MLOG: a strongly typed confluent functional language with logical variables

V. Poirriez
Institut des Sciences et Techniques de Valenciennes, Université de Valenciennes, B.P. 311, 59304 Valenciennes Cedex, 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. Although the evaluation strategy for the application is fixed, the order for the 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 worthwhile 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 Leroy [9]. 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 synchronization 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: 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 the variables without risking clashes. A complete operational semantics is given in the appendix.

We define SAE as the subset of our calculus limited to the functional part of the rules. SAE is a strict \(\lambda\)-calculus with explicit substitutions, named variables and pattern matching 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 \(\lambda\)-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 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 \(\lambda\)-calculus with pattern-matching, concrete types (either built-in, as \textit{int} or \textit{string}, or declared by the user), constructors, the \texttt{let} construct and the conditional. We first define the set \(P\) of programs of MLOG. We assume the existence of a countable set \(\text{Var}\) of term variables, with typical elements \(x, y, z\), 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
\[
p ::= x \mid c \mid (p_1, \ldots, p_n) \mid cp.
\]
As in ML, we limit ourselves to linear patterns. The syntax of programs, with
typical elements \( a, b \), is
\[
a ::= x \mid c \mid ab \mid (a_1, \ldots, a_n) \mid \text{let } x=a \text{ in } b \mid a; b
\]
\[
\quad \text{(function } p_1 \rightarrow a_1 \mid \cdots \mid p_n \rightarrow a_n) \mid \text{undef} \mid \text{unif}
\]
where \( a; b \) 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: \text{undef} is a
generator of fresh logical variables; \text{unif} is the unification primitive. \text{let_var } u \text{ in }
... is syntactic sugar for \text{let } u=\text{undef} \text{ in } ...

2.2. Types

In MLOG, the programmer has to declare specially the types that may contain un-
defined objects (i.e., logical variables and suspensions). The notion of \textit{logical type} is
introduced. We assume that given a countable set of type variables \( TVar \), with typi-
cal 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 ele-
ments \( \text{ident} \) and \( \text{lident} \). The sets of logical types \( L \), with typical element \( \tau_i \), and types
\( T \) (typical element \( t_i \)) are recursively defined by:
\[
\tau_i ::= 'a? \mid [t_i] \text{ident}
\]
and
\[
t_i ::= \tau_i \mid 'a \mid \text{bool} \mid \text{int} \mid \text{string} \mid \text{unit} \mid t_i \rightarrow t_j \mid t_i * t_j \mid [t_i] \text{ ident}
\]
Note that \( L \) is a strict subset of \( T \).

A logical type is declared by the new keyword: \textit{type logic}. The type \( \text{void} \) below
has a unique value \( \text{void} \) and logical variables of type \( \text{void} \) may be declared. The type
\( \text{void} \) is isomorphic to the type \( \text{unit} \) except that no logical variable can be declared
in unit. A value of the type \( \text{Bool} \) below is \( \text{True} \), \( \text{False} \), or a free logical variable
that will possibly be instantiated later to either \( \text{True} \) or \( \text{False} \):

\[
# \text{ type logic void = void;;}
\quad \text{Type void defined.}
# \text{ type logic Bool = True | False;;}
\quad \text{Type Bool defined.}
\]
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; and (3) \( 'a? \)
may not be instantiated by a nonlogical type.
We write as "\( a : t_i \)" the program \( a \) of type \( t_i \). Thus, the set of MLOG programs is in fact the subset of the well-typed programs \( \mathcal{P}_T \) of \( P \) defined by the familiar ML type system. We just have to specify that: (1) undef: 'a?'; and (2) unif: 'a →' a → void. Fortunately, as far as types are concerned, logical variables and assignable constructs are quite close, we have adapted logical variables of previous work done for typing assignable objects in ML. We have directly applied the idea of Leroy and Weis [10], and, using their notion of cautious generalization, we get the following extension of the ML type system to logical variables that is sound:

**Theorem 2.1.** No evaluation of a well-typed program can lead 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:

```plaintext
# 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:

```plaintext
# 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 A 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:

```ml
# let neg = function True + False | False + True;;
Value neg: Bool = False
# let (b, exp) = let_var u in (u, neg u);;
Value b: Bool = ? 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. A suspension is a first class citizen in MLOG. It may be handled in data structures, and used in other expressions.

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

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

```ml
# 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:

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

e is suspended waiting for the instantiation of b:

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

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 for a:

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

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 nonlogical type.
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. In Appendix A we give two programs which illustrate other possible uses of the suspension mechanism. The hamming program shows how 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. It also illustrates the ability to define recursively data structures even in a way that is not allowed in a nonlazy functional language. That program also uses demand-driven computation. The Abraham's descendance program (see Appendix A) illustrates how it is possible to compute a data structure, leaving holes because information is missing, and to plug these holes later when the information is available with no need to compute all the structure again.

3. A confluence result

To give an operational semantics for MLOG we have to deal with bindings of \( \lambda \)-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 \( EA \) the set of terms with environments. As our calculus is strict, we specialize a subset \( Val \) of \( EA \) which is the set of the values handled by the language. Typical elements of \( Val \) and \( EA \) are respectively noted \( v \) and \( t \):

\[
\begin{align*}
  e &::= \mathbb{[]} \mid (x,v) :: e \\
  v &::= c \mid c(v) \mid (v,v') \mid (\text{function} \ldots).e \\
  t &::= v \mid c(t) \mid (t,t') \mid t(t') \mid a.c
\end{align*}
\]

(\text{environment})

\( (Val) \)

\( (EA) \)

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 \( ELA \) the obtained sets of values and terms with environments. To manage the bindings of logical variables, we define substitutions as functions from \( U \) to \( ELA \). We will use greek letters to note substitu-

\(^3\) Recall that if no strategy for application is imposed, name clash may occur. To avoid that problem, the names of variables can be replaced by numbers "à la De Bruijn" [1, 5].
A strongly typed confluent functional language with logical variables

tions. We call the domain of $\sigma$ and note $\text{dom}(\sigma)$ 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 access 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 $\text{pat}$ in the substitution $\sigma$ and denotes $\Phi_{\sigma}(\text{pat}, t)$ the list of appropriate bindings of parameters of pat. Recall that patterns are linear. We define now a sequential pattern matching $\Phi_{\sigma}$ without entering into the optimization of the algorithm.\(^4\)

The list of patterns with the head $p_0$ and the tail $p_l$ is denoted by $p_0 :: p_l$:

\[
\text{if } \Phi_{\sigma}(p_0, t) = e \text{ then } \Phi_{\sigma}(i, p_0 :: p_l, t) = (i, e),
\]

\[
\text{if } \Phi_{\sigma}(p_0, t) = \text{Unknown then } \Phi_{\sigma}(i, p_0 ::_, t) = (i, \text{Unknown}),
\]

\[
\text{if } \Phi_{\sigma}(p_0, t) = \text{fail then } \Phi_{\sigma}(i, p_0 :: [], t) = (i, \text{fail}),
\]

\[
\text{if } \Phi_{\sigma}(p_0, t) = \text{fail and } p_l \neq [] \text{ then if } \Phi_{\sigma}(i, p_0 :: p_l, t) = \Phi_{\sigma}(i+1, p_l, t).
\]

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 the 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 $\alpha$ (see rules Susp and ASusp in Fig. 2 in the appendix). 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 $\sigma$ 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 $\text{dom}(\sigma)$ already met, $\sigma^*$ is defined by

\[
\sigma^* = \emptyset \vdash \sigma^* \text{ and }
\]

\[
M \vdash \sigma^*(u(i)) = u(i) \text{ if } u(i) \in M \text{ or } u(i) \notin \text{dom}(\sigma),
\]

\[
M \vdash \sigma^*(u(i)) = (\{u(i)\} \cup M) \vdash \sigma^*(\sigma(u(i))) \text{ if } u(i) \notin M,
\]

\[
M \vdash \sigma^*(c) = c,
\]

\[
M \vdash \sigma^*(t(t')) = (M \vdash \sigma^*(t))(M \vdash \sigma^*(t'))
\]

\[
M \vdash \sigma^*(t, t') = (M \vdash \sigma^*(t), M \vdash \sigma^*(t'))
\]

\[
M \vdash \sigma^*(p.e) = (M \vdash \sigma^*(p), M \vdash \sigma^*(e)).
\]

\(^4\) The interested reader is referred to [8, 14] 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.
3.3. Unification

The used unification procedure is adapted from [6]. 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 $u(j)$ with any other term, it stops and returns $\text{ susp}(u(j))$.

(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 $\text{ unif}_{0\sigma_0}(t, t')$ by:

(a) $\text{ unif}_{0\sigma_0}(t, t') = \sigma$ iff the unification procedure applied to $\{(t, t')\}$ with the initial substitution $0\sigma_0$ succeeds and builds the substitution $\sigma$.

(b) $\text{ unif}_{0\sigma_0}(t, t') = \text{ fail}$ iff the unification procedure applied to $\{(t, t')\}$ with the initial substitution $0\sigma_0$ stops with $\text{ fail}$.

(c) $\text{ unif}_{0\sigma_0}(t, t') = \text{ susp}(u(j))$ iff the unification procedure applied to $\{(t, t')\}$ with the initial substitution $0\sigma_0$ stops with $\text{ susp}(u(j))$.

The following result holds.

**Theorem 3.1.** For all terms $t, t'$ $\text{ unif}_{0\sigma_0}(t, t')$ terminates and:

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

3.4. Confluence of the reduction over ELA

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 $\text{ ELA} \times \text{ substitutions} \times \text{ substitutions}$ that verifies the rules given in Figs. 3-5 in Appendix B. A 4-tuple is denoted by $(t, \sigma, \alpha, \Gamma)$, where $t$ is the term to reduce. The substitution $\sigma$ stores the bindings of unified logical variables and updated suspensions. The valuation $\alpha$ stores the suspensions (recall they are bound to $u(j)$ with $j < 0$). The substitution $\Gamma$ stores the suspensions of which evaluations are running.

We use the classical notation $\xrightarrow{\ast}$ 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.

\[ \text{susp} \] is returned even if the procedure has to unify a free logical variable and a suspension.
Lemma 3.2. Let \( a \) be a program and \( \langle a[\cdot], \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\Rightarrow} \langle t, \sigma, \alpha, \Gamma \rangle \). For all subterms of \( t \) of the form \( t'.e \), \( t' \) is a program.

**Proof.** By induction over the length of the derivation, the case \( n = 1 \) results directly from the rules Const, AEnv, UEnv and PEnv.

We assume now that the lemma is true for all \( k \) less than \( n \). An easy discussion on the last rule used in the derivation leads to the result.  □

Lemma 3.3. Let \( a \) be a program and \( \langle a[\cdot], \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\Rightarrow} \langle t, \sigma, \alpha, \Gamma \rangle \) such that \( \langle t, \sigma, \alpha, \Gamma \rangle \) is a normal form. Then \( t \) is a value.

**Proof.** Simply recall that the set of values \( Val \) is the subset of \( EA \) of all the terms with no subterms of the form \( (t, t') \) or \( t = t' \) out of the scope of a function, and with no subterm of the form \( t.e \), where \( t \) is not a function. If \( \langle t, \sigma, \alpha, \Gamma \rangle \) is a normal form, it means that no rule is applicable. Then no subterm of \( t \) is an application or an unification out of the body of a function, there is also no subterm of the form \( a.e \) with \( a \) distinct from a function. □

We can deduce from these lemmas that all bindings in \( \sigma \) bind a variable with a value.

Let us look now at the confluence of \( \rightarrow \). First we show the strong confluence of \( \rightarrow \) if we exclude the suspensions.

**Proposition 3.4.** Let \( \langle t, \sigma, \alpha, \Gamma \rangle \rightarrow \langle t_1, \sigma_1, \alpha, \Gamma_1 \rangle \) and \( \langle t, \sigma, \alpha, \Gamma \rangle \rightarrow \langle t_2, \sigma_2, \alpha, \Gamma_2 \rangle \) two reduction using, respectively, the distinct rules \( r_1 \) and \( r_2 \) with \( r_i \) not a suspension rule. Then we have by the application of, respectively, \( r_2 \) and \( r_1 \): \( \langle t_1, \sigma_1, \alpha, \Gamma_1 \rangle \rightarrow \langle t_3, \sigma_3, \alpha, \Gamma_3 \rangle \) and \( \langle t_2, \sigma_2, \alpha, \Gamma_2 \rangle \rightarrow \langle t_3, \sigma_3, \alpha, \Gamma_3 \rangle \).

**Proof.** As the calculus is strict, no reduction is performed under a function, \( (a\lambda) \). Then if \( t \) is of the form \( a.env \), then by Lemma 3.2, no more than one rule is applicable. If \( t \) is of the form \( t(t') \), then \( t \) is a closure and \( t' \) is a value, then only the rule \( \beta \) applies; or the reductions are done one in \( t \) and the other in \( t' \) and then they are separate; or both reductions are done in the same subterm, then we can conclude with an induction over the number of applications and unifications in the term. The discussion for the other possibilities for \( t \) is similar. □

An important corollary of this result is that if we restrict ourselves to the functional subset of MLOG, we have to describe a strong confluent calculus with explicit substitutions, named variables and pattern matching. Let us call this subset \( SAE \); its rules are given in Fig. 6 (Appendix C).
That calculus is rather simple (all that concerns logical variables and suspensions is unnecessary) and describes all implementations of a strict $\lambda$-calculus, even a parallel one.

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

$$\langle((\text{function } c \rightarrow c'),[]) u(1), \text{unif } u(1) c), \theta, \theta \rangle.$$ We can see the use of a rule $\text{Susp}$, $\text{ASusp}$ or $\text{USusp}$ as the translation of a subterm from the term to $\Gamma$. From a reduction point of view we can say that these rules do not work. Thus, the idea is to define an equivalence between 4-tuples $\langle t, \sigma, \alpha, \Gamma \rangle$ which is stable for these suspension rules and then show the strong confluence of $\rightarrow$ up to that equivalence.

**Definition 3.5.** $\langle t, \sigma, \alpha, \Gamma \rangle \equiv \langle t', \sigma', \alpha', \Gamma' \rangle$ iff

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

Thus, we have verified the Church–Rosser property (the proof is in Appendix C):

**Theorem 3.5.** If $\langle t, \sigma, \alpha, \Gamma \rangle$ has a normal form for $\rightarrow$ then it is unique up to $\equiv$.

Remark that if we add types as defined in the section above, the rules do not have to be modified and the result holds.

4. MLOG: a conservative extension of ML

The fact that the type of $\text{undef}$ is $'a$? ensures that no logical variable occurs in a nonlogical type. That is not enough to ensure that no suspension of a nonlogical 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 nonlogical 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 (i) first, test if the argument is a bound logical variable or an updated suspension, and access the bound value; (ii) if type $t_2$ is a nonlogical type, test if the argument is a free variable or a suspension. If so, raise failure $\text{Unknown}$;
(iii) 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:

```ocaml
# type logic 'a partial = P of 'a;;

Type partial defined.

# (function (P x) \rightarrow x) undef;;

uncaught exception Unknown.
```

**Theorem 4.1.** Let \( a \) be a well-typed program. The evaluation of \( a \) cannot build a logical variable or a suspension of a nonlogical 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 longer Church–Rosser. To keep that property, we must not use functions from a logical type to a nonlogical type. Let call MLOG* the subset of MLOG that does not contain such functions. Thus, we have the following result.

**Proposition 4.2.** 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 the type \( (int \rightarrow int)\text{partial} \) as illustrated below.

```ocaml
#let app (P h) (P x) = P (h x);;

Value app : ('a \rightarrow 'b) partial \rightarrow ('a partial \rightarrow 'b partial)

#let (g : (int \rightarrow int) partial) = undef;;

Value g : (int \rightarrow int) partial g = ?

#let e2 = app g (P 2);;

Value e2 : int partial e2 = ...

#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 stimulate parallel processes. Such processes can communicate with each other through shared logical variables and the suspension mechanism allows synchronization.

MLOG includes a suspension mechanism, let us now compare it with some other proposals of integration that have made a similar choice. MLOG is close to the language
Qute defined by Sato and Sakurai [13]. 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 the needed information is provided. (2) The reduction of an application is allowed even if a subexpression of the argument is suspended, the only condition being that pattern matching succeeds. Then the binding of the suspension by a logical variable and the storage in \( \alpha \) avoid the duplication of that suspension.

MLOG is also close to GHC of Ueda [15]. The main difference (except from the typing point of view) is that MLOG does not have nondeterminism 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 Ait Kaci [2] 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 \( \lambda \)-calculus which uses names for parameters, has pattern matching and verifies the Church–Rosser property. Therefore, it can be used to describe any interpreter of strict \( \lambda \)-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 Ueda [15] 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.

Appendix A: MLOG programs

A.1. Quicksort

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.

\[
\begin{align*}
\texttt{#let partition order x =} \\
\texttt{let rec partrec = function}
\end{align*}
\]

\(^6\) The author has translated all programs given by Huet [7], he found that the use of types and of a functional formalism lead to more clear programs.
A strongly typed confluent functional language with logical variables

\[
\text{Nil} \rightarrow \text{Nil}, \text{Nil}\\
\text{St}(h, t) \rightarrow \text{let inf}, \text{supl} = \text{partrec} \ t \ \text{in} \\
\text{if order}(h, x) \ \text{then} \ \text{St}(h, \text{inf}), \ \text{supl} \ \text{else} \ \text{inf}, \text{St}(h, \text{supl}) \\
\text{in} \ \text{partrec} ::
\]

Value partition:

\[
('a * 'b \to \text{bool}) \rightarrow 'b \to 'a \text{ stream} \to 'a \text{ stream} * 'a \text{ stream}
\]

\#let quicksort order 1 =

\[
\text{let rec qsortrec = function} \\
(\text{Nil}, \text{result}, \text{sorted}) \rightarrow (\text{unif result sorted}; \ \text{result}) \\
| (\text{St}(h, t), \text{result}, \text{sorted}) \rightarrow \\
\text{let inf}, \text{supl} = \text{partition order} \ h \ t \ \text{in} \\
\text{let var} \ r \ \text{in} \ (\text{qsortrec}(\text{supl}, r, \text{sorted}); \\
(\text{qsortrec}(\text{inf}, \text{result}, \text{St}(h, r)))) \\
\text{in} \ \text{qsortrec} (1, \text{undef}, \text{Nil});
\]

Value quicksort:

\[
('a * 'a \rightarrow \text{bool}) \rightarrow 'a \text{ stream} \rightarrow 'a \text{ stream}
\]

A.2. Hamming

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 \(2^i \cdot 3^j \cdot 5^k\) [4]. The suspension mechanism is used in the evaluation of Hamming. The first place it is used is in the application of copy-stream. The function copy-stream takes two streams in arguments. In the first evaluation, the second argument \(r\) is a free logical variable. Then the application of copy-stream is suspended. It will be woken up by the evaluation of the next MLOG sentence: increase_stream Hamming 9 which in a way allocates 9 boxes in the data structure \(r\). Our calculus is strict then the value of \(r\) is needed to evaluate times (\(P 2, r\)), at that time \(r\) is a free logical variable. That application is suspended as times (\(P 3, r\)), times (\(P 5, r\)) and the two applications of merge. Then the awakening of copy_stream instantiate \(r\) with a stream \(\text{St}(P 1, \ldots)\). The tail of stream is the more extern merge application which is suspended. That instantiation wakes up the suspensions waiting for the “first box” of \(r\). Then the processes iterate while there is an allocated box in \(r\). When there is no more allocated box, it is the recursive call of copy_stream that is suspended. More computation of Hamming can be provided by a new call of increase_stream:

\#let mult (\(P x, P y\)) = \(P(x \cdot y)\);
Value mult: \(\text{int partial} * \text{int partial} \rightarrow \text{int partial}\)

\#let rec times (\(u, \text{St}(v, r)\)) = \(\text{St}(\text{mult}(u, v), \text{times}(u, r))\);
Value times:

\(\text{int partial} * \text{int partial stream} \rightarrow \text{int partial stream}\)

\#let rec merge (\(\text{St}(P x, s), \text{St}(P y, r)\)) =
\(\text{if x<y then} \ \text{St}(\text{P x}, \text{merge}(s, \text{St}(P y, r))) \ \text{else}\)
if \( x > y \) then \( \text{St}(y, \text{merge} (\text{St}(x, s), r)) \) else \( \text{St}(x, \text{merge}(a, r)) \);

Value merge : \text{int partial stream \* int partial stream \rightarrow 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 \rightarrow 'a stream \rightarrow 'a stream

#let Hamming = let_var r in
    copy_stream
    (St(P 1, \text{merge} (\text{merge} (\text{times}(P 2, r), \text{times}(P 3, r)),
    \text{times}(P 5, r)))) r;
Value Hamming : int partial stream

#let rec increase_stream st = function
    0 \rightarrow st
  | n \rightarrow let_var tail in unif st St(undef, tail);
    increase_stream tail (n-1) ;;
Value increase_stream : 'a? stream \rightarrow int \rightarrow '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, ?))))))))).

A.3. The descendance of Abraham

To establish the human descendance of Abraham (Figs. 1 and 2) is typically an open problem. We use here the suspension mechanism to leave holes in the descendance tree, these holes will be plugged when the corresponding information will be known. The suspension mechanism allow to do this without the necessity to perform again all the data structure.

In this program, we use the ML exception mechanism to handle the failure of the unification. This allow us to implement a function member which test in one crossing of the stream if its first argument is unifiable with an element of the stream, and the

Abraham
    ?
Isaac
    ?

Fig. 1. Abraham’s descendance 1.
unification is performed in that case, if no element of the stream is unifiable, then that term is added at the end of the stream:

```ocaml
#let rec member a st =
try (unif st (St(a,undef));True) with
Unify -> (match st with Nil -> False
| St(h,t) -> member a t));;
Value member : 'a -> 'a stream -> Bool
member = (fun)
#member (P 1) undef;;
-:Bool
- = True
#let s = St((P 1,undef),St((undef,True),undef));;
Value s:(int partial * Bool) stream
s = St ((P 1, ?), St ((?, True), ?))
#member (P 1, False) s;;
-:Bool
- = True
#s;;
-:(int partial * Bool) stream
- = St ((P 1, False), St ((?, True), ?))
#member (P 1,True) s ;;
-:Bool
- = True
#s;;
-: (int partial * Bool) stream
- = St ((P 1, False), St ((P 1, True), ?))
#member ((P 2,undef)) s ;;
-:Bool
- = True
#s;;
-: (int partial * Bool) stream
- = St ((P 1, False), St ((P 1, True), St ((P 2, ?), ?)))
```

Fig. 2. Abraham’s descendance II.
Here are the types that implement infinite trees and humanity and the initial register:

```ocaml
#type logic 'a infinite_tree = Node of 'a*
('a infinite_tree stream);;
Type infinite_tree defined.
#type logic human = Man of string | Woman of string;;
Type human defined.
#let register = St((Man "Cain", Man "Adam"), St((Man "Cain", Woman "Eve"), undef));;
Value register : (human * human) stream
register = St ((Man "Cain", Man "Adam"), St ((Man "Cain", Woman "Eve"), ?>)
```

The function after unifies its second argument with True if its first argument is unifiable with an element h of the stream it takes in third argument. The tail t of the stream is return:

```ocaml
#let rec after a found = function
Nil -> unif found False; Nil
| St(h,t) -> (try (unif a h; unif found True; t) with Unify + after a found t );;
Value after : 'a -> Bool -> 'a stream -> 'a stream
after = (fun)
```

Now we can give the principal function descendance which builds the descendance desc of the Human using the register:

```ocaml
#let rec desrec (Human,Descendance,Regist)=
let_var c, found in
let tail_reg = after (c, Human) found Regist in
match found with
  True ->
    let_var child_desc, tail_desc in
    (unif Descendance (St(Node(c) child_desc), tail_desc)) ;
  (match tail_reg with
   Nil -> (* no more child of Human *)
   desrec (c, child_desc,Regist); unif tail_desc Nil
   | _ -> (*perhaps other child *)
   desrec (c,child_desc,Regist); desrec
   (Human,tail_desc, tail_reg)))
| False -> (* child not found *)
  unif Descendance Nil ;;
Value desrec: 'a? * 'a? infinite_tree stream * ('a? * 'a?)
stream -> void
desrec = (fun)
```
#let descendance Human =
let desc in
desrec (Human,desc,register);Node(Human,desc);;
Value descendance : human → human infinite_tree
descendance = (fun)

And now if we provide information in the register, using the appropriate function
child, we can examine the improvement of the data structure named Abraham_desc.

#let child (c,p) = member (c,p) register;;
Value child:human * human → Bool
child = (fun)
= False

#let Abraham_desc = descendance (Man "Abraham'");;
Value Abraham_desc : human infinite_tree
= Node (Man "Abraham", St (Node (Man "Isaac", ?, ?), ?))

#child (Man "Isaac", Man "Abraham");;
child (Man "Zimran", Man "Abraham");;
child (Man "Nebayoth", Man "Ismael");;
= :Bool
= True

#Abraham_desc;;
- :human infinite_tree
= Node (Man "Abraham", St (Node (Man "Isaac", ?, ),
St (Node (Man "Ismael",
St (Node (Man "Nebayoth",?),?),),
St (Node (Man "Zimran",?), ?))))

Appendix B: reduction rules

Pair1F \[ \langle t, \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(s), \sigma, \alpha, \Gamma \rangle \]
\[ \langle (t, t'), \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(s), \sigma, \alpha, \Gamma \rangle \]

Pair2F \[ \langle t', \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(s), \sigma, \alpha, \Gamma \rangle \]
\[ \langle (t, t'), \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(s), \sigma, \alpha, \Gamma \rangle \]

Pair1 \[ \langle t, \sigma, \alpha, \Gamma \rangle \rightarrow \langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle \]
\[ \langle (t, t'), \sigma, \alpha, \Gamma \rangle \rightarrow \langle (t_1, t_1'), \sigma_1, \alpha_1, \Gamma_1 \rangle \]

Pair2 \[ \langle t', \sigma, \alpha, \Gamma \rangle \rightarrow \langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle \]
\[ \langle (t, t'), \sigma, \alpha, \Gamma \rangle \rightarrow \langle (t, t_1'), \sigma_1, \alpha_1, \Gamma_1 \rangle \]

Fig. 3. Structural rules.
The reduction rules are given in Fig. 3-5. Here are some remarks upon these rules:

- A typical trip through a 4-tuple for a suspended term is the following:
  - When $t$ is suspended, because the instantiation of a variable, say $u(i)$, is needed, then a link $(u(j), t)$ is put in $\alpha$. **Susp** or **USusp**.
  - Next, if $u(i)$ is instantiate, then $t$ can be woken up. Thus the link $(u(j), t)$ is transferred from $\alpha$ to $\Gamma$. **UnifT**.

\[
\begin{align*}
\text{Env} & \quad (x.(x, t) :: \epsilon, \sigma, \alpha, \Gamma) \rightarrow (t, \sigma, \alpha, \Gamma) \\
\text{Env0} & \quad (x.(y, t) :: e, \sigma, \alpha, \Gamma) \rightarrow (x.e, \sigma, \alpha, \Gamma) \\
\text{Const} & \quad (c.e, \sigma, \alpha, \Gamma) \rightarrow (c, \sigma, \alpha, \Gamma) \\
\text{AEnv} & \quad ((t \ t').e, \sigma, \alpha, \Gamma) \rightarrow ((t.e \ t').e), \sigma, \alpha, \Gamma) \\
\text{UEnv} & \quad ((\text{unif} t \ t').e, \sigma, \alpha, \Gamma) \rightarrow ((\text{unif} t.e \ t.e), \sigma, \alpha, \Gamma) \\
\text{PEnv} & \quad ((t, t').e, \sigma, \alpha, \Gamma) \rightarrow ((t.e, t'.e), \sigma, \alpha, \Gamma) \\
\text{DVar} & \quad ((\text{undef} e, \sigma, \alpha, \Gamma) \rightarrow (u(c), \sigma, \alpha, \Gamma) \\
& \quad \text{and } c \leftarrow (c + 1) \\
\phi & \quad (t, \sigma, \alpha, \emptyset) \text{ is in } \rightarrow \text{ normal form} \\
\phi & \quad \sigma^*(f) = (\text{fun} \ p_1 \rightarrow a_1 \ | \ \cdots \ | \ p_n \rightarrow a_n).e, \\
\phi & \quad \Phi_{\sigma}(1, pl, t) = i, e_i \\
\phi & \quad (f \ t, \sigma, \alpha, \Gamma) \rightarrow (a_i, e_i@e, \sigma, \alpha, \Gamma) \\
\phi & \quad (t, \sigma, \alpha, \Gamma) \text{ is in } \rightarrow \text{ normal form. } c_s = k \\
\phi & \quad \sigma^*(f) \rightarrow (\text{fun} \ p_1 \rightarrow a_1 \ | \ \cdots \ | \ p_n \rightarrow u_n).e, \\
\phi & \quad \Phi_{\sigma}(1, pl, t) = \text{Unknown} \\
\phi & \quad (f \ t, \sigma, \alpha, \Gamma) \rightarrow (u(-k), \sigma, (u(-k), \sigma^*(f) t) :: \alpha, \Gamma) \\
& \quad \text{and } c_s \leftarrow (k + 1) \\
\phi & \quad (t, \sigma, \alpha, \Gamma) \text{ is in } \rightarrow \text{ normal form. } c_s = n \\
\phi & \quad \sigma^*(f) = u(i) \\
\phi & \quad \Phi_{\sigma}(1, pl, t) = \text{fail} \\
\phi & \quad (f \ t, \sigma, \alpha, \Gamma) \rightarrow (\text{failwith(Pattern)}, \sigma, \alpha, \Gamma) \\
& \quad \text{and } c_s \leftarrow (n + 1)
\end{align*}
\]

Fig. 4. Environmental and application rules.
\langle t, \sigma, \alpha, \emptyset \rangle \text{ and } \langle t', \sigma, \alpha, \emptyset \rangle \text{ are in } \rightarrow \text{ normal form}

\text{unif}_\sigma(t, t') = \sigma'

\textbf{UnifT} \quad \text{Let } \text{inst} = \{u(i) \in \text{dom}(\sigma') \setminus \text{dom}(\sigma) \text{ such that } \sigma'(u(i)) \neq u(j)\}

\text{Let } L = \bigcup_{u(i) \in \text{inst}} \text{queue}_\alpha(u(i))

\langle \text{unif} t, t', \sigma, \alpha, \Gamma \rangle \rightarrow \langle \emptyset, \sigma', \alpha \setminus L, L \cup \Gamma \rangle

\langle t, \sigma, \alpha, \emptyset \rangle \text{ and } \langle t', \sigma, \alpha, \emptyset \rangle \text{ are in } \rightarrow \text{ normal form}

\text{unif}_\sigma(t, t') = \text{fail}

\langle \text{unif} t, t', \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(\text{Unif}), \sigma, \alpha, \Gamma \rangle

\langle t, \sigma, \alpha, \emptyset \rangle \text{ and } \langle t', \sigma, \alpha, \emptyset \rangle \text{ are in } \rightarrow \text{ normal form}

\text{unif}_\sigma(t, t') = \text{susp}(u(i)), c_\sigma = n

\langle \text{unif} t, t', \sigma, \alpha, \Gamma \rangle \rightarrow \langle u(-n), \sigma, (u(-n), \text{unif} t, t') :: \alpha, \Gamma \rangle

\text{and } c_\sigma \leftarrow (n + 1)

\langle t, \sigma, \alpha, \emptyset \rangle \rightarrow \langle t', \sigma', \alpha', \emptyset \rangle

u(i) \in \text{dom}(\Gamma) \text{ and } \Gamma(u(i)) = t

\langle t_0, \sigma, \alpha, \Gamma \rangle \rightarrow \langle t', \sigma', \alpha', \Gamma' \rangle

u(i) \in \text{dom}(\Gamma) \text{ and } \Gamma(u(i)) = t

\langle t, \sigma, \alpha, \emptyset \rangle \rightarrow \langle t', \sigma', \alpha', \Gamma'' \rangle

\langle t_0, \sigma, \alpha, \Gamma \rangle \rightarrow \langle t_0, (u(i), t') :: \sigma', \alpha' \setminus \Gamma'', \Gamma' \cup \Gamma \setminus \{(u(i), t)\} \rangle

u(i) \in \text{dom}(\Gamma) \text{ and } \Gamma(u(i)) = t

\langle t_0, \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(s), \sigma, \alpha, \emptyset \rangle

\langle t_0, \sigma, \alpha, \Gamma \rangle \rightarrow \langle \text{failwith}(s), \sigma, \alpha, \Gamma \rangle

\text{Fig. 5. Unification and awake rules.}

- While } t \text{ is not in a normal form, reduction rules are applied, and the link of } u(j) \text{ with the term remains in } \Gamma, \textbf{Aw}.

- When } t \text{ is evaluated in a value } v, \text{ then the link } (u(j), v) \text{ is transferred from } \Gamma \text{ to } \sigma \text{ and the suspensions that are waiting for that updating of } u(j) \text{ are transferred from } \alpha \text{ to } \Gamma', \textbf{AwUpd}.

The purpose of that trip is to allow the sharing of the evaluation of a suspended term by all the occurrences of that suspension in the global term.
• In rules $\beta$, $\text{Susp}$ and $\text{Fail}$, $pl$ is the list of patterns $p_1 \cdots p_n$.

• We assume that we have a function $\text{queue}$ such that $\text{queue}_\alpha u(i)$ returns all the suspensions in $\alpha$ waiting for instantiation of $u(i)$.

• The rule $\text{DV}ar$ uses a counter $c$ that is increased each time a new logical variable is created. $c$ is initially at 1.

• The rules $\text{Susp}$ and $\text{USusp}$ use another counter $c_s$ dedicated to suspensions also initially at 1, they increase $\alpha$ with the new suspension.

• The rules $\text{UnifT}$ and $\text{AwUpd}$ increase $\sigma$ with the new bindings and increase $I'$ with the suspensions waiting for these instantiations or update.

• In the rule $\text{UnifT}$, $\text{inst}$ is the set of variables that are bound with nonvariables value by the unification algorithm. If a variable $u(i)$ is bound with another variable (free) $u(j)$, then there is no need to wake up the suspensions waiting for $u(i)$ (they are added in the queue of suspensions waiting for $u(j)$). Thus the set $L$ can be empty if $\sigma' = \sigma$ or if all variables bound in that step of unification are bound to free variables.

• Note that we remain free to choose the order of evaluation of binary constructs as for $\text{EA}$ (We give in Fig. 3 the rules for pairs, rules for unification and application are similar.).

• Moreover, the order of evaluation of terms bound in $I'$ is also free (see rule $\text{Aw}$).

\[
\begin{align*}
\text{strict-Pair1F} & \quad t \rightarrow \text{failwith}(s) \\
& \quad (t, t') \rightarrow \text{failwith}(s) \\
\text{strict-Pair2F} & \quad t' \rightarrow \text{failwith}(s) \\
& \quad (t, t') \rightarrow \text{failwith}(s) \\
\text{strict-Pair1} & \quad t \rightarrow t_1 \\
& \quad (t, t') \rightarrow (t_1, t') \\
\text{strict-Pair2} & \quad t' \rightarrow t'_1 \\
& \quad (t, t') \rightarrow (t, t'_1) \\
\text{strict-Env} & \quad x.(x, t) :: e \rightarrow t \\
\text{strict-Env0} & \quad x.(y, t) :: e \rightarrow x.e \\
\text{strict-Const} & \quad c.e \rightarrow c \\
\text{strict-AEnv} & \quad (t \; t').e \rightarrow (t.e \; t'.e) \\
\text{strict-PEnv} & \quad (t, t').e \rightarrow (t.e, t'.e) \\
& \quad t \text{ is in } \rightarrow \text{ normal form} \\
\text{strict-}\beta & \quad f = (\text{fun } p_1 \rightarrow a_1 \mid \cdots \mid p_n \rightarrow a_n).e, \\
& \quad \Phi s(1, pl, t) = i, e_i \\
& \quad f \ t \rightarrow a, e, @e \\
& \quad t \text{ is in } \rightarrow \text{ normal form. } \Phi s(1, pl, t) = \text{fail} \\
\text{strict-Fail} & \quad f = (\text{fun } p_1 \rightarrow a_1 \mid \cdots \mid p_n \rightarrow a_n).e, \\
& \quad ft \rightarrow \text{failwith(Pattern)}
\end{align*}
\]

Fig. 6. $\text{SAE}$. 
B.1. The calculus SŁAE

We give in Fig. 6 the rules for the sub calculus defined on the subset of the functional terms. We do not have to take care of all the stuff needed for logical variables and suspensions. Thus, this calculus is rather simple. As above, we give only the rules for pairs, the rules for the evaluation of the members of an application are similar.

Appendix C: demonstration of Theorem 3.6

Let us give preliminary results.

Lemma C.1. If \( \langle t, \sigma, \alpha, \Gamma \rangle \to \langle t', \sigma', \alpha', \Gamma' \rangle \) by application of a suspension rule then \( \langle t, \sigma, \alpha, \Gamma \rangle \equiv \langle t', \sigma', \alpha', \Gamma' \rangle \).

Proposition C.2. If \( \langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle \to \langle t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 \rangle \) by application of a rule distinct of a suspension rule, and if \( \langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle \equiv \langle t_2, \sigma_2, \alpha_2, \Gamma_2 \rangle \) then we have \( \langle t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 \rangle \) such that \( \langle t_2, \sigma_2, \alpha_2, \Gamma_2 \rangle \to \langle t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 \rangle \) and \( \langle t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 \rangle \equiv \langle t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 \rangle \).

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

Let \( \langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle \) be reduced by \( \beta \) applied on a subterm of \( t_1 \). Let note that subterm \( (\text{fun } p_1 \to a_1 \mid \cdots \mid p_n \to a_n).\text{ev} \). By the hypothesis of \( \equiv \) we have \( (\sigma_2 \circ \alpha_2 \circ \Gamma_2)^*=\langle t_2, u(j) \rangle \) and \( \langle t_2, u(j) \rangle \not\in \text{dom}(\sigma_2) \). The \( \equiv \) hypothesis ensures that \( u(j) \not\in \text{dom}(\alpha_2) \) as in that case the application would be suspended when the rule \( \beta \) applies on \( t_1 \). Thus we have:

\[
\sigma^*_2(u(j)) = (\text{fun } p_1 \to a_1 \mid \cdots \mid p_n \to a_n).\text{ev}
\]

The \( \equiv \) hypothesis ensures that the same pattern matches in both reduction and then application of \( \text{Aw} \) with the rule \( \beta \) on that term clearly leads to an equivalent four_uple.

We have now the result of strong confluence of \( \to \) up to \( \equiv \).

Theorem C.3. For all \( \langle t, \sigma, \alpha, \Gamma \rangle \) such that

\[
\langle t, \sigma, \alpha, \Gamma \rangle \to \langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle
\]

\[
\langle t, \sigma, \alpha, \Gamma \rangle \to \langle t_2, \sigma_2, \alpha_2, \Gamma_2 \rangle
\]

there exists \( \langle t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 \rangle \) and \( \langle t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 \rangle \) such that

\[
\langle t_1, \sigma_1, \alpha_1, \Gamma_1 \rangle \stackrel{0.1}{\to} \langle t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 \rangle,
\]

\[
\langle t_2, \sigma_2, \alpha_2, \Gamma_2 \rangle \stackrel{0.1}{\to} \langle t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 \rangle,
\]

\( \langle t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 \rangle \equiv \langle t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 \rangle \).
**Proof.** It is illustrated in Fig. 7. The case where no suspension rule is used is resolved by Proposition 1. The cases where at least one reduction use a suspension rule are: if both \( r_1 \) and \( r_2 \) use suspension rules, then Lemma C.1 is enough to conclude. If one \( r_i \) use a suspension rule, then we conclude with Proposition C.2 and Lemma C1. □

**Proof of Theorem 3.6.** We show that the diagram of Fig. 8 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. For example,

\[
\langle \text{fun} (x, y) \to \text{unif} \ x \ \text{True}, \langle \rangle \rangle (u(1), \ \text{unif}(1) \ \text{False}, \emptyset, \emptyset, \emptyset)
\]

has two normal forms:

\[
\langle \text{void}, \{u(1), \ \text{True}\}, \emptyset, \emptyset \rangle
\]

and \( \langle \text{failwith} \ (\text{Unif}), \{u(1), \text{False}\}, \emptyset, \emptyset \rangle \).

**Acknowledgments**

We would like to thank all members of LIENS-INRIA Formel project for helpful discussions, in particular Therese Hardin for her accurate suggestions to improve our formalism and demonstration, and to referees of this paper for invaluable suggestions.
References