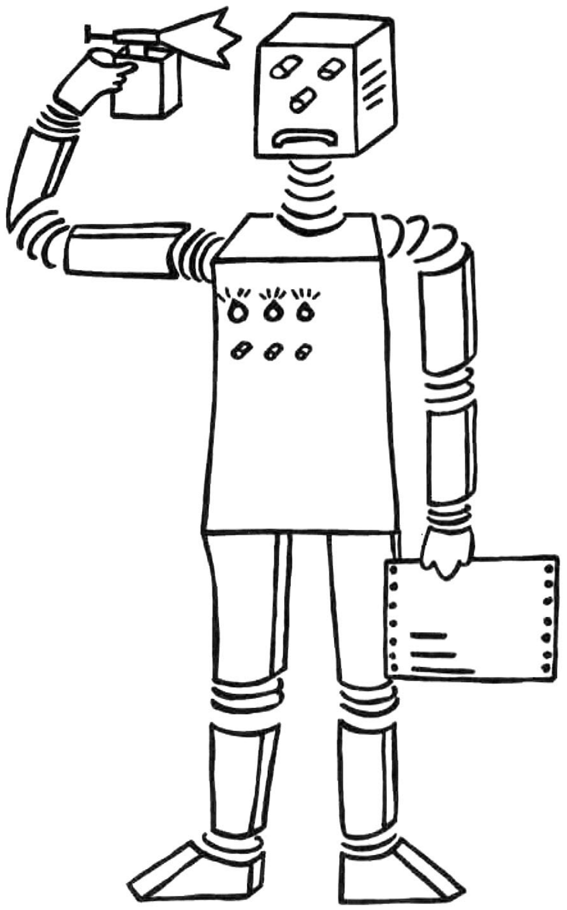


SASLOG: Lazy Evaluation Meets Backtracking

Knut Hinkelmann, Klaus Noekel,
Robert Rehbold

SEKI Report SR-88-01

SASLOG: Lazy Evaluation Meets Backtracking

Knut Hinkelmann, Klaus Nökel and Robert Rehbold

Abstract

We describe a combined functional / logic programming language SASLOG which contains Turner's SASL, a fully lazy, higher-order functional language, and pure Prolog as subsets.
Our integration is symmetric, i.e. functional terms can appear in the logic part of the program and v.v. Exploiting the natural correspondence between backtracking and lazy streams yields an elegant solution to the problem of transferring alternative variable bindings to the calling functional part of the program.
We replace the rewriting approach to function evaluation by combinator graph reduction, thereby regaining computational efficiency and the structure sharing properties. Our solution is equally well suited to a fixed combinator set and to a super combinator implementation. In the paper we use Turner's fixed combinator set.

Keywords:

functional programming, logic programming, lazy evaluation, combinators, graph reduction, streams, backtracking, set abstraction, semantic unification.

1. Introduction

Declarative programming languages have been discussed now for several years. Declarative languages describe certain situations and facts without giving explicit instructions for computation. They are developed based on mathematical models independent of the requirements of concrete computing machines. Their mathematical well-foundedness results in certain benefits, e.g. greater expressive power, ease of parallel evaluation and the possibilitiy of formal manipulations thus supporting the verification of programs.
Two kinds of programming styles belong to this group: functional programming and relational (logic) programming. The advantages of functional programming can be found in e.g. [Darlington, Henderson, Turner82]. Several authors ([Turner82], [Hughes84]) emphasize that lazy evaluation and higher-order functions are two of the most important features of purely functional languages, enabling the programmer to easily modularize his/her programs. Lazy evaluation requires normal-order reduction, which can be implemented quite efficiently using combinator graph reduction (as shown in [Turner79]). Logic programming, based on first-order Predicate Calculus, is the other major declarative approach. Its most prominent exponent is Prolog ([Clocksin, Mellish84]).
Both the functional and the logic programming style have their advantages. In order to be able to decide locally which part of a problem to represent using which style, many integrations of both paradigms have been published. Distinctions between these approaches can be made by the intensity of the integration, the functional and logic language used or the implementation technique used.
The rest of the paper is organized as follows: following a sketch of the new constructs we define an operational semantics for SASLOG. In paragraph 6 we address the central implementation issues that arise out of the need to interface logic variables and (inherently variable-free) combinator expressions. We conclude with a brief statement about the current status and the future goals of the project.

2. What's New?

We present a fully symmetric integration of a pure functional higher-order language featuring lazy evaluation (SASL) implemented via combinator graph reduction and a classical interpreter for a logic language (pure Prolog). Lazy lists in the functional part and backtracking in the logic part are interlinked and yield a natural interface between the two programming styles. The primary concern of the paper is to propose a solution to the difficult problem of reconciling combinator graph reduction and logic variables that avoids the FUNARG problem.

3. Related Works

We decided to integrate two (reasonably efficiently implemented) existing languages: SASL (St. Andrews Static Language, [Turner83]), since it features lazy evaluation and higher-order functions, and pure Prolog.
Unlike many known integrations we really interlink the two languages in both directions, i.e. Prolog goals can be proven from SASL as well as functions can be called from Prolog. Influenced by Wadler ([Wadler85]) and Narain ([Narain86]) our integration is based on the kinship between lazy evaluation and depth-first search with backtracking. This yields an elegant solution of the opposition between determinism in functional languages, where every expression has a definite value, and nondeterminism in logic languages, where one goal can have multiple solutions.
Prolog goals in functional expressions are represented via set abstraction (or better list abstraction) where the elements of such a potentially inifinite stream are the alternative solutions of the goal (cf. [Darlington, Field, Pull86] or SUPERLOGLISP[Robinson83]). Because SASL is lazy, each solution is computed only when it is needed.
Integrating a functional language in a logic language can be done at two distinct levels. At the predicate level the integration essentially consists in adding a new built-in predicate `eq(X,Y)`, which computes the functional expression `Y` and unifies the result with the term `X` (cf. `is` in Prolog). Systems like LISPLOG ([Boley 86]) or HORNE ([Frisch, Allen, Giuliano83]) use predicate-level integration. Integration at term level, as realized in SASLOG, consists in using functional expressions as terms in predicates (cf. FUNLOG [Subrahmanyam, You86]). Term-level integration requires an extension to the unification algorithm of the logic programming language to take into account the semantics of the function symbols. See [Dincbas, vanHentenryck87] for a discussion of different extended unification algorithms.

Several systems have been built that make available a functional and a logic language in a single environment thereby implementing the logic language (mostly Prolog) in the functional language (mostly LISP) with defined interfaces to evaluate functional expressions in the logic part. Examples are LISPLOG ([Boley 86]), LOGLISP ([Robinson, Sibert 82a,b]), HORNE ([Frisch, Allen, Giuliano83]) and LM-Prolog ([Kahn, Carlsson83]).

Instead of integrating two existing languages many attempts were made to invent a totally new language or to augment an existing language with new features to achieve functional and logic programming.
As can be found in [Reddy86] there are several ways to capture the additional expressive power of logic programming within the framework of functional languages. The first way is to execute nonground expressions by narrowing instead of reduction (as in FRESH[Smolka86]). Using set expressions with free logic variables allows the importation of fresh variables in the output of expressions. SUPERLOGLISP[Robinson83] and the language in [Darlington, Field, Pull86]

augment a functional language by set abstraction and unification to achieve relational programming. The translation of functional programs to logic programs with resolution used as their operational semantics results in the usage of functions as syntactic sugar (e.g. LEAF[Barbuti, Bellia, Levi86]).

The introduction of functional notation into relational languages is achieved by the extension of Prolog with the equality relation ([Kornfeld83]). EQLOG ([Goguen, Meseguer86]) combines horn-clause logic with confluent and terminating equational theories.

For a more detailled discussion of different methods to integrate functional and logic programming paradigms see [Bellia, Levi86].

4. A Review of SASLOG

The SASL part is based on [Turner83]. For convenience in linking SASL to Prolog we added the object type ***constant*** to SASL. Constants are identifiers marked by "`'`" (like the abbrevation for QUOTE in Lisp); they are atomic and do not correspond to strings.

Examples:

constants: `'john` `'mary` `'sam`

```
def lover 'mary = 'john
    lover x     = 'sam
```

The link from SASL to Prolog is possible through two constructs: `prove`- and ZF-expressions.

1. Instead of any boolean expression in a SASL term there can be an expression

   `prove(`*`prolog-goal`*`)`

   where *`prolog-goal`* can contain SASL variables and expressions. If the value of the `prove`-expression is needed (remember: SASL evaluation is lazy!) the Prolog interpreter is called with the given *`prolog-goal`*. If the goal can be proven, the expression yields `TRUE`; if the Prolog interpreter fails the result is `FALSE`.
   Regard the following SASL function which tests whether we know the mother of a given person (returning `'ok`) or not (`'unknown`):

   ```
   def test x = prove(mother(_M,x)) -> 'ok; 'unknown
   ```

   Note that the Prolog goal contains (local) Prolog variables (`_M`) and parameters from the SASL function (`x`).

2. The more interesting link to Prolog (of which 1. is just a syntactically sugared special case) is through an extension of the ZF-expression (named after the underlying Zermelo-Fraenkel set

abstraction). The syntax of a SASLOG ZF-expression is as follows:

$[E; Q_1; \ldots; Q_n]$

where the result term E is an expression and the qualifiers Q_k take the form

V_k <- E_k	("normal" generator) or
$[V_{k1}, \ldots, V_{km}]$ <- *prolog-goal*	(Prolog generator) or
E_k , E_k a boolean-valued expression	(filter)

The meaning of this ZF-expression is very much the same as the one of $\{E \mid Q_1;\ldots;Q_n\}$ in mathematical notation (reading "<-" for "$\in$"), except that the ZF-expression denotes a list instead of a real set (i.e. doubles can occur and the order of members is significant).
While V_k <- E_k binds V_k successively to the members of the list produced by E_k, $[V_{k1}, \ldots, V_{km}]$ <- *prolog-goal* binds the V_{ki} simultaneously to the values these Prolog variables take in the Prolog proof of *prolog-goal*. If the next set of values is needed, backtracking on *prolog-goal* is started.
Consider the following example, where the function f yields the list of all grandchilds of a given list of persons. Supposing the Prolog database contains several facts of the form parent('john,'sue) we can define f as follows:

```
def f L = [gc; old <- L; [gc] <- parent(old,_X), parent(_X,_gc)]
```

If we want to filter out only those grandchilds whose parent belongs to a certain set of people we could change the definition to

```
def g L = [gc; old <- L;
               [gc,X] <- parent(old,_X), parent(_X,_gc);
               member ['sue,'joe,'john,'mary] X ]
```

(* Note that in SASL the order of the arguments to member is changed to make currying easier. *)
While X and gc are logic variables inside the goal, they become SASL parameters outside the generator.

Let us take a closer look at this example showing the combined computation with lazy evaluation and backtracking. Given the following database definitions:

```
parent('john,'sue).
parent('john,'sam).
parent('sam,'mary).
parent('joe,'linus).
parent('sue,'charly).
parent('mary,'lucy).
parent('jeff,'joe).
```

and the definition of `g` as above, then the expression E = `g ['john,'jeff]` is evaluated as follows:

a. The first element of `['john,'jeff]` (= `'john`) is assigned to `old`.
b. The Prolog interpreter is started to prove `parent('john,_X), parent(_X,_gc).`
c. The interpreter returns with `X` bound to `'sue` and `gc` bound to `'charly`.
d. `member ['sue,'joe,'john,'mary] 'sue` is evaluated to `TRUE`.
e. Since no more qualifiers exist this is a acceptable solution and the value bound to `gc` (i.e. '`charlie`) is delivered as the first element of the global expression E.
f. If the next member of E is needed, the next element from the last satisfied generator must be generated; since in this case this is the Prolog interpreter, backtracking is started.
g. Prolog finds another solution binding `X` to `'sam` and `gc` to `'mary`.
h. `member ['sue,'joe,'john,'mary] 'sam` is `FALSE` so another backtracking is started.
i. Since there are no more solutions for `parent('john,_X), parent(_X,_gc)`, the Prolog interpreter fails, thus the next element from the generator of `old` must be examined (that is `'jeff`).
j. A "new" Prolog interpreter is started to prove
   `parent('jeff,_X), parent(_X,_gc).`
k. It finds a solution (`X = 'joe, gc = 'linus`) which satisfies the "`member`"-filter so that `'linus` is the next member of E.
l. No more solutions for `'jeff` can be found so the next element of `L` has to be used. Since there is no such element, the result list is terminated.

Thus: `g ['john,'jeff] = ['charly, 'linus].`

Calling SASL from Prolog is a little easier: any term in the goals on the right hand side of a Prolog clause may be an arbitrary SASL expression. These SASL terms may contain Prolog variables, which must be bound to a value when being evaluated (we do not perform residuation). Naturally SASL terms in Prolog goals can contain Prolog goals themselves (via `prove`- or ZF-expressions) etc. and vice versa. Note that there is no need for an operator like "is", since "==" (unification) serves the purpose equally well.
Suppose we want a predicate `P(X,Y)` which is true, if a person `X` has grandchilds who are older than 20 (assuming the presence of a function `age`) and whose parent is either Sue, John, Joe or Mary. It should allow to be called with any combination of bound/unbound variables. Of course we want to use our previously defined function `g`.

```
P(_X,_Y) :- person(_X),
            _Y == g[_X],
            some (< 20) (map age _Y) == TRUE.
```

```
def some f []    = FALSE
    some f [a|x] = (f a) or (some f x)
def map  f []    = []
    map  f [a|x] = [f a | map f x]
```

Note that in a call of the Prolog predicate `P` the SASL function `g` is called which itself calls Prolog again.

Looking at the usage of the well-known predicate `member` shows the interaction between lazy evaluation and backtracking in the Prolog part of SASLOG:

```
member(_e,[_e|_?]).
member(_e,[_?|_l]) :- member(_e,_l).
```

The goal

```
?-member(_x,(from 1 where from n = [n|(from (n+1))])), p(_x).
```

will succeed if there is a natural number `_x` such that `p(_x)` is true. The expression `(from 1 where from n = [n|(from (n+1))])` denotes the infinite list of natural numbers. Due to the lazy evaluation paradigm the list is evaluated only far enough so that `_x` can be unified with an element in the list. If the subsequent goal `p(_x)` fails with this value, backtracking causes the further computation until `_x` can be unified with another value.

5. The Operational Semantics of SASLOG

One reason to integrate two existing languages instead of creating a completely new one is to give the programmer a familiar basis to work on so that programs written in either of the two languages can still be used. This puts fairly strong restrictions on the semantics of the combined language: the separate semantics have to be retained as special cases and the new elements concerning the link between the languages must be injected into their union as unobtrusively as possible. In this paragraph we will construct an operational semantics for SASLOG building on the operational semantics for SASL (i.e. the reduction rules for the combinators together with normal-order reduction) and Prolog, respectively.

Before going into detail we give an informal overview of the semantics we have in mind for the new constructs:

(i) A ZF-generator of the form `list-of-vars <- prolog-goal` is taken to generate the stream of all success bindings of `prolog-goal` projected onto the `list-of-vars`. In order to make the stream truly lazy we will introduce a special Prolog combinator that takes a Prolog continuation and reduces to a list of values and a new continuation. Logic variables which on return to SASL are still uninstantiated are mapped to $\perp$, the SASL value for "undefined".

(ii) The basic unification algorithm is replaced by semantic unification. Two SASL terms which contain functions other than constructors unify iff they can be reduced to equal ground terms where ground terms are considered equal iff the SASL function `eq` yields `TRUE` for them. Prior to reduction all logic variables in the SASL terms are replaced by their current bindings. No residuation takes place, yet an uninstantiated variable does not automatically mean that unification fails. As in (i) we substitute $\perp$ for these variables; so if some of the functions in the term are non-strict, normal-order reduction may nevertheless produce a non-$\perp$ result (e.g. when the unbound variable occurs in the non-selected arm of a conditional).

(iii) To cope with logic variables in SASL terms we need the notion of reduction w.r.t. a binding environment which is in some way alien to the basic idea of combinator graph reduction. Since at any moment during the execution of a SASLOG program an arbitrary number of alternating Prolog incarnations and SASL reductions may be pending (each Prolog incarnation with its own current binding environment) we have to be careful in defining which term is to be reduced in which environment in order to evade both the upward and downward FUNARG problems. The convention regarding $\perp$ outlined in (i) and (ii) ensures that no uninstantiated Prolog variable can appear in a reduction taking place outside of the scope of the Prolog incarnation to which it belongs. We shall see in the following paragraph how the formal solution below can be implemented with only a slight loss in laziness.

The semantic unification algorithm

Although our algorithm bears some resemblance to the one given in [Subrahmanyam, You 86] there is an important difference. Embedded SASL terms are always reduced to their head-normal form by the combinator reduction machine rather than step by step under the control of the unification algorithm. This means that terms containing uninstantiated variables are not treated as irreducible but as containing $\perp$ subterms. As a direct consequence in SASLOG only one notion of equality is left: that of the SASL function `eq`. Unification fails in exactly those cases where `eq` is `FALSE` for the head-normal forms of the terms and is aborted when `eq` yields $\perp$.

One might note that by unifying terms containing non-constructor functions only if they are equal w.r.t. equality in SASL we avoid importing higher-order predicate logic into SASLOG. Of course, a logic variable may be bound to a SASL function which in turn is defined as the characteristic function of a predicate, as in

```
all(_F, []).
all(_F, [_A|_X]) :- F(_A) == TRUE, all(_F,_X).
human('socrates).
human('adam).
human('eve).
def  s-human x = prove(human(x))

?- all(s-human, ['adam,'eve]).
```

However, by definition the SASL-function eq is undefined for any two functional values so that they cannot be unified. This limits the use of functional values to exactly those cases covered by the 'call' meta-predicate in conventional Prolog implementations.

We can now specify our unification algorithm which is the only departure from standard Prolog semantics. It differs from syntactic unification in the last four cases where two SASL terms are to be unified.

```
UNIFY $t_1$ $t_2$ $\sigma$:

IF $\sigma$ = 'FAIL' THEN RETURN 'FAIL';
FOR i=1,2 DO $t_i$ := ULTIMATE-ASSOC($t_i$, $\sigma$);
IF $t_1$ is a variable THEN RETURN $\sigma \cup \{(t_1/t_2)\}$;
IF $t_2$ is a variable THEN RETURN $\sigma \cup \{(t_2/t_1)\}$;
IF $t_1$ and $t_2$ are in head-normal form
   THEN IF $t_i = g(s_{i1},\ldots,s_{ik})$, g constructor
        THEN RETURN UNIFY($s_{1k}$,$s_{2k}$,UNIFY(...UNIFY($s_{11}$,$s_{21}$,$\sigma$)))
        ELSE RETURN FAIL;
IF $t_1$ is in head-normal form and $t_2$ is not
   THEN RETURN UNIFY($t_1$,SASL-REDUCE(ULTIMATE-INST($t_2$,$\sigma'$)),$\sigma$);
IF $t_2$ is in head-normal form and $t_1$ is not
   THEN RETURN UNIFY(SASL-REDUCE(ULTIMATE-INST($t_1$,$\sigma'$)),$t_2$,$\sigma$);
FOR i=1,2 DO $t_i$ := ULTIMATE-INST($t_i$, $\sigma'$);
IF SASL-REDUCE(eq $t_1$ $t_2$) = TRUE THEN RETURN $\sigma$;
IF SASL-REDUCE(eq $t_1$ $t_2$) = FALSE THEN RETURN FAIL;
ABORT WITH ERROR
```

where for a variable x

$$\sigma'(x) := \begin{cases} \sigma(x), & \text{if } x \in Dom(\sigma) \\ \perp, & \text{else} \end{cases}$$

ULTIMATE-ASSOC dereferences a variable until a non-variable is encountered whereas ULTIMATE-INST applies ULTIMATE-ASSOC recursively to all variables in a term.

But notice, that this unification algorithm is incomplete. Thus the unification of the terms append [1|_x] [3,4] and append [1,2,3] [4] with function definition

```
def append []     l  = l
    append [a|l1] l2 = [a|(append l1 l2)]
```

does not succeed, if _x is a free variable. The function append is strict in its first argument and thus append [1|_x] [3,4] , with _x instantiated to ⊥, will be reduced to ⊥ and unification aborts with error.

One should note that although the unification algorithm above is described in the form of a procedure its sole purpose is the specification of the semantics. Some of the operations would be extremely costly if implemented in a straightforward manner. In particular we have avoided the issue of logic variables in SASL terms by grounding them prior to reduction, as in

SASL-REDUCE(ULTIMATE-INST(t_2,σ'))

where t_2 may be any combinator graph. Obviously the application of a substitution to a large and probably cyclic structure would be quite expensive. In the paragraph on implementation issues we will address this and other problems.

The augmented reduction semantics for SASL

As we have seen we need not worry about the treatment of logic variables in the reduction process (at least as long as we are not interested in the implementation). All we have to specify is how the augmented ZF-expressions and prove-expressions are translated into combinator graphs and how the newly introduced combinators are to be reduced. Atoms can be treated as distinct constants requiring only a small adjustment in the reduction rule for eq. The rest of the combinators and their reduction rules remain completely unchanged.

First, we observe that a prove-expression is in fact a special case of a ZF-expression since every expression of the form

prove *prolog-goal*

can be replaced by the equivalent

[TRUE ; [] <- *prolog-goal*] <> [].

Likewise ZF-expressions can be eliminated by a purely syntactical program transformation according to the following scheme:

A ZF-expression Z generally takes the form

    [E;Q1;...;Qn]

where the result term `E` is an expression and the qualifiers Q_k take the form

V_k <- E_k	("normal" generator) or
[V_{k1},...,V_{km}] <- *prolog-goal*	(Prolog generator) or
E_k, E_k a boolean-valued expression	(filter)

Assume that V_1,..., V_r are the generator variables of Z. We let T(Z) denote [V_r,..., V_1]. Then

map f [T(Z); Q_1;...;Q_n] where f T(Z) = E

is equivalent to the original Z.

A ZF-expression Z is said to be in normal form iff its result term is T(Z). Let NZ_n be a ZF-expression in normal form with n qualifiers. We inductively define norm(NZ_n), an equivalent ZF-free SASL expression as follows:

case n = 0: norm(NZ_0) := [[]].
case n > 0, Q_n = E_n (filter):
  norm(NZ_n) := filter f norm(NZ_{n-1}) where f T(NZ_{n-1}) = E_n.
case n > 0, Q_n = V_n <- E_n (normal generator):
  norm(NZ_n) := cp f norm(NZ_{n-1}) where f T(NZ_{n-1}) = E_n.
case n > 0, Q_n = [V_{n1},...,V_{nm}] <- *prolog-goal*:
  norm(NZ_n) := cpp f norm(NZ_{n-1})
    where f T(NZ_{n-1}) = goal *prolog-goal* [V_{n1},...,V_{nm}] .

The function `filter` is a predefined SASL function whereas `cp`, `cpp` (named for their superficial similarity to cartesian products) and the auxiliary functions `join` and `joinp` are new combinators with the reduction rules:

```
cp f []         = []
cp f [a|x]      = append (join (f a) a) (cp f x)
join []    e    = []
join [a|x] e    = [[a|e] | (join x e)]
cpp f []        = []
cpp f [a|x]     = append (joinp (f a) a) (cpp f x)
joinp []    e   = []
joinp [a|x] e   = [(append a e) | (joinp x e)]
```

The real interfacing between SASL and the Prolog interpreter is hidden in the reduction rules of the combinators `goal` and `next`:

```
goal prolog-goal resultlist = [], if prolog-goal is not provable
goal prolog-goal resultlist =
[ULTIMATE-INST (resultlist, first successful variable binding for prolog-goal)
    | (next prolog-continuation resultlist)], otherwise
next prolog-continuation resultlist =
        [], if there is no alternative proof of prolog-goal
next prolog-continuation resultlist =
[ULTIMATE-INST (resultlist, next successful variable binding for prolog-goal)
    | (next prolog-continuation' resultlist)], otherwise
```

In both cases `ULTIMATE-INST` replaces unbound variables by $\perp$, i.e. σ' instead of σ is used.

In addition to the operational semantics given above a denotational one would be desirable for a complete understanding. We feel, however, that resolving the clash between higher-order SASL functions and first-order Prolog requires substantial further work in this direction.

6. Implementation Aspects

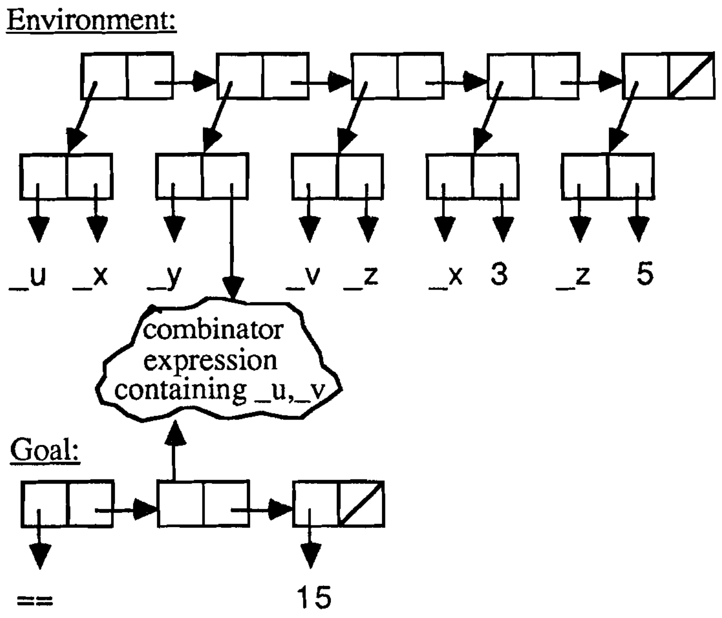
SASL expressions containing logic variables

One problem is caused by the destructive graph reduction technique. Being quite efficient in pure SASL applications it cannot be used for SASL expressions containing logic variables. The reason for this is that logic variables may change their value due to backtracking. Thus, a destructive first reduction of a SASL expression would prevent it from being evaluated a second time with new bindings for the logic variables it contains.

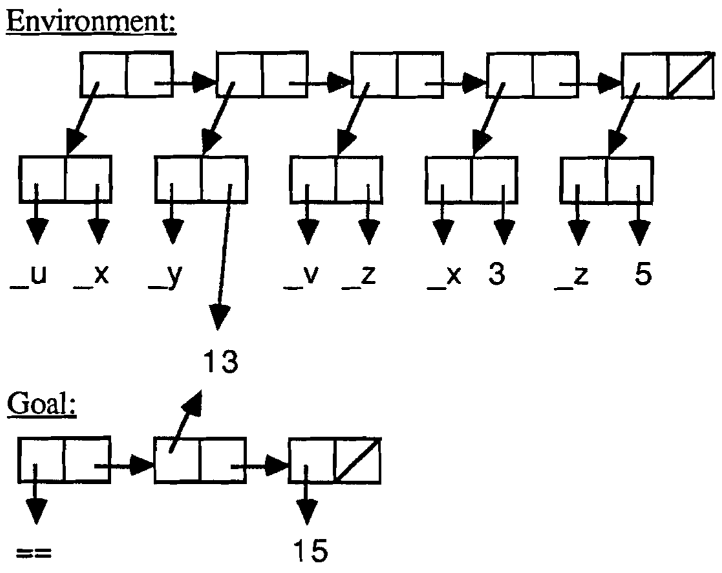
Example:

```
database:  p(_x,_y,_z) :- r(_x,_z), _y == 15.
           r(3,5).
           r(4,6).
goal:      p(_u,_u+(_v+5),_v).
```

Unifying the goal with the conclusion we get the substitution
σ = { `(_u/_x)`, `(_y/_u+_v+5)`, `(_v/_z)` }. Now we prove `r(_x,_z)` obtaining
σ ∪ { `(_x,3)`, `(_z,5)` }. To prove `_y == 15` requires the reduction of `_u+(_v+5)`. In a simplified form the internal representation before this reduction looks like:



A straightforward (and wrong!) reduction will simply destructively change the combinator expression. _u is physically replaced by its value `3`, _v by its value `5`; then the expression is reduced to `13` and we get the following situation:



Because unification fails, backtracking is required. But by now the value of _y is a constant which is obviously false.

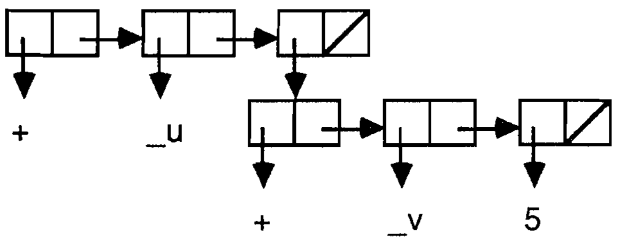
Abstracting logic variables

A naive solution would be to copy every expression before reduction, probably combined with instantiation. This would be correct but since the combinator expression may be a cyclic graph copying is an expensive (i.e. time and space consuming) operation.

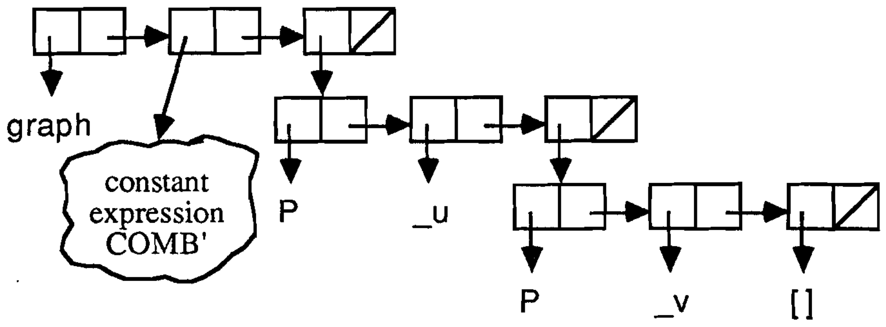
To solve this problem without having to copy at run-time we need to invest some effort at compile-time: First, we scan a clause for its global logic variables (i.e. those logic variables that occur outside of any embedded `prove-` or zf-term). Then we can translate the SASL expressions into a special form: After collecting the global logic variables occurring in an expression E into a list $(v_1 \ldots v_n)$ we translate E into a combinator expression COMB. If n=0, the translation is finished. Otherwise, we abstract the variables $v_1, \ldots, v_n$ from COMB, obtaining the variable-free combinator expression COMB', and finally replace E by the form `(graph COMB'` $[v_1, \ldots, v_n]$`)`. When reducing this form, we first instantiate the variables and replace the pointer to `(graph COMB'` $[v_1, \ldots, v_n]$`)` by one to the form (COMB' $v_1' \ldots v_n'$), where the v_i' are the ultimately instantiated values of the variables. If the environment contains no final (i.e. ground) binding for a v_i at that time, v_i is bound to $\perp$. This technique prevents us from copying of and instantiating through arbitrarily complex combinator graphs.

Example (continued):

The goal `p(_u,_u+(_v+5),_v)` is a literal with only one SASL-expression `_u+_v+5`. The internal representation of `_u+(_v+5)` is:

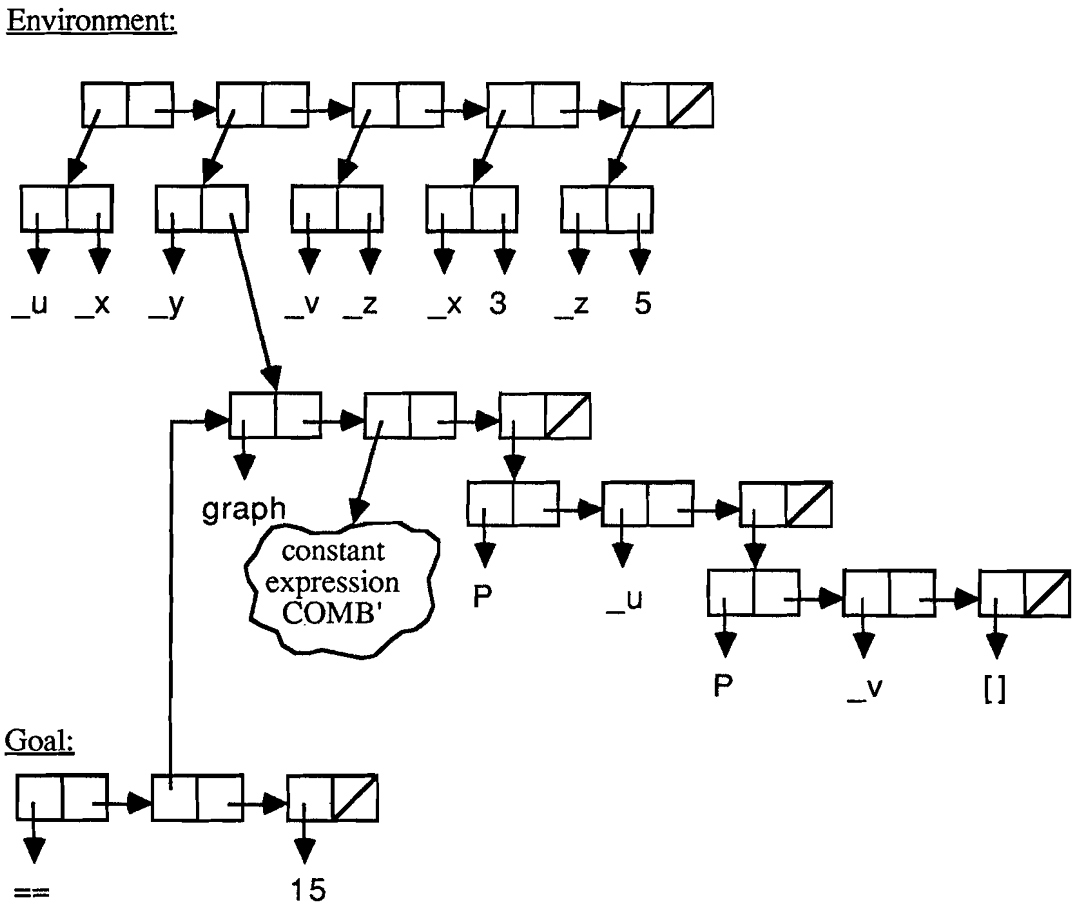


`COMB' = [_u][_v](_u+(_v+5)) = (C (B' +) (C + 5))` (where [x]E denotes x abstracted from E) so we get

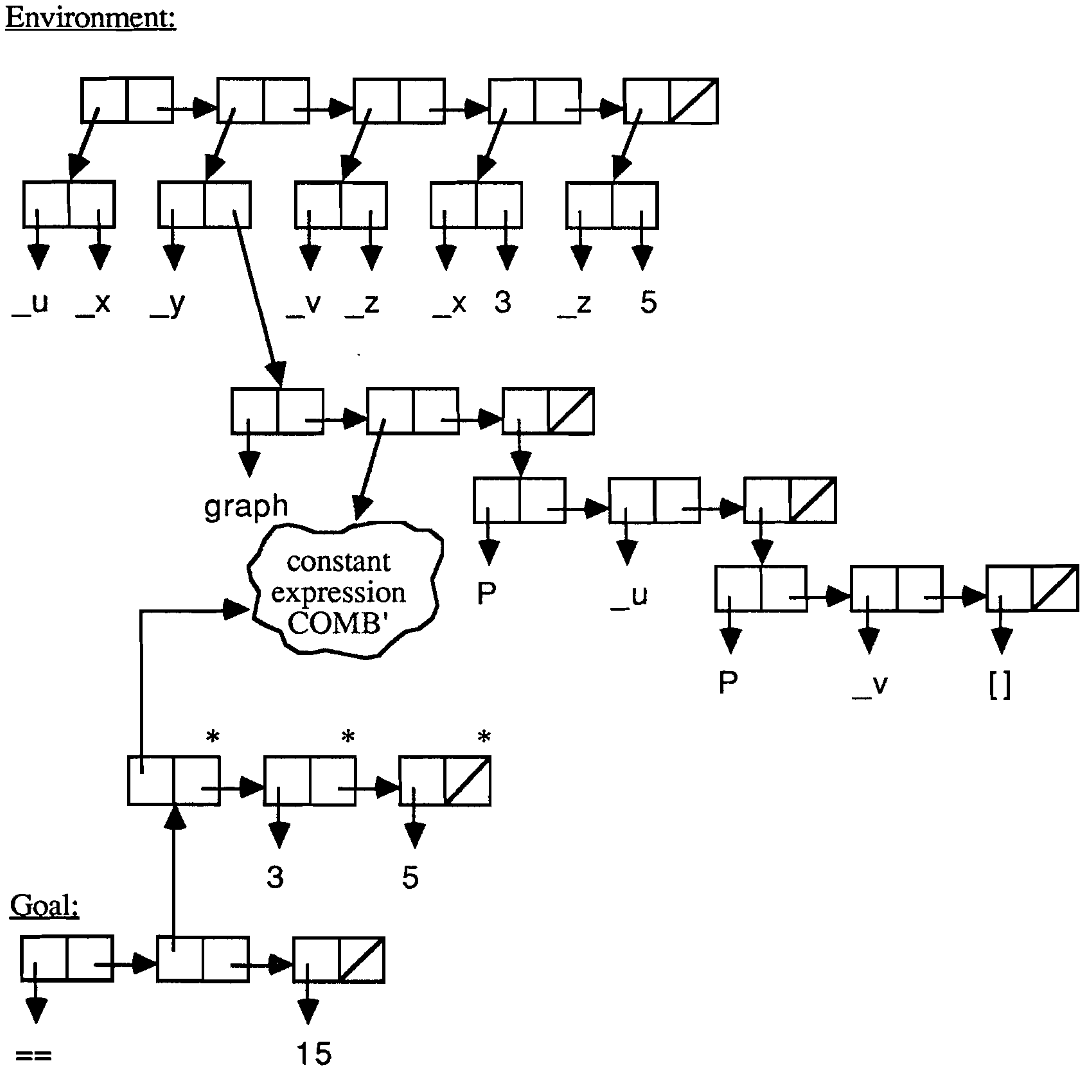


after translation. The combinator `P` is the internal list constructor.

When we reach the goal `_y == 15`, the environment and the goal have the following form:



The instantiation process results in the following situation:



The cloud representing the constant part of the expression `_u+(_v+5)` needs not to be copied, while the nodes signed with "*" are new nodes. During the reduction process only constant subterms of `COMB'` can be changed preserving the functionality of the combinator expression. Now we see that the instantiation pattern of `_y` will still be correct if backtracking causes alteration of the bindings of `_x` and `_z`.

Returning variable bindings to SASL

While this procedure solves the problem of calling SASL from Prolog we still have to answer the question of how variable bindings found by the Prolog interpreter should be returned to the calling SASL program. The problem here is exemplified by the following situation:

ZF-expression: `[X;[X] <- foo(_X,_Y)]`

variable binding found by the Prolog interpreter:
`{(_Z/3),(_X/[_U,_V]),(_U/_Z),(_V/(graph (+ 1) [_U]))}`

If we ultimately instantiate `_X` we get `_X = [3,((+ 1) 3)]`.

There are two important decisions to make at this point. First, we have to decide <u>when</u> to instantiate a variable.The option that fits best with the lazy evaluation strategy would be to postpone the instantiation of logic variables until they are actually needed during subsequent reduction steps. While this solution has the advantage of being conceptually pleasing its consequences have nevertheless lead us to reject it. Postponing the instantiation presupposes (among other things) that the entire binding environment is kept along with the variable which would be extremely space-consuming if e.g. several successive solutions were to be gathered in a list. We therefore perform the `ULTIMATE-INST` in the reduction rules of `goal` and `next` immediately when the final variable binding is handed back by the Prolog interpreter. In doing so we run the risk of instantiating variables which will not be referred to subsequently; however, apart from a few special cases (such as counting the number of solutions without actually inspecting them) we do not expect this situation to occur very frequently in reality.

The second major decision concerns the way that the method of `ULTIMATE-INST`antiating affects structure sharing. Imagine that in the example above `_Z`'s value were a large list structure instead of `3` and consider the following intermediate step during the ultimate instantiation of `_X`:

```
_X = [_Z, (graph ... [_Z])]
```

If the two occurrences of `_Z` were instantiated separately one would not only duplicate the computational effort but (even worse) `_Z` would be replaced by two copies of the list structure that would not be shared in memory. The structure sharing properties of combinator graph reduction which ordinarily guarantee that no expression has to be reduced more than once would be completely lost. The solution that we offer makes use of a memoized version of `ULTIMATE-INST` that records the ultimate instantiation of a variable in a working area when it is computed for the first time and looks it up when the same variable is again encountered later. We keep the working area until all the variables in the result list have been processed thereby achieving structure sharing <u>and</u> ultimate instantiation of all variables in only one sweep through the binding environment. A more detailed discussion of the enhanced `ULTIMATE-INST` is given in [Hinkelmann88].

7. Status of the Implementation and Future Work

The SASLOG interpreter has been implemented in Common Lisp and currently runs on a Symbolics Lisp machine. It contains as essential parts the former LISPLOG interpreter described in [Boley85] and the SASL interpreter written by two of the authors [Nökel,Rehbold86].
Our next step will be to implement a polymorphic type concept like the one in Miranda [Turner85] that allows typing of both functional and logic expressions. The introduction of named tuples into SASLOG will be a necessary prerequisite for the type concept; these tuples should be easy to implement and will allow constructors to be used throughout the Prolog and the SASL part as well.

8. Acknowledgments

We wish to thank Harold Boley who helped us in the earlier stages of the project and who provided the LISPLOG interpreter that forms an integral part of the SASLOG system.
We are also indebted to Gert Smolka whose valuable comments on the form and the contents of the paper we greatly appreciate.

9. Literature

[Barbuti, Bellia, Levi86]
Barbuti R., Bellia M., Levi G.: LEAF: A Language which integrates Logic, Equations and Functions,
in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Bellia, Levi86]
Bellia M., Levi G.: The Relation between Logic and Functional Languages: A Survey,
in: Journal of Logic Programming, Vol. 3, No. 3, 1986

[Boley85]
Boley H. and the LISPLOG group: LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung
MEMO SEKI-85-03, Universität Kaiserslautern, 1985

[Boley86]
Boley H. (Ed.): A Bird's-Eye view of LISPLOG: The LISP/PROLOG Integration with Initial-Cut Tools.
SEKI Working Paper SWP-86-08, Universität Kaiserslautern, 1985

[Clocksin, Mellish84]
Clocksin W.F., Mellish, C.S.: Programming in Prolog,
Berlin, 1984

[Darlington, Field, Pull86]
Darlington J., Field A.J., Pull H.: The Unification of Functional and Logic Languages,
in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Darlington, Henderson, Turner82]
Darlington J., Henderson P., Turner D.A. (eds.): Functional Programming and its applications, Cambridge, 1982

[Dincbas, van Hentenryck87]
Dincbas M., van Hentenryck P.: Extended Unification Algorithms for the Integration of Functional Programming into Logic Programming,
in: Journal of Logic Programming, Vol.4, No. 3, 1987

[Frisch, Allen, Giuliano83]
Frisch A.M., Allen J.F., Giuliano M.: An Overview of the HORNE Logic Programming System,
in: SIGART Newsletter, No. 84, April 1983

[Goguen, Meseguer86]
Goguen J.A., Meseguer J.: EQLOG: Equality, Types, and Generic Modules for Logic Programming,
in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Hinkelmann88]
Hinkelmann, K.: SASLOG: Eine funktional-logische Sprachintegration mit Lazy Evaluation und semantischer Unifikation
Diploma Thesis, Universität Kaiserslautern, FB Informatik, 1988

[Hughes84]
Hughes J.: Why functional programming matters,
Memo PMG-40, Göteborg, 1984

[Kahn, Carlsson83]
Kahn K.M., Carlsson M.: LM-Prolog User Manual,
UPMAIL, Department of Computing Science, Uppsala University, October 1983

[Kornfeld83]
Kornfeld W.: Equality for Prolog,
in: Proc. 8th IJCAI-83, Karlsruhe, Aufgust 1983,
also in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Narain86]
Narain S.: A Technique for Doing Lazy Evaluation in Logic,
in: Journal of Logic Programming, Vol.3, No. 3, 1986

[Nökel, Rehbold86]
Nökel, K., Rehbold, R.: SASL: Implementierung einer rein funktionalen Sprache mit Lazy Evaluation
SEKI Working Paper SWP-86-07, Universität Kaiserslautern, 1986

[Reddy86]
Reddy U.S.: On the Relationship between Logic and Functional Languages,
in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Robinson83]
Robinson J.A.: A Proposal to develop a "Fifth Generation" Programming System Based on Logic Programming and Highly Parallel Reduction Machine,
Logic Programming Research School of Computer and Information Science,
Syracuse University, New York, February 1983

[Robinson, Sibert82a]
Robinson J.A., Sibert E.: LOGLISP: Motivation, Design and Implementation,
in: Clark K., Tärnlund S.A.: Logic Programming,
Academic Press, London 1982

[Robinson, Sibert82b]
Robinson J.A., Sibert E.: LOGLISP: An Alternative to PROLOG.
in: Machine Intelligence 10, Chichester 1982

[Smolka86]
Smolka G.: FRESH: A Higher-Order Language Based on Unification,
in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Subrahmanyam,You86]
Subrahmanyam, P.A.,You, J.-H. : FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming
in: D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations
Prentice-Hall 1986

[Turner79]
Turner D.A.: A New Implementation Technique for Applicative Languages,
in: Software - Practise and Experience, Vol. 9(1), S. 31-49, 1979

[Turner82]
Turner D.A.: Recursion Equations as a Programming Language,
in: [Darlington, Henderson, Turner82]

[Turner83]
Turner D.A.: SASL Language Manual (revised version),
University of Kent, Canterbury, 1983

[Turner85]
Turner D.A.: Miranda: a Non-Strict Functional Language with Polymorphic Types
in: Functional Programming Languages and Computer Architectures
Springer Lecture Notes in Computer Science, vol. 201

[Wadler85]
Wadler P.: How to Replace Failure by a List of Successes,
in: 1985 Conference on Functional Programming and Computer Architecture,
Springer Verlag, Berlin 1985

Authors' Address:

Klaus Nökel, Robert Rehbold
Universität Kaiserslautern
FB Informatik
P.O.Box 3049
6750 Kaiserslautern
Fed. Rep. of Germany

Knut Hinkelmann
FAW Ulm
P.O.Box 2060
7900 Ulm
Fed. Rep. of Germany

UUCP: {noekel,rehbold}@uklirb.uucp
CSNET: {noekel,rehbold}%uklirb.uucp@ira.uka.de