Functional programming and the logical variable

G. Lindstrom

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.: (3) 954 90 20

Rapports de Recherche

N° 357

FUNCTIONAL PROGRAMMING AND THE LOGICAL VARIABLE

Gary LINDSTROM

Janvier 1985

FUNCTIONAL PROGRAMMING AND THE LOGICAL VARIABLE

Gary Lindstrom

Abstract :

Logic programming offers a variety of computational effects which go beyond those customarily found in functional programming languages. Among these effects is the notion of the "logical variable", i.e. a value determined by the intersection of constraints, rather than by direct binding. We argue that this concept is "separable" from logic programming, and can sensibly be incorporated into existing functional languages. Moreover, this extension appears to significantly widen the range of problems which can efficiently be addressed in function form, albeit at some loss of conceptual purity. In particular, a form of side-effects arises under this extension, since a function invocation can exert constraints on variables shared with other function invocations. Nevertheless, we demonstrate that determinacy can be retained, even under parallel execution. The graph reduction language FGL is used for this demonstration, by being extended to a language FGL+LV permitting formal parameter *expressions*, with variables occurring there inbound by unification. The determinacy argument is based on a novel dataflow-like rendering of unification. In addition the complete partial order employed in this proof is unusual in its explicit representation of demand, a necessity given the "benign" side-effects that arise. An implementation technique is suggested, suitable for reduction architectures.

Résumé :

La Programmation Logique permet une variété de mécanismes de calcul qui vont au-delà de ceux que l'on rencontre dans les langages de programmation purement applicatifs. Parmi ces mécanismes, on trouve la notion de "variable logique", c'est-à-dire d'une valeur qui est déterminée par l'intermédiaire de l'intersection de plusieurs contraintes plutôt que par une liaison explicite. Nous argumentons le fait que ce concept peut être séparé de la Programmation Logique et peut être introduit utilement dans les langages applicatifs présents. Qui plus est, cette extension semble étendre de manière significative la collection de problèmes qui peuvent être traités de manière efficace sous forme fonctionnelle, au prix d'une certaine inélégance. En particulier, une certaine forme d'effet de bord fait son apparition dans cette extension, puisqu'un appel de fonction peut créer des contraintes sur des variables partagées par d'autres appels. Néanmoins, nous prouvons que le résultat reste parfaitement déterminé, même dans un régime d'exécution parallèle. Le langage FGL est utilisé pour cela. On l'étend en un langage FGL+LV qui autorise de passer comme paramètres formels des expressions dans lesquelles les variables peuvent se trouver liées par unification. L'argument de la preuve est fondé sur une nouvelle façon de présenter l'unification comme une opération guidée par les données. De plus, l'ordre partiel complet utilisé dans cette démonstration est inhabituel. Une technique d'implémentation est discutée qui semble appropriée pour les machines à réduction.

Professeur invité, pour une durée de trois mois, dans le projet de M. KAHN.

To be presented at 12th ACM Symposium on Principles of Programming Languages
New, Orleans, Jan. 14-16, 1985:

FUNCTIONAL PROGRAMMING AND THE LOGICAL VARIABLE[1]

Gary Lindstrom

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

Abstract

Logic programming offers a variety of computational effects which go beyond those customarily found in functional programming languages. Among these effects is the notion of the "logical variable," i.e. a value determined by the intersection of constraints, rather than by direct binding. We argue that this concept is "separable" from logic programming, and can sensibly be incorporated into existing functional languages. Moreover, this extension appears to significantly widen the range of problems which can efficiently be addressed in function form, albeit at some loss of conceptual purity. In particular, a form of side-effects arises under this extension, since a function invocation can exert constraints on variables shared with other function invocations. Nevertheless, we demonstrate that determinacy can be retained, even under parallel execution. The graph reduction language FGL is used for this demonstration, by being extended to a language FGL+LV permitting formal parameter *expressions*, with variables occurring therein bound by unification. The determinacy argument is based on a novel dataflow-like rendering of unification. In addition the complete partial order employed in this proof is unusual in its explicit representation of demand, a necessity given the "benign" side-effects that arise. An implementation technique is suggested, suitable for reduction architectures.

---

[1]This material is based upon work supported by the National Science Foundation under Grant INT-83-00432, and by a Shared University Research Grant from the IBM Corporation. Document preparation was performed using equipment procured under NSF CER Grant MCS-81-21750.

Table of Contents

List of Figures

1. The Phenomenon of Logic Programming

From the perspective of functional programming, logic programming presents several new notions, including:

1. an incremental, rule-oriented program structure;

2. search-based or "relational" computation, and

3. variable binding by the intersection of constraints.

In some *apologias* for logic programming one is given the impression that these features form a monolithic semantic whole. However, these notions can indeed profitably be "unbundled" for individual consideration as possible extensions to existing languages. In this paper we address the specific issue of incorporating the third notion above, that of the *logical variable*, into a modern functional programming language, FGL ("Function Graph Language") [10]. By "modern" here we mean possessing infinite data structures, higher-order functions, efficient data constructors, etc.

1.1. What is a Logical Variable?

The use of structured formal parameter lists in functional programming languages has become quite common (e.g. in KRC, HOPE, ML, etc.). However, such notations heretofore have been simply handy abbreviations for "unpacking" structured actual parameter values. For example, in FEL [13] one can write

```
FUNCTION f([x, [y, z]]) = ...
```

as an abbreviation for

```
FUNCTION f(p) =
{       x = car(p),
        y = car(cdr(p)),
        z = cdr(cdr(p)),

RESULT  ... }
```

Since such structured formal parameter expressions must be translated at compile time into an equivalent set of selector application equations, they must obey the following restrictions:

1. no function applications or constants are permitted, and

2. all variables must be distinct.

We propose here to eliminate both these restrictions. Eliminating the first restriction means that function applications can be predicated on actual parameters meeting certain structural requirements. Eliminating the second restriction means that such requirements

can extend to equality among specified actual parameter components. Moreover, we propose to permit *actual* parameter variables to share in such equality relationships, and in some cases receive their values from the environment of the function being applied, rather than from the applying environment.

In sum, all these effects add up to the *logical variable* idea, as developed so elegantly in Prolog.

1.2. Adding Logical Variables to Functional Languages

What does it mean to add logical variables to a functional language? There are several implications; the first is a restatement of the liberalization just discussed.

* *Syntactically*, it means functions have formal parameter *expressions* rather than lists of distinct variables. Hence function invocations can appear within formal as well as actual parameters.

* *Semantically*, it means variables are given values by constraint intersection, rather than by being bound to particular value expressions.

* *Operationally*, parameter passage by unification becomes necessary, and information flow between formal and actual parameter computations is bidirectional [20].

* *Functionally*, the determinacy of such a language on a parallel architecture is open to serious question, since function invocations can have side effects through sharpening constraints on non-local variables.

2. Is the Extension Useful?

First one must ask if such an extension is worthwhile, given that the logical variable is being nominated for incorporation into a functional language without the customary supporting features from logic programming (search, clausal programs, etc.). We answer affirmatively, first by exhibiting several examples where logical variables appear to extend the range of efficient applications of functional programming, and second by indicating how this extension widens (or clouds!) the functional programmer's conceptual world.

2.1. Examples

We offer three examples to illustrate the power of this extension, using the notation of the Function Equation Language FEL [13], which is a textual representation for FGL. One example is given in full detail, and the other two are sketched intuitively. The essential features of FEL and their notations are as follows:

* *Block notation:* An expression in FEL may be a block of the form:

  {*equations* RESULT *expression*}

where the *equations* define locally bound names, generally used within the RESULT *expression*. The "main program" of a FEL computation is a block.

* *Tuples and lists:* Lisp-like lists may be constructed, using binary tuples as building blocks. The notations [a, b], a ^ b, and cons[(a, b)] are all equivalent in FEL (hence cons is the identity function, restricted to binary tuples!). The infix operator ^ is right associative. car and cdr are available with the customary meanings.

* *Quoting convention:* 'a evaluates to a itself. The atom 'nil designates the empty list (and is always the ultimate tail of a finite list).

* *Function definitions:* A FEL function may be defined using the syntax

```
FUNCTION f(p) = b
```

where

1. f is the function name,
2. p is a single formal parameter (multiple parameters are achieved using a tuple notation, with "unpacking" as described above), and
3. b is the function's body expression.

2.1.1. Difference Lists

This is the most famous demonstration of logical variables in Prolog. This involves a list representation convention whereby a list α is represented by a pair $[\alpha', \nu]$, where α' is the list α with its terminating 'nil replaced by an occurrence of the logical variable ν.

This trick has a surprising payoff: lists may now be concatenated in fixed time. For example, the computation in figure 2-1 would yield a result of [1, [2, [3, 'nil]]].

```
{       FUNCTION nconc([[x, y], [y, z]]) =
                [x, z],
        d1 = [[1, [2, u]], u],
        d2 = [[3, 'nil], 'nil],

RESULT  car(nconc([d1, d2]))}
```

Figure 2-1: Difference list concatenation.

The result is an applicative analog of Lisp's nconc. However, the same problems arise here (and in Prolog) that arise in Lisp: nconc($[\alpha, \beta]$) *makes* β *become* (or *asserts that* β *is*, if you wish, in Prolog) the tail of α. In other words, the meaning of α has been *refined*

to include β as a new list tail.

Thus, by normal functional programming standards, a side-effect has occurred. Or has it? In Prolog if we subsequently seek to concatenate $\chi = [\chi', \mu]$ to $\alpha = [\alpha', \nu]$, then such a "second" concatenation can proceed *as long as it does not contradict the previous concatenation!*

2.1.2. Objects and Messages

In "object oriented" functional programming, objects are represented by stream transducers, accepting input messages and emitting response messages. When used on a realistic scale, however, there becomes a problem routing each response message back to the source of the matching input message. Typically, routing networks are used to reverse the input message's logical arrival path in returning the result message. This is computational overkill, particularly irksome on architectures where uniform addressing could be used for direct delivery to a "mailbox."

Consider, in contrast, a technique where input messages are packaged in pairs [*msg_in, msg_out*], with *msg_out* being a variable to be constrained by the object to convey the result. The sender can then have direct access to the result, and even be "awakened" by its production. (Is this not, in large part, is what Shapiro means when he says Concurrent Prolog is an object-oriented language "par excellence"? [22])

2.1.3. Owner-Coupled Sets

A common data structuring technique for entity-relationship databases is to use the Codasyl notion of "owner coupled sets." Here records representing entities are linked into multiple rings indicating their sorted orders by various fields. This highly cyclic data structure is advantageous not only for space economy, but to provide unique occurrences of each entity to simplify update transaction processing.

Constructing such a data structure is not difficult in an imperative language, but very challenging in a functional language due to the need to relocate pointers in the data structure as each successive sort (and new "version" of the structure) occurs. Indeed, one may argue that the problem cannot even adequately be specified in functional terms, since it fundamentally involves uniqueness of object representation.

With the link fields represented as logical variables, however, the ordering is readily obtained since each sort order can be obtained independently, and the link fields subsequently bound by independent traversals, each constraining a separate subset of the links in a desired direction.[2]

---

[2]This answers a conjecture raised by the author at the Aspenas Functional Programming and Architecture Workshop [3].

2.2. Conceptual Extension

These examples demonstrate that in this enlargement of functional programming:

1. parameters can carry information both *into* and *out of* function invocations (like *ref* parameters);

2. constraint application serves to make producer-consumer relationships *isotropic* (behaving the same in all directions[3]; again, a keen observation exploited by Shapiro in Concurrent Prolog), and

3. multiple function invocations can cooperatively "refine" shared data structures (suggesting assignment-like activity).

In other words, the functional programmer is permitted a widened conceptual framework in which *object uniqueness* and *shared access* of objects have semantic substance. These are heretical ideas in some functional programming theologies, but ones secretly held by even the truest believers when they seek to efficiently exploit real functional programming implementations.

However, all knowledge has its price, and one may question whether this widened framework comes at too dear a cost, e.g. loss of functionality (determinacy), corruption of semantic purity, or hopeless implementational complexity.

We begin our consideration of this question by developing a semantics for FGL+LV, an extension of FGL including logical variables. This will be done in three stages: (i) informally; (ii) by a graph reduction model, and (iii) by a formal denotational semantics.

3. The Informal Semantics of FGL+LV

Suppose f(a) is an invocation of function f on actual parameter a. Let the body of f be b, and its formal parameter (expression, recall) be p. Then the informal semantics of f(a) are as follows:

1. a and p are evaluated sufficiently to determine whether their denoted values unify (in the usual Herbrand "syntactic" sense).

2. If so, b is evaluated observing the constraints on variables in p that resulted from the unification of a and p.

3. If not, a function invocation error has occurred, and the result is undefined.

Disclaimer: Notice that, in contrast to Prolog, we are viewing unification as a generalized parameter passage mechanism, rather than as a precondition for function invocation. A failing unification attempt is considered a function

---

[3]This lovely term is due to Alan J. Perlis.

invocation error.

Note that function invocation and unification are closely intertwined in FGL+LV; unification can occur during function invocation (obviously), and vice-versa, since function invocations can appear in formal and actual parameter expressions. Moreover, the binding of a logical variable by parameter unification in one function invocation can enable another function invocation to proceed, if the latter has been awaiting a binding for that variable. Our primary goal here is to develop a language which smoothly integrates these two activities, while facilitating parallel execution but preserving determinacy in correct programs.

4. A Reduction Model

We now present a reduction model for the semantics of FGL+LV. This will involve a graphical representation for FGL+LV programs, along with a set of rules for transforming a program graph to a final result. For simplicity, we only consider FGL+LV programs having explicitly invoked functions (i.e. no **apply** operators on functional data values). Our method generalizes to the latter case [11], but its inclusion would unnecessarily complicate exposition.

4.1. Graphical Representation

FGL+LV programs use the graphical representation of FGL as a basis. Under this representation, FGL function bodies consist of digraphs with operator occurrences as nodes and input-output relationships among them depicted as arcs. Exhaustive common subexpression elimination is performed within each block to exploit referential transparency. Formal parameters appear graphically as sourceless arcs entering function bodies. Function invocations are represented by nodes bearing the function name, with the actual parameter subgraph as input.

FGL+LV programs depart from this basis in two respects, as indicated in figure 4-1:

* "free" variables are represented by 0-ary nodes labelled with ubv ("unbound variable"), and

* formal parameter expressions manifest as secondary "roots" for function graphs.

Figure 4-2 illustrates this representation for the nconc example of figure 2-1.

4.2. Reduction Rules

Our first attempt at defining a satisfactory reduction semantics for FGL+LV begins by including all reduction rules for conventional FGL [12], except the rule for function invocation. To this base we add the special rules depicted in figures 4-3 and 4-4. Notice:

* The function invocation rule in figure 4-3 shows strictness with respect to

FUNCTION f(p) = b has graphical representation:

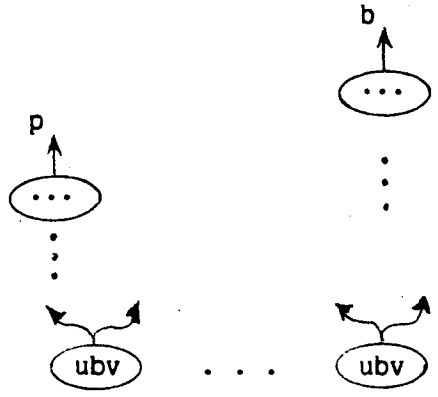


Figure 4-1: FGL+LV function graphs.

(local) unification success. That is, the sequential semantics for cond in FGL require the unify operation to yield a true result before permitting the function body to become the result of the invocation.

* The rules in figure 4-4 cause unify operators to traverse a and p in parallel until equality is either refuted or established, possibly with the aid of one or more ubv node mergings.

4.3. A Difficulty

This simple model, alas, is inadequate in a fundamental respect. Unlike in FGL, function invocations in FGL+LV have a form of side-effect: they can refine (through unification) the values of variables shared with other function invocations. Thus, for example, whether or not a function invocation is performed in a non-selected arm of a conditional can affect a computation's overall result, i.e. whether or not a function invocation error arises.

Such a side effect can occur even in the following simple program:

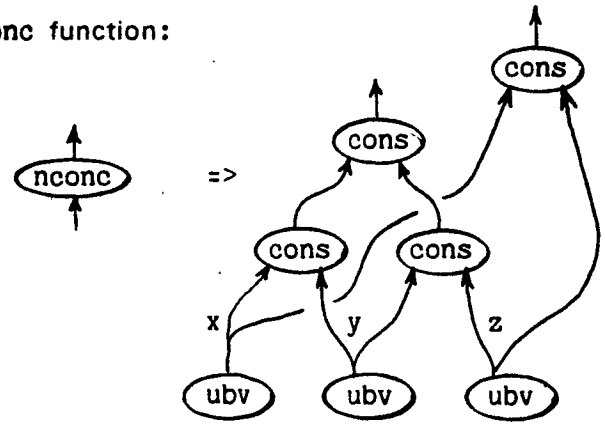
```
{       FUNCTION f(1) = 1,

        FUNCTION g(2) = 2,

RESULT  if 3>4 then f(u) else g(u)}
```

Clearly, the result of this program should be 2, since 3 is not greater than 4, so the result of the conditional must be g(u), which when invoked should cause u to unify successfully with 2, and returns 2.

nconc function:



main block:

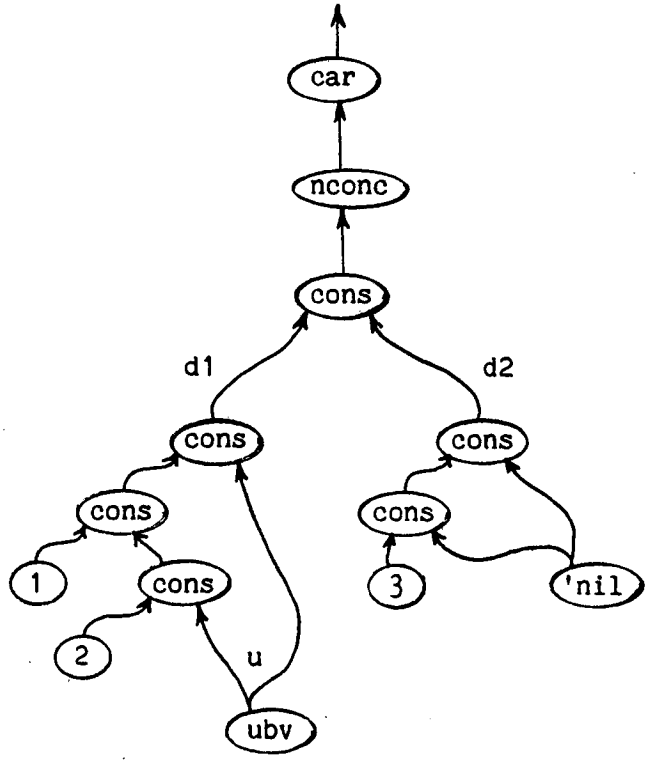
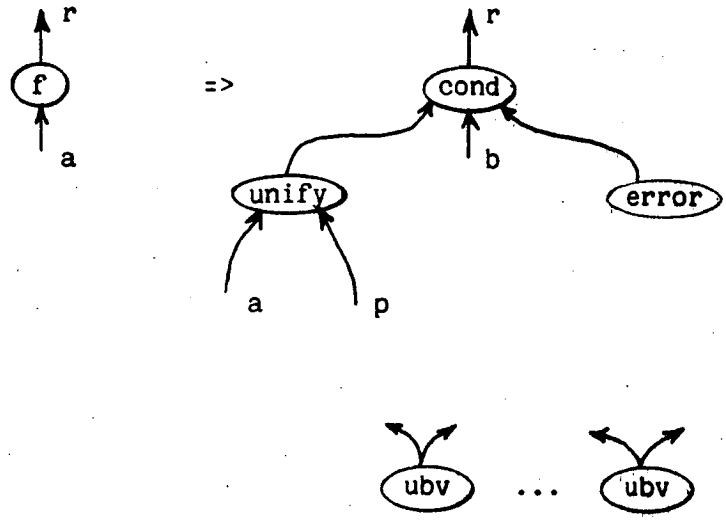


Figure 4-2: Function graph for figure 2-1.

But what if f(u) is also expanded and has its unification complete prior to the invocation of g(u)? This easily can happen by our reduction model, yet this sequence of events causes g(u) to return error, since the logical variable u is already bound to 1, which cannot unify with 2.



(function graph for f is copied)

Figure 4-3: FGL+LV reduction rules:
function invocation.

This small example indicates that demand is a fundamental semantic notion for FGL+LV, not from a divergence avoidance standpoint, but from a theoretical adequacy standpoint. Fortunately, the extension of graph reduction models to observe demand propagation patterns is straightforward (see e.g. [12], where such an embellishment is used as a tool for the validation of a distributed implementation). Moreover, essentially all graph reduction execution models are demand-driven in practice, so the cost pragmatically is *nil*.

Rather than immediately revising the rules of figures 4-3 and 4-4 to incorporate demand propagation effects (which will turn out to be a bit more subtle than for FGL itself), we shall wait until an adequate denotational semantics for FGL+LV is presented in sections 5 and 6.

However, we point out that even the inclusion of demand sensitivity does not eliminate another potentially worrisome problem with our graph reduction model: *local indeterminacy*. To see this, consider a second simple FGL+LV program:

```
{       FUNCTION f(1) = 1,

        FUNCTION g(2) = 2,

RESULT  f(u)+g(u)}
```

This is a program yielding an error result. However, consider the intermediate results

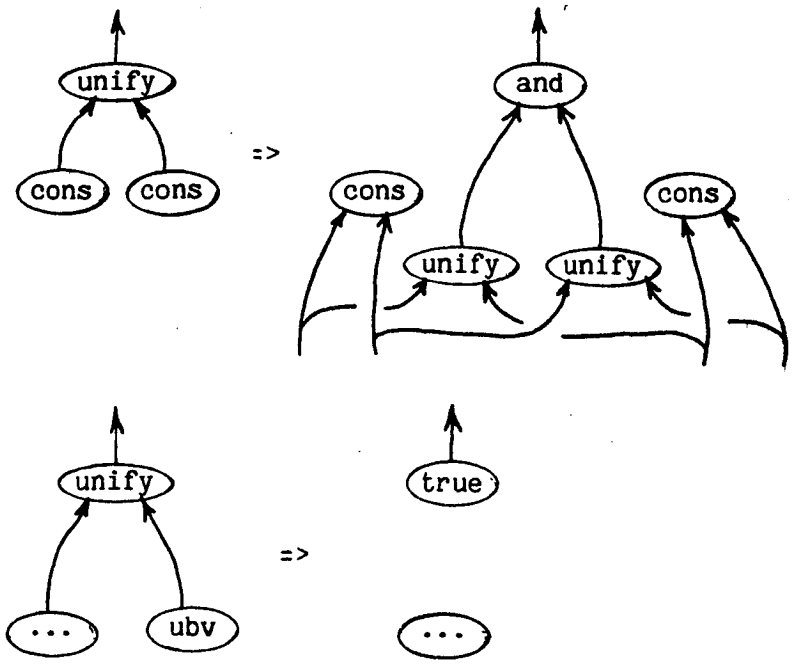


Figure 4-4: FGL+LV reduction rules: `unify`.

from the `f(u)` and `g(u)` invocations. Under our reduction model, one of them will have a successful parameter unification, and one will not. However, it is unpredictable whether it will be the `f` or the `g` invocation which will fail.

This raises the following interesting conjecture:

> FGL+LV programs under our graph reduction semantics (extended for demand sensitivity) are *globally determinate* (i.e. return a well-defined overall result) despite possibly being *locally indeterminate* (i.e. having unpredictable results

for function invocations internal to the program, as just described).

As we shall see, this conjecture is indeed true, in the sense that:

1. if no function invocation failure occurs, every function invocation is deterministic, and

2. if a function invocation failure does occur, that error is sure to become the computation's overall result.

5. A Denotational Model

We now address the problem of developing a denotational semantics for FGL+LV. The problem is a delicate one, since customary denotational semantics techniques do not lend themselves to FGL+LV given the local indeterminacy just noted.

Instead, we indirectly develop a denotational semantics for FGL+LV in the following stages:

1. FGL+LV is given a mapping from its graphs to graphs in FGL$^+$, a language which differs from conventional FGL in the following respects:

   a. the inclusion of four new operators (ubv, sup, merge and ifnoterror);

   b. the use of oppositely directed arc pairs throughout the function graph, and

   c. the generalization of all operators to operate on this "dual rail" graph structure.

   We emphasize that FGL$^+$ is being introduced as an expository device that aids us in defining the semantics of FGL+LV. FGL$^+$ is *not* intended as a language ever to implemented in a practical sense.

2. A complete partial order (C. P. O.) is defined, upon which all FGL$^+$ operators are monotonic and continuous.

3. The reduction semantics for FGL+LV, in the case of *non*-error results, is shown to be consistent with the denotational semantics of the corresponding computation in FGL$^+$.

4. Finally, a special argument is made that an error denotation in FGL$^+$ must result in an overall error value in the reduction model for FGL+LV.

As with our reduction model, for simplicity we first ignore the issue of demand evaluation control. We then refine the model to explicitly represent demand sensitivity, thereby preventing impertinent constraint applications.

5.1. Mapping to FGL$^+$

Consider an arbitrary FGL+LV function graph *G*. We now define an associated FGL$^+$ graph G^+.

5.1.1. Arc Mappings

We map each arc in *G* into a *pair* of oppositely directed arcs in G^+, as suggested in figure 5–1. Intuitively, within each arc pair $r = [r_o, r_i]$ in G^+:

* The *upward* arc r_o conveys the same meaning as the (upward) arc in *G*, namely the denotation of a result value.

* However, the *downward* arc r_i serves the new purpose of conveying *constraints* on the result value arising from its use in other contexts (e.g. unifications).

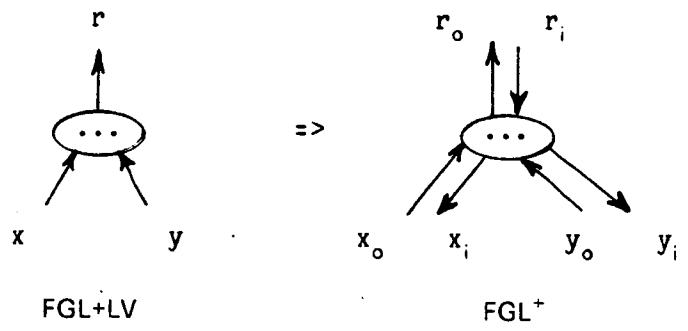


Figure 5-1: Arc mapping scheme.

5.1.2. Node Mappings

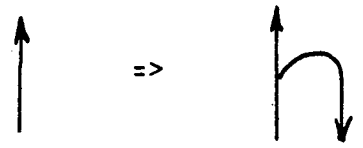
Nodes in FGL$^+$ in general have several output arcs (e.g. r_o, x_i, and y_i in figure 5-1). Such nodes are actually macro representations for clusters of conventional single output nodes. We shall shortly provide interpretations for such shorthand representations by giving rules for determining the output in each case from the associated set of inputs.

The structural conversion process from *G* to its associated graph G^+ proceeds as suggested in figures 5–2 and 5–3. These figures indicate the syntactic contexts in which the three special operators, `merge`, `ifnoterror` and `sup`, are introduced into G^+. Intuitively,

* `sup` is used whenever two values must be "intersected" to form a combined constraint, and

* `ifnoterror` is used to make each function invocation contingent on the success of its parameter unification.

output of overall graph:



forks:

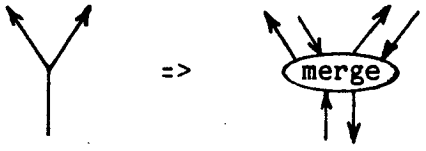
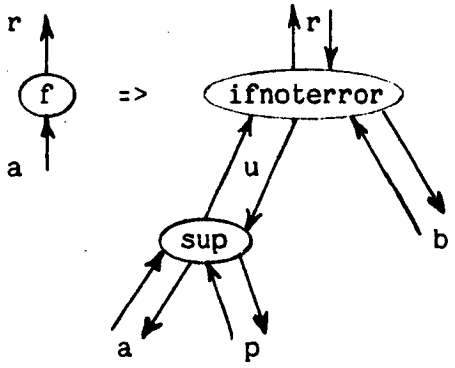


Figure 5-2: FGL+LV to FGL$^+$ mapping rules.



(function graph for f is copied)

Figure 5-3: FGL+LV to FGL$^+$ mapping for function invocation.

5.1.3. Overall Result

The upshot of this mapping is that any FGL+LV graph *G* is associated with an FGL$^+$ graph G^+ in which all constraints on a given logical variable are linked together in a cyclic dependency graph. Figure 5-4 illustrates this effect where the actual parameter [x, y] is unified with formal parameter [1, 2].

5.2. The Complete Partial Order D

We define the C. P. O. *D* in figure 5-5 as our first approximation for use with FGL$^+$.

Notice in particular the use of error as a *top* element in *D*. This means any value in a

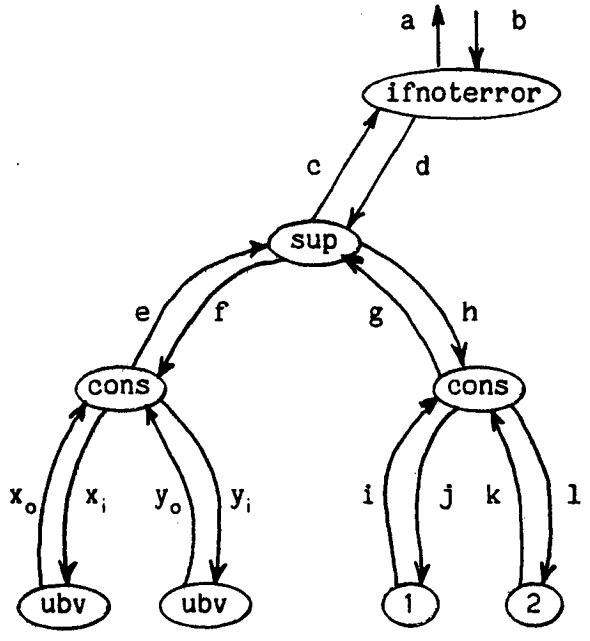


Figure 5-4: Cyclic dependency example: unification of [x, y] and [1, 2].

domain:

$D_0 = \{\bot\} \cup A \cup D_0 \times D_0$, where A is the atom set (including functions)

$D = D_0 \cup \{\text{error}\}$

ordering:

$\bot \leq x, \forall x$

$[a, b] \leq [a', b']$ iff $a \leq a'$ *and* $b \leq b'$

$x \leq \text{error}, \forall x.$

Figure 5-5: The C. P. O. D for FGL$^+$.

FGL$^+$ computation is potentially supercedable by error. A troubling implication of this treatment of error is that output values are in principle always "recallable". Nevertheless, we shall argue in section 8 that FGL+LV exhibits a sensible notion of "definiteness" whereby approximations of overall values, once delivered, cannot be superceded unless

new demands are injected into the computation.

Given *D*, we can define the operators of FGL$^+$ in a monotonic and continuous manner, as shown in figures 5-6 and 5-7. To simplify our equations, we follow two conventions:

1. Since we desire all operators to propagate errors unconditionally, we omit such values in the equations given there. Thus, for example, u_o=error causes r_o=error in ifnoterror; this is the crucial mechanism for reporting unification failures.

2. Operators in the equations are themselves interpreted to be monotonic on *D*. Thus, for example, the equation

$$r_o = \sup(op(x_o, y_o), r_i)$$

in the semantics for a strict binary operator is in fact an abbreviation for

$$r_o = \text{if } x_o=\perp \text{ or } y_o=\perp \text{ then } r_i \text{ else } \sup(op(x_o, y_o), r_i)$$

The operator sup(a, b) is of course the least upper bound of a and b on *D*. Thus if either of a or b equal error, or a and b are incomparable in *D* (i.e. represent inconsistent constraints), sup(a, b) yields error. To illustrate the application of these operator semantics, we show in figure 5-8 one of many possible sequences of value refinements on the arcs of figure 5-4.

We believe this dataflow-like rendering of unification to be novel, and of some potential interest itself. Conceivably it could be used as a basis for unification on a dataflow architecture.

6. Demand Sensitivity

As remarked in section 4.3, an adequate semantics for FGL+LV must incorporate a notion of function invocation control.

6.1. Representing Demand in FGL$^+$

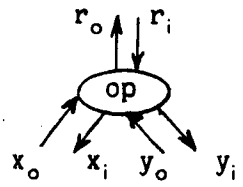
To accomplish this, we must:

1. determine what "demand" should mean in this new computational model;

2. extend *D* to include a representation of this notion of demand,

3. find a means of communicating this demand in appropriate patterns among the operators of an FGL$^+$ graph, and

4. refine the definitions of FGL$^+$ operators to encompass this added sensitivity.

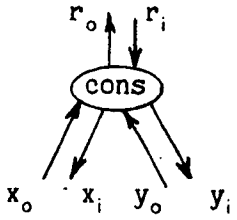


$r_o = sup(op(x_o, y_o), r_i)$

$x_i = x_o$

$y_i = y_o$

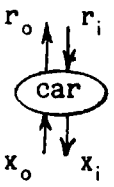
(for op a binary strict operator; generalizes to atoms)



$r_o = [x_o, y_o]$

$x_i = sup(x_o, car(r_i))$

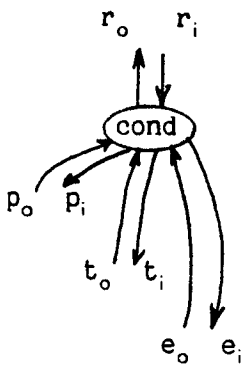
$y_i = sup(y_o, cdr(r_i))$



$r_o = sup(r_i, car(x_o))$

$x_i = [sup(r_i, car(x_o)),$
$cdr(x_o)]$

(and similarly for cdr)



r_o = if p_o=true then t_o else
if p_o=false then e_o else
error

$p_i = p_o$

t_i = if p_o=true then r_i else
if p_o=false then t_o
else error

e_i = (similarly)

Figure 5-6: FGL$^+$ operator semantics: strict operators, cons, car, and cond.

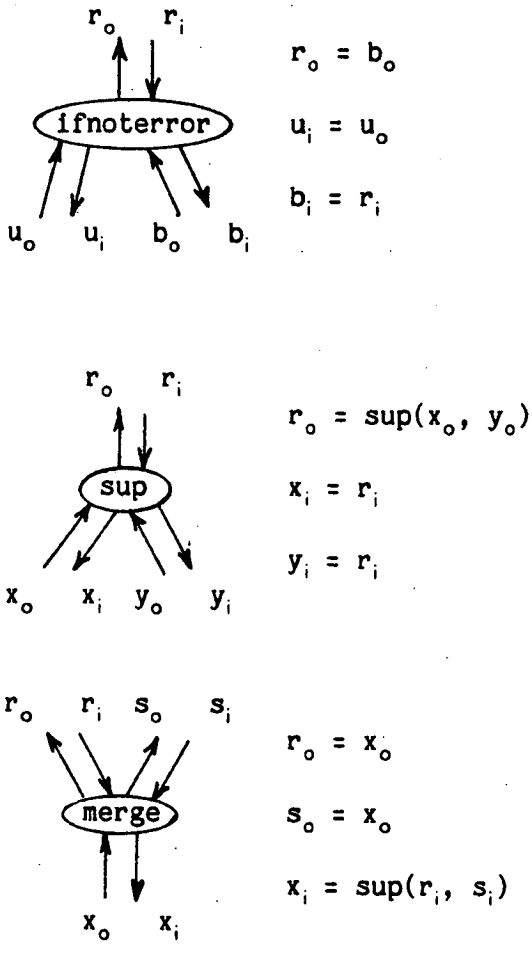


Figure 5-7: FGL$^+$ operator semantics: ifnoterror, sup and merge.

6.1.1. Demand Varieties in FGL$^+$

We now consider what form of demand evaluation is necessary to obtain coherent computations in FGL$^+$. Since a useful and implementable definition of FGL+LV is our ultimate goal, we begin by examining desired reduction control patterns in FGL+LV.

In this regard, one quickly sees that demands on ubv nodes must be handled differently depending on whether they originate in unification or evaluation contexts. That is:

* a demand on a ubv should be *satisfied* when issued by a unify node (so that the appropriate node merging can be done), but

* *not satisfied* when issued by an operator which needs a *bona fide* value from the ubv in order to be applied itself.

all arcs initially $\perp$

```
    e   =  [⊥, ⊥];
    c   =  [⊥, ⊥];
*   i   =  1;
    g   =  [1, ⊥];
*   k   =  2;
    d   =  [⊥, ⊥];
    c   =  [1, ⊥];
*   g   =  [1, 2];
    d   =  [1, ⊥];
*   f   =  [1, 2]
*   x_i =  1;
*   x_o =  1;
    e   =  [1, ⊥];
*   c   =  [1, 2];
*   d   =  [1, 2];
*   y_i =  2;
*   y_o =  2;
*   e   =  [1, 2];
*   h   =  [1, 2];
*   l   =  2;
*   j   =  1
```

* denotes final value on each arc.

Figure 5-8: Value refinements on arcs of fig. 5-4.

However, contextual analysis of demand origins is a semantically and implementationally unwieldy notion. Fortunately, the necessary discrimination information can be conveyed by adopting two *levels* of demand:

* the conventional *non-assertive* one, indicating "demand in evaluation context", which is not satisfied at a ubv until that variable becomes bound, and

* a more lenient or *assertive* demand, indicating "demand in constraining context", which is immediately satisfied by a ubv.

6.1.2. Extending D

This two-level demand requirement can easily be met in our C.P.O. D by extending it to D', in which

1. non-assertive demand is represented by d_v;

2. assertive demand is represented by d_c, and

3. $\perp \leq d_v \leq d_c$, $\forall x$ not in $\{\perp, d_v, d_c\}$.

Note that D' contains nested tuples with demand indicators in component positions, e.g.

$$[[1, [d_c, 2]], d_v].$$

As we shall see, these are very handy for signifying patterns of demand on structured data values.

6.1.3. Communicating Demand

We next face the question of how to communicate d_v and d_c values in appropriate patterns among the operators of an FGL$^+$ program graph. Again, we are fortunate in having a simple solution at hand:

> We use the downward arcs in FGL$^+$ program graph, originally developed for the purpose of constraint distribution, for demand propagation as well!

6.2. Demand-Sensitive Operators in FGL$^+$

All that now remains is to redefine the operators of FGL$^+$ to propagate d_v and d_c values in a manner indicating the desired application control. Certain aspects of this are straightforward, e.g. expansion of function invocation nodes only on demand, and the use of demand to obtain sequential cond operator semantics. However, more subtle issues must be addressed as well, including:

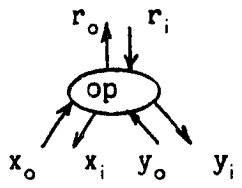
1. how to control demand introduction at the main program's output;

2. how to handle the situation where a non-assertive demand becomes "overtaken" by an assertive demand (this is the typical situation of a suspended function invocation resuming after a ubv has been given a binding by a unification action in another function invocation);

3. avoidance of "accidental" constraint mingling, whereby constraints exerted on a logical variable by one unification action might as a side-effect trigger over-eager unification actions in an undemanded invocation sharing access to that variable, and

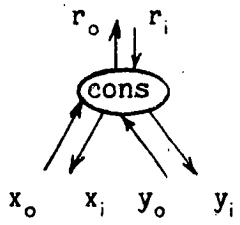
4. how to "favor" demand propagation on the *formal* parameter side of a unification, so that in the special case where the formal parameter is a tuple of distinct ubv's (corresponding to the situation in conventional FGL), compatibility (i.e. actual parameter laziness) results.

We assert that the C. P. O. D' and the "dual rail" structure of FGL$^+$ graphs are indeed rich enough to achieve the full measure of this control dexterity. To show this, we redefine our selected FGL$^+$ operators with full demand sensitivity in figures 6-1 and 6-2, and informally explain how the four special control effects listed above are obtained.



r_o = if d_v<op(x_o, y_o) then
sup(op(x_o, y_o), r_i) else $\perp$
x_i = if r_i=$\perp$ then $\perp$ else
sup(x_o, d_v)
y_i = (similarly)

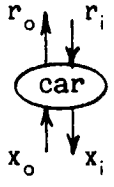
(for op a binary strict operator;
generalizes to atoms)



r_o = r_i=$\perp$ then $\perp$ else
[x_o, y_o]

x_i = sup(x_o, car(r_i))

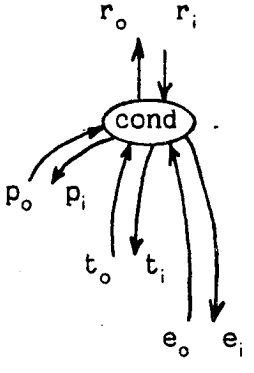
y_i = (similarly)



r_o = if d_v<car(x_o) then
sup(r_i, car(x_o)) else $\perp$

x_i = [sup(r_i, car(x_o)),
cdr(x_o)]

(and similarly for cdr)

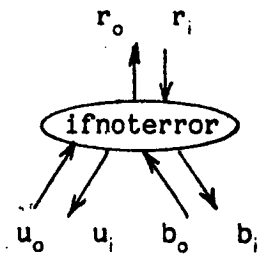


r_o = if r_i=$\perp$ then $\perp$ else
p_o=true then t_o else
if p_o=false then e_o else
error
p_i = if r_i=$\perp$ then $\perp$ else
sup(p_o, d_v)
t_i = if r_i=$\perp$ then $\perp$ else
if p_o=true then r_i else
if p_o=false then t_o
else error
e_i = (similarly)

Figure 6-1: Demand sensitive FGL$^+$ semantics:
strict operators, cons, car, and cond.

To aid in understanding these figures, we offer the following observations:

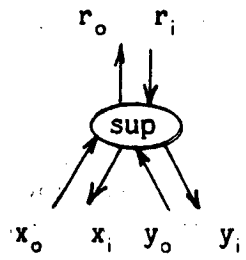
1. All our equations preserve two invariants:



r_o = if r_i=⊥ then ⊥ else b_o

u_i = if r_i=⊥ then ⊥
else sup(u_i, d_c)

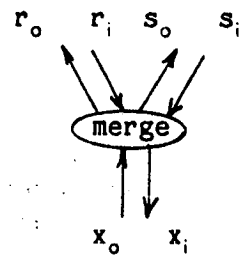
b_i = r_i



r_o = if r_i=⊥ then ⊥
else sup(x_o, y_o)

x_i = y_o

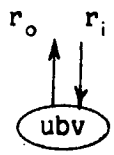
y_i = if r_i=⊥ then ⊥ else r_i



r_o = if r_i=⊥ then ⊥
else x_o

s_o = if s_i=⊥ then ⊥
else x_o

x_i = sup(r_i, s_i)



r_o = if d_c<r_i then r_i else ⊥

Figure 6-2: Demand sensitive FGL$^+$ semantics: ifnoterror, sup, merge, and ubv.

a. *Demand Propagation Rule:* no demands are sent downward until received from above at each node.

b. *Demand Anti-Reflection Rule:* Demand indicators never flow upward from a node.

With a few (important) exceptions, demand varieties are unchanged when propagated downward. These exceptions occur in strict operators, cond, and ifnoterror:

a. In strict operators, demands are converted from assertive to non-assertive; one cannot make, for example, plus(3, 4) equal anything simply by "assertion."

b. Similarly, in cond a non-assertive demand is always sent to the predicate arm, while the received demand is propagated unchanged to the then or else arms. Again, this is consistent the fact that one cannot "assert" whether the then or the else arm of a conditional is to prevail.

c. In contrast, ifnoterror always sends an assertive demand to its unification arm, and whatever demand it received to its function body arm, for reasons which should be clear by now.

2. Notice that cons is lenient, since demands are propagated to the appropriate components only after they can be obtained by "unpacking." Thus, for example, d_c on r_i triggers a response on r_o of $[\bot, \bot]$, but when, say, the value on r_i is refined to $[d_c, \bot]$, then d_c is propagated along x_i. All in all, full compatibility with FGL's treatment of tuple lenience is obtained.

We now pass to the four subtler demand-sensitive control effects listed above.

1. *Top-level demand:* To commence an FGL$^+$ computation we inject on the downward member of the output arc pair of the main program graph *any element of D not containing occurrences of* d_c. Thus (in contrast to Prolog), output value approximations delivered (e.g. "printed") by the main program cannot contain unbound logical variables. More will be said on this issue in section 11.

2. *Demand overtaking:* The positioning of $d_v \leq d_c$ in D' ensures that any non-assertional demand can be upgraded to an assertional one.

3. *Constraint mingling avoidance:* This is achieved by having ifnoterror not "bounce back" constraints on u_i unless a demand (or better) has been received on r_i. Hence values evolving for a logical variable in one function invocation, even if communicated eagerly to another, will not be acted upon until the latter context becomes tangibly active by issuing a demand of its own. (In fact, such eager communication cannot happen either, given the Demand Propagation Rule observed throughout, notably in merge.)

4. *Biased unification:* Finally, the asymmetry in the specification for sup ensures that demands (and constraints) will not be propagated along x_i until delivered from y_o. Since demand indicators are never reflected "upward" (note in particular how cons and ubv ensure this), any value received on y_o must convey its *bona fide* structure as determined thus far, with all demanded but unevaluated components replaced with $\bot$. Thus components associated with unbound logical variables in the current y_o value approximation are interpreted as undemanded or "don't care" components when passed along x_i.

Note this biased semantics for sup does *not* imply a sequential unification algorithm. It simply requires that at each *structural position* in the comparison between formal and actual parameters, a check is first made to see if the formal parameter contains an unbound variable in that position.

7. Correctness of Reduction Model

We now apply the semantics of FGL$^+$ to obtain semantics for FGL+LV. Let us assume the reduction rules for FGL+LV have been made demand sensitive. As observed earlier, this is simple in most cases; the only interesting cases are `unify`, `ifnoterror` and `ubv`, where the semantics of FGL$^+$ must be carefully consulted.

7.1. FGL+LV Semantics

Consider an FGL+LV program graph G, and its corresponding graph G^+ in FGL$^+$. We assume for convenience that G^+ is fully expanded, i.e. all function invocations have been performed. Then the following steps outline how the semantics of G^+ can be used to establish a semantics for G:

1. The denotation of the main program output arc of G is that of the (upward) main program arc in G^+.

2. Suppose we have the denotation of the output arc of a node n in G, where n is not a `unify` node, and the corresponding node n^+ in G^+ is not an `ifnoterror` node. Then the denotations of the input arcs of n are those of the corresponding (upward) input arcs of n^+ in G^+.

3. Let n be a cond node (with inputs arcs p, t, and e) corresponding to an `ifnoterror` node n^+ in G^+ with input arcs u_o and b_o. Then the denotation of p is that of u_o, of t is that of b_o, and of t is `error`.

4. Now let n be a `unify` node (with inputs arcs a and p). and output arc denotation d. Then the denotations of a and p are also d.

7.2. FGL+LV Overall Determinacy

We now argue that the (demand-sensitive) reduction rules for FGL+LV do indeed yield main program results consistent with the denoted value of the corresponding main program in FGL$^+$.

Theorem 1: Let G be an FGL+LV program graph and G^+ its corresponding FGL$^+$ graph. If the output of G^+ denotes the value v^+, not equal to `error`, then any value v produced by a (demand-sensitive) reduction execution of G is equal to v^+.

Proof: (Outline)

1. Under reduction, demands appear only on arcs of G having non-$\perp$

denotations.

2. Every reduction step transforms G in a manner consistent with the denoted values.

3. Every reduction step necessary to manifest the denoted result is performed.

Theorem 2: Let G be an FGL+LV program graph and G^+ its corresponding FGL$^+$ graph. If the output of G^+ denotes the value error, then any value v produced by a (demand-sensitive) reduction execution of G is equal to error.

Proof: (Outline)

1. If the output of G^+ denotes error, then some unification constraints in G^+ must be inconsistent.

2. Then some unification must eventually fail in G, by the correspondence of the demand propagation effects between G^+ and G.

3. Since function invocations in FGL+LV are strict on unification success, there must at all times be a demand propagation path to the output node of G, along which the resulting error value will be returned to the top level.

8. Definiteness

We briefly mention that demand evaluation also ensures computational *definiteness*, whereby each value approximation printed at the top-level of an FGL+LV computation is sure not to have involved any computations yielding error. Thus the theoretically troubling presence of error at the top of our C. P. O. does not mean that printed values are forever subject to being superceded by error. This is because any approximate value is never delivered until all its strictly required subcomputations have been successfully performed. Thus we have the following "confidence test" on FGL+LV computations:

1. Present a pattern of demand at the main program output arc.

2. Wait until a value is returned fulfilling that demand pattern, or an error is reported.

3. If the returned value fulfills submitted desired demand pattern, we can be sure no error will ever result from computations required to fulfill that demand pattern.

4. However, if we *refine* the demand pattern by asking for greater detail in the answer, the possibility of an error result is once again in effect until the new pattern is fulfilled, etc.

In short, FGL+LV behaves *exactly* the way FGL does with respect to *any* kind of error: each output approximation produced is sure to have depended on no erroneous subcomputations. However, continuing the computation always raises the possibility once again of incurring an error.

A formalization of another example of this kind of reasoning may be found in [17].

9. Related Work

The challenge of amalgamating functional and logic programming is an active research area, involving a wide variety of approaches. These include finding a common generalization [4, 6], interfacing functional and logical programming systems [8, 21], and using logic programming as a functional programming application [18, 19]. However, three efforts bear special mention for their proximity to the work reported here.

9.1. HASL

In many ways the closest work to this has been done by Abramson in designing HASL [1, 2]. Like FGL+LV, HASL uses genuine unification within a functional programming language framework (SASL). An implementation of HASL in Prolog is available. Unlike FGL+LV, HASL programs can recover from failing unification attempts. However, HASL is apparently not designed to exploit concurrency.

9.2. FUNLOG

The *semantic unification* idea of FUNLOG [24] is essentially the same as that in FGL+LV (as was a prime inspiration for us). However, the semantics of FUNLOG are given only informally, and parallel evaluation effects, while clearly a strong motivation, have not been fully elucidated.

9.3. Concurrent Prolog

Finally, the connections of FGL+LV with Concurrent Prolog [22] are unmistakable. Concurrent Prolog is more advanced than FGL+LV in that parallel unifications can be done in an OR-parallel manner, i.e. without environment interaction, which is inescapable in FGL+LV. On the other hand, Concurrent Prolog is in general indeterminate, and to our knowledge does not as yet have a well-defined parallel implementation strategy.

10. Architectural Considerations

The reduction model for FGL+LV was developed with implementation on the Rediflow architecture in mind [9, 14, 15]. We briefly suggest what the special needs of FGL+LV are in this regard, and indicate our preliminary thinking on how they might be met with a small increase in architectural support.

* *Demand propagation effects:* The two levels of demand required by FGL+LV may be supported by extending Rediflow machine words to have two rather than one demand bits. Since assertive demands can overtake non-assertive ones, in some cases demand propagation may occur twice (but no more

often) along a given arc.

* *Node merging:* There is already already on Rediflow an invisible pointer mechanism (the `fetch/forward` opcode pair; this is currently used for formal to actual parameter linkages, as well as remote accesses of tuple components. This appears to be adequate for accomplishing unbound variable merging. Since we bias our unification to always attempt merging of unbound variables in formal parameters first, happily we are protected from the problem of two unbound variables being simultaneously equated to each other. This is in fact the only manifestation (since we are computing over a domain with infinite data structures) of the *occurs check* problem, which is so vexing to logicians and Prolog implementors.

* *Tasks and notifiers:* Rediflow uses *notifier* fields in a word to schedule appropriate tasks (instruction executions) when that word has a demand satisfied. Given the two-level demand strategy in FGL+LV, these notifiers will need to be segregated by demand type. However, this seems straightforwardly implementable.

11. Future Work

These results suggest many avenues for continuing research, including:

1. Generalizing the reduction model to permit outputting of results containing unbound variables (*non-ground terms*). The need for this capability will become vital if, as we would like, FGL+LV is used as a *target language* for compiling logic programs. Note that simply injecting patterns containing assertive demands at the top-level does not suffice, since definiteness is then lost. That is, we would not in general be able to tell conclusively when unbound variables in the output term can be considered "final", unless some global computational quiescence test is available.

2. Protected environments for *trial unifications* is clearly necessary if logic programming compilation, as suggested above, is to become feasible. We hope the algorithms being developed for unification in Concurrent Prolog will give some insights here.

3. Generalization to *infinite structure unification*, as suggested in [7] would minimize the danger of divergence during unification in FGL+LV. Although we support infinite data structures in FGL+LV, divergence will occur if atoms or logical variable occurrences do not terminate all paths. We conjecture that the invisible pointer facility of Rediflow (as mentioned above) can be used to maintain the equivalence hypotheses required by the Haridi algorithm.

4. Finally, the speculative idea of using unification as a basis for maintaining *changeable* constraint relationships [5, 23] is very appealing. Here we plan to investigate the ideas of FGL+LV in a broadened graphical execution model where *incremental recomputation* [16] is supported.

12. Conclusions

This exercise in language accretion has both strengths and weaknesses. Its *strengths* include:

1. an upward compatible logical variable extension to a graph reduction language;

2. a globally, but not locally, deterministic resulting computation model, and

3. an approach for its implementation on a distributed reduction architecture.

However, the following *weaknesses* are also evident:

1. a loss of some purity in the functional model;

2. strictness of all functions with respect to unification failure, preventing "trial" unifications or Prolog-like searching, and

3. an inability to return overall computational results involving unbound variables, as, again, can be done in Prolog.

Acknowledgements

This research was performed while the author was a visitor at Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia-Antipolis, France. The insights and hospitality of Gilles Kahn during this period are warmly appreciated. In addition, the tutelage of contemporaneous visitor John Reynolds on denotational semantics is also gratefully acknowledged.

References

[1] H. Abramson.
Unification-Based Conditional Binding Constructs.
In *Proc. First International Logic Programming Conference*. Marseille, 1982.

[2] H. Abramson.
A Prological Definition of HASL, a Purely Functional Language with Unification Based on Conditional Binding Expressions.
New Generation Computing 2:3-35, 1984.

(3) Department of Computer Sciences, Chalmers University of Technology, Aspenas, Sweden.
Symposium on Functional Languages and Computer Architecture, Laboratory on Programming Methodology, 1981.

[4] R. Barbuti, M. Bellia, G. Levi and M. Martelli.
On the Integration of Logic Programming and Functional Programming.
In D. DeGroot (editor), *International Symposium on Logic Programming*, pages 160-166. IEEE, Atlantic City, N.J., February, 1984.

[5] A. Borning.
The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory.
ACM TOPLAS 3(4):353-387, October, 1981.

[6] J.A. Goguen and J. Meseguer.
Equality, Types, Modules and Generics for Logic Programming.
In Sten-Ake Tarnlund (editor), *Second International Logic Programming Conference*, pages 114-125. Uppsala University, Uppsala, Sweden, July, 1984.

[7] S. Haridi.
Logic Programming Based on a Natural Deduction System.
PhD thesis, Royal Inst. of Tech., 1981.

[8] K. Kahn.
The Implementation of Uniform -- a Knowledge-Representation/ Programming Language Based on Equivalence of Descriptions.
In *ECAI*. Orsay, France, 1982.

[9] R.M. Keller, G. Lindstrom, and S. Patil.
A Loosely-Coupled Applicative Multi-Processing System.
In *AFIPS Conference Proceedings*, pages 613-622. June, 1979.

[10] R. M. Keller, B. Jayaraman, D. Rose, G. Lindstrom.
FGL (Function Graph Language) Programmers' Guide.
Technical Report AMPS Technical Memorandum No. 1, University of Utah, Computer Science Department, July, 1980.

[11] R. M. Keller.
Semantics and Applications of Function Graphs.
Technical Report UUCS-80-112, University of Utah, Computer Science Department, 1980.

[12] R.M. Keller and G. Lindstrom.
Hierarchical Analysis of a Distributed Evaluator.
In *Proc. International Conference on Parallel Processing*, pages 299-310. August, 1980.

[13] R.M. Keller.
FEL (Function Equation Language) Programmer's Guide.
Technical Report 7, University of Utah, Department of Computer Science, AMPS Technical Memorandum, 1982.

[14] R.M. Keller, G. Lindstrom, E. Organick.
Rediflow: A Multiprocessing Architecture Combining Reduction and Data-Flow.
In *PAW 83: Visuals used at the Parallel Architecture Workshop.* Department of Energy, Office of Basic Energy Sciences, University of Colorado, Boulder, January, 1983.

[15] R.M. Keller, F.C.H. Lin, and J. Tanaka.
Rediflow Multiprocessing.
In *IEEE Compcon '84*, pages 410-417. Feb., 1984.

[16] G. Lindstrom and R. Wagner.
Incremental Recomputation on Data-Flow Graphs.
In *Symposium on Functional Languages and Computer Architecture*, pages 472-489. Department of Computer Sciences, Chalmers University of Technology, Aspenas, Sweden, June, 1981.

[17] G. Lindstrom and F. E. Hunt.
Consistency and Currency in Functional Databases.
In *Proc. Infocom 83*, pages 352-361. IEEE, San Diego, April, 1983.

[18] G. Lindstrom and P. Panangaden.
Stream-based execution of logic programs.
In *Proc. 1984 Int'l. Symp. on Logic Programming*, pages 168-176. February, 1984.

[19] G. Lindstrom.
OR-parallelism on applicative architectures.
In Sten-Ake Tarnlund (editor), *Proc. Second International Logic Programming Conference*, pages 159-170. Uppsala University, July, 1984.

[20] U.S. Reddy.
On the Relationship Between Functional and Logic Languages.
In D. DeGroot and G. Lindstrom (editors), *Logic and Functional Programming.* Prentice Hall, 1985.
To appear.

[21] J.A. Robinson and E.E. Sibert.
Loglisp: an alternative to Prolog.
In Hayes, Michie, and Pao (editors), *Machine Intelligence*, pages 399-419. Ellis Horwood, 1982.

[22] E.Y. Shapiro.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, Institute for New Generation Computer Technology, January, 1983.

[23] G. L. Steele, Jr. and G. J. Sussman.
Constraints.
In *APL Conf. Proc., Part 1*, pages 208-225. APL Quote Quad, June, 1979.

[24] P.A. Subrahmanyam and J.-H. You.
Conceptual Basis and Evaluation Strategies for Integrating Functional and Logical Programming.
In D. DeGroot (editor), *International Symposium on Logic Programming*. IEEE, Atlantic City, N.J., February, 1984.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique