503053

Purdue University Interlibrary Loan (IPL)

ILLiad TN: 1264766

ILL Number: 149195025

Borrower: IUL
Aging Date: 20150601
Transaction Date: 6/2/2015 9:44:47 AM
Lending String:
*IPL,EEM,UIU,EYM,UMC,GZM,UPM,INU,LHL,MNU,NJR,NUI,KUK,OUN,CWR

Patron:

Journal Title: 1984 International Symposium on Logic Programming, February 6-9, 1984, Bally's Park Place Casino, Atlantic City, New Jersey /

Volume: **Issue:**
Month/Year: 1984

Pages: 168--176

Article Author: Gary Lindstrom and Prakash Panangaden

Article Title: Stream-Based Execution of Logic Programming

Call #: 005 Sy685p 1984

Location: Hicks Repository

Ship Via: Odyssey
ODYSSEY ENABLED

Charge
Maxcost: 45.00IFM

Shipping Address:
Indiana University Libraries
DDS, Wells Library, E-065
1320 E 10th St
Bloomington IN US 47405-
Fax: 812 855-2576
Odyssey: 156.56.71.18
Ariel:
Email:
libILL@indiana.edu

COPY PROBLEM REPORT
Purdue University Libraries Interlibrary Loan Department (IPL)

Odyssey: 128.210.126.171
Ariel: 128.210.125.135
Fax: 765-494-9007
Phone: 765-494-2805

Please return this sheet within 5 BUSINESS DAYS if there are any problems.

_____ Resend legible copy

_____ Wrong article sent

_____ Other, explain ______________________________

DOCUMENTS ARE DISCARDED AFTER 5 DAYS. PLEASE RESUBMIT REQUEST AFTER THAT TIME.

STREAM-BASED EXECUTION OF LOGIC PROGRAMS[*]

GARY LINDSTROM[**] and PRAKASH PANANGADEN

Department of Computer Science, University of Utah, Salt Lake City, Utah 84112

ABSTRACT

We report a new execution model for Horn clause logic programs, combining the following key features:

- a stream-based analog of the 'standard' backtracking execution model;
- OR-parallelism, with a particular form of induced AND-parallelism;
- an applicative formulation, except for indeterminate stream merging;
- concurrent processing of several top-level goals, if desired, and
- a 'pure code' utilization of program clauses, with all instantiation done via composition of substitution records.

The model is informally sketched, and then formally defined by means of a skeletal interpreter expressed in the Function Equation Language FEL.

MOTIVATION

Logic programming (LP) is currently attracting considerable attention as a means for exploiting the highly parallel computer architectures of the next generation. Its appeal is also due in part to its promise as a meeting ground for two important software 'cultures': artificial intelligence (by 'codifying' production-style control) and functional programming (by retaining an applicative semantic basis).

LP involves a declarative programming style based on procedural interpretations of logic sentences. The sentence form most commonly studied in this regard is *clausal*, i.e. quantifier-free first-order constructs of the form

$$A_1, \ldots, A_n \text{ :- } B_1, \ldots, B_m,$$

where the A_i and B_i are *atomic formulae* (or simply *formulae*) involving literal predicate names and arguments which are constants and variables, or functions thereon, possibly nested. The statement thereby denoted is

$$B_1 \wedge \ldots \wedge B_m \supset A_1 \vee \ldots \vee A_n.$$

A particularly clean procedural interpretation may be made on clauses in the case that n is one[1]. Such clauses are termed *Horn clauses*, and form the basis for most existing LP languages, notably Prolog[2].

In a Horn clause such as

$$A \text{ :- } B_1, \ldots, B_m,$$

A is termed the *head* and $B_1, \ldots, B_m$ the *body* of the clause. If m is zero, the clause is termed an *assertion*, and A is considered to be unconditionally true.

We are currently designing a pipelined evaluator for Horn clause logic programs, to be run on a novel applicative computer architecture named *Rediflow*[3]. Our motivations in this effort include the following:

1. A desire to utilize the large-scale concurrency implicit in logic programs as a significant test of the deliverable computational power in Rediflow.

2. Special interest in parallel search as an application for architectures such as Rediflow.

3. A conjecture that Rediflow's hybrid evaluation method will be particularly apt for the implementation of LP languages. This method combines the two principal evaluation methods for functional programs[4]:

    - *reduction* (or 'structural'): involving a dynamic graph program representation, with unique values associated with each arc, and

    - *dataflow* (or 'token'): utilizing a static graph program representation, with temporally ordered streams of values associated with each arc.

4. A desire to directly interpret un-annotated logic programs (e.g. those without mode declarations or control pragmas), through an execution model embodying the 'standard' Prolog semantics of substitution and backtracking.

RELEVANT PREVIOUS WORK

Before presenting our evaluation mode, we pause to outline other execution models for Horn clause logic programs, not attempting to be comprehensive. The execution models that we shall discuss are the 'standard' backtracking execution model due

[*] This research was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4305, under contract No. MDA 903-81-C-0411, issued by the Defense Supply Service, Washington, DC 20310.

[**] Currently on sabbatical leave at the Laboratory for Computer Science, Massachusetts Institute of Technology.

to Kowalski[5], PARLOG due to Clark and Gregory[6], the backup parallelism execution model of Matsumoto, Nitta and Furukawa[7], and the OR-parallel execution model of Umeyama and Tamura[8].

The original backtracking model is purely sequential. The sequence of clauses comprising the program are viewed as procedure definitions. The program is invoked by a *goal* statement, which has the form of a (degenerate) clause with no head. Unification is attempted with each clause in turn. The first successful unification results in the generation of a sequence of new goal statements that must be now satisfied. This process is viewed as *procedure invocation*. Failure to unify with any clause will cause the execution to backtrack to the last unified clause choice, and unification will be attempted with the next clause at that point. The backtracking mechanism can also be used to generate multiple solutions to a given goal by inducing top-level failures. In our parallel execution model the concurrent search for multiple solutions emulates this backtracking mechanism in a concurrent fashion.

PARLOG is a language for logic programming which allows a user to control the degree of parallelism through the use of high level primitives. Both AND-parallelism and OR-parallelism are possible in PARLOG. The form of parallelism to be used in computing a relation is specified by the user through clause annotation. Further annotations control the degree of parallelism. In AND-parallel programs the evaluation of a program finds *one* instance of the relation defined by the program. In other words AND-parallelism in PARLOG feature 'don't care' (committed choice) nondeterminism. When variables are shared among the formulae of a clause in an AND-parallel program one of the formulae is delared to be the *producer* of the bindings for that variable and the other clauses are designated as *consumers* of those bindings. In our model we do not directly attempt AND-parallelism; rather, an 'induced' form of AND parallelism arises due to the eager reporting of bindings made by OR-parallel goals. Thus the leftmost occurrence of each variable in a clause body is the 'producer' of its bindings, and we avoid dealing with issues of mode declarations. The OR-parallel relations in PARLOG feature 'don't know' nondeterminism. The *set* of *all* solutions to a given goal is computed but the order in which solutions are obtained cannot be predicted. OR relations can be invoked by AND relations through a set expression construct in the language. The evaluation of these set expressions can be controlled by the user to be either *eager*, which leads to OR-parallelism, or *lazy* which would lead to a traditional sequential evaluation mechanism. PARLOG also provides a facility for defining functions via conditional equations. Our form of OR-parallelism is quite similar to this.

In the backup parallelism model of Matsumoto et al., each node of the AND/OR tree is viewed as a process. Each AND(OR) process activates a number of descendant OR(AND) processes. When a descendant process has returned a solution to its activating process it immediately begins searching for another solution so that it can produce the next solution as soon as the activating process needs it. If an additional solution is found and the activating process does not need one then the descendant process suspends until it is reactivated. Thus one level of OR-parallelism is maintained throughout the AND/OR tree. A process which returns a failure is deleted from the AND/OR tree. Our model also views each node as a process but the level of OR-parallelism is not constrained to be one. There is an AND/OR process tree in the Matsumoto approach, as in our model, but the streaming aspect is present in a somewhat form.

In the OR-parallel model of Umeyama and Tamura, each *clause* is viewed as a single process. The processes communicate with each other via two-way channels through which goal statements and solutions flow. The logic program thus defines a *static* program graph. If the program contains the clause

$$A(...) :- B(...), C(...), D(...).$$

the program graph will show the process for this clause connected to all processes which correspond to clauses with head either B or C or D. OR-parallelism is achieved by having each process send out copies of a goal statement to each of the possible clauses it could unify with. The solutions are processed as they are received so there is an induced AND-parallelism as in our model. This model captures the dataflow aspects of our model but it involves additional complexities and possible bottlenecks due to the 're-entrant' use of unique processes associated with each clause.

The lecture notes of Abelson and Sussman[9] also approach Horn clause logic program execution via streaming of goal and solution objects, which they term 'frames'. These are then 'extended' and 'filtered' at various stages of processing. However, no effort is made to exploit concurrency, and hence searches on multiple goals cannot be overlapped. Substitution composition, which is the heart of our method, is not overtly employed.

THE MODEL

We now present our stream-based evaluation model for Horn clause LP. The exposition will comprise three steps: an intuitive presentation, a very simple illustration of the method in use, and a formal framework.

An intuitive presentation

Our method is based on the following ideas:

1. The *streaming* of objects representing goals and solutions, through an

2. *AND/OR tree* of transducers, utilizing

3. *pure code* application of clauses to substitution records.

Streaming of goal and solution objects. A logic program can be viewed as a stream transducer, in the following sense.

Consider, for the present, the special case where all goals inputted to our logic program involve a fixed formula $\mathcal{F}_{top}$, being solved under various initial substitutions. We term $\mathcal{F}_{top}$ the *top-level* formula. For illustration, let $\mathcal{F}_{top}$ be FatherOf(U, V). (We follow the Prolog convention that non-functor identifiers beginning with uppercase letters are variables, and all others are constants.) We denote by $\mathcal{V}(\mathcal{F})$ the set of variables appearing in a formula $\mathcal{F}$, i.e. the *name space* of $\mathcal{F}$.

- *Goals* are denoted by objects containing *substitutions* on $\mathcal{V}(\mathcal{F}_{top})$, e.g.

$$[U := \text{cain}, V := V];$$

this denotes "simultaneously substitute all occurrences of U with cain, and all occurrences of V with V, i.e. leave all V occurrences uncommitted". (The purpose of such identity bindings will become clear when we later do compositions on substitutions.)

- *Solutions* are represented similarly, but typically with constants substituted for all variables, e.g.

  [U := cain, V := adam].

  If $\mathcal{S}$ is a substitution in a solution object, then $\mathcal{S}$ applied to $\mathcal{F}_{top}$ (denoted $\mathcal{F}_{top} \circ \mathcal{S}$) is a theorem.

- A logic program may then be viewed as a *stream transducer* if it:

  1. accepts a stream of goal objects;

  2. produces a stream of solution objects associated with those goal objects, and

  3. if the goal stream is terminated by an end-of-stream object Λ, the logic program will, resources permitting, eventually produce all solution objects corresponding to the given goal objects, followed by a Λ.

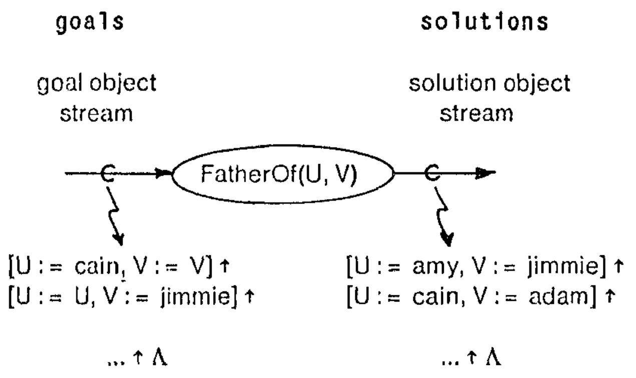


Figure 1. A logic program as a stream transducer.

Figure 1 depicts this top-level stream flow, where ↑ indicates the 'followed by' operator. Observe:

1. Several goal objects may be processed concurrently by the logic program, with each resulting in zero or more solution objects.

2. The serial order in which these solution objects are returned is not considered significant.

3. In particular, the solution objects associated with different goal objects may appear in interleaved order.

4. However, since each solution object echoes the bindings of a particular goal object, there is no danger of associating a solution object with an inappropriate goal object.

An AND/OR tree of stream transducers. To explain how such an overall streaming behavior can be achieved, we introduce the expository device of a (possibly infinite) AND/OR tree of stream transducers. The need for such an explicit transducer tree will be eliminated in a later section, where a more economical 'packet-based' approach is sketched.

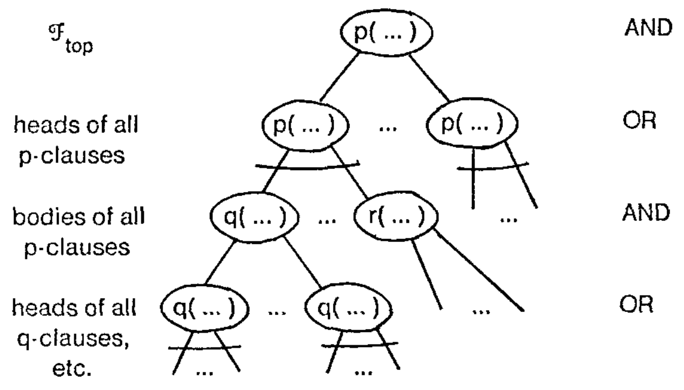


Figure 2: A 'syntactic' AND/OR tree.

Consider, to begin, a 'syntactic' AND/OR tree, as depicted in figure 2. In this tree:

- Each *node* is associated with a formula.

- The *root* node is associated with $\mathcal{F}_{top}$, and is an AND node (we adopt the convention of classifying AND and OR nodes in terms of the relation each bears to its siblings).

- Each *AND node* (other than the root) is associated with a formula appearing in the body of a clause. If that formula is for example $p(t_1, ..., t_k)$, the node has a descendant OR node for each clause head $p(t_1', ..., t_k')$. (For simplicity, we do not *a priori* 'prune' any obviously nonunifiable descendants.)

- Each *OR node* is thus associated with a formula occurring as the head of a clause, and has as descendants AND nodes associated with each formula, in left to right order, appearing in the body of its clause.

Nodes as stream transducers. Within this framework, we now view each node as a stream transducer communicating with its neighboring nodes as shown in figure 3. Notice the arcs are named in a standardized way, i.e. `goals`, `solutions`, `subgoals`, and `subsolutions`. Of course, each arc in general has *two* names, reflecting its dual purpose from the differing viewpoints of its source and destination nodes.

Note that in a subtree associated with a clause

$$A :- B_1, ..., B_m,$$

its stream interconnection is as suggested by

$$... \Rightarrow A \Rightarrow B_1 \Rightarrow ... \Rightarrow B_m \Rightarrow A \Rightarrow ... .$$

Assertions have the degenerate interconnection

$$... \Rightarrow A \Rightarrow A \Rightarrow ... .$$

The information flowing among the transducers is summarized in figure 4, in which a sample object is shown on each connecting arc. This figure bears careful study, since it captures the central idea underlying our method.

Notice first that the objects flowing among the transducers are *pairs* of subobjects, rather than simply single substitution objects as suggested earlier. These composite objects are termed ApplicationObjects, and observe the following recursive syntax:

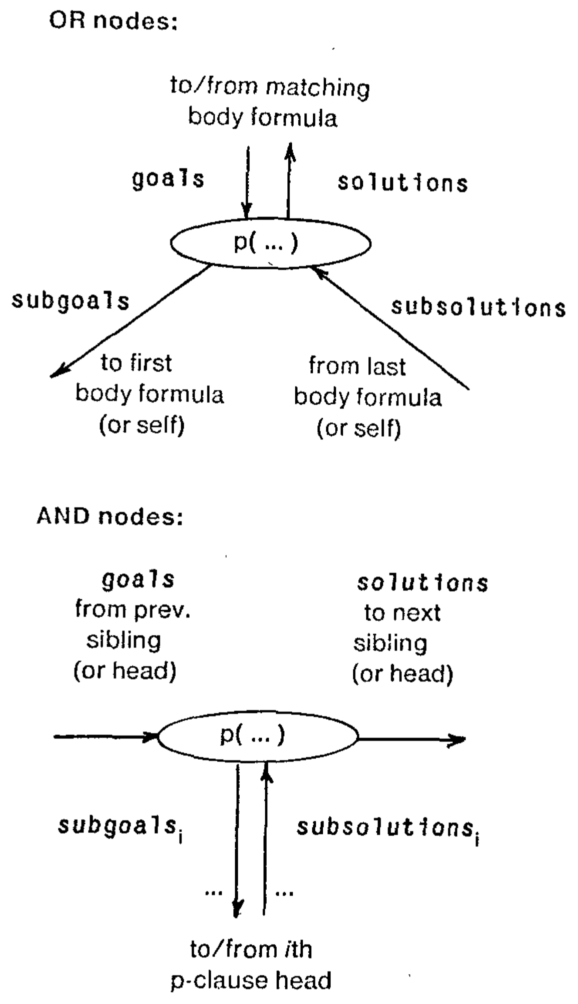
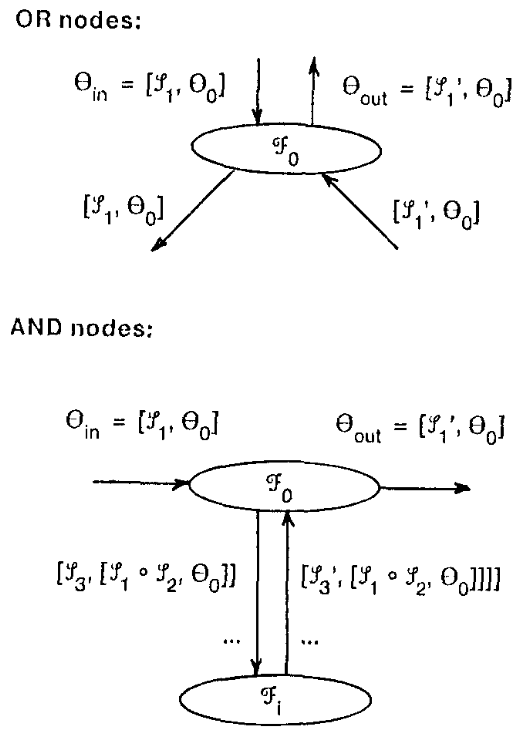

OR nodes:
to/from matching body formula
goals
solutions
p(...)
subgoals
subsolutions
to first body formula (or self)
from last body formula (or self)
AND nodes:
goals from prev. sibling (or head)
solutions to next sibling (or head)
p(...)
subgoals$_i$
subsolutions$_i$
...
...
to/from ith p-clause head

Figure 3. Individual tree nodes as transducers.


OR nodes:
$\Theta_{in} = [\mathcal{S}_1, \Theta_0]$
$\Theta_{out} = [\mathcal{S}_1', \Theta_0]$
$\mathcal{F}_0$
$[\mathcal{S}_1, \Theta_0]$
$[\mathcal{S}_1', \Theta_0]$
AND nodes:
$\Theta_{in} = [\mathcal{S}_1, \Theta_0]$
$\Theta_{out} = [\mathcal{S}_1', \Theta_0]$
$\mathcal{F}_0$
$[\mathcal{S}_3, [\mathcal{S}_1 \circ \mathcal{S}_2, \Theta_0]]$
$[\mathcal{S}_3', [\mathcal{S}_1 \circ \mathcal{S}_2, \Theta_0]]]]$
...
...
$\mathcal{F}_i$

Notes:

1. Let $\mathcal{S}_1$ be $[V_1 := t_1, \ldots, V_n := t_n]$. Then $\mathcal{S}_1 \circ \mathcal{S}_2$ denotes the composition of $\mathcal{S}_1$ and $\mathcal{S}_2$, i.e. $[V_1 := t_1 \circ \mathcal{S}_2, \ldots, V_n := t_n \circ \mathcal{S}_2]$.

2. The unification of $\mathcal{F}_0 \circ \mathcal{S}_1$ and $\mathcal{F}_i$ yields the substitution pair $[\mathcal{S}_2, \mathcal{S}_3]$, such that $\mathcal{F}_0 \circ \mathcal{S}_1 \circ \mathcal{S}_2 = \mathcal{F}_i \circ \mathcal{S}_3$.

3. $\mathcal{S}_1' = \mathcal{S}_1 \circ \mathcal{S}_2 \circ \mathcal{S}_3'$, where $\mathcal{S}_3'$ is a refinement of $\mathcal{S}_3$.

Figure 4. Substitution data flow.

ApplicationObject	→ [Substitution, ParentLink]
ParentLink	→ ApplicationObject | []
GoalObject	→ ApplicationObject
SolutionObject	→ ApplicationObject

The following remarks apply to both AND and OR nodes.

1. Consider the GoalObject $\Theta_{in} = [\mathcal{S}_1, \Theta_0]$ received along the **goals** arc in each case. Here $\mathcal{S}_1$ is a substitution on the variables of $\mathcal{F}_0$, and indicates that $\mathcal{F}_0 \circ \mathcal{S}_1$ is a goal to be solved.

2. Similarly, the SolutionObject $\Theta_{out} = [\mathcal{S}_1', \Theta_0]$ shown *departing* along the **solutions** arc indicates that $\mathcal{F}_0 \circ \mathcal{S}_1'$ is a theorem. Here $\mathcal{S}_1'$ is a *refinement* of $\mathcal{S}_1$, i.e. there exists some $\mathcal{S}_r$ such that $\mathcal{S}_1 \circ \mathcal{S}_r = \mathcal{S}_1'$.

3. In each case, Λ objects arriving along the **goals** arc may eventually give rise to a Λ object departing along the **solutions** arc. This signals that *all* SolutionObjects associated with all GoalObjects received at this node have been found and passed on.

We now consider the processing logic particular to each of the AND and OR node cases. In so doing, the role of ParentLinks will become clear.

- *OR nodes:*

   1. *Transformations:* As figure 4 suggests, OR nodes serve merely as ApplicationObject routers, with no transformations involved.

   2. Λ *signaling:* A Λ received by an OR node along its **goals** arc is passed along its **subgoals** arc, and propagated along its **solutions** arc when received along its **subsolutions** arc.

- *AND nodes:*

   1. *Transformations:*

      * *Downward flow:* When a Θ_{in} is received by an AND node along its **goals** arc, an image of it is passed along each of its **subgoals**$_i$ arcs if possible. Let $\mathcal{F}_0' = \mathcal{F}_0 \circ \mathcal{S}_1$. Then:

If $\mathcal{G}_0{}'$ unifies with $\mathcal{G}_i$, let $\mathcal{S}_2$ and $\mathcal{S}_3$ be resulting substitutions such that

$$\mathcal{G}_0{}' \circ \mathcal{S}_2 = \mathcal{G}_i \circ \mathcal{S}_3.$$

Moreover, we require *range*($\mathcal{S}_2$) $\subseteq$ *domain*($\mathcal{S}_3$), where *domain*($\mathcal{S}$) (resp. *range*) is the set of variables appearing on the *left* (resp. *right*) of assignments in $\mathcal{S}$. The derived GoalObject

$$[\mathcal{S}_3, [\mathcal{S}_1 \circ \mathcal{S}_2, \Theta_0]]$$

is then passed along the **subgoals**$_i$ arc.

If the unification attempt on $\mathcal{G}_i$ fails, no derived image of Θ_{in} is passed along the **subgoals**$_i$ arc.

* *Upward flow:* The node's *i*th descendant will deliver along the **subsolutions**$_i$ arc zero or more SolutionObjects

$$[\mathcal{S}_3{}', [\mathcal{S}_1 \circ \mathcal{S}_2, \Theta_0]]$$

associated with Θ_{in}. From each of these, we construct the derived SolutionObject $[\mathcal{S}_1{}', \Theta_0]$, where

$$\mathcal{S}_1{}' = \mathcal{S}_1 \circ \mathcal{S}_2 \circ \mathcal{S}_3{}'.$$

2. Λ *signaling:* A Λ received along the **goals** arc by an AND node is propagated downward along *all* its **subgoals**$_i$ arcs. However, the Λ is propagated onward along its **solutions** arc only when *all* its **subsolutions** arcs have echoed the Λ.

Since the effects of substitution composition in this approach are so crucial, we now provide further intuition on how it is used to obtain the desired results. In so doing, we will focus particularly on the name spaces involved in each step.

- As observed above, $\mathcal{G}_0 \circ \mathcal{S}_1$ is the 'generic' goal $\mathcal{G}_0$ mapped into a specific instance via the variable bindings in $\mathcal{S}_1$. We term $\mathcal{N}(\mathcal{G})$ the *native* name space of $\mathcal{G}_0$, and $\mathcal{N}(\mathcal{G}_0 \circ \mathcal{S}_1)$ the *instantiated* name space of $\mathcal{G}_0$.

- $\mathcal{S}_2$ further particularizes $\mathcal{G}_0 \circ \mathcal{S}_1$ to reflect its unification with $\mathcal{G}_0 \circ \mathcal{S}_1$ with $\mathcal{G}_i$. We term $\mathcal{N}(\mathcal{G} \circ \mathcal{S}_1 \circ \mathcal{S}_2)$ the *unified* name space of $\mathcal{G}$.

- The matching particularization of $\mathcal{G}_i$ is contained in $\mathcal{S}_3$, which provides $\mathcal{G}_i$'s *instantiated* name space for this goal.

- Each solution $\mathcal{S}_3{}'$ to this goal is a refinement of $\mathcal{S}_3$, so *domain*($\mathcal{S}_3$) = *domain*($\mathcal{S}_3{}'$). Moreover, since *range*($\mathcal{S}_2$) $\subseteq$ *domain*($\mathcal{S}_3$), the solution conveyed by $\mathcal{S}_3{}'$ may be *restated* in terms of a new instantiated name space for $\mathcal{G}_0$ by the composition $\mathcal{G}_0 \circ \mathcal{S}_1 \circ \mathcal{S}_2 \circ \mathcal{S}_3{}'$.

In passing the restated solution along an AND node's **solutions** arc, it is packaged in a SolutionObject Θ_{out} with the Θ_0 that appeared in its associated GoalObject Θ_{in}. Hence the ParentLinks in Application objects serve as a kind of dynamic binding stack, keeping a chain of solution restatement mappings to be applied as subsequent subsolutions are mapped upward to AND nodes.

In general, note that the clauses involved at each level in the AND/OR tree are referenced in a 'read only' fashion, and are not copied or explicitly modified. All instantiation is accomplished in a 'virtual' fashion by composition of substitution objects.

An illustration

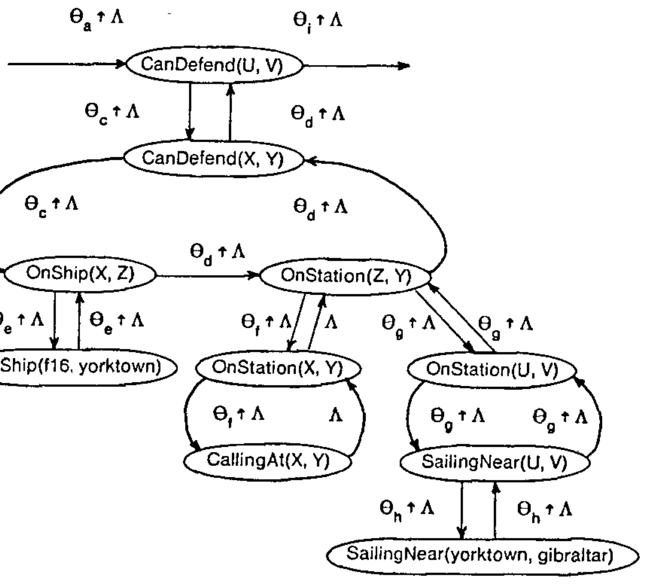
We illustrate our method through a very simple example. Consider the following clause set:

CanDefend(X, Y) :- OnShip(X, Z), OnStation(Z, Y).
OnStation(X, Y) :- CallingAt(X, Y).
OnStation(U, V) :- SailingNear(U, V).
OnShip(f16, yorktown).
CallingAt(forrestal, manila).
SailingNear(yorktown, gibraltar).

Let $\mathcal{G}_{top}$ be CanDefend(U, V), and the top-level GoalObject stream be

$$[[U := U, V := \text{gibraltar}], []] \uparrow \Lambda.$$

Figure 5 shows the resulting transducer streams throughout the tree.



$\Theta_a \uparrow \Lambda$
$\Theta_i \uparrow \Lambda$
CanDefend(U, V)
$\Theta_c \uparrow \Lambda$
$\Theta_d \uparrow \Lambda$
CanDefend(X, Y)
$\Theta_c \uparrow \Lambda$
$\Theta_d \uparrow \Lambda$
$\Theta_d \uparrow \Lambda$
OnShip(X, Z)
OnStation(Z, Y)
$\Theta_e \uparrow \Lambda$
$\Theta_e \uparrow \Lambda$
$\Theta_f \uparrow \Lambda$
Λ
$\Theta_g \uparrow \Lambda$
$\Theta_g \uparrow \Lambda$
OnShip(f16, yorktown)
OnStation(X, Y)
OnStation(U, V)
$\Theta_f \uparrow \Lambda$
Λ
$\Theta_g \uparrow \Lambda$
$\Theta_g \uparrow \Lambda$
CallingAt(X, Y)
SailingNear(U, V)
$\Theta_h \uparrow \Lambda$
$\Theta_h \uparrow \Lambda$
SailingNear(yorktown, gibraltar)

Where:

Θ_a	= [[U := U, V := gibraltar], []]
Θ_b	= [[U := X, V := gibraltar], []]
Θ_c	= [[X := X, Y := gibraltar, Z := Z], Θ_b]
Θ_d	= [[X := f16, Y := gibraltar, Z := yorktown], Θ_b]
Θ_e	= [[], Θ_d]
Θ_f	= [[X := yorktown, Y := gibraltar], Θ_d]
Θ_g	= [[U := yorktown, V := gibraltar], Θ_d]
Θ_h	= [[], Θ_g]
Θ_i	= [[U := f16, V := gibraltar], []].

Figure 5. A simple example.

A formal framework

Although a full definition of our Horn clause interpreter is beyond the scope of this paper, we can examine its central organization as expressed in the Function Equation Language FEL.[10]

Data objects. We adopt the data structures related to ApplicationObjects already defined. The remaining data objects of interest are simply those involved in clause representation:

Clause	→	[Head, Body]
Head	→	Formula
Body	→	FormulaList
FormulaList	→	Formula ↑ FormulaList] | []

Principal functions. A functional rendering of the stream transducer logic may be seen in figure 6. Notationally, FEL resembles ISWIM.[11] Block expressions are denoted

{*equations* RESULT *expression*},

where the *equations* define locally bound names, generally used within the result expression. Of particular interest are equations of the form

$$[name_1, ..., name_k] = expression,$$

and

$$name_1 \uparrow name_2 = expression,$$

which provide for data structure (i.e. tuple and stream) unpacking without explicit use of selector operations. In addition:

1. Right associative function application is denoted by an infix colon, i.e. f:x:y = f:(x:y).

2. "|" denotes left associative function 'Currying,' i.e. f|x|y = (f:x):y. When used in a function heading, e.g.

   or_node|*formulalist*|th1_str,

   a Curried or '2-tiered' function is defined. Hence or_node may be called in expressions of the form

   (or_node:f):t,

   or simply or_node|f|t.

3. "||" in FEL denotes stream-application (i.e. 'stream mapcar'), i.e. f||(x↑y) = f:x ↑ f||y, and f||[] = [].

The principal functions shown are and_node, right_side, and or_node. Note these functions are constructed in Curried form, with their GoalObject streams as the final ingredient in each 'Curry.' By binding the first argument of each to be formulae from a particular clause, we obtain a specialized transducer which can be used repeatedly on differing GoalObjects. This is how the desired AND/OR tree is constructed. A few other words of guidance in studying figure 6:

- or_node: The first argument to this function is a clause body, used to establish its sequence of AND node descendants. The local function restate accomplishes the subsolution restatement discussed above.

- right_side: This actually establishes the sequence of AND node functions necessary to stream ApplicationObjects through the transducers associated with a clause body occurrence.

- and_node: The function unify (not shown) applies each given GoalObject to the appropriate or_node transducers. Meanwhile, the merge pseudo-function permits additional GoalObjects to be entertained on an eager basis.

The root transducer is established by a call on and_node, which accepts a conjunction of goal formulae, as a 'psuedo' clause body.

```
FUNCTION or_node|formulalist|   (* right side of clause *)
              th1_str           (* stream of application
                                     objects for this clause *)
              =                 (* result is stream of solution
                                     objects, restated in
                                     parent nomenclature *)

     {son_str = right_side|formulalist|th1_str,

     FUNCTION restate|th1       (* final AND son solution
                                     object *)
              =                 (* result is same solution
                                     object, restated in
                                     parent nomenclature *)
              {[subst1, th0] = th1,
              [subst0, th_parent] = th0,
              RESULT  [compose:[subst0, subst1], th_parent]}

     RESULT  restate||son_str}

FUNCTION right_side|formulalist|  (* right hand side of clause *)
              th1_str             (* stream of application
                                          objects*)
     =                            (* returns stream of refined
                                          application objects *)
     {[formula, f_l_tail] = formulalist,
     RESULT  IF formulalist = []
             THEN    th1_str
             ELSE    right_side|f_l_tail|
                       (and_node|formula|th1_str)}

FUNCTION and_node|formula|  (* goal formula *)
              th1_str       (* stream of application objects *)
     =                      (* returns stream of refined
                                  application objects *)
     {th1 ↑ th1_str_tail = th1_str,
     RESULT  IF th1_str_tail = []
             THEN    []
             ELSE    merge:[unify|formula|th1,
                                and_node|formula|th1_str_tail]}
```

Figure 6. FEL code, explicit AND/OR tree case.

ALGORITHMIC DETAILS

Special unification approach

As suggested earlier, the reliance of this method on substitution composition poses some special requirements on how unification should be done. Figure 7 summarizes these requirements, on the base conditions where one of the terms is a variable. Since our unification process is asymmetric, $t_1 \Leftrightarrow t_2$ should be interpreted as 'the unification of goal term t_1 with head term t_2, yielding the substitution pair $[\mathcal{S}_{upper}, \mathcal{S}_{lower}]$.'

Variable clashes

As all Prolog implementors are aware, the binding effects of Horn clause logic programs can be very subtle when dealing with functions. This fact is reflected in our note 4 in figure 7, which provides for name translation when functional terms involving unbound variables are propagated downward. The introduction of Z' in the example illustrates this effect; if this were not done, the Z in f(Z) would conflict with the unbound Z already existing in the lower level term. However, the scope of name uniqueness required is not global -- simply such that *range*($\mathcal{R}(\alpha)$) is disjoint from *domain*($\mathcal{S}_{lower}$).

In fact, a similar problem can arise when unbound variables appear in *SolutionObjects. For example, consider the GoalObject*

$$[[U := U, V := V, W := X], \Theta]$$

applied to FatherOf(U, V). If unified with the assertion

FatherOf(X, dad(X)),

Cases:

$X \Leftrightarrow a$:	$[[X := a], []]$
$a \Leftrightarrow X$:	$[[], [X := a]]$
$X \Leftrightarrow Y$:	$[[X := Y], [Y := Y]]$
$X \Leftrightarrow f(\alpha)$:	$[[X := f(\alpha)], \mathit{ident}(\alpha)]$
$f(\alpha) \Leftrightarrow X$:	$[\mathfrak{R}(\alpha), [X := f(\alpha \circ \mathfrak{R}(\alpha))] \cup \mathit{ident}(\alpha \circ \mathfrak{R}(\alpha))]$

Example:

$p(X, a, f(Z), Y) \Leftrightarrow p(Z, X, Y, g(X)) =$

$$[[X := Z, Z := Z', Y := g(a)],$$
$$[Z := Z, X := a, Y := f(Z'), Z' := Z']].$$

Notes:

1. *ident*(α) denotes the identity substitution on the variables of α.

2. $\mathfrak{R}(\alpha)$ denotes a substitution mapping the variables of α to distinct new names.

3. $\cup$ denotes substitution concatenation, e.g. $[X := \alpha] \cup [Y := \beta] = [X := \alpha, Y := \beta]$. $\mathcal{S}_1 \cup \mathcal{S}_2$ is defined only when *domain*($\mathcal{S}_1$) and *domain*($\mathcal{S}_2$) are disjoint.

4. Identity substitutions are also provided in $\mathcal{S}_{lower}$ for clause variables not appearing in the head, e.g. Z in clause CanDefend(X, Y) :- OnShip(X, Z), OnStation(Z, Y).

Figure 7. Unification effects.

the resulting SolutionObject would be

$$[[U := X, V := dad(X), W := X], \Theta],$$

which falsely equates U and W. This situation arises only when *range*($\mathcal{S}_3$') overlaps with the variables of $\mathcal{S}_1$ not involved in the unification, i.e. *range*($\mathcal{S}_1$) $\cdot$ *domain*($\mathcal{S}_2$). If this occurs, renaming of clashing variables in *range*($\mathcal{S}_3$') must be performed. Again, the required scope of name uniqueness is limited to a single clause instance.

CONCURRENCY EFFECTS

Parallelism obtained

The method described here clearly exploits OR parallelism by propagating GoalObjects concurrently to all unifiable descendants of AND nodes. Moreover, an 'induced' form of AND parallelism also results, in the following sense. A merge operator within unify (not shown) indeterminately merges all subsolution streams at an AND node and eagerly passes each subsolution to the node's right sibling. Hence the right sibling may begin processing that subsolution while additional subsolutions are still being sought by its predecessor.

In addition, as noted earlier, all nodes, both AND and OR, can eagerly entertain new GoalObjects even if previous GoalObjects are still being processed.

Divergence control

It is worth noting, however, that the left to right search order within AND node siblings provides a degree of divergence control within this eager approach. That is, each AND node serves as a filter on subsolutions passed to its right siblings, exactly as is the case in a sequential implementation. Only the divergence control due to clause ordering within sequential implementations is lost. Hence one may conclude that the divergence characteristics of our approach will be the same in a sequential implementation *if* the sequential implementation is forced to give exhaustive solutions.

Conversion to Kahn processes

Throughout this paper, we have carefully used term 'streaming' rather than 'pipelining' to describe the transducer communication envisioned. This is because under the Rediflow 'lazy' evaluation method, stream order does not imply temporal order of computation. However, we have designed the method such that stream evaluation in serial order is permissible. This paves the way for conversion of node functions to 'Kahn Processes,' i.e. functionally encapsulated temporal stream transducers[12, 13]. The principal advantage of this functional form is its compatibility with reduction style execution, while providing more efficient space utilization. The recent dissertation by Tanaka[14] gives details.

ELIMINATING THE EXPLICIT AND/OR TREE.

The AND/OR tree of transducers we have described at length, while conceptually attractive, would be uneconomical to implement directly. Instead, the clear preference is to create multiplexible processes which can emulate transducers located anywhere in the logical tree. This can be done by extending ApplicationObjects to include code pointers (i.e. clause body tails), which are the only contextual information not included in our previous ApplicationObjects. One possible such revised data object format is:

$$\Pi = [\text{FormulaList}, [\mathcal{S}_3, \mathcal{S}_1 \circ \mathcal{S}_2], \Pi_0]$$

One then could have as many transducer processes as deemed physically appropriate. Each would draw from and deliver to a pool of such 'ApplicationPackets' in an indeterminate manner. For example, if the packet

$$\Pi_2 = [[], [\mathcal{S}_3', \mathcal{S}_1 \circ \mathcal{S}_2], \Pi_1]$$

were drawn from the pool, with

$$\Pi_1 = [\mathfrak{F} \uparrow \mathfrak{T}, [\mathcal{S}_1, \mathcal{S}_4], \Pi_0],$$

we would have an $\mathfrak{F}$ success, and the packet

$$[\mathfrak{T}, [\mathcal{S}_1 \circ \mathcal{S}_2 \circ \mathcal{S}_3', \mathcal{S}_4], \Pi_0]$$

would be returned to the pool. If the packet

$$\Pi_2 = [\mathfrak{F} \uparrow \mathfrak{T}, [\mathcal{S}_3, \mathcal{S}_4], \Pi_1]$$

were drawn, all packets of the form

$$[\mathfrak{B}, [\mathcal{S}_6, \mathcal{S}_3 \circ \mathcal{S}_5], \Pi_2]$$

would be returned to the pool, for each clause $\mathfrak{F}'$:- $\mathfrak{B}$, with

$$\mathfrak{F} \circ \mathcal{S}_3 \Leftrightarrow \mathfrak{F}' = [\mathcal{S}_5, \mathcal{S}_6].$$

Combining reduction and dataflow.

The suitability of our method for a dataflow implementation should be clear. However, we also believe the reduction mode of our hybrid evaluator will also pay dividends in a full implementation, by facilitating:

- the distributed construction of dynamic data structures, such as our chained ApplicationObjects;

- the sharing of intermediate results within such chained structures, e.g. the composition $\mathcal{I}_1 \circ \mathcal{I}_2$ in figure 4, needed by all Θ_{out} SolutionObjects derived from a given GoalObject Θ_{in};

- lazy evaluation effects, e.g. whereby $\mathcal{I}_1 \circ \mathcal{I}_2$ is evaluated for a given subgoal only when a subsolution using it is actually found and needed;

- flexible patterns of concurrency in auxiliary functions, notably the unification algorithm, and

- the use of structural lists to provide buffers between temporal stream transducers, as suggested by Tanaka[14].

EXTENSIONS AND FUTURE RESEARCH

One clear direction for the continuation of our work is the full demonstration of its merit, in quantitative terms, on the Rediflow simulator. However, several other interesting research directions are also indicated.

Compiled substitutions

Our reliance on goal and solution representation via substitution objects has been by motivated our need for an applicative, i.e. side-effect free, execution model. This simplifies concurrency control, at the cost of losing such von Neumann economies as 'structure sharing'.[15]

However, we conjecture an applicative analog of structure sharing can be found, which would provide:

- efficient substitution application (e.g. through a vectorized representation);

- elimination of name conflict problems, and

- localized copying cost when each subsolution is created.

This issue is intimately related to the issue of efficient concurrent unification on applicative architectures, which we judge to be an important open question.

Subgoal failure signaling

Throughout this paper we have made the simplifying assumption that failure (Λ) signals are used only to indicate termination of the solution stream for top-level goals. Hence Λ propagation is done only at 'system shutdown.' However, if this method is so successful that it might be used as a central 'utility' for logic program execution, then a need will arise to be able to report when all solutions for a *particular* top-level goal have been reported. We have extended our method to accommodate this, through a counting-based signaling technique integrated with the ApplicationPackets. However, discussion of this technique is left to a subsequent paper.

Concurrency control

The eager OR parallelism of our model must, in practice, be moderated by some caution exerted by the programmer. For example, the backtracking fence *cut* can be incorporated into our model in a partial way, by permitting only one subsolution per goal to propagate through a *cut* pseudo-formula. However, this leaves open the problem of garbage collecting the subgoals thereby rendered superfluous.

In general, we find such control forms as those in Concurrent Prolog[16] to be attractive, and are studying their possible inclusion within our framework.

Caching

Finally, we note that the potentially exponential cost of some backtracking searches, even when done with some degree of concurrency as we propose here, can be controlled by judicious use of previous result caching. We plan to investigate some combination of the ideas in[17] and [18] toward this end.

ACKNOWLEDGEMENTS

We gratefully acknowledge the insights and criticisms of Robert M. Keller and Uday Reddy in shaping this work.

REFERENCES

1. R. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, Vol. 22, No. 7, July 1979.

2. P. Roussel, "Prolog: manuel de reference et d'utilisation," Tech. report, Groupe d'Intelligence Artificielle, September 1975.

3. R. M. Keller, G. Lindstrom, and E. I. Organick, "Rediflow: A Multiprocessing Architecture Combining Reduction and Data-Flow," *Parallel Architecture Workshop*, Univ. of Colorado, Boulder, Colo., 1983, , Visuals published by Department of Energy, Office of Basic Energy Sciences

4. A.L. Davis and R.M. Keller, "Dataflow program graphs," *Computer*, Vol. 15, No. 2, February 1982, pp. 26-41.

5. R. A. Kowalski, "Predicate logic as a programming language," *Proceedings of IFIP 74*, Amsterdam: North Holland, 1974, pp. 556-574.

6. K. L. Clark and S. Gregory, "PARLOG: A Parallel Logic Programming Language (Draft)," Tech. report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, University of London, March 1983.

7. Y. Matsumoto, K. Nitta and K. Furukawa, "Prolog interpreter and its parallel execution", To appear in Lecture Notes in Computer Science.

8. Shinji Umeyama and Koichiro Tamura, "Parallel Execution of Logic Programs," Tech. report, Electrotechnical Lab., 1982.

9. H. Abelson and G. J. Sussman, "Structure and interpretation of computer programs," Draft, Department of EE and CS, Massachusetts Institute of Technology, July 31 1983, Sections 4.3 *Logic Programming* and 4.4 *Implementing the Query System*, pp. 263-298.

10. Keller R.M., "FEL Programmer's Guide," AMPS Technical Memorandum 7, Univ. of Utah, Dept. of Computer Science, April 1982.

11. W. H. Berge, *Recursive Programming Techniques* Addison-Wesley, 1975.

12. Kahn G., "The Semantics of a Simple Language for Parallel Programming," *Proc. IFIP 1974*, 1974, pp. 471-475.

13. Jiro Tanaka and Robert M. Keller, "S-code Extension in FEL," AMPS Technical Memorandum 10, University of Utah, July 1983.

14. Jiro Tanaka, *Optimized concurrent execution of an applicative language*, PhD dissertation, Univ. of Utah, 1983.

15. David H. D. Warren, "Implementing Prolog," Tech. report 39, D. A. I., May 1977, volume 1.

16. Ehud Y. Shapiro, "A subset of concurrent Prolog and its interpreter," Tech. report TR-003, ICOT, January 1983.

17. Robert M. Keller and M. Ronan Sleep, "Applicative Caching," *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, ACM, October 1981, pp. 131-140.

18. Gary Lindstrom, "Backtracking in a Generalized Control Setting," *TOPLAS*, Vol. 1, No. 1, July 1979, pp. 8-26.