PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

MLOG: A STRONGLY TYPED CONFLUENT FUNCTIONAL LANGUAGE WITH LOGICAL VARIABLES

Vincent Poirriez*

Université de Reims
INRIA-Ecole Normale Supérieure PARIS, FRANCE

Abstract

A new programming language called MLOG is introduced. MLOG is a conservative extension of ML with logical variables. To validate our concepts, a compiler named CAML Light FLUO was implemented. Numerous examples are presented to illustrate the possibilities of MLOG. The pattern-matching of ML is kept for λ-calculus bindings and an unification primitive is introduced for the logical variables bindings. A suspension mechanism allows cohabitation of pattern-matching and logical variables. Though the evaluation strategy for the application is fixed, the order for evaluation of the parts of pairs and application remains free. MLOG programs can be evaluated in parallel with the same result obtained irrespective of the particular order of evaluation. This is guaranteed by the Church Rosser property observed by the evaluation rules. As a corollary, a strict λ-calculus with explicit substitutions on named variables is shown to be confluent. A completely formal operational semantics of MLOG is given in this paper.

1 Introduction

Many attempts have been made at integrating functional and logical tools in the same language. It actually seems worthwile to combine the strengths of the two paradigms, allowing the programmer to choose the most appropriate tool to resolve his problem. The approach we have followed is to add "logical" tools to a well-known strongly typed functional language: ML. To validate our ideas and to demonstrate that MLOG is a realistic proposal, we have implemented a compiler for MLOG named "CAML Light FLUO". It is an extension of the CAML Light system of X.Leroy[Leroy 90]. Logical variables and unification serve two goals in logical languages: to handle partially defined values, and to provide a resolution mechanism. The implementation of logical variables and unification is a required step to implement a resolution mechanism, so we bypass that second goal and focus on the first one. MLOG is an extension of ML with built-in logical variables instantiable once, and unification. We allow a fruitful cohabitation of logical variables and ML pattern matching by introducing a suspension mechanism, thus when an application cannot be evaluated due to a lack of information, the application is suspended. In the designing of MLOG, we strive to obtain a *conservative extension of ML*. Pure ML programs are not penalized by the extension. This result is obtained by limiting the domain of logical variables and suspensions to specified *logical types*. Moreover, MLOG inherits from ML a strong system of types and a safety property for the execution of well-typed programs. Thus the programmer does not waste energy in checking types. In this article, we trace the execution of programs that illustrate that synchronisation algorithms, demand driven computation, algorithms using potentially infinite data structures or partially instantiated values are easily written in MLOG. Then we focus on the confluence property. In MLOG, the strategy for the evaluation of an application is strict evaluation: i.e. we impose the evaluation of the argument before reducing the application. Nevertheless, some freedom remains in the order of evaluation of a term: both parts of an application or of a pair for example. Then MLOG is independent of the implementation choices and it can be implemented on a parallel machine. As we fix the strategy for the evaluation of the applications, we can name variables without risking clashes. A complete operational semantics is given in appendix. A subset limited to the functional part of these rules is a strict λ-calculus with explicit substitutions and named variables that verify the Church Rosser property. That calculus is a very simple formalism and as it is confluent, it is a good candidate to describe any implementation of strict λ-calculus, even a parallel one.

2 MLOG syntax and examples

We describe here the added syntax to ML. As MLOG is an extension of ML, all programs of ML are programs of

*Projet Formel BP 105 Domaine de Voluceau
78153 Rocquencourt Cedex , FRANCE.
poirriez@margaux.inria.fr

MLOG. For clearness, we limit ourselves to a mini-ML. All examples are produced by a session of our system CAML Light FLUO. Note that # is the prompt and ;; the terminator of our system.

2.1 Syntax

The language we consider is λ-calculus with pattern-matching, concrete types (either built-in, as *int* or *string*, or declared by the user), constructors, the `let` construct and the conditional. We first define the set P of *programs* of MLOG. We assume the existence of a countable set Var of term variables, with typical elements x, y, and a disjoint countable set C of constructors, with typical elements c. Some constructors are predefined: integers, strings, booleans (true, false) and (), the element of type unit. In the following, i ranges over integers and s over strings. The syntax of patterns, with typical element p, is:
$p ::= x \mid c \mid (p_1, \ldots, p_n) \mid c\ p$
As in ML, we limit ourselves to linear patterns. The syntax of programs, with typical elements a, b, is:

$$a ::= x \mid c \mid a\ b \mid (a_1, \ldots, a_n) \mid let\ x = a\ in\ b \mid a; b$$
$$(function\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n) \mid undef \mid unif$$

$a; b$ is the ML notation for a sequence, it means evaluate a then evaluate b and return the value of b. The last two constructs are specific to MLOG: *undef* is a generator of fresh logical variables; *unif* is the unification primitive. `let_var u in ...` is syntactic sugar for `let u = undef in ...`.

2.2 Types

In MLOG, the programmer has to declare specially the types that may contain undefined objects (that is, logical variables and suspensions). The notion of *logical type*, is introduced. We assume given a countable set of type variables $TVar$, with typical elements *'a, 'b*, a disjoint countable set of variables over logical types $LTVar$ with typical elements *'a?, 'b?* and two countable sets of type constructors with typical elements ident and lident. The sets of logical types $\mathcal{L}$, with typical element τ_i, and types $\mathcal{T}$ (typical element t_i) are recursively defined by:

$\tau_i ::= {'a?} \mid [t_i]$ lident
and
$t_i ::= \tau_i \mid bool \mid int \mid string \mid unit \mid t_i \rightarrow t_j \mid t_i * t_j \mid$
$[t_i]$ ident

Note that $\mathcal{L}$ is a strict subset of $\mathcal{T}$. Expressions to declare new type are :

type ['a,...,'k]ident = c [of t_i][| ...|c' [of t_j]] |
type logic ['a,...,'k] lident = c [of t_i][| ...|c' [of t_j]]

where [ ] surround optional expressions. A logical type is declared by the new key-word: `type logic`. The type `void` below has a unique value `void` and logical variables of type `void` may be declared. The type `void` is isomorphic to the type `unit` except that no logical variable can be declared in `unit`. A value of the type `Bool` below is `True`, `False`, or a free logical variable that will possibly be instantiated later to either `True` or `False`.

```
#type logic void = void;;
 Type void defined.
#type logic Bool = True | False;;
 Type Bool defined.
```

The following rules govern type variable instantiations: (1) `'a` may be instantiated by any type (including `'b?`); (2) `'a?` may be instantiated by any logical type; (3) `'a?` may not be instantiated by a non logical type.

We write "a : t_i" the program a of type t_i. Thus, the set of MLOG programs is in fact the subset of the well-typed programs $P_{\mathcal{T}}$ of P defined by the familiar ML type system. We just have to specify that : (1) undef : *'a?*; (2) unif : $'a \rightarrow' a \rightarrow void$. Fortunately, as far as types are concerned, logical variables and assignable constructs are quite close, we have adapted to logical variables previous work done for typing assignable objects in ML. We have directly applied the idea of Pierre Weis and Xavier Leroy [LeroyWeis 91], and, using their notion of cautious generalization, we get an extension of the ML type system to logical variables that is *sound*:

Theorem 1 *No evaluation of a well-typed program can leads to a run-time type error.*

Thus CAML Light Fluo has a type-checker that infers and checks the types of programs.

2.3 Examples

We give below very simple examples to illustrate the semantics of unification and logical variables in MLOG. First logical variables are instantiable once, when the unification fails, the exception `Unify` is raised:

```
#let (u:Bool) = undef;;
Value u : Bool u = ?
#unif u True; unif u False;;
- : void  Uncaught exception: Unify
#u;;
- : Bool - = True
```

CAML Light FLUO prints "?" for a free logical variable. Rational trees are allowed; `unif` does not perform any occur-check. Moreover, `unif` does not loop when unifying rational trees. The type `'a stream` below implements the potentially infinite lists.

```
#type logic 'a stream = Nil |St of 'a * 'a stream;;
Type stream defined.
#let (u:int stream) = undef;;
Value u : int stream u = ?
#unif u (St(1,u));u;;
- : int stream
- = St (1, St (1, St (1, St (1,Interrupted.
```

The printing of u was interrupted by a system break. At that point we can use classical technics used in the logical languages, see for example in the appendix the classical functional quicksort program, except that difference lists are used instead of lists to improve the concatenation of sorted sublists.

2.4 Suspensions: an intuitive semantics

Consider first the example below:

```
#let neg = function True -> False |False -> True;;
Value neg : Bool -> Bool
#let b,exp = let_var u in (u, neg u);;
Value b : Bool Value exp : Bool  b = ? exp = ...
```

b is a new free logical variable of type Bool. The application cannot match u with True or False: u is free. So what is the meaning of exp? The answer is: the application neg u is suspended. Thus, exp is a suspension of type Bool[1]. A suspension is a first class citizen in MLOG. It may be handled in data structures, and used in other expressions.

```
#let exp' = unif exp False;;
Value exp' : void  exp' = ...
```

Since exp is a suspension, MLOG cannot perform the unification of exp with False. Therefore this unification is also suspended[2]. Let us now instantiate b with True, and look at exp and exp'.

```
#unif b True; exp,exp';;
Value - : Bool * void  -: (False,void)
```

We have to clarify when a suspension is awakened. Awakening a suspension could be delayed until it is actually needed. We must define when such an evaluation is needed :

```
#let (a,b,e) = let_var a,b in
 (a,b,(function True ->(unif a True))b);;
Value a : Bool  Value b : Bool Value e: void
a = ?  b = ?  e = ...
```

e is suspended waiting for the instantiation of b.

```
#unif b True;;
Value - : void - = void
```

[1]Note that CAML Light FLUO prints suspensions as "...".

[2]That is why the type of the result of unif has to be a logical type. We do not want to have suspension in a non logical type.

As b is instantiated, e can be awakened. If we choose to wake up a suspension only if its value is needed, e remains suspended and then a remains free. If the value of a is needed, nothing indicates that the evaluation of e will instantiate a. This motivates our choice to *wake up all suspended evaluations that can be awakened.* Another motivation is that, if an expression is suspended, it is because its evaluation was needed and unfortunately was stopped by lack of information. So if we look at a:

```
#a;; Value - : Bool - = True
```

The example above illustrates the fine control on evaluation allowed by the suspension mechanism. The application is performed and then a is instantiated only when b is instantiated.

3 A confluence result

To give an operational semantics for MLOG we have to deal with bindings of λ-calculus variables, bindings of logical variables and suspensions. We give here a simple formalism that allows us to keep named parameters and we show that this calculus is strongly confluent[3]. In this section we neglect types.

3.1 A strict calculus with environment

We store bindings of parameters in environments. We call $E\Lambda$ the set of terms with environments. As our calculus is strict, we specialize a subset Val of $E\Lambda$ which is the set of the values handled by the language. Typical elements of Val and $E\Lambda$ are respectively noted v and t.

e ::= [] | (x,v)::e
v ::= c | c(v) | (v,v') | (function ...).e
t ::= c | c(t) | (t,t') | t(t') |a.e

3.2 Logical variables, substitutions and suspensions

Now we have to extend the set Val with logical variables. We assume the existence of a countable set U disjoint with V and C with typical element $u(i)$, distinct logical variables have distinct indexes. We call $LVal$ and $EL\Lambda$ the obtained sets of values and terms with environments. To manage the bindings of logical variables we define substitutions as functions from U to $EL\Lambda$. We will use greek letters to note substitutions. We call the domain of σ and note $dom(\sigma)$ the set $\{u(i)\ s.t.\ \sigma(u(i)) \neq u(i)\}$. We will note $\sigma \circ \alpha$ the composition of substitutions. The MLOG pattern matching algorithm has to deal with logical variables. It has to

[3]Recall that if no strategy for application is imposed, name clash may occurs. To avoid that problem, the names of variables can be replaced by numbers "à la De Bruijn"[AbadiCaCuLe 90, HardinLevy 90]

access to the pointed value when it checks a bound variable, it fails with *Unknown* when it tries to match a free logical variable with a construct pattern. We define the match of a term t with a pattern *pat* in the substitution σ and note $\Phi_\sigma(pat,t)$ as the list of appropriate bindings of parameters of pat. Recall that patterns are linear. We define now a sequential pattern matching without entering into the optimization of the algorithm[4].

if $\Phi_\sigma(p_0,t) = e$ then $\Phi s_\sigma(i, p_0 :: pl, t) = i, e$
if $\Phi_\sigma(p_0,t) = Unknown$ then $\Phi s_\sigma(i, p_0 :: \_, t) = i, Unknown$
if $\Phi_\sigma(p_0,t) = fail$ then $\Phi s_\sigma(i, p_0 :: [], t) = i, fail$
if $\Phi_\sigma(p_0,t) = fail$ and $pl \neq []$then
$\Phi s_\sigma(i, p_0 :: patl, t) = \Phi s_\sigma(i+1, patl, t)$

When the pattern matching fails with *Unknown*, we suspend the application. We do not want to have to go throughout the term to wake up suspensions or to duplicate suspensions when reducing application. On other hand, we note that both free logical variables and suspensions are holes in the term that will be plugged in when more information is broadcast. So we replace the new suspension by a logical variable $u(j)$ (with $j < 0$ to recall that it is created for a suspension) and we bind $u(j)$ with the suspension in a dedicated substitution α(See rules **Susp** and **ASusp** in figure 2). As explained above, unification may build rational trees, thus a naive recursive application of a substitution to a term may loop. We define $\sigma^*(t)$ as the recursive application of σ to t that does not substitute a logical variable if it has already been substituted in a prenex occurrence of t. More precisely, we call M the set of the logical variables of $dom(\sigma)$ already met, σ^* is defined by:

$\sigma^* = \emptyset \vdash \sigma^*$ and
$M \vdash \sigma^*(u(i)) = u(i)\ if\ u(i) \in M$ or $u(i) \in dom(\sigma)$
$M \vdash \sigma^*(u(i)) = \{u(i)\} \cup M \vdash \sigma^*(\sigma(u(i)))\ if\ u(i) \notin M$
$M \vdash \sigma^*(c) = c$
$M \vdash \sigma^*(t(t')) = (M \vdash \sigma^*(t))(M \vdash \sigma^*(t'))$
$M \vdash \sigma^*(t,t') = (M \vdash \sigma^*(t), M \vdash \sigma^*(t'))$
$M \vdash \sigma^*(p.e) = (M \vdash \sigma^*(p).M \vdash \sigma^*(e))$

3.3 Unification

The used unification procedure is adapted from [Huet 76]. We do not discuss here the whole algorithm but the three following points deserve mention: (1) We do not want to open the Pandora's Box of higher order unification, so when we compare closures we limit ourselves to physical identity (we assume an appropriate primitive *eq*).(2) When the procedure has to unify a suspension with any other term, it stops and returns *susp*[5]. (3) When the procedure has to unify a free logical variable with a construct term, the unification is performed even if a suspension occurs in the term. We define $unif_{\sigma_0}(t,t')$ by:
(a) $unif_{\sigma_0}(t,t') = \sigma$ iff the unification procedure applied to$\{(t,t')\}$ with the initial substitution σ_0 succeeds and builds the substitution σ.
(b) $unif_{\sigma_0}(t,t') = fail$ iff the unification procedure applied to $\{(t,t')\}$ with the initial substitution σ_0 stops with *fail*.
(c) $unif_{\sigma_0}(t,t') = susp(u(i))$ iff the unification procedure applied to $\{(t,t')\}$ with the initial substitution σ_0 stops with $susp(u(i))$.
The following result holds:

Theorem 2 *For all terms t,t' $unif_{\sigma_0}(t,t')$ terminates and:(a) if t and t' are not unifiable in the initial substitution σ_0, then $unif_{\sigma_0}(t,t') = fail$ or $susp(\_)$; (b) otherwise if there is at least one pair of the form $(u(j), t")$ with $j < 0$ built then $unif_{\sigma_0}(t,t') = susp(\_)$ (c) else $unif_{\sigma_0}(t,t') = \sigma$ which is the most general unifier of (t,t'), moreover there is no cycle in σ of the form $\sigma^*(u(i)) = u(i)$.*

3.4 Confluence of the reduction over $EL\Lambda$

The reduction has to account for the bindings of logical variables and those of logical variables created for the suspensions. Moreover, it has to deal with waking up the suspensions. Thus we define $\rightarrow$ as the smallest relation over $EL\Lambda \times substitutions \times substitutions \times substitutions$ that verifies the rules given in figures 1 and 2 in appendix . A 4-tuple is note by $< t, \sigma, \alpha, \Gamma >$ where t is the term to reduce. The substitution σ stores the bindings of unified logical variables and updated suspensions. The valuation α stores the suspensions (recall they are bound to $u(j)$ with $j < 0$). The substitution Γ stores the suspensions of which evaluations are running. We use the classical notation $\xrightarrow{*}$ and $\xrightarrow{n}$ for reflexive transitive closure of $\rightarrow$ and for derivations of length n. We first have two lemmas that say that no term of the form (a.e).e' is produced and that the term component of a normal form is a value.

Lemma 1 *Let a be a program and $< a.[], \emptyset, \emptyset, \emptyset > \xrightarrow{n} < t, \sigma, \alpha, \Gamma >$. For all subterms of t of the form $t'.e$, t' is a program.*

Lemma 2 *Let a be a program and $< a.[], \emptyset, \emptyset, \emptyset > \xrightarrow{*} < t, \sigma, \alpha, \Gamma >$ such that $< t, \sigma, \alpha, \Gamma >$ is a normal form. Then t is a value.*

We can deduce from these lemmas that all bindings in σ bind a variable with a value. Let us remark now that if no suspension rule is applied, as we do not reduce under a λ and we impose a strict calculus we have strong confluence for our reduction rules.

[4] The interested reader is referred to [Laville 88] and [PuelSuarez 90] for presentation of optimized algorithms in the framework of functional lazy evaluation. Such algorithms may be of some interest for our language as they avoid useless tests and then avoid useless suspensions.

[5] *susp* is returned even if the procedure has to unify a free logical variable and a suspension.

Proposition 1 *Let* $< t, \sigma, \alpha, \Gamma > \rightarrow < t_1, \sigma_1, \alpha, \Gamma_1 >$ *and* $< t, \sigma, \alpha, \Gamma > \rightarrow < t_2, \sigma_2, \alpha, \Gamma_2 >$ *two reduction using respectively rules* $r1$ *and* $r2$ *with* ri *not a suspension rule. Then we have by the application of respectively* $r2$ *and* $r1$*:* $< t_1, \sigma_1, \alpha, \Gamma_1 > \rightarrow < t_3, \sigma_3, \alpha, \Gamma_3 >$ *and* $< t_2, \sigma_2, \alpha, \Gamma_2 > \rightarrow < t_3, \sigma_3, \alpha, \Gamma_3 >$

An important corollary of that result is that if we restrict ourselves to the functional subset of MLOG, we have describe a strong confluent calculus with explicit substitutions and named variables. That calculus is rather simple (all that concerns logical variables and suspensions is unnecessary) and describes all implementations of a strict λ-calculus, even a parallel one.

Remark that $\rightarrow$ is not strongly confluent on the whole language. That is illustrated by the example below where the choice is between **UnifT** and **Susp** and the diagram cannot be closed in one step as even if **UnifT** is chosen after **Susp** waking up the suspension remains to be done.

$$< ((fun\ c \rightarrow c').[]\ u(1), unif\ u(1)\ c), \emptyset, \emptyset, \emptyset >$$

We can see the use of a rule **Susp**, **ASusp** or **USusp** as the translation of subterm from the term to Γ. From a reduction point of view we can say that these rules do not work. Thus the idea is to define an equivalence between four_uples $< t, \sigma, \alpha, \Gamma >$ which is stable for these suspension rules and then show the strong confluence of $\rightarrow$ up to that equivalence.

Definition 1 $< t, \sigma, \alpha, \Gamma > \equiv < t', \sigma', \alpha', \Gamma' >$ *iff*

1. *there exists a permutation* $\mathcal{P}$ *over positive variable index such that* $(\sigma \circ \alpha \circ \Gamma)^*(t) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(t')$
2. *and for all* $u(i)$ *in* $dom(\sigma)$ *with* $i > 0$, $(\sigma \circ \alpha \circ \Gamma)^*(u(i)) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(u(\mathcal{P}(i)))$
3. *and for all* $u(i)$ *in* $dom(\alpha) \cup dom(\Gamma)$ *or there exists* $j < 0$ *such that* $u(j)$ *in* $dom(\alpha') \cup dom(\Gamma')$ *and* $(\sigma \circ \alpha \circ \Gamma)^*(u(i)) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(u(j))$, *either there exists a subterm* t'_i *of* t' *such that* $(\sigma \circ \alpha \circ \Gamma)^*(u(i)) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(t'_i)$

   and vice versa for all $u(i)$ *in* $dom(\alpha') \cup dom(\Gamma')$

or $t = t' = failwith(s)$

Thus we have verified the Church Rosser property (the proof is in appendix C):

4 MLOG: a conservative extension of ML

The fact that the type of `undef` is `'a?` ensures that no logical variable occurs in a non-logical type. That is not enough to ensure that no suspension of a non-logical type is built. Fortunately, we handle type information when we compile the pattern matching. Thus we have the following rules for the application:

Let f be a function of type $t_1 \rightarrow t_2$: (1) if type t_1 is a non-logical type, then do not do any test to check if the argument is a free variable or a suspension. (2) if type t_1 is a logical type, then (21) first, test if the argument is a bound logical variable or an updated suspension, and access the bound value. (22) if type t_2 is a non-logical type, test if the argument is a free variable or a suspension. If so, raise failure *Unknown*. (23) if type t_2 is a logical type, test if the argument is a free variable or a suspension. If so build and return the appropriate suspension.

Example:

```
#type logic 'a partial = P of 'a;;
 Type partial defined.
#(function (P x) ->x) undef;;
uncaught exception Unknown
```

Theorem 4 *Let a be a well-typed program. The evaluation of a cannot build a logical variable or a suspension of a non-logical type.*

We can now deduce that MLOG is a conservative extension of ML as pure ML programs need not know for the extension. However, it is clear that with that rule of failure, our calculus is no longuer Church Rosser. To keep that property, we must not use functions from a logical type to a non-logical type. Let call $MLOG^*$ the subset of MLOG that does not contain such functions. Thus, we have the following result.

Proposition 2 *The relation* $\rightarrow$ *is confluent on* $MLOG^*$.

Remark: The counterpart of the conservative property of MLOG is the need to be cautious with logical variables and "functional types". First, for any instances of $'a$ and $'b$ the type $'a \rightarrow' b$ cannot include a logical variable as it is a "pure ML" type. Anyway, it is correct to have logical variables of type $(int \rightarrow int)partial$ as illustrated below.

```
#unif g (P (fun x -> x*x));;
- : void  - = void
#e2;;
- : int partial  - = P 4
```

5 Conclusion

We have defined MLOG as an extension of ML. We have shown that it verifies a Church Rosser property and then it may be parallelized or used to simulate parallel processes. Such processes can communicate with each other through shared logical variables and the suspension mechanism allows synchronization. Partial data are handled by MLOG, for example potentially infinite lists can be implemented by the use of free logical variables for the tail of the structure (see example in appendix).

MLOG includes a suspension mechanism, let us now compare it to some other proposals of integration that have made a similar choice. MLOG is close to the language Qute defined by M.Sato and T.Sakurai in [SatoSakurai 86]. However, it differs from it in the following points: (1) its evaluation strategy ensures that the evaluation of a suspended expression will be tried only when needed information is provided; (2) the reduction of an application is allowed even if a subexpression of the argument is suspended, the only condition is that pattern matching succeeds, in that case the binding of the suspension by a logical variable and the storage in α avoid duplication of that suspension.

MLOG is also close to GHC of K.Ueda [Ueda 86], the main difference (except for typing point of view) is that MLOG does not have non-determinism for rule selection and that we have preferred to keep the functional formalism in place of the predicate one as selection of rules is done by pattern matching. However, determinist GHC programs are easily translated in MLOG[6].

The use of a suspension mechanism and the cohabitation of logical variables and functions are common to Le Fun of H.Ait Kaci[Ait Kaci 89] and MLOG. Here the main differences are that Le Fun provides a resolution mechanism based on backtracks and that MLOG is strongly typed.

Perhaps the main difference between MLOG and these related works is that MLOG is a conservative extension of ML. We demonstrate that the type system of ML can be extended to MLOG and we gave a safety property for well typed programs. As a side effect, we have described an operational semantics for strict λ-calculus which uses names for parameters and verifies the Church Rosser property. Therefore it can be used to describe any interpreter of strict λ-calculus, even parallel one. If it seems desirable, further work can be done to provide a resolution mechanism in MLOG. Note that the exhaustive search transformation described by K.Ueda in [Ueda 86] is applicable.

We hope that MLOG is an attractive extension of ML as from a "logical paradigm" point of view it allows handling incomplete data structures and controlled parallel evaluation with the improvement of the ML type system. And from a "functional paradigm" point of view, it respects functional programs with the improvement of partial data and a fair control mechanism.

Acknowledgments: We would like to thanks all members of LIENS-INRIA Formel project for helpful discussions. In particular Therese Hardin for her accurate suggestions to improve our formalism and demonstration.

A Appendix: MLOG programs

The program below is the classical functional quicksort program, except that difference lists are used instead of lists to improve the concatenation of sorted sublists. This is done by the use of the same variable r in both recursive calls of qsortrec.

```
#let partition order x =
let rec partrec = function
 Nil -> Nil,Nil
|St(h,t) -> let infl,supl = partrec t in
  if order(h,x) then St(h,infl),supl else infl,St(h,supl)
 in partrec ;;
Value partition :
('a*'b->bool)->'b->'a stream->'a stream*'a stream
#let quicksort order l =
 let rec qsortrec = function
 (Nil,result,sorted) -> (unif result sorted); result
|(St(h,t),presult,sorted) ->
  let infl,supl = partition order h t in
  let_var r in (qsortrec(supl,r,sorted);
              qsortrec(infl,presult,St(h,r)))
 in qsortrec (l,undef,Nil) ;;
Value quicksort:('a*'a->bool)->'a stream->'a stream
```

The following example illustrates the use of potentially infinite lists and demand driven computation. The confluence property allows to parallelize the evaluation of nested applications in the definition of the Hamming sequence of integers of the form $2^i * 3^j * 5^k$[Dijkstra 76].

```
#let mult (P x,P y) = P(x*y);;
Value mult : int partial * int partial -> int partial
#let rec times (u,St(v,r)) = St(mult(u,v),times(u,r));;
Value times:
 int partial*int partial stream->int partial stream
#let rec merge (St(P x,s),St(P y,r)) =
 if x<y then St(P x,merge (s,St(P y,r))) else
 if x>y then St(P y,merge (St(P x,s),r)) else
 St(P x, merge(s,r));;
Value merge: int partial stream*int partial stream ->
              int partial stream
#let rec copy_stream (St(a,b)as s) (St(h,t)) =
  unif a h; copy_stream b t; s;;
Value copy_stream : 'a stream -> 'a stream -> 'a stream
```

[6] The author has traduced all programs given by G.Huet in [Huet 88], he found that the use of types and of a functional formalism lead to more clear programs.

```
#let Hamming = let_var r in
 copy_stream
  (St(P 1,merge(merge(times(P 2,r),times(P 3,r)),
              times(P 5,r)))) r;
 r;;
Value Hamming : int partial stream  Hamming = ?
#let rec increase_stream st = function
   0 -> st
 | n -> let_var tail in unif st St(undef,tail);
        increase_stream tail (n-1) ;;
Value increase_stream : 'a? stream -> int -> 'a? stream
#increase_stream Hamming 9; Hamming;;
Value - : int partial stream
- =St(P 1,St(P 2,St(P 3,St(P 4,St(P 5,St(P 6,St(P 8,
     St(P 9,St(P 10,?)))))))))
```

B Reduction rules

Pair1F $$\frac{< t,\sigma,\alpha,\Gamma > \rightarrow < failwith(s),\sigma,\alpha,\Gamma >}{< (t,t'),\sigma,\alpha,\Gamma > \rightarrow < failwith(s),\sigma,\alpha,\Gamma >}$$

Pair2F $$\frac{< t',\sigma,\alpha,\Gamma > \rightarrow < failwith(s),\sigma,\alpha,\Gamma >}{< (t,t'),\sigma,\alpha,\Gamma > \rightarrow < failwith(s),\sigma,\alpha,\Gamma >}$$

Pair1 $$\frac{< t,\sigma,\alpha,\Gamma > \rightarrow < t_1,\sigma_1,\alpha_1,\Gamma_1 >}{< (t,t'),\sigma,\alpha,\Gamma > \rightarrow < (t_1,t'),\sigma_1,\alpha_1,\Gamma_1 >}$$

Pair2 $$\frac{< t',\sigma,\alpha,\Gamma > \rightarrow < t'_1,\sigma_1,\alpha_1,\Gamma_1 >}{< (t,t'),\sigma,\alpha,\Gamma > \rightarrow < (t,t'_1),\sigma_1,\alpha_1,\Gamma_1 >}$$

Figure 1: Structural rules

We assume that we have a function *queue* such that $queue_{\sigma,\alpha}u(i)$ returns all the suspensions in α waiting for instantiation of $u(i)$. The rule **DVar** uses a counter c that is increased each time a new logical variable is created. c is initially at 1. The rules **Susp** and **USusp** use an other counter c_s dedicated to suspensions also initially at 1, they increase α with the new suspension. The rules **UnifT** and **AwUpd** increase σ with the new bindings and increase Γ with the suspensions waiting for these instantiations or update. Note that we remain free to choose the order of evaluation of binary constructs as for $E\Lambda$ (We give in figure 1 the rules for pairs, rules for unification and application are similar.). Moreover, the order of evaluation of terms bound in Γ is also free (see rule **Aw**).

C Demonstration of theoreme 3

Let us give preliminary results.

Lemma 3 *If* $< t,\sigma,\alpha,\Gamma > \rightarrow < t',\sigma',\alpha',\Gamma' >$ *by application of a suspension rule then* $< t,\sigma,\alpha,\Gamma > \equiv < t',\sigma',\alpha',\Gamma' >$

Proposition 3 *If* $< t_1,\sigma_1,\alpha_1,\Gamma_1 > \rightarrow < t'_1,\sigma'_1,\alpha'_1,\Gamma'_1 >$ *by application of a rule distinct of a suspension rule, and if* $< t_1,\sigma_1,\alpha_1,\Gamma_1 > \equiv < t_2,\sigma_2,\alpha_2,\Gamma_2 >$ *then we have* $< t'_2,\sigma'_2,\alpha'_2,\Gamma'_2, >$ *such that* $< t_2,\sigma_2,\alpha_2,\Gamma_2 > \rightarrow < t'_2,\sigma'_2,\alpha'_2,\Gamma'_2 >$ *and* $< t'_1,\sigma'_1,\alpha'_1,\Gamma'_1 > \equiv < t'_2,\sigma'_2,\alpha'_2,\Gamma'_2 >$

Proof: We carefully discuss one case, others are similar:

Env $< x.(x,t) :: \_,\sigma,\alpha,\Gamma > \rightarrow < t,\sigma,\alpha,\Gamma >$

Env0 $< x.(y,t) :: e,\sigma,\alpha,\Gamma > \rightarrow < x.e,\sigma,\alpha,\Gamma >$

Const $< c.e,\sigma,\alpha,\Gamma > \rightarrow < c,\sigma,\alpha,\Gamma >$

AEnv $< (t\ t').e,\sigma,\alpha,\Gamma > \rightarrow < (t.e\ t'.e),\sigma,\alpha,\Gamma >$

UEnv $< (unif\ t\ t').e,\sigma,\alpha,\Gamma > \rightarrow < (unif\ t.e\ t'.e),\sigma,\alpha,\Gamma >$

PEnv $< (t,t').e,\sigma,\alpha,\Gamma > \rightarrow < (t.e,t'.e),\sigma,\alpha,\Gamma >$

DVar $< undef.e,\sigma,\alpha,\Gamma > \rightarrow < u(c),\sigma,\alpha,\Gamma >$ and $c \leftarrow (c+1)$

β $$\frac{\begin{array}{l} < t,\sigma,\alpha,\emptyset > \text{ is in } \rightarrow \text{ normal form} \\ \sigma^*(f) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e, \\ \Phi s_\sigma(1,[p_i],t) = i,e_i \end{array}}{< f\ t,\sigma,\alpha,\Gamma > \rightarrow < a_i.e_i\ @\ e,\sigma,\alpha,\Gamma >}$$

Susp $$\frac{\begin{array}{l} < t,\sigma,\alpha,\Gamma > \text{ is in } \rightarrow \text{ normal form. } c_s\text{=k} \\ \sigma^*(f) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e, \\ \Phi s_\sigma(1,[p_i],t) = Unknown \end{array}}{\begin{array}{l} < f\ t,\sigma,\alpha,\Gamma > \rightarrow < u(-n),\sigma,(u(-n),\sigma^*(f)\ t) :: \alpha,\Gamma > \\ \text{and } c_s \leftarrow (k+1) \end{array}}$$

ASusp $$\frac{\begin{array}{l} < t,\sigma,\alpha,\Gamma > \text{ is in } \rightarrow \text{ normal form. } c_s\text{=n} \\ \sigma^*(f) = u(i) \end{array}}{\begin{array}{l} < f\ t,\sigma,\alpha,\Gamma > \rightarrow < u(-n),\sigma,(u(-n),u(i)\ t) :: \alpha,\Gamma > \\ \text{and } c_s \leftarrow (n+1) \end{array}}$$

Fail $$\frac{\begin{array}{l} < t,\sigma,\alpha,\emptyset > \text{ is in } \rightarrow \text{ normal form. } \Phi s_\sigma(1,[p_i],t) = fail \\ \sigma^*(f) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e, \end{array}}{< f\ t,\sigma,\alpha,\Gamma > \rightarrow < failwith(Pattern),\sigma,\alpha,\Gamma >}$$

UnifT $$\frac{\begin{array}{l} < t,\sigma,\alpha,\emptyset > \text{ and } < t',\sigma,\alpha,\emptyset > \text{are in } \rightarrow \text{ normal form} \\ unif_\sigma(t,t') = \sigma' \\ \text{Let } L = \emptyset \text{ if } \sigma' = \sigma \text{ or } \sigma'(u(i)) = u(j) \\ for\ all\ u(i) \in dom(\sigma')\backslash dom(\sigma) \\ \text{and } L = queue_{\sigma,\alpha}(u(i)) \text{ in other cases} \end{array}}{< unif\ t\ t',\sigma,\alpha,\Gamma > \rightarrow < void,\sigma',\alpha\backslash L >, L \cup \Gamma >}$$

UnifF $$\frac{\begin{array}{l} < t,\sigma,\alpha,\emptyset > \text{ and } < t',\sigma,\alpha,\emptyset > \\ \text{are in } \rightarrow \text{ normal form} \\ unif_\sigma(t,t') = fail \end{array}}{< unif\ t\ t',\sigma,\alpha,\Gamma > \rightarrow < failwith(Unif),\sigma,\alpha,\Gamma >}$$

USusp $$\frac{\begin{array}{l} < t,\sigma,\alpha,\emptyset > \text{ and } < t',\sigma,\alpha,\emptyset > \\ \text{are in } \rightarrow \text{ normal form} \\ unif_\sigma(t,t') = susp(u(i)), \quad c_s = \text{n} \end{array}}{\begin{array}{l} < unif\ t\ t',\sigma,\alpha,\Gamma > \rightarrow \\ \quad < u(-n),\sigma,(u(-n),unif\ t\ t') :: \alpha,\Gamma > \\ \text{and } c_s \leftarrow (n+1) \end{array}}$$

Aw $$\frac{\begin{array}{l} u(i) \in dom(\Gamma) \text{ and } \Gamma(u(i)) = t \\ < t,\sigma,\alpha,\emptyset > \rightarrow < t',\sigma',\alpha',\emptyset > \\ \text{and } < t',\sigma',\alpha',\emptyset > \text{ not in normal form} \end{array}}{< t_0,\sigma,\alpha,\Gamma > \rightarrow < t_0,\sigma',\alpha',\Gamma[u(i) \leftarrow t'] >}$$

AwUpd $$\frac{\begin{array}{l} u(i) \in dom(\Gamma) \text{ and } \Gamma(u(i)) = t \\ < t,\sigma,\alpha,\emptyset > \rightarrow < t',\sigma',\alpha',\Gamma'' > \\ \text{and } < t',\sigma',\alpha',\emptyset > \text{ is in normal form} \\ \Gamma' = queue_{\sigma,\alpha}(u(j)) \end{array}}{\begin{array}{l} < t_0,\sigma,\alpha,\Gamma > \rightarrow \\ < t_0,(u(j),t') :: \sigma',\alpha'\backslash\Gamma',\Gamma'' \cup \Gamma' \cup \Gamma\backslash\{(u(j),t)\} > \end{array}}$$

AwFail $$\frac{\begin{array}{l} u(i) \in dom(\Gamma) \text{ and } \Gamma(u(i)) = t \\ < t,\sigma,\alpha,\emptyset > \rightarrow < failwith(s),\sigma,\alpha,\emptyset > \end{array}}{< t_0,\sigma,\alpha,\Gamma > \rightarrow < failwith(s),\sigma,\alpha,\Gamma >}$$

Let $< t_1, \sigma_1, \alpha_1, \Gamma_1 >$ be reduced by β applied on a subterm of t_1. Let note that subterm $(fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e\ v$. By the hypothesis of $\equiv$ we have $(\sigma_2 \circ \alpha_2 \circ \Gamma_2)^*(t_2) = t_1$, thus the corresponding subterm of t_2 is of one of the following forms: $u(i)$; $u(i)\ u(j)$; $(fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e\ w$. We examine the first two forms:

(1): $u(i)$. First as σ_2 binds variable with values, we have $\sigma_2^*(u(i)) = u(j)$ and $u(j) \notin dom(\sigma_2)$. The $\equiv$ hypothesis ensures that $u(j) \notin dom(\alpha_2)$ as in that case the application would be suspended when the rule β applies on t_1. Thus we have: $\sigma_2^*(\Gamma_2(u(j))) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e\ v$. The $\equiv$ hypothesis ensures that the same pattern matchs in both reduction and then application of **Aw** with the rule β on that term clearly leads to an equivalent four_uple.

(2) $u(i)\ u(j)$. The fact that bindings in α_2 and Γ_2 are bindings of logical variable to non value terms ensure that $\sigma_2^*(u(i)) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e$ and $\sigma_2^*(u(j)) = v$; then β applies on $u(i)\ u(j)$ and leads to an equivalent four_uple.⋄ We have now the result of strong confluence of $\rightarrow$ up to $\equiv$,

Theorem 5 *For all* $< t, \sigma, \alpha, \Gamma >$ *such that:*
$< t, \sigma, \alpha, \Gamma > \rightarrow < t_1, \sigma_1, \alpha_1, \Gamma_1 >$
$< t, \sigma, \alpha, \Gamma > \rightarrow < t_2, \sigma_2, \alpha_2, \Gamma_2 >$
There exists $< t_1', \sigma_1', \alpha_1', \Gamma_1' >$ *and* $< t_2', \sigma_2', \alpha_2', \Gamma_2' >$ *such that*

$$< t_1, \sigma_1, \alpha_1, \Gamma_1 > \xrightarrow{0,1} < t_1', \sigma_1', \alpha_1', \Gamma_1' >$$
$$< t_2, \sigma_2, \alpha_2, \Gamma_2 > \xrightarrow{0,1} < t_2', \sigma_2', \alpha_2', \Gamma_2' >$$
$$< t_1', \sigma_1', \alpha_1', \Gamma_1' > \equiv < t_2', \sigma_2', \alpha_2', \Gamma_2' >$$

Proof: it is illustrated in figure 3. The cases where at least one reduction use a suspension rule are: if both r_1 and r_2 use suspension rules, then the lemma 3 is enough to conclude. If one r_i use a suspension rule, then we conclude with the proposition 3 and the lemma 3.⋄

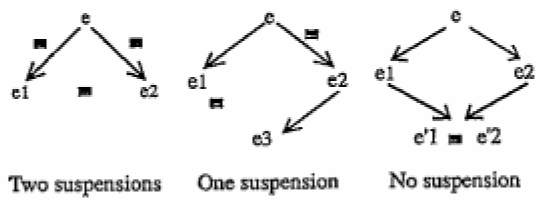

Figure 3: Strong confluence

Proof of the theorem: We show that the diagram of figure 4 holds with the theorem above and by successive inductions on lengths of d_1 and d_2.⋄

Remark that the limitation to a strict calculus is necessary. If we permit reducing application without reducing the argument, as some unification may occur in that argument different normal forms are possible. Example:

$$< (fun\ (x, y) \rightarrow unif\ x\ True).[](u(1), unif\ u(1)\ False), \emptyset, \emptyset, \emptyset >$$

has two normal forms:

$$< void, \{(u(1), True)\}, \emptyset, \emptyset >$$

and $< failwith(Unif), \{(u(1), False)\}, \emptyset, \emptyset >$.

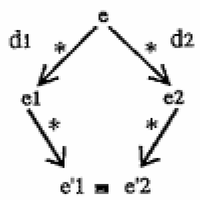

Figure 4: Church Rosser property

References

[AbadiCaCuLe 90] M.Abadi, L Cardelli, P.L. Curien, J.J. Levy, "Explicit substitutions", Proc. Symp. POPL 1990

[Ait Kaci 89] H.Ait-Kaci, R. Nasr, "Integrating Logic and Functional Programming", Lisp and Symbolic Computation,2,51-89 (1989)

[DeGrootLindstrom 86] D DeGroot G Lindstrom (eds),"Logic Programming - Functions, Relations and Equations", Prentice-Hall, New-Jersey, 1986.

[Dijkstra 76] E.W. Dijkstra, " A Discipline of Programming", Prentice Hall, New Jersey, 1976.

[HardinLevy 90] T. Hardin, J.J. Levy, "A Confluent Calculus of Substitutions", third symposium (IZU) on I.A.

[Huet 76] G.Huet, "Résolution d'équations dans les langages d'ordre 1,2,...,ω" Thèse d'état de l'Univ. de Paris 7, 1976

[Huet 88] G. Huet, "Experiments with GHC prototypes", may 1988, unpublished

[Laville 88] Alain Laville, "Implementation of Lazy Pattern Matching Algorithms", ESOP'88,LNCS 300

[Leroy 90] X.Leroy, "The ZINC experiment: an economical implementation of the ML language", INRIA technical report 117, 1990.

[LeroyWeis 91] X.Leroy P.Weis, "Polymorphic type inference and assignment", "Principles of Programming Languages", 1991.

[Poirriez 91] V.Poirriez, "Intégration de fonctionnalités logiques dans un langage fonctionnel fortement typé: MLOG une extension de ML" Thèse, Univ. Paris 7, 1991

[Poirriez 92a] Vincent Poirriez, "FLUO: an implementation of MLOG", Fifth Nordic Workshop on Programming Languages in Tampere, 1992

[SatoSakurai 86] M.Sato et T.Sakurai "QUTE: a functional Language Based on Unification". In [DeGrootLindstrom 86] pp 131-155.

[PuelSuarez 90] A. Suarez and L.Puel, "Compiling pattern matching by term decomposition", LFP'90

[Ueda 86] K. Ueda, " Guarded Horn Clauses", Ph.D.Thesis, Information Engineering Course, Univ. of Tokyo, 1986.