J. Symbolic Computation (1986) **2**, 123–138

Modelling the Combination of Functional and Logic Programming Languages

U. FURBACH AND S. HÖLLDOBLER

Universität der Bundeswehr, München, F.R.G.

(*Received* 14 *January* 1985, *and in revised form* 7 *January* 1986)

The combination of functional and pure Horn clause logic languages is formally introduced. To give a framework for the investigation of implementations we define a complete and consistent model, which retains full invertibility and allows separation of logic and control. Some existing implementations are discussed from this viewpoint. An extended unification algorithm is suggested, which incorporates the features demanded by our model.

Introduction

The rapid success of logic programming has led to numerous proposals for increasing the efficiency of logic programming systems. In many cases these systems are enriched with control mechanisms, thus putting the burden of controlling the evaluation of programs on the programmer (e.g. Clark *et al.*, 1982). However, we see a real advantage of logic programming in the possibility of preserving the principle of separation of logic from control.

One way to bring together the demand for efficiency and for logic without control mechanisms is the combination of a functional and a logic programming language (e.g. Robinson & Sibert, 1982; Sato & Sakurai, 1983). Another advantage of such a combination is that both functional and logic programming styles can be used within one system. In this paper we define a semantic model for the combination of a functional and a logic language. This aims at a theoretical framework for both, the specification of such a combination, and the discussion of the correctness of implementations.

For our purpose there is no need to define a functional language completely or to rely on an existing language. We only have to specify sufficient conditions to provide a framework for our semantic model. Concerning the logic programming language we assume the general case of pure Horn clause programs. In particular, we do not include any control mechanisms. Since we see control mechanisms mainly as a tool for increasing efficiency of logic programs we expect that in a combination of functional and logic programming this can be done by stressing the functional part of a program.

In section 1 we define FHCL (functional and Horn clause logic language) as a class of languages which combines functional and logic programming. In section 2 we give a model of an FHCL program which we use in section 3 as a framework for the discussion of implementations. In section 4 we propose an implementation based on an extended unification algorithm, which incorporates the features demanded by our model, and prove its soundness. Section 5 contains an overview of related work.

We assume the reader to be familiar with the usual notations and basic results of logic

0747–7171/86/020123+16 $03.00/0 © 1986 Academic Press Inc. (London) Ltd.

programming and theorem proving (see, e.g. van Emden & Kowalski, 1976; Chang & Lee, 1973).

1. Combining Functional and Logic Languages

Let HCL (*Horn clause logic*) be the language of a first order predicate calculus restricted to Horn clauses, such that

— F_{HCL} is a countable set of j-place function symbols, $j > 0$,
— C_{HCL} is a countable set of constants.

Let FL (*functional language*) be a functional language, such that

(a) Syntax
 — F_{FL} is a countable set of j-place function symbols, $j > 0$,
 — C_{FL} is a countable set of constants,
 — def $f \in$ FL if $f \in F_{\mathrm{FL}}$.

(b) Semantics
 If def $f \in$ FL and f is an n-place function symbol, then there is a valuation function M, such that

$$M[\![\mathrm{def}\, f]\!] \in (C^n_{\mathrm{FL}} \to C_{\mathrm{FL}}).$$

An example for such a functional language is an FFP system (Backus, 1978) in which "DEF ρ F" corresponds to the above "def f" and the representation function ρ to our valuation function M. Another example is the collection of equations in KRC (Turner, 1981).

The *combination of* HCL *and* FL is the set of pairs $(\{\mathrm{def}\, f_1, \ldots, \mathrm{def}\, f_n\}, S)$, called FHCL, where

— for all $1 \leqslant i \leqslant n$: def $f_i \in$ FL,
— S is a finite set of clauses of HCL.

The set of function symbols of FHCL is $F = F_{\mathrm{HCL}} \cup F_{\mathrm{FL}}$ and the set of constants of FHCL is $C = C_{\mathrm{HCL}} \cup C_{\mathrm{FL}}$. Elements of FHCL are called programs.

The first component of an FHCL-program can be regarded as a functional environment for S; note that in case of $F_{\mathrm{HCL}} \cap F_{\mathrm{FL}} = \emptyset$ this environment cannot be used by the second component S.

In the following we assume that an FHCL-program can only be activated from within the logic part, i.e. by an initial goal statement. If the functional language will completely be defined it should be possible to activate an FHCL-program also from within the functional part.

As an example of an FHCL-program, take the following program which can be used to prove that the edges a, b and c of a triangle satisfy the law of Pythagoras. The set of clauses is given by

$$
\begin{aligned}
S = \{&\mathrm{PYTHAGORAS}(a, b, c) \leftarrow \mathrm{SQUARE}(c, a*a+b*b)\\
&\mathrm{SQUARE}(v, v*v) \leftarrow\\
&\mathrm{MULT}(x, 0, 0) \leftarrow\\
&\mathrm{MULT}(0, y, 0) \leftarrow\\
&\mathrm{MULT}(x+1, y+1, y+1+z) \leftarrow \mathrm{MULT}(x, y+1, z)\}.
\end{aligned}
$$

and def+ and def$*$ are elements of an arbitrary functional language FL such that $M[\![\text{def}+]\!]$ and $M[\![\text{def}*]\!]$ are the operations on natural numbers one would expect.

Note that functional expressions can be used freely within a Horn clause and that the multiplication of two numbers can be performed functionally as well as logically. It seems that the clauses for MULT are not necessary for a computation containing a PYTHAGORAS-clause. In section 4, however, we will demonstrate that in certain cases only a logic evaluation of MULT guarantees a refutation of a PYTHAGORAS-clause.

2. Semantics of FHCL-programs

In the above definition of FHCL both components, the functional and the logic language, seem to play a balanced role. In this section, however, we focus on the logic part of FHCL by describing the semantics of a combined language entirely within the logic framework. There are at least two reasons for this: firstly, the simple and well-known semantics of Horn clause programs provide an elegant way of describing the combination, and secondly, our view of the functional part as a tool for increasing efficiency is reflected.

FUNCTIONAL CLOSURE

We shall now shift the function definitions from the FL-part of an FHCL-program into the HCL-part by transforming every definition into Horn clauses which represent the graph of its meaning. To provide an inference mechanism with the possibility of using these new Horn clauses we also have to include the set of equality axioms.

Let K be the *set of equality axioms* for a given program $\text{prog} = (\{\text{def}f_1, \ldots, \text{def}f_n\}, S)$ as follows:

$$x = x \leftarrow \tag{K1}$$

$$y = x \leftarrow x = y \tag{K2}$$

$$x = z \leftarrow x = y \wedge y = z \tag{K3}$$

$P(x_1, \ldots, x_j, \ldots, x_n) \leftarrow x_0 = x_j \wedge P(x_1, \ldots, x_0, \ldots, x_n)$
for each n-place predicate symbol occurring in S and for all $1 \leqslant j \leqslant n$, (K4)

$f(x_1, \ldots, x_j, \ldots, x_n) = f(x_1, \ldots, x_0, \ldots, x_n) \leftarrow x_j = x_0$
for each n-place function symbol occurring in prog and for all $1 \leqslant j \leqslant n$. (K5)

The *graph of* def$f$ (*graph* (f)) is the possibly infinite set of clauses

$$\{f(c_1, \ldots, c_n) = c \leftarrow -: c_i, c \in C_{\text{FL}} \wedge M[\![\text{def}f]\!](c_1, \ldots, c_n) = c\}.$$

If $\text{prog} = (\{\text{def}f_1, \ldots, \text{def}f_n\}, S)$ is a program, its *functional closure* fcl(prog) is defined as

$$S \cup K \cup F,$$

where

$$K = \text{the set of equality axioms for prog}$$

and

$$F = \text{graph}(f_1) \cup \ldots \cup \text{graph}(f_n).$$

The functional closure of a program will be used as a base for the following two semantics of FHCL.

Note that for some classes of functional languages it is possible to transform a functional program f into a logic program which computes the same results, thus avoiding

the possibly infinite set of clauses graph(f). If, for example, the functional programs consist of a set of equations together with a call-by-name semantics the transformations suggested in Kowalski (1983) can be used.

MODEL-THEORETIC SEMANTICS

The *denotation of a program* prog = ({def f_1, . . ., def f_n}, S) is given by

$$D[\![\text{prog}]\!] = \cap\, \text{Mod}(fcl(\text{prog})),$$

where Mod(fcl(prog)) denotes the set of all Herbrand models for fcl(prog).

If P is an n-place predicate symbol occurring in S, its denotation is

$$D[\![P]\!] = \{(t_1, \ldots, t_n) : P(t_1, \ldots, t_n) \in D[\![\text{prog}]\!]\}.$$

As a trivial consequence of the definition of a functional closure we get the following

PROPOSITION 1

The interpretation of a function symbol $f \in F_{FL}$ determined by each element of Mod(fcl(prog)) is the same as defined by $M[\![\text{def}\, f]\!]$.

OPERATIONAL SEMANTICS

To define the operational semantics by LUSH-resolution (Hill, 1974) we regard a clause as a chain of atoms instead of a set of atoms. A *goal statement* is a chain consisting only of negative atoms. A goal statement *is based on* a set of chains S iff all the constants, function, and predicate symbols occurring in it occur also in S. Without loss of generality we define the operational semantics only for goal statements which contain one atom.

The *success-set* of a set of clauses S is the set of all A in the Herbrand base of S, such that $S \cup \{A\}$ has a LUSH-refutation.

The *operational semantics* of a program prog = ({def f_1, . . ., def f_n}, S) is given by

$$0[\![\text{prog}]\!] = \text{success-set of fcl(prog)}.$$

If P is an n-place predicate symbol occurring in S, its operational semantics is

$$0[\![P]\!] = \{(t_1, \ldots, t_n) : P(t_1, \ldots, t_n) \in 0[\![\text{prog}]\!]\}.$$

From the completeness of LUSH-resolution (Hill, 1974; Apt & van Emden, 1982) we derive

PROPOSITION 2

$D[\![\text{prog}]\!] = 0[\![\text{prog}]\!]$ holds for each FHCL-program.

3. Discussion of Implementations

So far we have given a formal specification of a semantic framework for the combination of functional and logic programming languages. We propose to use this semantics of FHCL as a measure for implementations.

In this section we investigate whether LUSH-evaluation, i.e. the formalisation of the evaluation-strategy used in LOGLISP (Robinson & Sibert, 1982), and input–output patterns satisfy the requirements of our model.

LUSH-EVALUATION

In the definition of LUSH-derivation, a selection function s is used to select atoms from a non-empty goal statement. We extend this concept by allowing s to select also certain terms from within a goal statement.

Let prog $= (\{\text{def} f_1, \ldots, \text{def} f_n\}, S)$ be a program and G the set of goal statements based on S.

A *selection function* s for prog is a total function on $S - \{\square\}$, such that for each $g \in G$, its value $s(g)$ is either an atom from g or a term $f_i(c_1, \ldots, c_r)$ from g and $f_i \in \{f_1, \ldots, f_n\}$.

A LUSH-*evaluation* from prog by s is a sequence $D = (g_0, \ldots, g_m)$ of goal clauses such that for $i = 1, \ldots, m$, g_i is obtained from g_{i-1} by

— input resolution if $s(g_{i-1})$ yields an atom to be resolved upon or
— by substituting the occurrence of $f(c_1, \ldots, c_r)$ in g_{i-1} by $M[\![\text{def} f]\!](c_1, \ldots, c_r)$ if

$$s(g_{i-1}) = f(c_1, \ldots, c_r).$$

LUSH-evaluation can be seen as LUSH-resolution enriched by the possibility of evaluating terms corresponding to the FL-part of a program. An implementation very close to our concept can be found in LOGLISP (Robinson & Sibert, 1982). The substitution accomplished during a LUSH-evaluation step is done in LOGLISP within the deduction cycle as a simplification step, i.e. before selecting the next atom every reducible functional application within the goal statement is reduced.

In the framework of the preceding chapter the substitution-part of LUSH-evaluation can be modelled by the following resolution strategy.

If $s(g)$ yields a term $t_i \equiv f(c_1, \ldots, c_r)$, resolve the atom containing this term, say

$$P(t_1, \ldots, t_n)$$

against the clause from the set of equality axioms K with head P. The resolvent is a goal chain containing

$$f(c_1, \ldots, c_r) = x_0 \wedge P(t_1, \ldots, x_0, \ldots, t_n).$$

Resolving the atom $f(c_1, \ldots, c_r) = x_0$ against the unique clause $f(c_1, \ldots, c_r) = c \leftarrow$ from graph(f), we get a goal chain containing

$$P(t_1, \ldots, c, \ldots, t_n),$$

which is clearly the result of the LUSH-evaluation step.

To demonstrate that LUSH-evaluation is not a correct implementation, take our previous example program for PYTHAGORAS, together with the goal $\leftarrow$ PYTHAGORAS(3, 4, 5) and a selection function such that we get the derivation:

$\leftarrow$ PYTHAGORAS(3, 4, 5)
$\leftarrow$ SQUARE(5, 3 $*$ 3 + 4 $*$ 4) by resolution
$\leftarrow$ SQUARE(5, 9 + 4 $*$ 4)
$\leftarrow$ SQUARE(5, 9 + 16) } by substitutions with respect to LUSH-evaluation.
$\leftarrow$ SQUARE(5, 25)

Since 25 is a constant there does not exist a substitution which unifies SQUARE(5, 25) and SQUARE(v, $v * v$). This implies that PYTHAGORAS(3, 4, 5) is not in the success-set with respect to LUSH-evaluation.

As described above we can simulate in our model the above LUSH-evaluation, but

additionally we can "compute" further

$\leftarrow$SQUARE(5, 25)	
$\leftarrow x_0 = 25 \wedge$ SQUARE(5, x_0)	by resolving against the appropriate equality axiom
$\leftarrow$SQUARE(5, 5 $*$ 5)	by resolving against $5 * 5 = 25 \leftarrow$from graph($*$)
$\square$	by resolving against SQUARE(v, $v * v$)$\leftarrow$.

In other words PYTHAGORAS(3, 4, 5) $\in$ 0⟦PYTHAGORAS⟧, but cannot be computed by LUSH-evaluation.

This incompleteness seems to be well known by language designers; e.g. in LOGLISP the programmer can find a context condition, which allows to use LISP-functions freely only within the hypothesis of a Horn clause.

INPUT–OUTPUT PATTERNS

A less restrictive alternative to control the use of defined function symbols can be based on the idea of mode-declarations or input–output patterns (e.g. Clark & Gregory, 1981; Conery & Kibler, 1983).

For each n-ary predicate symbol there is assumed to exist a mode-declaration, which is simply an integer $i \leqslant n$ together with the convention that

—for all $1 \leqslant j \leqslant i : x_j$ is an input-argument
—for all $i < j \leqslant n : x_j$ is an output-argument.

If no atom in the clause-set of a program contains a defined function symbol within an input-argument and if goal statements only contain constants as input- and variables as output-arguments, it should be possible to prove completeness of LUSH-evaluation for this restricted class of Horn clauses.

However, an essential feature of logic programs gets lost by this restriction of the language, namely the possibility to invert a program. This can usually be done by giving a goal statement, where some of the "input-arguments" are variables and/or some of the "output-arguments" are constants.

In the next section we suggest an implementation which incorporates invertibility and we prove its correctness.

4. Extended Unification

There are two obvious arguments against the use of our semantic model as a base for evaluating FHCL-programs. Firstly, we cannot in general compute the set of clauses F, and secondly, taking input clauses from $S \cup K$ will lead to an exploding deduction tree, where most of the branches are investigated needlessly.

In this section we propose a unification algorithm which takes into account, that terms containing function symbols can be evaluated within the FL-part of an FHCL-program. An algorithm which supports the reduction of functional expressions will be the starting point of our development. We will then demonstrate that in order to incorporate the features demanded by our model, we have to extend this unification procedure. First of all we need some simple definitions.

Let X be an atom or a term and t a term. Then X is *reducible with respect to t*, iff t is a subterm of X, such that

$$t \equiv f(c_1, \ldots, c_n) \wedge \text{def} f \in \text{FL} \wedge M[\![\text{def} f]\!](c_1, \ldots, c_n) = c.$$

To define the reduction of an atom or a term we introduce the notion of a term-substitution $[(e_1/s_1), \ldots, (e_m/s_m)]$, where e_i and s_i are terms, every e_i is different from s_i, and no two elements in the list have the same term after the stroke symbol.

The *application of a term-substitution* $[(e_1/s_1), \ldots, (e_m/s_m)]$ is given by

— $P(t_1, \ldots, t_n)[(e_1/s_1), \ldots, (e_m/s_m)]$
$= P(t_1[(e_1/s_1), \ldots, (e_m/s_m)], \ldots, t_n[(e_1/s_1), \ldots, (e_m/s_m)])$,

— $t[(e_1/s_1), \ldots, (e_m/s_m)] = t(e_1/s_1)[(e_2/s_2), \ldots, (e_m/s_m)]$, $t[\,] = t$,

— $t(e/s) = t'$, where t' is obtained from the term t by simultaneously substituting each occurrence of a subterm s by e.

In the following we use the obvious extension of the application of a term substitution to goal statements and sets

The *reduction of an atom or a term* X, red(X), is given by $\text{red}(X) = X[(c/s)]$, where X is reducible with respect to $s \equiv f(c_1, \ldots, c_n)$ and $M[\![\text{def} f]\!](c_1, \ldots, c_n) = c$.

The *extended disagreement set* of a non-empty set W of expressions is obtained by locating the leftmost symbol, called disagreement symbol, at which not all the expressions in W have exactly the same symbol, and then extracting from each expression in W the subexpression that begins

— with the leftmost function symbol from F_{FL} and contains the disagreement symbol, if such a function symbol exists,

— with the disagreement symbol, otherwise.

We can now formulate our first version of the unification algorithm:

Step 1 Set $k = 0$, $W_k = \{A, B\}$, and $\sigma_k = \varepsilon$.

Step 2 If W_k is a singleton, stop; σ_k is a unifier for $\{A, B\}$. Otherwise, find the extended disagreement set D_k of W_k.

Step 3 If an element of D_k, say t_k, is reducible, let $\sigma_{k+1} = \sigma_k$ and $W_{k+1} = W_k[(\text{red}(t_k)/t_k)]$, and go to Step 6.

Step 4 If there exist elements v_k and t_k in D_k such that v_k is a variable that does not occur in t_k, go to Step 5. Otherwise, stop; $\{A, B\}$ is not unifiable.

Step 5 Let $\sigma_{k+1} = \sigma_k\{t_k/v_k\}$ and $W_{k+1} = W_k\{t_k/v_k\}$.

Step 6 Set $k = k+1$ and go to Step 2.

The reader might have noted that the above algorithm is not a real unification algorithm in the sense that an application of the resulting substitution σ on the input atoms A and B yields in general not two syntactic equal atoms $A\sigma$ and $B\sigma$. However, if one defines an appropriate equivalence class with respect to the reducibility of terms over the Herbrand base, $A\sigma$ and $B\sigma$ are "semantically" equivalent.

THE EXTENDED UNIFICATION ALGORITHM

It is obvious that the above algorithm is very similar to LUSH-evaluation. Using this unification in the Pythagoras-example of section 3 we get the same intermediate goal SQUARE(5, 25), but while LUSH-evaluation does not permit a unification with SQUARE(v, $v * v$), the algorithm above yields $v = 5$ as a proper result. This is obtained by

reducing terms during the unification process, i.e. unification of 25 and $v*v\{v/5\}=5*5$ is possible with

$$5*5[(\mathrm{red}(5*5)/5*5)] = 5*5[(25/5*5)] = 25.$$

However, an attempt to resolve $\leftarrow \mathrm{SQUARE}(z, 25)$ with

$$\mathrm{SQUARE}(v, v*v)\leftarrow \tag{1}$$

will lead to $D_1 = \{25, z*z\}$ within the unification algorithm. Since there is no element in D_1, which is either a variable or a reducible functional expression, the unification will fail.

To avoid the above incompleteness, we assume that an appropriate logic program for the multiplication, say with predicate symbol MULT, is defined. If the procedure were to have the form

$$\mathrm{SQUARE}(v, y) \leftarrow \mathrm{MULT}(v, v, y) \tag{2}$$

unification and resolution could be performed yielding the new goal statement

$$\leftarrow \mathrm{MULT}(z, z, 25).$$

To use the above property we define a partial function eql which, applied to any $f \in F_{\mathrm{FL}}$, yields the *equivalent predicate symbol* for f. More precisely, if $M[\![\mathrm{def}\, f]\!](c_1, \ldots, c_n) = c$, then $\mathrm{eql}(f)$ is an $(n+1)$-ary predicate symbol, for which S contains clauses such that

$$\{(c_1, \ldots, c_n, c) : M[\![\mathrm{def}\, f]\!](c_1, \ldots, c_n) = c\} \subseteq D[\![\mathrm{eql}(f)]\!].$$

In the above example $* \in F_{\mathrm{FL}}$ and $\mathrm{eql}(*) = \mathrm{MULT}$. Note that $D[\![\mathrm{eql}(f)]\!]$ must contain a model for $\mathrm{graph}(f)$ and for the equality axioms as subset such that

$$(s_1, \ldots, s_n, t) \in D[\![\mathrm{eql}(f)]\!] \text{ iff } f(s_1, \ldots, s_n) = t \in D[\![\mathrm{prog}]\!].$$

From the definition of eql and the equality axiom (K4) we conclude that (1) and (2) are equivalent.

To embed this transformation in the unification algorithm we define an extended unifier of the form $\langle \sigma, \tau, \alpha \rangle$, where σ is a substitution, τ is a term substitution, and α is a set of atoms. The elements of α will be added as conditions to the procedure whose head is to be unified. σ and τ will initially be the empty substitutions and α will be the empty set. In the above example

$$\langle \sigma, \tau, \alpha \rangle = \langle \varepsilon, [(25/z*z)], \{\mathrm{MULT}(z, z, 25)\} \rangle.$$

Let $G \equiv \leftarrow A_1 \wedge \ldots \wedge A_m$ be a goal statement and $P \equiv B \leftarrow B_1 \wedge \ldots \wedge B_n$ be a procedure. If A_i and B are unifiable with the extended unifier $\langle \sigma, \tau, \{C_1, \ldots, C_k\} \rangle$, then the goal statement

$$((\leftarrow A_1 \wedge \ldots \wedge A_{i-1} \wedge B_1 \wedge \ldots \wedge B_n \wedge A_{i+1} \wedge \ldots \wedge A_m \wedge C_1 \wedge \ldots \wedge C_k)\tau)\sigma$$

is called the *extended resolvent* of G and P.

The extended unification algorithm is laid down below where

$$\mathrm{eql}'(f(t_1, \ldots, t_n), t) = \mathrm{eql}(f)(t_1, \ldots, t_n, t).$$

Step 1 Set $k=0$, $W_k = \{A, B\}$, and $\langle \sigma_k, \tau_k, \alpha_k \rangle = \langle \varepsilon, [\,], \emptyset \rangle$.

Step 2 If W_k is a singleton, stop; $\langle \sigma_k, \tau_k, \alpha_k \rangle$ is an extended unifier for $\{A, B\}$. Otherwise, find the extended disagreement set D_k of W_k.

Step 3 If an element of D_k, say t_k, is reducible, let

$$\langle \sigma_{k+1}, \tau_{k+1}, \alpha_{k+1} \rangle = \langle \sigma_k, \tau_k . (\mathrm{red}(t_k)/t_k), \alpha_k \rangle,$$
$$W_{k+1} = W_k[(\mathrm{red}(t_k)/t_k)],$$

and go to Step 7.

Step 4 If there exist elements v_k and t_k in D_k such that v_k is a variable that does not occur in t_k, go to Step 5. Otherwise, go to Step 6.

Step 5 Let

$$\langle \sigma_{k+1}, \tau_{k+1}, \alpha_{k+1} \rangle = \langle \sigma_k\{t_k/v_k\}, \tau_k, \alpha_k \rangle,$$
$$W_{k+1} = W_k\{t_k/v_k\},$$

and go to Step 7.

Step 6 If an element of D_k, say t_k, is of the form $f(t_1, \ldots, t_n)$ and $f \in F_{\mathrm{FL}}$ and the other element of D_k, say s_k, is not of the form $g(t_1, \ldots, t_m)$ and $g \notin F_{\mathrm{FL}}$, then let

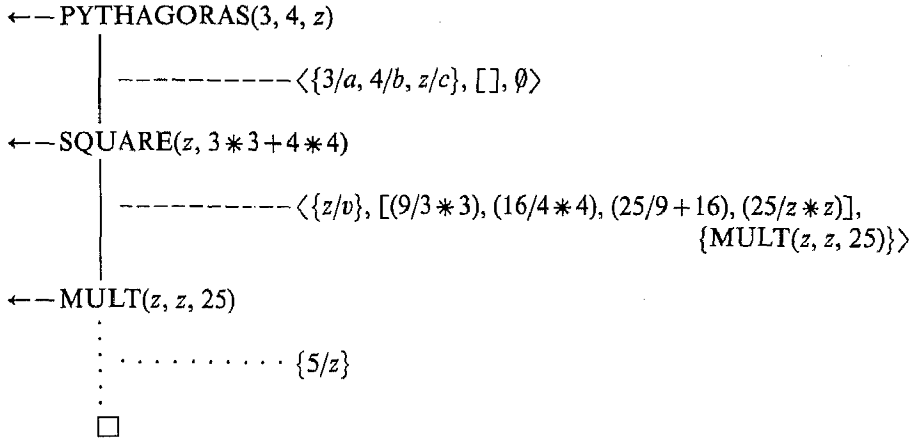
$$\langle \sigma_{k+1}, \tau_{k+1}, \alpha_{k+1} \rangle = \langle \sigma_k, \tau_k . (s_k/t_k), \alpha_k \cup \{\mathrm{eql}'(s_k, t_k)\} \rangle$$
$$W_{k+1} = W_k[(s_k/t_k)];$$

and go to Step 7. Otherwise, stop; $\{A, B\}$ is not unifiable.

Step 7 Set $k = k+1$ and go to Step 2.

Note, in the case that $F_{\mathrm{FL}} = \emptyset$, the extended disagreement set reduces to the disagreement set in pure HCL. Furthermore, there does not exist a reducible term and the extended unification algorithm reduces to the unification algorithm in pure HCL. From these facts we can conclude that the extended resolvent reduces to the well-known resolvent in pure Horn clause logic.

In the following the extended unification algorithm is applied to compute $\leftarrow$PYTHAGORAS(3, 4, z), where eql($*$) = MULT



CORRECTNESS OF THE EXTENDED UNIFICATION ALGORITHM

In this section we prove the soundness of the extended unification algorithm, i.e. its correctness with respect to our model.

Let $A, A_1, \ldots, A_n$ be atoms and prog be a program. Then $0[\![\leftarrow A]\!]_{\mathrm{prog}}$, the meaning of $\leftarrow A$ with respect to prog, is given by

$$0[\![\leftarrow A]\!]_{\mathrm{prog}} = \{\sigma | A\sigma \in 0[\![\mathrm{prog}]\!]\}.$$

The meaning of $\leftarrow A_1 \wedge \ldots \wedge A_n$ is defined by

$$0[\![\leftarrow A_1 \wedge \ldots \wedge A_n]\!]_{\text{prog}} = \cap \{0[\![A_i]\!]_{\text{prog}} \mid 1 \leqslant i \leqslant n\}.$$

Note that from $\sigma \in 0[\![\leftarrow A]\!]_{\text{prog}}$ we can derive $A\sigma \in 0[\![\text{prog}]\!]$ and hence with proposition 2 we derive $A\sigma \in D[\![\text{prog}]\!]$.

Notation: In the following we assume one given program $(\{\text{def} f_1, \ldots, \text{def} f_n\}, S)$ and therefore we will omit the index prog. Let $X[s]$ denote that the subterm s occurs in the atom or term X and $X[s/t]$ denotes X, where one occurrence of t has been replaced by s. $g \Rightarrow g'$ denotes that g' is a LUSH-resolvent of g in fcl(prog) and $g \Rightarrow_E g'$ denotes that g' is an extended resolvent of g in S. Let D, D_1 and D_2 be conjunctions of atoms.

The following technical lemma simply states that we can perform the substitution of a variable for an arbitrary subterm of an atom or a term using only LUSH-resolution.

Lemma 1. *Let X be an atom or a term and s be a subterm of X. The variable y does not occur in X, then*

$$\leftarrow X[s] \Rightarrow^* \leftarrow X[y/s] \wedge y = s.$$

The proof is obtained by induction on the depth of the occurrence of the subterm s in X by applying axioms from K. □

In the extended unification algorithm we can identify two cases where the length of the list of term-substitutions τ is increased.

In step 3, whenever a reduction is performed and the reduced term is substituted, this reduction is additionally memorised in τ and in step 6 when equivalent predicates are introduced. After leaving the unifier we can use these term-substitutions to replace other occurrences of subterms. This reflects the structure-sharing property of an intended implementation. However, from a semantical point of view these term-substitutions can as well be postponed to later extended unification steps.

The following lemma proves that in both cases the term-substitutions in τ can be omitted without changing the semantics.

Lemma 2. *Let $\leftarrow (D\tau)\sigma$ be an extended resolvent, then*

$$0[\![\leftarrow (D\tau)\sigma]\!] = 0[\![\leftarrow D\sigma]\!].$$

Proof. From the extended unification algorithm we derive that $(t/s) \in \tau$ if (i) $\text{red}(s) = t$ or (ii) $\text{eql}'(t, s)$. The proof is obtained by induction on the length of τ. Let τ be $[(t/s)]$ and assume further that s occurs only once in D (the general case is obtained by an induction on the number of occurrences of s).

(i) The reducibility of s implies that there exists a subterm u of s and a constant c such that $u \equiv f(t_1, \ldots, t_n)$ and $u = c \leftarrow \in F$. Let x be a new variable.

"$\Rightarrow$": Let λ be any element of $0[\![\leftarrow (D\tau)\sigma]\!]$. Then by the definition of $\leftarrow (D\tau)\sigma)$, we have

$$\leftarrow ((D\tau)\sigma)\lambda \Rightarrow^* \square$$

and we have to show that

$$\leftarrow (D\sigma)\lambda \Rightarrow^* \square:$$

$$\leftarrow (D\sigma)\lambda[u] \Rightarrow^* \leftarrow (D\sigma)\lambda[x/u] \wedge x = u \qquad \text{(Lemma 1)}$$

$$\begin{aligned}
&\Rightarrow \;\leftarrow (D\sigma)\lambda[x/u] \wedge u = x && \text{(K2)}\\
&\Rightarrow \;\leftarrow (D\sigma)\lambda[c/u] && (u = c \leftarrow)\\
&\equiv \;\leftarrow (D[c/u])\sigma\lambda\\
&\equiv \;\leftarrow (D\tau)\sigma\lambda\\
&\Rightarrow^* \Box.
\end{aligned}$$

Hence $\lambda \in 0[\![\leftarrow D\sigma]\!]$.

"$\Leftarrow$": Assume $\lambda \in 0[\![\leftarrow D\sigma]\!]$. We have to show that

$$\begin{aligned}
&\leftarrow D[t/s]\sigma\lambda \Rightarrow^* \Box :\\
&\leftarrow D[t/s]\sigma\lambda \equiv \;\leftarrow D[c/u]\sigma\lambda\\
&\qquad \equiv \;\leftarrow D\sigma\lambda[c/u]\\
&\qquad \Rightarrow^* \leftarrow D\sigma\lambda[x/u] \wedge x = c && \text{(Lemma 1)}\\
&\qquad \Rightarrow \;\leftarrow D\sigma\lambda[u/u] && (u = c \leftarrow)\\
&\qquad \equiv \;\leftarrow D\sigma\lambda\\
&\qquad \Rightarrow^* \Box.
\end{aligned}$$

(ii) Let $D \equiv D_1 \wedge P[s] \wedge \text{eql}'(t, s)$, where P is the atom containing s. For $\lambda \in 0[\![\leftarrow (D\tau)\sigma]\!]$ we conclude

$$\{(P[t])\sigma\lambda, \text{eql}'(t, s)\sigma\lambda\} \subseteq D[\![\text{prog}]\!],$$

and by the definition of D eql, $(t = s)\sigma\lambda \in D[\![\text{prog}]\!]$. Since $D[\![\text{prog}]\!]$ is also a model for the equality axioms we derive with Lemma 1 that $(P[s])\sigma\lambda \in D[\![\text{prog}]\!]$ and therefore $\lambda \in 0[\![\leftarrow D\sigma]\!]$. Similarly, if $\lambda \in 0[\![\leftarrow D\sigma]\!]$, then $\lambda \in 0[\![\leftarrow (D\tau)\sigma]\!]$ holds.

For $\tau \equiv \tau' . (t/s)$ it follows from the induction hypotheses that

$$0[\![\leftarrow (D\tau)\sigma]\!] = 0[\![\leftarrow (D[(t/s)])\sigma]\!],$$

which is equal to $0[\![\leftarrow D\sigma]\!]$ as shown above. $\Box$

In the following lemma we show that in an extended resolvent each eql$'(s, t)$ can be replaced by $s = t$. In the example of the deduction of $\leftarrow$PYTHAGORAS(3, 4, z), this means that eql$'(z * z, 25)$, which is identical to MULT(z, z, 25), can be replaced by $z * z = 25$.

LEMMA 3.

$$0[\![\leftarrow D \wedge \text{eql}(f)(s_1, \ldots, s_n, t)]\!] = 0[\![\leftarrow D \wedge f(s_1, \ldots, s_n) = t]\!].$$

PROOF. Assume that

$$\lambda \in 0[\![\leftarrow D \wedge \text{eql}(f)(s_1, \ldots, s_n, t)]\!],$$

it follows that

$$\{D\lambda, (\text{eql}(f)(s_1, \ldots, s_n, t))\lambda\} \subseteq D[\![\text{prog}]\!]$$

and from the definition of $D[\![\text{eql}(f)]\!]$, we derive that

$$(f(s_1, \ldots, s_n) = t)\lambda \in D[\![\text{prog}]\!].$$

Therefore,

$$\lambda \in 0[\![\leftarrow D \wedge f(s_1, \ldots, s_n) = t]\!].$$

Similarly, if

$$\lambda \in 0[\![\leftarrow D \wedge f(s_1, \ldots, s_n) = t]\!],$$

then

$$\lambda \in 0[\![\leftarrow D \wedge \text{eql}(f)(s_1, \ldots, s_n, t)]\!]. \quad \Box$$

We will prove the soundness of the extended unification algorithm by constructing a deduction sequence in our model for each extended resolution step. Because of Lemmas 2 and 3 we can omit the term substitution in our extended unification algorithm and we can replace each eql'(s, t) by $s=t$ in the extended resolvent. Using E-unify as an abbreviation for a call of the extended unification algorithm modified as described above, we can state the main lemma for the soundness proof.

LEMMA 4.

$$\text{E-unify}(\{s, t\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle \text{ implies } \leftarrow D \wedge s = t \Rightarrow^* \leftarrow (D \wedge (\Lambda\alpha))\sigma.$$

PROOF. The proof is by induction on the structure of s. Let EST denote E-unify$(\{s, t\}, \langle \varepsilon, \emptyset \rangle)$.

(1) If s is a constant, then EST terminates successfully if

(i) $t \equiv s$: EST $= \langle \varepsilon, \emptyset \rangle$ implies that $\leftarrow D \wedge s = t \Rightarrow \leftarrow D$ by (K1), or

(ii) t is a variable:

EST $= \langle \{s/t\}, \emptyset \rangle$ implies that $\leftarrow D \wedge s = t \Rightarrow \leftarrow D\{s/t\}$ by (K1), or

(iii) $t \equiv f(t_1, \ldots, t_n), f \in F_{FL}$, and t is irreducible:

EST $= \langle \varepsilon, \{t = s\} \rangle$ implies $\leftarrow D \wedge s = t \Rightarrow \leftarrow D \wedge t = s$ by (K2), or

(iv) $t \equiv f(t_1, \ldots, t_n), f \in F_{FL}$, and t is reducible:

obviously, the set of terms with the relation reducible is a well-founded set, and therefore we show by induction on the length m of the reduction sequence performed within the extended unification algorithm that the lemma holds. The reducibility of t implies that there exists a subterm u of t and a constant c such that $u \equiv f'(t'_1, \ldots, t'_n), f' \in F_{FL}$ and $u = c \leftarrow \in F$.

$m = 1$: i.e. red(t) is irreducible. There are two possibilities:

—if red(t) is a constant and red$(t) \equiv s$, then EST $= \langle \varepsilon, \phi \rangle$ implies

$$\begin{aligned}
\leftarrow D \wedge s = t[u] &\Rightarrow^* \leftarrow D \wedge s = t[x/u] \wedge x = u && \text{(Lemma 1)}\\
&\Rightarrow \leftarrow D \wedge s = t[x/u] \wedge u = x && \text{(K2)}\\
&\Rightarrow \leftarrow D \wedge s = t[c/u] && (u = c \leftarrow)\\
&\equiv \leftarrow D \wedge s = \text{red}(t) &&\\
&\Rightarrow \leftarrow D && \text{(K1)}
\end{aligned}$$

—if red$(t) \equiv h(s_1, \ldots, s_n)$ and $h \in F_{FL}$, then EST $= \langle \varepsilon, \{\text{red}(t) = s\} \rangle$ implies

$$\begin{aligned}
\leftarrow D \wedge s = t[u] &\Rightarrow^* \leftarrow D \wedge s = \text{red}(t) && \text{(see above)}\\
&\Rightarrow \leftarrow D \wedge \text{red}(t) = s. && \text{(K2)}
\end{aligned}$$

$m \Rightarrow m+1$: EST $=$ E-unify$(\{s, \text{red}(t)\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle$ implies

$$\begin{aligned}
\leftarrow D \wedge s = t[u] &\Rightarrow^* \leftarrow D \wedge s = \text{red}(t) && \text{(see above)}\\
&\Rightarrow^* \leftarrow (D \wedge (\Lambda\alpha))\sigma && \text{(induction hypotheses)}
\end{aligned}$$

(2) Let s be a variable.

(i) if t is irreducible, then

—if s occurs not in t:

EST $= \langle \{t/s\}, \emptyset \rangle$ implies $\leftarrow D \wedge s = t \Rightarrow \leftarrow D\{t/s\}$ by (K1).

— if s occurs in t, then EST terminates successfully if t is of the form $f(t_1, \ldots, t_n)$ and $f \in F_{FL}$:

EST $= \langle \varepsilon, \{t = s\} \rangle$ implies $\leftarrow D \wedge s = t \Rightarrow \leftarrow D \wedge t = s$ by (K2).

(ii) if t is reducible, then the lemma follows by an induction similar to the one given in (1) and applying (2(i)).

(3) — If $s \equiv f(s_1, \ldots, s_n)$ and $f \in F_{FL}$, we have symmetric cases to (1) and (2) with s and t reversed and the proof is a variation of the theme.

— If $s \equiv f(s_1, \ldots, s_n)$ and $f \notin F_{FL}$, then EST terminates successfully if:

(i) t is a variable: see (2(i))

(ii) $t \equiv f(t_1, \ldots, t_n)$.

By induction on n we prove that

$$\text{E-unify}(\{(s_1, \ldots, s_n), (t_1, \ldots, t_n)\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle$$

implies

$$\leftarrow D \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow^* \leftarrow (D \wedge (\Lambda\alpha))\sigma.$$

$n = 1$:

$$\text{E-unify}(\{(s_1), (t_1)\}, \langle \varepsilon, \emptyset \rangle) = \text{E-unify}(\{s_1, t_1\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle$$

implies

$$\leftarrow D \wedge s_1 = t_1 \Rightarrow^* \leftarrow (D \wedge (\Lambda\alpha))\sigma$$

by the outer induction hypotheses.

$n \Rightarrow n+1$: since

$$\text{E-unify}(\{(s_1, \ldots, s_{n+1}), (t_1, \ldots, t_{n+1})\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle,$$

there exist substitutions σ_1, σ_2 and sets of atoms α_1, α_2 such that

$$\text{E-unify}(\{(s_1, \ldots, s_n), (t_1, \ldots, t_n)\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma_1, \alpha_1 \rangle$$

and

$$\text{E-unify}(\{s_{n+1}\sigma_1, t_{n+1}\sigma_1\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma_2, \alpha_2 \rangle,$$

where

$$\langle \sigma, \alpha \rangle = \langle \sigma_1\sigma_2, \alpha_1 \cup \alpha_2 \rangle.$$

Therefore,

$$\begin{aligned}
& \leftarrow D \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge s_{n+1} = t_{n+1} \\
\Rightarrow^* & \leftarrow (D \wedge (\Lambda\alpha_1) \wedge s_{n+1} = t_{n+1})\sigma_1 \qquad \text{(inner induction hypotheses)} \\
\equiv & \leftarrow (D \wedge (\Lambda\alpha_1))\sigma_1 \wedge (s_{n+1} = t_{n+1})\sigma_1 \\
\Rightarrow^* & \leftarrow ((D \wedge (\Lambda\alpha_1))\sigma_1 \wedge (\Lambda\alpha_2))\sigma_2 \qquad \text{(outer induction hypotheses)} \\
\equiv & \leftarrow ((D \wedge (\Lambda\alpha_1))\sigma_1 \wedge (\Lambda\alpha_2)\sigma_1)\sigma_2 \\
\equiv & \leftarrow (D \wedge (\Lambda\alpha_1) \wedge (\Lambda\alpha_2))\sigma_1\sigma_2 \\
\equiv & \leftarrow (D \wedge (\Lambda\alpha))\sigma.
\end{aligned}$$

It follows that

$$\begin{aligned}
& \leftarrow D \wedge f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \\
\Rightarrow^* & \leftarrow D \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n \\
\Rightarrow & \leftarrow (D \wedge (\Lambda\alpha))\sigma. \quad \square
\end{aligned}$$

Note that

$$\text{E-unify}(\{P(s_1, \ldots, s_n), P(t_1, \ldots, t_n)\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle$$

implies

$$\leftarrow D \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow^* \leftarrow (D \wedge (\Lambda\alpha))\sigma.$$

THEOREM. (*Soundness of the extended unification algorithm*): *Let $(g_0, \ldots, g_n)$ be a sequence of goal statements. Then, for all $0 \leqslant i < n$, $g_i \Rightarrow_E g_{i+1}$ implies $g_i \Rightarrow^* g_{i+1}$.*

PROOF. Because of Lemmas 2 and 3 it is sufficient to show that $g'_i \Rightarrow_E g'_{i+1}$ implies $g'_i \Rightarrow^* g'_{i+1}$, where g' is obtained from g by replacing each eql'(s, t) by $s = t$ and omitting τ.

Let $g'_i \equiv \leftarrow D_1 \wedge P(s_1, \ldots, s_n)$; since $g'_i \Rightarrow_E g'_{i+1}$, there exists a clause of the form $P(t_1, \ldots, t_n) \leftarrow D_2$ in S, such that

$$\text{E-unify}(\{P(s_1, \ldots, s_n), P(t_1, \ldots, t_n)\}, \langle \varepsilon, \emptyset \rangle) = \langle \sigma, \alpha \rangle.$$

Therefore we can construct the following deduction sequence in our model:

$$\begin{aligned}
&\leftarrow D_1 \wedge P(s_1, \ldots, s_n) \\
&\Rightarrow^* \leftarrow D_1 \wedge P(x_1, \ldots, x_n) \wedge s_1 = x_1 \wedge \ldots \wedge s_n = x_n, && \text{(K2, K4, K5)} \\
&\quad \text{where } x_1, \ldots, x_n \text{ are variables not occurring in } g'_i \\
&\Rightarrow \leftarrow D_1 \wedge D_2 \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n && (P(t_1, \ldots, t_n) \leftarrow D_2) \\
&\Rightarrow^* \leftarrow (D_1 \wedge D_2 \wedge (\Lambda\alpha))\sigma \equiv g'_{i+1}. && \text{(Lemma 4)} \quad \square
\end{aligned}$$

IMPROVING THE EXTENDED UNIFICATION ALGORITHM

Our intention is to design a combined functional and logic system where functional evaluation is preferred. Assume we want to carry out some computation using natural numbers. Very likely the functions "+" and "−" and their logic equivalents "ADD" and "MINUS" would be at our disposal. If we try to resolve

$$\leftarrow P(x+3) \quad \text{and} \quad P(5) \leftarrow,$$

we will receive the extended unifier $\langle \varepsilon, [(5/x+3)], \{\text{ADD}(x, 3, 5)\} \rangle$ and the resolvent

$$\leftarrow \text{ADD}(x, 3, 5).$$

Note that ADD$(x, 3, 5)$ will be evaluated logically, However, the 1-dimensional inverse of "+", "−", is also an element of the system and $5-3$ is reducible. Therefore, we may prefer an extended unifier of the form

$$\langle \{5-3/x\}, [(5/x+3)], \emptyset \rangle,$$

which would immediately lead to $\square$.

Step 6 of the extended unification algorithm can be improved such that a functional expression is replaced by its equivalent logic expression only if a reducible inverse functional application does not exist. Otherwise the unification proceeds with the inverse functional application.

We consider it an interesting problem to study the derivation or synthesis of inverse functions as part of an advanced logic programming environment.

5. Related Research

Some of the features of our extended unification algorithm can be found in Subrahmanyam & You (1984). They propose a "semantic unification" where two terms $f(t_1, \ldots, t_p)$ and $g(s_1, \ldots, s_q)$ are semantically unifiable iff there exist semantically equivalent forms (obtained by reduction(s) on any reducible term in $f(t_1, \ldots, t_p)$ and $g(s_1, \ldots, s_q)$), which are unifiable. However, in the execution model based on semantic unification, invertibility is not possible in general. In our extended unification algorithm, this is achieved by using equivalent logic expressions and inverse functions, respectively.

Kornfeld (1983) extends Prolog to include equality: If an attempt to unify two terms t_1 and t_2 fails, the system will establish a goal (EQUAL t_1 t_2). If this goal succeeds, then the unification succeeds with the new variable bindings generated by the refutation of (EQUAL t_1 t_2). Therefore, functional expressions are always evaluated logically, while in our extended unification, algorithm functional expressions are evaluated functionally whenever this is possible. Because a new goal (EQUAL t_1 t_2) takes into account only the terms t_1 and t_2 and not, as we do, the functional expressions that contain t_1 and t_2, the system is incomplete.

Barbuti *et al.* (1984) propose a combined declarative (Horn clause logic) and procedural (functional) programming language whose components operate on the same data (Herbrand terms) and share the basic control mechanisms (rewriting) and basic data control mechanisms (unification). A procedural process can occur as a subgoal in the body of a declarative clause and is rewritten only when it has the necessary information on its input channels. The rationale for this is that a procedural process lacks the invertibility property which is achieved by using equivalent logic expressions and inverse functions in our system.

Fribourg (1984) presents the formalism of equational logic programming. A program is a set of equational definite clauses which is activated by an initial equational goal clause. The computation rules are reflection and superposition. Superposition can produce new definite clauses and hence an equational logic program can be seen as an—sometimes unlimited—extending object. Though equational clauses allow only one predicate, namely the equality, general predicates of the form $P(t)$ can be handled by translating them into the equation $P(t) = \text{true}$, where $P(t)$ is a term and true is a constant. While reflection and clausal superposition are based on the Knuth–Bendix completion rule, our computation is based on unification and resolution.

6. Conclusion

We have given a model for the combination of functional and logic languages. This model relies on the transformation of the functional part of a program "prog" into Horn clauses, yielding the functional closure of "prog". The functional closure together with the LUSH-inference rule preserves the advantages of a pure Horn clause calculus:

— completeness
— separation of logic and control
— invertibility.

We have demonstrated that current proposals for implementations lack some of these properties and have developed an extended unification algorithm taking these features into account. This algorithm is implemented in FranzLISP on a VAX 11/750 (Hölldobler *et al.*, 1985).

We would like to thank Bruno Buchberger for his valuable suggestions concerning the improvement of this paper. Thanks go to Klaus Aspetsberger for his help during the revision of the soundness proof.

References

Apt, K. R., van Emden, M. H. (1982). Contributions to the theory of logic programming. *J. Assoc. Comp. Mach.* **29**, 3.

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* **21,** 8.
Barbuti, R., Bellia, M., Levi, G., Martelli, M. (1984). On the integration of logic programming and functional programming. *IEEE Int. Symp. Logic Prog.*, pp. 160–166.
Chang, C. L., Lee, R. C. T. (1973). *Symbolic Logic and Mechanical Theorem-proving.* New York: Academic Press.
Clark, K. L., Gregory, S. (1981). A relational language for parallel programming. *Proc. of the ACM conf. on functional programming languages and computer architecture.*
Clark, K. L., McCabe, F. G., Gregory, S. (1982). IC-PROLOG language features. In: (Clark, Tarnlund, eds) *Logic Programming.* New York: Academic Press.
Conery, J., Kibler, D. (1983). AND parallelism in logic programs. *Proc. 8th IJCAI.* Karlsruhe, West Germany.
Fribourg, L. (1984). Oriented equational clauses as a programming language. *J. Logic Prog.*, 165–177.
Hill, R. (1974). *LUSH resolution and its completeness.* Memo 78, DCL Univ. of Edinburgh.
Hölldobler, S., Furbach, U., Laußermair, T. (1985). Extended unification and its implementation. *Proc. GWAI '85*, Informatik Fachberichte 188. Berlin: Springer.
Kornfeld, W. A. (1983). Equality for Prolog. *Proc. 8th IJCAI.* Karlsruhe, West Germany.
Kowalski, R. (1983). Logic programming. *IFIP 83.* Amsterdam: Elsevier–North Holland.
Robinson, J. A., Sibert, E. E. (1982). LOGLISP: An alternative to PROLOG. In: (Hayes, Michie, Pao, eds) *Machine Intelligence 10.* New York: John Wiley & Sons.
Sato, M., Sakurai, T. (1983). QUTE: A PROLOG/LISP type language for logic programming. *Proc. 8th IJCAI.* Karlsruhe, West Germany.
Subrahmanyam, P. A., You, J-H. (1984). Conceptual basis and evaluation strategies for integrating functional and logic programming. *Proc. of the 1984 Int. Symp. on Logic Programming.*
Turner, D. A. (1981). *The semantic elegance of applicative languages.* Univ. of Kent at Canterbury.
van Emden, M. H., Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *J. Assoc. Comp. Mach.* **23,** 4.