and 7 are logic languages which allow us to define functions. Some of them allow us also to combine functions and relations in a single coherent linguistic framework. The different proposals can be grouped in four classes:

(1) *Functions only.* Relations must be represented as boolean functions. This class includes

Dershowitz's conditional equations [18], based on conditional narrowing,

SLOG [22, 24], based on equational clauses and inner superposition,

Reddy's equational theories with constructors [48], based on narrowing.

All the languages in this class are based on a single notation and a single inference system. The main drawback is the lack of an adequate notation for relations. If functions are useful in relational logic programming, relations are essential to logic programming.

(2) *Equations (functions) and clauses (relations), with translation of clauses into equations.* These are typical of the completion approach [17], which, however, leads to rather complex inference systems.

(3) *Functions and set-valued functions to denote relations.* This is the case of SUPERLOGLISP [8, 9] and of Darlington's language [16]. The approach seems very promising. The inference systems (epsilon-reduction or narrowing combined with set-oriented operations) have still to be compared with SLD-resolution from the performance viewpoint.

(4) *Functions and relations.* The languages in this class allow one to combine functional equations and relational Horn clauses. In all the languages functional expressions can occur in relational atoms. Most of the languages are based on two separate inference systems. The main languages in this class

are:

EQLOG [26, 27], based on Horn clauses and confluent and finitely terminating equational theories. The inference systems are resolution and narrowing.

FUNLOG [56], essentially similar to EQLOG, with semantic unification instead of narrowing.

The declarative component of LEAF [2, 3], where equations are translated into an annotated clause form and the inference system is SLD-resolution only. The language allows one to use relational atoms within function definitions. This is achieved through conditional equations. Consider the example

```
union(cons(x,y),z)=union(y,z):-member(x,z),
```

where `union` is a function and `member` is a relation. The canonical form of the conditional equation is

```
union(cons(x,y),z)=w:-member(x,z),union(y,z)=w.
```

The natural deduction logic language in [29] is similar to LEAF.

9. LAZY EVALUATION, INFINITE DATA STRUCTURES AND CONCURRENCY

Lazy evaluation has been shown to be very useful in functional languages to define an efficient external evaluation rule and to handle infinite data structures. Functional concurrent processes are also essentially based on lazy computational models.

It is very hard to extend lazy evaluation techniques to relational logic programming. This would require the existence of a partial ordering relation on the atoms, similar to the one which is implicit in the nesting of applications in a functional program. On the other hand, lazy evaluation is highly desirable, mainly in a functional-relational logic language, in order to define nonstrict functions (and relations) and for stream-based logic programming.

Logic languages with functional syntax, as already mentioned, have the partial ordering control information of functional languages. Lazy evaluation can be obtained by a suitable computation strategy which exploits this information. Examples of logic languages featuring lazy evaluation (in the functional component) are:

The language proposed by Reddy [48], which defines a special version of narrowing (lazy narrowing) and proves its completeness.

FUNLOG [56], which is based on a demand-driven computational model.

The language in [18], which uses conditional narrowing, similar in effect to lazy narrowing.

Darlington's language [16], which uses a strategy similar to lazy narrowing.

SuperLOGLISP [9], which does not actually have lazy evaluation. However, a lazy version of epsilon-reduction is being studied.

LEAF [2, 3], which has lazy evaluation of relations too. The partial ordering is defined by annotating variables also in the relational atoms. The semantics of

annotations is that described for annotations automatically generated for functional equations and is complete.

The language in [29], which is similar to LEAF.

Functions and relations acting on infinite data structures (streams) can be nonterminating. This rules out all those logic languages which are based on (or reduced to) equational theories having the finite-termination property (for example, EQLOG [26, 27]). This is one more argument in support of equational theories with constructors, where narrowing completeness does not require such a property.

In the case of relational logic languages, nonterminating relations (sometimes called perpetual processes) require specific inference rules to stop the computation when a sufficiently approximated answer has been computed. Both LEAF [2, 3] and the language in [29] feature such a rule.

The issue of control information is related also to the problem of defining parallel implementations of logic-functional languages. The presence of a functional component does not affect OR-parallelism. Note, however, that in languages based on set abstraction [8, 9, 16], nondeterminism is replaced by union on sets, where the various disjuncts can be evaluated concurrently. A depth-first-like solution can also be easily realized by using a "lazy" union operator.

AND-parallelism is strongly affected by the functional component. In fact, the intrinsic partial ordering of functional programs provides the basic process synchronization mechanism. It has been noted [43, 48] that parallel Horn-clause languages [15, 54] are forced to introduce annotations on logical variables to control the order of evaluation of processes. These annotations make the semantics of logic programs much more complex (see, for example, [43]). In the case of logic languages with functional notation, the order of evaluation can be controlled without restricting the use of logical variables and without affecting the language declarative semantics. Using the control information in AND-parallel computational models has been considered in various logic-functional languages, namely in Darlington's language [16], in Reddy's language [48] (where parallel narrowing is considered), and in LEAF [3]. In LEAF the mechanism of annotations allows one also to control the order of evaluation of relational processes, without any constraint on the behavior of logical variables and without affecting the declarative semantics of the language. It is therefore possible to define an efficient AND-parallel resolution based interpreter for a logic language featuring both relations and functions.

10. HIGHER-ORDER FEATURES

As already mentioned, higher-order functions are one of the most appealing features of functional programming and should be preserved in an extension of functional programming with logic-programming features. In the case of languages based on lazy evaluation, it is possible to handle function-valued expressions, binding them to variables and applying them on arguments [48], using narrowing instead of reduction. A complete implementation of narrowing for a higher-order functional language should also be able to solve equations with existentially quantified variables ranging over functions. This, in turn, requires a higher-order unification algorithm. Higher-order unification is theoretically well understood [32, 33]. However, computationally feasible algorithms do not exist for the general case, and satisfactory

solutions can be reasonably expected only for specific subclasses of higher-order functional programs. This is therefore still an open research problem.

Let us finally remark that higher-order features could play a very important role in logic programs, in addition to what is relevant to programming in general. In fact, they could provide a mathematically clean solution to the need for manipulating programs as data. In the case of first-order logic programming, this need has led to the language-metalanguage amalgamation proposal [10, 11], which is less powerful than higher-order logic and sometimes rather difficult to understand.

REFERENCES

1. Abramson, H., A Prological Definition of HASL, a Purely Functional Language with Unification Based Conditional Binding Expressions, *New Generation Comput.* 2:3–35 (1984).
2. Barbuti, R., Bellia, M., Levi, G., and Martelli, M., On the integration of logic programming and functional programming, in: *Proceedings of the 1984 International Symposium on Logic Programming*, IEEE Computer Soc. Press, 1984, pp. 160–166.
3. Barbuti, R., Bellia, M., Levi, G., and Martelli, M., LEAF: A Language Which Integrates Logic, Equations and Functions, in: D. DeGroot and G. Lindstrom (eds.), *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1985.
4. Bellia, M., Degano, P., and Levi, G., A Functional Plus Predicate Logic Programming Language, in: *Proceedings of the Logic Programming Workshop*, 1980, pp. 334–347.
5. Bellia, M., Degano, P., and Levi, G., The Call by Name Semantics of a Clause Language with Functions, in: K. L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 281–295.
6. Bellia, M., Dameri, E., Degano, P., Levi, G., and Martelli, M., Applicative Communicating Processes in First Order Logic, in: *Proceedings of the 5th International Symposium on Programming, LNCS 137*, Springer, 1982, pp. 1–14.
7. Bellia, M., Dameri, E., Degano, P., Levi, G., and Martelli, M., A Formal Model for Lazy Implementation of a PROLOG Compatible Functional Language, in: J. A. Campbell (ed.), *Implementations of* PROLOG, Ellis Horwood, 1984, pp. 309–326.
8. Berkling, K., Robinson, J. A., and Sibert E. E., A Proposal for a Fifth Generation Logic and Functional Programming System, Based on Highly Parallel Reduction Machine Architecture, Syracuse Univ., Nov. 1982.
9. Berkling, K., Epsilon-reduction: Another View of Unification, CASE Center, Syracuse Univ., 1985.
10. Bowen, K. A., and Kowalski, R. A., Amalgamating Language and Metalanguage in Logic Programming, in: K. L. Clark and S.-A. Tarnlund, (eds.), *Logic Programming*, Academic, 1982, pp. 153–172.
11. Bowen, K. A., and Weinberg, T., A Meta-level Extension of Prolog, in: *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Soc. Press, 1985, pp. 48–53.
12. Burstall, R. M., MacQueen, D. B., and Sannella, D. T., HOPE: An Experimental Applicative Language, in: *Conference Record of the 1980 LISP Conference*, 1980, pp. 136–143.
13. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum, 1978, pp. 292–322.
14. Clark, K. L., and Gregory, S., A Relational Language for Parallel Programming, in: *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1981, pp. 171–178.
15. Clark, K. L., and Gregory, S., PARLOG: A Parallel Logic Programming Language, Imperial College Research Report 83/5, May 1983.

16. Darlington, J., Field, A. J., and Pull, H., The Unification of Functional and Logic Languages, in: D. DeGroot and G. Lindstrom (eds.), *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1985.
17. Dershowitz, N., and Josephson, N. A., Logic Programming by Completion, in: *Proceedings of the 2nd International Logic Programming Conference*, 1984, pp. 313–320.
18. Dershowitz, N., and Plaisted, D. A., Logic Programming cum Applicative Programming, in: *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Soc. Press, 1985, pp. 54–66.
19. van Emden, M. H., and Kowalski, R. A., The Semantics of Predicate Logic as a Programming Language, *J. Assoc. Comput. Mach.* 23:733–742 (1976).
20. van Emden, M. H., and de Lucena Filho, G. T., Predicate Logic as a Language for Parallel Programming, in: K. L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 189–198.
21. Fay, M., First Order Unification in an Equational Theory, in: *Proceedings of the 4th Workshop on Automated Deduction*, 1979, pp. 161–167.
22. Fribourg, L., Oriented Equational Clauses as a Programming Language, in: *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, 1984, pp. 162–173; *J. Logic Programm.* 1:165–177 (1984).
23. Fribourg, L., A Narrowing Procedure for Theories with Constructors, in: *Proceedings of the 7th International Conference on Automated Deduction, LNCS 170*, Springer, 1984, pp. 259–301.
24. Fribourg, L., SLOG: A Logic Programming, Language Interpreter Based on Clausal Superposition and Rewriting, in: *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Soc. Press, 1985, pp. 172–184.
25. Goguen, J. A., and Tardo, J. J., An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications, in *Proceedings of the IEEE Conference on Specifications of Reliable Software*, 1979, pp. 179–189.
26. Goguen, J. A., and Meseguer, J., Equality, Types, Modules and Generics for Logic Programming, in: *Proceedings of the 2nd International Logic Programming Conference*, 1984, pp. 115–125.
27. Goguen, J. A., and Meseguer, J., Equality, Types, Modules and (Why Not?) Generics for Logic Programming, *J. Logic Programm.* 1:179–210 (1984).
28. Gordon, M., Milner, R., and Wadsworth, C., in: *Edinburgh LCF. LNCS 78*, Springer, 1979.
29. Hansson, A., Haridi, S., and Tarnlund, S.-A., Properties of a Logic Programming Language, in: K. L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 267–280.
30. Hoffmann, C. M., and O'Donnell, M. J., Programming with Equations, *Trans. Programm. Languages and Systems* 4:83–112 (1982).
31. Hsiang, J., and Dershowitz, N., Rewrite Methods for Clausal and Non-clausal Theorem Proving, in: *Proceedings of the 10th ICALP*, 1983.
32. Huet, G., Constrained Resolution: A Complete Method for Higher Order Logic, Dept. of Computer and Information Sciences, Case Western Reserve Univ., 1972.
33. Huet, G., a Unification Algorithm for Typed λ-calculus, *Theoret. Comput. Sci.* 1:27–57 (1975).
34. Huet, G., and Oppen, D. C., Equations and Rewrite Rules: A Survey, in: R. Book (ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic, 1980, pp. 349–405.
35. Hullot, J.-M., Canonical Forms and Unification, in: *Proceedings of the 5th Conference on Automated Deduction, LNCS 87*, Springer, 1980, pp. 318–334.
36. Jaffar, J., Lassez, J.-L., and Maher, M. J., A Theory of Complete Logic Programs with Equality, *J. Logic Programm.* 1:211–223 (1984).
37. Kahn, K. M., Uniform: A Language Based Upon Unification Which Unifies Much of Lisp, Prolog and Act1, in: *Proceedings of the 7th IJCAI*, 1981.

38. Knuth, D. E., and Bendix, P. B., Simple Word Problems in Universal Algebras, in: J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergamon, 1970, pp. 263–297.
39. Komorowski, H. J., QLOG—the Programming Environment for Prolog in Lisp, in: K. L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 315–322.
40. Kornfeld, W. A., Equality for PROLOG, in: *Proceedings of the 8th IJCAI*, 1983, pp. 514–519.
41. Levi, G., and Sirovich, F., Proving Program Properties, Symbolic Evaluation and Logical Procedural Semantics, in: *Proceedings of MFCS 75*, *LNCS*, Springer, 1975, pp. 294–301.
42. Levi, G., and Pegna, A., Top-Down Mathematical Semantics and Symbolic Execution, *RAIRO Inform. Théor.* 17:55–70 (1983).
43. Levi, G., and Palamidessi, C., The Declarative Semantics of Logical Read-Only Variables, in: *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Soc. Press, 1985, pp. 128–137.
44. Lindstrom, G., Functional Programming and the Logical Variable, in: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, 1985.
45. Mellish, C., and Hardy, S., Integrating PROLOG in the POPLOG Environment, in: J. A. Campbell (ed.), *Implementations of* PROLOG, Ellis Horwood, 1984, pp. 147–162.
46. Milner, R., A Proposal for Standard ML, in: *ACM Symposium on* LISP *and Functional Programming*, 1984, pp. 184–197.
47. Paul, E., A New Interpretation of the Resolution Principle, in: *Proceedings of the 7th International Conference on Automated Deduction*, *LNCS 170*, Springer, 1984, pp. 333–355.
48. Reddy, U. S., Narrowing as the Operational Semantics of Functional Languages, in: *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Soc. Press, 1985, pp. 138–151.
49. Reddy, U. S., On the Relationship between Logic and Functional Languages, in: D. DeGroot and G. Lindstrom (eds.), *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1985.
50. Rety, P., Kirchner, C., Kirchner, H., and Lescanne, P., NARROWER: A New Algorithm for Unification and Its Application to Logic Programming, in: *Proceedings of the First International Conference on Rewriting Techniques and Applications*, 1985.
51. Robinson, J. A., and Sibert, E. E., LOGLISP: Motivations, Design and Implementation, in: K. L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 299–314.
52. Robinson, J. A., and Sibert, E. E., LOGLISP: An Alternative to PROLOG, in: *Machine Intelligence 10*, Ellis Horwood, 1982.
53. Sato, M., and Sakurai, T., Qute: A Functional Language Based on Unification, in: *Proceedings of FGCS'84*, 1984, pp. 157–165.
54. Shapiro, E. Y., A Subset of Concurrent Prolog and Its Interpreter, Techn. Rep. TR-003, ICOT, 1983.
55. Srivastava, A., Oxley, D., and Srivastava, A., An(other) Integration of Logic and Functional Programming, in: *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Soc. Press, 1985, pp. 254–260.
56. Subrahmanyam, P. A., and You, J.-H., FUNLOG = Functions + Logic: A Computational Model Integrating Functional and Logic Programming, in: *Proceedings of the 1984 International Symposium on Logic Programming*, IEEE Computer Soc. Press, 1984, pp. 144–153.
57. Turner, D. A., SASL Language Manual, Dept. of Computational Science, Univ. of St. Andrews, 1979.
58. Ueda, K., Guarded Horn Clauses, ICOT Tech. Report TR-103, 1985.