EXTENDED UNIFICATION ALGORITHMS FOR THE INTEGRATION OF FUNCTIONAL PROGRAMMING INTO LOGIC PROGRAMMING

MEHMET DINCBAŞ AND PASCAL VAN HENTENRYCK*

Extended unification algorithms are considered for the integration of a functional language into a logic programming language. The extended language is a particular case of logic programming language with equality. A comprehensive survey is given which is structured following the procedural semantics taken for the functional language. This survey includes past works based on evaluation and derivation (as procedural semantics of the functional language) and new algorithms based on surderivation. These algorithms are compared especially regarding their completeness. Also, we discuss issues arising in practice when different surderivation strategies are used, as these influence directly efficiency and especially termination. This leads us to propose an algorithm based on lazy surderivation which compares favorably with the others and endows logic programming with two advanced features of functional programming: automatic coroutining and handling of infinite data structures without extra control.

1. INTRODUCTION

We consider the integration of a functional language into a logic programming language. Such an integration can be done at two distinct levels:

the predicate level,
the term level.

*Address correspondence to Dr. M. Dincbas, ECRC, Arabellastrasse 17, D-8000 München 81, West Germany.
Received April 1986; accepted August 1986.
*At same address as Dincbas.
At the predicate level, the integration essentially consists in adding a new built-in predicate of the form eq(\(X, Y\)) (like Edinburgh PROLOG's "is" predicate) which computes the functional expression \(Y\) and unifies the result with the variable (or term) \(X\). Systems such as PROLISP [51], HORNE [18], and LM-PROLOG [5], which combine LISP and PROLOG, use predicate-level integration (see [12] for a survey of languages combining LISP and PROLOG).

At the term level, the integration consists in using functional expressions as terms in predicates. In that case, the functional symbols used in a logic program are partitioned into two sets:

the set \(C\) of "constructors", which are the classical function symbols of logic programming languages and which are used to build up structured data objects;

the set \(F\) of "defined functions", which are the names of functions defined in the functional language and are used to perform computations on the data objects.

In a recent paper Van Emden and Yukawa refer to the predicate-level integration as weak amalgamation and to the term-level integration as strong amalgamation [49]. In the following, we restrict ourselves to term-level integration, as its advantages over the predicate-level are manifold, including more expressiveness and more efficiency (computation of functions at the unification level and not at the resolvent level, avoidance of intermediate variables). In fact, if we have term-level integration, we also have predicate-level integration (in this case the predicate \(eq\) would just be the predicate \(=\) of Prolog). Term-level integration requires a modification to the unification algorithm used in the logic programming language in order to take into account the semantics of the functional symbols of \(F\). Thus, term-level (or strong) integration is achieved through extended unification. In this paper, we give different extended unification algorithms for that purpose. Moreover, the procedural semantics taken for the functional language will have a strong influence on the properties of these algorithms. More precisely, depending on the procedural semantics taken for the functional language, the "extended" logic language will behave differently regarding soundness, completeness, and termination. There are three methods for computing the procedural semantics of functional languages: evaluation, derivation (which is based on reduction), and surderivation (which is based on surreduction). Inside each computation procedure, different strategies (such as innermost, outermost, and lazy) can be used.

Although a lot of work has been done in the integration of functional and logic programming, the extended unification algorithms used so far for the integration of functional and logic languages are (almost) all based on evaluation or on derivation, and therefore, unfortunately, they are incomplete and thus unsound when we consider completed programs [6,34]. However, the use of surderivation leads to sound and complete languages. The idea of using narrowing (a slightly different form of surreduction) for the integration of functional and logic languages, was recently introduced by Goguen and Meseguer in [22] (this idea is also implicit in Fribourg's work [16]). However, neither their paper nor subsequent papers have proposed unification algorithms for this integration, although different strategies behave differently with respect to efficiency and termination. Indeed, the main problem in the use of the surreduction is the termination of the unification
algorithms: using innermost or outermost strategies can lead to (functional + logic) programs which loop although the corresponding logic programs do not.

Therefore, in this paper we present:

A comprehensive survey of the algorithms which can be used to integrate functional and logic programming. This survey is structured following the procedural semantics taken for the functional language and classifies much of the past works within this framework.

Different algorithms based on surderivation (as procedural semantics of the functional language) and an analysis of the issues arising in practice when different surderivation strategies are used. This will lead us to propose an algorithm based on lazy surderivation which compares favorably with other algorithms with regard to termination and efficiency. In addition to providing a complete integration of the two languages, this algorithm also brings into logic languages two advanced features of functional languages: automatic coroutining and computation on infinite data structures without extra control.

Our aim is the full integration of a functional language into a PROLOG-like logic programming language. The resulting extended logic language is a particular case of a logic programming language with equality, the set of rewrite rules composing the functional part of the program defining the equational theory (see Section 4.1). Other approaches for combining the features of these two languages have been proposed, among which we can mention works aiming to add logic-language features to functional languages (see [9], [39], and [43]), and works in which each function definition is required to have its equivalent logic definition in the same program in order not to lose completeness (see [19]). We are not concerned with these approaches in this paper.

This paper is structured as follows: The next section briefly recalls some notions and fixes the notation used throughout the paper. Then, we present “extended unification” algorithms based on evaluation, reduction, and surderivation. We discuss in particular the issues arising when different surderivation strategies are used. Finally, the last two sections discuss respectively the results of the previous sections and some advantages of the last algorithm.

2. THEORETICAL FRAMEWORK AND NOTATION

In this section we will begin by defining notions of term-rewriting systems, reduction, and normal form. Then, we will define the notion of “surderivation” and finish by defining the syntax and the (procedural) semantics of the functional language which will be associated with the logic programming language.

2.1. Reduction

In order to precisely define the notion of reduction, the notions of occurrence, subterm, and replacement must be introduced. Our definitions and notations are consistent with those of [27] and [25].

We consider first-order terms defined on a denumerable set \( V \) of elements called “variables” (which will be noted by capital letters), a finite set \( C \) of elements called
“constructors”, and a finite set \( F \) of elements called “defined function symbols”. Terms have their usual meaning. We define \( V(m) \) as the set of variables of the term \( m \). We also consider sequences of integers (which will represent an access path in a term), with the empty sequence denoted by \( \Lambda \). The operation of concatenation on sequences is denoted by “.”, and the set of finite sequences of positive integers by \((N^+)^*\).

**Definition 1.** We call the elements of \((N^+)^*\) occurrences and we will denote them by \( u, v \). In the same way the set \( O(m) \) of occurrences of a term \( m \) is defined as follows:

1. \( \Lambda \in O(m) \).
2. \( i.u \in O(m) \) iff \( m \) is of the form \( f(m_1, \ldots, m_i, \ldots, m_n) \) and \( 1 \leq i \leq n \) and \( u \in O(m_i) \).

**Definition 2.** The subterm of \( m \) at occurrence \( u \) [with \( u \in O(m) \)], denoted \( m/u \), is defined as follows:

1. \( m \) if \( u = \Lambda \).
2. \( m_i/u \) if \( m \) is of the form \( f(m_1, \ldots, m_i, \ldots, m_n) \) and \( u = i.v \) with \( i \in N^+ \).

In order to distinguish between variable and nonvariable occurrences we define \( O^-(m) \), the set of nonvariable occurrences of a term \( m \), by

\[
O^-(m) = \{ u \in O(m) | m/u \in V \}.
\]

**Example 1.** Let \( m = f(g(X, a), b) \). Then

\[
O(m) = \{ \Lambda, 1, 2, 1.1, 1.2 \},
\]

\[
O^-(m) = \{ \Lambda, 1, 2, 1.2 \},
\]

\[
m/\Lambda = f(g(X, a), b), \quad m/1 = g(X, a), \quad m/1.2 = a.
\]

Note that \( m/v \) is a subterm of \( m/u \) iff \( v = u.w \).

**Definition 3.** The replacement of a subterm in \( m \) at occurrence \( u \) by \( m' \), denoted \( m[u \leftarrow m'] \), is defined as follows:

1. \( m' \) if \( u = \Lambda \).
2. \( f(m_1, \ldots, m_i[v \leftarrow m'], \ldots, m_n) \) if \( m = f(m_1, \ldots, m_i, \ldots, m_n) \) and \( u = i.v \) with \( i \in N^+ \).

**Example 2.** Let \( m = f(g(X, a), b) \). Then

\[
m[1 \leftarrow f(a, a)] = f(f(a, a), b).
\]

**Definition 4.** The depth of an occurrence \( v \) is the integer \( dp(v) \) defined as follows:

\[
dp(v) = |v|, \text{ where } |v| \text{ is the length of the string } v.
\]

We can now define the notions of term-rewriting systems, reduction, and normal form.
Definition 5. A term-rewriting system \([31]\) is a set of pairs of terms \(\Psi \rightarrow \Omega\) such that \(V(\Omega) \subseteq V(\Psi)\).

Definition 6. The term \(m\) reduces to the term \(n\) at occurrence \(u\) in the term-rewriting system \(R\) (in symbols \(m \rightarrow_{u,R} n\)) if and only if there exist \(\Psi \rightarrow \Omega\), a substitution \(\Theta\), and \(u \in O(m)\) such that

\[
m/u = \Theta(\Psi) \quad \text{and} \quad n = m[u \leftarrow \Theta(\Omega)],
\]

where \(\Theta(X)\) is the application of the substitution \(\Theta\) to the term \(X\).

When there is no ambiguity we will denote this reduction by \(m \Rightarrow n\).

Definition 7. We will denote by \(\Rightarrow^*\) the reflexive and transitive closure of the relation \(\Rightarrow\). We call it derivation, and if \(m \Rightarrow^* n\), we say that \(n\) derives from \(m\) in \(R\).

Definition 8. A term \(m\) is in normal form in \(R\) iff there is no \(n\) such that \(m \Rightarrow_R n\).

Definition 9. If \(m \Rightarrow_R n\) and \(n\) is in normal form in \(R\), then \(n\) is a normal form of \(m\) in \(R\).

We define now the notions of confluent and noetherian systems.

Definition 10. A term-rewriting system \(R\) is noetherian if there is no infinite derivation \(m_1 \Rightarrow \cdots \Rightarrow m_i \Rightarrow \cdots\) in \(R\).

Definition 11. A term-rewriting system \(R\) is confluent\(^1\) if for all \(m, m_1, m_2\) such that

\[
m \Rightarrow m_1 \quad \text{and} \quad m \Rightarrow m_2,
\]

there exists \(m'\) such that

\[
m_1 \Rightarrow m' \quad \text{and} \quad m_2 \Rightarrow m'.
\]

Definition 12. A term-rewriting system \(R\) is said to be a canonical term-rewriting system iff it is noetherian and confluent.

Note that in a canonical term-rewriting system each term \(m\) has a unique normal form, which is noted by \(R(m)\) [or \(\text{red}(m)\)] [25].

A term can be reduced in different ways, depending on the choice of the occurrence to be matched with a rewrite rule. We will define two strategies for this purpose: innermost and outermost.

An equivalent characterization of confluence is the so-called Church-Rosser property [25].
Definition 13. The reduction of the term \( m \) at occurrence \( u \) to the term \( n \) is said to be \emph{innermost} in \( R \), denoted

\[
m \xrightarrow{i_R} n,
\]

if \( m \) cannot be reduced in \( R \) to a term \( n' \) at occurrence \( v \) such that \( dp(v) > dp(u) \).

Definition 14. The reduction of the term \( m \) at occurrence \( u \) to the term \( n \) is said to be \emph{outermost} in \( R \), denoted \( m \xrightarrow{o_R} n \), if \( m \) cannot be reduced in \( R \) to a term \( n' \) at occurrence \( v \) such that \( dp(v) < dp(u) \).

2.2. Surreduction

In Definition 6, the substitution applies only to the term \( \Psi_k \) and not to the term \( m/u \). The surreduction allows the substitution to be applied to both terms. Informally speaking, surreducing a term \( t \) is applying to \( t \) the minimum substitution \( \Theta \) such that the term \( \Theta(t) \) can be reduced using rewriting rules.

Let

\[
m_1 \quad \text{be a subterm of} \quad m, \quad \text{at occurrence} \quad u \in O^{-}(m), \quad \text{such that} \quad m_1 \quad \text{can be unified with the left part of a rule} \quad \Psi_k \Rightarrow \Omega_k;
\]

\[
\text{\( \Theta \) be the most general substitution such that} \quad \Theta(\Psi_k) = \Theta(m_1).
\]

Definition 15. The term \( m \) \emph{surreduces} to the term \( m' \) at occurrence \( u \) with the rule \( \Psi_k \Rightarrow \Omega_k \) if and only if

\[
m' = \Theta(m)[u \leftarrow \Theta(\Omega_k)] = \Theta(m[u \leftarrow \Omega_k]) \quad \text{with} \quad \Theta(m/u) = \Theta(\Psi_k),
\]

which is denoted \( m \xrightarrow{[u, k, \Theta]} m' \).

Note that \emph{surreduction} is defined by Hullot in [28] and slightly differs from \emph{narrowing}, defined by Slagle [45] and Lankford [33] and further used by Fay for \( T \)-unification [15], in that it does not require that \( m' \) be put in normal form.

Definition 16. We will denote by \( \Rightarrow \) the reflexive and transitive closure of the relation \( \Rightarrow \). We call it \emph{surderivation}, and if

\[
m \xrightarrow{\ast} n,
\]

we say that \( n \) \emph{surderives} from \( m \) in \( R \).

Definition 17. A term \( m \) is in \emph{surreduced form} in \( R \) iff there is no such \( n \) such that \( m \xrightarrow{R} n \).

Definition 18. If \( m \xrightarrow{\ast} n \) and \( n \) is in surreduced form in \( R \), then \( n \) is a surreduced form of \( m \) in \( R \).

The notions of innermost and outermost surreduction are defined by analogy with the case of innermost and outermost reduction. It must be clear from the
previous definitions that if the term $m$ reduces to term $n$ in $R$, then the term $m$
surreduces to term $n$ in $R$, but the converse is not true.

2.3. Definition of the Functional Language

2.3.1. Syntax. In the following we will define a constructor-based functional
language. Let $F$ be the set of "defined function" symbols and $C$ the set of
"constructor" symbols. We impose $F \cap C = \emptyset$. This strict division between con-
stuctors and defined functions is similar to the strict division between predicate and
functional symbols in logic programming. In practice, a function is "defined" if it
appears as outermost functional symbol in the left-hand side of a rule; it is a
"constructor" otherwise.

**Definition 19.** An $F$-term is a term $f(t_1, \ldots, t_i, \ldots, t_n)$ where $f \in F$. We call $f$ the
label of the $F$-term.

**Definition 20.** A $C$-term is a term $c(t_1, \ldots, t_i, \ldots, t_n)$ where $c \in C$. We call $c$ the
label of the $C$-term.

Note that $F$-terms and $C$-terms can be of null arity. A $C$-term of null arity is
called a constant. The functional language is defined as a confluent term-rewriting
system $\{ \Psi_i \Rightarrow \Omega_i \}$, $1 \leq i \leq n$, with the following restriction:

$\Psi_i$ is an $F$-term $f(t_1, \ldots, t_i, \ldots, t_n)$ where no symbol in $F$ occurs in any $t_i$
($1 \leq i \leq n$).

This restriction is called the "constructor discipline" by O'Donnell in [36]. It
prevents the definition of the rewrite rules like

$\text{revn}(\text{revn}(X)) \Rightarrow X$

and the definition of the so-called "relations between constructors" as allowed in
abstract data types (ADT).

Note that, in practice, the confluence property is achieved with the two following
restrictions on term-rewriting systems [26]:

left linearity: each variable in left-hand side occurs only once;
nonambiguity: there are no critical pairs [31].

Our constructor-based functional language is similar to pattern-directed func-
tional languages like SASL [48], HOPE [4], and the equational language of O'Donnell
[36].

2.3.2. Semantics. If the declarative semantics of such a functional language is
well defined (see for instance [25]), there are many ways to use these rewrite rules to
compute, and thus many different procedural semantics can be defined. In general,
the procedural semantics of functional languages is defined by an "eval" function

---

2See also [47] for a study of the correspondence between different sets of restrictions.
which can be described as follows
\[
\text{eval}(t) = \{
\text{let } t\text{red} = \text{red}(t)
\text{if } t\text{red} \text{ does not contain any } F\text{-term}
\text{then } t\text{red}
\text{else } \bot
\}
\]

Given a ground term, this function computes its normal form. If the normal form contains an \( F\)-term, then the result is \textit{undefined} (\( \bot \)). Otherwise, the result of this function is its normal form. However, the procedural semantics will also depend on the strategy used in the derivation. If an innermost strategy is used, the procedural semantics is “call-by-value”; if an outermost strategy is used, it is “call-by-name”. This semantics will be called \textit{evaluation} in the following.

The procedural semantics can be based on the reduction operation. In this case, given a term \( t \), the result of the computation is its normal form even if that is an \( F\)-term or contains some \( F\)-terms. It is clear that different strategies can be used to compute this normal form. This semantics will be called \textit{derivation} in the following.

Finally, as recently proposed in [10],[38] among others, the procedural semantics of such a language can be based on surreduction. This gives to functional languages some features of logic programming (like the logical variable). Given a term \( t \) (which may contain variables), the interpreter computes the surreduced form(s) of \( t \). Once again, different strategies can be used to compute the surreduced form(s). This semantics will be called \textit{surerivation} in the following.

Note that a lazy strategy can be implemented for the last two procedural semantics. Given a term \( t \), this consists in applying a derivation (surerivation) until \( t \) is reduced (surreduced) to a \( C\)-term. This gives us two additional procedural semantics, and we call them \textit{lazy derivation} \(^3\) and \textit{lazy surerivation}.

It is interesting to consider the different unification algorithms presented in the following with these different procedural semantics in mind. Depending on the procedural semantics taken for the functional language, the extended logic language will behave differently with regard to soundness, completeness, and termination. We will investigate the unification algorithms based respectively on evaluation, on reduction, and on surreduction (as procedural semantics for the functional language).

3. EXTENDED UNIFICATION ALGORITHMS

In this section, we will present several extended unification algorithms based on evaluation, reduction and surreduction. We will assume that we have at our disposal a (syntactical) unification algorithm (in the sense of Robinson [41]) \textit{unif}(\( t_1, t_2 \)) which returns the most general substitution \( \Theta \) such that \( \Theta(t_1) = \Theta(t_2) \) if it exists, and “fail” otherwise. Note also that we will ignore the “occur-check” problem. The reader should have no difficulties in modifying the presented algorithms to include it.

\(^3\)This semantics is usually referred to as “lazy evaluation” in functional languages.
3.1. Algorithms Based on Evaluation

These algorithms have been conceived especially to integrate LISP and PROLOG. Therefore, they use evaluation as procedural semantics of the functional language in order not to modify the LISP interpreter.

3.1.1. Unification with Evaluation: unif1(t₁, t₂). This simple algorithm consists in evaluating the F-terms before unification. More precisely,

\[
\text{unif1}(t₁, t₂) = \{
\begin{array}{l}
\text{if } t₁ \text{ is an } F\text{-term} \\
\quad \text{then} \\
\quad \quad \text{let } t'_₁ = \text{eval}(t₁) \\
\quad \quad \text{if } t'_₁ = \bot \\
\quad \quad \quad \text{then "fail"} \\
\quad \quad \quad \text{else } \text{unif1}(t'_₁, t₂) \\
\quad \text{elseif } t₂ \text{ is an } F\text{-term} \\
\quad \quad \text{then } \{\text{similar to previous case}\} \\
\quad \text{else } \text{unif}(t₁, t₂)
\end{array}
\]

This algorithm is included, among others, in LOVLISP [23] and OBLOGIS [21]. Let us look at some examples of how this algorithm works.

**Example 3.** Consider the clause

\[
factorial(N, \text{fact}(N)) \leftarrow .
\]

and the goal

\[
\leftarrow factorial(5, X).
\]

Assuming that fact(N) is defined in the functional language and that the unification of the arguments is performed from left to right, this goal leads to

\[
\text{unif1}(\text{fact}(5), X) = \text{unif1}(120, X) = \{(120/X)\}.
\]

This algorithm requires that F-terms do not contain variables when evaluated. This is a very strong requirement.

**Example 4.** Consider the clauses

\[
factorial(0, 1) \leftarrow .
\]

\[
factorial(N, N \times M) \leftarrow factorial(N - 1, M).
\]

Assuming that \(\times\) and \(-\) are defined in the functional language, the goal \(\leftarrow factorial(5, X)\) will involve the calls of the unification \text{unif1}(X,5 \times M)\) and the evaluation \text{eval}(5 \times M) = \bot\), which will cause \text{unif1}(X,5 \times M) = "fail". An idea to remedy to such problems consists in delaying as much as possible the evaluation of the F-terms. This leads to the following algorithms.

3.1.2. Unification with Delayed Evaluation: unif2(t₁, t₂). This algorithm allows a variable to unify with any term \(t\). Thus, if \(t\) is an F-term, it is not evaluated before
unification (the evaluation is delayed). The algorithm is defined as follows:

\[
\text{unif2}(t_1, t_2) = \{
\text{if } t_1 \text{ is a variable then } \{(t_2/t_1)\}
\}
\]

\[
\text{else if } t_2 \text{ is a variable then } \{(t_1/t_2)\}
\]

\[
\text{else if } t_1 \text{ is an F-term then}
\begin{align*}
\text{let } t'_1 &= \text{eval}(t_1) \\
\text{if } t'_1 \neq \bot &\text{ then unif2}(t'_1, t_2) \\
&\text{else "fail"}
\end{align*}
\]

\[
\text{else } t_2 \text{ is an F-term then}
\begin{align*}
\text{let } t'_1 &= \text{eval}(t_1) \\
\text{if } t'_1 \neq \bot &\text{ then unif2}(t'_1, t_2) \\
&\text{else "fail"}
\end{align*}
\]

\[
\text{else if } t_1 \text{ is a C-term } c(t_{11}, \ldots, t_{1i}, \ldots, t_{1n})
\text{ then unif2V([t_{11}, \ldots, t_{1i}, \ldots, t_{1n}], [t_{21}, \ldots, t_{2i}, \ldots, t_{2n}])}
\]

\[
\text{else "fail"}
\]

\[
\text{unif2V([t_{11}, \ldots, t_{1i}, \ldots, t_{1n}], [t_{21}, \ldots, t_{2i}, \ldots, t_{2n}])}
\]

\[
\text{is defined as follows:}
\]

1. \[
\text{unif2V([],[])} = \{ \}
\]

2. \[
\text{let } \Delta = \text{unif2}(t_{11}, t_{21})
\text{ if } \Delta \neq "\text{fail}\" \text{ then}
\begin{align*}
\text{let } \Theta &= \text{unif2V([}\Delta(t_{12}), \ldots, \Delta(t_{1i}), \ldots, \Delta(t_{1n})], [\Delta(t_{22}), \ldots, \Delta(t_{2i}), \ldots, \Delta(t_{2n})])
\text{ if } \Theta \neq "\text{fail}\" &\text{ then } (\Theta \circ \Delta) \\
&\text{else "fail"}
\end{align*}
\]

A variant of this algorithm consists in, before binding an F-term to a variable, trying the evaluation of the F-term. If the evaluation give \( \bot \), then the evaluation is delayed and the variable is bound to the F-term; otherwise it is bound to the result of the evaluation.

\[
\text{unif2.2}(t_1, t_2) = \{
\text{if } t_1 \text{ is a variable then}
\begin{align*}
\text{if } t_2 \text{ is an F-term then}
\text{let } t'_2 &= \text{eval}(t_2) \\
\text{if } t'_2 \neq \bot &\text{ then } \{(t_2/t_1)\} \\
&\text{else } \{(t'_2/t_1)\}
\end{align*}
\]

\[
\text{else } t_2 \text{ is a variable}
\begin{align*}
&\text{similar to previous case}
\end{align*}
\]
elseif $t_1$ is an $F$-term
    then
        let $t'_1 = \text{eval}(t_1)$
        if $t'_1 \neq \bot$
            then $\text{unif2.2}(t'_1,t_2)$
        else "fail"
    elseif $t_2$ is an $F$-term
        then (similar to previous case)
    elseif $t_1$ is a $C$-term $c(t_{i1}, \ldots, t_{i1}, \ldots, t_{in})$
    $t_2$ is a $C$-term $c(t_{21}, \ldots, t_{21}, \ldots, t_{2n})$
        then $\text{unif2.2V}([t_{i1}, \ldots, t_{i1}, \ldots, t_{in}], [t_{21}, \ldots, t_{21}, \ldots, t_{2n}])$
    else "fail"

$\text{unif2.2V}(X,Y)$ is defined by analogy with $\text{unif2V}(X,Y)$.
This last algorithm is included in the LISLOG system [2,3]. Let us see how these two algorithms solve the problem of Example 4 and what other problems they still suffer from.

Example 5. Consider the clauses of Example 4,
\[
\text{factorial}(0,1) \leftarrow .
\]
\[
\text{factorial}(N, N \times M) \leftarrow \text{factorial}(N - 1, M).
\]
The goal "$\leftarrow \text{factorial}(5,X)$" will involve the call of the unification $\text{unif2}(X,5 \times M)$, which will succeed with the substitution $\Theta = \{(5 \times M/X)\}$, and at the end of the computation, $X$ can be evaluated to 120. However, this algorithm is not sufficient to solve the goal "$\leftarrow \text{factorial}(5,120)$", because the unification of 120 and $5 \times M$ fails.

Example 6. Let the rewrite system (defining a functional program for "$\Rightarrow") be
\[
0 + X \Rightarrow X.
\]
\[
s(X) + Y \Rightarrow s(X + Y).
\]
and the logic program
\[
\text{plus}(X,Y,X + Y) \leftarrow .
\]
The goal "$\leftarrow \text{plus}(s(X2),X3,s(X1))$" leads to $\text{unif2}(s(X1), s(X2) \neq X3)$, which fails. This clearly shows the limitations of the algorithms based on evaluation. Moreover, taking derivation as the procedural semantics of the functional language will allow us to find the substitution $\{(X2 + X3/X1)\}$ for the unification of $s(X1)$ and $s(X2) + X3$, as their respective normal forms are $s(X1)$ and $s(X2 + X3)$. This leads us to consider the algorithms based on reduction.

3.2. Algorithms Based on Reduction

3.2.1. Unification with Derivation: $\text{unif3}(t_1,t_2)$. This algorithm is very simple and consists in computing the normal form of $t_1$ and of $t_2$ before unifying them (in the
sense of Robinson). Therefore we have

\[ \text{unif}^3(t_1, t_2) = \{ \text{unif}(\text{red}(t_1), \text{red}(t_2)) \} \]

This algorithm is that of the LOGLISP system [42]. Note that the reduction operation will require access to all subterms of terms to be unified and thus can be very costly compared to evaluation.

Example 7. Consider the problem of Example 3,

\[ \text{factorial}(N, \text{fact}(N)) \leftarrow . \]

The goal \( \leftarrow \text{factorial}(5, X) \) leads to

\[ \text{unif}^3(\text{fact}(5), X) = \text{unif}(\text{red}(\text{fact}(5)), \text{red}(X)) = \text{unif}(120, X) = \{(120/X)\} \]

Example 8. Let us take the clauses of Examples 4 and 5:

\[ \text{factorial}(0,1) \leftarrow . \]
\[ \text{factorial}(N, N \times M) \leftarrow \text{factorial}(N - 1, M) \]

The goal \( \leftarrow \text{factorial}(5, X) \) leads to the unification \( \text{unif}^3(X, 5 \times M) \), which will succeed with the substitution \( \Theta = \{(5 \times M/X)\} \), and at the end of the computation, \( X \) can be evaluated to 120. However, this algorithm like the previous ones, does not solve the goal

\( \leftarrow \text{factorial}(5, 120) \)
as 120 and \( 5 \times M \) cannot be unified.

3.2.2. Unification with Lazy Derivation: \( \text{unif}^4(t_1, t_2) \). The aim of this algorithm is to reduce the F-terms only when this is necessary. This avoids a lot of inefficiency, as it does not require the algorithm to scan all the terms at each unification and detects failures as soon as possible. Intuitively, if \( h \) is a new constructor, this amounts to finding a sequence of “outermost” reductions

\[ h(t_1, t_2) \Rightarrow h(t_{11}, t_{21}) \Rightarrow \cdots \Rightarrow h(t_{1n}, t_{2n}) \]

such that there exists a substitution \( \Theta \) such that \( \Theta(t_{1n}) = \Theta(t_{2n}) \).

Before defining \( \text{unif}^4(t_1, t_2) \), we define an auxiliary function lazy-reduce \( (t) \) which reduces \( t \) to a C-term or a variable:

\[ \text{laz}y\text{-reduce}(t) = \{
\begin{align*}
&\text{if } t \text{ is a C-term or a variable} \\
&\quad \text{then } t \\
&\text{else (} t \text{ is an F-term)} \\
&\quad \text{if } t \overset{\sigma}{\Rightarrow} t' \\
&\quad \quad \text{then } \text{laz}y\text{-reduce}(t') \\
&\quad \text{else } \bot \}
\end{align*}
\]

We can now give the definition of \( \text{unif}^4(t_1, t_2) \):

\[ \text{unif}^4(t_1, t_2) = \{
\begin{align*}
&\text{if } t_1 \text{ is a variable} \\
&\quad \text{then } \{(t_2/t_1)\}
\end{align*}
\]
elseif $t_2$ is a variable
then \(((t_1, t_2))\)
elseif $t_1$ is an $F$-term
then
\[
\text{let } t'_1 = \text{lazy-reduce}(t_1) \\
\text{if } t'_1 \neq \perp \\
\text{then unif4}(t'_1, t_2) \\
\text{else } \text{"fail"}
\]
elseif $t_2$ is an $F$-term
then
(similar to previous case)
elseif $t_1$ is a $C$-term $c(t_{11}, \ldots, t_{1n})$
$t_2$ is a $C$-term $c(t_{21}, \ldots, t_{2n})$
then unif4V([t_{11}, \ldots, t_{1n}], [t_{21}, \ldots, t_{2n}])
else "fail".

unif4V($X, Y$) is defined by analogy with unif2V($X, Y$).

This algorithm presents the advantages of the unif3($t_1, t_2$) but avoids its inefficiency. A similar algorithm is included in FUNLOG [46]. Note that if, in the function lazy-reduce($t$), $t'$ has been obtained by innermost reduction, lazy-reduce($t$) would be equivalent, in the case of termination with a result different from $\perp$, to red($t$). Also, in the case where an $F$-term $t$ cannot be reduced, lazy-reduce($t$) yields $\perp$. Another convention could be taken which yields the $F$-term. This would allow unification of two terms $f(b, X)$ and $f(Y, c)$ even if the functional language is defined by the single rule

\[ f(a, X) \Rightarrow X. \]

Example 9. Consider the functional program

\[
\text{conc}([], X) \Rightarrow X. \\
\text{conc}([X|Y], Z) \Rightarrow [X|\text{conc}(Y, Z)].
\]

and the logic program

\[
\text{append}(X, Y, \text{conc}(X, Y)) \leftarrow .
\]

The goal " $\leftarrow \text{append}([a_1, a_2, \ldots, a_n], [b], [c|X])$" will fail after one step of reduction with unif4, while it will fail after $N$ steps of reduction with unif3. On one side, conc([a_1, a_2, \ldots, a_n], [b]) is reduced in unif3 before unifying its normal form with [c|X]. On the other side, in unif4, lazy-reduce(conc([a_1, a_2, \ldots, a_n], [b])) yields \([a_1|\text{conc}([a_2, \ldots, a_n], [b])], which is unified with [c|X]. This unification fails without evaluating \(\text{conc}([a_2, \ldots, a_n], [b]). Moreover, this algorithm has also two additional advantages (automatic coroutining and handling of infinite data structures) which will be presented in the context of unif6. However, the algorithms based on reduction cannot unify, for instance, $N + 1$ and 3. One way to carry out such a unification is to introduce some knowledge of arithmetical operations, providing for example that if two of the arguments are instantiated we can compute the third. Such an approach is proposed in [32]. However, it is still not enough to solve the
problem (see Example 12) where we must find a substitution \( \Theta \) such that \( \Theta(t_1) = \Theta(t_2) \) and \( \Theta(t'_1) = \Theta(t'_2) \) with \( t_1 = N + M, \ t_2 = 8, \ t'_1 = (2 \times M + 4 \times N), \ t'_2 = 20. \) This leads us to consider the algorithms based on surreduction.

3.3. Algorithms Based on Surreduction

There exists a major difference between the previous algorithms and those based on surreduction. In logic programming, the unification of two terms yields a most general substitution which makes both terms equal or fails if such a substitution does not exist. The same can be said of integrations of functional and logic languages based on evaluation and reduction, as in a confluent term Rewriting system, a term has only one normal form. On the contrary, a term can have a set of surreduced forms even in a canonical term-rewriting system. Thus, when surderivation is taken as procedural semantics of the functional language, the unification can yield several substitutions. In this case, the unification of two terms \( t_1 \) and \( t_2 \) defines a set of substitutions \( \{ \Theta_1, \ldots, \Theta_i, \ldots \} \) such that \( \forall i \ \Theta_i(t_1) \) and \( \Theta_i(t_2) \) have the same reduced form.

**Example 20.** Let \( T \) be defined by the term-rewriting system of Example 6:

\[
0 + X \Rightarrow X,
\]

\[
s(X) + Y \Rightarrow s(X + Y).
\]

where 0 represents the integer zero and \( s \) is the successor constructor. \( F = \{ + \} \).

The term \( "N + M" \) has an infinite number of surreduced forms, for instance \( M, \ s(M), s(s(M)), \ldots \). Therefore, the unification will no longer yield a substitution as result, but a set \( E \) of substitutions, and the resolution of two clauses will return as many resolvents as the cardinality of \( E \). Consider the unification of the terms \( "N + M" \) and \( s(0) \). The result of the unification will be the set \( \{(0/N), (s(0)/M)\}, \{(s(0)/N), (0/M)\} \). In the same way, the resolution of the clauses

\[
\neg p(s(0)),
\]

\[
p(N + M) \lor q(N, M)
\]

will give the resolvents

\[
q(0,s(0)),
\]

\[
q(s(0),0).
\]

Moreover, the set of possible substitutions can be infinite. For instance, the unification of \( "N + M" \) and \( "X \times Y" \) will yield an infinite set of substitutions, and thus the resolution of two clauses can lead to an infinity of resolvents. Therefore, the result of the unification is best considered as a stream of substitutions. As a matter of fact, when unifying two terms \( t_1 \) and \( t_2 \), all the substitutions which make the two terms have the same reduced form must be an instance or a variant of a substitution in the stream. *The unification is thus a nondeterministic choice point which yields successively the different substitutions when necessary (i.e. backtracking).* Moreover,
the strategy will have here a great influence on the efficiency and the termination of
the unification algorithm.

3.3.1. Unification with Surderivation: \text{unif5}(t_1, t_2)

3.3.1.1. Presentation of the Algorithm. This algorithm consists in computing
a surreduced form of \( t_1 \) and \( t_2 \) before unifying them (in the sense of Robinson).
Let's designate by \( \text{surreduced}(t) \) a surreduced form of \( t \) in \( R \), by \( \text{sursubst}(t) \) the
composition of the substitutions used during this surderivation, and by \( \text{unif}(X, Y) \)
an algorithm which computes the most general substitution \( \Theta \) such that \( \Theta(X) \) and
\( \Theta(Y) \) are syntactically equal or returns "fail" if \( X \) and \( Y \) are not unifiable. Then we
have

\[
\text{unif5}(t_1, t_2) = \begin{cases}
\text{let } \Theta = \text{unif}(\text{surreduced}(t_1), \text{surreduced}(t_2)) \\
\text{if } \Theta = \text{"fail"} \\
\text{then} \\
\text{"fail"} \\
\text{else} \\
\Theta \circ (\text{sursubst}(t_1) \cup \text{sursubst}(t_2))
\end{cases}
\]

Example 11. Let \( R \) be the term-rewriting system defined as follows:

\begin{enumerate}
\item\( 0 + X \Rightarrow X \).
\item\( S(X) + Y \Rightarrow S(X + Y) \).
\item\( 0 \times X \Rightarrow 0 \).
\item\( s(X) \times Y \Rightarrow Y + (X \times Y) \).
\end{enumerate}

The unification of \( N + s(0) \) and \( s(s(s(0))) \) will give \( \text{unif5}(N + s(0), s(s(s(0)))) = \{(s(s(0))/N)\} \). Indeed,

\[
N + s(0) \Rightarrow s(X_1 + s(0))
\]

\[
\Rightarrow s(s(X_2 + s(0))) \Rightarrow s(s(s(0))).
\]

Example 12. Consider the following clause (taken from Colmerauer [8]):

\text{horse\&man}(X, Y, X + Y, 2 \times X + 4 \times Y) \leftarrow .

such that \text{horse\&man}(\text{Man}, \text{Horse}, \text{Nbhead}, \text{Nbfoot}) holds if \text{Man} and \text{Horse}
are integers which represent the numbers of men and of horses and \text{Nbhead} and \text{Nbfoot}
are respectively the numbers of heads and of feet of the men and the horses. Note that,
for short, we use here the usual notation for integers (e.g. 2 instead of \( s(s(0)) \)).
Contrary to all the algorithms based on evaluation and on reduction, this algorithm
can solve the goal \( \leftarrow \text{horse\&man}(X, Y, 8, 20) \).\]

It will find a substitution \( \Theta \) such that \( X + Y = 8 \) and \( 2 \times X + 4 \times Y = 20 \). This is achieved by performing \( \text{unif5}(X + Y, 8) \) and \( \text{unif5}(\Theta(2 \times X + 4 \times Y), 20) \), where \( \Theta \) is the result of \( \text{unif5}(X + Y, 8) \). Here
\( \text{unif5}(X + Y, 8) \) plays the role of the generator of values for \( X \) and \( Y \), and
\( \text{unif5}(\Theta(2 \times X + 4 \times Y), 20) \) the role of the test.
3.3.1.2. LIMITATIONS OF THE ALGORITHM

3.3.1.2.1. OUTERMOST VERSUS INNERMOST STRATEGIES. However, the strategy used for the surreduction will have a considerable influence on the termination and the efficiency of the algorithm. We will therefore distinguish two algorithms:

- \( \text{unif}5.1(t_1, t_2) = \text{unif}5(t_1, t_2) \), where the surreduced form is obtained by innermost surreduction,
- \( \text{unif}5.2(t_1, t_2) = \text{unif}5(t_1, t_2) \), where the surreduced form is obtained by outermost surreduction.

**Example 13.** Consider the system \( R_1 \) formed by \( R \) (see Example 11) augmented by the following rule:

\[
(5) \quad f(X, s(Y)) \Rightarrow X \times f(X, f(X, Y)).
\]

When unifying \( f(0, Y) \) and 0, \( \text{unif}5.1(f(0, X), 0) \) will enter into an infinite loop because it will generate the infinite sequence of surreductions

\[
f(0, Y) \quad \Rightarrow \quad 0 \times f(0, f(0, Y1))
\]

\[
\quad \Rightarrow \quad 0 \times f(0, 0 \times f(0, f(0, Y2))) \cdots
\]

while \( \text{unif}5.2(f(0, Y), 0) \) will terminate with \( \Theta = \{ (s(Y1)/Y) \} \) because

\[
f(0, Y) \quad \Rightarrow \quad 0 \times f(0, f(0, Y1)) \mapsto 0.
\]

The main difference between innermost surreduction and outermost surreduction is that the latter does the simplifications whenever possible without knowing the value of some arguments. However, there exist some cases where, whatever the choice of the algorithm (\( \text{unif}5.1 \) or \( \text{unif}5.2 \)), unification will loop although the equivalent logic program does not.

3.3.1.2.2. \( \text{unif}5(X, Y) \) VERSUS PURE PROLOG.

**Example 14.** Consider the following logic program:

\[
\begin{align*}
\text{plus}(0, X, X) & \leftarrow . \\
\text{plus}(s(X), Y, s(Z)) & \leftarrow . \\
\text{plus}(X, Y, Z). & \end{align*}
\]

The goal "\( \leftarrow \text{plus}(s(X), Y, 0) \)" will fail, as no clause heads can be unified with it. However, \( \text{unif}5.1(s(X) + Y, 0) \) and \( \text{unif}5.2(s(X) + Y, 0) \) will loop. Indeed, \( "s(X) + Y" \) can be reduced to "\( s(X + Y) \)" which must be unified to 0. These two terms can never be unified, as "\( s(X + Y) \)" represent all the integers greater than zero. However, "\( s(X + Y) \)" has an infinity of surreduced forms, and both algorithms will consider all these surreduced form in order to unify them with 0. But, when the constructor discipline is used, it is not necessary to do that, as the term "\( s(X + Y) \)" can only be surreduced to a term of the form \( s(t) \) where \( t \) is a term, and thus the unification must fail.

\*Note that outermost surderivation can be incomplete.*
The problem arises also in the equational logic language of Fribourg [16], where superposition (a kind of surreduction) is used as inference rule with an innermost strategy. There are programs which can also loop in his language although the corresponding PROLOG program does not. In a subsequent paper [17], Fribourg introduces negative information in order to handle such cases, or more precisely to detect failures.

The main point here is that, in some cases and whatever the strategy used for the surreduction, unif5(\(X, Y\)) (either unif5.1 or unif5.2) will loop although the corresponding PROLOG program does not loop. Moreover, these algorithms, even when they terminate, are not quite satisfactory in terms of efficiency, as they do not detect failure as soon as possible and make unnecessary surreductions. In the abovementioned example, the failure must be detected as soon as “\(s(X + Y)\)” and 0 are to be unified without surreducing “\(s(X + Y)\)”. This is done by the logic program, which considers simultaneously all the arguments (and thus also the “result” argument) during unification. In the same way, when \(s(X + Y)\) and \(s(Z)\) must be unified, the only thing to do is to unify “\(X + Y\)” and \(Z\) without surreducing “\(X + Y\)”. As a matter of fact, the number of surreductions must be minimized. A lazy strategy seems appropriate for this purpose. This leads to the following algorithm.

3.3.2. **Unification with Lazy Surderivation: unif6\((t_1, t_2)\).** We propose the following algorithm to take into account the drawbacks of the algorithm “unif5\((X, Y)\)” by integrating a demand-driven (lazy) strategy. Roughly speaking, the algorithm tries to surreduce \(t_1\) and \(t_2\) in coroutining until they become unifiable in the sense of Robinson. So, if \(h\) is a new constructor (see Hullot [27]), the algorithm tries to find a sequence of surreductions

\[
\begin{align*}
h(t_1, t_2) &\Rightarrow h(t_{11}, t_{22}) \Rightarrow \cdots \Rightarrow h(t_{1n}, t_{2n})
\end{align*}
\]

such that there exists a substitution \(\Theta\) such that \(\Theta(t_{1n}) = \Theta(t_{2n})\) and where \(t_1\) and \(t_2\) have been surreduced in parallel. In order to define the algorithm more precisely, we need an auxiliary function lazy-surred\((t)\) which will compute, when it exists, a pair \(\langle t', \Delta \rangle\) corresponding to a shortest sequence of surreductions such that \(t' \Rightarrow t_1 \Rightarrow \cdots \Rightarrow t'\), where \(t'\) is a variable or a C-term and \(\Delta\) is the composition of substitutions of the different surreductions. More precisely:

\[
\text{lazy-surred}(t) = \begin{cases} 
\text{if } t \text{ is a variable or a C-term} & \langle t, \{ \} \rangle \\
\text{else (t is an F-term)} & \begin{cases} 
\text{if } t \Rightarrow t' & \langle t", \Delta \rangle = \text{lazy-surred}(t') \\
\text{then} & \langle t", \Delta \circ \Theta \rangle \\
\text{else} & \langle \bot, \bot \rangle 
\end{cases} \\
\end{cases}
\]

Note, that, as in the case of lazy-reduce, the above algorithm is called lazy-surdered because it sur-reduces a term until it becomes a C-term (i.e., a constructor appears as the outermost symbol), so the term is not entirely sur-reduced.

We can now define the unification based on lazy surderivation:

\[
\text{unif}_6(t_1, t_2) = \begin{cases} 
(t_2/t_1) & \text{if } t_1 \text{ is a variable} \\
(t_1/t_2) & \text{if } t_2 \text{ is a variable} \\
\{t_1, t_2\} & \text{if } t_1 \text{ is a C-term } c(t_{11}, \ldots, t_{1n}) \\
\{t_2, t_1\} & \text{if } t_2 \text{ is a C-term } c(t_{21}, \ldots, t_{2n}) \\
\text{unif}_6([t_{11}, \ldots, t_{1n}], [t_{21}, \ldots, t_{2n}]) & \text{else}
\end{cases}
\]

where \( \text{unif}_6(X, Y) \) is defined by analogy with \( \text{unif}_2(X, Y) \). Note that the convention of \( \text{unif}_4(X, Y) \) has been also taken here.

**Example 15.** Reconsider Example 14. As \( s(X) + Y \) is an F-term, it is sur-reduced until a constructor appears as outermost symbol. In this case, it is sur-reduced to \( s(X + Y) \). Therefore, \( \text{unif}_6(s(X) + Y, 0) \) leads to \( \text{unif}_6(s(X + Y), 0) \), which fails, contrary to \( \text{unif}_5(s(X + Y), 0) \), which loops.

Moreover, sometimes PROLOG programs which do not terminate have a correspondent which terminates if \( \text{unif}_6(X, Y) \) is used.

**Example 16.** Consider the unification of \( s(N) \times s(M) \) and 0. Then \( \text{unif}_5.1(s(X) \times s(M), 0) \) will loop because it will generate an infinite sequence of sur-reductions (we only show the substitutions concerning the variables of the initial term):

\[
s(N) \times s(M) \quad \overset{1}{\Rightarrow} \quad s(M) + (N \times s(M)) \\
\overset{[\Delta, s(\{\})]}{\Rightarrow} \quad s(M) + (s(M) + (X1 \times s(M))) \\
\overset{[2,4, \{(s(X1)/N)\}]}{\Rightarrow} \quad \cdots
\]
In the same way, \( \text{unif5.2}(s(X) \times s(M), 0) \) will loop because

\[
s(N) \times s(M) \not\overset\circ{\Rightarrow} s(M) + (N \times s(M)) \not\overset\circ{\Rightarrow} s(M + (N \times s(M))) \ldots.
\]

Now \( s(M + (N \times s(M))) \) has an infinity of surreduced forms, so that \( \text{unif5.2}(s(N) \times s(M), 0) \) will generate all these surreduced forms one after another but without ever unifying them with 0. Now, it is obvious that a term of the form \( s(X) \) can never be unified with 0 (providing that the constructor discipline is used). Note that the corresponding PROLOG program will also loop:

\[
\begin{align*}
\text{plus}(0, X, X) & \leftarrow. \\
\text{plus}(s(X), Y, s(Z)) & \leftarrow \\
& \quad \text{plus}(X, Y, Z).
\end{align*}
\]

\[
\begin{align*}
\text{mult}(0, X, 0) & \leftarrow. \\
\text{mult}(s(X), Y, \text{Res}) & \leftarrow \\
& \quad \text{mult}(X, Y, \text{Int}) \& \\
& \quad \text{plus}(Y, \text{Int}, \text{Res}).
\end{align*}
\]

Indeed, a goal \( \leftarrow \text{mult}(s(N), s(M), 0) \) leads to the goal \( \leftarrow \text{mult}(N, s(M), \text{Int}) \& \text{plus}(s(M), \text{Int}, 0) \)”. However, the \( \text{mult}(N, s(M), \text{Int}) \)” will generate an infinite number of results for “\( \text{Int} \)” so the initial goal will never terminate. Now, when our algorithm based on lazy surderivation is considered, the unification terminates, as \( \text{unif6}(s(N) \times s(M), 0) \) leads to \( \text{unif6}(s(M + (N \times s(M))), 0) = \text{"fail"} \).

4. SYNTHESIS

In the following, we consider the properties of the different extended unification algorithms. We will consider mainly the notions of interpreter completeness and program termination.

4.1. Model-Theoretic and Proof-Theoretic Semantics

From a model-theoretic point of view, the integration of a logic and functional language is a particular case of logic languages with equality. Indeed, in logic programming, the equational theory is empty (i.e., two terms are equal if they are syntactically equal). In the extended language, the equality theory is defined by the set of equations (which are no longer oriented in this case). The model theoretic semantics of such a language has been studied in [29] and the reader can refer to that article for more details.\(^5\) When the proof-theoretic semantics is considered, at least two interpretations are available.

A first interpretation consists in introducing a new inference rule in the resolution-based theorem prover [41], which generates a new clause from a clause and a term-rewriting rule. In the case of reduction, the inference rule is the following: Let

\[
\begin{align*}
C = C_1 \lor \cdots \lor C_i \lor \cdots \lor C_n, \\
\Psi_k \Rightarrow \Omega_k.
\end{align*}
\]

\(^5\) Note that in that paper the equational theory is defined by definite equality clauses.
Suppose there exist

an integer \( i \) (\( 1 \leq i \leq n \)),

an occurrence \( u \in O^{-}(C_i) \), and

a substitution \( \Theta \)

such that \( \Theta(C_i/u) = \Psi_k \). Then the resolvent is defined by

\[
C' = \Theta(C_1 \lor \cdots \lor C_i[u \leftarrow \Omega_k] \lor \cdots \lor C_n).
\]

In the same way, in the case of surreduction, let

\[
\Psi_k \Rightarrow \Omega_k.
\]

Suppose there exist

an integer \( i \) (\( 1 \leq i \leq n \)),

an occurrence \( u \in O^{-}(C_i) \), and

a substitution \( \Theta \)

such that \( \Theta(C_i/u) = \Psi_k \). Then the resolvent is defined by

\[
C' = \Theta(C_1 \lor \cdots \lor C_i[u \leftarrow \Omega_k] \lor \cdots \lor C_n).
\]

These two rules correspond to restricted forms of the two well-known "equality-handling" inference rules in classical resolution-based theorem provers. Reduction corresponds to the left-to-right oriented demodulation [50], while surreduction corresponds to the left-to-right oriented paramodulation [40] on unit clauses applied to nonvariable terms in predicates. The inefficiency in the use of these inference rules led to the study of unification in equational theories.

The second interpretation consists in keeping resolution as the unique inference rule but "extending" the unification from a syntactical unification to a unification in an equational theory. In logic programming languages, two terms are equal if they are syntactically equal. In the extended language, the set of rewrite rules composing the functional program defines an equational theory \( T \). The unification of two terms \( t_1 \) and \( t_2 \) in this theory, which is also called \( T \)-unification, consists in finding a substitution \( \Theta \) which makes the two terms equal in the theory \( T \), say

\[
\Theta(t_1) =_T \Theta(t_2).
\]

When the equational theory can be defined by a canonical term-rewriting system \( R \), \( =_T \) is defined via the equality of normal forms in \( R \), i.e.

\[
\Theta(t_1) =_T \Theta(t_2) = \text{rcd}(\Theta(t_1)) = \text{rcd}(\Theta(t_2)).
\]

Unification in equational theories was first studied by Plotkin [37]. In the case of theories defined by a canonical term-rewriting system, we can mention the works of Lankford [33], Fay [15], Fages [14], and Hullot [27]. The paper of Siekmann [44] gives an up-to-date survey, and the thesis of C. Kirchner [30] describes the state of the art of what is called now "universal unification" or unification in (general) equational theories (not necessarily defined by a canonical term-rewriting system, as assumed in our case). Therefore, integrating functional and logic languages amounts to modifying the unification in order to move from a unification in an empty theory
towards unification in an equational theory defined by a term-rewriting system (see also [20]).

Theoretically speaking, when $T$ is a canonical term-rewriting system, there exists a
unification algorithm using surreduction which, given two terms $t_1$ and $t_2$, enumerates all the substitutions $\Theta$ which make $\Theta(t_1)$ and $\Theta(t_2)$ equal in $T$ [15]. However, in practice, depending on the strategy used for the surderivation, different
algorithms behave very differently regarding efficiency and termination, as shown in the
previous section.

4.2. Completeness

A logic program is a set of first-order logic axioms expressed in Horn clauses (and thus universally quantified). The initial goal is a theorem to be proved, and its
variables are existentially quantified. An interpreter for this language is a theorem prover, and one can investigate its soundness and its completeness. However, in practice, the best we can hope for in such an interpreter is the B-completeness property.

\textbf{Definition 21.} An interpreter $I$ is said to be \textit{B-complete} iff for any program $P$ and a

goal $G$,

\begin{itemize}
  \item if $I$ terminates with a substitution $\Theta$, then $^6 P \models \forall(\Theta(G))$ (soundness);
  \item if $I$ terminates with answer “no”, then $^7 P \not\models \exists(G)$;
  \item otherwise $I$ never terminates.
\end{itemize}

Thus, an interpreter based on SLD resolution [34, 1] is B-complete. Consider our
interpreter $I$ implementing SLD resolution with an extended unification. It should be clear that the integration of logic and functional languages must also be
B-complete. However, an interpreter with an extended unification based on evaluation or on reduction does not have this property. It can happen that the unification fails even when two terms can be made equal in the equality theory defined by the functional language (see Examples 4, 5, 6, and 8). Thus, when such an interpreter terminates with answer “no”, the only conclusion that we can draw is that the interpreter cannot prove $\exists(G)$, which is quite different from $P \not\models \exists(G)$. This is particularly important in that the negation in logic programming languages is usually implemented by the “negation as failure” rule [6]. Therefore, these algorithms are not sound if the semantics of a logic program is given by the semantics of the completed program.

On the other hand, for canonical term-rewriting systems, interpreters with a
unification operation based on surreduction are B-complete (providing some complete surderivation strategies). Consider, for example, \texttt{unif6}(t_1, t_2). When the unification “fails”, it means one of two things:

$t_1$ and $t_2$ are both C-terms with distinct labels. In this case, there is no way that the two terms can be unified, because no relation between constructors is used in the functional language (see Section 2.3).

\textsuperscript{6}$\forall(A)$ means that all the variables in $A$ are universally quantified.

\textsuperscript{7}$\exists(A)$ means that all the variables in $A$ are existentially quantified.
$t_1$ or $t_2$ is an $F$-term (say $t_1$) and lazy-surreduced($t_1$) = ⊥. In this case, it means that, whatever the values taken by its variables, $t_1$ can never be surreduced to a $C$-term. Therefore, $t_1$ is a function which will never produce a result, and the unification fails.

4.3. Termination

However, if the interpreters with an extended unification based on surreduction are $B$-complete, their behaviors are quite different in their termination, depending on the strategy used for surreduction (see Examples 13, 15, and 16). Indeed, termination is much harder to obtain than completeness. We will denote by $C1$ [$C2$] the class of programs and associated goals which never terminate when executed within an interpreter based on $\text{unif5}(t_1, t_2)$ [$\text{unif6}(t_1, t_2)$]. There are three possibilities:

1. $C1 \subset C2$.
2. $C2 \subset C1$.
3. $C1$ and $C2$ are not comparable.

The above examples (Example 15 and Example 16) show that statement (1) is false. Intuitively, it seems that statement (2) is true. Indeed, if $\text{unif6}(t_1, t_2)$ never terminates it means that at least one of the terms has an infinity of surreduced forms. Now, in this case, $\text{unif5}(t_1, t_2)$ will also loop because it must also consider an infinity of surreduced forms. Therefore an interpreter based on $\text{unif6}$ must terminate for a larger class of programs than an interpreter based on $\text{unif5}$. This problem of termination is studied in [27] and in [39], where some sufficient conditions are given. We are currently investigating the application of these methods to our particular case (i.e. use of the constructor discipline).

5. ADVANTAGES OF THE UNIFICATION BASED ON LAZY SURDERIVATION

In this section, we will present some advantages of the algorithm $\text{unif6}(t_1, t_2)$. This unification algorithm with lazy surderivation has three main advantages:

1. It preserves the $B$-completeness of logic programs (contrary to unifications based on evaluation or on reduction).
2. It terminates for a larger class of programs (thanks to the lazy strategy).
3. It allows coroutining and computation with infinite data structures [thanks to the demand-driven (call-by-need) strategy] without any extra control.

In the previous sections, we have shown the advantages related to the first and the second points. In this section, we will show with several examples, the advantages related to the third point.

5.1. Automatic Coroutining

The evaluation strategy of the algorithm $\text{unif6}(t_1, t_2)$ allows us to write programs which automatically coroutine. The reason is that the terms $t_1$ and $t_2$ are surreduced
in parallel and therefore their evaluation, if both are F-terms, simulates the coroutining of functions.

*Example 17.* Consider the following logic program “sameleaves(\(X, Y\))” which holds if the trees \(X\) and \(Y\) have the same profile of leaves. A tree \(t\) is represented by

1. \(l(u)\) if \(t\) consists only of a leaf \(u\).
2. \(t(t_1, t_2)\) if \(t\) consists of the two trees \(t_1\) and \(t_2\), which are called subtrees of \(t\).

Then we take

\[
\text{sameleaves}(X, Y) \leftarrow \\
\text{profil}(X, W) \& \text{profil}(Y, W).
\]

\[
\text{profil}(l(U), [U]) \leftarrow .
\]

\[
\text{profil}(t(l(U), X), [U|\text{Res}]) \leftarrow \\
\text{profil}(X, \text{Res}).
\]

\[
\text{profil}(t(t(X, Y), Z), \text{Res}) \leftarrow \\
\text{profil}(t(X, t(Y, Z)), \text{Res}).
\]

This program, used with the PROLOG standard strategy, is extremely inefficient. Indeed, it generates the entire list \(W\) of all leaves of the tree \(X\) before considering the tree \(Y\). Different solutions to this problem have been proposed in the past, such as the annotations of IC-PROLOG [7], the wait declarations in MU-PROLOG [35], and the use of metacontrol in METALOG [11]. In our case, it will be sufficient to express profil(\(X, Y\)) in the functional language, say

\[
\text{profil}(l(U)) \Rightarrow [U].
\]

\[
\text{profil}(t(l(U), X)) \Rightarrow [U|\text{profil}(X)].
\]

\[
\text{profil}(t(t(X, Y), Z)) \Rightarrow \text{profil}(t(X, t(Y, Z))).
\]

and to define the sameleaves procedure as

\[
\text{sameleaves}(X, Y) \leftarrow \\
\text{profil}(X) = \text{profil}(Y).
\]

where the predicate “\(=\)” is defined as usual by

\[
X = X \leftarrow .
\]

This program involves the unification of profil(\(X\)) and profil(\(Y\)), which explores both trees in “coroutining”. Let us see in an example how this works. Let

\[
X_0 = t(l(1), t(l(2), l(3))), l(4)),
\]

\[
Y_0 = t(l(1), l(2)), t(l(3), l(4))).
\]

The goal “\(\leftarrow \text{sameleaves}(X_0, Y_0)\)” will entail the unification of the two trees profil(\(X_0\)) and profil(\(Y_0\)). The unification algorithm “unif6” will call the “lazy-surred” algorithm when necessary. So, we will have the following sequence of calls for unif6:

\[
\text{unif6}(\text{profil}(t(l(1), t(l(2), l(3))), l(4))), \text{profil}(t(l(1), l(2)), t(l(3), l(4))))
\]

\[
\text{unif6}(l(1)|\text{profil}(t(l(2), l(3)), l(4))), \text{profil}(t(l(1), l(2)), t(l(3), l(4))))
\]

\[
\text{unif6}(l(1)|\text{profil}(t(l(2), l(3), l(4))), \text{profil}(t(l(1), l(2)), t(l(3), l(4))))
\]

\[
\text{unif6}(l(1), l(2)) = “\text{fail}”.\]
This shows that the unification fails without computing the remaining parts of the trees \( \text{profit}(l(l(2),l(3)),l(4))) \) and \( \text{profit}(l(l(1)),l(l(3),l(4))) \).

**Example 18.** Consider the procedure append3(\( X_1, X_2, X_3, \text{Res} \)) which holds if the list \( \text{Res} \) is the concatenation of the lists \( X_1, X_2, \) and \( X_3 \). The easiest way to express this program in PROLOG is the following:

\[
\text{append3}(X_1, X_2, X_3, \text{Res}) \leftarrow \\
\text{append}(X_1, X_2, \text{Int}) \& \\
\text{append}(\text{Int}, X_3, \text{Res}).
\]

\[
\text{append}([], X, X) \leftarrow \\
\text{append}(X[X], Z, [X|R]) \leftarrow \\
\text{append}(Y, Z, R).
\]

However, this program never terminates with the goal "\( \leftarrow \text{append3}([1], X, Z, [2|\text{Res}]) \)". The reason is that \( \text{append}([1], X, \text{Int}) \) generates lists of increasing length beginning with 1. Control information must be used in order to prevent this program from looping. The following program solves the problem:

Logic program.

\[
\text{append3}(X_1, X_2, X_3, \text{conc}(\text{conc}(X_1, X_2), X_3)) \leftarrow.
\]

Functional program

\[
\text{conc}([], X) \Rightarrow X.
\]

\[
\text{conc}([X], Y, Z) \Rightarrow [X|\text{conc}(Y, Z)].
\]

Indeed, \( \text{append3}([1], X, Z, [2|\text{Res}]) \) will entail the unification of \([2|\text{Res}]\) and \( \text{conc}(\text{conc}([1], X, Y), Z) \). Furthermore, \( \text{lazy-sured}(\text{conc}(\text{conc}([1], X, Y), Z)) = [1|\text{conc}(\text{conc}(X, Y), Z)] \), so that its unification with \([2|\text{Res}]\) will fail. Therefore the program will terminate. The reason is that the elements of the two lists are compared one by one. Note that the same program can also be used for non-deterministic computations like "\( \leftarrow \text{append3}(X, Y, Z, [1, 2, 3]) \)". This cannot be done by a lazy-derivation strategy.

5.2. Computation on Infinite Data Structures

The “demand-driven” evaluation strategy of the algorithm \( \text{unif6}(t_1, t_2) \) allows the handling of infinite data structures, which can simplify, in some practical cases, the expression of the problem.

**Example 19.** Consider the following term-rewriting rules:

\[
\text{integer} \Rightarrow \text{int}(0).
\]

\[
\text{int}(X) \Rightarrow [X|\text{int}(s(X))].
\]

Here “integer” can be seen as an infinite data structure (in this particular case, an infinite list of nonnegative integers). The ‘demand-driven’ evaluation strategy allows us to manipulate infinite data structures as long as we explore only a finite part of them. This strategy is well known in functional languages (where it is called lazy
evaluation), and the handling of infinite data structures is largely developed in this context [24]. We restrict ourselves to the presentation of two examples.

**Example 20 (Sieve of Eratosthenes).** The program “nprime(N, Res)” holds if Res is the list of the first $N$ prime integers. The predicate prefix($N$, $L$, Res) holds if Res is a prefix of length $N$ of the list $L$. “prime” is the infinite list of prime integers.

**Logic program.**

```
nprime(N,Res) ← prefix (N,prime,Res).
prefix (0,_,[]) ← .
prefix(s(Y),[X|L],[X|Res]) ← prefix(Y,L,Res).
```

**Functional program.**

```
prime ← sift(int(s(s(0)))�).
int(X) ← [X|int(s((X)))�.
sift([N|L]) ← [N|sift(filter(N,L))�.
filter(M,[N|L]) =
  if(divide(M,N),filter(M,L),[N|filter(M,L))�.
if(true, X,Y) = X.
if(false, X,Y) = Y.
```

where:

divide($M$, $N$) is a built-in function which reduces to true if $N$ is divisible by $M$
and to false otherwise.

int($X$) computes the infinite list of integers greater than $X$.

filter($E$, $L$) computes the (infinite) list $LR$ which is obtained from $L$ by
removing all the elements divisible by $E$.

sift($L$) removes from $L$ an element in position $i$ if it is divisible by an element in
position $j$ with $j < i$. Thus sift(int(s(s(0)))) gives the infinite list of all prime
numbers, as int(s(s(0))) represents the infinite list of integers greater than 2.
This is quite convenient way to express this problem.

As we can see, the predicate “prefix” entails the computation of the first $N$
elements of the infinite list “prime” and does not require the computation of the
(infinite) remaining part of it. Let us see in an example how this program works.
Note that, for short, we use here the usual notation for integers [e.g. 3 instead of
$s^3(0)$].

The goal
```
← nprime(3, Res)
```
will entail the call of the new goal
```
← prefix(3, prime, Res).
```
The application of the second clause defining "prefix" involves the unification algorithm "unif6". So we will have (we do not show all intermediate calls)

\[
\text{unif6}([3, \text{prime}, \text{Res}], [s(Y_1), [X_1|L_1], [X_1|R_1]])
\]
\[
\text{unif6}([\text{prime}, \text{Res}], [[X_1|L_1], [X_1|R_1]]) \cdot \{(2/Y_1)\}
\]
\[
\text{unif6}([2, \text{sift}(\text{filter}(2, \text{int}(3))), \text{Res}], [[X_1|L_1], [X_1|R_1]]) \cdot \{(2/Y_1)\}
\]
\[
((2/Y_1), (2/X_1), (\text{sift}(\text{filter}(2, \text{int}(3)))/L_1), ([2|R_1]/\text{Res}))
\]

In the same way, we will have the following sequence of new goals and substitutions

\[\leftarrow \text{prefix}(2, \text{sift}(\text{filter}(2, \text{int}(3))), R_1),\]
which yields \([3|R_2]/R_1\) and the new goal

\[\leftarrow \text{prefix}(1, \text{sift}(\text{filter}(3, \text{filter}(2, \text{int}(4)))), R_2),\]
which yields \([5|R_3]/R_2\) and the last goal

\[\leftarrow \text{prefix}(0, \text{sift}(\text{filter}(5, \text{filter}(3, \text{filter}(2, \text{int}(6)))), R_3),\]
which is unified with the first clause "prefix(0,_,_)". This gives

\[
\text{unif6}([0, \text{sift}(\text{filter}(5, \text{filter}(3, \text{filter}(2, \text{int}(6)))), R_3], [0,_,_])
\]
\[
((1/R_3))\]

So the program will terminate with \(\text{Res} = [2, 3, 5]\). Note that we did not have to compute the infinite list \("\text{sift}(\text{filter}(5, \text{filter}(3, \text{filter}(2, \text{int}(6))))\)\", which represents all the prime numbers greater than 6. In the case of a logic programming language, even with the abovementioned coroutining mechanisms, the program will never terminate (without some extralogical procedures), because the resolvent will contain literals needed to compute this infinite list of prime numbers greater than 6.

**Example 21.** This example [36] allows us to illustrate the potentialities of the handling of infinite data structures. It consists of the addition of two nonnegative integers \(i\) and \(j\). To do so, we use an array \(a\), infinite in its two dimensions. The sum of two integers \(i\) and \(j\) is given by the element \(a[i, j]\) of this array. This array is represented by an infinite list of infinite lists, called "addtable". The predicate "element\(J, \text{List}, \text{Res}\)" holds if \(\text{Res}\) is the \(J\)th element of the list \(\text{List}\). The program is the following:

**Logic program.**

\[
\text{add}(I, J, \text{Res}) \leftarrow \text{element}(J, \text{addtable}, \text{Res}_{\text{int}}) \& \text{element}(I, \text{Res}_{\text{int}}, \text{Res}).
\]
\[
\text{element}(0, [X|Y], X) \leftarrow .
\]
\[
\text{element}(s(X), [X|Y], \text{Res}) \leftarrow \text{element}(X, Y, \text{Res}).
\]

**Functional program.**

\[
\text{addtable} \Rightarrow \text{[int(0)|inclist(addtable)].}
\]
\[
\text{int}(X) \Rightarrow [X|\text{int}(s(X))].
\]
\[
\text{inclist}(0) \Rightarrow s(0).
\]
\[
\text{inclist}(s(J)) \Rightarrow s(s(J)).
\]
\[
\text{inclist}([I|L]) \Rightarrow \text{[inclist}(I)\text{inclist}(L)].
\]
Note that the computation on infinite data structures requires coroutining. However, the fact that we dispose of a coroutining mechanism is not in itself sufficient for manipulating infinite data structures. Indeed, even the logic languages which include a coroutining mechanism cannot deal with the above example.

6. CONCLUSION

A comprehensive survey of extended unification algorithms which can be used in the integration of a functional language into a logic programming language has been presented and structured, following the procedural semantics taken for the functional language. When evaluation and derivation are taken as procedural semantics, the extended logic language, which is a special case of a logic programming language with equality, is not complete. The use of surderivation allows the completeness of the logic language, but in practice the best we can hope for in an interpreter is the B-completeness property. Indeed, the main problem when using surderivation is the termination of the extended unification algorithms. Thus, we have discussed issues which arise "in practice" when different surderivation strategies are used, regarding efficiency and especially termination. This has led us to propose an extended unification algorithm based on lazy surderivation, which compares favorably with others and which brings into logic programming two advanced features of functional programming: automatic coroutining and handling of infinite data structures without any extra control.

We would like to thank Herve Gallaire for his suggestions and valuable comments on the previous version of this paper [13]. We are also grateful to Laurent Fribourg and to an anonymous referee for their helpful remarks.

REFERENCES


