COMBINATOR EVALUATION OF FUNCTIONAL PROGRAMS WITH LOGICAL VARIABLES[1]

Technical Report UUCS-87-027

Göran Båge
Computer Science Laboratory
Ericsson Telecom
S-126 25 Stockholm Sweden

Gary Lindstrom
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112 USA

October 1987

[1]The research was supported in part by grant CCR-8704778 from the National Science Foundation, and by an unrestricted gift from Telefonaktiebolaget LM Ericsson, Stockholm.

COMBINATOR EVALUATION OF FUNCTIONAL PROGRAMS WITH LOGICAL VARIABLES

Abstract

A technique is presented that brings logical variables into the scope of the well known Turner method for evaluating normal order functional programs by `S, K, I` combinator graph reduction. This extension is illustrated by SASL+LV, an extension of Turner's language SASL in which general expressions serve as formal parameters, and parameter passage is done by unification. The conceptual and practical advantages of such an extension are discussed, as well as semantic pitfalls that arise from the attendant weakening of referential transparency. Only four new combinators (**`LV, BV, FN`** and **`UNIFY`**) are introduced. The resulting object code is fully upward compatible in the sense that previously compiled SASL object code remains executable with unchanged semantics. However, "read-only" variable usage in SASL+LV programs requires a "multi-tasking" extension of the customary stack-based evaluation method. Mechanisms are presented for managing this multi-tasking on both single and multi-processor systems. Finally, directions are examined for applying this technique to implementations involving larger granularity combinators, and fuller semantic treatment of logical variables (e.g. accommodation of failing unifications).

Contents

1 Extending Functional Programming With Logical Variables

1.1 Added Power

Functional and logic programming languages are alternative approaches to *applicative* (i.e. side-effect free) programming. While rivalry between partisans of these two approaches has at times been intense, there is now an atmosphere of *detente*, and each is adopting good ideas from the other. For example, a number of logic programming researchers are now carefully studying the *data directionality* effects of functional programming in an effort to achieve a satisfactory formulation of *AND-parallelism*, and some functional language designers are coveting the elegance of unification.

We offer here another step down the latter path, by considering how to incorporate logical variables into a functional language while preserving determinacy. Logical variables are variables in the mathematician's sense rather than in the computer scientist's: place holders for fixed but initially unknown quantities, which become known in stages by the application of equational constraints.

Logical variables can extend the power of functional programming in many important directions, including:

- Providing building blocks for monotonically refineable data structures (e.g. Prolog-style difference lists), aiding in the coordination of parallel processes;
- Representation of constraint-based problem solvers such as the Milner polymorphic typing algorithm [Mil78];
- Deft treatment of forward reference problems, such as symbol table management in compilers [CH87];
- Modeling of fundamentally bidirectional information flow, such as buses and pass transistors in hardware specification systems [She85,CGM87,PSE85], and
- Playing a "micro object" role in object oriented programming dialects (e.g. for shareable message "mailboxes") [KTMB86,Lin87b].

Note that none of these fundamentally depends on multi-path search, backtracking or even accommodation of failing unification (although the semantic merit of such extensions is undeniable). Thus our viewpoint throughout will be that a failing unification constitutes a run-time error. Only in Section 8.1 will we briefly consider the semantic and operational consequences of responding to unification failures.

1.2 Existing Language Designs

The design of FGL+LV, a normal order (lazy) functional language with logical variables, is described in [Lin85]. FGL+LV is based on the function graph language FGL [KJRL80]. More recently, the MIT Dataflow Architecture Group has extended the dataflow language Id to "Id Nouveau", which offers a specialized form of logical variables through write-once *I-structures* [NPA86]. Danforth has critically examined this language design area as a whole [Dan85].

1.3 Prior Implementations

A single processor implementation of Id Nouveau has recently been announced [Nik87]. An implementation of FGL+LV in the context of the Rediflow multiprocessor architecture [KL84] is outlined in [Lin87a]. The latter design is the only one known to us that combines logical variables and normal order evaluation in a thorough way. However, it was never implemented (or simulated), due primarily to its complexity and specific dependence on the Rediflow architecture. This complexity arose from two factors:

1. The use of two levels of explicit demand (*nonassertive*, or conventional demand, and *assertive*, arising from unification operators, powerful enough to elicit logical variable references as well as ordinary values);
2. The Rediflow two level task notification scheme (according to whether notification is within or across code blocks).

In retrospect, both these difficulties are due to Rediflow's one-to-one mapping of complete function definitions to individual large granularity combinators. By compiling to fine granularity combinators, both these problems are eliminated. (In Section 8.2 we briefly return to the issue of incorporating logical variables into "super" combinators.)

2 Compiling to Combinators

2.1 Basics

A combinator is an environment-free function.

- *Environment-free* means that a combinator involves no "imports", "fluids", "globals", or identifier scoping (static, dynamic, or otherwise), although senses of these can be supported indirectly through compilation (see Section 4.4).
- *Function* means that a combinator can be applied to argument(s) to compute a result without incurring any semantic effect except production of that result (i.e. no side-effects).

Combinators are typically *curried*, i.e. defined on one parameter at a time. Thus a curried add function might be `plus a b`, so that one could define `plus1 = plus 1`, and then apply `plus1` several times, e.g. `plus1 2` yielding `3`, `plus1 10` yielding `11`, etc.

2.2 SKI Combinators

The combinator notion originated with the `S, K, I` family, developed by Curry and Feys as a contribution to the foundations of mathematics [CF58]. In addition to `S, K, I`, this family generally includes curried combinators for the primitive operators of interest (e.g. `+`, `OR`, `P` (pair), `HD` (head), etc.), and conditional `COND`.

In 1979 Turner discovered [Tur79] that this family could plausibly be used for executing normal order functional programs, with several advantages:

```
a+b
where x             = 1 : (add1 x)
      add1 (h : t) = (h+1) : (add1 t)
      a : (b : c)  = x
```

Figure 1: Sample SASL source code.

1. Their fine granularity provides a homogenous program representation (e.g. all programmer defined function boundaries disappear) that greatly simplifies evaluation and storage management.
2. The two most important semantic features of modern functional languages, viz. normal order evaluation and higher order functions, are elegantly supported.
3. Opportunities for concurrent evaluation naturally arise at non-unary *strict* combinators, i.e. those known *a priori* to require the evaluation of two or more arguments.
4. The evaluation rules for the `S, K, I` family have direct interpretations as graph reduction rules. Moreover, "full laziness" results: subgraphs are evaluated at most once, even when shared across curried functions [AKP84].

2.3 SASL

The base language for our extension is a subset of SASL, a predecessor of Miranda[1] [Tur85]. A very simple example of SASL source code is shown in Fig. 1. This program defines the function `add1` which maps a stream (infinite list) of integers into another stream with component-wise incrementation (`u : v` denotes a right-associative pair construction, i.e. the Lisp `(cons u v)`). The function `add1` is used to define `x`, a stream representing the infinite sequence of natural numbers starting with `1` (note the cyclic definition of `x`). Finally, the first two elements `a` and `b` of `x` are summed to yield the program's overall result.

All SASL functions are curried, and syntactically parentheses are necessary only where default left associativity is not desired. Thus `(+ 1 (* 2 3))` means `((+ 1) ((* 2) 3))`, which evaluates to 7; `(+ 1 * 2 3)` evaluates to either an arithmetic error or to itself (i.e. it is already in normal form), depending on the error checking policy adopted. We will employ this minimal parenthesis notation henceforth.

2.4 Turner Compilation

Reference [Tur79] gives a method for compiling SASL to combinators, which we call *Turner compilation.*

[1] "Miranda" is a trademark of Research Software Ltd.

1. Since SASL is already curried, to compile expressions we need only convert their infix operators to their corresponding curried prefix form. For example, `1+3 : 2*(5-8)` compiles to `(P (+ 1 3) (* 2 (- 5 8)))`, where the `P` combinator is the curried prefix version of the infix : operator.

2. Applications of programmer defined functions compiled similarly, e.g. `(f (1:3:TRUE))` compiles to `(f' (P 1 (P 3 TRUE)))`, where `f'` is the compilation of SASL function `f` (see step 3).

3. Function definitions are compiled using the cornerstone of the Turner method, the *abstraction operation* `[x]exp`:

   ```
   [x](f p) = S [x]f [x]p [x]y = K y [x]x = I
   ```

   This operation is the inverse of function application, in the sense that

   ```
   ([x]E x) = E
   ```

A sample application of this operation is shown below on `twist a b = b:a` (using the second optimization described in Section 4.5).

```
twist = [a]([b]b:a)
      = [a]([b](P b a))
      = [a](S [b](P b) [b]a)
      = [a](S (S [b]P [b]b) (K a))
      = [a](S (S (K P) I) (K a))
      = [a]((S ((S (K P)) I)) (K a))
      = S [a](S ((S (K P)) I)) [a](K a)
      = S (K (S ((S (K P)) I))) (S [a]K [a]a)
      = S (K (S ((S (K P)) I))) (S (K K) I)
      = S (K (S (S (K P) I))) (S (K K) I)
```

This compilation method (without optimizations) is illustrated more fully in Fig. 2 on the program in Fig. 1. Note that direct "knot-tying" is used to represent recursion, rather than the more elegant but less efficient `Y` combinator (although the `Y` combinator has the advantage of yielding only acyclic graphs). Roots of shared subgraphs are prefixed by `i->` labels, and references (after the leftmost, in place of which the shared subgraph is shown) are indicated by `->i` notations. (We refrain from using the more traditional `1:(...)` notation, in light of possible confusion with SASL's : infix pair constructor.)

3 Combinator Evaluators

3.1 Sequential

The Turner compilation algorithm is accompanied by a clever sequential evaluation method. By repeatedly performing the outermost reduction, this method mechanizes normal order (lazy) semantics without explicit demand indicators or simulated multi-tasking. This approach underlies

```
(+ (HD

      1->(P 1

            (2->(S (S (K P) (S (S (K +) (U (S (K K) I))) (K 1)))

                   (S (U (S (K K) (K ->2))) (U (K I)))) ->1)))

   (HD (TL ->1)))
```

Figure 2: SKI combinator object code (unoptimized).

all efficient sequential evaluation methods for larger granularity combinators, e.g. the G-machine [Joh84].

Outermost reductions are efficiently located by a recursive traversal algorithm. In typical implementations this traversal is administered by a link reversal method which eliminates the need for an explicit stack (see Fig. 3). Since this technique forms the basis for our SASL+LV evaluation method, we review it briefly here. The key ideas are:

1. Two pointers are used, `c` (*current*) and `p` (*previous*). Initially `c` points to the root of the graph to be evaluated, and `p` has a `null` value. These pointers are used to traverse the graph by a technique known as *link permutation* [Lin73].
2. We assume that each pointer value, when stored, also records whether it resides in the *left* or *right* half of its node (the lowest order bit is generally available for this purpose, given byte addressing). This pointer tagging (not indicated in our figures) is sufficient for the evaluator to always be aware of whether it is *ascending* or *descending* to the current node `c`.
3. When descending from a node, the "backpointer" (old `p` value) is always stored in the right half of the node (this convention will simplify our extended method described in Section 3.2).

3.2 Multi-Tasking

As remarked in Section 3.1, the Turner evaluation method performs lazy evaluation simply by steadily performing the outermost reduction until normal form results. On a sequential machine, no further evaluation order embellishment is necessary. However, natural opportunities for concurrency arise within this framework which can be exploited on parallel architectures. The simplest non-speculative such opportunities are generated by non-unary (henceforth, *binary*) strict operators, e.g. addition [CP85]. A very clean technique exploiting these opportunities by stack discarding and reconstruction has been presented by Hughes [Hug87].

We sketch another approach here, exploiting link permutation:

1. Each task in our system is represented by a `[c, p]` pointer pair. We assume there is a background mechanism for allocating tasks to reduction processes.

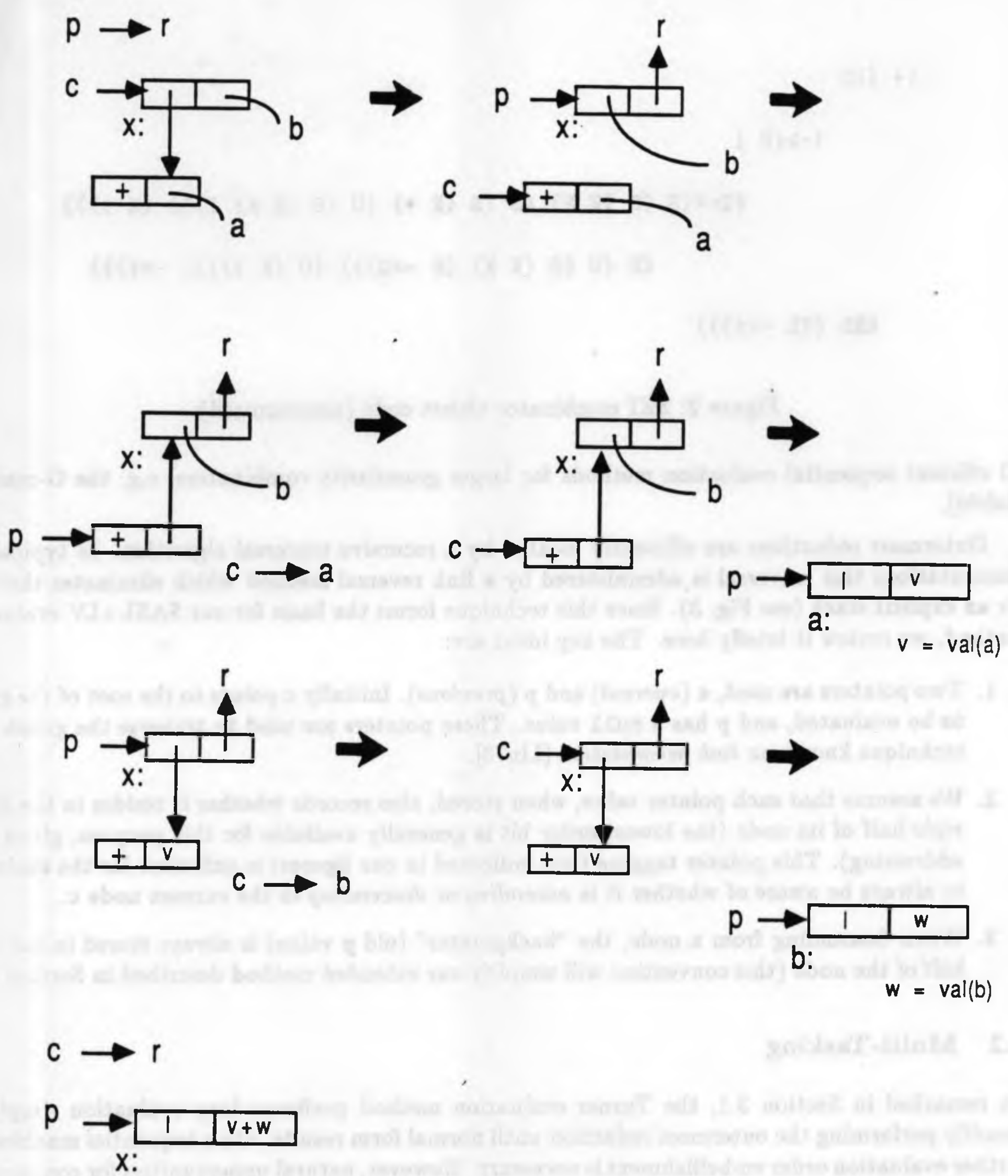

Figure 3: Normal order evaluation by link permutation.

2. Each node is augmented by a `busy` bit, set to `1` iff a task is currently evaluating that node. If `busy(n) = 1`, we say node `n` *is busy.*

3. When a task descends to a binary strict operator, it digresses to evaluate the operator's first argument. Before beginning that evaluation, however, a new task is created for evaluating the second argument. By convention, new tasks always are created in the *descending* mode.

4. If a task `[c, p]` descends to a busy node `n`, information is enqueued to permit creation of a new task `[n, p]` when evaluation of `n` is completed. Note that:

   - Any node undergoing evaluation has a backpointer (or set of backpointers, see below) in its right half.
   - Furthermore, since necessarily `c = n`, we need only record `p` to represent task `[c, p]`.
   - Finally, we note that the backpointer in `right(n)` is simply another such `p'` from a task `[n, p']` that happened to arrive at `n` first. Hence we view `right(n)` as holding a *set* of pointers.[2]

5. Now suppose that evaluation of a busy node `n` is completed, with the waiting task set in `right(n)` equal to {`p1, ..., pk`}. Then the current task continues as `[p1, n]`, and the remaining pointers are used to create tasks `[n, p2], ..., [n, pk]`. For consistency (see point 3), these `k-1` tasks are restarted created in the *descending* mode. However, they will reverse direction when they determine that node `n` is already in normal form (a `normal_form` mark bit could expedite this).

Fig. 4 illustrates this mechanism on an addition subgraph. Conceivably, this technique could have formed the basis of a shared memory multiprocessor version of the Burroughs Norma co-processor [Sch86], or the SKIM machine [CGMN80].

4 Introducing Logical Variables

4.1 SASL+LV

Given our focus on semantic and implementational matters, we wish to put aside the important language design issues of how logical variables should be introduced in a full, "rounded" manner into a lazy functional language. Thus we seek here an expedient approach which permits us to explore the fundamental "semantic impact" of logical variables when introduced into a mature base language.

Our choice for this purpose is SASL+LV, a subset of SASL extended to include logical variables and parameter passage by unification. SASL+LV is a "laboratory" language, and is not claimed to be complete or useful in any practical sense. However, it is more than simply a "gedanken" language, since serious implementations are being constructed (see Section 7.1).

[2]We assume availability of a second low order pointer bit (in addition the bit indicating `left/right` source, see Section 3.1) to distinguish the final value in a task descriptor chain. Two such bits are available under byte addressing, assuming nodes are at least four bytes wide, and word alignment is observed.

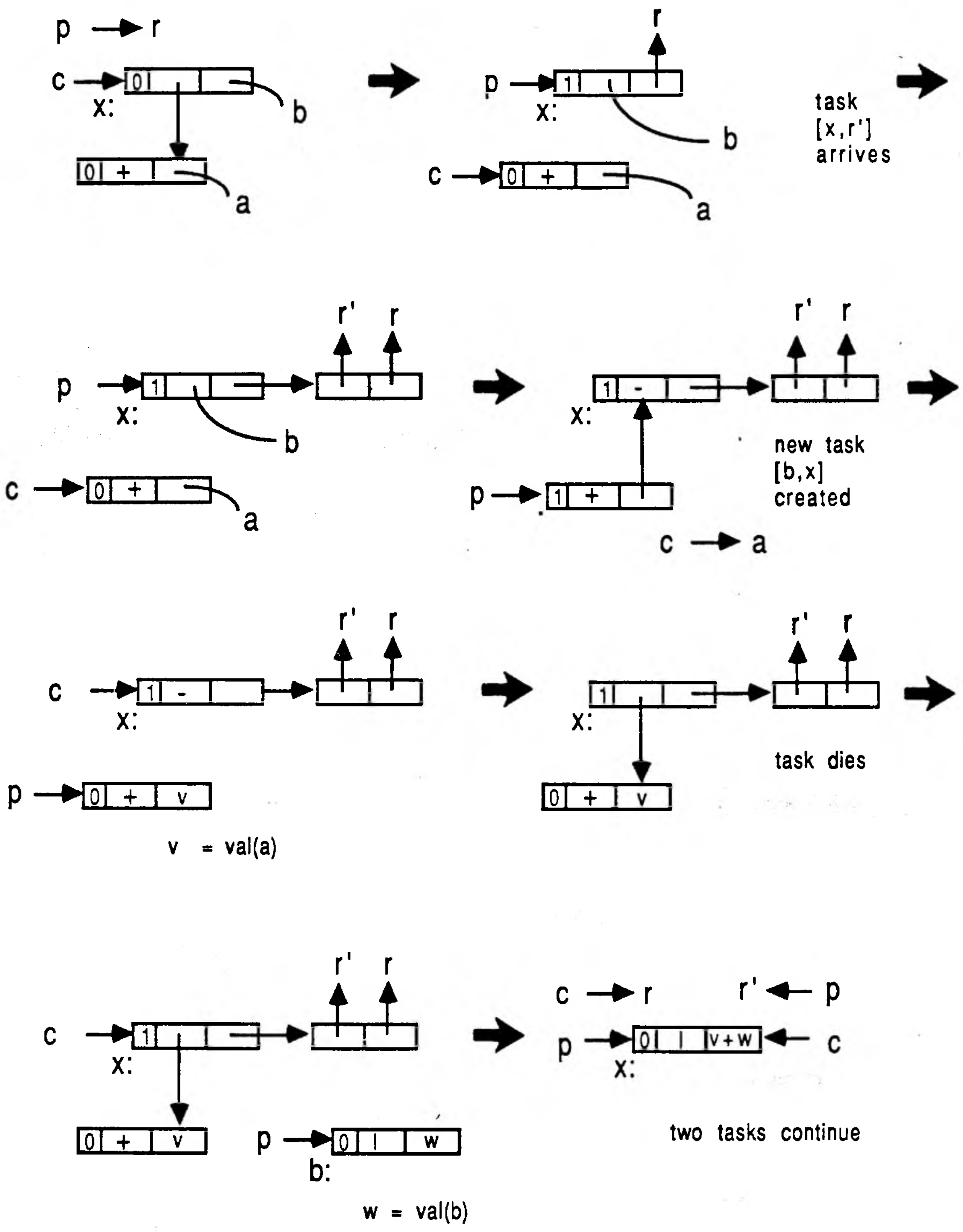

Figure 4: Strictness-based multi-tasking evaluation.

SASL+LV differs syntactically from SASL only through the generalization of formal parameters to arbitrary expressions, rather than the customary sequence of distinct identifiers (or patterns reducing to such). The informal semantics of a SASL+LV function `f formal = body` applied to `exp` are:

1. Create new instances of `formal` and `body`;
2. Unify `exp` with this instance of `formal`;
3. If successful, return the value of this instance of `body` under the bindings that resulted from the unification;
4. Otherwise, return an error indication.

4.2 Semantic Impact

The impact of logical variables on the formal semantics of SASL+LV is essentially the same as in FGL+LV: the injection of a well-behaved (i.e. monotonic) form of side-effects. In particular, the *sharing* of all references to a given logical variable is semantically *visible* via unification. Consequently, the law of "referential transparency" (roughly, that replication of expressions is semantically invisible) must be partially repealed.

To illustrate, consider the SASL+LV function `diff a:b b:c = a:c`, which mimics the Prolog `append` relation on "difference lists". We assume that `a`, `b`, and `c` all denote logical variables. Then the application `diff (1:2:x):x (3:nil):nil` (where `nil` is a special list ending constant) returns `(1:2:3:nil):nil`. Its correctness directly depends on all occurrences of the symbol `x` denoting *the same* logical variable.

4.3 Representation

A logical variable is represented by an "application" of the new combinator `LV`, e.g. `(LV tailx)`. As noted in Section 4.2, shared references to such nodes are crucial to correct logical variable semantics. Hence in Section 5.2 when reduction rules on `LV` combinators are described, we will refine this notation to `x->(LV tailx)`, in the manner of Fig. 2. Initially, an `LV` application has a cyclic self-reference as its argument, i.e. `x->(LV x)`. This configuration indicates that the *set* (or chain) *anchored* at `x` is empty. In Section 5.3 we shall see that when two distinct `LV` combinator nodes are unified, their chains are merged, and one has its combinator transformed to `BV` ("bound variable").

4.4 Compiling Logical Variables

Given the semantic visibility of logical variable sharing, we must ensure that each logical variable in a SASL+LV function is instantiated *precisely* when semantically dictated, i.e. *once* per function application. Any *less* and function bodies are no longer "pure code"; any *more* and we lose some unification correctness (e.g. desired inter-task communication effects). To arrange this:

1. We assume that *every* variable is a logical variable if not immediately bound by appearance in the left side of an equation.
2. We treat each logical variable in a function definition as a special kind of formal parameter that gets bound to a "fresh" `LV` instance prior to *each* application of the function that introduces it.

However, this semantic requirement poses a language design question concerning the interaction of logical variables and currying. To illustrate, consider the SASL+LV function `diff`, defined in Section 4.2. What should be the meaning of `df = diff x:y`? Clearly, we must create an instance of the `diff` function with `a` unified to `x`, and `b` unified to `y`. But should this be done:

- *once* when `df` is created by the application of `diff` to `x:y`, or
- *repeatedly* when `df` is applied to various arguments?

Either approach is semantically defensible and implementationally viable. However, we adopt the first policy, on the basis of consistency with conventional currying.

- For a curried function of n arguments, there are n opportunities for the instantiation of logical variables, i.e. after application to the i-th argument, for $i = 1, ..., n$.
- Each logical variable `x` occurring in a formal parameter expression is instantiated at the i-th stage of curried application, for the minimum i such that `x` occurs in the i-th formal parameter expression.
- Logical variables occurring only in the function body are considered to occur in the last formal parameter expression.

Thus we compile `diff` in two stages, as though it were defined:

```
diff a:b = diff1
where
        diff1 b:c = a:c
```

The compiled representation `cdiff` of `diff` uses the new combinator `FN`:

```
cdiff  =  FN [a]([b](P (a:b) cdiff1))
cdiff1 =  FN [c](P (b:c) (a:c))
```

As always, the Turner abstraction operator is applied innermost-first, so the construction of `cdiff1` is done before `cdiff` is constructed.

The resulting compiled form `cdiff` thus has two levels of currying: an inner one needing one logical variable "parameter" (`c`), and an outer one needing two (`a` and `b`). These needs are satisfied at application time by the complementary action of "distributing in" appropriate numbers of new logical variables (see Section 5.2).

In summary, the general rule for compiling a function `f formal = body` is as follows. Suppose `formal` introduces `n` logical variables, `v1, ..., vn`. Then the compiled image of `f` is `FN fb`, where:

- [Rule CFN0:] If `n = 0`, `fb` is simply the Turner compilation of `P formal body`. However, function definitions within `body` (or `formal`, for that matter!) must be compiled with awareness of rules `CFN0` and `CFN1`.
- [Rule CFN1:] Otherwise, `n > 0`. In this case `fb` is `g'`, where:
  - `g'` is the compilation of `g v1 ... vn = P formal body`.
  - Since the variables `v1, ..., vn` are all distinct, conventional Turner compilation applies, i.e. `g' = [v1](...([vn](P formal body))...)`.
  - As in rule `CFN0`, inner function definitions must be compiled in cognizance of rules `CFN0` and `CFN1`.

4.5 Recognizing Logical Variables

In Section 4.4 it is assumed that *every* symbol not occurring in the left side of an equation is a logical variable. This is a safe assumption, but can lead to needless overhead whereby ordinary parameter passage is done by unification. Much better object code can result by recognizing contexts where the standard Turner compilation method can be applied.

To assist in this, one may simply adopt the Prolog convention of using uppercase initial letters to designate logical variables. Alternatively, one can use a simple contextual discrimination method, as follows. A variable `x` is considered to denote a logical variable only if at least one of the following conditions applies:

1. `x` occurs more than once in the formal parameter sequence of a function.
2. `x` occurs in a *complex* formal parameter expression, i.e. one containing operator or function applications.
3. `x` occurs only in the function body, and is not defined by an equation (i.e. `x` is *free* in the function body).

Now, suppose that in compiling `f formal = body` as described in Section 4.4, `formal` is determined by the above analysis to be an ordinary parameter `x`. In this case, compilation to the `FN` combinator is not obligatory, and the Turner compilation `[x]body` can be applied. No confusion will occur at application time, for the resulting compiled form will not have `FN` as its outermost combinator.

Finally, we note that one well known optimization continues to apply to the Turner compilation method, as extended here. The compilation of `[x](fn arg)` yields `S [x]fn [x]arg`. This roughly doubles code size, and repeated application for heavily curried functions can be burdensome in both space and run time. However, if `x` is known by pre-scan *not* to occur in `fn arg`, the compilation can be short-circuited to `K (fn arg)`. More generally, `[x]E` can compile to `K E` if `x` is known not to occur in `E`. Since rule `CFN1` performs abstraction over logical variables by converting them to ordinary parameters, this optimization applies equally to logical variables and ordinary parameters.

5 Reduction Rules

5.1 Special Precautions

We now present a reduction semantics for SASL+LV. For truly functional languages, reduction semantics are *sound* in the sense that appropriate rules can always be applied without compromising semantic validity. Strategic application of these rules is thus an operational, but not semantic, issue.

However, this is not true of SASL+LV, since unneeded (or "speculative" [Bur85]) UNIFY reductions can cause binding conflicts that would not otherwise arise. A suitable rule application strategy will be described in Section 6. For the present, we enumerate the "bare" reduction rules for SASL+LV, assuming temporarily that somehow all, and only, appropriate rules are applied during evaluation.

5.2 FN Reduction

Consider now the actions to be taken when we apply a function that has been compiled to `FN fb`. New instances of all the logical variables introduced in `fb` must be created and distributed into a new copy of `fb` prior to commencing unification. This is accomplished by the following rules:

```
FN (P formal body) actual =>                          [Rule RFN0]
    COND (UNIFY formal actual) body ERROR

FN fb actual =>                                       [Rule RFN1]
    FN (fb new_lv) actual
```

Note that:

1. Rule RFN1 is to be applied only if rule RFN0 does not apply.
2. Rule RFN0 is the "base case" resulting from compilation rule CFN0, i.e. where no further logical variables need to be instantiated and distributed into `formal` or `body`.
3. Correspondingly, rule RFN1 applies to representations resulting from compilation rule CFN1. Since compile time abstraction was done over at least one variable, the outermost combinator cannot be P.[3]
4. The notation `new_lv` indicates the creation of a new logical variable node `x->(LV x)` in initialized self-referential form.

5.3 UNIFY Reduction

An application `UNIFY a b` has the following informal semantics:

[3]If certain optimizations are applied, this criterion may not be totally reliable. For example, `head1 b = 1:b` might be compiled to `P 1`. If this is a possibility, a combinator FN0 could be introduced to indicate explicitly the rule CFN0 case.

(Highest priority)

```
[Rule UN1]      UNIFY x->(LV tailx) x->(LV tailx)  =>  I TRUE

[Rule UN2]      UNIFY x->(LV tailx) y->(LV taily)  =>  I TRUE
                                    x->(LV tailx)  =>  x->(LV taily)
                                    y->(LV taily)  =>  y->(BV tailx)

[Rule UN3]      UNIFY x->(LV tailx) y              =>  UNIFY tailx y
                                    x->(LV tailx)  =>  x->(I y)

[Rule UN4]      UNIFY x->(BV tailx) y              =>  UNIFY tailx y
                                    x->(BV tailx)  =>  x->(I y)

[Rule UN5]      UNIFY x y->(LV taily))             =>  UNIFY x taily
                                    y->(LV taily)  =>  y->(I x)

[Rule UN6]      UNIFY x y->(BV taily)              =>  UNIFY x taily
                                    y->(LV taily)  =>  y->(I x)

[Rule UN7]      UNIFY x->(I z) y                   =>  I TRUE

[Rule UN8]      UNIFY x y->(I z)                   =>  I TRUE

[Rule UN9]      UNIFY (P ax bx) (P ay by)          =>  AND (UNIFY ax ay)
                                                           (UNIFY bx by)

[Rule UN10]     UNIFY x x                          =>  I TRUE

[Rule UN11]     UNIFY x y                          =>  I FALSE
```

(Lowest priority)

```
[Rule BV1]      f (BV tail)                        =>  f tail
```

Figure 5: Unification combinator reduction rules.

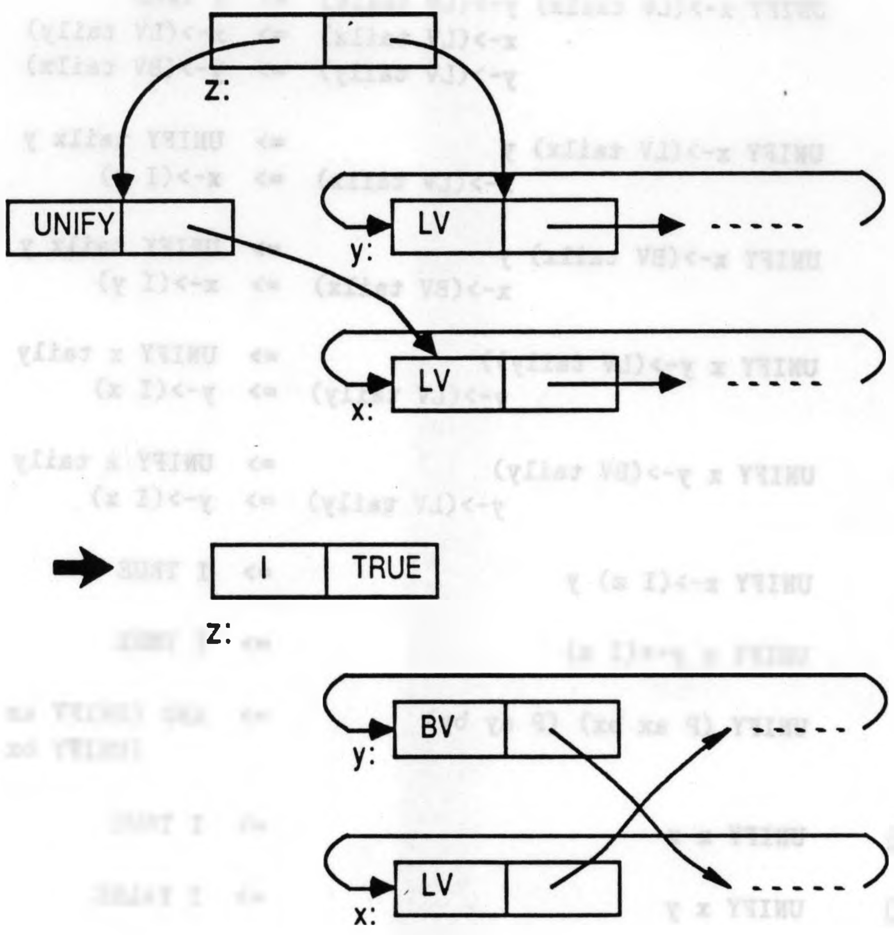

Figure 6: Unification rule UN2.

- Expressions `a` and `b` are evaluated to normal form, which can now include "applications" of the `LV` combinator.
- If `a` and `b` evaluate to *the same* logical variable, `UNIFY a b` simply reduces by rule `UN1` to `I TRUE` (see Fig. 6).
- If `a` and `b` evaluate to *different* logical variables `x->(LV tailx)` and `y->(LV taily)`, then the two logical variables are *equated* in rule `UN2` by:
  - *interchanging* `tailx` and `taily`, and
  - changing the `LV` combinator of one variable (say `y`) to `BV`.

  This merges the two cyclic lists anchored by `x` and `y` into one cyclic list with `x` remaining as the `LV` anchor. `UNIFY a b` reduces to `I TRUE`.
- If one of `a` or `b` (say `a`) evaluates to a logical variable `x->(LV tailx)` and the other (say `b`) evaluates to some `y` which is *not* a logical variable, then we bind `x` to `y` by reducing it to `x->(I y)` (rules `UN3` and `UN5`). The process is then repeated by rules `UN4` and `UN6` if `BV` combinator nodes are in the chain that was anchored at `x`. Ultimately the base of the cyclic chain at `x` is reached, i.e. `UNIFY x->(I z) y` (rules `UN7` and `UN8`), at which time this binding sequence is completed.[4]
- Otherwise, `a` and `b` have evaluated to `x` and `y`, respectively, neither of which are logical variables. We then do the usual unification case analysis:
  - If `x` and `y` are pairs `P ax bx` and `P ay by`, respectively, then `UNIFY a b` reduces by rule `UN9` to `AND (UNIFY ax ay) (UNIFY bx by)`.
  - Otherwise, `UNIFY a b` reduces to `I TRUE` or `I FALSE`, according to whether `x` is the same as `y` (rules `UN10` and `UN11`).

This informal reduction semantics is formalized by the reduction rules in Fig. 5. Also included is rule `BV1`, which removes `BV` node references from the set of normal forms (i.e. forces a chain of `BV` nodes to be "walked" to its `LV` node anchor).

5.4 A Closer Look at `UNIFY`

This formulation of unification takes a middle road with respect to each of three controversial issues.

Treatment of equality: Rules `UN10` and `UN11` call for `UNIFY a b` to reduce to `I TRUE` or `I FALSE` according to whether `a` and `b` are the same. To remain on semantically solid ground, this "sameness" condition should only apply to identical symbols. However, we suggest that two references to the same node should also be considered to be the same. This policy is trivial to implement (like Lisp's `eq`), and does extend unification completeness (see below), albeit at the cost of further erosion of referential transparency.

Occur check: The underlying normal order (lazy) semantics means that our computational domain includes meaningful infinite structures (e.g. `x = 1 : (add1 x)` in Fig. 1), in contrast to

[4]Necessarily, `y = z` in this case, but no test for this condition is required.

the Herbrand domain underlying Prolog. Hence unifications resulting in expressions with infinite denotations must not be rejected as semantically meaningless.

Our unification algorithm is incomplete in dealing with infinite structures (whether represented cyclically or by infinite recursion). Roughly speaking, it is complete in unifying `u` and `v` if each selector (`head/tail`) sequence applied to `u` and `v` yields an atom or an unbound logical variable in either `u` or `v` (or the same node, see previous point). For example, `x` in Fig. 1 would unify with `1 : 2 : y` where `y` is an unbound logical variable, but not with `z = 1 : (add1 z)`. A more powerful unification method involving node equivalencing is reported in [Har81]; related techniques have been developed for applicative caching [Hug85].

Unification of functions: General unification of functions constitutes higher order unification, which is known to be undecidable. Many approaches involving unification of functions represented by compiled code take the simplifying view that functions are *atoms*, and hence are equal only if identical (i.e. have the same code pointer). In our approach, functions are networks of nodes just like any other value. Hence functions can be unified to unbound logical variables, an essential feature for retaining the crucial capability of using functions as actual parameters.

But should we allow two functions to be unified? The rules in Fig. 5 are conservative in this respect, permitting successful unification of two functions only if their representations are identical, i.e. are the same subgraph (assuming rules `UN10` and `UN11` are given `eq` interpretations). However, a change to rule `UN9` *can* permit other function unifications to succeed: we simply replace it with `UNIFY (ax bx) (ay by) => AND (UNIFY ax ay) (UNIFY bx by)` (note the original effect of rule `UN9` is subsumed by this new specification).

Now two functions will unify not only if they share the same representation, but also if they have unifiable representations. Of course, this is still theoretically incomplete, even though normal order evaluation effects will permit *some* differing representations to unify (e.g. `+ (* 2 3)` and `+ 6`). Nevertheless, this approach does permit unification to play a role in computation on functional values, e.g. on-the-fly function construction as in interpretive Lisps.[5]

6 Sequential Implementation of SASL+LV

6.1 Basic Requirements

As suggested in Section 5.1, there are three areas in which the operational semantics of SASL+LV presents special challenges in comparison to SASL and other purely functional languages:

1. The set of normal form values must be enlarged to include references to logical variables;
2. Access of a variable can occur before the source of its binding is known [Red86]. Hence Turner's stack-based normal order evaluation is no longer adequate, and some means of multi-thread control is inescapable [LGY87b].
3. Special care must be taken to avoid making reductions that are uncalled for by normal order semantics, lest conflicting unifications be applied needlessly.

[5]The problem of unifying "equal" objects with differing representations is not new to anyone who has tried using floating point numbers in a Prolog system.

Fortunately, the absence of reduction rules for the LV combinator (outside the UNIFY rules) means *de facto* the set of normal form expressions is enlarged to include LV references. However, we do need to extend our evaluation method so that when a `x->(BV tailx)` node is descended to, the chain at `tailx` is "walked around" until LV node anchoring the chain is reached. This was specified in Fig. 5 by rule BV1; however, we now need to "step over" chained task pointers as well. This is achieved by the following "pseudo"-reduction rule applied when a task `[c, p]` descends to a BV node at c:

```
c = x->(BV tailx)  =>
        repeat c := right(c);
        until  left(c) = LV;
        c <-> p;              /* swap c and p */
```

6.2 (LV ...) Values

We now must decide what combinators are to do when they need an "ordinary" value (atom or pair), but are delivered instead an `x->(LV tailx)`. The answer is clear: treat such situations as *read-only* accesses, à la Concurrent Prolog [Sha83,Sha86]. This involves ending the task evaluating that combinator, in the assurance that a new task will resume its evaluation when the logical variable at x is bound to an ordinary value.

To implement this, we simply use `tailx` as a chain, just as is done for references to busy nodes. Specifically, assume task `[c, p]` ascends to c with a value `p = x->(LV tailx)`. This results in a reference to task `[c, p]` being added to the set chained at `right(x)`. Again, since p is implicitly represented by x, the LV base of the chain, we need only chain store c in the chain anchored at p:

```
right(x) := node(c, tailx);
```

To review, we now have two representations for sets of waiting tasks, which are kept distinct and never directly interact:

1. *Linear chains* originating in the `right` field of busy nodes, holding tasks awaiting that node's value, and
2. *Cyclic chains* anchored at LV nodes, holding equivalenced BV nodes and tasks awaiting binding of that logical variable.

6.3 Bind Operation

Consider now what happens when an LV node is bound to a non-LV value y. The applicable reduction rules from Fig. 5 (ignoring symmetric cases) are:

```
[UN3]   UNIFY x->(LV tailx) y              =>  UNIFY tailx y
                            x->(LV tailx)  =>  x->(I y)
```

```
[UN4]    UNIFY x->(BV tailx) y               =>  UNIFY tailx y
                             x->(BV tailx)   =>  x->(I y)

[UN7]    UNIFY x->(I z) y                    =>  I TRUE
```

The sequence of actions represented by this rule set is a good candidate for macro execution in a real implementation. In such a routine, we must (first!) reduce x to I y to record its evaluation to y, and then create new tasks as indicated in the chain at tailx:

```
q := right(x);
left(x) := I;  right(x) := y;  /* Reduce x to (I y) */
while (q != x)
{ if (left(q) != BV)
     /* step over BV nodes */
     create_task(x, left(q));  /* descending mode, as always */
  q := right(q); }
```

6.4 BV Combinator

Let's now examine the role of BV nodes in cyclic chains anchored by LV nodes. Beyond being simply vestiges of LV equivalencing, BV nodes constitute potential "entry points" for subsequent LV accesses. Recall that in rule UN2 a y->(LV taily) which is bound to a x->(LV tailx) gets converted to y->(BV tailx). Later, a shared access of y may cause some other task to descend to y and walk the chain to the base x->(LV tailx) (or its successor by equivalencing). The task will then access x as the "value" of y, i.e. the unique representative of the LV equivalence class containing x and y.

All this is achieved by the pseudo-reduction rule given in Section 6.1. Note, however, that if the access is from a combinator needing an "ordinary" (non-LV) value, then that task can be directly spliced into the chain at y->(BV taily), since in this case the exact identity of the base LV occurrence is irrelevant.

7 Parallel Evaluation

7.1 Pseudo-Parallel Implementation

A prudent step in the development of any parallel programming system is the construction of a pseudo-parallel version in which concurrency is simulated on a single processor. In such a system every "active phase" between scheduling operations is *de facto* an atomic action. This intermediate development stage facilitates verification of task representation and coordination techniques, before dealing with issues of true concurrency (locking, load distribution, concurrent storage management, etc.).

The link-permutation SASL evaluator described in Section 3.1 has been implemented in C, and is currently being extended to execute SASL+LV via multi-tasking (the two issues are inseparable, as explained in Section 6.1). This is being done very conveniently via the tasking library [Str85] of C++ [Str86].

This simulation is based on a "task heap" model, whereby a fixed number of processes draw tasks from a central pool. Tasks are matched to processes through a *dual queue*, in which either task or process descriptors are enqueued, depending on which at the moment is in surplus. If the dual queue is nonempty, it contains exclusively tasks or exclusively processes, since available tasks are paired to waiting processes without delay.

7.2 Concurrent Implementation

Upon completion of its pseudo-parallel implementation, our SASL+LV evaluator will be ported to the true multiprocessing environment of our 18-node BBN Butterfly. This machine offers a shared memory abstraction, whereby local and nonlocal memory can be uniformly addressed, with approximately 4 to 1 speed penalty for nonlocal accesses. Our selection of the dual queue task management model is motivated by the Butterfly's hardware support for this mechanism, as well as our belief that a *multi-sequential* (one process per processor) [TL87] process organization is economically most appropriate in this environment.

In a concurrent implementation, care must be taken that certain *test and set* operations occur atomically. We now enumerate those operations for our SASL+LV evaluation method. In the following, ▷ and ◁ mark the beginning and end of an atomic action, respectively.

1. *Descend:* When a task `[c, p]` undertakes the evaluation of the node at `c`, ▷ (i) test if `busy(c) = 1`, and if so, enter `p` into the chain at `right(c)`; ◁ otherwise (ii) set `busy(c) = 1` ◁, and begin evaluation at `c`.

2. *Combinator application:* When a task `[c, p]` completes evaluation of node `c`, it must ▷ (i) save `right(c)` in a temporary variable; (ii) rewrite node `c`, and (iii) set `busy(c) = 0` ◁. Other tasks are now permitted to access node `c` in its evaluated state, while tasks are created from the set of pointers obtained from `right(c)`.

3. *Strict node rendezvous:* When a task `[c, p]` completes the evaluation of an operand to a binary strict operator at a node `c` it must: ▷ (i) test if the other operand at `c` has been evaluated, and if so, ◁ perform the operation obtaining result `val` (see step 2), and continue; otherwise: (ii) permute links at `c` ◁ and terminate.

4. *Enqueueing a read-only task:* When a task `[c, p]` ascends to a node `c` needing an ordinary value, it must: ▷ (i) test if `p` points to an `LV` combinator node; if so, (ii) do `right(p) := node(c, right(p))` ◁, otherwise: (iii) ◁ continue evaluation at `c`.

5. *Binding a logical variable:* Recall that a task accessing a logical variable is enqueued in that variable's chain only when its value (reference) is reported to a combinator needing an ordinary value. Hence any number of active tasks may hold outstanding accesses on a given logical variable. When a `UNIFY` node attempts (by rule `UN3`) to bind a node `x->(LV tailx)` to a value `val`, it must be prepared to react if the variable at `x` is in fact already bound by the time it seeks to do so. In this case, the recursive comparison of values in `UNIFY` is resumed using `val` and the new value, via rules `UN9` - `UN11` [Lin84]. The only critical region

in this binding operation as described in Section 6.3 is ▷ `q := right(x); left(x) := I; right(x) := y;` ◁. This can be accomplished by a simple spin lock at `x`.

6. *Equating two logical variables:* Similarly, the operation of unifying two logical variables must be robust enough to arbitrate among all competing such actions involving one of the variables. Fortunately, the cycle merging technique described in Section 5.3 lends itself well to this requirement. A distributed (message based) version of this operation is presented in [Lin87a]. Again, only test and set operations on individual nodes are required.

7. *Dual queue operations:* The dual queue operations `Wait_DualQ()` and `Post_DualQ()` used for allocating tasks to processes are guaranteed to be atomic by the Butterfly hardware.

Finally, we acknowledge that a heap-intensive concurrent programming system such as this must include an efficient, reliable parallel garbage collection facility. General methods such as [AH87] can be applied; however we point out that in our approach all "extra" nodes created simply to chain tasks are guaranteed not to be shared, and can summarily be recycled when removed from their chain.

8 Future Work

Two clear paths lie ahead for this work: completing and critically appraising our concurrent evaluator, and gaining understanding of the many unresolved language design issues through experimentation with significant sized programs. In addition, three deeper areas beckon.

8.1 Dealing With Unification Failures

In SASL+LV a unification failure results in an `ERROR` value. Since no rules are specified to consume this value, the only reductions that will ensue are those which are not strict on that value. Eventually, the reduction evaluation will halt with the resulting normal form providing in effect a postmortem of the context(s) in which that `ERROR` blocked further evaluation.

Why not support conditional unification by testing for resulting `ERROR` values? There are good reasons, both semantic and pragmatic.

- *Semantic:* Determinacy would be lost, since if two unifications apply conflicting bindings to a shared logical variable, the *first* to occur would succeed, and the *second* would fail. For determinacy to be retained, one would need `FALSE` above `TRUE (so far)` in the underlying domain, and no result depending on a `TRUE` value could ever be fully trusted!

- *Pragmatic:* Detecting a failing unification is not sufficient; all bindings done in that failing attempt would need to be retracted. In a distributed reduction model, the feasibility of this in general is dubious.

However, some cold comfort may be drawn from the committed choice logic programming community, which is experiencing a similar conundrum. Their reaction has been to introduce "flat" indeterminacy [IMT87,TSS86,FT87,Mie84], whereby unifications are done on a "shadow" basis until commitment, and relations in guards are required to be primitive.

We conjecture that an analogous policy is feasible within an extension of our reduction model for SASL+LV. To illustrate, we note that a restricted form of committed choice indeterminacy is supported by the current model, if the binary combinator COMMIT is added:

```
COMMIT x y  =>  TRUE          /* if x is not equal to ERROR */
COMMIT x y  =>  FALSE         /* if y is not equal to ERROR */
COMMIT ERROR ERROR  =>  TRUE
COMMIT ERROR ERROR  =>  FALSE
```

COMMIT is clearly a pseudo-combinator, because the value of COMMIT **x y** for if **x** and **y** are either both equal or both unequal to ERROR is indeterminate. The easily implemented operational intent of COMMIT is to evaluate its arguments in parallel, and return TRUE or FALSE as soon as either argument evaluation terminates, according to whether or not that argument delivered ERROR.

Now suppose we wish to hold a committed choice "competition" between two SASL+LV functions f1 and f2 each applied to the same "goal" parameter g. Importantly, we only consider here the case where at least one of f1 and f2 are sure to succeed on g, and their unifications are exclusively *one-way*, i.e. only bind variables they introduce. Bindings of non-local variables are of course permitted in the bodies of f1 and f2 after commitment. We endow f1 and f2 with an extra parameter, used to signal commitment, e.g. f1 actual goahead = body. Then the desired committed choice effect is obtained by the SASL+LV function **solve**, defined as follows:

```
solve f1 f2 g = if    commit x y
                then  x true
                else  y true

                where x = f1 g
                      y = f2 g
```

8.2 Larger Granularity Combinators

The initial excitement with Turner's application of SKI combinators to functional programming has been tempered by empirical evidence that their granularity is too fine for efficient implementation on the architectures of today. In response, a wide range of "supercombinator" schemes have been proposed [Hug82,GH85,Joh84,Kie85]. These approaches exploit compile time analysis to aggregate operators into clusters that can be mapped to blocks of conventional machine code. We are currently examining how one such method, based on automated strictness analysis [LGY87a], can be adapted to include logical variables.

8.3 Generalized Abstract Interpretation

Finally, it has been pointed out that strictness analysis, polymorphic type checking [KM87], and logical variable mode analysis [DW86] are all complementary facets of the general technique of abstract interpretation on applicative languages. We seek as a longer term goal to construct a suitable domain for combining these analyses into one comprehensive method, and applying its results to the optimized compilation of an advanced language integrating functional and logic programming.

References

[AH87] K. M. Ali and S. Haridi. Global Garbage Collection for Distributed Heap Storage Systems. *International Journal of Parallel Programming*, 15(35):339–387, October 1987.

[AKP84] Arvind, Vinod Kathail, and Keshav Pingali. Sharing of Computation in Functional Language Implementations. In *Proceedings of International Workshop on High-level Computer Architecture*, Los Angeles, May 21-25 1984.

[Bur85] F. W. Burton. Speculative Computation, Parallelism and Functional Programming. *IEEE Transactions on Computers*, C-34(12):1190–1193, 1985.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic.* Volume 1, North Holland, 1958.

[CGM87] Albert Camilleri, Michael C. Gordon, and Tom Melham. Hardware Specification and Verification using Higher Order Logic. In Dominique Borrione, editor, *Proceedings of the IFIP Working conference "From HDL Descriptions to Guaranteed Correct Circuit Designs"*, North-Holland, Grenoble, France, 1987.

[CGMN80] T.J.W. Clarke, P.J.S. Gladstone, C.D. Maclean, and A.C. Norman. SKIM - The S, K, I Reduction Machine. In *Proc. Symp. on Lisp and Func. Pgmming. and Computer Architectures*, pages 128–135, ACM, 1980.

[CH87] Jacques Cohen and Timothy Hickey. Parsing and Compiling Using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

[CP85] C. Clark and S. L. Peyton-Jones. Generating Parallelism From Strictness Analysis. In *Prof. Conf. on Func. Prog. Lang. and Comp. Arch.*, IFIP, Nancy, France, September 1985.

[Dan85] S. H. Danforth. *Logical Variables for a Functional Language.* Technical Report PP-120-85, Microelectronics and Computer Technology Corp., 1985.

[DW86] Saumya K. Debray and David S. Warren. Automatic Mode Inference for Prolog Programs. In Robert M. Keller, editor, *Symposium on Logic Programming*, pages 78–88, IEEE Computer Society, Salt Lake City, September 1986.

[FT87] Ian Foster and Stephen Taylor. Flat Parlog: a Basis for Comparison. *International Journal of Parallel Programming*, 16(2), April 1987.

[GH85] B. Goldberg and P. Hudak. Serial Combinators: Optimal Grains of Parallelism. In *Proc. Conf. on Functional Programming Languages and Computer Architectures*, pages 382–399, Springer Verlag, Nancy, France, 1985. Lecture Notes in Computer Science, number 201.

[Har81] A. Seif Haridi. *Logic Programming Based on a Natural Deduction System.* PhD thesis, Royal Institute of Technology, 1981.

[Hug82] J. Hughes. Super Combinators: a New Implementation Method for Applicative Languages. In *Proc. Symp. on Lisp and Func. Pgmming. and Computer Architectures*, pages 1–10, ACM, Pittsburgh, Pa., 1982.

[Hug85] J. Hughes. Lazy Memo-functions. In *Prof. Conf. on Func. Prog. Lang. and Comp. Arch.*, IFIP, Nancy, France, September 1985.

[Hug87] J. Hughes. A Simple Implementation of Concurrent Graph Reduction. In R. M. Keller and J. Fasel, editors, *Proc: Santa Fe Workshop on Graph Reduction*, Springer-Verlag, 1987. Lecture Notes in Computer Science 279.

[IMT87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In Jean-Louis Lassez, editor, *Proc. International Conference on Logic Programming*, pages 257–293, MIT Press, Melbourne, Australia, May 1987.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proc. Symp. on Compiler Const.*, ACM SIGPLAN, Montreal, 1984.

[Kie85] R. B. Kiebutz. The G Machine: A Fast Graph-Reduction Evaluator. In *Proc. Conf. on Functional Programming Languages and Computer Architectures*, pages 400–413, Springer Verlag, Nancy, France, 1985. Lecture Notes in Computer Science, number 201.

[KJRL80] R. M. Keller, B. Jayaraman, D. Rose, and G. Lindstrom. *FGL (Function Graph Language) Programmers' Guide.* Technical Report AMPS Technical Memorandum No. 1, University of Utah, Computer Science Department, July 1980.

[KL84] R. M. Keller and F. C. H. Lin. Simulated Performance of a Reduction-Based Multiprocessor. *IEEE Computer*, 17(7):70–82, July 1984.

[KM87] T.-M. Kuo and P. Mishra. On Strictness and its Analysis. In *Proc. Symp. on Princ. of Pgmming. Lang.*, ACM, Munich, West Germany, March 1987.

[KTMB86] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. Objects in Concurrent Object Programming Systems. In *Proc. OOPSLA '86*, pages 242–257, Portland, OR, 1986.

[LGY87a] G. Lindstrom, L. George, and D. Yeh. Generating Efficient Code from Strictness Annotations. In *TAPSOFT'87: Proc. Second International Joint Conference on Theory and Practice of Software Development*, pages 140–154, Pisa, Italy, March 1987. Springer Lecture Notes in Computer Science No. 250.

[LGY87b] Gary Lindstrom, Lal George, and Dowming Yeh. Compiling Normal Order to Fair and Incremental Persistence. August 1987. Technical summary; 12 pp.

[Lin73] Gary Lindstrom. Scanning List Structures Without Stacks or Tag Bits. *Information Processing Letters*, 2:47–51, 1973.

[Lin84] G. Lindstrom. OR-Parallelism on Applicative Architectures. In *Proc. 2nd Int'l. Logic Programming Conf.*, pages 159–170, Uppsala University, July 1984.

[Lin85] G. Lindstrom. Functional Programming and the Logical Variable. In *Proc. Symp. on Princ. of Pgmming. Lang.*, pages 266–280, ACM, New Orleans, January 1985. Also available as INRIA Rapport de Recherche No. 357.

[Lin87a] G. Lindstrom. Implementing Logical Variables on a Graph Reduction Architecture. In R. M. Keller and J. Fasel, editors, *Proc. Santa Fe Workshop on Graph Reduction*, Springer-Verlag, 1987. Lecture Notes in Computer Science 279. Summary appears in Proc. ARO Workshop on Future Directions in Computer Architecture and Software, Seabrook Island, SC, May 5-7, 1986.

[Lin87b] G. Lindstrom. Objects in Functional and Logic Programming Languages. In *Proceedings of Object Oriented Programming Workshop*, IBM Watson Research Center, 1987. Paper subsequently invited as chapter for book on object oriented programming, to be edited by Kristen Nygaard.

[Mie84] Colin Mierowsky. *Design and Implementation of Flat Concurrent Prolog.* MS Thesis, Weizmann Institute of Science, 1984.

[Mil78] R. Milner. A Theory of Type Polymorphism. *J. of Comp. and Sys. Sci.*, 17(3):348–375, 1978.

[Nik87] R. S. Nikhil. *Id World Reference Manual (for Lisp Machines).* Technical Report Computation Structures Group Memo, MIT Laboratory for Computer Science, April 24, 1987.

[NPA86] R. S. Nikhil, K. Pingali, and Arvind. *Id Nouveau.* Technical Report Computation Structures Group Memo 265, MIT Laboratory for Computer Science, July 1986.

[PSE85] Dorab Patel, Martine Schlag, and Miloš Ercegovac. $\nu\mathcal{FP}$, an Environment for the Multi-Level Specification, Analysis and Synthesis of Hardware Algorithms. In *Proc. Conf. on Functional Programming Languages and Computer Architectures*, pages 238–255, Springer Verlag, Nancy, France, 1985. Lecture Notes in Computer Science, number 201.

[Red86] U.S. Reddy. On the Relationship Between Functional and Logic Languages. In *Logic Programming: Functions, Relations, and Equations*, Prentice Hall, 1986.

[Sch86] Mark Scheevel. NORMA: A Graph Reduction Processor. In *Proc. Symp. on Lisp and Func. Pgmming. and Computer Architectures*, pages 212–219, ACM, Cambridge, MA, 1986.

[Sha83] E.Y. Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* Technical Report TR-003, Institute for New Generation Computer Technology, January 1983.

[Sha86] Ehud Shapiro. Concurrent Prolog: a Progress Report. *IEEE Computer*, 19(8):44–58, August 1986.

[She85] Mary Sheeran. Designing Regular Array Architectures Using Higher Order Functions. In *Proc. Conf. on Functional Programming Languages and Computer Architectures*, pages 220–237, Springer Verlag, Nancy, France, 1985. Lecture Notes in Computer Science, number 201.

[Str85] Bjarne Stroustrup. A Set of C++ Classes for Co-routine Style Programming. 1985. Appendix to *UNIX (tm) System V C++ Translator Release Notes.*

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986. Paperback, ISBN 0-201-12078-X.

[TL87] P. Tinker and G. Lindstrom. A Performance Oriented Design for OR-Parallel Logic Programming. In Jean-Louis Lassez, editor, *Proc. International Conference on Logic Programming*, pages 601–615, MIT Press, Melbourne, Australia, May 1987.

[TSS86] Stephen Taylor, Shmuel Safra, and Ehud Shapiro. A Parallel Implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming*, 15(3):245–275, June 1986.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, 9:31–49, 1979.

[Tur85] David A. Turner. Miranda: A Non-Strict Functional Language With Polymorphic Types. In J.-P. Jouannaud, editor, *Proc. Symp. on Functional Programming Languages and Computer Architectures*, pages 1–16, Springer-Verlag, 1985.